

EXPERT INSIGHT

Mastering Go

Create Golang production applications using network libraries, concurrency, machine learning, and advanced data structures



Second Edition



Mihalis Tsoukalos

Packt

Mastering Go

Second Edition

Create Golang production applications using network libraries,
concurrency, machine learning, and advanced data structures

Mihalis Tsoukalos

Packt >

BIRMINGHAM - MUMBAI

—

Mastering Go Second Edition

Copyright © 2019 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Acquisition Editor: Andrew Waldron

Acquisition Editor – Peer Reviews: Suresh Jain

Project Editor: Kishor Rit

Development Editor: Joanne Lovell

Technical Editor: Aniket Shetty

Copy Editor: Safis Editing

Proofreader: Safis Editing

Indexer: Pratik Shirodkar

Production Designer: Sandip Tadge

First published: April 2018

Second edition: August 2019

Production reference: 1270819

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-83855-933-5

www.packtpub.com



[Packt.com](https://www.packt.com)

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and, as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Mihalis Tsoukalos is a UNIX administrator, a programmer, a DBA, and a mathematician who enjoys writing technical books and articles and learning new things. He is the author of *Go Systems Programming* and *Mastering Go*. He has written more than 250 technical articles for many magazines, including *Sys Admin*, *MacTech*, *Linux User and Developer*, *Usenix ;login:*, *Linux Format*, and *Linux Journal*. His research interests include databases, visualization, statistics and machine learning.

You can reach him at <https://www.mtsoukalos.eu/> and @mactsouk.

Mihalis is also a photographer.

I would like to thank the people at Packt Publishing for helping me to write this book, including my technical reviewer, Mat Ryer, and Kishor Rit for answering all my questions and encouraging me during the whole process.

I would like to dedicate this book to the loving memory of my parents, Ioannis and Argetta.

About the reviewer

Mat Ryer has been programming computers since he was six years old. He would build games and programs, first in BASIC on a ZX Spectrum, and then in AmigaBASIC and AMOS on Commodore Amiga with his father. Many hours were spent manually copying the code from the Amiga Format magazine and tweaking variables or moving GOTO statements around to see what might happen. The same spirit of exploration and obsession with programming led Mat to starting work with a local agency in Mansfield, England, when he was 18, where he started to build websites and other online services.

After several years of working with various technologies and industries in London and around the world, Mat noticed a new systems language called Go that Google was pioneering. Since it addressed very pertinent and relevant modern technical challenges, he started using it to solve problems while the language was still in the beta stage, and he has been using it ever since. Mat contributes to open-source projects and has founded Go packages, including Testify, Moq, Silk, and Is, as well as a macOS developer tool called BitBar.

In 2018, Mat co-founded Machine Box and still spends a lot of time speaking at conferences, writing about Go on his blog, and being an active member of the Go community.

Table of Contents

Title Page	
Copyright and Credits	
Mastering Go Second Edition	
About Packt	
Why subscribe?	
Contributors	
About the author	
About the reviewer	
Preface	
Who this book is for	
What this book covers	
To get the most out of this book	
Download the example code files	
Download the color images	
Conventions used	
Get in touch	
Reviews	
1. Go and the Operating System	
The history of Go	
Where is Go going?	
The advantages of Go	
Is Go perfect?	
What is a preprocessor?	
The godoc utility	
Compiling Go code	
Executing Go code	
Two Go rules	
You either use a Go package or you do not include it	
There is only one way to format curly braces	
Downloading Go packages	
UNIX stdin, stdout, and stderr	
About printing output	
Using standard output	

Getting user input
About := and =
Reading from standard input
Working with command-line arguments
About error output
Writing to log files
Logging levels
Logging facilities
Log servers
A Go program that sends information to log files
About log.Fatal()
About log.Panic()
Writing to a custom log file
Printing line numbers in log entries
Error handling in Go
The error data type
Error handling
Using Docker
Exercises and links
Summary

2. Understanding Go Internals

The Go compiler
Garbage collection
The tricolor algorithm
More about the operation of the Go garbage collector
Maps, slices, and the Go garbage collector
Using a slice
Using a map with pointers
Using a map without pointers
Splitting the map
Comparing the performance of the presented techniques
Unsafe code
About the unsafe package
Another example of the unsafe package
Calling C code from Go
Calling C code from Go using the same file
Calling C code from Go using separate files

- The C code
- The Go code
- Mixing Go and C code
- Calling Go functions from C code
 - The Go package
 - The C code
- The defer keyword
 - Using defer for logging
- Panic and recover
 - Using the panic function on its own
- Two handy UNIX utilities
 - The strace tool
 - The dtrace tool
- Your Go environment
- The go env command
- The Go assembler
- Node trees
- Finding out more about go build
- Creating WebAssembly code
 - A quick introduction to WebAssembly
 - Why is WebAssembly important?
 - Go and WebAssembly
 - An example
 - Using the generated WebAssembly code
- General Go coding advice
- Exercises and links
- Summary

3. Working with Basic Go Data Types

- Numeric data types
 - Integers
 - Floating-point numbers
 - Complex numbers
 - Number literals in Go 2
- Go loops
 - The for loop
 - The while loop
 - The range keyword

```
An example with multiple Go loops
Go arrays
    Multi-dimensional arrays
    The shortcomings of Go arrays
Go slices
    Performing basic operations on slices
    Slices are expanded automatically
    Byte slices
    The copy() function
    Multi-dimensional slices
    Another example with slices
    Sorting slices using sort.Slice()
    Appending an array to a slice
Go maps
    Storing to a nil map
    When you should use a map
Go constants
    The constant generator iota
Go pointers
    Why use pointers?
Times and dates
    Working with times
    Parsing times
    Working with dates
    Parsing dates
    Changing date and time formats
Measuring execution time
    Measuring the operation of the Go garbage collector
Web links and exercises
Summary
```

4. The Uses of Composite Types

```
About composite types
Structures
    Pointers to structures
    Using the new keyword
Tuples
Regular expressions and pattern matching
```

- Introducing some theory
- A simple example
- A more advanced example
- Matching IPv4 addresses
- Strings
 - What is a rune?
 - The unicode package
 - The strings package
- The switch statement
- Calculating Pi with high accuracy
- Developing a key-value store in Go
- Go and the JSON format
 - Reading JSON data
 - Saving JSON data
 - Using Marshal() and Unmarshal()
 - Parsing JSON data
- Go and XML
 - Reading an XML file
 - Customizing XML output
- Go and the YAML format
- Additional resources
- Exercises and web links
- Summary

5. How to Enhance Go Code with Data Structures

- About graphs and nodes
- Algorithm complexity
- Binary trees in Go
 - Implementing a binary tree in Go
 - Advantages of binary trees
- Hash tables in Go
 - Implementing a hash table in Go
 - Implementing the lookup functionality
 - Advantages of hash tables
- Linked lists in Go
 - Implementing a linked list in Go
 - Advantages of linked lists
- Doubly linked lists in Go

Go modules

 Creating and using a Go module

 Creating version v1.0.0

 Using version v1.0.0

 Creating version v1.1.0

 Using version v1.1.0

 Creating version v2.0.0

 Using version v2.0.0

 Creating version v2.1.0

 Using version v2.1.0

 Using two different versions of the same Go module

 Where Go stores Go modules

 The go mod vendor command

Creating good Go packages

The syscall package

 Finding out how fmt.Println() really works

The go/scanner, go/parser, and go/token packages

 The go/ast package

 The go/scanner package

 The go/parser package

 A practical example

 Finding variable names with a given string length

Text and HTML templates

 Generating text output

 Constructing HTML output

Additional resources

Exercises

Summary

7. Reflection and Interfaces for All Seasons

Type methods

Go interfaces

 About type assertions

Writing your own interfaces

 Using a Go interface

 Using switch with interface and data types

Reflection

 A simple reflection example

- A more advanced reflection example
- The three disadvantages of reflection
- The reflectwalk library

Object-oriented programming in Go

An introduction to git and GitHub

- Using git
 - The git status command
 - The git pull command
 - The git commit command
 - The git push command
 - Working with branches
 - Working with files
 - The .gitignore file
 - Using git diff
 - Working with tags
 - The git cherry-pick command

Debugging with Delve

- A debugging example

Additional resources

Exercises

Summary

8. Telling a UNIX System What to Do

- About UNIX processes
- The flag package
- The viper package
 - A simple viper example
 - From flag to viper
 - Reading JSON configuration files
 - Reading YAML configuration files
- The cobra package
 - A simple cobra example
 - Creating command aliases
- The io.Reader and io.Writer Interfaces
 - Buffered and unbuffered file input and output
- The bufio package
- Reading text files
 - Reading a text file line by line

- Reading a text file word by word
- Reading a text file character by character
- Reading from /dev/random
- Reading a specific amount of data
- The advantages of binary formats
- Reading CSV files
- Writing to a file
- Loading and saving data on disk
- The strings package revisited
- About the bytes package
- File permissions
- Handling UNIX signals
 - Handling two signals
 - Handling all signals
- Programming UNIX pipes in Go
 - Implementing the cat(1) utility in Go
- About syscall.PtraceRegs
- Tracing system calls
- User ID and group ID
- The Docker API and Go
- Additional resources
- Exercises
- Summary

9. Concurrency in Go - Goroutines, Channels, and Pipelines

- About processes, threads, and goroutines
 - The Go scheduler
 - Concurrency and parallelism
- Goroutines
 - Creating a goroutine
 - Creating multiple goroutines
- Waiting for your goroutines to finish
 - What if the number of Add() and Done() calls do not agree?
- Channels
 - Writing to a channel
 - Reading from a channel
 - Receiving from a closed channel
 - Channels as function parameters

- Pipelines
- Race conditions
- Comparing Go and Rust concurrency models
- Comparing Go and Erlang concurrency models
- Additional resources
- Exercises
- Summary

10. Concurrency in Go - Advanced Topics

- The Go scheduler revisited
 - The GOMAXPROCS environment variable
- The select keyword
- Timing out a goroutine
 - Timing out a goroutine - take 1
 - Timing out a goroutine - take 2
- Go channels revisited
 - Signal channels
 - Buffered channels
 - Nil channels
 - Channels of channels
 - Specifying the order of execution for your goroutines
 - How not to use goroutines
- Shared memory and shared variables
 - The sync.Mutex type
 - What happens if you forget to unlock a mutex?
 - The sync.RWMutex type
 - The atomic package
 - Sharing memory using goroutines
- Revisiting the go statement
- Catching race conditions
- The context package
 - An advanced example of the context package
 - Another example of the context package
 - Worker pools
- Additional resources
- Exercises
- Summary

11. Code Testing, Optimization, and Profiling

About optimization
Optimizing Go code
Profiling Go code
 The net/http/pprof standard Go package
 A simple profiling example
 A convenient external package for profiling
 The web interface of the Go profiler
 A profiling example that uses the web interface
 A quick introduction to Graphviz
The go tool trace utility
Testing Go code
 Writing tests for existing Go code
 Test code coverage
Testing an HTTP server with a database backend
 The testing/quick package
 What if testing takes too long or never finishes?
Benchmarking Go code
A simple benchmarking example
 Wrongly defined benchmark functions
Benchmarking buffered writing
Finding unreachable Go code
Cross-compilation
Creating example functions
From Go code to machine code
 Using assembly with Go
Generating documentation
Using Docker images
Additional resources
Exercises
Summary

12. The Foundations of Network Programming in Go

About net/http, net, and http.RoundTripper
 The http.Response type
 The http.Request type
 The http.Transport type
About TCP/IP
About IPv4 and IPv6

The nc(1) command-line utility
Reading the configuration of network interfaces
Performing DNS lookups
 Getting the NS records of a domain
 Getting the MX records of a domain
Creating a web server in Go
 Using the atomic package
 Profiling an HTTP server
 Creating a website in Go
HTTP tracing
 Testing HTTP handlers
Creating a web client in Go
 Making your Go web client more advanced
Timing out HTTP connections
 More information about SetDeadline
 Setting the timeout period on the server side
 Yet another way to time out
The Wireshark and tshark tools
gRPC and Go
 Defining the interface definition file
 The gRPC client
 The gRPC server
Additional resources
Exercises
Summary

13. Network Programming - Building Your Own Servers and Clients

Working with HTTPS traffic
 Creating certificates
 An HTTPS client
 A simple HTTPS server
 Developing a TLS server and client
The net standard Go package
A TCP client
 A slightly different version of the TCP client
A TCP server
 A slightly different version of the TCP server
A UDP client

Developing a UDP server
A concurrent TCP server
 A handy concurrent TCP server
Creating a Docker image for a Go TCP/IP server
Remote Procedure Call (RPC)
 The RPC client
 The RPC server
Doing low-level network programming
 Grabbing raw ICMP network data
Additional resources
Exercises
Summary

14. Machine Learning in Go

Calculating simple statistical properties
Regression
 Linear regression
 Implementing linear regression
 Plotting data
Classification
Clustering
Anomaly detection
Neural networks
Outlier analysis
Working with TensorFlow
Talking to Kafka
Additional resources
Exercises
Summary
Where to go next?

Other Books You May Enjoy

Leave a review - let other readers know what you think

Preface

The book you are reading right now is *Mastering Go, Second Edition*, and is all about helping you become a better Go developer!

There are many exciting new topics, including an entirely new chapter that talks about Machine Learning in Go as well as information and code examples relating to the Viper and Cobra Go packages, gRPC, working with Docker images, working with YAML files, working with the `go/scanner` and `go/token` packages, and generating WebAssembly code from Go. In total, there are more than 130 new pages in this second edition of Mastering Go.

Who this book is for

This book is for amateur and intermediate Go programmers who want to take their Go knowledge to the next level, as well as experienced developers in other programming languages who want to learn Go without learning again how a `for` loop works.

Some of the information found in this book can be also found in my other book, *Go Systems Programming*. The main difference between these two books is that *Go Systems Programming* is about developing system tools using the capabilities of Go, whereas *Mastering Go* is about explaining the capabilities and the internals of Go in order to become a better Go developer. Both books can be used as a reference after reading them for the first or the second time.

What this book covers

[Chapter 1](#), *Go and the Operating System*, begins by talking about the history of Go and the advantages of Go before describing the `godoc` utility and explaining how you can compile and execute Go programs. After that, it talks about printing output and getting user input, working with the command-line arguments of a program and using log files. The final topic in the first chapter is error handling, which plays a key role in Go.

[Chapter 2](#), *Understanding Go Internals*, discusses the Go garbage collector and the way it operates. Then, it talks about unsafe code and the `unsafe` package, how to call C code from a Go program, and how to call Go code from a C program.

After that, it showcases the use of the `defer` keyword and presents the `strace(1)` and `dtrace(1)` utilities. In the remaining sections of the chapter, you will learn how to find information about your Go environment, the use of the Go assembler, and how to generate WebAssembly code from Go.

[Chapter 3](#), *Working with Basic Go Data Types*, talks about the data types offered by Go, which includes arrays, slices, and maps, as well as Go pointers, constants, loops, and working with dates and times. You would not want to miss this chapter!

[Chapter 4](#), *The Uses of Composite Types*, begins by teaching you about Go structures and the `struct` keyword before discussing tuples, strings, runes, byte slices, and string literals. The remainder of the chapter talks about regular expressions and pattern matching, the `switch` statement, the `strings` package, the `math/big` package, about developing a key-value store in Go, and about working with XML and JSON files.

[Chapter 5](#), *How to Enhance Go Code with Data Structures*, is about developing your own data structures when the structures offered by Go do not fit a particular problem. This includes developing binary trees, linked

lists, hash tables, stacks, and queues, and learning about their advantages. This chapter also showcases the use of the structures found in the `container` standard Go package, as well as how to use Go to verify Sudoku puzzles and generate random numbers.

[Chapter 6](#), *What You Might Not Know About Go Packages and Functions*, is all about packages and functions, which also includes the use of the `init()` function, the `syscall` standard Go package, and the `text/template` and `html/template` packages. Additionally, it shows the use of the `go/scanner`, `go/parser`, and `go/token` advanced packages. This chapter will definitely make you a better Go developer!

[Chapter 7](#), *Reflection and Interfaces for All Seasons*, discusses three advanced Go concepts: reflection, interfaces, and type methods. Additionally, it discusses the object-oriented capabilities of Go and how to debug Go programs using Delve!

[Chapter 8](#), *Telling a UNIX System What to Do*, is about Systems Programming in Go, which includes subjects such as the `flag` package for working with command-line arguments, handling UNIX signals, file input and output, the `bytes` package, the `io.Reader` and `io.Writer` interfaces, and the use of the Viper and Cobra Go packages. As I told you before, if you are really into systems programming in Go, then getting *Go Systems Programming* after reading *Mastering Go, Second Edition*, is highly recommended!

[Chapter 9](#), *Concurrency in Go – Goroutines, Channels, and Pipelines*, discusses goroutines, channels, and pipelines, which is the Go way of achieving concurrency.

You will also learn about the differences between processes, threads, and goroutines, the `sync` package, and the way the Go scheduler operates.

[Chapter 10](#), *Concurrency in Go – Advanced Topics*, will continue from the point where the previous chapter left off and make you a master of goroutines and channels! You will learn more about the Go scheduler, the use of the powerful `select` keyword and the various types of Go channels, as

well as shared memory, mutexes, the `sync.Mutex` type, and the `sync.RWMutex` type. The final part of the chapter talks about the `context` package, worker pools, and how to detect race conditions.

[Chapter 11](#), *Code Testing, Optimization, and Profiling*, discusses code testing, code optimization and code profiling, as well as cross-compilation, creating documentation, benchmarking Go code, creating example functions, and finding unreachable Go code.

[Chapter 12](#), *The Foundations of Network Programming in Go*, is all about the `net/http` package and how you can develop web clients and web servers in Go. This also includes the use of the `http.Response`, `http.Request`, and `http.Transport` structures, and the `http.NewServeMux` type. You will even learn how to develop an entire web site in Go! Furthermore, in this chapter, you will learn how to read the configuration of your network interfaces and how to perform DNS lookups in Go. Additionally, you will learn how to use gRPC with Go.

[Chapter 13](#), *Network Programming – Building Your Own Servers and Clients*, talks about working with HTTPS traffic, and creating UDP and TCP servers and clients in Go using the functionality offered by the `net` package. Other topics included in this chapter are creating RPC clients and servers as well as developing a concurrent TCP server in Go and reading raw network packages!

[Chapter 14](#), *Machine Learning in Go*, talks about machine learning in Go, including classification, clustering, anomaly detection, outliers, neural networks and TensorFlow, as well as working with Apache Kafka with Go.

This book can be divided into three logical parts. The first part takes a sophisticated look at some important Go concepts, including user input and output, downloading external Go packages, compiling Go code, calling C code from Go, and creating WebAssembly from Go, as well as using Go basic types and Go composite types.

The second part starts with [Chapter 5](#), *How to Enhance Go Code with Data Structures*, and also includes [Chapter 6](#), *What You Might Not Know About Go*.

Packages and Go Functions, and [Chapter 7, Reflection and Interfaces for All Seasons](#). These three chapters deal with Go code organization in packages and modules, the design of Go projects, and some advanced features of Go, respectively.

The last part includes the remaining seven chapters and deals with more practical Go topics. Chapters 8, 9, 10, and 11 talk about systems programming in Go, concurrency in Go, code testing, optimization, and profiling. The last three chapters of this book will talk about network programming and machine learning in Go.

The book includes content such as Go and WebAssembly, using Docker with Go, creating professional command-line tools with the Viper and Cobra packages, parsing JSON and YAML records, performing operations with matrices, working with Sudoku puzzles, the `go/scanner` and `go/token` packages, working with `git(1)` and GitHub, the `atomic` package, gRPC and Go, and HTTPS.

The book will present relatively small yet complete Go programs that illustrate the presented concepts. This has two main advantages: firstly, you do not have to look at an endless code listing when trying to learn a single technique and secondly, you can use this code as a starting point when creating your own applications and utilities.



*Realizing the importance of **containers** and **Docker**, this book includes various examples of Go executable files that are used from within Docker images because Docker images offer a great way to deploy server software.*

To get the most out of this book

This book requires a UNIX computer with a relatively recent Go version installed, which includes any machine running Mac OS X, macOS, or Linux. Most of the code presented will also run on Microsoft Windows machines.

To get the most out of this book, you should try to apply the knowledge of each chapter in your own programs as soon as possible and see what works and what does not! As I told you before, try to solve the exercises found at the end of each chapter or create your own programming problems.

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packt.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Mastering-Go-Second-Edition>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: <https://static.packt-cdn.com/downloads/9781838559335.pdf>.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The first way is similar to using the `man(1)` command, but for Go functions and packages."

A block of code is set as follows:

```
package main
import (
    "fmt"
)
func main() {
    fmt.Println("This is a sample Go program!")
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
import (
    "fmt"
)
func main() {
    fmt.Println("This is a sample Go program!")
}
```

Any command-line input or output is written as follows:

```
$ date
Sat Oct 21 20:09:20 EEST 2017
$ go version
go version go1.12.7 darwin/amd64
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at

customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packt.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

Go and the Operating System

This chapter is an introduction to various Go topics that beginners will find very useful. More experienced Go developers can also use this chapter as a refresher course on the fundamentals of Go. As it happens with most practical subjects, the best way to understand something is to experiment with it. In this case, experimenting means writing Go code on your own, making your own mistakes, and learning from them! Just don't let error messages and bugs discourage you.

In this chapter, you will learn about:

- The history and the future of the Go programming language
- The advantages of Go
- Compiling Go code
- Executing Go code
- Downloading and using external Go packages
- UNIX standard input, output, and error
- Printing data on the screen
- Getting user input
- Printing data to standard error
- Working with log files
- Using Docker to compile and execute a Go source file
- Error handling in Go

The history of Go

Go is a modern, generic-purpose, open source programming language that was officially announced at the end of 2009. Go began as an internal Google project, which means that it was started as an experiment, and has since been inspired by many other programming languages, including **C**, **Pascal**, **Alef**, and **Oberon**. Go's spiritual fathers are the professional programmers *Robert Griesemer*, *Ken Thomson*, and *Rob Pike*.

They designed Go as a language for professional programmers who want to build reliable, robust, and efficient software. Apart from its syntax and its standard functions, Go comes with a pretty rich standard library.

At the time of writing, the latest stable Go version is version 1.13. However, even if your version number is higher, the contents of the book will still be relevant.

If you are going to install Go for the first time, you can start by visiting <http://golang.org/dl/>. However, there is a big chance that your UNIX variant has a ready-to-install package for the Go programming language, so you might want to get Go by using your favorite package manager.

Where is Go going?

The Go community is already discussing the next major version of Go, which is going to be called Go 2, but there is nothing definitive at the moment.

The intention of the current Go 1 team is to make Go 2 more community driven. Although this is a good idea in general, it is always dangerous when lots of people try to make important decisions about a programming language that was initially designed and developed as an internal project by a small group of great people.

Some of the big changes that are being considered for Go 2 are generics, package versioning, and improved error handling. All these new features are under discussion at the moment and you should not be worried about them, but it is worthwhile to have an idea of the direction that Go is going in.

The advantages of Go

Go has many advantages, and some of them are unique to Go, while others are shared with other programming languages.

The list of the most significant Go advantages and features includes the following:

- Go is a modern programming language that is easy to read, easy to understand, and was made by experienced developers.
- Go wants happy developers because happy developers write better code!
- The Go compiler prints practical warning and error messages that help you to solve the actual problem. Putting it simply, the Go compiler is there to help you, not to make your life miserable by printing pointless output!
- Go code is portable, especially among UNIX machines.
- Go has support for procedural, concurrent, and distributed programming.
- Go supports **garbage collection**, so you do not have to deal with memory allocation and deallocation.
- Go does not have a **preprocessor** and does high-speed compilation. As a consequence, Go can also be used as a scripting language.
- Go can build web applications and provides a simple web server for testing purposes.
- The standard Go library offers many packages that simplify the work of the developer. Additionally, the functions found in the standard Go library are tested and debugged in advance by the people who develop Go, which means that, most of the time, they come without bugs.
- Go uses **static linking** by default, which means that the binary files produced can be easily transferred to other machines with the same OS. As a consequence, once a Go program is compiled successfully and an executable file is generated, you do not need to worry about libraries, dependencies, and different library versions anymore.

- You will not need a **graphical user interface (GUI)** for developing, debugging, and testing Go applications, as Go can be used from the command-line, which I think many UNIX people prefer.
- Go supports **Unicode**, which means that you do not need any extra code for printing characters from multiple human languages.
- Go keeps concepts orthogonal because a few orthogonal features work better than many overlapping ones.

Is Go perfect?

There is no such thing as the perfect programming language, and Go is not an exception to this rule. However, some programming languages are better at some areas of programming or we like them more than other programming languages. Personally, I do not like Java, and while I used to like C++, I do not like it anymore. C++ has become too complex as a programming language, whereas, in my opinion, Java, code does not look good.

Some of the disadvantages of Go are:

- Go does not have direct support for **object-oriented programming**, which can be a problem for programmers who are used to writing code in an object-oriented manner. Nevertheless, you can use composition in Go to mimic inheritance.
- For some people, Go will never replace C.
- C is still faster than any other programming language for systems programming and this is mainly because UNIX is written in C.

Nevertheless, Go is a pretty decent programming language that will not disappoint you if you find the time to learn it and program in it.

What is a preprocessor?

I said earlier that Go does not have a **preprocessor** and that this is a good thing. A preprocessor is a program that processes your input data and generates output that will be used as the input to another program. In the context of programming languages, the input of a preprocessor is source code that will be processed by the preprocessor before being given as input to the compiler of the programming language.

The biggest disadvantage of a preprocessor is that it knows nothing about the underlying language or its syntax! This means that when a preprocessor is used, you cannot be certain that the final version of your code will do what you really want because the preprocessor might alter the logic as well as the semantics of your original code.

The list of programming languages with a preprocessor includes C, C++, Ada, and PL/SQL. The infamous C preprocessor processes lines that begin with `#` and are called **directives** or **pragmas**. As stated before, directives and pragmas are not part of the C programming language!

The godoc utility

The Go distribution comes with a plethora of tools that can make your life as a programmer easier. One of these tools is the `godoc` utility, which allows you to see the documentation of existing Go functions and packages without needing an internet connection.

The `godoc` utility can be executed either as a normal command-line application that displays its output on a terminal, or as a command-line application that starts a web server. In the latter case, you will need a web browser to look at the Go documentation.



If you type `godoc` without any command-line parameters, you will get a list of the command-line options supported by `godoc`.

The first way is similar to using the `man(1)` command, but for Go functions and packages. So, in order to find information about the `printf()` function of the `fmt` package, you should execute the following command:

```
| $ go doc fmt.Println
```

Similarly, you can find information about the entire `fmt` package by running the following command:

```
| $ go doc fmt
```

The second way requires executing `godoc` with the `-http` parameter:

```
| $ godoc -http=:8001
```

The numeric value in the preceding command, which in this case is `8001`, is the port number the HTTP server will listen to. You can choose any port number that is available provided that you have the right privileges. However, note that port numbers 0-1023 are restricted and can only be used

by the root user, so it is better to avoid choosing one of those and pick something else, provided that it is not already in use by a different process.

You can omit the equal sign in the presented command and put a space character in its place. So, the following command is completely equivalent to the previous one:

```
| $ godoc -http :8001
```

After that, you should point your web browser to the `http://localhost:8001/pkg/` URL in order to get the list of available Go packages and browse their documentation.

Compiling Go code

In this section, you will learn how to compile Go code. The good news is that you can compile your Go code from the command line without the need for a graphical application. Furthermore, Go does not care about the name of the source file of an autonomous program as long as the package name is `main` and there is a single `main()` function in it. This is because the `main()` function is where the program execution begins. As a result, you cannot have multiple `main()` functions in the files of a single project.

We will start our first Go program compilation with a program named `aSourceFile.go` that contains the following Go code:

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("This is a sample Go program!")
}
```

Notice that the Go community prefers to name the Go source file `source_file.go` instead of `aSourceFile.go`. Whatever you choose, be consistent.

In order to compile `aSourceFile.go` and create a **statically linked** executable file, you will need to execute the following command:

```
$ go build aSourceFile.go
```

After that, you will have a new executable file named `aSourceFile` that you will need to execute:

```
$ file aSourceFile
aSourceFile: Mach-O 64-bit executable x86_64
$ ls -l aSourceFile
-rwxr-xr-x 1 mtsouk  staff  2007576 Jan 10 21:10 aSourceFile
$ ./aSourceFile
This is a sample Go program!
```

The main reason why the file size of `aSourceFile` is that big is because it is statically linked, which means that it does not require any external libraries to run.

Executing Go code

There is another way to execute your Go code that does not create any permanent executable files – it just generates some intermediate files that are automatically deleted afterward.



The way presented allows you to use Go as if it is a scripting programming language like Python, Ruby, or Perl.

So, in order to run `aSourceFile.go` without creating an executable file, you will need to execute the following command:

```
$ go run aSourceFile.go  
This is a sample Go program!
```

As you can see, the output of the preceding command is exactly the same as before.



Please note that with `go run`, the Go compiler still needs to create an executable file. It is because you do not see it, it is automatically executed, and it is automatically deleted after the program has finished that you might think that there is no need for an executable file.

This book mainly uses `go run` to execute the example code; primarily because it is simpler than running `go build` and then executing the executable file. Additionally, `go run` does not leave any files on your hard disk after the program has finished its execution.

Two Go rules

Go has strict coding rules that are there to help you avoid silly errors and bugs in your code, as well as to make your code easier to read for the Go community. This section will present two such Go rules that you need to know.

As mentioned earlier, please remember that the Go compiler is there to help and not make your life miserable. As a result, the main purpose of the Go compiler is to compile and increase the quality of your Go code.

You either use a Go package or you do not include it

Go has strict rules about package usage. Therefore, you cannot just include any package you might think that you will need and then not use it afterward.

Look at the following naive program, which is saved as `packageNotUsed.go`:

```
package main

import (
    "fmt"
    "os"
)

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



In this book, you are going to see lots of error messages, error situations, and warnings. I believe that looking at code that fails to compile is also useful and sometimes even more valuable than just looking at Go code that compiles without any errors. The Go compiler usually displays useful error messages and warnings that will most likely help you to resolve an erroneous situation, so do not underestimate Go error messages and warnings.

If you execute `packageNotUsed.go`, you will get the following error message from Go and the program will not get executed:

```
$ go run packageNotUsed.go
# command-line-arguments
./packageNotUsed.go:5:2: imported and not used: "os"
```

If you remove the `os` package from the `import` list of the program, `packageNotUsed.go` will compile just fine; try it on your own.

Although this is not the perfect time to start talking about breaking Go rules, there is a way to bypass this restriction. This is showcased in the following Go code that is saved in the `packageNotUsedUnderscore.go` file:

```
package main

import (
    "fmt"
    _ "os"
)

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

So, using an underscore character in front of a package name in the `import` list will not create an error message in the compilation process even if that package will not be used in the program:

```
$ go run packageNotUsedUnderscore.go
Hello there!
```



The reason that Go is allowing you to bypass this rule will become more evident in [Chapter 6](#), What You Might Not Know About Go Packages and Go Functions.

There is only one way to format curly braces

Look at the following Go program named `curly.go`:

```
package main

import (
    "fmt"
)

func main()
{
    fmt.Println("Go has strict rules for curly braces!")
}
```

Although it looks just fine, if you try to execute it, you will be fairly disappointed, because you will get the following **syntax error** message and the code will not compile and therefore run:

```
$ go run curly.go
# command-line-arguments
./curly.go:7:6: missing function body for "main"
./curly.go:8:1: syntax error: unexpected semicolon or newline before {
```

The official explanation for this error message is that Go requires the use of semicolons as statement terminators in many contexts, and the compiler automatically inserts the required semicolons when it thinks that they are necessary. Therefore, putting the opening curly brace (`{`) in its own line will make the Go compiler insert a semicolon at the end of the previous line (`func main()`), which is the cause of the error message.

Downloading Go packages

Although the standard Go library is very rich, there are times that you will need to download external Go packages in order to use their functionality. This section will teach you how to download an external Go package and where it will be placed on your UNIX machine.



*Have in mind that although Go **modules**, which is a new Go feature that is still under development, might introduce changes to the way you work with external Go code, the process of downloading a single Go package into your computer will remain the same.*



You will learn a lot more about Go packages and Go modules in [Chapter 6](#), What You Might Not Know About Go Packages and Go Functions.

Look at the following naive Go program that is saved as `getPackage.go`:

```
package main

import (
    "fmt"
    "github.com/mactsouk/go/simpleGitHub"
)

func main() {
    fmt.Println(simpleGitHub.AddTwo(5, 6))
}
```

This program uses an external package because one of the `import` commands uses an internet address. In this case, the external package is called `simpleGitHub` and is located at `github.com/mactsouk/go/simpleGitHub`.

If you try to execute `getPackage.go` right away, you will be disappointed:

```
$ go run getPackage.go
getPackage.go:5:2: cannot find package "github.com/mactsouk/go/simpleGitHub" in any of:
/usr/local/Cellar/go/1.9.1/libexec/src/github.com/mactsouk/go/
simpleGitHub (from $GOROOT)
/Users/mtsouk/go/src/github.com/mactsouk/go/simpleGitHub (from $GOPATH)
```

So, you will need to get the missing package on your computer. In order to download it, you will need to execute the following command:

```
$ go get -v github.com/mactsouk/go/simpleGitHub
github.com/mactsouk/go (download)
github.com/mactsouk/go/simpleGitHub
```

After that, you can find the downloaded files at the following directory:

```
$ ls -l ~/go/src/github.com/mactsouk/go/simpleGitHub/
total 8
-rw-r--r-- 1 mtsouk  staff  66 Oct 17 21:47 simpleGitHub.go
```

However, the `go get` command also compiles the package. The relevant files can be found at the following place:

```
$ ls -l ~/go/pkg/darwin_amd64/github.com/mactsouk/go/simpleGitHub.a
-rw-r--r-- 1 mtsouk  staff  1050 Oct 17 21:47 /Users/mtsouk/go/pkg/darwin_amd64/github.com/mactsouk/go/simpleGitHub.a
```

You are now ready to execute `getPackage.go` without any problems:

```
$ go run getPackage.go
11
```

|
You can delete the intermediate files of a downloaded Go package as follows:

```
$ go clean -i -v -x github.com/mactsouk/go/simpleGitHub
cd /Users/mtsouk/go/src/github.com/mactsouk/go/simpleGitHub
rm -f simpleGitHub.test simpleGitHub.test.exe
rm -f /Users/mtsouk/go/pkg/darwin_amd64/github.com/mactsouk/go/
simpleGitHub.a
```

Similarly, you can delete an entire Go package you have downloaded locally using the `rm(1)` UNIX command to delete its Go source after using `go clean`:

```
$ go clean -i -v -x github.com/mactsouk/go/simpleGitHub
$ rm -rf ~/go/src/github.com/mactsouk/go/simpleGitHub
```

After executing the former commands, you will need to download the Go package again.

UNIX stdin, stdout, and stderr

Every UNIX OS has three files open all the time for its processes. Remember that UNIX considers everything, even a printer or your mouse, a file.

UNIX uses **file descriptors**, which are positive integer values, as an internal representation for accessing all of its open files, which is much prettier than using long paths.

So, by default, all UNIX systems support three special and standard filenames: `/dev/stdin`, `/dev/stdout`, and `/dev/stderr`, which can also be accessed using file descriptors 0, 1, and 2, respectively. These three file descriptors are also called **standard input**, **standard output**, and **standard error**, respectively. Additionally, file descriptor 0 can be accessed as `/dev/fd/0` on a macOS machine and as both `/dev/fd/0` and `/dev/pts/0` on a Debian Linux machine.

Go uses `os.Stdin` for accessing standard input, `os.Stdout` for accessing standard output, and `os.Stderr` for accessing standard error. Although you can still use `/dev/stdin`, `/dev/stdout`, and `/dev/stderr` or the related file descriptor values for accessing the same devices, it is better, safer, and more portable to stick with `os.Stdin`, `os.Stdout`, and `os.Stderr` offered by Go.

About printing output

As with UNIX and C, Go offers a variety of ways for printing your output on the screen. All the printing functions of this section require the use of the `fmt` Go standard package and are illustrated in the `printing.go` program, which will be presented in two parts.

The simplest way to print something in Go is by using the `fmt.Println()` and the `fmt.Printf()` functions. The `fmt.Printf()` function has many similarities with the C `printf(3)` function. You can also use the `fmt.Print()` function instead of `fmt.Println()`. The main difference between `fmt.Print()` and `fmt.Println()` is that the latter automatically adds a newline character each time you call it.

On the other hand, the biggest difference between `fmt.Println()` and `fmt.Printf()` is that the latter requires a **format specifier** for each *thing* that you want to print, just like the C `printf(3)` function, which means that you have better control of what you are doing, but you have to write more code. Go calls these format specifiers **verbs**. You can find more information about verbs at <https://golang.org/pkg/fmt/>.

If you have to perform any formatting before printing something or have to arrange multiple variables, then using `fmt.Printf()` might be a better choice. However, if you only have to print a single variable, then you might need to choose either `fmt.Print()` or `fmt.Println()`, depending on whether you need a newline character or not.

The first part of `printing.go` contains the following Go code:

```
package main

import (
    "fmt"
)

func main() {
    v1 := "123"
    v2 := 123
```

```
|     v3 := "Have a nice day\n"
|     v4 := "abc"
```

In this part, you see the `import` of the `fmt` package and the definition of four Go variables. The `\n` used in `v3` is the line break character. However, if you just want to insert a line break in your output, you can call `fmt.Println()` without any arguments, instead of using something like `fmt.Print("\n")`.

The second part is as follows:

```
|     fmt.Println(v1, v2, v3, v4)
|     fmt.Println()
|     fmt.Println(v1, v2, v3, v4)
|     fmt.Print(v1, " ", v2, " ", v3, " ", v4, "\n")
|     fmt.Printf("%s%d %s %s\n", v1, v2, v3, v4)
| }
```

In this part, you print the four variables using `fmt.Println()`, `fmt.Print()`, and `fmt.Sprintf()` in order to better understand how they differ.

If you execute `printing.go`, you will get the following output:

```
$ go run printing.go
123123Have a nice day
abc
123 123 Have a nice day
abc
123 123 Have a nice day
abc
123123 Have a nice day
abc
```

As you can see from the preceding output, the `fmt.Println()` function also adds a space character between its parameters, which is not the case with `fmt.Print()`.

As a result, a statement such as `fmt.Println(v1, v2)` is equivalent to `fmt.Print(v1, " ", v2, "\n")`.

Apart from `fmt.Println()`, `fmt.Print()`, and `fmt.Sprintf()`, which are the simplest functions that can be used for generating output on the screen, there is also the `S` family of functions that includes `fmt.Sprintln()`, `fmt.Sprint()`, and `fmt.Sprintf()`. These functions are used to create strings based on a given format.

Finally, there is the F family of functions, which includes `fmt.Fprintln()`, `fmt.Fprint()`, and `fmt.Fprintf()`. They are used for writing to files using an `io.Writer`.



You will learn more about the `io.Writer` and `io.Reader` interfaces in [Chapter 8](#), Telling a UNIX System What to Do.

The next section will teach you how to print your data using standard output, which is pretty common in the UNIX world.

Using standard output

Standard output is more or less equivalent to printing on the screen. However, using standard output might require the use of functions that do not belong to the `fmt` package, which is why it is presented in its own section.

The relevant technique will be illustrated in `stdOUT.go` and will be offered in three parts. The first part of the program is as follows:

```
package main

import (
    "io"
    "os"
)
```

So, `stdOUT.go` will use the `io` package instead of the `fmt` package. The `os` package is used for reading the command-line arguments of the program and for accessing `os.Stdout`.

The second portion of `stdOUT.go` contains the following Go code:

```
func main() {
    myString := ""
    arguments := os.Args
    if len(arguments) == 1 {
        myString = "Please give me one argument!"
    } else {
        myString = arguments[1]
    }
}
```

The `myString` variable holds the text that will be printed on the screen, which is either the first command-line argument of the program or, if the program was executed without any command-line arguments, a hardcoded text message.

The third part of the program is as follows:

```
    io.WriteString(os.Stdout, myString)
    io.WriteString(os.Stdout, "\n")
}
```

In this case, the `io.WriteString()` function works in the same way as the `fmt.Println()` function; however, it takes only two parameters. The first parameter is the file you want to write to, which, in this case, is `os.Stdout`, and the second parameter is a `string` variable.



*Strictly speaking, the type of the first parameter of the `io.WriteString()` function should be `io.Writer`, which requires a *slice* of bytes as the second parameter. However, in this case, a `string` variable does the job just fine. You will learn more about slices in [Chapter 3](#), Working with Basic Go Data Types.*

Executing `stdOUT.go` will produce the following output:

```
$ go run stdOUT.go  
Please give me one argument!  
$ go run stdOUT.go 123 12  
123
```

The preceding output verifies that the `io.WriteString()` function sends the contents of its second parameter onto the screen when its first parameter is `os.Stdout`.

Getting user input

There are three main ways to get user input: firstly, by reading the command-line arguments of a program; secondly, by asking the user for input; or thirdly, by reading external files. This section will present the first two ways. Should you wish to learn how to read an external file, you should visit [Chapter 8](#), *Telling a UNIX System What to Do*.

About := and =

Before continuing, it will be very useful to talk about the use of `:=` and how it differs from `=`. The official name for `:=` is the **short assignment statement**. The short assignment statement can be used in place of a `var` declaration with an implicit type.



You will rarely see the use of `var` in Go; the `var` keyword is mostly used for declaring global variables in Go programs, as well as for declaring variables without an initial value. The reason for the former is that every statement that exists outside of the code of a function must begin with a keyword such as `func` or `var`. This means that the short assignment statement cannot be used outside of a function because it is not available there.

The `:=` operator works as follows:

```
| m := 123
```

The result of the preceding statement is a new integer variable named `m` with a value of `123`.

However, if you try to use `:=` on an already declared variable, the compilation will fail with the following error message, which makes perfect sense:

```
$ go run test.go
# command-line-arguments
./test.go:5:4: no new variables on left side of :=
```

So, you might now ask, what will happen if you are expecting two or more values from a function and you want to use an existing variable for one of them. Should you use `:=` or `=`? The answer is simple: you should use `:=`, as in the following code example:

```
| i, k := 3, 4
| j, k := 1, 2
```

As the `j` variable is used for the first time in the second statement, you use `:=` even though `k` has already been defined in the first statement.

Although it may seem boring to talk about such insignificant things, knowing them will save you from various types of errors in the long run!

Reading from standard input

The reading of data from the standard input will be illustrated in `stdin.go`, which you will see in two parts. The first part is as follows:

```
package main

import (
    "bufio"
    "fmt"
    "os"
)
```

In the preceding code, you can see the use of the `bufio` package for the first time in this book.



You will learn more about the `bufio` package in [Chapter 8](#), Telling a UNIX System What to Do.

Although the `bufio` package is mostly used for file input and output, you will keep seeing the `os` package all the time in this book because it contains many handy functions; its most common functionality is that it provides a way to access the command-line arguments of a Go program (`os.Args`).

The official description of the `os` package tells us that it offers functions that perform OS operations. This includes functions for creating, deleting, and renaming files and directories, as well as functions for learning the UNIX permissions and other characteristics of files and directories. The main advantage of the `os` package is that it is platform independent. Put simply, its functions will work on both UNIX and Microsoft Windows machines.

The second part of `stdin.go` contains the following Go code:

```
func main() {
    var f *os.File
    f = os.Stdin
    defer f.Close()

    scanner := bufio.NewScanner(f)
    for scanner.Scan() {
        fmt.Println(">", scanner.Text())
```

```
|    }
```

First, there is a call to `bufio.NewReader()` using standard input (`os.Stdin`) as its parameter. This call returns a `bufio.Scanner` variable, which is used with the `Scan()` function for reading from `os.Stdin` line by line. Each line that is read is printed on the screen before getting the next one. Please note that each line that is printed by the program begins with the `>` character.

The execution of `stdIN.go` will produce the following kind of output:

```
$ go run stdIN.go
This is number 21
> This is number 21
This is Mihalis
> This is Mihalis
Hello Go!
> Hello Go!
Press Control + D on a new line to end this program!
> Press Control + D on a new line to end this program!
```

According to the UNIX way, you can tell a program to stop reading data from standard input by pressing *Ctrl + D*.



The Go code of `stdIN.go` and `stdOUT.go` will be very useful when we talk about UNIX pipes in chapter 8, Telling a UNIX System What to Do, so do not underestimate their simplicity.

Working with command-line arguments

The technique of this section will be illustrated using the Go code of `cla.go`, which will be presented in three parts. The program will find the minimum and the maximum of its command-line arguments.

The first part of the program is as follows:

```
package main

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

What is important here is realizing that getting the command-line arguments requires the use of the `os` package. Additionally, you need another package, named `strconv`, in order to be able to convert a command-line argument, which is given as a string, into an arithmetical data type.

The second part of the program is the following:

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Please give one or more floats.")
        os.Exit(1)
    }

    arguments := os.Args
    min, _ := strconv.ParseFloat(arguments[1], 64)
    max, _ := strconv.ParseFloat(arguments[1], 64)
```

Here, `cla.go` checks whether you have any command-line arguments by checking the length of `os.Args`. This is because the program needs at least one command-line argument to operate. Please note that `os.Args` is a Go slice with `string` values. The first element in the slice is the name of the executable program. Therefore, in order to initialize the `min` and `max`

variables, you will need to use the second element of the `string` type `os.Args` slice that has an index value of 1.

There is an important point here: the fact that you are expecting one or more floats does not necessarily mean that the user will give you valid floats, either by accident or on purpose. However, as we have not talked about error handling in Go so far, `cla.go` assumes that all command-line arguments are in the right format and therefore will be acceptable. As a result, `cla.go` ignores the `error` value returned by the `strconv.ParseFloat()` function using the following statement:

```
| n, _ := strconv.ParseFloat(arguments[i], 64)
```

The preceding statement tells Go that you only want to get the first value returned by `strconv.ParseFloat()` and that you are not interested in the second value, which in this case is an `error` variable, by assigning it to the underscore character. The underscore character, which is called **blank identifier**, is the Go way of discarding a value. If a Go function returns multiple values, you can use the blank identifier multiple times.



Ignoring all or some of the return values of a Go function, especially the `error` values, is a very dangerous technique that should not be used in production code!

The third part comes with the following Go code:

```
for i := 2; i < len(arguments); i++ {
    n, _ := strconv.ParseFloat(arguments[i], 64)

    if n < min {
        min = n
    }
    if n > max {
        max = n
    }
}

fmt.Println("Min:", min)
fmt.Println("Max:", max)
```

Here, you use a `for` loop that will help you to visit all the elements of the `os.Args` slice, which was previously assigned to the `arguments` variable.

Executing `cla.go` will create the following kind of output:

```
$ go run cla.go -10 0 1
Min: -10
Max: 1
$ go run cla.go -10
Min: -10
Max: -10
```

As you might expect, the program does not behave well when it receives erroneous input; the worst thing of all is that it does not generate any warnings to inform the user that there was an error (or several) while processing the command-line arguments of the program:

```
$ go run cla.go a b c 10
Min: 0
Max: 10
```

About error output

This section will present a technique for sending data to **UNIX standard error**, which is the UNIX way of differentiating between actual values and error output.

The Go code for illustrating the use of standard error in Go is included in `stdERR.go` and will be presented in two parts. As writing to standard error requires the use of the file descriptor related to standard error, the Go code of `stdERR.go` will be based on the Go code of `stdOUT.go`.

The first part of the program is as follows:

```
package main

import (
    "io"
    "os"
)
func main() {
    myString := ""
    arguments := os.Args
    if len(arguments) == 1 {
        myString = "Please give me one argument!"
    } else {
        myString = arguments[1]
    }
}
```

So far, `stdERR.go` is almost identical to `stdOUT.go`.

The second portion of `stdERR.go` is the following:

```
    io.WriteString(os.Stdout, "This is Standard output\n")
    io.WriteString(os.Stderr, myString)
    io.WriteString(os.Stderr, "\n")
}
```

You call `io.WriteString()` two times to write to standard error (`os.Stderr`) and one more time to write to standard output (`os.Stdout`).

Executing `stdERR.go` will create the following output:

```
$ go run stdERR.go  
This is Standard output  
Please give me one argument!
```

The preceding output cannot help you to differentiate between data written to standard output and data written to standard error, which can be very useful sometimes. However, if you are using the `bash(1)` shell, there is a trick you can use in order to distinguish between standard output data and standard error data. Almost all UNIX shells offer this functionality in their own way.

When using `bash(1)`, you can redirect the standard error output to a file as follows:

```
$ go run stdERR.go 2>/tmp/stdError  
This is Standard output  
$ cat /tmp/stdError  
Please give me one argument!
```

 *The number after the name of a UNIX program or system call refers to the section of the manual its page belongs to. Although most of the names can be found only once in the manual pages, which means that putting the section number is not required, there are names that can be located in multiple sections because they have multiple meanings, such as `crontab(1)` and `crontab(5)`. Therefore, if you try to retrieve the manual page of a name with multiple meanings without stating its section number, you will get the entry that has the smallest section number.*

Similarly, you can discard error output by redirecting it to the `/dev/null` device, which is like telling UNIX to completely ignore it:

```
$ go run stdERR.go 2>/dev/null  
This is Standard output
```

In the two examples, we redirected the file descriptor of standard error into a file and `/dev/null`, respectively. If you want to save both standard output and standard error to the same file, you can redirect the file descriptor of standard error (`2`) to the file descriptor of standard output (`1`). The following command shows the technique, which is pretty common in UNIX systems:

```
$ go run stdERR.go >/tmp/output 2>&1  
$ cat /tmp/output  
This is Standard output  
Please give me one argument!
```

Finally, you can send both standard output and standard error to `/dev/null` as follows:

```
| $ go run stdERR.go >/dev/null 2>&1
```

Writing to log files

The `log` package allows you to send log messages to the system logging service of your UNIX machine, whereas the `syslog` Go package, which is part of the `log` package, allows you to define the **logging level** and the **logging facility** your Go program will use.

Usually, most system log files of a UNIX OS can be found under the `/var/log` directory. However, the log files of many popular services, such as Apache and Nginx, can be found elsewhere, depending on their configuration.

Generally speaking, using a log file to write some information is considered a better practice than writing the same output on the screen for two reasons: firstly, because the output does not get lost as it is stored on a file, and secondly, because you can search and process log files using UNIX tools, such as `grep(1)`, `awk(1)`, and `sed(1)`, which cannot be done when messages are printed on a terminal window.

The `log` package offers many functions for sending output to the syslog server of a UNIX machine. The list of functions includes `log.Printf()`, `log.Print()`, `log.Println()`, `log.Fatalf()`, `log.Fatalln()`, `log.Panic()`, `log.Panicln()`, and `log.Panicf()`.



Please note that logging functions can be extremely handy for debugging your programs, especially server processes written in Go, so you should not underestimate their power.

Logging levels

The **logging level** is a value that specifies the severity of the log entry. There are various logging levels, including *debug*, *info*, *notice*, *warning*, *err*, *crit*, *alert*, and *emerg*, in reverse order of severity.

Logging facilities

A **logging facility** is like a category used for logging information. The value of the logging facility part can be one of *auth*, *authpriv*, *cron*, *daemon*, *kern*, *lpr*, *mail*, *mark*, *news*, *syslog*, *user*, *UUCP*, *local0*, *local1*, *local2*, *local3*, *local4*, *local5*, *local6*, or *local7* and is defined inside `/etc/syslog.conf`, `/etc/rsyslog.conf` or another appropriate file depending on the server process used for system logging on your UNIX machine.

This means that if a logging facility is not defined and therefore handled, the log messages you send to it might get ignored and therefore lost.

Log servers

All UNIX machines have a separate server process that is responsible for receiving logging data and writing it to log files. There are various log servers that work on UNIX machines. However, only two of them are used on most UNIX variants: `syslogd(8)` and `rsyslogd(8)`.

On macOS machines, the name of the process is `syslogd(8)`. On the other hand, most Linux machines use `rsyslogd(8)`, which is an improved and more reliable version of `syslogd(8)`, which was the original UNIX system utility for message logging.

However, despite the UNIX variant you are using or the name of the server process used for logging, logging works the same way on every UNIX machine and therefore does not affect the Go code that you will write.

The configuration file of `rsyslogd(8)` is usually named `rsyslog.conf` and is located in `/etc`. The contents of a `rsyslog.conf` configuration file, without the lines with comments and lines starting with `$`, might look like the following:

```
$ grep -v '^#' /etc/rsyslog.conf | grep -v '^$' | grep -v '^$\$'  
auth,authpriv.*          /var/log/auth.log  
*.*;auth,authpriv.none  -/var/log/syslog  
daemon.*                 -/var/log/daemon.log  
kern.*                   -/var/log/kern.log  
lpr.*                    -/var/log/lpr.log  
mail.*                   -/var/log/mail.log  
user.*                   -/var/log/user.log  
mail.info                -/var/log/mail.info  
mail.warn                -/var/log/mail.warn  
mail.err                 -/var/log/mail.err  
news.crit                -/var/log/news/news.crit  
news.err                 -/var/log/news/news.err  
news.notice              -/var/log/news/news.notice  
*.=debug;\n    auth,authpriv.none;\n    news.none;mail.none      -/var/log/debug  
*.=info;*.=notice;*.=warn;\n    auth,authpriv.none;\n    cron,daemon.none;\n    mail,news.none          -/var/log/messages  
.emerg                  :omusrmsg:  
daemon.*;mail.*;\n    news.err;\n
```

```
*.=debug;*=info;\  
*=notice;*=warn    |/dev/xconsole  
local7.* /var/log/cisco.log
```

So, in order to send your logging information to `/var/log/cisco.log`, you will need to use the `local7` logging facility. The star character after the name of the facility tells the logging server to catch every logging level that goes to the `local7` logging facility and write it to `/var/log/cisco.log`.

The `syslogd(8)` server has a pretty similar configuration file that is usually `/etc/syslog.conf`. On macOS High Sierra, the `/etc/syslog.conf` file is almost empty and has been replaced by `/etc/asl.conf`. Nevertheless, the logic behind the configuration of `/etc/syslog.conf`, `/etc/rsyslog.conf`, and `/etc/asl.conf` is the same.

A Go program that sends information to log files

The Go code of `logFiles.go` will explain the use of the `log` and `log/syslog` packages to write to the system log files.



Please note that the `log/syslog` package is not implemented on the Microsoft Windows version of Go.

The first part of `logFiles.go` is as follows:

```
package main

import (
    "fmt"
    "log"
    "log/syslog"
    "os"
    "path/filepath"
)

func main() {
    programName := filepath.Base(os.Args[0])
    sysLog, err := syslog.New(syslog.LOG_INFO|syslog.LOG_LOCAL7,
    programName)
```

The first parameter to the `syslog.New()` function is the priority, which is a combination of the logging facility and the logging level. Therefore, a priority of `LOG_NOTICE | LOG_MAIL`, which is mentioned as an example, will send notice logging level messages to the `MAIL` logging facility.

As a result, the preceding code sets the default logging to the `local7` logging facility using the `info` logging level. The second parameter to the `syslog.New()` function is the name of the process that will appear on the logs as the sender of the message. Generally speaking, it is considered a good practice to use the real name of the executable in order to be able to easily find the information you want in the log files at another time.

The second part of the program contains the following Go code:

```
if err != nil {
    log.Fatal(err)
} else {
    log.SetOutput(sysLog)
}
log.Println("LOG_INFO + LOG_LOCAL7: Logging in Go!")
```

After the call to `syslog.New()`, you will have to check the `error` variable that is returned from it so that you can make sure that everything is fine. If everything is OK, which means that the value of the `error` variable is equal to `nil`, you call the `log.SetOutput()` function, which sets the output destination of the default logger, which, in this case, is the logger you created earlier on (`sysLog`). Then, you can use `log.Println()` to send information to the log server.

The third part of `logFiles.go` comes with the following code:

```
sysLog, err = syslog.New(syslog.LOG_MAIL, "Some program!")
if err != nil {
    log.Fatal(err)
} else {
    log.SetOutput(sysLog)
}

log.Println("LOG_MAIL: Logging in Go!")
fmt.Println("Will you see this?")
}
```

The last part shows that you can change the logging configuration in your programs as many times as you want and that you can still use `fmt.Println()` for printing output on the screen.

The execution of `logFiles.go` will create the following output on the screen of a Debian Linux machine:

```
$ go run logFiles.go
Broadcast message from systemd-journald@mail (Tue 2017-10-17 20:06:08 EEST):
logFiles[23688]: Some program![23688]: 2017/10/17 20:06:08 LOG_MAIL: Logging in Go!

Message from syslogd@mail at Oct 17 20:06:08 ...
Some program![23688]: 2017/10/17 20:06:08 LOG_MAIL: Logging in Go!
Will you see this?
```

Executing the same Go code on a macOS High Sierra machine generated the following output:

```
$ go run logFiles.go
Will you see this?
```

Please bear in mind that most UNIX machines store logging information in more than one log file, which is also the case with the Debian Linux machine used in this section. As a result, `logFiles.go` sends its output to multiple log files, which can be verified by the output of the following shell commands:

```
$ grep LOG_MAIL /var/log/mail.log
Oct 17 20:06:08 mail Some program![23688]: 2017/10/17 20:06:08 LOG_MAIL: Logging in Go!
```

```
$ grep LOG_LOCAL7 /var/log/cisco.log
Oct 17 20:06:08 mail logFiles[23688]: 2017/10/17 20:06:08 LOG_INFO + LOG_LOCAL7: Logging in Go!
$ grep LOG_ /var/log/syslog
Oct 17 20:06:08 mail logFiles[23688]: 2017/10/17 20:06:08 LOG_INFO + LOG_LOCAL7: Logging in Go!
Oct 17 20:06:08 mail Some program![23688]: 2017/10/17 20:06:08 LOG_MAIL: Logging in Go!
```

The preceding output shows that the message of the `log.Println("LOG_INFO + LOG_LOCAL7: Logging in Go!")` statement was written on both `/var/log/cisco.log` and `/var/log/syslog`, whereas the message of the `log.Println("LOG_MAIL: Logging in Go!")` statement was written on both `/var/log/syslog` and `/var/log/mail.log`.

The important thing to remember from this section is that if the logging server of a UNIX machine is not configured to catch all logging facilities, some of the log entries you send to it might get discarded without any warnings.

About log.Fatal()

In this section, you will see the `log.Fatal()` function in action. The `log.Fatal()` function is used when something really bad has happened and you just want to exit your program as fast as possible after reporting the bad situation.

The use of `log.Fatal()` is illustrated in the `logFatal.go` program, which contains the following Go code:

```
package main

import (
    "fmt"
    "log"
    "log/syslog"
)

func main() {
    sysLog, err := syslog.New(syslog.LOG_ALERT|syslog.LOG_MAIL, "Some program!")
    if err != nil {
        log.Fatal(err)
    } else {
        log.SetOutput(sysLog)
    }

    log.Fatal(sysLog)
    fmt.Println("Will you see this?")
}
```

Executing `log.Fatal()` will create the following output:

```
$ go run logFatal.go
exit status 1
```

As you can easily understand, the use of `log.Fatal()` terminates a Go program at the point where `log.Fatal()` was called, which is the reason that you did not see the output from the `fmt.Println("Will you see this?")` statement.

However, because of the parameters of the `syslog.New()` call, a log entry has been added to the log file that is related to mail, which is `/var/log/mail.log`:

```
$ grep "Some program" /var/log/mail.log
Jan 10 21:29:34 iMac Some program![7123]: 2019/01/10 21:29:34 &{17 Some program! iMac.local {0 0} 0xc00000c220}
```

About log.Panic()

There are situations where a program will fail for good and you want to have as much information about the failure as possible.

In such difficult times, you might consider using `log.Panic()`, which is the logging function that is illustrated in this section using the Go code of `logPanic.go`.

The Go code of `logPanic.go` is as follows:

```
package main

import (
    "fmt"
    "log"
    "log/syslog"
)

func main() {
    sysLog, err := syslog.New(syslog.LOG_ALERT|syslog.LOG_MAIL, "Some program!")
    if err != nil {
        log.Fatal(err)
    } else {
        log.SetOutput(sysLog)
    }

    log.Panic(sysLog)
    fmt.Println("Will you see this?")
}
```

Executing `logPanic.go` on macOS Mojave will produce the following output:

```
$ go run logPanic.go
panic: &{17 Some program! iMac.local {0 0} 0xc0000b21e0}

goroutine 1 [running]:
log.Panic(0xc0004ef68, 0x1, 0x1)
    /usr/local/Cellar/go/1.11.4/libexec/src/log/log.go:326 +0xc0
main.main()
    /Users/mtsouk/Desktop/mGo2nd/Mastering-Go-Second-Edition/ch01/logPanic.go:17 +0xd6
exit status 2
```

Running the same program on a Debian Linux with Go version 1.3.3 will generate the following output:

```
$ go run logPanic.go
panic: &{17 Some program! mail {0 0} 0xc2080400e0}
```

```
goroutine 16 [running]:  
runtime.panic(0x4ec360, 0xc208000320)  
    /usr/lib/go/src/pkg/runtime/panic.c:279 +0xf5  
log.Panic(0xc208055f20, 0x1, 0x1)  
    /usr/lib/go/src/pkg/log/log.go:307 +0xb6  
main.main()  
    /home/mtsouk/Desktop/masterGo/ch/ch1/code/logPanic.go:17 +0x169  
  
goroutine 17 [runnable]:  
runtime.MHeap_Scavenger()  
    /usr/lib/go/src/pkg/runtime/mheap.c:507  
runtime.goexit()  
    /usr/lib/go/src/pkg/runtime/proc.c:1445  
  
goroutine 18 [runnable]:  
bgsweep()  
    /usr/lib/go/src/pkg/runtime/mgc0.c:1976  
runtime.goexit()  
    /usr/lib/go/src/pkg/runtime/proc.c:1445  
  
goroutine 19 [runnable]:  
runfinq()  
    /usr/lib/go/src/pkg/runtime/mgc0.c:2606  
runtime.goexit()  
    /usr/lib/go/src/pkg/runtime/proc.c:1445  
exit status 2
```

So, the output of `log.Panic()` includes additional low-level information that will hopefully help you to resolve difficult situations that happened in your Go code.

Analogous to the `log.Fatal()` function, the use of the `log.Panic()` function will add an entry to the proper log file and will immediately terminate the Go program.

Writing to a custom log file

Sometimes, you just need to write your logging data in a file of your choice. This can happen for many reasons, including writing debugging data, which sometimes can be too much, without messing with the system log files, keeping your own logging data separate from system logs in order to transfer it or store it in a database, and storing your data using a different format. This subsection will teach you how to write to a custom log file.

The name of the Go utility will be `customLog.go`, and the log file used will be `/tmp/mGo.log`.

The Go code of `customLog.go` will be presented in three parts. The first part is as follows:

```
package main

import (
    "fmt"
    "log"
    "os"
)

var LOGFILE = "/tmp/mGo.log"
```

The path of the log file is hardcoded into `customLog.go` using a global variable named `LOGFILE`. For the purposes of this chapter, that log file resides inside the `/tmp` directory, which is not the usual place for storing data because usually, the `/tmp` directory is emptied after each system reboot. However, at this point, this will save you from having to execute `customLog.go` with root privileges and from putting unnecessary files into system directories. If you ever decide to use the code of `customLog.go` in a real application, you should change that path into something more rational.

The second part of `customLog.go` is as follows:

```
func main() {
    f, err := os.OpenFile(LOGFILE, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)

    if err != nil {
```

```
    fmt.Println(err)
    return
}
defer f.Close()
```

Here, you create a new log file using `os.OpenFile()` using the desired UNIX file permissions (`0644`).

The last part of `customLog.go` is the following:

```
iLog := log.New(f, "customLogLineNumber ", log.LstdFlags)

iLog.SetFlags(log.LstdFlags)
iLog.Println("Hello there!")
iLog.Println("Another log entry!")
```

If you look at the documentation page of the `log` package, which, among other places, can be found at <https://golang.org/pkg/log/>, you will see that the `SetFlags` function allows you to set the output flags (options) for the current logger. The default values as defined by `LstdFlags` are `Ldate` and `Ltime`, which means that you will get the current date and the time in each log entry you write in your log file.

Executing `customLog.go` will generate no visible output. However, after executing it twice, the contents of `/tmp/mGo.log` will be as follows:

```
$ go run customLog.go
$ cat /tmp/mGo.log
customLog 2019/01/10 18:16:09 Hello there!
customLog 2019/01/10 18:16:09 Another log entry!
$ go run customLog.go
$ cat /tmp/mGo.log
customLog 2019/01/10 18:16:09 Hello there!
customLog 2019/01/10 18:16:09 Another log entry!
customLog 2019/01/10 18:16:17 Hello there!
customLog 2019/01/10 18:16:17 Another log entry!
```

Printing line numbers in log entries

In this section, you are going to learn how to print the line number of the source file that executed the statement that wrote the log entry to a log file using the Go code of `customLogLineNumber.go`. This will be presented in two parts. The first part is as follows:

```
package main

import (
    "fmt"
    "log"
    "os"
)

var LOGFILE = "/tmp/mGo.log"

func main() {
    f, err := os.OpenFile(LOGFILE, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)

    if err != nil {
        fmt.Println(err)
        return
    }
    defer f.Close()
```

So far, there is nothing special when compared to the code of `customLog.go`.

The remaining Go code of `customLogLineNumber.go` is as follows:

```
iLog := log.New(f, "customLogLineNumber ", log.LstdFlags)
iLog.SetFlags(log.LstdFlags | log.Lshortfile)
iLog.Println("Hello there!")
iLog.Println("Another log entry!")
}
```

All the magic happens with the `iLog.SetFlags(log.LstdFlags | log.Lshortfile)` statement, which, apart from `log.LstdFlags`, also includes `log.Lshortfile`. The latter flag adds the full filename as well as the line number of the Go statement that printed the log entry in the log entry itself.

Executing `customLogLineNumber.go` will generate no visible output. However, after two executions of `customLogLineNumber.go`, the contents of the `/tmp/mGo.log` log file will be similar to the following:

```
$ go run customLogLineNumber.go
$ cat /tmp/mGo.log
customLogLineNumber 2019/01/10 18:25:14 customLogLineNumber.go:26: Hello there!
customLogLineNumber 2019/01/10 18:25:14 customLogLineNumber.go:27: Another log entry!
$ go run customLogLineNumber.go
$ cat /tmp/mGo.log
customLogLineNumber 2019/01/10 18:25:14 customLogLineNumber.go:26: Hello there!
customLogLineNumber 2019/01/10 18:25:14 customLogLineNumber.go:27: Another log entry!
customLogLineNumber 2019/01/10 18:25:23 customLogLineNumber.go:26: Hello there!
customLogLineNumber 2019/01/10 18:25:23 customLogLineNumber.go:27: Another log entry!
```

As you can see, using long names for your command-line utilities makes your log files difficult to read.



In [Chapter 2](#), Understanding Go Internals, you will learn how to use the `defer` keyword for printing the log messages of a Go function more elegantly.

Error handling in Go

Errors and error handling are two very important Go topics. Go likes error messages so much that it has a separate data type for errors, named `error`. This also means that you can easily create your own **error messages** if you find that what Go gives you is not adequate.

You will most likely need to create and handle your own errors when you are developing your own Go packages.

Please note that having an error condition is one thing, but deciding how to react to an error condition is a totally different thing. Putting it simply, not all error conditions are created equal, which means that some error conditions might require that you immediately stop the execution of a program, whereas other error situations might require printing a warning message for the user to see while continuing the execution of the program. It is up to the developer to use common sense and decide what to do with each `error` value the program might get.



Errors in Go are not like exceptions or errors in other programming languages; they are normal Go objects that get returned from functions or methods just like any other value.

The error data type

There are many scenarios where you might end up having to deal with a new error case while you are developing your own Go application. The `error` data type is here to help you to define your own errors.

This subsection will teach you how to create your own `error` variables. As you will see, in order to create a new `error` variable, you will need to call the `New()` function of the `errors` standard Go package.

The example Go code to illustrate this process can be found in `newError.go` and will be presented in two parts. The first part of the program is as follows:

```
package main

import (
    "errors"
    "fmt"
)

func returnError(a, b int) error {
    if a == b {
        err := errors.New("Error in returnError() function!")
        return err
    } else {
        return nil
    }
}
```

There are many interesting things happening here. First of all, you can see the definition of a Go function other than `main()` for the first time in this book. The name of this new naive function is `returnError()`. Additionally, you can see the `errors.New()` function in action, which takes a `string` value as its parameter. Lastly, if a function should return an `error` variable but there is not an error to report, it returns `nil` instead.



You will learn more about the various types of Go functions in [Chapter 6, What You Might Not Know About Go Packages and Go Functions](#).

The second part of `newError.go` is the following:

```
func main() {
    err := returnError(1, 2)
    if err == nil {
        fmt.Println("returnError() ended normally!")
    } else {
        fmt.Println(err)
    }

    err = returnError(10, 10)
    if err == nil {
        fmt.Println("returnError() ended normally!")
    } else {
        fmt.Println(err)
    }

    if err.Error() == "Error in returnError() function!" {
        fmt.Println("!!!")
    }
}
```

As the code illustrates, most of the time, you need to check whether an `error` variable is equal to `nil` and then act accordingly. What is also presented here is the use of the `errors.Error()` function, which allows you to convert an `error` variable into a `string` variable. This function lets you compare an `error` variable with a `string` variable.



It is considered good practice to send your error messages to the logging service of your UNIX machine, especially when a Go program is a server or some other critical application. However, the code of this book will not follow this principle everywhere



in order to avoid filling your log files with unnecessary data.

Executing `newError.go` will produce the following output:

```
$ go run newError.go
returnError() ended normally!
Error in returnError() function!
!!
```

If you try to compare an `error` variable with a `string` variable without first converting the `error` variable to a `string` variable, the Go compiler will create the following error message:

```
# command-line-arguments
./newError.go:33:9: invalid operation: err == "Error in returnError() function!" (mismatched types error and string)
```

Error handling

Error handling is a very important feature of Go because almost all Go functions return an error message or `nil`, which is the Go way of saying whether there was an error condition while executing a function. You will most likely get tired of seeing the following Go code not only in this book but also in every other Go program you can find on the internet:

```
if err != nil {  
    fmt.Println(err)  
    os.Exit(10)  
}
```



Please do not confuse error handling with printing to error output because they are two totally different things. The former has to do with Go code that handles error conditions, whereas the latter has to do with writing something to the standard error file descriptor.

The preceding code prints the generated error message on the screen and exits your program using `os.Exit()`. Please note that you can also exit your program by calling the `return` keyword inside the `main()` function. Generally speaking, calling `os.Exit()` from a function other than `main()` is considered a bad practice. Functions other than `main()` tend to return the error message before exiting, which is handled by the calling function.

Should you wish to send the error message to the logging service instead of the screen, you should use the following variation of the preceding Go code:

```
if err != nil {  
    log.Println(err)  
    os.Exit(10)  
}
```

Lastly, there is another variation of the preceding code that is used when something really bad has happened and you want to terminate the program:

```
if err != nil {  
    panic(err)  
    os.Exit(10)  
}
```

Panic is a built-in Go function that stops the execution of a program and starts panicking! If you find yourself using `panic` too often, you might want to reconsider your Go implementation. People tend to avoid panic situations in favor of errors wherever possible.

As you will see in the next chapter, Go also offers the `recover` function, which might be able to save you from some bad situations. For now, you will need to wait for [Chapter 2, *Understanding Go Internals*](#), to learn more about the power of the `panic` and `recover` function duo.

It is now time to see a Go program that not only handles error messages generated by standard Go functions, but also defines its own error message. The name of the program is `errors.go` and it will be presented to you in five parts. As you will see, the `errors.go` utility tries to improve the functionality of the `cla.go` program you saw earlier in this chapter by examining whether its command-line arguments are acceptable floats.

The first part of the program is as follows:

```
package main

import (
    "errors"
    "fmt"
    "os"
    "strconv"
)
```

This part of `errors.go` contains the expected `import` statements.

The second portion of `errors.go` comes with the following Go code:

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Please give one or more floats.")
        os.Exit(1)
    }

    arguments := os.Args
    var err error = errors.New("An error")
    k := 1
    var n float64
```

Here, you create a new `error` variable named `err` in order to initialize it with your own value.

The third part of the program is as follows:

```
for err != nil {
    if k >= len(arguments) {
        fmt.Println("None of the arguments is a float!")
        return
    }
    n, err = strconv.ParseFloat(arguments[k], 64)
    k++
}

min, max := n, n
```

This is the trickiest part of the program because if the first command-line argument is not a proper float, you will need to check the next one and keep checking until you find a suitable command-line argument. If none of the command-line arguments are in the correct format, `errors.go` will terminate and print a message on the screen. All this checking happens by examining the `error` value that is returned by `strconv.ParseFloat()`. All this code is just for the accurate initialization of the `min` and `max` variables.

The fourth portion of the program comes with the following Go code:

```
for i := 2; i < len(arguments); i++ {
    n, err := strconv.ParseFloat(arguments[i], 64)
    if err == nil {
        if n < min {
            min = n
        }
        if n > max {
            max = n
        }
    }
}
```

Here, you just process all the right command-line arguments in order to find the minimum and maximum floats among them.

Finally, the last code portion of the program deals with just printing out the current values of the `min` and `max` variables:

```
fmt.Println("Min:", min)
fmt.Println("Max:", max)
```

As you can see from the Go code of `errors.go`, most of its code is about error handling rather than about the actual functionality of the program.

Unfortunately, this is the case with most modern software developed in Go, as well as most other programming languages.

If you execute `errors.go`, you will get the following kind of output:

```
$ go run errors.go a b c
None of the arguments is a float!
$ go run errors.go b c 1 2 3 c -1 100 -200 a
Min: -200
Max: 100
```

Using Docker

In the last section of this chapter, you will learn how to use a Docker image in order to compile and execute your Go code inside the Docker image.

As you might already know, everything in Docker begins with a Docker image; you can either build your own Docker image from scratch or begin with an existing one. For the purposes of this section, the base Docker image will be downloaded from Docker Hub and we will continue with building the Go version of the `Hello World!` program inside that Docker image.

The contents of the `Dockerfile` that will be used are as follows:

```
FROM golang:alpine
RUN mkdir /files
COPY hw.go /files
WORKDIR /files
RUN go build -o /files/hw hw.go
ENTRYPOINT ["/files/hw"]
```

The first line defines the Docker image that will be used. The remaining three commands create a new directory in the Docker image, copy a file (`hw.go`) from the current user directory into the Docker image, and change the current working directory of the Docker image, respectively. The last two commands create a binary executable from the Go source file and specify the path of the binary file that will be executed when you run that Docker image.

So, how do you use that `Dockerfile`? Provided that a file named `hw.go` exists in the current working directory, you can build a new Docker image as follows:

```
$ docker build -t go_hw:v1 .
Sending build context to Docker daemon 2.237MB
Step 1/6 : FROM golang:alpine
alpine: Pulling from library/golang
cd784148e348: Pull complete
```

```

7e273b0dfc44: Pull complete
952c3806fd1a: Pull complete
ee1f873f86f9: Pull complete
7172cd197d12: Pull complete
Digest: sha256:198cb8c94b9ee6941ce6d58f29aadb855f64600918ce602cdeacb018ad77d647
Status: Downloaded newer image for golang:alpine
--> f56365ec0638
Step 2/6 : RUN mkdir /files
--> Running in 18fa7784d82c
Removing intermediate container 18fa7784d82c
--> 9360e95d7cb4
Step 3/6 : COPY hw.go /files
--> 680517bc4aa3
Step 4/6 : WORKDIR /files
--> Running in f3f678fcc38d
Removing intermediate container f3f678fcc38d
--> 640117aea82f
Step 5/6 : RUN go build -o /files/hw hw.go
--> Running in 271cae1fa7f9
Removing intermediate container 271cae1fa7f9
--> dc7852b6aeeb
Step 6/6 : ENTRYPOINT ["/files/hw"]
--> Running in cdadf286f025
Removing intermediate container cdadf286f025
--> 9bec016712c4
Successfully built 9bec016712c4
Successfully tagged go_hw:v1

```

The name of the newly created Docker image is `go_hw:v1`.

If the `golang:alpine` Docker image is already present on your computer, the output of the preceding command will be as follows:

```

$ docker build -t go_hw:v1 .
Sending build context to Docker daemon  2.237MB
Step 1/6 : FROM golang:alpine
--> f56365ec0638
Step 2/6 : RUN mkdir /files
--> Running in 982e6883bb13
Removing intermediate container 982e6883bb13
--> 0632577d852c
Step 3/6 : COPY hw.go /files
--> 68a0feb2e7dc
Step 4/6 : WORKDIR /files
--> Running in d7d4d0c846c2
Removing intermediate container d7d4d0c846c2
--> 6597a7cb3882
Step 5/6 : RUN go build -o /files/hw hw.go
--> Running in 324400d532e0
Removing intermediate container 324400d532e0
--> 5496dd3d09d1
Step 6/6 : ENTRYPOINT ["/files/hw"]
--> Running in bbd24840d6d4
    Removing intermediate container bbd24840d6d4
--> 5a0d2473aa96
Successfully built 5a0d2473aa96

```

```
| Successfully tagged go_hw:v1
```

You can verify that the `go_hw:v1` Docker image exists on your machine as follows:

```
$ docker images
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE
go_hw           v1       9bec016712c4  About a minute ago  312MB
golang          alpine   f56365ec0638  11 days ago   310MB
```

The contents of the `hw.go` file are as follows:

```
package main

import (
    "fmt"
)

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

You can use a Docker image that is on your local computer as follows:

```
$ docker run go_hw:v1
Hello World!
```

There are other more complex ways to execute a Docker image, but for such a naive Docker image, this is the simplest way to use it.

If you want, you can store (**push**) a Docker image at a Docker registry on the internet in order to be able to retrieve it (**pull**) from there afterward.

Docker Hub can be such a place, provided that you have a Docker Hub account, which is easy to create and free. So, after creating a Docker Hub account, you should execute the following commands on your UNIX machine and push that image to Docker Hub:

```
$ docker login
Authenticating with existing credentials...
Login Succeeded
$ docker tag go_hw:v1 "mactsouk/go_hw:v1"
$ docker push "mactsouk/go_hw:v1"
The push refers to repository [docker.io/mactsouk/go_hw]
bdb6946938e3: Pushed
```

```
99e21c42e35d: Pushed
0257968d27b2: Pushed
e121936484eb: Pushed
61b145086eb8: Pushed
789935042c6f: Pushed
b14874cfef59: Pushed
7bff100f35cb: Pushed
v1: digest:
sha256:c179d5d48a51b74b0883e582d53bf861c6884743eb51d9b77855949b5d91dd
el size: 1988
```

The first command is needed to log in to Docker Hub and should be executed only once. The `docker tag` command is needed for specifying the name that a local image will have on Docker Hub and should be executed before the `docker push` command. The last command sends the desired Docker image to Docker Hub, hence the rich output it generates. If you make your Docker image public, anyone will be able to pull it and use it.

You can delete one or more Docker images from your local UNIX machine in many ways. One of them is by using the `IMAGE ID` of a Docker image:

```
$ docker rmi 5a0d2473aa96 f56365ec0638
Untagged: go_hw:v1
Deleted:
sha256:5a0d2473aa96bcdafbef92751a0e1c1bf146848966c8c971f462eb1eb242d2
a6
Deleted:
sha256:5496dd3d09d13c63bf7a9ac52b90bb812690cdfd33cfcc3340509f9bfe6215c
48
Deleted:
sha256:598c4e474b123eccb84f41620d2568665b88a8f176a21342030917576b9d82
a8
Deleted:
sha256:6597a7cb3882b73855d12111787bd956a9ec3abb11d9915d32f2bba4d0e92e
c6
Deleted:
sha256:68a0feb2e7dc5a139eaa7ca04e54c20e34b7d06df30bcd4934ad6511361f2c
b8
Deleted:
sha256:c04452ea9f45d85a999bdc54b55ca75b6b196320c021d777ec1f766d115aa5
14
Deleted:
sha256:0632577d852c4f9b66c0eff2481ba06c49437e447761d655073eb034fa0ac3
33
Deleted:
sha256:52efd0fa2950c8f3c3e2e44fb4eb076c92c0f85fff46a07e060f5974c1007
a9
Untagged: golang:alpine
Untagged:
golang@sha256:198cb8c94b9ee6941ce6d58f29aadb855f64600918ce602cdeacb01
8ad77d647
Deleted:
sha256:f56365ec0638b16b752af4bf17e6098f2fda027f8a71886d6849342266cc3a
```

```
b7
Deleted:
sha256:d6a4b196ed79e7ff124b547431f77e92dce9650037e76da294b3b3aded709b
dd
Deleted:
sha256:f509ec77b9b2390c745af76cd8dd86977c86e9ff377d5663b42b664357c35
22
Deleted:
sha256:1ee98fa99e925362ef980e651c5a685ad04cef41dd80df9be59f158cf9e529
51
Deleted:
sha256:78c8e55f8cb4c661582af874153f88c2587a034ee32d21cb57ac1fef51c610
9e
Deleted:
sha256:7bff100f35cb359a368537bb07829b055fe8e0b1cb01085a3a628ae9c187c7
b8
```



Docker is a huge and really important topic that we will revisit in several chapters of this book.

Exercises and links

- Visit the Go site: <https://golang.org/>.
- Visit the site of Docker: <https://www.docker.com/>.
- Visit the Docker Hub site: <https://hub.docker.com/>.
- Go 2 Draft Designs: <https://blog.golang.org/go2draft>.
- Browse the Go documentation site: <https://golang.org/doc/>.
- Visit the documentation of the `log` package at <https://golang.org/pkg/log/>.
- Visit the documentation of the `log/syslog` package at <https://golang.org/pkg/log/syslog/>.
- Visit the documentation of the `os` package at <https://golang.org/pkg/os/>.
- Have a look at <https://golang.org/cmd/gofmt/>, which is the documentation page of the `gofmt` tool that is used for formatting Go code.
- Write a Go program that finds the sum of all command-line arguments that are valid numbers.
- Write a Go program that finds the average value of floating-point numbers that are given as command-line arguments.
- Write a Go program that keeps reading integers until it gets the word `END` as input.
- Can you modify `customLog.go` in order to write its log data into two log files at the same time? You might need to read [Chapter 8, *Telling a UNIX System What to Do*](#), for help.
- If you are working on a Mac machine, check the **TextMate** editor at <http://macromates.com/>, as well as **BBEdit** at <https://www.barebones.com/products/bbedit/>.
- Visit the documentation page of the `fmt` package at <https://golang.org/pkg/fmt/> to learn more about verbs and the available functions.
- Please visit <https://blog.golang.org/why-generics> to learn more about Go and Generics.

Summary

This chapter talked about many interesting Go topics, including compiling Go code, working with standard input, standard output, and standard error in Go, processing command-line arguments, printing on the screen, and using the logging service of a UNIX system, as well as error handling and some general information about Go. You should consider all these topics as foundational information about Go.

The next chapter is all about the internals of Go, which includes talking about garbage collection, the Go compiler, calling C code from Go, the `defer` keyword, the Go assembler, and WebAssembly, as well as `panic` and `recover`.

Understanding Go Internals

All the Go features that you learned in the previous chapter are extremely handy and you will be using them all the time. However, there is nothing as rewarding as being able to see and understand what is going on in the background and how Go operates behind the scenes.

In this chapter, you are going to learn about the Go garbage collector and the way it works. Additionally, you will find out how to call C code from your Go programs, which you might find indispensable in some situations. However, you will not need to use this capability too often because Go is a very capable programming language.

Furthermore, you will learn how to call Go code from your C programs, along with how to use the `panic()` and `recover()` functions and the `defer` keyword.

This chapter covers:

- The Go compiler
- How garbage collection works in Go
- How to check the operation of the garbage collector
- Calling C code from your Go code
- Calling Go code from a C program
- The `panic()` and `recover()` functions
- The `unsafe` package
- The handy, yet tricky, `defer` keyword
- The `strace(1)` Linux utility
- The `dtrace(1)` utility, which can be found in FreeBSD systems, including macOS Mojave
- Finding out information about your Go environment
- The node trees created by Go
- Creating WebAssembly code from Go
- The Go assembler

The Go compiler

The Go compiler is executed with the help of the `go` tool, which does many more things than just generating executable files.



The `unsafe.go` file used in this section does not contain any special code – the presented commands will work on every valid Go source file.

You can compile a Go source file using the `go tool compile` command. What you will get is an **object file**, which is a file with the `.o` file extension. This is illustrated in the output of the next commands, which were executed on a macOS Mojave machine:

```
$ go tool compile unsafe.go
$ ls -l unsafe.o
-rw-r--r--  1 mtsouk  staff  6926 Jan 22 21:39 unsafe.o
$ file unsafe.o
unsafe.o: current ar archive
```

An object file is a file that contains **object code**, which is machine code in relocatable format that, most of the time, is not directly executable. The biggest advantage of the relocatable format is that it requires as low memory as possible during the linking phase.

If you use the `-pack` command-line flag when executing `go tool compile`, you will get an **archive file** instead of an object file:

```
$ go tool compile -pack unsafe.go
$ ls -l unsafe.a
-rw-r--r--  1 mtsouk  staff  6926 Jan 22 21:40 unsafe.a
$ file unsafe.a
unsafe.a: current ar archive
```

An archive file is a binary file that contains one or more files, and it is mainly used for grouping multiple files into a single file. One of these formats is `ar`, which is used by Go.

You can list the contents of an `.a` archive file as follows:

```
$ ar t unsafe.a  
_.PKGDEF  
_go_.o
```

Another truly valuable command-line flag of the `go tool compile` command that is worth mentioning is `-race`, which allows you to detect **race conditions**. You will learn more about what a race condition is and why you want to avoid it in [Chapter 10, *Concurrency in Go – Advanced Topics*](#).

You will find more uses of the `go tool compile` command near the end of this chapter when we will talk about assembly language and node trees. However, for a tester, try executing the next command:

```
$ go tool compile -S unsafe.go
```

The preceding command generates lots of output that you might find difficult to understand, which means that Go does a pretty good job of hiding any unnecessary complexities, unless you ask for them!

Garbage collection

Garbage collection is the process of freeing up memory space that is not being used. In other words, the garbage collector sees which objects are out of scope and cannot be referenced anymore, and frees the memory space they consume. This process happens in a concurrent way while a Go program is running and not before or after the execution of the program. The documentation of the Go garbage collector implementation states the following:

"The GC runs concurrently with mutator threads, is type accurate (also known as precise), allows multiple GC threads to run in parallel. It is a concurrent mark and sweep that uses a write barrier. It is non-generational and non-compacting. Allocation is done using size segregated per P allocation areas to minimize fragmentation while eliminating locks in the common case."

There is lots of terminology here that will be explained in a while. But first, I will show you a way to look at some parameters of the garbage collection process.

Fortunately, the Go standard library offers functions that allow you to study the operation of the garbage collector and learn more about what the garbage collector secretly does. The relevant code is saved as `gColl.go` and will be presented in three parts.

The first code segment of `gColl.go` is the following:

```
package main

import (
    "fmt"
    "runtime"
    "time"
)

func printStats(mem runtime.MemStats) {
    runtime.ReadMemStats(&mem)
    fmt.Println("mem.Alloc:", mem.Alloc)
    fmt.Println("mem.TotalAlloc:", mem.TotalAlloc)
    fmt.Println("mem.HeapAlloc:", mem.HeapAlloc)
    fmt.Println("mem.NumGC:", mem.NumGC)
    fmt.Println("-----")
}
```

Note that each time you need to get the more recent garbage collection statistics, you will need to call the `runtime.ReadMemStats()` function. The purpose of the `printStats()` function is to avoid writing the same Go code all the time.

The second part of the program is the following:

```
func main() {
    var mem runtime.MemStats
    printStats(mem)

    for i := 0; i < 10; i++ {
        s := make([]byte, 50000000)
        if s == nil {
            fmt.Println("Operation failed!")
        }
    }
    printStats(mem)
```

The `for` loop creates many big Go **slices** in order to allocate large amounts of memory and trigger the garbage collector.

The last part of `gColl.go` comes with the next Go code, which does more memory allocations using Go slices:

```
for i := 0; i < 10; i++ {
    s := make([]byte, 100000000)
    if s == nil {
        fmt.Println("Operation failed!")
    }
    time.Sleep(5 * time.Second)
}
printStats(mem)
```

The output of `gColl.go` on a macOS Mojave machine is next:

```
$ go run gColl.go
mem.Alloc: 66024
mem.TotalAlloc: 66024
mem.HeapAlloc: 66024
mem.NumGC: 0
-----
mem.Alloc: 50078496
mem.TotalAlloc: 500117056
mem.HeapAlloc: 50078496
mem.NumGC: 10
-----
mem.Alloc: 76712
mem.TotalAlloc: 1500199904
mem.HeapAlloc: 76712
```

```
| mem.NumGC: 20
```

Although you are not going to examine the operation of the Go garbage collector all the time, being able to watch the way that it operates on a slow application can save so much time in the long run. I can assure you that you will not regret the time you spend learning about garbage collection in general and, more specifically, about the way the Go garbage collector works.

There is a trick that allows you to get even more detailed output about the way the Go garbage collector operates, which is illustrated in the next command:

```
| $ GODEBUG=gctrace=1 go run gColl.go
```

So, if you put `GODEBUG=gctrace=1` in front of any `go run` command, Go will print analytical data about the operation of the garbage collector. The data will be in this form:

```
| gc 4 @0.025s 0%: 0.002+0.065+0.018 ms clock,  
|   0.021+0.040/0.057/0.003+0.14 ms cpu, 47->47->0 MB, 48 MB goal, 8 P  
| gc 17 @30.103s 0%: 0.004+0.080+0.019 ms clock,  
|   0.033+0/0.076/0.071+0.15 ms cpu, 95->95->0 MB, 96 MB goal, 8 P
```

The preceding output tells us more information about the heap sizes during the garbage collection process. So, let us take the `47->47->0 MB` trinity of values as an example. The first number is the heap size when the garbage collector is about to run. The second value is the heap size when the garbage collector ends its operation. The last value is the size of the live heap.

The tricolor algorithm

The operation of the Go garbage collector is based on the **tricolor algorithm**.



Please note that the tricolor algorithm is not unique to Go and can be used in other programming languages.

Strictly speaking, the official name for the algorithm used in Go is the **tricolor mark-and-sweep algorithm**. It can work concurrently with the program and uses a **write barrier**. This means that when a Go program runs, the Go scheduler is responsible for the scheduling of the application and the garbage collector. This is as if the Go scheduler has to deal with a regular application with multiple **goroutines**! You will learn more about goroutines and the Go scheduler in [chapter 9, Concurrency in Go – Goroutines, Channels, and Pipelines](#).

The core idea behind this algorithm came from Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens and was first illustrated in a paper named *On-the-Fly Garbage Collection: An Exercise in Cooperation*.

The primary principle behind the tricolor mark-and-sweep algorithm is that it divides the objects of the heap into three different sets according to their color, which is assigned by the algorithm. It is now time to talk about the meaning of each color set. The objects of the **black set** are guaranteed to have no pointers to any object of the **white set**.

However, an object of the white set can have a pointer to an object of the black set because this has no effect on the operation of the garbage collector. The objects of the **gray set** might have pointers to some objects of the white set. Finally, the objects of the white set are the candidates for garbage collection.

Please note that no object can go directly from the black set to the white set, which allows the algorithm to operate and be able to clear the objects on the white set. Additionally, no object of the black set can directly point to an object of the white set.

So, when the garbage collection begins, all objects are white and the garbage collector visits all the root objects and colors them gray. The **roots** are the objects that can be directly accessed by the application, which includes global variables and other things on the stack. These objects mostly depend on the Go code of a particular program.

After that, the garbage collector picks a gray object, makes it black, and starts looking at whether that object has pointers to other objects of the white set. This means that when a gray object is being scanned for pointers to other objects, it is colored black. If that scan discovers that this particular object has one or more pointers to a white object, it puts that white object in the gray set. This process keeps going for as long as there exist objects in the gray set. After that, the objects in the white set are unreachable and their memory space can be reused. Therefore, at this point, the elements of the white set are said to be garbage collected.



Please note that if an object of the gray set becomes unreachable at some point in a garbage collection cycle, it will not be collected at this garbage collection cycle but in the next one! Although this is not an optimal situation, it is not that bad.

During this process, the running application is called the **mutator**. The mutator runs a small function named **write barrier**, which is executed each time a pointer in the heap is modified. If the pointer of an object in the heap is modified, which means that this object is now reachable, the write barrier colors it gray and puts it in the gray set.



The mutator is responsible for the invariant that no element of the black set has a pointer to an element of the white set. This is accomplished with the help of the write barrier function. Failing to accomplish this invariant will ruin the garbage collection process and will most likely crash your program in a pretty bad and undesirable way!

As a result, the heap is pictured as a graph of connected objects, which is also shown in *Figure 2.1*, which demonstrates a single phase of a garbage collection cycle.

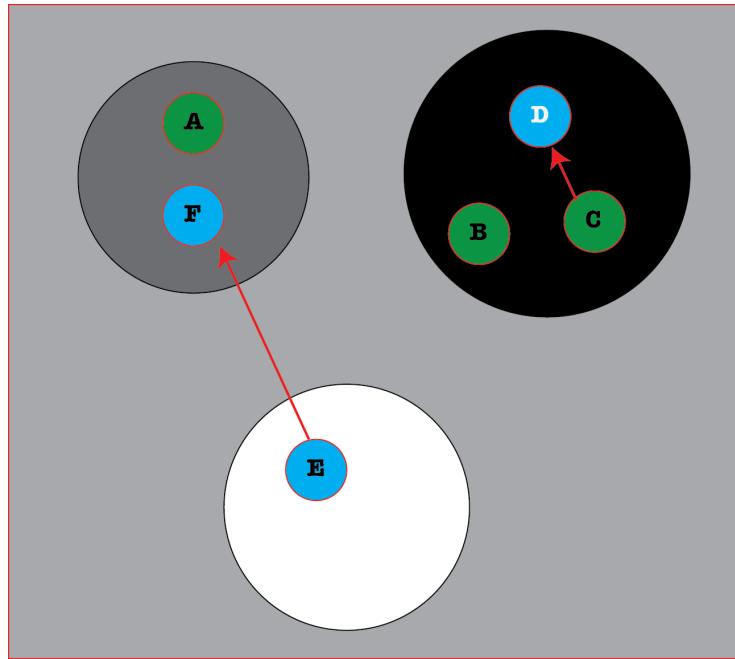


Figure 2.1: The Go garbage collector represents the heap of a program as a graph

So, there are three different colors: black, white, and gray. When the algorithm begins, all objects are colored white. As the algorithm keeps going, white objects are moved into one of the other two sets. The objects that are left in the white set are the ones that are going to be cleared at some point.

In the presented graph, you can see that while object E, which is in the white set, can access object F, it cannot be accessed by any other object because no other object points to object E, which makes it a perfect candidate for garbage collection! Additionally, objects A, B, and C are root objects and are always reachable; therefore, they cannot be garbage collected.

Can you guess what will happen next in that graph? Well, it is not that difficult to realize that the algorithm will have to process the remaining elements of the gray set, which means that both objects A and F will go to the black set. Object A will go to the black set because it is a root element and F will go to the black set because it does not point to any other object while it is in the gray set.

After object A is garbage collected, object F will become unreachable and will be garbage collected in the next cycle of the garbage collector because an unreachable object cannot magically become reachable in the next iteration of the garbage collection cycle.

Go garbage collection can also be applied to variables such as **channels**. When the garbage collector finds out that a channel is unreachable, which is when the channel variable cannot be accessed anymore, it will free its resources even if the channel has not been closed. You will learn more about channels in [Chapter 9, Concurrency in Go – Goroutines, Channels, and Pipelines](#).

Go allows you to manually initiate garbage collection by putting a `runtime.GC()` statement in your Go code. However, have in mind that `runtime.GC()` will block the caller and it might block the entire program, especially if you are running a very busy Go program with many objects. This mainly happens because you cannot perform garbage collections while everything else is rapidly changing, as this will not give the garbage collector the opportunity to clearly identify the members of the white, black, and gray sets. This garbage collection status is also called a **garbage collection safe-point**.

You can find the long and relatively advanced Go code of the garbage collector at <https://github.com/golang/go/blob/master/src/runtime/mgc.go>, which you can study if you want to learn even more information about the garbage collection operation. You can even make changes to that code if you are brave enough!



Note that the Go garbage collector is always being improved by the Go team. They are trying to make it faster by lowering the number of scans it needs to perform over the data of the three sets. However, despite the various optimizations, the general idea behind the algorithm remains the same.

More about the operation of the Go garbage collector

This section will talk more about the Go garbage collector and present additional information about its activities. The main concern of the Go garbage collector is low latency, which basically means short pauses in its operation in order to have a real-time operation. On the other hand, what a program does is create new objects and manipulate existing objects with pointers all the time. This process can end up creating objects that cannot be accessed anymore because there are no pointers pointing to these objects. These objects are then garbage and wait for the garbage collector to clean them up and free their memory space. After that, the memory space that has been freed is ready to be used again.

The mark-and-sweep algorithm is the simplest algorithm used. The algorithm stops the program execution (**stop-the-world garbage collector**) in order to visit all the accessible objects of the heap of a program and *marks* them. After that, it *sweeps* the inaccessible objects. During the mark phase of the algorithm, each object is marked as white, gray, or black. The children of a gray object are colored gray, whereas the original gray object is now colored black. The sweep phase begins when there are no more gray objects to examine. This technique works because there are no pointers from the black set to the white set, which is a fundamental invariant of the algorithm.

Although the mark-and-sweep algorithm is simple, it suspends the execution of the program while it is running, which means that it adds latency to the actual process. Go tries to lower that particular latency by running the garbage collector as a concurrent process and by using the tricolor algorithm described in the previous section. However, other processes can move pointers or create new objects while the garbage collector runs concurrently. This fact can make things pretty difficult for the garbage collector. As a result, the point that will allow the tricolor algorithm

to run concurrently will be when maintaining the fundamental invariant of the mark-and-sweep algorithm: no object of the black set can point to an object of the white set.

The solution to this problem is fixing all the cases that can cause a problem for the algorithm. Therefore, new objects must go to the gray set because this way the fundamental invariant of the mark-and-sweep algorithm cannot be altered. Additionally, when a pointer of the program is moved, you color the object that the pointer points to as gray. The gray set acts like a barrier between the white set and the black set. Finally, each time a pointer is moved, some Go code gets automatically executed, which is the write barrier mentioned earlier, which does some recoloring. The latency introduced by the execution of the write barrier code is the price we have to pay for being able to run the garbage collector concurrently.

Please note that the **Java** programming language has many garbage collectors that are highly configurable with the help of multiple parameters. One of these Java garbage collectors is called **G1** and it is recommended for low-latency applications.



It is really important to remember that the Go garbage collector is a real-time garbage collector that runs concurrently with the other goroutines of a Go program and only optimizes for low latency.

In [Chapter 11, *Code Testing, Optimization, and Profiling*](#), you will learn how to graphically represent the performance of a program, which also includes information about the operations of the Go garbage collector.

Maps, slices, and the Go garbage collector

In this section, I am going to present you with some examples showing why you should be cautious regarding the operation of the garbage collector. The point of this section is to understand that the way you store pointers has a great impact on the performance of the garbage collector, especially when you are dealing with very large amounts of pointers.



The presented examples use pointers, slices, and maps, which are all native Go data types. You will learn more about pointers, slices, and maps in Go in [Chapter 3, Working with Basic Go Data Types](#).

Using a slice

The example in this section will use a **slice** to store a large amount of structures. Each structure stores two integer values. The Go code `sliceGC.go` is as follows:

```
package main

import (
    "runtime"
)

type data struct {
    i, j int
}

func main() {
    var N = 40000000
    var structure []data
    for i := 0; i < N; i++ {
        value := int(i)
        structure = append(structure, data{value, value})
    }

    runtime.GC()
    _ = structure[0]
}
```

The last statement (`_ = structure[0]`) is used for preventing the garbage collector from garbage collecting the `structure` variable too early, as it is not referenced or used outside of the `for` loop. The same technique will be used in the three Go programs that follow. Apart from this important detail, a `for` loop is used for putting all values into structures that are stored in the slice.

Using a map with pointers

In this subsection, we are going to use a map for storing all our pointers as integers. The name of the program is `mapStar.go` and it contains the following Go code:

```
package main

import (
    "runtime"
)

func main() {
    var N = 40000000
    myMap := make(map[int]*int)
    for i := 0; i < N; i++ {
        value := int(i)
        myMap[value] = &value
    }
    runtime.GC()
    _ = myMap[0]
}
```

The name of the map that stores the integer pointers is `myMap`. A `for` loop is used for putting the integer values into the map.

Using a map without pointers

In this subsection, we are going to use a map that stores plain values without pointers. The Go code of `mapNoStar.go` is as follows:

```
package main

import (
    "runtime"
)

func main() {
    var N = 40000000
    myMap := make(map[int]int)
    for i := 0; i < N; i++ {
        value := int(i)
        myMap[value] = value
    }
    runtime.GC()
    _ = myMap[0]
}
```

As before, a `for` loop is used for putting the integer values into the map.

Splitting the map

The implementation of this subsection will split the map into a map of maps, which is also called **sharding**. The program of this subsection is saved as `mapSplit.go` and will be presented in two parts. The first part of `mapSplit.go` contains the following Go code:

```
package main

import (
    "runtime"
)

func main() {
    var N = 40000000
    split := make([]map[int]int, 200)
```

This is where the hash of hashed is defined.

The second part is as follows:

```
for i := range split {
    split[i] = make(map[int]int)
}
for i := 0; i < N; i++ {
    value := int(i)
    split[i%200][value] = value
}
runtime.GC()
_ = split[0][0]
```

This time, we are using two `for` loops: one `for` loop for creating the hash of hashes and another one for storing the desired data in the hash of hashes.

Comparing the performance of the presented techniques

As all four programs are using huge data structures, they are consuming large amounts of memory. Programs that consume lots of memory space trigger the Go garbage collector more often. So, in this subsection, we are going to compare the performance of each one of these four implementations using the `time(1)` command.

What will be important in the presented output is not the exact numbers but the time difference between the four different approaches. Here we go:

```
$ time go run sliceGC.go
real 1.50s
user 1.72s
sys 0.71s
$ time go run mapStar.go

real 13.62s
user 23.74s
sys 1.89s
$ time go run mapNoStar.go

real 11.41s
user 10.35s
sys 1.15s
$ time go run mapSplit.go

real 10.60s
user 10.01s
sys 0.74s
```

So, it turns out that maps slow down the Go garbage collector whereas slices collaborate much better with it. It should be noted here that this is not a problem with maps but a result of the way the Go garbage collector works. However, unless you are dealing with maps that store huge amounts of data, this problem will not become evident in your programs.



*You will learn more about **benchmarking** in Go in [Chapter 11, Code Testing, Optimization, and Profiling](#). Additionally, you will learn a more professional way to*



measure the time it took a command or a program in Go to execute in [Chapter 3](#), Working with Basic Go Data Types.

Enough with garbage collection and its quirks; the topic of the next section will be unsafe code and the `unsafe` standard Go package.

Unsafe code

Unsafe code is Go code that bypasses the type safety and the memory security of Go. Most of the time, unsafe code is related to pointers. However, have in mind that using unsafe code can be dangerous for your programs, so if you are not completely sure that you need to use unsafe code in one of your programs, do not use it!

The use of unsafe code will be illustrated in the `unsafe.go` program, which will be presented in three parts.

The first part of `unsafe.go` is next:

```
package main

import (
    "fmt"
    "unsafe"
)
```

As you will notice, in order to use unsafe code, you will need to import the `unsafe` standard Go package.

The second part of the program comes with the following Go code:

```
func main() {
    var value int64 = 5
    var p1 = &value
    var p2 = (*int32)(unsafe.Pointer(p1))
```

Note the use of the `unsafe.Pointer()` function here, which allows us, at our own risk, to create an `int32` pointer named `p2` that points to an `int64` variable named `value`, which is accessed using the `p1` pointer. Any Go pointer can be converted to `unsafe.Pointer`.



A pointer of type `unsafe.Pointer` can override the type system of Go. This is unquestionably fast but it can also be dangerous if used incorrectly or carelessly. Additionally, it gives developers more control over data.

The last part of `unsafe.go` has the next Go code:

```
    fmt.Println("*p1: ", *p1)
    fmt.Println("*p2: ", *p2)
    *p1 = 5434123412312431212
    fmt.Println(value)
    fmt.Println("*p2: ", *p2)
    *p1 = 54341234
    fmt.Println(value)
    fmt.Println("*p2: ", *p2)
}
```



*You can **dereference a pointer** and get, use, or set its value using the star character (*).*

If you execute `unsafe.go`, you will have the next output:

```
$ go run unsafe.go
*p1: 5
*p2: 5
5434123412312431212
*p2: -930866580
54341234
*p2: 54341234
```

What does this output tell us? It tells us that a 32-bit pointer cannot store a 64-bit integer.

As you will see in the next section, the functions of the `unsafe` package can do many more interesting things with memory.

About the `unsafe` package

Now you have seen the `unsafe` package in action, it is a good time to talk more about what makes it a special kind of package. First of all, if you look at the source code of the `unsafe` package, you might be a little surprised. On a macOS Mojave system with Go version 1.11.4 that is installed using Homebrew (<https://brew.sh/>), the source code of the `unsafe` package is located at `/usr/local/Cellar/go/1.11.4/libexec/src/unsafe/unsafe.go` and its contents without the comments are the following:

```
$ cd /usr/local/Cellar/go/1.11.4/libexec/src/unsafe/
$ grep -v '^//' unsafe.go | grep -v '^$'
package unsafe
type ArbitraryType int
type Pointer *ArbitraryType
func Sizeof(x ArbitraryType) uintptr
func Offsetof(x ArbitraryType) uintptr
func Alignof(x ArbitraryType) uintptr
```

So, where is the rest of the Go code of the `unsafe` package? The answer to that question is relatively simple: the Go compiler implements the `unsafe` package when you import it in your programs.



Many low-level packages, such as `runtime`, `syscall`, and `os`, constantly use the `unsafe` package.

Another example of the unsafe package

In this subsection, you will learn more things about the `unsafe` package and its capabilities with the help of another small Go program named `moreUnsafe.go`. This will be presented in three parts. What `moreUnsafe.go` does is access all the elements of an array using pointers.

The first part of the program is next:

```
package main  
import (  
    "fmt"  
    "unsafe"  
)
```

The second part of `moreUnsafe.go` comes with the next Go code:

```
func main() {  
    array := [...]int{0, 1, -2, 3, 4}  
    pointer := &array[0]  
    fmt.Println(*pointer, " ")  
    memoryAddress := uintptr(unsafe.Pointer(pointer)) +  
        unsafe.Sizeof(array[0])  
  
    for i := 0; i < len(array)-1; i++ {  
        pointer = (*int)(unsafe.Pointer(memoryAddress))  
        fmt.Println(*pointer, " ")  
        memoryAddress = uintptr(unsafe.Pointer(pointer)) +  
            unsafe.Sizeof(array[0])  
    }  
}
```

At first, the `pointer` variable points to the memory address of `array[0]`, which is the first element of the array of integers. Then, the `pointer` variable that points to an integer value is converted to an `unsafe.Pointer()` and then to an `uintptr`. The result is stored in `memoryAddress`.

The value of `unsafe.Sizeof(array[0])` is what gets you to the next element of the array because this is the memory occupied by each array element. So, that value is added to the `memoryAddress` variable in each iteration of the `for`

loop, which allows you to get the memory address of the next array element. The `*pointer` notation dereferences the pointer and returns the stored integer value.

The third part is the following:

```
fmt.Println()
pointer = (*int)(unsafe.Pointer(memoryAddress))
fmt.Print("One more: ", *pointer, " ")
memoryAddress = uintptr(unsafe.Pointer(pointer)) +
unsafe.Sizeof(array[0])
fmt.Println()
```

In the last part, we are trying to access an element of the array that does not exist using pointers and memory addresses. The Go compiler cannot catch such a logical error due to the use of the `unsafe` package and therefore will return something inaccurate.

Executing `moreUnsafe.go` will create the next output:

```
$ go run moreUnsafe.go
0 1 -2 3 4
One more: 824634208008
```

You have just accessed all the elements of a Go array using pointers. However, the real problem here is that when you tried to access an invalid array element, the program did not complain and returned a random number instead.

Calling C code from Go

Although Go intends to make your programming experience better and save you from the quirks of C, C remains a very capable programming language that is still useful. This means that there are situations, such as when using a database or a device driver written in C, that still require the use of C, which means that you will need to work with C code in your Go projects.



If you find yourself using this capability many times in the same project, you might need to reconsider your approach or your choice of programming language.

Calling C code from Go using the same file

The simplest way to call C code from a Go program is to include the C code in your Go source file. This needs special treatment but it is pretty fast and not that difficult.

The name of the Go source file that contains both C and Go code is `cGo.go` and will be presented in three parts.

The first part is next:

```
package main

//#include <stdio.h>
//void callC() {
//    printf("Calling C code!\n");
//}
import "C"
```



As you can see, the C code is included in the comments of the Go program. However, the go tool knows what to do with these kind of comments because of the use of the c Go package.

The second part of the program has the next Go code:

```
import "fmt"

func main() {
```

So, all the other packages should be imported separately.

The last part of `cGo.go` contains the next code:

```
    fmt.Println("A Go statement!")
    C.callC()
    fmt.Println("Another Go statement!")
}
```

In order to execute the `callC()` C function, you will need to call it as `C.callC()`.

Executing `cGo.go` will create the next output:

```
$ go run cGo.go
A Go statement!
Calling C code!
Another Go statement!
```

Calling C code from Go using separate files

Now let us continue with how to call C code from a Go program when the C code is located in a separate file.

First, I will explain the imaginary problem that we will solve with our program. We will need to use two C functions that we have implemented in the past and that we do not want or cannot rewrite in Go.

The C code

This subsection will present you with the C code for the example. It comes in two files: `callc.h` and `callc.c`. The included file (`callc.h`) contains the next code:

```
#ifndef CALLC_H
#define CALLC_H

void cHello();
void printMessage(char* message);

#endif
```

The C source file (`callc.c`) contains the next C code:

```
#include <stdio.h>
#include "callc.h"

void cHello() {
    printf("Hello from C!\n");
}

void printMessage(char* message) {
    printf("Go send me %s\n", message);
}
```

Both the `callc.c` and `callc.h` files are stored in a separate directory, which in this case is going to be `callcLib`. However, you can use any directory name you want.



The actual C code is not important as long as you call the right C functions with the correct type and number of parameters. There is nothing in the C code that tells us that it is going to be used from a Go program. You should look at the Go code for the juicy part.

The Go code

This subsection will present you with the Go source code for the example, which will be named `callc.go` and will be presented to you in three parts.

The first part of `callc.go` comes with the next Go code:

```
package main

// #cgo CFLAGS: -I${SRCDIR}/callClib
// #cgo LDFLAGS: ${SRCDIR}/callC.a
// #include <stdlib.h>
// #include <callC.h>
import "C"
```

The single most important Go statement of the entire Go source file is the inclusion of the `c` package using a separate `import` statement. However, `c` is a virtual Go package that just tells `go build` to preprocess its input file using the `cgo` tool before the Go compiler processes the file. You can still see that you need to use comments to inform the Go program about the C code. In this case, you tell `callc.go` where to find the `callC.h` file as well as where to find the `callC.a` library file that we will create in a while. Such lines begin with `#cgo`.

The second part of the program is the following:

```
import (
    "fmt"
    "unsafe"
)

func main() {
    fmt.Println("Going to call a C function!")
    C.cHello()
```

The last part of `callc.go` is next:

```
    fmt.Println("Going to call another C function!")
    myMessage := C.CString("This is Mihalis!")
    defer C.free(unsafe.Pointer(myMessage))
    C.printMessage(myMessage)

    fmt.Println("All perfectly done!")
}
```

In order to pass a string to a C function from Go, you will need to create a C string using `c.String()`. Additionally, you will need a `defer` statement in order to free the memory space of the C string when it is no longer needed. The `defer` statement includes a call to `c.free()` and another one to `unsafe.Pointer()`.

In the next section, you will see how to compile and execute `callc.go`.

Mixing Go and C code

Now that you have the C code and the Go code, it is time to learn what to do next in order to execute the Go file that calls the C code.

The good news is that you do not need to do anything particularly difficult because all the critical information is contained in the Go file. The only critical thing that you will need to do is compile the C code in order to create a library, which requires the execution of the following commands:

```
$ ls -l callClib/
total 16
-rw-r--r--@ 1 mtsouk  staff  162 Jan 10 09:17 callC.c
-rw-r--r--@ 1 mtsouk  staff   89 Jan 10 09:17 callC.h
$ gcc -c callClib/*.c
$ ls -l callC.o
-rw-r--r--  1 mtsouk  staff  952 Jan 22 22:03 callC.o
$ file callC.o
callC.o: Mach-O 64-bit object x86_64
$ /usr/bin/ar rs callC.a *.o
ar: creating archive callC.a
$ ls -l callC.a
-rw-r--r--  1 mtsouk  staff  4024 Jan 22 22:03 callC.a
$ file callC.a
callC.a: current ar archive
$ rm callC.o
```

After that, you are going to have a file named `callC.a` located in the same directory as the `callC.go` file. The `gcc` executable is the name of the **C compiler**.

Now you are ready to compile the file with the Go code and create a new executable file:

```
$ go build callC.go
$ ls -l callC
-rwxr-xr-x  1 mtsouk  staff  2403184 Jan 22 22:10 callC
$ file callC
callC: Mach-O 64-bit executable x86_64
```

Executing the `callC` executable file will create the next output:

```
$ ./callC
Going to call a C function!
Hello from C!
Going to call another C function!
Go send me This is Mihalis!
All perfectly done!
```



If you are going to call a small amount of C code, then using a single Go file for both C and Go code is highly recommended because of its simplicity. However, if you are going to do something more complex and advanced, creating a static C library should be the preferred way.

Calling Go functions from C code

It is also possible to call a Go function from your C code. Therefore, this section will present you with a small example where two Go functions are going to be called from a C program. The Go package is going to be converted into a **C shared library** that is going to be used in the C program.

The Go package

This subsection will present you with the code of the Go package that will be used in a C program. The name of the Go package needs to be `main` but its filename can be anything you want; in this case, the filename will be `usedByC.go` and it will be presented in three parts.



You will learn more about Go packages in [Chapter 6](#), What You Might Not Know About Go Packages and Go Functions.

The first part of the code of the Go package is next:

```
package main  
  
import "C"  
  
import (  
    "fmt"  
)
```

As I said before, it is mandatory to name the Go package `main`. You will also need to import the `c` package in your Go code.

The second part comes with the following Go code:

```
//export PrintMessage  
func PrintMessage() {  
    fmt.Println("A Go function!")  
}
```

Each Go function that will be called by the C code needs to be exported first. This means that you should put a comment line starting with `//export` before its implementation. After `//export`, you will need to put the name of the function because this is what the C code will use.

The last part of `usedByC.go` is next:

```
//export Multiply  
func Multiply(a, b int) int {  
    return a * b  
}
```

```
| func main() {  
| }
```

The `main()` function of `usedByC.go` needs no code because it is not going to be exported and therefore used by the C program. Additionally, as you also want to export the `Multiply()` function, you will need to put `//export Multiply` before its implementation.

After that, you will need to generate a C shared library from the Go code by executing the following command:

```
| $ go build -o usedByC.o -buildmode=c-shared usedByC.go
```

The preceding command will generate two files named `usedByC.h` and `usedByC.o`:

```
| $ ls -l usedByC.*  
-rw-r--r--@ 1 mtsouk  staff      204 Jan 10 09:17 usedByC.go  
-rw-r--r--  1 mtsouk  staff     1365 Jan 22 22:14 usedByC.h  
-rw-r--r--  1 mtsouk  staff  2329472 Jan 22 22:14 usedByC.o  
$ file usedByC.o  
usedByC.o: Mach-O 64-bit dynamically linked shared library x86_64
```

You should not make any changes to `usedByC.h`.

The C code

The relevant C code can be found in the `willUseGo.c` source file, which will be presented in two parts. The first part of `willUseGo.c` is next:

```
#include <stdio.h>
#include "usedByC.h"

int main(int argc, char **argv) {
    GoInt x = 12;
    GoInt y = 23;

    printf("About to call a Go function!\n");
    PrintMessage();
```

If you already know C, you will understand why you need to include `usedByC.h`; this is the way the C code knows about the available functions of a library.

The second part of the C program is next:

```
    GoInt p = Multiply(x,y);
    printf("Product: %d\n", (int)p);
    printf("It worked!\n");
    return 0;
}
```

The `GoInt p` variable is needed for getting an integer value from a Go function, which is converted to a C integer using the `(int) p` notation.

Compiling and executing `willUseGo.c` on a macOS Mojave machine will create the next output:

```
$ gcc -o willUseGo willUseGo.c ./usedByC.o
$ ./willUseGo
About to call a Go function!
A Go function!
Product: 276
It worked!
```

The defer keyword

The `defer` keyword postpones the execution of a function until the surrounding function returns, which is widely used in file input and output operations because it saves you from having to remember when to close an opened file. The `defer` keyword allows you to put the function call that closes an opened file near to the function call that opened it. As you will learn about the use of `defer` in file-related operations in [Chapter 8, Telling a UNIX System What to Do](#), this section will present two different usages of `defer`. You will also see `defer` in action in the section that talks about the `panic()` and `recover()` built-in Go functions, as well as in the section that is related to logging.

It is very important to remember that **deferred functions** are executed in **last in, first out (LIFO)** order after the return of the surrounding function. Putting it simply, this means that if you `defer` function `f1()` first, function `f2()` second, and function `f3()` third in the same surrounding function, when the surrounding function is about to return, function `f3()` will be executed first, function `f2()` will be executed second, and function `f1()` will be the last one to get executed.

As this definition of `defer` is a little unclear, I think that you will understand the use of `defer` a little better by looking at the Go code and the output of the `defer.go` program, which will be presented in three parts.

The first part of the program is next:

```
package main

import (
    "fmt"
)

func d1() {
    for i := 3; i > 0; i-- {
        defer fmt.Println(i, " ")
    }
}
```

Apart from the `import` block, the preceding Go code implements a function named `d1()` with a `for` loop and a `defer` statement that will be executed three times.

The second part of `defer.go` comes with the next Go code:

```
func d2() {
    for i := 3; i > 0; i-- {
        defer func() {
            fmt.Println(i, " ")
        }()
    }
    fmt.Println()
}
```

In this part of the code, you can see the implementation of another function, which is named `d2()`. The `d2()` function also contains a `for` loop and a `defer` statement, which will also be executed three times. However, this time, the `defer` keyword is applied to an **anonymous function** instead of a single `fmt.Println()` statement. Additionally, the anonymous function takes no parameters.

The last part has the following Go code:

```
func d3() {
    for i := 3; i > 0; i-- {
        defer func(n int) {
            fmt.Println(n, " ")
        }(i)
    }
}

func main() {
    d1()
    d2()
    fmt.Println()
    d3()
    fmt.Println()
}
```

Apart from the `main()` function that calls the `d1()`, `d2()`, and `d3()` functions, you can also see the implementation of the `d3()` function, which has a `for` loop that uses the `defer` keyword on an anonymous function. However, this time, the anonymous function requires one integer parameter named `n`. The Go code tells us that the `n` parameter takes its value from the `i` variable used in the `for` loop.

Executing `defer.go` will create the next output:

```
$ go run defer.go
1 2 3
0 0 0
1 2 3
```

You will most likely find the generated output complicated and challenging to understand, which proves that the operation and the results of the use of `defer` can be tricky if your code is not clear and unambiguous. Let me explain the results in order to get a better idea of how tricky `defer` can be if you do not pay close attention to your code.

We will start with the first line of the output (`1 2 3`) that is generated by the `d1()` function. The values of `i` in `d1()` are `3`, `2`, and `1` in that order. The function that is deferred in `d1()` is the `fmt.Println()` statement; as a result, when the `d1()` function is about to return, you get the three values of the `i` variable of the `for` loop in reverse order because deferred functions are executed in LIFO order.

Now, let me explain the second line of the output that is produced by the `d2()` function. It is really strange that we got three zeros instead of `1 2 3` in the output; however, the reason for that is relatively simple.

After the `for` loop has ended, the value of `i` is `0`, because it is that value of `i` that made the `for` loop terminate. However, the tricky point here is that the deferred anonymous function is evaluated after the `for` loop ends because it has no parameters, which means that it is evaluated three times for an `i` value of `0`, hence the generated output. This kind of confusing code is what might lead to the creation of nasty bugs in your projects, so try to avoid it.

Finally, we will talk about the third line of the output, which is generated by the `d3()` function. Due to the parameter of the anonymous function, each time the anonymous function is deferred, it gets and therefore uses the current value of `i`. As a result, each execution of the anonymous function has a different value to process, hence the generated output.

After that, it should be clear that the best approach to the use of `defer` is the third one, which is exhibited in the `d3()` function, because you intentionally pass the desired variable in the anonymous function in an easy-to-understand way.

Using defer for logging

This section will present an application of `defer` related to logging. The purpose of this technique is to help you organize the logging information of a function in a better way and make it more visible. The name of the Go program that will illustrate the use of the `defer` keyword in logging is `logDefer.go` and it will be presented in three parts.

The first part of `logDefer.go` is as follows:

```
package main

import (
    "fmt"
    "log"
    "os"
)

var LOGFILE = "/tmp/mGo.log"

func one(aLog *log.Logger) {
    aLog.Println("-- FUNCTION one -----")
    defer aLog.Println("-- FUNCTION one -----")

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

The function named `one()` is using `defer` to make sure that the second `aLog.Println()` call is going to be executed just before the function is about to return. Therefore, all log messages from the function are going to be embedded between the opening `aLog.Println()` and the closing `aLog.Println()` calls. As a result, it will be much easier to discover the log messages of that function in your log files.

The second part of `logDefer.go` is the following:

```
func two(aLog *log.Logger) {
    aLog.Println("---- FUNCTION two")
    defer aLog.Println("FUNCTION two -----")

    for i := 10; i > 0; i-- {
        aLog.Println(i)
    }
}
```

```
| } }
```

The function named `two()` also uses `defer` to easily group its log messages. However, this time `two()` uses slightly different messages than function `one()`. It is up to you to choose the format of the logging messages.

The last part of `logDefer.go` contains the following Go code:

```
func main() {
    f, err := os.OpenFile(LOGFILE,
os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer f.Close()

    iLog := log.New(f, "logDefer ", log.LstdFlags)
    iLog.Println("Hello there!")
    iLog.Println("Another log entry!")

    one(iLog)
    two(iLog)
}
```

Executing `logDefer.go` will generate no visible output. However, looking at the contents of `/tmp/mGo.log`, which is the log file that is being used by the program, will make it pretty clear how handy the use of `defer` is in this case:

```
$ cat /tmp/mGo.log
logDefer 2019/01/19 21:15:11 Hello there!
logDefer 2019/01/19 21:15:11 Another log entry!
logDefer 2019/01/19 21:15:11 -- FUNCTION one -----
logDefer 2019/01/19 21:15:11 0
logDefer 2019/01/19 21:15:11 1
logDefer 2019/01/19 21:15:11 2
logDefer 2019/01/19 21:15:11 3
logDefer 2019/01/19 21:15:11 4
logDefer 2019/01/19 21:15:11 5
logDefer 2019/01/19 21:15:11 6
logDefer 2019/01/19 21:15:11 7
logDefer 2019/01/19 21:15:11 8
logDefer 2019/01/19 21:15:11 9
logDefer 2019/01/19 21:15:11 -- FUNCTION one -----
logDefer 2019/01/19 21:15:11 ---- FUNCTION two
logDefer 2019/01/19 21:15:11 10
logDefer 2019/01/19 21:15:11 9
logDefer 2019/01/19 21:15:11 8
logDefer 2019/01/19 21:15:11 7
logDefer 2019/01/19 21:15:11 6
logDefer 2019/01/19 21:15:11 5
logDefer 2019/01/19 21:15:11 4
logDefer 2019/01/19 21:15:11 3
```

```
| logDefer 2019/01/19 21:15:11 2
| logDefer 2019/01/19 21:15:11 1
| logDefer 2019/01/19 21:15:11 FUNCTION two -----
```

Panic and recover

This section will present you with a tricky technique that was first mentioned in the previous chapter. This technique involves the use of the `panic()` and `recover()` functions and will be presented in `panicRecover.go`, which you will see in three parts.

Strictly speaking, `panic()` is a built-in Go function that terminates the current flow of a Go program and starts panicking. On the other hand, the `recover()` function, which is also a built-in Go function, allows you to take back control of a `goroutine` that just panicked using `panic()`.

The first part of the program is next:

```
package main

import (
    "fmt"
)

func a() {
    fmt.Println("Inside a()")
    defer func() {
        if c := recover(); c != nil {
            fmt.Println("Recover inside a()!")
        }
    }()
    fmt.Println("About to call b()")
    b()
    fmt.Println("b() exited!")
    fmt.Println("Exiting a()")
}
```

Apart from the `import` block, this part includes the implementation of the `a()` function. The most important part of function `a()` is the `defer` block of code, which implements an anonymous function that will be called when there is a call to `panic()`.

The second code segment of `panicRecover.go` is next:

```
func b() {
    fmt.Println("Inside b()")
    panic("Panic in b()!")
```

```
|     fmt.Println("Exiting b()")  
| }
```

The last part of the program that illustrates the `panic()` and `recover()` functions is the following:

```
| func main() {  
|     a()  
|     fmt.Println("main() ended!")  
| }
```

Executing `panicRecover.go` will create the next output:

```
| $ go run panicRecover.go  
| Inside a()  
| About to call b()  
| Inside b()  
| Recover inside a()!  
| main() ended!
```

What just happened was really impressive. However, as you can see from the output, the `a()` function did not end normally because its last two statements did not get executed:

```
|     fmt.Println("b() exited!")  
|     fmt.Println("Exiting a()")
```

Nevertheless, the good thing is that `panicRecover.go` ended according to our will without panicking because the anonymous function used in `defer` took control of the situation. Also note that function `b()` knows nothing about function `a()`; however, function `a()` contains Go code that handles the panic condition of function `b()`.

Using the panic function on its own

You can also use the `panic()` function on its own without any attempt to **recover** and this subsection will show its results using the Go code of `justPanic.go`, which will be presented in two parts.

The first part of `justPanic.go` is next:

```
package main

import (
    "fmt"
    "os"
)
```

As you can see, the use of `panic()` does not require any extra Go packages.

The second part of `justPanic.go` comes with the next Go code:

```
func main() {
    if len(os.Args) == 1 {
        panic("Not enough arguments!")
    }

    fmt.Println("Thanks for the argument(s)!")
}
```

If your Go program does not have at least one command-line argument, it will call the `panic()` function. The `panic()` function takes one parameter, which is the error message that you want to print on the screen.

Executing `justPanic.go` on a macOS Mojave machine will create the next output:

```
$ go run justPanic.go
panic: Not enough arguments!

goroutine 1 [running]:
main.main()
    /Users/mtsouk/ch2/code/justPanic.go:10 +0x91
exit status 2
```

Therefore, using the `panic()` function on its own will terminate the Go program without giving you the opportunity to recover. So, the use of the `panic()` and `recover()` pair is much more practical and professional than just using `panic()`.



The output of the `panic()` function looks like the output of the `Panic()` function from the `log` package. However, the `panic()` function sends nothing to the logging service of your UNIX machine.

Two handy UNIX utilities

There are times when a UNIX program fails for some unknown reason or does not perform well and you want to find out why without having to rewrite your code and add a plethora of debugging statements.

This section will present two command-line utilities that allow you to see the C system calls executed by an executable file. The names of the two tools are `strace(1)` and `dtrace(1)` and they allow you to inspect the operation of a program.



Please remember that at the end of the day, all programs that work on UNIX machines end up using C system calls to communicate with the UNIX kernel and perform most of their tasks.

Although both tools can work with the `go run` command, you will get less unrelated output if you first create an executable file using `go build` and use this file. This mainly occurs because, as you already know, `go run` makes various temporary files before actually running your Go code and both tools will see that and try to display information about the temporary files, which is not what you want.

The strace tool

The `strace(1)` command-line utility allows you to trace system calls and signals. As `strace(1)` only works on Linux machines, this section will use a Debian Linux machine to showcase `strace(1)`.

The output that `strace(1)` generates looks like the following:

```
$ strace ls
execve("/bin/ls", ["ls"], /* 15 vars */) = 0
brk(0)                                = 0x186c000
fstat(3, {st_mode=S_IFREG|0644, st_size=35288, ...}) = 0
```

The `strace(1)` output displays each system call with its parameters as well as its return value. Please note that in the UNIX world, a return value of 0 is a good thing.

In order to process a binary file, you will need to put the `strace(1)` command in front of the executable you want to process. However, you will need to interpret the output on your own in order to make useful conclusions from it. The good thing is that tools like `grep(1)` can get you the output that you are really looking for:

```
$ strace find /usr 2>&1 | grep ioctl
ioctl(0, SNDCTL_TMR_TIMEBASE or SNDDRV_TIMER_IOCTL_NEXT_DEVICE or
TCGETS, 0x7ffe3bc59c50) = -1 ENOTTY (Inappropriate ioctl for device)
ioctl(1, SNDCTL_TMR_TIMEBASE or SNDDRV_TIMER_IOCTL_NEXT_DEVICE or
TCGETS, 0x7ffe3bc59be0) = -1 ENOTTY (Inappropriate ioctl for device)
```

The `strace(1)` tool can print count time, calls, and errors for each system call when used with the `-c` command-line option:

```
$ strace -c find /usr 1>/dev/null
% time      seconds   usecs/call    calls    errors syscall
----- -----
 82.88     0.063223        2     39228          getdents
 16.60     0.012664        1     19587          newfstatat
  0.16     0.000119        0     19618          13 open
```

As the normal program output is printed in standard output and the output of `strace(1)` is printed in standard error, the previous command discards the output of the command that is examined and shows the output of `strace(1)`. As you can see from the last line of the output, the `open(2)` system call was called 19,618 times, generated 13 errors, and took about 0.16% of the execution time of the entire command or about 0.000119 seconds.

The dtrace tool

Although debugging utilities such as `strace(1)` and `truss(1)` can trace system calls produced by a process, they can be slow and therefore not appropriate for solving performance problems on busy UNIX systems. Another tool, named **DTrace**, allows you to see what happens behind the scenes on a system-wide basis without the need to modify or recompile anything. It also allows you to work on production systems and watch running programs or server processes dynamically without introducing a big overhead.



Although there is a version of `dtrace(1)` that works on Linux, the `dtrace(1)` tool works best on macOS and the other FreeBSD variants.

This subsection will use the `dtruss(1)` command-line utility that comes with macOS, which is just a `dtrace(1)` script that shows the system calls of a process and saves us from having to write `dtrace(1)` code. Note that both `dtrace(1)` and `dtruss(1)` need root privileges to run.

The output that `dtruss(1)` generates looks like the following:

```
$ sudo dtruss godoc
ioctl(0x3, 0x80086804, 0x7FFEEFBFEC20)          = 0 0
close(0x3)           = 0 0
access("/AppleInternal/XBS/.isChrooted\0", 0x0, 0x0)      = -1 Err#2
thread_selfid(0x0, 0x0, 0x0)           = 1895378 0
geteuid(0x0, 0x0, 0x0)           = 0 0
    getegid(0x0, 0x0, 0x0)           = 0 0
```

So, `dtruss(1)` works the same way as the `strace(1)` utility. Analogous to `strace(1)`, `dtruss(1)` will print system call counts when used with the `-c` parameter:

```
$ sudo dtruss -c go run unsafe.go 2>&1
CALL                                COUNT
access                               1
bsdthread_register                   1
getuid                               1
ioctl                                1
issetugid                            1
```

kqueue	1
write	1
mkdir	2
read	244
kevent	474
fcntl	479
lstat64	553
psynch_cvsignal	649
psynch_cvwait	654

The preceding output will quickly inform you about potential bottlenecks in your Go code or allow you to compare the performance of two different command-line programs.



Utilities like `strace(1)`, `dtrace(1)`, and `dtruss(1)` need some getting used to, but such tools can make our lives so much easier and more comfortable that I strongly suggest you start learning at least one such tool right now.

You can learn more about the `dtrace(1)` utility by reading *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD* by Brendan Gregg and Jim Mauro or by visiting <http://dtrace.org/>.

Please have in mind that `dtrace(1)` is much more powerful than `strace(1)` because it has its own programming language. However, `strace(1)` is more versatile when all you want to do is watch the system calls of an executable file.

Your Go environment

This section will talk about finding out information about your current Go environment using the functions and the properties of the `runtime` package. The name of the program that will be developed in this section is `goEnv.go` and it will be presented in two parts.

The first part of `goEnv.go` is next:

```
package main  
import (  
    "fmt"  
    "runtime"  
)
```

As you will see in a while, the `runtime` package contains functions and properties that will reveal the desired information. The second code portion of `goEnv.go` contains the implementation of the `main()` function:

```
func main() {  
    fmt.Println("You are using ", runtime.Compiler, " ")  
    fmt.Println("on a ", runtime.GOARCH, "machine")  
    fmt.Println("Using Go version", runtime.Version())  
    fmt.Println("Number of CPUs:", runtime.NumCPU())  
    fmt.Println("Number of Goroutines:", runtime.NumGoroutine())  
}
```

Executing `goEnv.go` on a macOS Mojave machine with Go version 1.11.4 will create the next output:

```
$ go run goEnv.go  
You are using gc on a amd64 machine  
Using Go version go1.11.4  
Number of CPUs: 8  
Number of Goroutines: 1
```

The same program generates the next output on a Debian Linux machine with Go version 1.3.3:

```
$ go run goEnv.go  
You are using gc on a amd64 machine  
Using Go version go1.3.3
```

```
| Number of CPUs: 1  
| Number of Goroutines: 4
```

However, the real benefit you can get from being able to find information about your Go environment is illustrated in the next program, named `requiredVersion.go`, which tells you if you are using Go version 1.8 or higher:

```
package main

import (
    "fmt"
    "runtime"
    "strconv"
    "strings"
)

func main() {
    myVersion := runtime.Version()
    major := strings.Split(myVersion, ".") [0] [2]
    minor := strings.Split(myVersion, ".") [1]
    m1, _ := strconv.Atoi(string(major))
    m2, _ := strconv.Atoi(minor)

    if m1 == 1 && m2 < 8 {
        fmt.Println("Need Go version 1.8 or higher!")
        return
    }

    fmt.Println("You are using Go version 1.8 or higher!")
}
```

The `strings` Go standard package is used for splitting the Go version string you get from `runtime.Version()` in order to get its first two parts, whereas the `strconv.Atoi()` function is used for converting a string to an integer.

Executing `requiredVersion.go` on the macOS Mojave machine will create the next output:

```
| $ go run requiredVersion.go  
| You are using Go version 1.8 or higher!
```

However, if you run `requiredVersion.go` on the Debian Linux machine you saw earlier in this section, it will generate the next output:

```
| $ go run requiredVersion.go  
| Need Go version 1.8 or higher!
```

So, by using the Go code of `requiredVersion.go`, you will be able to identify whether your UNIX machine has the required Go version or not.

The go env command

If you need to get a list of all environment variables supported by Go and the Go compiler, along with their current values, then the solution is to execute `go env`.

On my macOS Mojave, which uses Go version 1.11.4, the pretty rich output of `go env` is as follows:

```
$ go env
GOARCH="amd64"
GOBIN=""
GOCACHE="/Users/mtsouk/Library/Caches/go-build"
GOEXE=""
GOFLAGS=""
GOHOSTARCH="amd64"
GOHOSTOS="darwin"
GOOS="darwin"
GOPATH="/Users/mtsouk/go"
GOPROXY=""
GORACE=""
GOROOT="/usr/local/Cellar/go/1.11.4/libexec"
GOTMPDIR=""
GOTOOLDIR="/usr/local/Cellar/go/1.11.4/libexec/pkg/tool/darwin_amd64"
GCCGO="gccgo"
CC="clang"
CXX="clang++"
CGO_ENABLED="1"
GOMOD=""
CGO_CFLAGS="-g -O2"
CGO_CPPFLAGS=""
CGO_CXXFLAGS="-g -O2"
CGO_FFLAGS="-g -O2"
CGO_LDFLAGS="-g -O2"
PKG_CONFIG="pkg-config"
GOGCCFLAGS="-fPIC -m64 -pthread -fno-caret-diagnostics -funused-
arguments -fmessage-length=0 -fdebug-prefix-
map=/var/folders/sk/1tk8cnw501zdtr2hxcj5sv2m0000gn/T/go-
build790367620=/tmp/go-build -gno-record-gcc-switches -fno-common"
```

Please note that some of these environment variables might change if you are using a different Go version, if your username is not `mtsouk`, if you are using a different UNIX variant on a different hardware, or if you are using Go modules (`GOMOD`) by default.

The Go assembler

This section will briefly talk about the assembly language and the **Go assembler**, which is a Go tool that allows you to see the assembly language used by the Go compiler.

As an example, you can see the assembly language of the `goEnv.go` program you saw in the previous section of this chapter by executing the next command:

```
$ GOOS=darwin GOARCH=amd64 go tool compile -S goEnv.go
```

The value of the `GOOS` variable defines the name of the target operating system whereas the value of the `GOARCH` variable defines the compilation architecture. The preceding command was executed on a macOS Mojave machine, hence the use of the `darwin` value for the `GOOS` variable.

The output of the previous command is pretty large even for a small program such as `goEnv.go`. Some of its output is next:

```
""".main STEXT size=859 args=0x0 locals=0x118
    0x0000 00000 (goEnv.go:8)      TEXT      """.main(SB), $280-0
    0x00be 00190 (goEnv.go:9)      PCDATA   $0, $1
    0x0308 00776 (goEnv.go:13)     PCDATA   $0, $5
    0x0308 00776 (goEnv.go:13)     CALL     runtime.convT2E64(SB)
""".init STEXT size=96 args=0x0 locals=0x8
    0x0000 00000 (<autogenerated>:1) TEXT      """.init(SB), $8-0
    0x0000 00000 (<autogenerated>:1) MOVQ     (TLS), CX
    0x001d 00029 (<autogenerated>:1) FUNCDATA $0,
glocals d4dc2f11db048877dbc0f60a22b4adb3(SB)
    0x001d 00029 (<autogenerated>:1) FUNCDATA $1,
glocals 33cdeccccebe80329f1fdbbee7f5874cb(SB)
```

The lines that contain the `FUNCDATA` and `PCDATA` directives are read and used by the Go garbage collector and are automatically generated by the Go compiler.

An equivalent variant of the preceding command is next:

```
| $ GOOS=darwin GOARCH=amd64 go build -gcflags -S goEnv.go
```

The list of valid `GOOS` values includes `android`, `darwin`, `dragonfly`, `freebsd`, `linux`, `nacl`, `netbsd`, `openbsd`, `plan9`, `solaris`, `windows`, and `zos`. On the other hand, the list of valid `GOARCH` values includes `386`, `amd64`, `amd64p32`, `arm`, `armbe`, `arm64`, `arm64be`, `ppc64`, `ppc64le`, `mips`, `mipsle`, `mips64`, `mips64le`, `mips64p32`, `mips64p32le`, `ppc`, `s390`, `s390x`, `sparc`, and `sparc64`.



If you are really interested in the Go assembler and you want more information, you should visit <https://golang.org/doc/asm>.

Node trees

A Go node is a `struct` with a large number of properties. You will learn more about defining and using Go **structures** in [Chapter 4, The Uses of Composite Types](#). Everything in a Go program is parsed and analyzed by the modules of the Go compiler according to the grammar of the Go programming language. The final product of this analysis is a **tree** that is specific to the provided Go code and represents the program in a different way that is suited to the compiler rather than to the developer.



Please note that `go tool 6g -W test.go` does not work on newer Go versions. You should use `go tool compile -W test.go` instead.

This section will first use the following Go code, which is saved as `nodeTree.go`, as an example in order to see the kind of low-level information the `go` tool can provide us with:

```
package main

import (
    "fmt"
)

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

The Go code of `nodeTree.go` is pretty easy to understand, so you will not be surprised by its output, which is next:

```
$ go run nodeTree.go
Hello there!
```

Now it is time to see some internal Go workings by executing the next command:

```
$ go tool compile -W nodeTree.go
before walk main
.  CALLFUNC 1(8) tc(1) STRUCT-(int, error)
.  .  NAME-fmt.Println a(true) l(263) x(0) class(PFUNC) tc(1) used FUNC-func(...interface {}) (int, error)
.  .  DDDARG 1(8) esc(no) PTR64-*[1]interface {}
CALLFUNC-list
.  CONVIFACE 1(8) esc(h) tc(1) implicit(true) INTER-interface {}
.  .  NAME-main.statictmp_0 a(true) l(8) x(0) class(PEXTERN) tc(1) used string

.  VARKILL 1(8) tc(1)
.  .  NAME-main..autotmp_0 a(true) l(8) x(0) class(PAUTO) esc(N) used ARRAY-[1]interface {}
after walk main
.  CALLFUNC-init
.  .  AS 1(8) tc(1)
.  .  .  NAME-main..autotmp_0 a(true) l(8) x(0) class(PAUTO) esc(N) tc(1) addrtaken assigned used ARRAY-[1]interface {}

.  .  AS 1(8) tc(1)
.  .  .  NAME-main..autotmp_2 a(true) l(8) x(0) class(PAUTO) esc(N) tc(1) assigned used PTR64-*[1]interface {}
.  .  .  ADDR 1(8) tc(1) PTR64-*[1]interface {}
.  .  .  .  NAME-main..autotmp_0 a(true) l(8) x(0) class(PAUTO) esc(N) tc(1) addrtaken assigned used ARRAY-[1]interface {}

.  .  BLOCK 1(8)
.  .  BLOCK-list
.  .  .  AS 1(8) tc(1) hascall
.  .  .  .  INDEX 1(8) tc(1) assigned bounded hascall INTER-interface {}
.  .  .  .  .  IND 1(8) tc(1) implicit(true) assigned hascall ARRAY-[1]interface {}
.  .  .  .  .  .  NAME-main..autotmp_2 a(true) l(8) x(0) class(PAUTO) esc(N) tc(1) assigned used PTR64-*[1]interface {}
.  .  .  .  .  .  LITERAL-0 1(8) tc(1) int
.  .  .  .  EFACE 1(8) tc(1) INTER-interface {}
.  .  .  .  .  .  ADDR a(true) l(8) tc(1) PTR64-*uint8
.  .  .  .  .  .  .  NAME-type.string a(true) x(0) class(PEXTERN) tc(1) uint8
.  .  .  .  .  .  .  ADDR 1(8) tc(1) PTR64-*string
.  .  .  .  .  .  .  .  NAME-main.statictmp_0 a(true) l(8) x(0) class(PEXTERN) tc(1) addrtaken used string

.  .  BLOCK 1(8)
.  .  BLOCK-list
.  .  .  AS 1(8) tc(1) hascall
.  .  .  .  NAME-main..autotmp_1 a(true) l(8) x(0) class(PAUTO) esc(N) tc(1) assigned used SLICE-[]interface {}
.  .  .  .  .  SLICEARR 1(8) tc(1) hascall SLICE-[]interface {}
.  .  .  .  .  .  NAME-main..autotmp_2 a(true) l(8) x(0) class(PAUTO) esc(N) tc(1) assigned used PTR64-*[1]interface {}
.  .  CALLFUNC 1(8) tc(1) hascall STRUCT-(int, error)
.  .  .  NAME-fmt.Println a(true) l(263) x(0) class(PFUNC) tc(1) used FUNC-func(...interface {}) (int, error)
.  .  DDDARG 1(8) esc(no) PTR64-*[1]interface {}
CALLFUNC-list
```

```

. . . AS l(8) tc(1)
. . . . INDREGSP-SP a(true) l(8) x(0) tc(1) addrtaken main. __ SLICE-[]interface {}
. . . . NAME-main..autotmp_1 a(true) l(8) x(0) class(PAUTO) esc(N) tc(1) assigned used SLICE-[]interface {}

. VARKILL l(8) tc(1)
. . NAME-main..autotmp_0 a(true) l(8) x(0) class(PAUTO) esc(N) tc(1) addrtaken assigned used ARRAY-[1]interface {}

before walk init
. IF l(1) tc(1)
. . GT l(1) tc(1) bool
. . . NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1) assigned used uint8
. . . LITERAL-1 l(1) tc(1) uint8
. IF-body
. . RETURN l(1) tc(1)

. IF l(1) tc(1)
. . EQ l(1) tc(1) bool
. . . NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1) assigned used uint8
. . . LITERAL-1 l(1) tc(1) uint8
. IF-body
. . CALLFUNC l(1) tc(1)
. . . NAME-runtime.throwinit a(true) x(0) class(PFUNC) tc(1) used FUNC-func()

. AS l(1) tc(1)
. . NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1) assigned used uint8
. . LITERAL-1 l(1) tc(1) uint8

. CALLFUNC l(1) tc(1)
. . NAME-fmt.init a(true) l(1) x(0) class(PFUNC) tc(1) used FUNC-func()

. AS l(1) tc(1)
. . NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1) assigned used uint8
. . LITERAL-2 l(1) tc(1) uint8

. RETURN l(1) tc(1)
after walk init
. IF l(1) tc(1)
. . GT l(1) tc(1) bool
. . . NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1) assigned used uint8
. . . LITERAL-1 l(1) tc(1) uint8
. IF-body
. . RETURN l(1) tc(1)

. IF l(1) tc(1)
. . EQ l(1) tc(1) bool
. . . NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1) assigned used uint8
. . . LITERAL-1 l(1) tc(1) uint8
. IF-body
. . CALLFUNC l(1) tc(1) hascall
. . . NAME-runtime.throwinit a(true) x(0) class(PFUNC) tc(1) used FUNC-func()

. AS l(1) tc(1)
. . NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1) assigned used uint8
. . LITERAL-1 l(1) tc(1) uint8

. CALLFUNC l(1) tc(1) hascall
. . NAME-fmt.init a(true) l(1) x(0) class(PFUNC) tc(1) used FUNC-func()

. AS l(1) tc(1)
. . NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1) assigned used uint8
. . LITERAL-2 l(1) tc(1) uint8

. RETURN l(1) tc(1)

```

As you can understand, the Go compiler and its tools do many things behind the scenes, even for a small program such as `nodeTree.go`.



The `-w` parameter tells the `go tool compile` command to print the debug parse tree after the type checking.

Look at the output of the next two commands:

```

$ go tool compile -W nodeTree.go | grep before
before walk main
before walk init
$ go tool compile -W nodeTree.go | grep after
after walk main
after walk init

```

As you can see, the `before` keyword is about the beginning of the execution of a function. If your program had more functions, you would have got more output, which is illustrated in the next example:

```
$ go tool compile -W defer.go | grep before
before d1
before d2
before d3
before main
before d2.func1
before d3.func1
before init
before type..hash.[2]interface {}
before type..eq.[2]interface {}
```

The previous example uses the Go code of `defer.go`, which is much more complicated than `nodeTree.go`. However, it should be obvious that the `init()` function is automatically generated by Go as it exists in both outputs of `go tool compile -W` (`nodeTree.go` and `defer.go`). I will now present you with a juicier version of `nodeTree.go`, named `nodeTreeMore.go`:

```
package main

import (
    "fmt"
)

func functionOne(x int) {
    fmt.Println(x)
}

func main() {
    varOne := 1
    varTwo := 2
    fmt.Println("Hello there!")
    functionOne(varOne)
    functionOne(varTwo)
}
```

The `nodeTreeMore.go` program has two variables, named `varOne` and `varTwo`, and an extra function named `functionOne`. Searching the output of `go tool compile -W` for `varOne`, `varTwo`, and `functionOne` will reveal the following information:

```
$ go tool compile -W nodeTreeMore.go | grep functionOne | uniq
before walk functionOne
after walk functionOne
. . NAME-main.functionOne a(true) l(7) x(0) class(PFUNC) tc(1) used FUNC-func(int)
$ go tool compile -W nodeTreeMore.go | grep varTwo | uniq
. . NAME-main.varTwo a(true) g(2) l(13) x(0) class(PAUTO) tc(1) used int
. . . NAME-main.varTwo a(true) g(2) l(13) x(0) class(PAUTO) tc(1) used int
$ go tool compile -W nodeTreeMore.go | grep varOne | uniq
. . . NAME-main.varOne a(true) g(1) l(12) x(0) class(PAUTO) tc(1) used int
. . . . NAME-main.varOne a(true) g(1) l(12) x(0) class(PAUTO) tc(1) used int
```

So, `varOne` is represented as `NAME-main.varOne` while `varTwo` is denoted by `NAME-main.varTwo`. The `functionOne()` function is referenced as `NAME-main.functionOne`. Consequently, the `main()` function is referenced as `NAME-main`.

Now, let us see the next code of the debug parse tree of `nodeTreeMore.go`:

```
before walk functionOne
. AS l(8) tc(1)
. . NAME-main..autotmp_2 a(true) l(8) x(0) class(PAUTO) esc(N) tc(1) assigned used int
. . NAME-main.x a(true) g(1) l(7) x(0) class(PPARAM) tc(1) used int
```

This data is related to the definition of `functionOne()`. The `l(8)` string tells us that the definition of this node can be found in line eight, that is, after reading line seven. The `NAME-main..autotmp_2` integer variable is automatically generated by the compiler.

The next part of the debug parse tree output that will be explained here is:

```
. CALLFUNC l(15) tc(1)
. . NAME-main.functionOne a(true) l(7) x(0) class(PFUNC) tc(1) used FUNC-func(int)
. CALLFUNC-list
. . NAME-main.varOne a(true) g(1) l(12) x(0) class(PAUTO) tc(1) used int
```

The first line says that at line 15 of the program, which is specified by `l(15)`, you will call `NAME-main.functionOne`, which is defined at line seven of the program, as specified by `l(7)`, which is a function that requires a single integer parameter, as specified by `FUNC-func(int)`. The function list of parameters, which is specified after `CALLFUNC-list`, includes the `NAME-main.varOne` variable, which is defined at line 12 of the program, as `l(12)` shows.

Finding out more about go build

If you want to learn more about what is happening behind the scenes when you execute a `go build` command, you should add the `-x` flag to it:

```
$ go build -x defer.go
WORK=/var/folders/sk/1tk8cnw501zdtr2hxcj5sv2m000gn/T/go-build254573394
mkdir -p $WORK/b001/
cat >$WORK/b001/importcfg.link << 'EOF' # internal
packagefile command-line-arguments=/Users/mtsouk/Library/Caches/go-build/9d/9d6ca8651e083f3662adf82bb90a00837fc76f55839e65c710'
packagefile fmt=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/fmt.a
packagefile runtime=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/runtime.a
packagefile errors=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/errors.a
packagefile io=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/io.a
packagefile math=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/math.a
packagefile os=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/os.a
packagefile reflect=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/reflect.a
packagefile strconv=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/strconv.a
packagefile sync=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/sync.a
packagefile unicode=utf8=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/unicode=utf8.a
packagefile internal/bytealg=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/internal/bytealg.a
packagefile internal/cpu=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/internal/cpu.a
packagefile runtime/internal/atomic=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/runtime/internal/atomic.a
packagefile runtime/internal/sys=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/runtime/internal/sys.a
packagefile sync/atomic=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/sync/atomic.a
packagefile internal/poll=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/internal/poll.a
packagefile internal/syscall/unix=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/internal/syscall/unix.a
packagefile internal/testlog=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/internal/testlog.a
packagefile syscall=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/syscall.a
packagefile time=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/time.a
packagefile unicode=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/unicode.a
packagefile math/bits=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/math/bits.a
packagefile internal/race=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/internal/race.a
EOF
mkdir -p $WORK/b001/exe/
cd .
/usr/local/Cellar/go/1.11.4/libexec/pkg/tool/darwin_amd64/link -o $WORK/b001/exe/a.out -importcfg $WORK/b001/importcfg.link -b /usr/local/Cellar/go/1.11.4/libexec/pkg/tool/darwin_amd64/buildid -w $WORK/b001/exe/a.out # internal
mv $WORK/b001/exe/a.out defer
rm -r $WORK/b001/
```

Once again, there are many things happening in the background and it is good to be aware of them. However, most of the time, you will not have to deal with the actual commands of the compilation process.

Creating WebAssembly code

Go allows you to create WebAssembly code with the help of the `go` tool. Before I illustrate the process, I will share more information about WebAssembly.

A quick introduction to WebAssembly

WebAssembly (Wasm) is a machine model and executable format targeting a virtual machine. It is designed for efficiency, both in speed and file size. This means that you can use a WebAssembly binary on any platform you want without a single change.

WebAssembly comes in two formats: plain text format and binary format. Plain text format WebAssembly files have the `.wat` extension, whereas binary files have the `.wasm` file extension. Notice that once you have a WebAssembly binary file, you will have to load and use it using the JavaScript API.

Apart from Go, WebAssembly can also be generated from other programming languages that have support for static typing, including Rust, C, and C++.

Why is WebAssembly important?

WebAssembly is important for the following reasons:

- WebAssembly code runs at a speed that is pretty close to the native speed, which means that WebAssembly is fast.
- You can create WebAssembly code from many programming languages, which might include programming languages that you already know.
- Most modern web browsers natively support WebAssembly without the need for a plugin or any other software installation.
- WebAssembly code is much faster than **JavaScript** code.

Go and WebAssembly

For Go, WebAssembly is just another architecture. Therefore, you can use the cross-compilation capabilities of Go in order to create WebAssembly code.

You will learn more about the cross-compilation capabilities of Go in [Chapter 11](#), *Code Testing, Optimization, and Profiling*. For now, notice the values of the `GOOS` and `GOARCH` environment variables that are being used while compiling Go code into WebAssembly because this is where all the magic happens.

An example

In this section, we are going to see how a Go program can be compiled into WebAssembly code. The Go code of `toWasm.go` is the following:

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Creating WebAssembly code from Go!")
}
```

The important things to notice here are that there is no sign of WebAssembly in this code and that `toWasm.go` can be compiled and executed on its own, which means that it has no external dependencies related to WebAssembly.

The last step that you will need to take in order to create the WebAssembly code is executing the following command:

```
$ GOOS=js GOARCH=wasm go build -o main.wasm toWasm.go
$ ls -l
total 4760
-rwxr-xr-x 1 mtsouk  staff  2430633 Jan 19 21:00 main.wasm
-rw-r--r--@ 1 mtsouk  staff       100 Jan 19 20:53 toWasm.go
$ file main.wasm
main.wasm: , created: Thu Oct 25 20:41:08 2007, modified: Fri May 28 13:51:43 2032
```

So, the values of `GOOS` and `GOARCH` found in the first command tell Go to create WebAssembly code. If you do not put the right `GOOS` and `GOARCH` values, the compilation will not generate WebAssembly code or it might fail.

Using the generated WebAssembly code

So far, we have only generated a WebAssembly binary file. However, there are still some steps that you will need to take in order to use that WebAssembly binary file and see its results on the window of a web browser.



*If you are using **Google Chrome** as your web browser, then there is a flag that allows you to enable Liftoff, which is a compiler for WebAssembly that will theoretically improve the running time of WebAssembly code. It does not hurt to try it! In order to change that flag, you should visit*

`chrome://flags/#enable-webassembly-baseline.`

The first step is to copy `main.wasm` into a directory of your web server. Next, you will need to execute the following command:

```
$ cp "$(go env GOROOT)/misc/wasm/wasm_exec.js" .
```

This will copy `wasm_exec.js` from the Go installation into the current directory. You should put that file in the same directory of your web server that you put `main.wasm`.

The JavaScript code found in `wasm_exec.js` will not be presented here. On the other hand, the HTML code of `index.html` will be presented:

```
<HTML>

<head>
  <meta charset="utf-8">
  <title>Go and WebAssembly</title>
</head>

<body>
  <script src="wasm_exec.js"></script>
  <script>
    if (!WebAssembly.instantiateStreaming) { // polyfill
      WebAssembly.instantiateStreaming = async (resp, importObject) => {
        const source = await (await resp).arrayBuffer();
        return await WebAssembly.instantiate(source, importObject);
      };
    }

    const go = new Go();
    let mod, inst;
    WebAssembly.instantiateStreaming(fetch("main.wasm"), go.importObject).then((result) => {
      mod = result.module;
      inst = result.instance;
      document.getElementById("runButton").disabled = false;
    }).catch((err) => {
      console.error(err);
    });
  </script>
  <div>
    <button id="runButton">Run</button>
    <pre>${mod}</pre>
  </div>
</body>
</HTML>
```

```

        inst = await WebAssembly.instantiate(mod, go.importObject);
    }
</script>

<button onClick="run()" id="runButton" disabled>Run</button>
</body>
</HTML>

```

Please note that the `Run` button created by the HTML code will not get activated until the WebAssembly code is loaded.

The next figure shows the output of the WebAssembly code as presented in the JavaScript console of the Google Chrome web browser. Other web browsers will show a similar output.

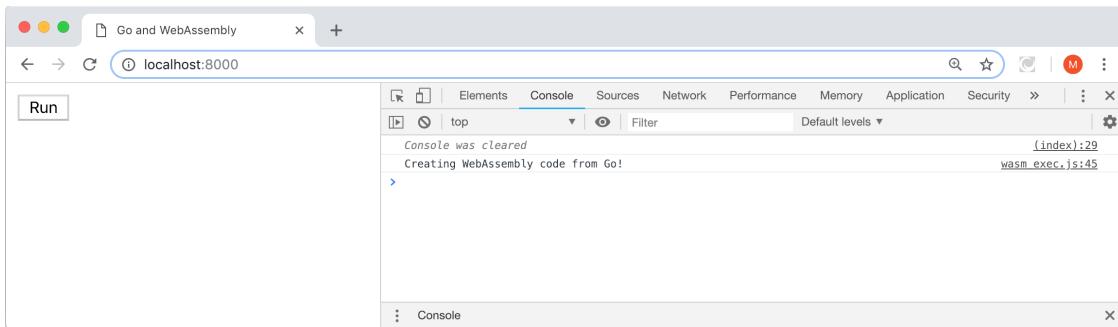


Figure 2.2: Serving WebAssembly code generated by Go



*In [Chapter 12](#), *The Foundations of Network Programming in Go*, you will learn how to develop your own web servers in Go.*

However, I believe there is a much easier and simpler way to test your WebAssembly applications and that includes the use of **Node.js**. There is no need for a web server because Node.js is a **JavaScript** runtime built on Chrome's V8 JavaScript engine.

Provided that you have Node.js already installed on your local machine, you can execute the following command:

```

$ export PATH="$PATH:$(/go env GOROOT)/misc/wasm"
$ GOOS=js GOARCH=wasm go run .
Creating WebAssembly code from Go!

```

The output of the second command verifies that the WebAssembly code is correct and generates the desired message. Please note that the first command is not strictly required as it just changes the current value of the `PATH` environment variable in order to include the directory where the current Go installation stores its WebAssembly-related files.

General Go coding advice

The following is a list of practical advice that will help you to write better Go code:

- If you have an error in a Go function, either log it or return it; do not do both unless you have a really good reason for doing so.
- Go interfaces define behaviors not data and data structures.
- Use the `io.Reader` and `io.Writer` interfaces when possible because they make your code more extensible.
- Make sure that you pass a pointer to a variable to a function only when needed. The rest of the time, just pass the value of the variable.
- Error variables are not `string` variables; they are `error` variables!
- Do not test your Go code on production machines unless you have a really good reason to do so.
- If you do not really know a Go feature, test it before using it for the first time, especially if you are developing an application or a utility that will be used by a large number of users.
- If you are afraid of making mistakes, you will most likely end up doing nothing really interesting. Experiment as much as you can!

Exercises and links

- Learn more about the `unsafe` standard Go package by visiting its documentation page at <https://golang.org/pkg/unsafe/>.
- Visit the web site of DTrace at <http://dtrace.org/>.
- Use `strace(1)` on your Linux machine to inspect the operation of some standard UNIX utilities such as `cp(1)` and `ls(1)`. What do you see?
- If you are using a macOS machine, use `dtruss(1)` to see how the `sync(8)` utility works.
- Write your own example where you use your own C code from a Go program.
- Write a Go function and use it in a C program.
- You can find more information about the functions of the `runtime` package by visiting <https://golang.org/pkg/runtime/>.
- Reading research papers might be difficult but it is very rewarding. Please download the *On-the-Fly Garbage Collection: An Exercise in Cooperation* paper and read it. The paper can be found in many places, including <https://dl.acm.org/citation.cfm?id=359655>.
- Visit <https://github.com/gasche/gc-latency-experiment> in order to find benchmarking code for the garbage collector of various programming languages.
- The Node.js web site can be found at <https://nodejs.org/en/>.
- You can learn more about WebAssembly at <https://webassembly.org/>.
- Should you wish to learn more about garbage collection, you should definitely visit <http://gchandbook.org/>.
- Visit the documentation page of `cgo` at <https://golang.org/cmd/cgo/>.

Summary

This chapter discussed many interesting Go topics, including theoretical and practical information about the Go garbage collector; how to call C code from your Go programs; the handy and sometimes tricky `defer` keyword; the `panic()` and `recover()` functions; the `strace(1)`, `dtrace(1)`, and `ptrace(1)` UNIX tools; the use of the `unsafe` standard Go package; how to generate WebAssembly code from Go; and assembly code generated by Go. Finally, it shared information about your Go environment using the `runtime` package and showed how to reveal and explain the node tree of a Go program, before giving you some handy Go coding advice.

What you should remember from this chapter is that tools such as the `unsafe` Go package and the ability to call C code from Go are usually used on three occasions: firstly, when you want the best performance and you want to sacrifice some Go safety for it; secondly, when you want to communicate with another programming language; and thirdly, when you want to implement something that cannot be implemented in Go.

In the next chapter, we will start learning about the basic data types that come with Go, including **arrays**, **slices**, and **maps**. Despite their simplicity, these data types are the building blocks of almost every Go application because they are the basis of more complex data structures, which allows you to store your data and move information inside your Go projects.

Additionally, you will learn about **pointers**, which can also be found in other programming languages, Go **loops**, and the unique way that Go works with dates and times.

Working with Basic Go Data Types

The previous chapter talked about many fascinating topics including the way the Go garbage collector works, the `panic()` and `recover()` functions, the `unsafe` package, how to call C code from a Go program, and how to call Go code from a C program, as well as the node tree created by the Go compiler when compiling a Go program.

The core subject of this chapter is the basic data types of Go. This list includes **numeric types**, **arrays**, **slices**, and **maps**. Despite their simplicity, these data types can help you to make numeric calculations. You can also store, retrieve, and alter the data of your programs in a very convenient and quick way. The chapter also covers **pointers**, **constants**, **loops**, and working with dates and times in Go.

In this chapter, you will learn about:

- Numeric data types
- Go arrays
- Go slices and why slices are much better than arrays
- How to append an array to an existing slice
- Go maps
- Pointers in Go
- Looping in Go
- Constants in Go
- Working with times
- Measuring the execution time of commands and functions
- Operating with dates

Numeric data types

Go has native support for integers and floating-point numbers, as well as complex numbers. The subsections that follow will tell you more about each numeric type supported by Go.

Integers

Go offers support for four different sizes of **signed** and **unsigned** integers, named `int8`, `int16`, `int32`, `int64`; and `uint8`, `uint16`, `uint32`, and `uint64`, respectively. The number at the end of each type shows the number of bits used for representing each type.

Additionally, `int` and `uint` exist and are the most efficient signed and unsigned integers for your current platform. Therefore, when in doubt, use `int` and `uint`, but have in mind that the size of these types changes depending on the architecture.

The difference between signed and unsigned integers is the following: if an integer has eight bits and no sign, then its values can be from binary `00000000` (`0`) to binary `11111111` (`255`). If it has a sign, then its values can be from `-127` to `127`. This means that you get to have seven binary digits to store your number because the eighth bit is used for keeping the sign of the integer. The same rule applies to the other sizes of unsigned integers.

Floating-point numbers

Go supports only two types of floating-point numbers: `float32` and `float64`. The first one provides about six decimal digits of precision, whereas the second one gives you 15 digits of precision.

Complex numbers

Similar to floating-point numbers, Go offers two complex number types named `complex64` and `complex128`. The first one uses two `float32`: one for the real part and the other for the imaginary part of the complex number, whereas `complex128` uses two `float64`. Complex numbers are expressed in the form of $a + bi$, where a and b are real numbers, and i is a solution of the equation $x^2 = -1$.

All these numeric data types are illustrated in the code of `numbers.go`, which will be presented in three parts.

The first part of `numbers.go` is as follows:

```
package main

import (
    "fmt"
)

func main() {
    c1 := 12 + 1i
    c2 := complex(5, 7)
    fmt.Printf("Type of c1: %T\n", c1)
    fmt.Printf("Type of c2: %T\n", c2)

    var c3 complex64 = complex64(c1 + c2)
    fmt.Println("c3:", c3)
    fmt.Printf("Type of c3: %T\n", c3)

    cZero := c3 - c3
    fmt.Println("cZero:", cZero)
```

In this part, we are working with complex numbers and making some calculations with them. There are two ways to create a complex number: directly, as with `c1` and `c2`, or indirectly by making calculations with existing complex numbers, as with `c3` and `czero`.

If you mistakenly try to create a complex number as `aComplex := 12 + 2 * i`, there will be two possible outcomes because this statement tells Go that you want to perform an addition and a multiplication.



If there is no numeric variable named `i` in the current scope, this statement will create a syntax error and the compilation of your Go code will fail. However, if a numeric variable named `i` is already defined, the calculation will be successful, but you will not get the desired complex number as the result (**bug**).

The second part of `numbers.go` is the following:

```

x := 12
k := 5
fmt.Println(x)
fmt.Printf("Type of x: %T\n", x)

div := x / k
fmt.Println("div", div)

```

In this part, we are working with signed integers. Please note that if you divide two integers, Go thinks that you want the result of the integer division and will calculate and return the **quotient** of the integer division. So, trying to divide 11 by 2 will result in 5 in an integer division and not 5.5. If you are not familiar with mathematics, this might come as a surprise.



When you are converting a floating-point number to an integer, the fraction is discarded by truncating the floating-point number toward zero, which means that some data might get lost in the process.

The last part of `numbers.go` includes the following code:

```

var m, n float64
m = 1.223
fmt.Println("m, n:", m, n)

y := 4 / 2.3
fmt.Println("y:", y)

divFloat := float64(x) / float64(k)
fmt.Println("divFloat", divFloat)
fmt.Printf("Type of divFloat: %T\n", divFloat)
}

```

In this last part of the program, we are working with floating-point numbers. In the presented code, you can see how you can use `float64()` to tell Go to create a floating-point number when dividing two integers. If you just type `divFloat := float64(x) / k`, then you will get the following error message when running the code:

```

$ go run numbers.go
# command-line-arguments
./numbers.go:35:25: invalid operation: float64(x) / k (mismatched types float64 and int)

```

Executing `numbers.go` will generate the following output:

```

Type of c1: complex128
Type of c2: complex128
c3: (17+8i)
Type of c3: complex64
cZero: (0+0i)
12
Type of x: int

```

```
|div 2
|m, n: 1.223 0
|y: 1.7391304347826086
|divFloat 2.4
|Type of divFloat: float64
```

Number literals in Go 2

At the time of writing this, there is a proposal for changes in the way Go handles **number literals**. Number literals are related to the way you can define and use numbers in a programming language. This particular proposal is related to the representation of binary integer literals, octal integer literals, the digit separator, and support for hexadecimal floating-point numbers.

You can find more information about the proposal related to Go 2 and number literals at <https://golang.org/design/19308-number-literals>.



The people who develop Go keep a detailed release dashboard that you can view at <http://dev.golang.org/release>.

Go loops

Every programming language has a way of looping and Go is no exception. Go offers the `for` loop, which allows you to iterate over many kinds of data types.



Go does not offer support for the `while` keyword. However, `for` loops in Go can replace `while` loops.

The `for` loop

The `for` loop allows you to iterate a predefined number of times, for as long as a condition is valid, or according to a value that is calculated at the beginning of the `for` loop. Such values include the size of a slice or an array, or the number of keys on a map. This means that the most common way of accessing all the elements of an array, a slice, or a map is the `for` loop.

The simplest form of a `for` loop follows. A given variable takes a range of predefined values:

```
| for i := 0; i < 100; i++ {  
| }
```

Generally speaking, a `for` loop has three sections: the first one is called the initialization, the second one is called the condition, and the last one is called the afterthought. All sections are optional.

In the previous loop, the values that `i` will take are from 0 to 99. As soon as `i` reaches `100`, the execution of the `for` loop will stop. In this case, `i` is a local and temporary variable, which means that after the termination of the `for` loop, `i` will be garbage collected at some point and disappear. However, if `i` was defined outside the `for` loop, it will keep its value after the termination of the `for` loop. In this case, the value of `i` after the termination of the `for` loop will be `100` as this was the last value of `i` in this particular program at this particular point.

You can completely exit a `for` loop using the `break` keyword. The `break` keyword also allows you to create a `for` loop without an exit condition, such as `i < 100` used in the preceding example, because the exit condition can be included in the code block of the `for` loop. You are also allowed to have multiple exit conditions in a `for` loop. Additionally, you can skip a single iteration of a `for` loop using the `continue` keyword.

The while loop

As discussed earlier, Go does not offer the `while` keyword for writing `while` loops but allows you to use a `for` loop instead of a `while` loop. This section will present two examples where a `for` loop does the job of a `while` loop.

Firstly, let's look at a typical case where you might want to write something like `while(true):`:

```
| for {  
| }  
| }
```

It is the job of the developer to use the `break` keyword to exit this `for` loop.

However, the `for` loop can also emulate a `do...while` loop, which can be found in other programming languages.

As an example, the following Go code is equivalent to a `do...while(anExpression)` loop:

```
| for ok := true; ok; ok = anExpression {  
| }  
| }
```

As soon as the `ok` variable has the `false` value, the `for` loop will terminate.

There is also a `for condition {}` loop in Go where you specify the condition and the `for` loop is executed for as long as the condition is `true`.

The range keyword

Go also offers the `range` keyword, which is used in `for` loops and allows you to write easy-to-understand code for iterating over supported Go data types including Go **channels**. The main advantage of the `range` keyword is that you do not need to know the **cardinality** of a slice, a map, or a channel in order to process its elements one by one. You will see `range` in action later in the chapter.

An example with multiple Go loops

This section will display multiple examples of `for` loops. The name of the file is `loops.go` and will be presented in four parts. The first code segment of `loops.go` is next:

```
package main

import (
    "fmt"
)

func main() {
    for i := 0; i < 100; i++ {
        if i%20 == 0 {
            continue
        }

        if i == 95 {
            break
        }

        fmt.Println(i, " ")
    }
}
```

The preceding code shows a typical `for` loop, as well as the use of the `continue` and `break` keywords.

The next code segment is the following:

```
fmt.Println()
i := 10
for {
    if i < 0 {
        break
    }
    fmt.Println(i, " ")
    i--
}
fmt.Println()
```

The offered code emulates a typical `while` loop. Note the use of the `break` keyword to exit the `for` loop.

The third part of `loops.go` is next:

```
i = 0
anExpression := true
for ok := true; ok; ok = anExpression {
    if i > 10 {
        anExpression = false
    }

    fmt.Println(i, " ")
    i++
}
fmt.Println()
```

In this part, you see the use of a `for` loop that does the job of a `do...while` loop, as discussed earlier on in this chapter. Notice that this `for` loop is difficult to read.

The last part of `loops.go` comes with the next Go code:

```
anArray := [5]int{0, 1, -1, 2, -2}
for i, value := range anArray {
    fmt.Println("index:", i, "value: ", value)
}
```

Applying the `range` keyword to an array variable returns two values: an array index and the value of the element at that index, respectively.

You can use both of them, one of them, or none of them in case you just want to count the elements of the array or perform some other task the same number of times as there are items in an array.

Executing `loops.go` will produce the next output:

```
$ go run loops.go
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 41 42 43 44 45 46 47 +
10 9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8 9 10 11
index: 0 value: 0
index: 1 value: 1
index: 2 value: -1
index: 3 value: 2
index: 4 value: -2
```

Go arrays

Arrays are one of the most popular data structures for two reasons. The first reason is that they are simple and easy to understand, while the second reason is that they are very versatile and can store many different kinds of data.

You can declare an array that stores four integers as follows:

```
| anArray := [4]int{1, 2, 4, -4}
```

The size of the array is stated before its type, which is defined before its elements. You can find the length of an array with the help of the `len()` function: `len(anArray)`.

The index of the first element of any dimension of an array is zero; the index of the second element of any array dimension is one and so on. This means that for an array with one dimension named `a`, the valid indexes are from `0` to `len(a)-1`.

Although you might be familiar with accessing the elements of an array in other programming languages and using a `for` loop and one or more numeric variables, there exist more idiomatic ways to visit all the elements of an array in Go. They involve the use of the `range` keyword and allow you to bypass the use of the `len()` function in the `for` loop. Look at the Go code of `loops.go` for such an example.

Multi-dimensional arrays

Arrays can have more than one dimension. However, using more than three dimensions without a serious reason can make your program difficult to read and might create bugs.



Arrays can store all the types of elements; we are just using integers here because they are easier to understand and type.

The following Go code shows how you can create an array with two dimensions (`twoD`) and another one with three dimensions (`threeD`):

```
twoD := [4][4]int{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12},  
{13, 14, 15, 16}}  
threeD := [2][2][2]int{{{{1, 0}, {-2, 4}}, {{5, -1}, {7, 0}}}}
```

Accessing, assigning, or printing a single element from one of the previous two arrays can be done easily. As an example, the first element of the `twoD` array is `twoD[0][0]` and its value is `1`.

Therefore, accessing all the elements of the `threeD` array with the help of multiple `for` loops can be done as follows:

```
for i := 0; i < len(threeD); i++ {  
    for j := 0; j < len(v); j++ {  
        for k := 0; k < len(m); k++ {  
        }  
    }  
}
```

As you can see, you need as many `for` loops as the dimensions of the array in order to access all of its elements. The same rule applies to slices, which will be presented in the next section. Using `x`, `y`, and `z` as variable names instead of `i`, `j`, and `k` might be a good idea here.

The code of `usingArrays.go`, which will be presented in three parts, presents a complete example of how to deal with arrays in Go.

The first part of the code is the following:

```

package main

import (
    "fmt"
)

func main() {
    anArray := [4]int{1, 2, 4, -4}
    twoD := [4][4]int{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}, {13, 14, 15, 16}}
    threeD := [2][2][2]int{{{1, 0}, {-2, 4}}, {{5, -1}, {7, 0}}}}

```

Here, you define three array variables named `anArray`, `twoD`, and `threeD`, respectively.

The second part of `usingArrays.go` is next:

```

fmt.Println("The length of", anArray, "is", len(anArray))
fmt.Println("The first element of", twoD, "is", twoD[0][0])
fmt.Println("The length of", threeD, "is", len(threeD))

for i := 0; i < len(threeD); i++ {
    v := threeD[i]
    for j := 0; j < len(v); j++ {
        m := v[j]
        for k := 0; k < len(m); k++ {
            fmt.Print(m[k], " ")
        }
    }
    fmt.Println()
}

```

What you get from the first `for` loop is a two-dimensional array (`threeD[i]`), whereas what you get from the second `for` loop is an array with one dimension (`v[j]`). The last `for` loop iterates over the elements of the array with one dimension.

The last code part comes with the next Go code:

```

for _, v := range threeD {
    for _, m := range v {
        for _, s := range m {
            fmt.Print(s, " ")
        }
    }
    fmt.Println()
}

```

The `range` keyword does exactly the same job as the iteration variables used in the `for` loops of the previous code segment but in a more elegant and clear

way. However, if you want to know the number of iterations that are going to be executed in advance, you cannot use the `range` keyword.



The `range` keyword also works with Go maps, which makes it pretty handy and my preferred way of iteration. As you will see in [Chapter 9](#), Concurrency in Go – Goroutines, Channels, and Pipelines, the `range` keyword also works with channels.

Executing `usingArrays.go` will generate the following output:

```
$ go run usingArrays.go
The length of [1 2 4 -4] is 4
The first element of [[1 2 3 4] [5 6 7 8] [9 10 11 12] [13 14 15 16]] is 1
The length of [[[1 0] [-2 4]] [[5 -1] [7 0]]] is 2
1 0 -2 4
5 -1 7 0
1 0 -2 4
5 -1 7 0
```

One of the biggest problems with arrays is out-of-bounds errors, which means trying to access an element that does not exist. This is like trying to access the sixth element of an array with only five elements. The Go compiler considers compiler issues that can be detected as compiler errors because this helps the development workflow. Therefore, the Go compiler can detect out-of-bounds array access errors:

```
./a.go:10: invalid array index -1 (index must be non-negative)
./a.go:10: invalid array index 20 (out of bounds for 2-element array)
```

The shortcomings of Go arrays

Go arrays have many disadvantages that will make you reconsider using them in your Go projects. First of all, once you define an array, you cannot change its size, which means that Go arrays are not dynamic. Putting it simply, if you need to add an element to an existing array that has no space left, you will need to create a bigger array and copy all the elements of the old array to the new one. Also, when you pass an array to a function as a parameter, you actually pass a copy of the array, which means that any changes you make to an array inside a function will be lost after the function exits. Lastly, passing a large array to a function can be pretty slow, mostly because Go has to create a copy of the array. The solution to all these problems is to use Go slices, which will be presented in the next section.



Because of their disadvantages, arrays are rarely used in Go!

Go slices

Go **slices** are very powerful and it would not be an exaggeration to say that slices could totally replace the use of arrays in Go. There are only a few occasions when you will need to use an array instead of a slice. The most obvious one is when you are absolutely sure that you will need to store a fixed number of elements.



Slices are implemented using arrays internally, which means that Go uses an underlying array for each slice.

As slices are **passed by reference** to functions, which means that what is actually passed is the memory address of the slice variable, any modifications you make to a slice inside a function will not get lost after the function exits. Additionally, passing a big slice to a function is significantly faster than passing an array with the same number of elements because Go will not have to make a copy of the slice; it will just pass the memory address of the slice variable.

Performing basic operations on slices

You can create a new **slice literal** as follows:

```
|     aSliceLiteral := []int{1, 2, 3, 4, 5}
```

This means that slice literals are defined just like arrays but without the element count. If you put an element count in a definition, you will get an array instead.

However, there is also the `make()` function, which allows you to create empty slices with the desired **length** and **capacity** based on the parameters passed to `make()`. The capacity parameter can be omitted. In that case, the capacity of the slice will be the same as its length. So, you can define a new empty slice with 20 places that can be automatically expanded when needed as follows:

```
|     integer := make([]int, 20)
```

Please note that Go automatically initializes the elements of an empty slice to the zero value of its type, which means that the value of the initialization depends on the type of the object stored in the slice. It is good to know that Go initializes the elements of every slice created with `make`.

You can access all the elements of a slice in the following way:

```
|     for i := 0; i < len(integer); i++ {  
|         fmt.Println(integer[i])  
|     }
```

If you want to empty an existing slice, the zero value for a slice variable is `nil`:

```
|aSliceLiteral = nil
```

You can add an element to the slice, which will automatically increase its size, using the `append()` function:

```
| integer = append(integer, 12345)
```

You can access the first element of the `integer` slice as `integer[0]`, whereas you can access the last element of the `integer` slice as `integer[len(integer)-1]`.

Lastly, you can access multiple continuous slice elements using the `[:]` notation. The next statement selects the second and the third elements of a slice:

```
| integer[1:3]
```

Additionally, you can use the `[:]` notation for creating a new slice from an existing slice or array:

```
| s2 := integer[1:3]
```

Please note that this process is called **re-slicing** and can cause problems in some cases. Look at the next program:

```
package main

import "fmt"

func main() {

    s1 := make([]int, 5)
    reSlice := s1[1:3]
    fmt.Println(s1)
    fmt.Println(reSlice)

    reSlice[0] = -100
    reSlice[1] = 123456
    fmt.Println(s1)
    fmt.Println(reSlice)

}
```

First, note that in order to select the second and third elements of a slice using the `[:]` notation, you should use `[1:3]`, which means starting with index number 1 and going up to index number 3, without including index number 3.



Given an array, `a1`, you can create a slice, `s1`, that references that array by executing `s1 := a1[:]`.

Executing the previous code, which is saved as `reslice.go`, will create the next output:

```
$ go run reslice.go
[0 0 0 0 0]
[0 0]
[0 -100 123456 0 0]
[-100 123456]
```

So, at the end of the program, the contents of the `s1` slice will be `[0 -100 123456 0 0]` even though we did not change them directly! This means that altering the elements of a re-slice modifies the element of the original slice because they both point to the same underlying array. Putting it simply, the re-slice process does not make a copy of the original slice.

The second problem from re-slicing is that even if you re-slice a slice in order to use a small part of the original slice, the underlying array from the original slice will be kept in memory for as long as the smaller re-slice exists because the original slice is being referenced by the smaller re-slice. Although this is not very important for small slices, it can cause problems when you are reading big files into slices and you want to use only a small part of them.

Slices are expanded automatically

Slices have two main properties: **capacity** and **length**. The tricky thing is that usually these two properties have different values. The length of a slice is the same as the length of an array with the same number of elements and can be found using the `len()` function. The capacity of a slice is the current room that has been allocated for this particular slice and can be found with the `cap()` function. As slices are dynamic in size, if a slice runs out of room, Go automatically doubles its current length to make room for more elements.

Putting it simply, if the length and the capacity of a slice have the same values and you try to add another element to the slice, the capacity of the slice will be doubled whereas its length will be increased by one.

Although this might work well for small slices, adding a single element to a really huge slice might take more memory than expected.

The code of `lenCap.go` illustrates the concepts of capacity and length in more detail and will be presented in three parts. The first part of the program is next:

```
package main

import (
    "fmt"
)

func printSlice(x []int) {
    for _, number := range x {
        fmt.Print(number, " ")
    }
    fmt.Println()
}
```

The `printSlice()` function helps you to print a one-dimensional slice without having to repeat the same Go code all the time.

The second part of `lenCap.go` contains the next Go code:

```

func main() {
    aSlice := []int{-1, 0, 4}
    fmt.Printf("aSlice: ")
    printSlice(aSlice)

    fmt.Printf("Cap: %d, Length: %d\n", cap(aSlice), len(aSlice))
    aSlice = append(aSlice, -100)
    fmt.Printf("aSlice: ")
    printSlice(aSlice)
    fmt.Printf("Cap: %d, Length: %d\n", cap(aSlice), len(aSlice))
}

```

In this part, as well as the next one, you will add some elements to the `aSlice` slice to alter its length and its capacity.

The last portion of Go code is the following:

```

    aSlice = append(aSlice, -2)
    aSlice = append(aSlice, -3)
    aSlice = append(aSlice, -4)
    printSlice(aSlice)
    fmt.Printf("Cap: %d, Length: %d\n", cap(aSlice), len(aSlice))
}

```

The execution of `lenCap.go` will create the next output:

```

$ go run lenCap.go
aSlice: -1 0 4
Cap: 3, Length: 3
aSlice: -1 0 4 -100
Cap: 6, Length: 4
-1 0 4 -100 -2 -3 -4
Cap: 12, Length: 7

```

As you can see, the initial size of the slice was three. As a result, the initial value of its capacity was also three. After adding one element to the slice, its size became four, whereas its capacity became six. After adding three more elements to the slice, its size became seven, whereas its capacity was doubled one more time and became 12.

Byte slices

A **byte slice** is a slice where its type is `byte`. You can create a new byte slice named `s` as follows:

```
| s := make([]byte, 5)
```

Go knows that most slices of bytes are used to store strings and so makes it easy to switch between this type and the `string` type. There is nothing special in the way you can access a byte slice compared to the other types of slices. It is just that byte slices are used in file input and output operations. You will see byte slices in action in [Chapter 8, *Telling a UNIX System What to Do.*](#)

The copy() function

You can create a slice from the elements of an existing array and you can copy an existing slice to another one using the `copy()` function. However, as the use of `copy()` can be very tricky, this subsection will try to clarify its usage with the help of the Go code of `copySlice.go`, which will be presented in four parts.



You should be very careful when using the `copy()` function on slices because the built-in `copy(dst, src)` copies the minimum number of `len(dst)` and `len(src)` elements.

The first part of the program comes with the next Go code:

```
package main

import (
    "fmt"
)

func main() {
    a6 := []int{-10, 1, 2, 3, 4, 5}
    a4 := []int{-1, -2, -3, -4}
    fmt.Println("a6:", a6)
    fmt.Println("a4:", a4)

    copy(a6, a4)
    fmt.Println("a6:", a6)
    fmt.Println("a4:", a4)
    fmt.Println()
```

So, in the preceding code, we define two slices named `a6` and `a4`, we print them, and then we try to copy `a4` to `a6`. As `a6` has more elements than `a4`, all the elements of `a4` will be copied to `a6`. However, as `a4` has only four elements and `a6` has six elements, the last two elements of `a6` will remain the same.

The second part of `copySlice.go` is next:

```
b6 := []int{-10, 1, 2, 3, 4, 5}
b4 := []int{-1, -2, -3, -4}
fmt.Println("b6:", b6)
fmt.Println("b4:", b4)
copy(b4, b6)
```

```
    |     fmt.Println("b6:", b6)
    |     fmt.Println("b4:", b4)
```

In this case, only the first four elements of `b6` will be copied to `b4` because `b4` has only four elements.

The third code segment of `copySlice.go` comes with the next Go code:

```
    |     fmt.Println()
    |     array4 := [4]int{4, -4, 4, -4}
    |     s6 := []int{1, 1, -1, -1, 5, -5}
    |     copy(s6, array4[0:])
    |     fmt.Println("array4:", array4[0:])
    |     fmt.Println("s6:", s6)
    |     fmt.Println()
```

Here you try to copy an array with four elements to a slice with six elements. Please note that the array is converted to a slice with the help of the `[:] notation (array4[0:])`.

The last code portion of `copySlice.go` is the following:

```
    |     array5 := [5]int{5, -5, 5, -5, 5}
    |     s7 := []int{7, 7, -7, -7, 7, -7, 7}
    |     copy(array5[0:], s7)
    |     fmt.Println("array5:", array5)
    |     fmt.Println("s7:", s7)
    | }
```

Here, you can see how you can copy a slice to an array that has space for five elements. As `copy()` only accepts slice arguments, you should also use the `[:] notation to convert the array into a slice.`

If you try to copy an array into a slice or vice versa without using the `[:] notation`, the program will fail to compile, giving one of the next error messages:

```
# command-line-arguments
./a.go:42:6: first argument to copy should be slice; have [5]int
./a.go:43:6: second argument to copy should be slice or string; have [5]int
./a.go:44:6: arguments to copy must be slices; have [5]int, [5]int
```

Executing `copySlice.go` will create the next output:

```
$ go run copySlice.go
a6: [-10 1 2 3 4 5]
```

```
a4: [-1 -2 -3 -4]
a6: [-1 -2 -3 -4 4 5]
a4: [-1 -2 -3 -4]

b6: [-10 1 2 3 4 5]
b4: [-1 -2 -3 -4]
b6: [-10 1 2 3 4 5]
b4: [-10 1 2 3]

array4: [4 -4 4 -4]
s6: [4 -4 4 -4 5 -5]

array5: [7 7 -7 -7 7]
s7: [7 7 -7 -7 7 -7 7]
```

Multi-dimensional slices

Slices can have many dimensions just like arrays. The next statement creates a slice with two dimensions:

| s1 := make([][]int, 4)



If you find yourself using slices with many dimensions all the time, you might need to reconsider your approach and choose a simpler design that does not require multi-dimensional slices.

You will find a code example with a **multi-dimensional slice** in the next section.

Another example with slices

The Go code of the `slices.go` program will hopefully clarify many things about slices and will be presented in five parts.

The first part of the program contains the expected preamble as well as the definition of two slices:

```
package main

import (
    "fmt"
)

func main() {
    aSlice := []int{1, 2, 3, 4, 5}
    fmt.Println(aSlice)
    integers := make([]int, 2)
    fmt.Println(integers)
    integers = nil
    fmt.Println(integers)
```

The second part shows how to use the `[:] notation to create a new slice that references an existing array. Remember that you are not creating a copy of the array, just a reference to it that will be verified in the output of the program:`

```
anArray := [5]int{-1, -2, -3, -4, -5}
refAnArray := anArray[:]

fmt.Println(anArray)
fmt.Println(refAnArray)
anArray[4] = -100
fmt.Println(refAnArray)
```

The third code segment defines a slice with one dimension and another one with two dimensions using the `make()` function:

```
s := make([]byte, 5)
fmt.Println(s)
twoD := make([][]int, 3)
fmt.Println(twoD)
fmt.Println()
```

As slices are automatically initialized by Go, all the elements of the two preceding slices will have the zero value of the slice type, which for integers is `0` and for slices is `nil`. Keep in mind that the elements of a multi-dimensional slice are slices.

In the fourth part of `slices.go` that comes with the next Go code, you will learn how to manually initialize all the elements of a slice with two dimensions:

```
for i := 0; i < len(twoD); i++ {
    for j := 0; j < 2; j++ {
        twoD[i] = append(twoD[i], i*j)
    }
}
```

The preceding Go code shows that in order to expand an existing slice and make it grow, you will need to use the `append()` function and not just reference an index that does not exist! The latter would create a `panic: runtime error: index out of range` error message. Please note that the values of the slice elements have been chosen arbitrarily.

The last part shows how to use the `range` keyword to visit and print all the elements of a slice with two dimensions:

```
for _, x := range twoD {
    for i, y := range x {
        fmt.Println("i:", i, "value:", y)
    }
    fmt.Println()
}
```

If you execute `slices.go`, you will get the next output:

```
$ go run slices.go
[1 2 3 4 5]
[0 0]
[]
[-1 -2 -3 -4 -5]
[-1 -2 -3 -4 -5]
[-1 -2 -3 -4 -100]
[0 0 0 0 0]
[[[] [] []]]

i: 0 value: 0
i: 1 value: 0

i: 0 value: 0
```

```
| i: 1 value: 1  
| i: 0 value: 0  
| i: 1 value: 2
```

It should not come as a surprise to you that the objects of the slice with the two dimensions are initialized to `nil` and therefore are printed as empty; this happens because the zero value for the slice type is `nil`.

Sorting slices using `sort.Slice()`

This subsection will illustrate the use of the `sort.Slice()` function, which was first introduced in Go version 1.8. This means that the presented code, which is saved in `sortSlice.go`, will not run on older Go versions. The program will be presented in three parts. This is the first part:

```
package main

import (
    "fmt"
    "sort"
)

type aStructure struct {
    person string
    height int
    weight int
}
```

Apart from the expected preamble, you can also see the definition of a Go structure for the first time in this book. [Chapter 4, *The Uses of Composite Types*](#), will thoroughly explore Go structures. For now, bear in mind that structures are types with multiple variables of multiple types.

The second part of `sortSlice.go` comes with the next Go code:

```
func main() {
    mySlice := make([]aStructure, 0)
    mySlice = append(mySlice, aStructure{"Mihalis", 180, 90})
    mySlice = append(mySlice, aStructure{"Bill", 134, 45})
    mySlice = append(mySlice, aStructure{"Marietta", 155, 45})
    mySlice = append(mySlice, aStructure{"Epifanios", 144, 50})
    mySlice = append(mySlice, aStructure{"Athina", 134, 40})

    fmt.Println("0:", mySlice)
```

Here, you create a new slice named `mySlice` with elements from the `aStructure` structure created earlier.

The final part of the program is the following:

```
    sort.Slice(mySlice, func(i, j int) bool {
        return mySlice[i].height < mySlice[j].height
    })
    fmt.Println("<:", mySlice)
    sort.Slice(mySlice, func(i, j int) bool {
        return mySlice[i].height > mySlice[j].height
    })
```

```
|     })
|     fmt.Println(">:", mySlice)
| }
```

Here, you sort `mySlice` two times using `sort.Slice()` and two anonymous functions. This happens one anonymous function at a time, using the `height` field of `aStructure`.



Please note that `sort.Slice()` changes the order of the elements in the slice according to the sorting function.

Executing `sortslice.go` will create the next output:

```
$ go run sortSlice.go
0: [{Mihalis 180 90} {Bill 134 45} {Marietta 155 45} {Epifanios 144 50} {Athina 134 40}]
<: [{Bill 134 45} {Athina 134 40} {Epifanios 144 50} {Marietta 155 45} {Mihalis 180 90}]
>: [{Mihalis 180 90} {Marietta 155 45} {Epifanios 144 50} {Bill 134 45} {Athina 134 40}]
```

If you try to execute `sortslice.go` on a UNIX machine with a Go version older than 1.8, you will get the next error message:

```
$ go version
o version go1.3.3 linux/amd64
$ go run sortSlice.go
# command-line-arguments
./sortSlice.go:24: undefined: sort.Slice
./sortSlice.go:28: undefined: sort.Slice
```

Appending an array to a slice

In this subsection, you will learn how to append an existing array to an existing slice using the technique found in `appendArrayToSlice.go`. The program will be presented in two parts. The first part is the following:

```
package main

import (
    "fmt"
)

func main() {
    s := []int{1, 2, 3}
    a := [3]int{4, 5, 6}
```

So far, we have just created and initialized a slice named `s` and an array named `a`.

The second part of `appendArrayToSlice.go` is as follows:

```
    ref := a[:]
    fmt.Println("Existing array:\t", ref)
    t := append(s, ref...)
    fmt.Println("New slice:\t", t)
    s = append(s, ref...)
    fmt.Println("Existing slice:\t", s)
    s = append(s, s...)
    fmt.Println("s+s:\t\t", s)
}
```

Two important things are happening here. The first is that we create a new slice named `t` that contains the elements of `a + s`, we append the array named `a` to the slice named `s`, and we store the result to the `s` slice. So, you have a choice on whether to store the new slice in an existing slice variable or not. This mainly depends on what you want to accomplish.

The second important thing is that you have to create a reference to the existing array (`ref := a[:]`) for this to work. Please notice the way the `ref` variable is used in the two `append()` calls: the three dots (...) are exploding the array into arguments that are appended to the existing slice.

The last two statements of the program show how you can copy a slice to the end of itself. The three dots (...) are still required.

Executing `appendArrayToSlice.go` will generate the following output:

```
$ go run appendArrayToSlice.go
Existing array:      [4 5 6]
New slice:          [1 2 3 4 5 6]
Existing slice:    [1 2 3 4 5 6]
s+s:                [1 2 3 4 5 6 1 2 3 4 5 6]
```

Go maps

A Go **map** is equivalent to the well-known **hash table** found in many other programming languages. The main advantage of maps is that they can use any data type as their index, which in this case is called a **map key** or just a **key**. Although Go maps do not exclude any data types from being used as keys, for a data type to be used as a key it must be **comparable**, which means that the Go compiler must be able to differentiate one key from another or, putting it simply, that the keys of a map must support the `==` operator.

The good news is that almost all data types are comparable. However, as you can imagine, using the `bool` data type as the key to a map will definitely limit your options. Additionally, using floating-point numbers as keys might present problems caused by the precision used for different machines and operating systems.



As mentioned, a Go map is a reference to a hash table. The good thing is that Go hides the implementation of the hash table and therefore its complexity. You will learn more about implementing a hash table on your own in Go in chapter 5, How to Enhance Go Code with Data Structures.

You can create a new empty map with `string` keys and `int` values with the help of the `make()` function:

```
| iMap = make(map[string]int)
```

Alternatively, you can use the next **map literal** in order to create a new map that will be populated with data:

```
| anotherMap := map[string]int {  
| "k1": 12  
| "k2": 13  
| }
```

You can access the two objects of `anotherMap` as `anotherMap["k1"]` and `anotherMap["k2"]`. You can delete an object of a map using the `delete()` function:

```
| delete(anotherMap, "k1")
```

You can iterate over all the elements of a map using the next technique:

```
|   for key, value := range iMap {  
|     fmt.Println(key, value)  
| }
```

The Go code of `usingMaps.go` will illustrate the use of maps in more detail. The program will be presented in three parts. The first part comes with the following Go code:

```
package main  
  
import (  
    "fmt"  
)  
  
func main() {  
  
    iMap := make(map[string]int)  
    iMap["k1"] = 12  
    iMap["k2"] = 13  
    fmt.Println("iMap:", iMap)  
  
    anotherMap := map[string]int{  
        "k1": 12,  
        "k2": 13,  
    }  
}
```

The second part of `usingMaps.go` contains the next code:

```
fmt.Println("anotherMap:", anotherMap)  
delete(anotherMap, "k1")  
delete(anotherMap, "k1")  
delete(anotherMap, "k1")  
fmt.Println("anotherMap:", anotherMap)  
  
, ok := iMap["doesNotExist"]  
if ok {  
    fmt.Println("Exists!")  
} else {  
    fmt.Println("Does NOT exist")  
}
```

Here you see a technique that allows you to determine whether a given key is in the map or not. This is a vital technique because without it you would not know whether a given map has the required information or not.

The bad thing is that if you try to get the value of a map key that does not exist in the map, you will end up getting zero, which gives you no way of determining whether the





result was zero because the key you requested was not there or because the element with the corresponding key actually had a zero value. This is why we have `_`, `ok` in maps.

Additionally, you can see the `delete()` function in action. Calling the same `delete()` statement multiple times does not make any difference and does not generate any warning messages.

The last part of the program is next:

```
|     for key, value := range iMap {  
|         fmt.Println(key, value)  
|     }  
| }
```

Here, you see the use of the `range` keyword on a map, which is pretty elegant and handy.

If you execute `usingMaps.go`, you will get the next output:

```
$ go run usingMaps.go  
iMap: map[k1:12 k2:13]  
anotherMap: map[k1:12 k2:13]  
anotherMap: map[k2:13]  
Does NOT exist  
k1 12  
k2 13
```



Please note that you cannot and should not make any assumptions about the order the map pairs are going to be displayed on your screen because that order is totally random.

Storing to a nil map

The following Go code will work:

```
aMap := map[string]int{}  
aMap["test"] = 1
```

However, the next Go code will not work because you have assigned the `nil` value to the map you are trying to use:

```
aMap := map[string]int{}  
// var aMap map[string]int  
aMap = nil  
fmt.Println(aMap)  
aMap["test"] = 1
```

Saving the preceding code to `failMap.go` and trying to compile it will generate the next error message:

```
$ go run failMap.go  
map[]  
panic: assignment to entry in nil map  
...
```

This means that trying to insert data to a `nil` map will fail. However, looking up, deleting, finding the length, and using `range` loops on `nil` maps will not crash your code.

When you should use a map

Maps are more versatile than both slices and arrays but this flexibility comes at a cost: the extra processing power required for the implementation of a Go map. However, built-in Go structures are very fast, so do not hesitate to use a Go map when you need to. What you should remember is that Go maps are very convenient and can store many different kinds of data, while being both easy to understand and fast to work with.

Go constants

Go supports **constants**, which are variables that cannot change their values. Constants in Go are defined with the help of the `const` keyword.



*Generally speaking, constants are **global variables**, so you might rethink your approach if you find yourself defining too many constant variables with a local scope.*

The main benefit you get from using constants in your programs is the guarantee that their value will not change during program execution. Strictly speaking, the value of a constant variable is defined at compile time not at run time.

Behind the scenes, Go uses Boolean, string, or number as the type for storing a constant variable because this gives Go more flexibility when dealing with constants.

You can define a new constant as follows:

```
| const HEIGHT = 200
```

Please note that in Go we do not use ALL CAPS for constants; this is just a personal preference of mine.

Additionally, if you want to declare many constants at once, mainly because they are related to each other, you can use the next notation:

```
const (
    C1 = "C1C1C1"
    C2 = "C2C2C2"
    C3 = "C3C3C3"
)
```

Please note that the Go compiler considers the results of all operations applied to constants as constants. However, if a constant is part of a larger expression, this will not be the case.

Now, for something completely different, let's look at the following three variable declarations that mean exactly the same thing in Go:

```
| s1 := "My String"  
| var s2 = "My String"  
| var s3 string = "My String"
```

However, as none of them have the `const` keyword in their declaration, none of them are constants. This does not mean that you cannot define two constants in a similar way:

```
| const s1 = "My String"  
| const s2 string = "My String"
```

Although both `s1` and `s2` are constants, `s2` comes with a type declaration (`string`), which makes its declaration more restrictive than the declaration of `s1`. This is because a typed Go constant must follow all the strict rules of a typed Go variable. On the other hand, a constant without a type doesn't need to follow all the strict rules of a typed variable, which means that it can be mixed with expressions more liberally. Additionally, even constants without a type have a default type that is used when, and only when, no other type information is available. The main reason for this behavior is that as you do not know how a constant is going to be used, you do not desire to use all the available Go rules.

A simple example is the definition of a numeric constant such as `const value = 123`. As you might use the `value` constant in many expressions, declaring a type would make your job much more difficult. Look at the next Go code:

```
| const s1 = 123  
| const s2 float64 = 123  
  
| var v1 float32 = s1 * 12  
| var v2 float32 = s2 * 12
```

Although the compiler will not have a problem with the definition of `v1`, the code used for the definition of `v2` will not compile because `s2` and `v2` have different types:

```
| $ go run a.go  
| # command-line-arguments  
| ./a.go:12:6: cannot use s2 * 12 (type float64) as type float32 in assignment
```

As general advice, if you are using lots of constants in your programs, it might be a good idea to gather all of them in a Go package or a Go structure.

The constant generator iota

The **constant generator iota** is used for declaring a sequence of related values that use incrementing numbers without the need to explicitly type each one of them.

Most of the concepts related to the `const` keyword, including the constant generator iota, will be illustrated in the `constants.go` file, which will be presented in four parts.

The first code segment of `constants.go` is next:

```
package main

import (
    "fmt"
)

type Digit int
type Power2 int

const PI = 3.1415926

const (
    C1 = "C1C1C1"
    C2 = "C2C2C2"
    C3 = "C3C3C3"
)
```

In this part, we declare two new types named `Digit` and `Power2`, and four new constants named `PI`, `C1`, `C2`, and `C3`.



*A Go type is a way of defining a new **named type** that uses the same underlying type as an existing type. This is mainly used for differentiating between different types that might use the same kind of data.*

The second part of `constants.go` comes with the next Go code:

```
func main() {
    const s1 = 123
    var v1 float32 = s1 * 12
    fmt.Println(v1)
    fmt.Println(PI)
```

In this part, you define another constant (`s1`) that is used in an expression (`v1`).

The third part of the program is as follows:

```
const (
    Zero Digit = iota
    One
    Two
    Three
    Four
)
fmt.Println(One)
fmt.Println(Two)
```

Here you see the definition of a **constant generator iota** based on `Digit`, which is equivalent to the next declaration of four constants:

```
const (
    Zero = 0
    One = 1
    Two = 2
    Three = 3
    Four = 4
)
```

The last portion of `constants.go` is the following:

```
const (
    p2_0 Power2 = 1 << iota
    p2_2
    p2_4
    p2_6
)
fmt.Println("2^0:", p2_0)
fmt.Println("2^2:", p2_2)
fmt.Println("2^4:", p2_4)
fmt.Println("2^6:", p2_6)
}
```

There is another constant generator iota here that is a little different than the previous one. Firstly, you can see the use of the underscore character in a `const` block with a constant generator iota, which allows you to skip unwanted values. Secondly, the value of `iota` always increments and can be used in expressions, which is what occurred in this case.

Now let us see what really happens inside the `const` block. For `p2_0`, `iota` has the value of `0` and `p2_0` is defined as `1`. For `p2_2`, `iota` has the value of `2` and `p2_2` is defined as the result of the expression `1 << 2`, which is `00000100` in binary representation. The decimal value of `00000100` is `4`, which is the result and the value of `p2_2`. Analogously, the value of `p2_4` is `16` and the value of `p2_6` is `32`.

As you can see, the use of `iota` can save your time when it fits your needs.

Executing `constants.go` will generate the next output:

```
$ go run constants.go
1476
3.1415926
1
2
2^0: 1
2^2: 4
2^4: 16
2^6: 64
```

Go pointers

Go supports **pointers**, which are memory addresses that offer improved speed in exchange for difficult-to-debug code and nasty bugs. Ask any C programmer you know to learn more about this.

You have already seen pointers in action in [Chapter 2, Understanding Go Internals](#), when we talked about unsafe code and the `unsafe` package, as well as the Go garbage collector, but this section will try to shed more light on this difficult and tricky subject. Additionally, native Go pointers are safe provided that you know what you are doing.

When working with pointers, you need `*` to get the value of a pointer, which is called **dereferencing the pointer**, and `&` to get the memory address of a non-pointer variable.



Generally speaking, amateur developers should use pointers only when the libraries they use require it because pointers can be the cause of horrible and difficult-to-discover bugs when used carelessly.

You can make a function accept a pointer parameter as follows:

```
| func getPointer(n *int) {  
| }
```

Similarly, a function can return a pointer as follows:

```
| func returnPointer(n int) *int {  
| }
```

The use of safe Go pointers is illustrated in `pointers.go`, which will be presented in four parts. The first code segment of `pointers.go` is next:

```
package main  
  
import (  
    "fmt"  
)  
  
func getPointer(n *int) {  
    *n = *n * *n
```

```

    }

func returnPointer(n int) *int {
    v := n * n
    return &v
}

```

The good thing with `getPointer()` is that it allows you to update the variable passed to it without the need to return anything to the caller function. This happens because the pointer passed as a parameter contains the memory address of the variable.

On the other hand, `returnPointer()` gets an integer parameter and returns a pointer to an integer, which is denoted by `return &v`. Although this might not look that useful, you will really appreciate this capability in [Chapter 4, *The Uses of Composite Types*](#), when we talk about pointers to Go structures as well as in later chapters where more complex data structures will be involved.

Both the `getPointer()` and `returnPointer()` functions find the square of an integer. However, they use a totally different approach as `getPointer()` stores the result to the provided parameter whereas `returnPointer()` returns the result and requires a different variable for storing it.

The second part of the program contains the next Go code:

```

func main() {
    i := -10
    j := 25

    pI := &i
    pJ := &j

    fmt.Println("pI memory:", pI)
    fmt.Println("pJ memory:", pJ)
    fmt.Println("pI value:", *pI)
    fmt.Println("pJ value:", *pJ)
}

```

Both `i` and `j` are normal integer variables. However, `pI` and `pJ` are both pointers pointing to `i` and `j`, respectively. `pI` is the memory address of the pointer, whereas `*pI` is the value stored to that memory address.

The third part of `pointers.go` is the following:

```
| *pI = 123456  
| *pI--  
| fmt.Println("i:", i)
```

Here, you can see how you can change the `i` variable through the `pI` pointer that points to `i` in two different ways: firstly, by directly assigning a new value to it and secondly, by using the `--` operator.

The last code portion of `pointers.go` comes with the next Go code:

```
| getPointer(pJ)  
| fmt.Println("j:", j)  
| k := returnPointer(12)  
| fmt.Println(*k)  
| fmt.Println(k)  
| }
```

Here you call the `getPointer()` function using `pJ` as its parameter. As we talked about before, any changes made to the variable that `pJ` points to inside `getPointer()` will have an effect on the value of the `j` variable, which will be verified by the output of the `fmt.Println("j:", j)` statement. The call to `returnPointer()` returns a pointer that is assigned to the `k` pointer variable.

Running `pointers.go` will create the next output:

```
$ go run pointers.go  
pI memory: 0xc0000160b0  
pJ memory: 0xc0000160b8  
pI value: -10  
pJ value: 25  
i: 123455  
j: 625  
144  
0xc0000160f0
```

I recognize that you might have trouble understanding the Go code of `pointers.go` because we have not talked about functions and function definitions yet. Feel free to look at [Chapter 6, *What You Might Not Know About Go Packages and Go Functions*](#), where things related to functions are explained in more detail.



Please note that strings in Go are value types not pointers as is the case in C.

Why use pointers?

There are two main reasons for using pointers in your programs:

- Pointers allow you to share data, especially between Go functions.
- Pointers can be extremely useful when you want to differentiate between a zero value and a value that is not set.

Times and dates

In this section, you are going to learn how to parse time and date strings in Go, how to convert between different time and date formats, and how to print times and dates in the format you desire. Although this task might look insignificant at first, it can be truly critical when you want to synchronize multiple tasks or when your application needs to read the date from one or more text files, or directly from the user.

The `time` package is the star of working with times and dates in Go; you will see some of its functions in action in this section.

Before learning how to parse a string and convert it into a time or a date, you will see a simple program named `usingTime.go` that will introduce you to the `time` package. The program will be presented in three parts. The first part is the following:

```
package main

import (
    "fmt"
    "time"
)
```

The second code segment of `usingTime.go` comes with the next Go code:

```
func main() {
    fmt.Println("Epoch time:", time.Now().Unix())
    t := time.Now()
    fmt.Println(t, t.Format(time.RFC3339))
    fmt.Println(t.Weekday(), t.Day(), t.Month(), t.Year())

    time.Sleep(time.Second)
    t1 := time.Now()
    fmt.Println("Time difference:", t1.Sub(t))
```

The `time.Now().Unix()` function returns the **UNIX epoch time**, which is the number of seconds that have elapsed since 00:00:00 UTC, 1 January, 1970. The `Format()` function allows you to convert a `time` variable to another format; in this case, the `RFC3339` format.

You will see the `time.Sleep()` function many times in this book as a naive way of emulating the delay from the execution of a true function. The `time.Second` constant allows you to use a one-second duration in Go. If you want to define a duration of 10 seconds, you will need to multiply `time.Second` by 10. Other similar constants include `time.Nanosecond`, `time.Microsecond`, `time.Millisecond`, `time.Minute`, and `time.Hour`. So, the smallest amount of time that can be defined with the `time` package is the nanosecond. Lastly, the `time.Sub()` function allows you to find the time difference between two times.

The last part of the program is next:

```
formatT := t.Format("01 January 2006")
fmt.Println(formatT)
loc, _ := time.LoadLocation("Europe/Paris")
londonTime := t.In(loc)
fmt.Println("Paris:", londonTime)
```

Here you define a new date format using `time.Format()` in order to use it for printing out a `time` variable.

Executing `usingTime.go` will generate the next output:

```
$ go run usingTime.go
Epoch time: 1548753515
2019-01-29 11:18:35.01478 +0200 EET m=+0.000339641 2019-01-29T11:18:35+02:00
Tuesday 29 January 2019
Time difference: 1.000374985s
01 January 2019
Paris: 2019-01-29 10:18:35.01478 +0100 CET
```

Now that you know the basics of the `time` package, it is time to dig deeper into its functionality, starting with working with times.

Working with times

When you have a `time` variable, it is easy to convert it into anything that is related to time or date. However, the main problem is when you have a string and you want to see whether this is a valid time or not. The function used for parsing time and date strings is called `time.Parse()` and it accepts two parameters. The first one denotes the expected format of the string that you are going to parse, whereas the second parameter is the actual string that is going to be parsed. The first parameter is composed of elements from a list of Go constants related to date and time parsing.



The list of constants that can be used for creating your own parse format can be found at <https://golang.org/src/time/format.go>. Go does not define the format of a date or a time in a form like `DDYYYYMM` or `%D %Y %M` as other programming languages do but uses its own approach. Although you might find this approach strange at first, you will certainly appreciate it as it prevents the developer from making silly mistakes.

The Go constants for working with times are `_15` for parsing the hour, `_04` for parsing the minutes, and `_05` for parsing the seconds. You can easily guess that all these numeric values must be unique. You can use `_PM` for parsing the `PM` string in uppercase and `_pm` for parsing the lowercase version.

Please note that you are not obligated to use every available Go constant. The main task of the developer is putting the various Go constants in the desired order to match the kind of strings that the program will have to process. You can consider the final version of the string that is passed as the first parameter to the `time.Parse()` function as a **regular expression**.

Parsing times

This section will tell you how to parse a string, which is given as a command-line argument to the `parseTime.go` utility in order to convert it into a `time` variable. However, this is not always possible because the given string might not be in the correct format or might contain invalid characters. The `parseTime.go` utility will be presented in three parts.

The first code segment of `parseTime.go` is next:

```
package main

import (
    "fmt"
    "os"
    "path/filepath"
    "time"
)
```

The second part comes with the next code:

```
func main() {
    var myTime string
    if len(os.Args) != 2 {
        fmt.Printf("usage: %s string\n", filepath.Base(os.Args[0]))
        os.Exit(1)
    }
    myTime = os.Args[1]
```

The last portion of `parseTime.go`, which is where the magic happens, is the following:

```
    d, err := time.Parse("15:04", myTime)
    if err == nil {
        fmt.Println("Full:", d)
        fmt.Println("Time:", d.Hour(), d.Minute())
    } else {
        fmt.Println(err)
    }
}
```

In order to parse an hour and minute string, you will need to use `"15:04"`. The value of the `err` variable tells you whether the parsing was successful or not.

Executing `parseTime.go` will create the next kind of output:

```
$ go run parseTime.go
usage: parseTime string
exit status 1
$ go run parseTime.go 12:10
Full: 0000-01-01 12:10:00 +0000 UTC
Time: 12 10
```

As you can see here, Go prints a full date and time string because this is what is stored in a `time` variable. If you are only interested in the time and not in the date, you should print the parts of a `time` variable that you want.

If you use a wrong Go constant like `22:04` when trying to parse a string and convert it into a time, you will get the next error message:

```
$ go run parseTime.go 12:10
parsing time "12:10" as "22:04": cannot parse ":10" as "2"
```

However, if you use a Go constant like `11` that is used for parsing months when the month is given as a number, the error message will be slightly different:

```
$ go run parseTime.go 12:10
parsing time "12:10": month out of range
```

Working with dates

In this subsection, you will learn how to parse strings that denote dates in Go, which still requires the use of the `time.Parse()` function.

The Go constants for working with dates are `Jan` for parsing the three-letter abbreviation used for describing a month, `2006` for parsing the year, and `02` for parsing the day of the month. If you use `January` instead of `Jan`, you will get the long name of the month instead of its three-letter abbreviation, which makes perfect sense.

Additionally, you can use `Monday` for parsing strings that contain a long weekday string and `Mon` for the abbreviated version of the weekday.

Parsing dates

The name of the Go program that will be developed in this subsection is `parseDate.go` and it will be presented in two parts.

The first part of `parseDate.go` is next:

```
package main

import (
    "fmt"
    "os"
    "path/filepath"
    "time"
)

func main() {

    var myDate string
    if len(os.Args) != 2 {
        fmt.Printf("usage: %s string\n",
filepath.Base(os.Args[0]))
        return
    }

    myDate = os.Args[1]
```

The second part of `parseDate.go` contains the following Go code:

```
    d, err := time.Parse("02 January 2006", myDate)
    if err == nil {
        fmt.Println("Full:", d)
        fmt.Println("Time:", d.Day(), d.Month(), d.Year())
    } else {
        fmt.Println(err)
    }
}
```

If there is a character such as - between the name of the month and the year, you can use "02 January-2006" instead of "02 January 2006" as the first parameter to `time.Parse()`.

Executing `parseDate.go` will generate the next output:

```
$ go run parseDate.go
usage: parseDate string
$ go run parseDate.go "20 July 2000"
```

```
| Full: 2000-07-20 00:00:00 +0000 UTC
| Time: 20 July 2000
```

As `parseDate.go` does not expect data about the time, the `00:00:00 +0000 UTC` string is automatically added at the end of the full date and time string.

Changing date and time formats

In this section, you will learn how to change the format of a string that contains both a date and a time. A very common place for finding such strings is the log files of web servers such as **Apache** and **Nginx**. As we do not know how to read a text file line by line, the text will be hard coded in the program; however, this fact does not change the functionality of the program.

The Go code of `timeDate.go` will be presented in four parts. The first part is the expected preamble:

```
package main
import (
    "fmt"
    "regexp"
    "time"
)
```

You need the `regexp` standard Go package for supporting regular expressions.

The second code portion of `timeDate.go` is the following:

```
func main() {
    logs := []string("127.0.0.1 - - [16/Nov/2017:10:49:46 +0200]
325504", "127.0.0.1 - - [16/Nov/2017:10:16:41 +0200] \"GET /CVEN
HTTP/1.1\\" 200 12531 \"-\\" \"Mozilla/5.0 AppleWebKit/537.36\", "127.0.0.1 200 9412 - - [12/Nov/2017:06:26:05 +0200]
\"GET \"/http://www.mtsoukalos.eu/taxonomy/term/47\\" 1507",
"[12/Nov/2017:16:27:21 +0300]",
"[12/Nov/2017:20:88:21 +0200]",
"[12/Nov/2017:20:21 +0200]",
}
```

As you cannot be sure about your data and its format, the sample data used for this program tries to cover many different cases, including incomplete data like `[12/Nov/2017:20:21 +0200]` where there are no seconds in the time part, and erroneous data such as `[12/Nov/2017:20:88:21 +0200]` where the value of the minutes is 88.

The third part of `timeDate.go` contains the next Go code:

```
for _, logEntry := range logs {
    r := regexp.MustCompile(`^[\d\d/\w+/\d\d\d:\d\d:\d\d\d.*]`)
    if r.MatchString(logEntry) {
        match := r.FindStringSubmatch(logEntry)
```

The main benefit you get from such a difficult-to-read regular expression in this particular program is that it allows your code to find out whether you have a date and time string somewhere in your line or not. After you get that string, you will feed `time.Parse()` with it and let `time.Parse()` do the rest of the job.

The last part of the program comes with the next Go code:

```
    dt, err := time.Parse("02/Jan/2006:15:04:05 -0700", match[1])
    if err == nil {
        newFormat := dt.Format(time.RFC850)
        fmt.Println(newFormat)
    } else {
        fmt.Println("Not a valid date time format!")
    }
} else {
    fmt.Println("Not a match!")
}
}
```

Once you find a string that matches the regular expression, you parse it using `time.Parse()` to make sure that it is a valid date and time string. If yes, `timeDate.go` will print the date and time according to the **RFC850 format**.

If you execute `timeDate.go`, you will get the next kind of output:

```
$ go run timeDate.go
Thursday, 16-Nov-17 10:49:46 EET
Thursday, 16-Nov-17 10:16:41 EET
Sunday, 12-Nov-17 06:26:05 EET
Sunday, 12-Nov-17 16:27:21 +0300
Not a valid date time format!
Not a match!
```

Measuring execution time

In this section, you are going to learn how to measure the execution time of one or more commands in Go. The same technique can be applied for measuring the execution time of a function or a group of functions. The name of the Go program is `execTime.go` and it will be presented in three parts.



This is an easy-to-implement technique that is both very powerful and handy. Do not underestimate the simplicity of Go.

The first part of `execTime.go` is as follows:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    start := time.Now()
    time.Sleep(time.Second)
    duration := time.Since(start)
    fmt.Println("It took time.Sleep(1)", duration, "to finish.")
```

You will need the functionality offered by the `time` package in order to measure the execution time of a command. All the job is done by the `time.Since()` function that accepts a single argument, which should be a time in the past. In this case, we are measuring the time it took Go to execute a `time.Sleep(time.Second)` call as this is the only statement between `time.Now()` and `time.Since()`.

The second part of `execTime.go` has the following code:

```
start = time.Now()
time.Sleep(2 * time.Second)
duration = time.Since(start)
fmt.Println("It took time.Sleep(2)", duration, "to finish.")
```

This time we are measuring the time it took Go to execute a `time.Sleep(2 * time.Second)` call. This can be very useful for finding out how accurate the

`time.Sleep()` function is, which mainly has to do with how accurate the internal Go clock is.

The last part of `execTime.go` is as follows:

```
start = time.Now()
for i := 0; i < 200000000; i++ {
    _ = i
}
duration = time.Since(start)
fmt.Println("It took the for loop", duration, "to finish.")

sum := 0
start = time.Now()
for i := 0; i < 200000000; i++ {
    sum += i
}
duration = time.Since(start)
fmt.Println("It took the for loop", duration, "to finish.")
```

In the last part, we are measuring the speed of two `for` loops. The first one does nothing, whereas the second one does some calculations. As you will see in the output of the program, the second `for` loop is faster than the first one.

Executing `execTime.go` will generate the following kind of output:

```
$ go run execTime.go
It took time.Sleep(1) 1.000768881s to finish.
It took time.Sleep(2) 2.00062487s to finish.
It took the for loop 50.497931ms to finish.
It took the for loop 47.70599ms to finish.
```

Measuring the operation of the Go garbage collector

Now we can rewrite `sliceGC.go`, `mapNoStar.go`, `mapStar.go`, and `mapSplit.go` from the previous chapter and get more accurate results from them without the need to use the `time(1)` UNIX command-line utility. Actually, the only thing that needs to be done in each one of these files is embed the call to `runtime.GC()` between `time.Now()` and `time.Since()`, and print the results. The updated versions of `sliceGC.go`, `mapNoStar.go`, `mapStar.go`, and `mapSplit.go` will be called `sliceGCTime.go`, `mapNoStarTime.go`, `mapStarTime.go`, and `mapSplitTime.go`, respectively.

Executing the updated versions will generate the following output:

```
$ go run sliceGCTime.go
It took GC() 281.563µs to finish
$ go run mapNoStarTime.go
It took GC() 9.483966ms to finish
$ go run mapStarTime.go
It took GC() 651.704424ms to finish
$ go run mapSplitTime.go
It took GC() 12.743889ms to finish
```

These results are far more accurate than before because they only show the time it took `runtime.GC()` to execute without including the time it took the program to populate the slice or the map used for storing the values. Nevertheless, the results still verify the findings about how slow it is for the Go garbage collector to deal with map variables with lots of data.

Web links and exercises

- Write a constant generator iota for the days of the week.
- Write a Go program that converts an existing array into a map.
- Visit the documentation page of the `time` package, which can be found at <https://golang.org/pkg/time/>.
- Can you write a constant generator iota for the powers of the number four?
- You can also visit the GitHub page where Go 2 and the changes to number literals are being discussed, which will help you to understand how changes in Go are happening: <https://github.com/golang/proposal/blob/master/design/19308-number-literals.md>.
- Write your own version of `parseDate.go`.
- Write your own version of `parseTime.go`. Do not forget to test your program.
- Can you create a version of `timeDate.go` that can process two date and time formats?

Summary

In this chapter, you learned about many interesting Go topics, including numeric data types, maps, arrays, and slices, as well as Go pointers, Go constants and loops, and how Go allows you to work with dates and times. You should understand by now why slices are superior to arrays.

The next chapter will be about building and using composite types in Go, which mainly includes types that are created with the `struct` keyword and are called **structures**. After that, we will talk about `string` variables and **tuples**.

Additionally, the next chapter will talk about **regular expressions** and **pattern matching**, which are tricky subjects not only in Go but also in every other programming language. However, when used properly and carefully, regular expressions and pattern matching can make the life of a developer so much easier that it is totally worth learning more about them.

JSON is a very popular text format, so the next chapter will also discuss how you can create, import, and export JSON data in Go.

Lastly, you will learn about the `switch` keyword and the `strings` package, which allows you to manipulate **UTF-8** strings.

The Uses of Composite Types

In the previous chapter, we talked about many core Go topics, including numeric data types, arrays, slices, maps, pointers, constants, the `for` loop, the `range` keyword, and how to work with times and dates.

This chapter will explore more advanced Go features, such as **tuples** and **strings**, the `strings` standard Go package, and the `switch` statement, but, most importantly, it will look at **structures**, which are used extensively in Go.

The chapter will also show you how to work with **JavaScript Object Notation (JSON)** and **Extensible Markup Language (XML)** text files, how to implement a simple **key-value store**, how to define **regular expressions**, and how to perform **pattern matching** in Go.

The following topics will be covered:

- Go **structures** and the `struct` keyword
- Go **tuples**
- Go strings, runes, and string literals
- Working with the **JSON** text format
- Working with the **XML** text format
- Regular expressions in Go
- Pattern matching in Go
- The `switch` statement
- The functionality that the `strings` package offers
- Calculating **Pi** with high accuracy
- Developing a key-value store

About composite types

Although standard Go types are pretty handy, fast, and flexible, they most likely cannot cover every type of data you want to support in your Go code. Go solves this problem by supporting **structures**, which are custom types defined by the developer. Additionally, Go has its own way of supporting **tuples**, which mainly allows functions to return multiple values without the need to group them in structures as is the case in C.

Structures

Although arrays, slices, and maps are all very useful, they cannot group and hold multiple values in the same place. When you need to group various types of variables and create a new handy type, you can use a structure. The various elements of a structure are called the **fields of the structure** or just fields.

I will start this section by explaining a simple structure that was first defined in the `sortSlice.go` source file of the previous chapter:

```
| type aStructure struct {  
|     person string  
|     height int  
|     weight int  
| }
```

For reasons that will become evident in [Chapter 6, "What You Might Not Know About Go Packages and Go Functions](#), the fields of a structure usually begin with an uppercase letter – this mainly depends on what you want to do with the fields. This structure has three fields named `person`, `height`, and `weight`, respectively. You can create a new variable of the `aStructure` type as follows:

```
| var s1 aStructure
```

Additionally, you can access a specific field of a structure by its name. So, in order to get the value of the `person` field of the `s1` variable, you should type `s1.person`.

A **structure literal** can be defined as follows:

```
| p1 := aStructure{"fmt", 12, -2}
```

However, since remembering the order of the fields of a structure can be pretty hard, Go allows you to use another form for defining a structure literal:

```
| p1 := aStructure{weight: 12, height: -2}
```

In this case, you do not need to define an initial value for every field of the structure.

Now that you know the basics of structures, it is time to show you a more practical example. It is named `structures.go` and will be presented in four parts.

The first part of `structures.go` contains the following code:

```
package main

import (
    "fmt"
)
```



Structures, in particular, and Go types, in general, are usually defined outside the `main()` function in order that they have a global scope and are available to the entire Go package, unless you want to clarify that a type is only useful within the current scope and is not expected to be used elsewhere.

The second code segment from `structures.go` is shown in the following Go code:

```
func main() {

    type XYZ struct {
        X int
        Y int
        Z int
    }

    var s1 XYZ
    fmt.Println(s1.Y, s1.Z)
}
```

As you can see, there is nothing that prevents you from defining a new structure type inside a function, but you should have a reason for doing so.

The third portion of `structures.go` follows:

```
p1 := XYZ{23, 12, -2}
p2 := XYZ{Z: 12, Y: 13}
fmt.Println(p1)
fmt.Println(p2)
```

Here you define two structure literals named `p1` and `p2`, which you print afterward.

The last part of `structures.go` contains the following Go code:

```
pSlice := [4]XYZ{}
pSlice[2] = p1
pSlice[0] = p2
fmt.Println(pSlice)
p2 = XYZ{1, 2, 3}
fmt.Println(pSlice)
```

In this last part, we created an array of structures named `pSlice`. As you will understand from the output of `structures.go`, when you assign a structure to an array of structures, the structure is copied into the array so changing the value of the original structure will have no effect on the objects of the array.

Executing `structures.go` will generate the next output:

```
$ go run structures.go
0 0
{23 12 -2}
{0 13 12}
[{{0 13 12} {0 0 0} {23 12 -2} {0 0 0}}
 {{0 13 12} {0 0 0} {23 12 -2} {0 0 0}}]
```



*Note that the order in which you put the fields in the definition of a structure type is significant for the **type identity** of the defined structure. Put simply, two structures with the same fields will not be considered identical in Go if their fields are not in exactly the same order.*

The output of `structures.go` illustrates that the zero value of a `struct` variable is constructed by zeroing all the fields of the `struct` variable according to their types.

Pointers to structures

In [Chapter 3](#), *Working with Basic Go Data Types*, we talked about pointers. In this section, we will look at an example that is related to **pointers to structures**. The name of the program will be `pointerStruct.go` and will be presented in four parts.

The first part of the program contains the next Go code:

```
package main

import (
    "fmt"
)

type myStructure struct {
    Name    string
    Surname string
    Height  int32
}
```

The second code segment from `pointerStruct.go` follows:

```
func createStruct(n, s string, h int32) *myStructure {
    if h > 300 {
        h = 0
    }
    return &myStructure{n, s, h}
}
```

The approach used in `createStruct()` for creating a new structure variable has many advantages over initializing structure variables on your own, including the fact that you are allowed to check whether the provided information is both correct and valid. Additionally, this approach is cleaner – there is a central point where structure variables are initialized so if there is something wrong with your `struct` variables, you know where to look and who to blame! Note that some people might prefer to name the `createStruct()` function `NewStruct()`.



For those with a C or C++ background, it is perfectly legal for a Go function to return the memory address of a local variable. Nothing gets lost, so everybody is happy.

The third portion of `pointerStruct.go` is as follows:

```
func retStructure(n, s string, h int32) myStructure {
    if h > 300 {
        h = 0
    }
    return myStructure{n, s, h}
}
```

This part presents the no-pointer version of the `createStruct()` function named `retStructure()`. Both functions work fine, so choosing between the implementation of `createStruct()` and `retStructure()` is just a matter of personal preference. More appropriate names for these two functions might have been `NewStructurePointer()` and `NewStructure()`, respectively.

The last part of `pointerStruct.go` is shown in the following Go code:

```
func main() {
    s1 := createStruct("Mihalis", "Tsoukalos", 123)
    s2 := retStructure("Mihalis", "Tsoukalos", 123)
    fmt.Println((*s1).Name)
    fmt.Println(s2.Name)
    fmt.Println(s1)
    fmt.Println(s2)
}
```

If you execute `pointerStruct.go`, you will get the next output:

```
$ go run pointerStruct.go
Mihalis
Mihalis
&{Mihalis Tsoukalos 123}
{Mihalis Tsoukalos 123}
```

Here you can see one more time that the main difference between `createStruct()` and `retStructure()` is that the former returns a pointer to a structure, which means that you will need to dereference that pointer in order to use the object it points to, whereas the latter returns an entire structure object. This can make your code a little uglier.



Structures are very important in Go and are used extensively in real-world programs because they allow you to group as many values as you want and treat those values as a single entity.

Using the `new` keyword

Go supports the `new` keyword, which allows you to allocate new objects. However, there is a very important detail that you need to remember about `new`: `new` returns the memory address of the allocated object. Put simply, `new` returns a pointer.

So, you can create a fresh `aStructure` variable as follows:

```
| ps := new(aStructure)
```

After executing the `new` statement, you are ready to work with your fresh variable that has its allocated memory zeroed but not initialized.



The main difference between `new` and `make` is that variables created with `make` are properly initialized without just zeroing the allocated memory space. Additionally, `make` can only be applied to maps, channels, and slices, and does not return a memory address, which means that `make` does not return a pointer.

The next statement will create a slice with `new` that points to `nil`:

```
| sp := new([]aStructure)
```

Tuples

Strictly speaking, a **tuple** is a finite ordered list with multiple parts. The most important thing about tuples is that Go has no support for the tuple type, which means that Go does not officially care about tuples, despite the fact that it has support for certain uses of tuples.

One interesting thing here is that we have been using Go tuples in this book since [Chapter 1](#), *Go and the Operating System*, in statements such as the next one, where a function returns two values that you get in a single statement:

```
|min, _ := strconv.ParseFloat(arguments[1], 64)
```

The name of the Go program that will illustrate Go tuples is `tuples.go` and it will be presented in three code segments. Please note that the presented code uses a function that returns three values as a tuple. You will learn more about functions in [Chapter 6](#), *What You Might Not Know About Go Packages and Go Functions*.

The first part of `tuples.go` is as follows:

```
package main

import (
    "fmt"
)

func retThree(x int) (int, int, int) {
    return 2 * x, x * x, -x
}
```

You can see the implementation of a function named `retThree()` that returns a tuple containing three integer values. This capability permits the function to return multiple values, without the need to group the various return values into a structure, and return a structure variable instead.

In [Chapter 6](#), *What You Might Not Know About Go Packages and Go Functions*, you will learn how to put names to the return values of a Go

function, which is a very handy feature that can save you from various types of bugs.

The second part of `tuples.go` is as follows:

```
func main() {
    fmt.Println(retThree(10))
    n1, n2, n3 := retThree(20)
    fmt.Println(n1, n2, n3)
```

Here, we use the `retThree()` function twice. Firstly, we do this without saving its return values. Secondly, we do this by saving the three return values of `retThree()` into three different variables using a single statement, which in Go terminology is called a **tuple assignment**, hence the confusion about Go supporting tuples.

If you do not care about one or more return values of a function, you can put an **underscore character** (`_`) in their place. Note that it is a compile time error in Go if you declare a variable and do not use it afterward.

The third part of the program is shown in the following Go code:

```
n1, n2 = n2, n1
fmt.Println(n1, n2, n3)

x1, x2, x3 := n1*2, n1*n1, -n1
fmt.Println(x1, x2, x3)
}
```

As you can see, tuples can do many intelligent things, such as swapping values without the need for a temporary variable, as well as evaluating expressions.

Executing `tuples.go` will create the next output:

```
$ go run tuples.go
20 100 -10
40 400 -20
400 40 -20
800 160000 -400
```

Regular expressions and pattern matching

Pattern matching, which plays a key role in Go, is a technique for searching a string for some set of characters based on a specific search pattern that is based on regular expressions and **grammars**. If pattern matching is successful, it allows you to extract the desired data from the string, replace it, or delete it.

The Go package responsible for defining regular expressions and performing pattern matching is called `regexp`. You will see it in action later in this chapter.



When using a regular expression in your code, you should consider the definition of the regular expression as the most important part of the relevant code because the functionality of that code depends on the regular expression.

Introducing some theory

Every regular expression is compiled into a recognizer by building a generalized transition diagram called a **finite automaton**. A finite automaton can be either deterministic or nondeterministic.

Nondeterministic means that more than one transition out of a state can be possible for the same input. A **recognizer** is a program that takes a string x as input and is able to tell whether x is a sentence of a given language.

A **grammar** is a set of production rules for strings in a formal language. The production rules describe how to create strings from the alphabet of the language that are valid according to the syntax of the language. A grammar does not describe the meaning of a string or what can be done with it in whatever context – it only describes its form. What is important here is to realize that grammars are at the heart of regular expressions because without a grammar, you cannot define or use a regular expression.



Although regular expressions allow you to solve problems that it would be extremely difficult to solve otherwise, do not try to solve every problem you face with a regular expression. Always use the right tool for the job.

The rest of this section will present three examples of regular expressions and pattern matching.

A simple example

In this subsection, you will learn how to select a particular column from a line of text. To make things more interesting, you will also learn how to read a text file line by line. However, file I/O is the subject of [Chapter 8](#), *Telling a UNIX System What to Do*, so you should refer to that chapter in order to get more information about the relevant Go code.

The name of the Go source file is `selectColumn.go`. It will be presented in five segments. The utility needs at least two command-line arguments to operate; the first one is the required column number and the second one is the path of the text file to process. However, you can use as many text files as you want – `selectColumn.go` will process all of them one by one.

The first part of `selectColumn.go` follows:

```
package main

import (
    "bufio"
    "fmt"
    "io"
    "os"
    "strconv"
    "strings"
)
```

The second code portion of `selectColumn.go` contains the following Go code:

```
func main() {
    arguments := os.Args
    if len(arguments) < 2 {
        fmt.Printf("usage: selectColumn column <file1> [<file2> [... <fileN>]]\n")
        os.Exit(1)
    }

    temp, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println("Column value is not an integer:", temp)
        return
    }

    column := temp
    if column < 0 {
        fmt.Println("Invalid Column number!")
```

```
|     os.Exit(1)
| }
```

The first test the program performs is to make sure that it has an adequate number of command-line arguments (`len(arguments) < 2`). Additionally, you need two more tests to make sure that the provided column value is actually a number and that it is bigger than 0.

The third part of `selectColumn.go` follows:

```
| for _, filename := range arguments[2:] {
|     fmt.Println("\t\t", filename)
|     f, err := os.Open(filename)
|     if err != nil {
|         fmt.Printf("error opening file %s\n", err)
|         continue
|     }
|     defer f.Close()
```

The program performs various tests to make sure that the text file does exist and that you can read it – the `os.Open()` function is used for opening the text file. Remember that the UNIX file permissions of a text file might not allow you to read it.

The fourth code piece of `selectColumn.go` is as follows:

```
|     r := bufio.NewReader(f)
|     for {
|         line, err := r.ReadString('\n')
|
|         if err == io.EOF {
|             break
|         } else if err != nil {
|             fmt.Printf("error reading file %s", err)
|         }
|     }
```

As you will learn in [Chapter 8, Telling a UNIX System What to Do](#), the `bufio.ReadString()` function reads a file until the first occurrence of its parameter. As a result, `bufio.ReadString('\n')` tells Go to read a file line by line because `\n` is the UNIX newline character. The `bufio.ReadString()` function returns a **byte slice**.

The last code fragment of `selectColumn.go` is next:

```
|     data := strings.Fields(line)
|     if len(data) >= column {
```

```
        fmt.Println((data[column-1]))  
    }  
}  
}  
}
```

The logic behind the program is pretty simple: you split each line of text and select the desired column. However, as you cannot be sure that the current line has the required number of fields, you check that before printing any output. This is the simplest form of pattern matching because each line is split using space characters as word separators.

If you want more information about the splitting of lines, then you will find it useful to know that the `strings.Fields()` function splits a string based on the whitespace characters that are defined in the `unicode.IsSpace()` function and returns a slice of strings.

Executing `selectColumn.go` will generate the next kind of output:

```
$ go run selectColumn.go 15 /tmp/swtag.log /tmp/adobegc.log | head
                                /tmp/swtag.log
                                /tmp/adobegc.log
AdobeGCData
Successfully
Initializing
Stream
*****AdobeGC
Perform
Perform
Trying
```

The `selectColumn.go` utility prints the name of each processed file even if you get no output from that file.



The important thing to remember is that you should never trust your data, especially when it comes from non-technical users. Put simply, always verify that the data you expect to grab is there.

A more advanced example

In this section, you will learn how to match a date and time string as found in the log files of an Apache web server. To make things even more interesting, you will also learn how to change the date and time format of the log file into a different format. Once again, this requires reading the Apache log file, which is a plain text file, line by line.

The name of the command-line utility is `changeDT.go` and it will be presented in five parts. Note that `changeDT.go` is an improved version of the `timeDate.go` utility presented in [Chapter 3](#), *Working with Basic Go Data Types*, not only because it gets its data from an external file, but also because `changeDT.go` uses two regular expressions and therefore it is able to match strings in two different time and date formats.



There is a very important point here: do not try to implement every possible feature in the first version of your utilities. It is a better approach to build a working version with fewer features and improve that version in small steps.

The first chunk of code from `changeDT.go` follows:

```
package main

import (
    "bufio"
    "fmt"
    "io"
    "os"
    "regexp"
    "strings"
    "time"
)
```

Lots of packages are needed because `changeDT.go` does so many fascinating things.

The second piece of code from `changeDT.go` is the following:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
```

```

        fmt.Println("Please provide one text file to process!")
        os.Exit(1)
    }

    filename := arguments[1]
    f, err := os.Open(filename)
    if err != nil {
        fmt.Printf("error opening file %s", err)
        os.Exit(1)
    }
    defer f.Close()

    notAMatch := 0
    r := bufio.NewReader(f)
    for {
        line, err := r.ReadString('\n')
        if err == io.EOF {
            break
        } else if err != nil {
            fmt.Printf("error reading file %s", err)
        }
    }
}

```

In this part, you just try to open your input file for reading in order to read it line by line. The `notAMatch` variable holds the number of lines in the input file that did not match any one of the two regular expressions of the program.

The third code segment of `changeDT.go` comes with the following Go code:

```

r1 := regexp.MustCompile(`.*\[ (\d\d\w+/\d\d\d\d:\d\d:\d\d:\d\d.*)\] .*`)
if r1.MatchString(line) {
    match := r1.FindStringSubmatch(line)
    d1, err := time.Parse("02/Jan/2006:15:04:05 -0700", match[1])
    if err == nil {
        newFormat := d1.Format(time.Stamp)
        fmt.Print(strings.Replace(line, match[1], newFormat, 1))
    } else {
        notAMatch++
    }
    continue
}

```

Here you can see that if the first date and time format is not a match, the program will continue its execution. However, if it enters the `if` block, the `continue` statement will get executed, which means that it will skip the remaining code of the surrounding `for` loop. Thus, in the first supported format, the time and date string has the `21/Nov/2017:19:28:09 +0200` format.

The `regexp.MustCompile()` function is like `regexp.Compile()` but panics if the expression cannot be parsed. The parentheses around the regular expression

allows you to use the matches afterwards. In this case, you can only have one match, which you will get using the `regexp.FindStringSubmatch()` function.

The fourth part of `changeDT.go` follows:

```
r2 := regexp.MustCompile(`.*\[ (\w+-\d\d-\d\d:\d\d:\d\d:\d\d.*) \] .*`)
if r2.MatchString(line) {
    match := r2.FindStringSubmatch(line)
    d1, err := time.Parse("Jan-02-06:15:04:05 -0700", match[1])
    if err == nil {
        newFormat := d1.Format(time.Stamp)
        fmt.Println(strings.Replace(line, match[1], newFormat, 1))
    } else {
        notAMatch++
    }
    continue
}
```

The second supported time and date format is `Jun-21-17:19:28:09 +0200`. As you can appreciate, there are not many differences between the two formats. Note that although the program uses just two date and time formats, you can have as many of these types of formats as you desire.

The last code portion from `changeDT.go` contains the following Go code:

```
}
```

Here, you print the number of lines that did not match any one of the two formats.

The text file that will be used for testing `changeDT.go` will contain the next lines:

```
$ cat logEntries.txt
- - [21/Nov/2017:19:28:09 +0200] "GET /AMEv2.tif.zip HTTP/1.1" 200 2188249 "-"
- - [21/Jun/2017:19:28:09 +0200] "GET /AMEv2.tif.zip HTTP/1.1" 200
- - [25/Lun/2017:20:05:34 +0200] "GET /MongoDjango.zip HTTP/1.1" 200 118362
- - [Jun-21-17:19:28:09 +0200] "GET /AMEv2.tif.zip HTTP/1.1" 200
- - [20/Nov/2017:20:05:34 +0200] "GET /MongoDjango.zip HTTP/1.1" 200 118362
- - [35/Nov/2017:20:05:34 +0200] "GET MongoDjango.zip HTTP/1.1" 200 118362
```

Executing `changDT.go` will generate the next output:

```
$ go run changeDT.go logEntries.txt
- - [Nov 21 19:28:09] "GET /AMEv2.tif.zip HTTP/1.1" 200 2188249 "-"
```

```
| - - [Jun 21 19:28:09] "GET /AMEv2.tif.zip HTTP/1.1" 200  
| - - [Jun 21 19:28:09] "GET /AMEv2.tif.zip HTTP/1.1" 200  
| - - [Nov 20 20:05:34] "GET /MongoDjango.zip HTTP/1.1" 200 118362  
2 lines did not match!
```

Matching IPv4 addresses

An **IPv4 address**, or simply an **IP address**, has four discrete parts. As an IPv4 address is stored using 8-bit binary numbers, each part can have values from 0, which is 00000000 in the binary format, to 255, which is equal to 11111111 in the binary format.



The format of an IPv6 address is much more complicated than the format of an IPv4 address so the presented program will not work with IPv6 addresses.

The name of the program will be `findIPv4.go` and it is going to be presented in five parts. The first part of `findIPv4.go` is shown here:

```
package main

import (
    "bufio"
    "fmt"
    "io"
    "net"
    "os"
    "path/filepath"
    "regexp"
)
```

As `findIPv4.go` is a pretty sophisticated utility, it needs many standard Go packages.

The second part is shown in the following Go code:

```
func findIP(input string) string {
    partIP := "(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|1[0-9]?[0-9])"
    grammar := partIP + "\\. " + partIP + "\\. " + partIP + "\\. "
    partIP
    matchMe := regexp.MustCompile(grammar)
    return matchMe.FindString(input)
}
```

The preceding code contains the definition of the regular expression that will help you to discover an IPv4 address inside a function. This is the most critical part of the program because if you define the regular expression incorrectly, you will never be able to catch any IPv4 addresses.

Before explaining the regular expression a little further, it is important to understand that prior to defining one or more regular expressions, you should be aware of the problem you are trying to solve. In other words, if you are not aware of the fact that the decimal values of an IPv4 address cannot be larger than 255, no regular expression can save you!

Now that we are on the same page, let us talk about the next two statements:

```
| partIP := "(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|1-9]?[0-9])"  
| grammar := partIP + "\\. " + partIP + "\\. " + partIP + "\\. " + partIP
```

The regular expression defined in `partIP` matches each one of the four parts of an IP address. A valid IPv4 address can begin with 25 and end with 0, 1, 2, 3, 4, or 5 because that is the biggest 8-bit binary number ($25[0-5]$), or it can begin with 2 followed by 0, 1, 2, 3, or 4 and end with 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9 ($2[0-4][0-9]$).

Alternatively, it can begin with 1 followed by two more digits from 0 to 9 ($1[0-9][0-9]$). The last alternative would be a natural number that has one or two digits. The first digit, which is optional, can be from 1 to 9 and the second, which is mandatory, can be from 0 to 9 ($1-9)?[0-9]$).

The `grammar` variable tells us that what we are looking for has four distinct parts, and each one of them must match `partIP`. That `grammar` variable is what matches the complete IPv4 address that we seek.



As `findIPv4.go` works with regular expressions to find an IPv4 address in a file, it can process any kind of text file that contains valid IPv4 addresses.

Finally, if you have any special requirements, such as excluding certain IPv4 addresses or watching for specific addresses or networks, you can easily change the Go code of `findIPv4.go` and add the extra functionality you desire, which is the kind of flexibility achieved when you develop your own tools.

The third part of the `findIPv4.go` utility contains the following Go code:

```

func main() {
    arguments := os.Args
    if len(arguments) < 2 {
        fmt.Printf("usage: %s logFile\n", filepath.Base(arguments[0]))
        os.Exit(1)
    }

    for _, filename := range arguments[1:] {
        f, err := os.Open(filename)
        if err != nil {

            fmt.Printf("error opening file %s\n", err)
            os.Exit(-1)
        }
        defer f.Close()
    }
}

```

Firstly, you make sure that you have a sufficient number of command-line arguments by checking the length of `os.Args`. Then, you use a `for` loop to iterate over all of the command-line arguments.

The fourth code portion is as follows:

```

r := bufio.NewReader(f)
for {
    line, err := r.ReadString('\n')
    if err == io.EOF {
        break
    } else if err != nil {
        fmt.Printf("error reading file %s", err)
        break
    }
}

```

As happened in `selectColumn.go`, you use `bufio.ReadString()` to read your input line by line.

The last part of `findIPv4.go` contains the following Go code:

```

        ip := findIP(line)
        trial := net.ParseIP(ip)
        if trial.To4() == nil {
            continue
        }
        fmt.Println(ip)
    }
}

```

For each line of the input text file, you call the `findIP()` function. The `net.ParseIP()` function double-checks that we are dealing with a valid IPv4 address – it is never a bad thing to double-check! If the call to `net.ParseIP()`

is successful, you print the IPv4 address you just found. After that, the program will deal with the next line of input.

Executing `findIPv4.go` will generate the next kind of output:

```
$ go run findIPv4.go /tmp/auth.log
116.168.10.9
192.31.20.9
10.10.16.9
10.31.160.9
192.168.68.194
```

Thus, the output of `findIPv4.go` can have lines that are displayed multiple times. Apart from that detail, the output of the utility is pretty straightforward.

Processing the preceding output with some traditional UNIX command-line utilities might help you to reveal more information about your data:

```
$ go run findIPv4.go /tmp/auth.log.1 /tmp/auth.log | sort -rn | uniq -c | sort -rn
38 xxx.zz.116.9
33 x.zz.2.190
25 xx.zzz.1.41
20 178.132.1.18
18 x.zzz.63.53
17 178.zzz.y.9
15 103.yyy.zzz.179
10 213.z.yy.194
10 yyy.zzz.110.4
 9 yy.xx.65.113
```

What we did here is find out the top-10 IPv4 addresses located in the processed text files using the `sort(1)` and `uniq(1)` UNIX command-line utilities. The logic behind this pretty long `bash(1)` shell command is simple: the output of the `findIPv4.go` utility will become the input of the first `sort -rn` command in order to be sorted numerically in reverse order. Then, the `uniq -c` command removes the lines that appear multiple times by replacing them with a single line that is preceded with the count of the number of times the line occurred in the input. The output is then sorted once again so that the IPv4 addresses with the higher number of occurrences will appear first.

Once again, it is important to realize that the core functionality of `findIPv4.go` is implemented through the regular expression. If the regular expression is defined





*incorrectly or does not match all the cases (**false negative**) or matches things that should not be matched (**false positive**) then your program will not work correctly.*

Strings

Strictly speaking, a **string** in Go is not a composite type, but there are so many Go functions that support strings that I decided to describe strings in more detail in this chapter.

As discussed in [Chapter 3](#), *Working with Basic Go Data Types*, strings in Go are value types, not pointers as is the case with **C strings**. Additionally, Go supports UTF-8 strings by default, which means that you do not need to load any special packages or do anything tricky in order to print Unicode characters. However, there are subtle differences between a character, a rune, and a byte, as well as differences between a string and a string literal, which are going to be clarified here.

A Go string is a read-only **byte slice** that can hold any type of bytes and can have an arbitrary length.

You can define a new **string literal** as follows:

```
| const sLiteral = "\x99\x42\x32\x55\x50\x35\x23\x50\x29\x9c"
```

You might be surprised with the look of the string literal. You can define a string variable as follows:

```
| s2 := "€£³"
```

You can find the length of a string variable or a string literal using the `len()` function.

The `strings.go` file will illustrate many standard operations related to strings and will be presented in five parts. The first is shown in the following Go code:

```
| package main
| import (
|     "fmt"
| )
```

The second portion of Go code is as follows:

```
func main() {
    const sLiteral = "\x99\x42\x32\x55\x50\x35\x23\x50\x29\x9c"
    fmt.Println(sLiteral)
    fmt.Printf("x: %x\n", sLiteral)

    fmt.Printf("sLiteral length: %d\n", len(sLiteral))
```

Each `\xAB` sequence represents a single character of `sLiteral`. As a result, calling `len(sLiteral)` will return the number of characters of `sLiteral`. Using `%x` in `fmt.Printf()` will return the `AB` part of a `\xAB` sequence.

The third code segment of `strings.go` is shown in the following Go code:

```
for i := 0; i < len(sLiteral); i++ {
    fmt.Printf("%x ", sLiteral[i])
}
fmt.Println()

fmt.Printf("q: %q\n", sLiteral)
fmt.Printf("+q: %+q\n", sLiteral)
fmt.Printf(" x: % x\n", sLiteral)

fmt.Printf("s: As a string: %s\n", sLiteral)
```

Here, you can see that you can access a string literal as if it is a slice. Using `%q` in `fmt.Printf()` with a string argument will print a double-quoted string that is safely escaped with Go syntax. Using `%+q` in `fmt.Printf()` with a string argument will guarantee ASCII-only output.

Last, using `% x` (note the space between the `%` character and the `x` character) in `fmt.Printf()` will put spaces between the printed bytes. In order to print a string literal as a string, you will need to call `fmt.Printf()` with `%s`.

The fourth code portion of `strings.go` is as follows:

```
s2 := "€£³"
for x, y := range s2 {
    fmt.Printf("%#U starts at byte position %d\n", y, x)
}

fmt.Printf("s2 length: %d\n", len(s2))
```

Here you define a string named `s2` with three Unicode characters. Using `fmt.Printf()` with `%#U` will print the characters in the `U+0058` format. Using the

`range` keyword on a string that contains Unicode characters will allow you to process its Unicode characters one by one.

The output of `len(s2)` might surprise you a little. As the `s2` variable contains Unicode characters, its byte size is larger than the number of characters in it.

The last part of `strings.go` is as follows:

```
const s3 = "ab12AB"
fmt.Println("s3:", s3)
fmt.Printf("x: % x\n", s3)

fmt.Printf("s3 length: %d\n", len(s3))

for i := 0; i < len(s3); i++ {
    fmt.Printf("%x ", s3[i])
}
fmt.Println()
```

Running `strings.go` will generate the next output:

```
$ go run strings.go
◆B2UP5#P)◆
x: 9942325550352350299c
sLiteral length: 10
99 42 32 55 50 35 23 50 29 9c
q: "\x99B2UP5#P) \x9c"
+q: "\x99B2UP5#P) \x9c"
x: 99 42 32 55 50 35 23 50 29 9c
s: As a string: ◆B2UP5#P)◆
U+20AC '€' starts at byte position 0
U+00A3 'ƒ' starts at byte position 3
U+00B3 '߃' starts at byte position 5
s2 length: 7
s3: ab12AB
x: 61 62 31 32 41 42
s3 length: 6
61 62 31 32 41 42
```

It will be no surprise if you find the information presented in this section pretty strange and complex, especially if you are not familiar with Unicode and UTF-8 representations of characters and symbols. The good thing is that you will not need most of them in your everyday life as a Go developer; you will most likely get away with using simple `fmt.Println()` and `fmt.Printf()` commands in your programs to print your output. However, if

you are living outside of Europe and the U.S., you might find some of the information in this section pretty handy.

What is a rune?

A **rune** is an `int32` value, and therefore a Go type, that is used for representing a **Unicode code point**. A Unicode code point, or code position, is a numerical value that is usually used for representing single Unicode characters; however, it can also have alternative meanings, such as providing formatting information.



NOTE: You can consider a string as a collection of runes.

A **rune literal** is a character in single quotes. You may also consider a rune literal as a **rune constant**. Behind the scenes, a rune literal is associated with a Unicode code point.

Runes are going to be illustrated in `runes.go`, which is going to be presented in three parts. The first part of `runes.go` follows:

```
package main  
  
import (  
    "fmt"  
)
```

The second part of `runes.go` contains the following code:

```
func main() {  
    const r1 = '€'  
    fmt.Println("(int32) r1:", r1)  
    fmt.Printf("(HEX) r1: %x\n", r1)  
    fmt.Printf("(as a String) r1: %s\n", r1)  
    fmt.Printf("(as a character) r1: %c\n", r1)
```

First you define a rune literal named `r1`. (Please note that the Euro sign does not belong to the ASCII table of characters.) Then, you print `r1` using various statements. Next, you print its `int32` value and its hexadecimal value. After that, you try printing it as a string. Finally, you print it as a character, which is what gives you the same output as the one used in the definition of `r1`.

The third and last code segment of `runes.go` is shown in the following Go code:

```
fmt.Println("A string is a collection of runes:", []byte("Mihalis"))
aString := []byte("Mihalis")
for x, y := range aString {
    fmt.Println(x, y)
    fmt.Printf("Char: %c\n", aString[x])
}
fmt.Printf("%s\n", aString)
```

Here, you see that a byte slice is a collection of runes and that printing a byte slice with `fmt.Println()` might not return what you expected. In order to convert a rune into a character, you should use `%c` in a `fmt.Printf()` statement. In order to print a byte slice as a string, you will need to use `fmt.Sprintf()` with `%s`.

Executing `runes.go` will create the following output:

```
$ go run runes.go
(int32) r1: 8364
(HEX) r1: 20ac
(as a String) r1: %!s(int32=8364)
(as a character) r1: €
A string is a collection of runes: [77 105 104 97 108 105 115]
0 77
Char: M
1 105
Char: i
2 104
Char: h
3 97
Char: a
4 108
Char: l
5 105
Char: i
6 115
Char: s
Mihalis
```

Finally, the easiest way to get an `illegal rune literal` error message is by using single quotes instead of double quotes when importing a package:

```
$ cat a.go
package main

import (
    'fmt'
)
```

```
| func main() {  
| }  
| $ go run a.go  
| package main:  
| a.go:4:2: illegal rune literal
```

The unicode package

The `unicode` standard Go package contains various handy functions. One of them, which is called `unicode.IsPrint()`, can help you to identify the parts of a string that are printable using runes. This technique will be illustrated in the Go code of `unicode.go`, which will be presented in two parts. The first part of `unicode.go` is as follows:

```
package main

import (
    "fmt"
    "unicode"
)

func main() {
    const sL = "\x99\x00ab\x50\x00\x23\x50\x29\x9c"
```

The second code segment of `unicode.go` is shown in the following Go code:

```
for i := 0; i < len(sL); i++ {
    if unicode.IsPrint(rune(sL[i])) {
        fmt.Printf("%c\n", sL[i])
    } else {
        fmt.Println("Not printable!")
    }
}
```

As stated before, all of the dirty work is done by the `unicode.IsPrint()` function, which returns `true` when a rune is printable and `false` otherwise. If you are really into Unicode characters, you should definitely check the documentation page of the `unicode` package. Executing `unicode.go` will generate the following output:

```
$ go run unicode.go
Not printable!
Not printable!
a
b
p
Not printable!
#
p
)
```

| Not printable!

The strings package

The `strings` standard Go package allows you to manipulate UTF-8 strings in Go and includes many powerful functions. Most of these functions will be illustrated in the `useStrings.go` source file, which will be presented in five parts. Note that the functions of the `strings` package that are related to file input and output will be demonstrated in [Chapter 8, Telling a UNIX System What to Do](#).

The first part of `useStrings.go` follows:

```
package main

import (
    "fmt"
    "strings"
    "unicode"
)
```

```
var f = fmt.Printf
```

There is a difference in the way the `strings` package is imported. This kind of import statement makes Go create an alias for that package. So, instead of writing `strings.FunctionName()`, you can now write `s.FunctionName()`, which is a bit shorter. Please note that you will not be able to call a function of the `strings` package as `strings.FunctionName()` anymore.

 **TIP** Another handy trick is that if you find yourself using the same function all the time, and you want to use something shorter instead, you can assign a variable name to that function and use that variable name instead. Here, you can see that feature applied to the `fmt.Printf()` function. Nevertheless, you should not overuse that feature because you might end up finding it difficult to read the code!

The second part of `useStrings.go` contains the following Go code:

```
func main() {
    upper := s.ToUpper("Hello there!")
    f("To Upper: %s\n", upper)
    f("To Lower: %s\n", s.ToLower("Hello THERE"))
    f("%s\n", s.Title("tHis will be A title!"))
    f("EqualFold: %v\n", s.EqualFold("Mihalis", "MIHALIS"))
    f("EqualFold: %v\n", s.EqualFold("Mihalis", "MIHALI"))
```

In this code segment, you can see many functions that allow you to play with the case of a string. Additionally, you can see that the `strings.EqualFold()` function allows you to determine whether two strings are the same in spite of the differences in their letters.

The third code portion of `useStrings.go` follows:

```
f("Prefix: %v\n", s.HasPrefix("Mihalis", "Mi"))
f("Prefix: %v\n", s.HasPrefix("Mihalis", "mi"))
f("Suffix: %v\n", s.HasSuffix("Mihalis", "is"))
f("Suffix: %v\n", s.HasSuffix("Mihalis", "IS"))

f("Index: %v\n", s.Index("Mihalis", "ha"))
f("Index: %v\n", s.Index("Mihalis", "Ha"))
f("Count: %v\n", s.Count("Mihalis", "i"))
f("Count: %v\n", s.Count("Mihalis", "I"))
f("Repeat: %s\n", s.Repeat("ab", 5))

f("TrimSpace: %s\n", s.TrimSpace(" \tThis is a line. \n"))
f("TrimLeft: %s", s.TrimLeft(" \tThis is a\t line. \n", "\n\t"))
")
f("TrimRight: %s\n", s.TrimRight(" \tThis is a\t line. \n",
"\n\t"))
```

The `strings.Count()` function counts the number of non-overlapping times the second parameter appears in the string that is given as the first parameter. The `strings.HasPrefix()` function returns `true` when the first parameter string begins with the second parameter string, and `false` otherwise. Similarly, the `strings.HasSuffix()` function returns `true` when the first parameter, which is a string, ends with the second parameter, which is also a string, and `false` otherwise.

The fourth code segment of `useStrings.go` contains the following Go code:

```
f("Compare: %v\n", s.Compare("Mihalis", "MIHALIS"))
f("Compare: %v\n", s.Compare("Mihalis", "Mihalis"))
f("Compare: %v\n", s.Compare("MIHALIS", "MIHalis"))

f("Fields: %v\n", s.Fields("This is a string!"))
f("Fields: %v\n", s.Fields("Thisis\na\tstring!"))

f("%s\n", s.Split("abcd efg", ""))
```

This code portion contains some pretty advanced and ingenious functions. The first handy function is `strings.Split()`, which allows you to split the given string according to the desired separator string – the `strings.Split()`

function returns a **string slice**. Using `""` as the second parameter of `strings.Split()` will allow you to process a string character by character.

The `strings.Compare()` function compares two strings lexicographically, and it may return `0` if the two strings are identical, and `-1` or `+1` otherwise.

Lastly, the `strings.Fields()` function splits the string parameter using whitespace characters as separators. The whitespace characters are defined in the `unicode.IsSpace()` function.



`strings.Split()` is a powerful function that you should learn because sooner rather than later, you will have to use it in your programs.

The last part of `useStrings.go` is shown in the following Go code:

```
f("%s\n", s.Replace("abcd efg", "", "-", -1))
f("%s\n", s.Replace("abcd efg", "", "-", 4))
f("%s\n", s.Replace("abcd efg", "", "-", 2))

lines := []string{"Line 1", "Line 2", "Line 3"}
f("Join: %s\n", s.Join(lines, "++"))

f("SplitAfter: %s\n", s.SplitAfter("123++432++", "++"))

trimFunction := func(c rune) bool {
    return !unicode.IsLetter(c)
}
f("TrimFunc: %s\n", s.TrimFunc("123 abc ABC \t .",
    trimFunction))
}
```

As in the previous part of `useStrings.go`, the last code segment contains functions that implement some very intelligent functionality in a simple-to-understand and easy-to-use way.

The `strings.Replace()` function takes four parameters. The first parameter is the string that you want to process. The second parameter contains the string that, if found, will be replaced by the third parameter of `strings.Replace()`. The last parameter is the maximum number of replacements that are allowed to happen. If that parameter has a negative value, then there is no limit to the number of replacements that can take place.

The last two statements of the program define a **trim function**, which allows you to keep the runes of a string that interest you and utilize that function as the second argument to the `strings.TrimFunc()` function.

Lastly, the `strings.SplitAfter()` function splits its first parameter string into substrings based on the separator string that is given as the second parameter to the function.



For the full list of functions similar to `unicode.IsLetter()`, you should visit the documentation page of the `unicode` standard Go package.

Executing `useStrings.go` will create the next output:

```
$ go run useStrings.go
To Upper: HELLO THERE!
To Lower: hello there
THis WILL Be A Title!
EqualFold: true
EqualFold: false
Prefix: true
Prefix: false
Suffix: true
Suffix: false
Index: 2
Index: -1
Count: 2
Count: 0
Repeat: ababababab
TrimSpace: This is a line.
TrimLeft: This is a     line.
TrimRight:      This is a     line.
Compare: 1
Compare: 0
Compare: -1
Fields: [This is a string!]
Fields: [Thisis a string!]
[a b c d   e f g]
_a_b_c_d_ _e_f_g_
_a_b_c_d efg
_a_bcd efg
Join: Line 1+++Line 2+++Line 3
SplitAfter: [123++ 432++ ]
TrimFunc: abc ABC
```

Please note that the list of functions presented from the `strings` package is far from complete. You should see the documentation page of the `strings` package at <https://golang.org/pkg/strings/> for the complete list of available functions.

If you are working with text and text processing, you will definitely need to learn all the gory details and functions of the `strings` package, so make sure that you experiment with all these functions and create many examples that will help you to clarify things.

The switch statement

The main reason for presenting the `switch` statement in this chapter is because a `switch` case can use regular expressions. Firstly, however, take a look at this simple `switch` block:

```
switch asString {
    case "1":
        fmt.Println("One!")
    case "0":
        fmt.Println("Zero!")
    default:
        fmt.Println("Do not care!")
}
```

The preceding `switch` block can differentiate between the "`1`" string, the "`0`" string, and everything else (`default`).



Having a match all remaining cases case in a `switch` block is considered a very good practice. However, as the order of the cases in a `switch` block does matter, the match all remaining cases case should be put last. In Go, the name of the match all remaining cases case is `default`.

However, a `switch` statement can be much more flexible and adaptable:

```
switch {
    case number < 0:
        fmt.Println("Less than zero!")
    case number > 0:
        fmt.Println("Bigger than zero!")
    default:
        fmt.Println("Zero!")
}
```

The preceding `switch` block does the job of identifying whether you are dealing with a positive integer, a negative integer, or zero. As you can see, the branches of a `switch` statement can have conditions. You will soon see that the branches of a `switch` statement can also have regular expressions in them.

All these examples, and some additional ones, can be found in `switch.go`, which will be presented in five parts.

The first part of `switch.go` is as follows:

```
package main

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

func main() {

    arguments := os.Args
    if len(arguments) < 2 {
        fmt.Println("usage: switch number")
        os.Exit(1)
    }
}
```

The `regexp` package is needed for supporting regular expressions in `switch`.

The second code segment of `switch.go` is shown in the following Go code:

```
number, err := strconv.Atoi(arguments[1])
if err != nil {
    fmt.Println("This value is not an integer:", number)
} else {
    switch {
    case number < 0:
        fmt.Println("Less than zero!")
    case number > 0:
        fmt.Println("Bigger than zero!")
    default:
        fmt.Println("Zero!")
    }
}
```

The third part of `switch.go` is as follows:

```
asString := arguments[1]
switch asString {
case "5":
    fmt.Println("Five!")
case "0":
    fmt.Println("Zero!")
default:
    fmt.Println("Do not care!")
}
```

In this code segment, you can see that a `switch` case can also contain hardcoded values. This mainly occurs when the `switch` keyword is followed by the name of a variable.

The fourth code portion of `switch.go` contains the following Go code:

```
var negative = regexp.MustCompile(`-`)
var floatingPoint = regexp.MustCompile(`\d?\.\d`)
var email = regexp.MustCompile(`^@[^\@]+\@[^\.]+[^\@.]`)

switch {
case negative.MatchString(asString):
    fmt.Println("Negative number")
case floatingPoint.MatchString(asString):
    fmt.Println("Floating point!")
case email.MatchString(asString):
    fmt.Println("It is an email!")
    fallthrough
default:
    fmt.Println("Something else!")
}
```

Many interesting things are happening here. Firstly, you define three regular expressions named `negative`, `floatingPoint`, and `email`, respectively. Secondly, you use all three of them in the `switch` block with the help of the `regexp.MatchString()` function, which does the actual matching.

Lastly, the `fallthrough` keyword tells Go to execute the branch that follows the current one, which, in this case, is the `default` branch. This means that when the code of the `email.MatchString(asString)` case is the one that will get executed, the `default` case will also be executed.

The last part of `switch.go` is as follows:

```
var aType error = nil
switch aType.(type) {
case nil:
    fmt.Println("It is nil interface!")
default:
    fmt.Println("Not nil interface!")
}
```

Here, you can see that `switch` can differentiate between types. You will learn more about working with `switch` and Go **interfaces** in [Chapter 7, *Reflection and Interfaces for All Seasons*](#).

Executing `switch.go` with various input arguments will generate the next kind of output:

```
$ go run switch.go
usage: switch number.
exit status 1
$ go run switch.go mike@g.com
This value is not an integer: 0
Do not care!
It is an email!
Something else!
It is nil interface!
$ go run switch.go 5
Bigger than zero!
Five!
Something else!
It is nil interface!
$ go run switch.go 0
Zero!
Zero!
Something else!
It is nil interface!
$ go run switch.go 1.2
This value is not an integer: 0
Do not care!
Floating point!
It is nil interface!
$ go run switch.go -1.5
This value is not an integer: 0
Do not care!
Negative number
It is nil interface!
```

Calculating Pi with high accuracy

In this section, you will learn how to calculate **Pi** with high accuracy using a standard Go package named `math/big` and the special purpose types offered by that package.



This section contains the ugliest Go code that I have even seen; even Java code looks better than this!

The name of the program that uses Bellard's formula to calculate Pi is `calculatePi.go` and it will be presented in four parts.

The first part of `calculatePi.go` follows:

```
package main

import (
    "fmt"
    "math"
    "math/big"
    "os"
    "strconv"
)
var precision uint = 0
```

The `precision` variable holds the desired precision of the calculations, and it is made global in order to be accessible from everywhere in the program.

The second code segment of `calculatePi.go` is shown in the following Go code:

```
func Pi(accuracy uint) *big.Float {
    k := 0
    pi := new(big.Float).SetPrec(precision).SetFloat64(0)
    k1k2k3 := new(big.Float).SetPrec(precision).SetFloat64(0)
    k4k5k6 := new(big.Float).SetPrec(precision).SetFloat64(0)
    temp := new(big.Float).SetPrec(precision).SetFloat64(0)
    minusOne := new(big.Float).SetPrec(precision).SetFloat64(-1)
    total := new(big.Float).SetPrec(precision).SetFloat64(0)

    two2Six := math.Pow(2, 6)
    two2SixBig := new(big.Float).SetPrec(precision).SetFloat64(two2Six)
```

The `new(big.Float)` call creates a new `big.Float` variable with the required precision, which is set by `SetPrec()`.

The third part of `calculatePi.go` contains the remaining Go code of the `Pi()` function:

```
for {
    if k > int(accuracy) {
        break
    }
    t1 := float64(float64(1) / float64(10*k+9))
    k1 := new(big.Float).SetPrec(precision).SetFloat64(t1)
    t2 := float64(float64(64) / float64(10*k+3))
    k2 := new(big.Float).SetPrec(precision).SetFloat64(t2)
    t3 := float64(float64(32) / float64(4*k+1))
    k3 := new(big.Float).SetPrec(precision).SetFloat64(t3)
    k1k2k3.Sub(k1, k2)
    k1k2k3.Sub(k1k2k3, k3)

    t4 := float64(float64(4) / float64(10*k+5))
    k4 := new(big.Float).SetPrec(precision).SetFloat64(t4)
    t5 := float64(float64(4) / float64(10*k+7))
    k5 := new(big.Float).SetPrec(precision).SetFloat64(t5)
    t6 := float64(float64(1) / float64(4*k+3))
    k6 := new(big.Float).SetPrec(precision).SetFloat64(t6)
    k4k5k6.Add(k4, k5)
    k4k5k6.Add(k4k5k6, k6)
    k4k5k6 = k4k5k6.Mul(k4k5k6, minusOne)
    temp.Add(k1k2k3, k4k5k6)

    k7temp := new(big.Int).Exp(big.NewInt(-1), big.NewInt(int64(k)), nil)
    k8temp := new(big.Int).Exp(big.NewInt(1024), big.NewInt(int64(k)), nil)

    k7 := new(big.Float).SetPrec(precision).SetFloat64(0)
    k7.SetInt(k7temp)
```

```

    k8 := new(big.Float).SetPrec(precision).SetFloat64(0)
    k8.SetInt(k8temp)

    t9 := float64(256) / float64(10*k+1)
    k9 := new(big.Float).SetPrec(precision).SetFloat64(t9)
    k9.Add(k9, temp)
    total.Mul(k9, k7)
    total.Quo(total, k8)
    pi.Add(pi, total)

    k = k + 1
}
pi.Quo(pi, two2SixBig)
return pi
}

```

This part of the program is the Go implementation of Bellard's formula. The bad thing with `math/big` is that you need a special function of it for almost every kind of calculation, which mainly happens because those functions can keep the precision at the desired level. So, without using `big.Float` and `big.Int` variables, as well as the functions of `math/big` all the time, you cannot compute Pi with the desired precision.

The last part of `calculatePi.go` shows the implementation of the `main()` function:

```

func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide one numeric argument!")
        os.Exit(1)
    }

    temp, _ := strconv.ParseUint(arguments[1], 10, 32)
    precision = uint(temp) * 3

    PI := Pi(precision)
    fmt.Println(PI)
}

```

Executing `calculatePi.go` will generate the next kind of output:

```

$ go run calculatePi.go
Please provide one numeric argument!
exit status 1
$ go run calculatePi.go 20
3.141592653589793258
$ go run calculatePi.go 200
3.1415926535897932569603993617387624040191831562485732434931792835710464502489134671185117843176153542820179294162928090508139:

```

Developing a key-value store in Go

In this section, you will learn how to develop an unsophisticated version of a **key-value store** in Go, which means that you will learn how to implement the core functionality of a key-value store without any additional bells and whistles. The idea behind a key-value store is modest: answer queries fast and work as fast as possible. This translates into using simple algorithms and simple data structures.

The presented program will implement the four fundamental tasks of a key-value store:

1. Adding a new element
2. Deleting an existing element from the key-value store based on a key
3. Looking up the value of a specific key in the store
4. Changing the value of an existing key

These four functions allow you to have full control over the key-value store. The commands for these four functions will be named `ADD`, `DELETE`, `LOOKUP`, and `CHANGE`, respectively. This means that the program will only operate when it gets one of these four commands. Additionally, the program will stop when you enter the `STOP` word as input and will print the full contents of the key-value store when you enter the `PRINT` command.

The name of the program will be `keyValue.go` and it will be presented in five code segments.

The first code segment of `keyValue.go` follows:

```
package main

import (
    "bufio"
    "fmt"
    "os"
    "strings"
)

type myElement struct {
```

```

    Name      string
    Surname   string
    Id        string
}

var DATA = make(map[string]myElement)

```

The key-value store is stored in a native Go map because using a built-in Go structure is usually faster. The map variable is defined as a global variable, where its keys are `string` variables and its values are `myElement` variables. You can also see the definition of the `myElement` struct type here.

The second code segment of `keyValue.go` is as follows:

```

func ADD(k string, n myElement) bool {
    if k == "" {
        return false
    }

    if LOOKUP(k) == nil {
        DATA[k] = n
        return true
    }
    return false
}

func DELETE(k string) bool {
    if LOOKUP(k) != nil {
        delete(DATA, k)
        return true
    }
    return false
}

```

This code contains the implementation of two functions that support the functionality of the `ADD` and `DELETE` commands. Note that if the user tries to add a new element to the store without giving enough values to populate the `myElement` struct, the `ADD` function will fail. For this particular program, the missing fields of the `myElement` struct will be set to the empty string. However, if you try to add a key that already exists, you will get an error message instead of modifying the value of the existing key.

The third portion of `keyValue.go` contains the following code:

```

func LOOKUP(k string) *myElement {
    _, ok := DATA[k]
    if ok {
        n := DATA[k]
        return &n
    }
}

```

```

    } else {
        return nil
    }
}

func CHANGE(k string, n myElement) bool {
    DATA[k] = n
    return true
}

func PRINT() {
    for k, d := range DATA {
        fmt.Printf("key: %s value: %v\n", k, d)
    }
}

```

In this Go code segment, you can see the implementation of the functions that support the functionality of the `LOOKUP` and `CHANGE` commands. If you try to change a key that does not exist, the program will add that key to the store without generating any error messages. In this part, you can also see the implementation of the `PRINT()` function that prints the full contents of the key-value store.

The reason for using ALL CAPS for the names of these functions is that they are really important for the program.

The fourth part of `keyValue.go` is as follows:

```

func main() {
    scanner := bufio.NewScanner(os.Stdin)
    for scanner.Scan() {
        text := scanner.Text()
        text = strings.TrimSpace(text)
        tokens := strings.Fields(text)

        switch len(tokens) {
        case 0:
            continue
        case 1:
            tokens = append(tokens, "")
            tokens = append(tokens, "")
            tokens = append(tokens, "")
            tokens = append(tokens, "")
        case 2:
            tokens = append(tokens, "")
            tokens = append(tokens, "")
            tokens = append(tokens, "")
        case 3:
            tokens = append(tokens, "")
            tokens = append(tokens, "")
        case 4:
            tokens = append(tokens, "")
        }
    }
}

```

In this part of `keyValue.go`, you read the input from the user. Firstly, the `for` loop makes sure that the program will keep running for as long as the user provides some input. Secondly, the program makes sure that the `tokens` slice has at least five elements, even though only the `ADD` command needs that number of elements. Thus, for an `ADD` operation to be complete and not have any missing values, you will need an input that looks like `ADD aKey Field1 Field2 Field3.`

The last part of `keyValue.go` is shown in the following Go code:

```
switch tokens[0] {
    case "PRINT":
        PRINT()
    case "STOP":
        return
    case "DELETE":
        if !DELETE(tokens[1]) {
            fmt.Println("Delete operation failed!")
        }
    case "ADD":
        n := myElement{tokens[2], tokens[3], tokens[4]}
        if !ADD(tokens[1], n) {
            fmt.Println("Add operation failed!")
        }
    case "LOOKUP":
        n := LOOKUP(tokens[1])
        if n != nil {
            fmt.Printf("%v\n", *n)
        }
    case "CHANGE":
        n := myElement{tokens[2], tokens[3], tokens[4]}
        if !CHANGE(tokens[1], n) {
            fmt.Println("Update operation failed!")
        }
    default:
        fmt.Println("Unknown command - please try again!")
}
```

In this part of the program, you process the input from the user. The `switch` statement makes the design of the program very clean and saves you from having to use multiple `if...else` statements.

Executing and using `keyValue.go` will create the following output:

```
$ go run keyValue.go
UNKNOWN
Unknown command - please try again!
ADD 123 1 2 3
ADD 234 2 3 4
```

```

ADD 345
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
key: 345 value: {}
ADD 345 3 4 5
Add operation failed!
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
key: 345 value: {}
CHANGE 345 3 4 5
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
key: 345 value: {3 4 5}
DELETE 345
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
DELETE 345
Delete operation failed!
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
ADD 345 3 4 5
ADD 567 -5 -6 -7
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
key: 345 value: {3 4 5}
key: 567 value: {-5 -6 -7}
CHANGE 345
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
key: 345 value: {}
key: 567 value: {-5 -6 -7}
STOP

```

You will have to wait until [Chapter 8, Telling a UNIX System What to Do](#), in order to learn how to add data persistence to the key-value store.

You can also improve `keyValue.go` by adding goroutines and channels to it. However, adding goroutines and channels to a single-user application has no practical purpose. But if you make `keyValue.go` able to operate over **Transmission Control Protocol/Internet Protocol (TCP/IP)** networks, then the use of goroutines and channels will allow it to accept multiple connections and serve multiple users.

You will learn more about routines and channels in [Chapter 9, Concurrency in Go – Goroutines, Channels, and Pipelines](#), and in [Chapter 10, Concurrency](#)

in Go – Advanced Topics.

You will then learn about creating network applications in Go in [Chapter 12](#), *The Foundations of Network Programming in Go*, and in [Chapter 13](#), *Network Programming – Building Your Own Servers and Clients*.

Go and the JSON format

JSON is a very popular text-based format designed to be an easy and light way to pass information between JavaScript systems. However, JSON is also being used for creating configuration files for applications and storing data in a structured format.

The `encoding/json` package offers the `Encode()` and `Decode()` functions, which allow the conversion of a Go object into a JSON document and vice versa. Additionally, the `encoding/json` package offers the `Marshal()` and `Unmarshal()` functions, which work similarly to `Encode()` and `Decode()` and are based on the `Encode()` and `Decode()` methods. The main difference between the `Marshal()` and `Unmarshal()` pair and the `Encode()` and `Decode()` pair is that the former pair works on single objects, whereas the latter pair of functions can work on multiple objects as well as streams of bytes.

Reading JSON data

In this section, you will learn how to read a JSON record from disk using the code of `readJSON.go`, which will be presented in three parts.

The first part of `readJSON.go` is shown in the following Go code:

```
package main

import (
    "encoding/json"
    "fmt"
    "os"
)

type Record struct {
    Name      string
    Surname   string
    Tel       []Telephone
}

type Telephone struct {
    Mobile    bool
    Number   string
}
```

In this Go code, we define the structure variables that are going to keep the JSON data.

The second part of `readJSON.go` is as follows:

```
func loadFromJSON(filename string, key interface{}) error {
    in, err := os.Open(filename)
    if err != nil {
        return err
    }

    decodeJSON := json.NewDecoder(in)
    err = decodeJSON.Decode(key)
    if err != nil {
        return err
    }
    in.Close()
    return nil
}
```

Here, we define a new function named `loadFromJSON()` that is used for decoding the data of a JSON file according to a data structure that is given

as the second argument to it. We first call the `json.NewDecoder()` function to create a new JSON decode variable that is associated with a file, and then we call the `Decode()` function for actually decoding the contents of the file and putting them into the desired variable.

The last part of `readJSON.go` is the following:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a filename!")
        return
    }

    filename := arguments[1]

    var myRecord Record
    err := loadFromJSON(filename, &myRecord)
    if err == nil {
        fmt.Println(myRecord)
    } else {
        fmt.Println(err)
    }
}
```

The contents of `readMe.json` are the following:

```
$ cat readMe.json
{
    "Name": "Mihalis",
    "Surname": "Tsoukalos",
    "Tel": [
        {"Mobile": true, "Number": "1234-567"}, 
        {"Mobile": true, "Number": "1234-abcd"}, 
        {"Mobile": false, "Number": "abcc-567"}
    ]
}
```

Executing `readJSON.go` will generate the following output:

```
$ go run readJSON.go readMe.json
{Mihalis Tsoukalos [{true 1234-567} {true 1234-abcd} {false abcc-567}]}
```

Saving JSON data

In this subsection, you will learn how to write JSON data. The utility that will be presented in three parts is called `writeJSON.go` and it will write to standard output (`os.Stdout`), which means that it will write on the terminal screen.

The first part of `writeJSON.go` is as follows:

```
package main

import (
    "encoding/json"
    "fmt"
    "os"
)

type Record struct {
    Name     string
    Surname string
    Tel      []Telephone
}

type Telephone struct {
    Mobile bool
    Number string
}
```

The second part of `writeJSON.go` is the following:

```
func saveToJSON(filename *os.File, key interface{}) {
    encodeJSON := json.NewEncoder(filename)
    err := encodeJSON.Encode(key)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

The `saveToJSON()` function does all the work for us as it creates a JSON encoder variable named `encodeJSON`, which is associated with a filename, which is where the data is going to be put. The call to `Encode()` is what saves the data into the desired file after encoding it.

The last part of `writeJSON.go` is as follows:

```
func main() {
    myRecord := Record{
        Name:     "Mihalis",
        Surname: "Tsoukalos",
        Tel:      []Telephone{Telephone{Mobile: true, Number: "1234-567"}, Telephone{Mobile: true, Number: "1234-abcd"}, Telephone{Mobile: false, Number: "abcc-567"}},
    },
    saveToJSON(os.Stdout, myRecord)
}
```

The previous code is all about defining a structure variable that holds the data that we want to save in the JSON format using the `saveToJSON()` function. As we are using `os.Stdout`, the data will be printed on the screen instead of being saved into a file.

Executing `writeJSON.go` will generate the following output:

```
$ go run writeJSON.go
{"Name": "Mihalis", "Surname": "Tsoukalos", "Tel": [{"Mobile": true, "Number": "1234-567"}, {"Mobile": true, "Number": "1234-abcd"}, {"Mobile": false, "Number": "abcc-567"}]}
```

Using Marshal() and Unmarshal()

In this subsection, you will see how to use the `Marshal()` and `Unmarshal()` methods in order to implement the functionality of `readJSON.go` and `writeJSON.go`. The Go code that illustrates the `Marshal()` and `Unmarshal()` functions can be found in `mUJSON.go`, and it will be presented in three parts.

The first part of `mUJSON.go` is as follows:

```
package main

import (
    "encoding/json"
    "fmt"
)

type Record struct {
    Name      string
    Surname   string
    Tel       []Telephone
}

type Telephone struct {
    Mobile bool
    Number  string
}
```

In this part of the program, we are defining two structures named `Record` and `Telephone`, which will be used for storing the data that will be put into a JSON record.

The second part of `mUJSON.go` is as follows:

```
func main() {
    myRecord := Record{
        Name:      "Mihalis",
        Surname:   "Tsoukalos",
        Tel:       []Telephone{Telephone{Mobile: true, Number: "1234-567"}, Telephone{Mobile: true, Number: "1234-abcd"}, Telephone{Mobile: false, Number: "abcc-567"}},
    }

    rec, err := json.Marshal(&myRecord)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(string(rec))
```

In this part of the utility, we define the `myRecord` variable, which holds the desired data. You can also see the use of the `json.Marshal()` function, which accepts a reference to the `myRecord` variable. Note that `json.Marshal()` requires a pointer variable that converts into the JSON format.

The last part of `mUJSON.go` contains the following code:

```
var unRec Record
err1 := json.Unmarshal(rec, &unRec)
if err1 != nil {
    fmt.Println(err1)
    return
}
fmt.Println(unRec)
```

The `json.Unmarshal()` function gets JSON input and converts it into a Go structure. As it happened with `json.Marshal()`, `json.Unmarshal()` also requires a pointer argument.

Executing `mUJSON.go` will generate the following output:

```
$ go run mUJSON.go
{"Name": "Mihalis", "Surname": "Tsoukalos", "Tel": [{"Mobile": true, "Number": "1234-567"}, {"Mobile": true, "Number": "1234-abcd"}, {"Mobile": false, "Number": "abcc-567"}]}
Mihalis Tsoukalos [{true 1234-567} {true 1234-abcd} {false abcc-567}]
```



The `encoding/json` Go package includes two interfaces named `Marshaler` and `Unmarshaler`. Each one of these interfaces requires the implementation of a single method, named `MarshalJSON()` and `UnmarshalJSON()`, respectively. Should you wish to perform any custom JSON marshalling and unmarshalling, these two interfaces will allow you to do so.

Parsing JSON data

So far, we have seen how to process structured JSON data with a format that is known in advance. This kind of data can be stored in Go structures using the methods that are already described in the previous subsections.

This subsection will tell you how to read and store **unstructured JSON data**. The critical thing to remember is that unstructured JSON data is put into Go maps instead of Go structures – this will be illustrated in `parsingJSON.go`, which will be presented in four parts.

The first part of `parsingJSON.go` is as follows:

```
package main

import (
    "encoding/json"
    "fmt"
    "io/ioutil"
    "os"
)
```

In this part, we just import the required Go packages.

The second part of `parsingJSON.go` is the following:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a filename!")
        return
    }
    filename := arguments[1]
```

The presented code reads the command-line arguments of the program and gets the first one, which is the JSON file that is going to be read.

The third part of `parsingJSON.go` is as follows:

```
fileData, err := ioutil.ReadFile(filename)
if err != nil {
    fmt.Println(err)
    return
}

var parsedData map[string]interface{}
json.Unmarshal([]byte(fileData), &parsedData)
```

The `ioutil.ReadFile()` function allows you to read a file all at once, which is what we want here.

In this part, there is also a definition of a map named `parsedData` that will hold the contents of the JSON file that was read. Each map key, which is a `string`, corresponds to a JSON property. The value of each map key is of the type `interface{}`, which can be of any type – this means that the value of a map key can also be another map.

The `json.Unmarshal()` function is used for putting the contents of the file into the `parsedData` map.



In Chapter 7, Reflection and Interfaces for All Seasons, you will learn more about interfaces and `interface{}`, as well as reflection, which allows you to dynamically learn the type of an arbitrary object, as well as information about its structure.

The last part of `parsingJSON.go` contains the following code:

```

    for key, value := range parsedData {
        fmt.Println("key:", key, "value:", value)
    }
}

```

The presented code shows that you can iterate over the map and get its contents. However, interpreting these contents is a totally different story because this depends on the structure of the data, which is not known.

The JSON file with the sample data that will be used in this subsection is called `noStr.json` and it has the following contents:

```

$ cat noStr.json
{
    "Name": "John",
    "Surname": "Doe",
    "Age": 25,
    "Parents": [
        "Jim",
        "Mary"
    ],
    "Tel": [
        {"Mobile":true,"Number":"1234-567"},
        {"Mobile":true,"Number":"1234-abcd"},
        {"Mobile":false,"Number":"abcc-567"}
    ]
}

```

Executing `parsingJSON.go` will generate the following output:

```

$ go run parsingJSON.go noStr.json
key: Tel value: [map[Mobile:true Number:1234-567] map[Mobile:true Number:1234-abcd] map[Mobile:false Number:abcc-567]]
key: Name value: John
key: Surname value: Doe
key: Age value: 25
key: Parents value: [Jim Mary]

```

Once again, you can see from the output that map keys are printed in random order.

Go and XML

Go has support for **XML**, which is a markup language similar to HTML but much more advanced than HTML.

The developed utility, which is called `rwxmL.go`, will read a JSON record from disk, make a change to it, convert it to XML, and print it on screen. Then it will convert the XML data into JSON. The related Go code will be presented in four parts.

The first part of `rwxmL.go` is as follows:

```
package main

import (
    "encoding/json"
    "encoding/xml"
    "fmt"
    "os"
)

type Record struct {
    Name      string
    Surname   string
    Tel       []Telephone
}

type Telephone struct {
    Mobile    bool
    Number   string
}
```

The second part of `rwxmL.go` is shown in the following Go code:

```
func loadFromJSON(filename string, key interface{}) error {
    in, err := os.Open(filename)
    if err != nil {
        return err
    }

    decodeJSON := json.NewDecoder(in)
    err = decodeJSON.Decode(key)
    if err != nil {
        return err
    }
    in.Close()
    return nil
}
```

The third part of `rwXML.go` contains the following Go code:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a filename!")
        return
    }

    filename := arguments[1]

    var myRecord Record
    err := loadFromJSON(filename, &myRecord)
    if err == nil {
        fmt.Println("JSON:", myRecord)
    } else {
        fmt.Println(err)
    }

    myRecord.Name = "Dimitris"

    xmlData, _ := xml.MarshalIndent(myRecord, "", "    ")
    xmlData = []byte(xml.Header + string(xmlData))
    fmt.Println("\nxmlData:", string(xmlData))
```

After we read the input file and convert it into JSON, we put its data into a Go structure. Then, we make a change to the data of that structure (`myRecord`). After that, we convert that data into the XML format using the `MarshalIndent()` function and add a header using `xml.Header`.

The `MarshalIndent()` function, which can also be used with JSON data, works like `Marshal()`, but each XML element begins with a new line and is indented according to its nesting depth. This mainly has to do with the presentation of the XML data, not the values.

The last part of `rwXML.go` is the following:

```
data := &Record{}
err = xml.Unmarshal(xmlData, data)
if nil != err {
    fmt.Println("Unmarshalling from XML", err)
    return
}

result, err := json.Marshal(data)
if nil != err {
    fmt.Println("Error marshalling to JSON", err)
    return
}

_ = json.Unmarshal([]byte(result), &myRecord)
```

```
|     fmt.Println("\nJSON:", myRecord)
| }
```

In this part of the program, we convert the XML data into JSON using `Marshal()` and `Unmarshal()` and print it on screen.

Executing `rwXML.go` will generate the following output:

```
$ go run rwXML.go readMe.json
JSON: {Mihalis Tsoukalos [{true 1234-567} {true 1234-abcd} {false abcc-567}]}

xmlData: <?xml version="1.0" encoding="UTF-8"?>
<Record>
    <Name>Dimitris</Name>
    <Surname>Tsoukalos</Surname>
    <Tel>
        <Mobile>true</Mobile>
        <Number>1234-567</Number>
    </Tel>
    <Tel>
        <Mobile>true</Mobile>
        <Number>1234-abcd</Number>
    </Tel>
    <Tel>
        <Mobile>false</Mobile>
        <Number>abcc-567</Number>
    </Tel>
</Record>

JSON: {Dimitris Tsoukalos [{true 1234-567} {true 1234-abcd} {false abcc-567}]}
```

Reading an XML file

In this subsection, you will learn how to read an XML file from disk and store it into a Go structure. The name of the program is `readXML.go` and it will be presented in three parts. The first part of `readXML.go` is as follows:

```
package main

import (
    "encoding/xml"
    "fmt"
    "os"
)

type Record struct {
    Name      string
    Surname   string
    Tel       []Telephone
}

type Telephone struct {
    Mobile    bool
    Number   string
}
```

The second part of `readXML.go` is the following:

```
func loadFromXML(filename string, key interface{}) error {
    in, err := os.Open(filename)
    if err != nil {
        return err
    }

    decodeXML := xml.NewDecoder(in)
    err = decodeXML.Decode(key)
    if err != nil {
        return err
    }
    in.Close()
    return nil
}
```

The presented process is very similar to the way that you read a JSON file from disk.

The last part of `readXML.go` is as follows:

```

func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a filename!")
        return
    }

    filename := arguments[1]

    var myRecord Record
    err := loadFromXML(filename, &myRecord)

    if err == nil {
        fmt.Println("XML:", myRecord)
    } else {
        fmt.Println(err)
    }
}

```

Executing `readXML.go` will generate the following output:

```

$ go run readXML.go data.xml
XML: {Dimitris Tsoukalos [{true 1234-567} {true 1234-abcd} {false abcc-567}]}

```

The contents of `data.xml` are the following:

```

$ cat data.xml
xmlData: <?xml version="1.0" encoding="UTF-8"?>
<Record>
    <Name>Dimitris</Name>
    <Surname>Tsoukalos</Surname>
    <Tel>
        <Mobile>true</Mobile>
        <Number>1234-567</Number>
    </Tel>
    <Tel>
        <Mobile>true</Mobile>
        <Number>1234-abcd</Number>
    </Tel>
    <Tel>
        <Mobile>false</Mobile>
        <Number>abcc-567</Number>
    </Tel>
</Record>

```

Customizing XML output

In this subsection, you will learn how to modify and customize the generated XML output. The name of the utility, which will be presented in three parts, is `modXML.go`. Note that the data that is going to be converted into XML is hardcoded in the program for reasons of simplicity.

The first part of `modXML.go` is as follows:

```
package main

import (
    "encoding/xml"
    "fmt"
    "os"
)

func main() {
    type Address struct {
        City, Country string
    }
    type Employee struct {
        XMLName     xml.Name `xml:"employee"`
        Id          int      `xml:"id,attr"`
        FirstName   string   `xml:"name>first"`
        LastName    string   `xml:"name>last"`
        Initials   string   `xml:"name>initials"`
        Height     float32  `xml:"height,omitempty"`
        Address
        Comment string   `xml:",comment"`
    }
}
```

This is where the structure for the XML data is defined. However, there is additional information regarding the name and the type of the XML elements. The `XMLName` field provides the name of the XML record, which in this case will be `employee`.

A field with the tag `”,comment”` is a comment and it is formatted as such in the output. A field with the tag `”,attr”` appears as an attribute to the provided field name (which is `id` in this case) in the output. The `”name>first”` notation tells Go to embed the `first` tag inside a tag called `name`.

Lastly, a field with the `”omitempty”` option is omitted from the output if it is empty. An empty value is any of 0, false, nil pointer, or interface, and any array, slice, map, or string with a length of zero.

The second part of `modXML.go` is as follows:

```
| r := &Employee{Id: 7, FirstName: "Mihalis", LastName: "Tsoukalos", Initials: "MIT"}  
| r.Comment = "Technical Writer + DevOps"  
| r.Address = Address{"SomeWhere 12", "12312, Greece"}
```

Here we define and initialize an employee structure.

The last part of `modXML.go` is shown in the following Go code:

```
| output, err := xml.MarshalIndent(r, " ", " ")  
| if err != nil {  
|     fmt.Println("Error:", err)  
| }  
| output = []byte(xml.Header + string(output))  
| os.Stdout.Write(output)  
| os.Stdout.Write([]byte("\n"))  
| }
```

Executing `modXML.go` will generate the following output:

```
$ go run modXML.go  
<?xml version="1.0" encoding="UTF-8"?>  
<employee id="7">  
    <name>  
        <first>Mihalis</first>  
        <last>Tsoukalos</last>  
        <initials>MIT</initials>  
    </name>  
    <City>SomeWhere 12</City>  
    <Country>12312, Greece</Country>  
    <!--Technical Writer + DevOps-->  
</employee>
```

Go and the YAML format

YAML Ain't Markup Language (YAML) is another very popular text format. Although the standard Go library offers no support for the YAML format, you can look at <https://github.com/go-yaml/yaml> for a Go package that offers YAML support for Go.



*The YAML format is supported by the `Viper` package, which is illustrated in [Chapter 8](#), *Telling a UNIX System What to Do*. If you want to learn more about how `Viper` parses YAML files, you can look at the `Viper` Go code at <https://github.com/spf13/viper>.*

Additional resources

Take a look at the following resources:

- Read the documentation of the `regexp` standard Go package, which can be found at <https://golang.org/pkg/regexp/>.
- Visit the main page of the `grep(1)` utility and find out how it supports regular expressions.
- You can find more information about the `math/big` Go package at [http://golang.org/pkg/math/big/](https://golang.org/pkg/math/big/).
- You can find more information about YAML at <https://yaml.org/>.
- You might find it interesting to look at the `sync.Map` type explained at <https://golang.org/pkg/sync/>.
- Please have a look at the documentation of the `unicode` standard Go package at <https://golang.org/pkg/unicode/>.
- Although you might find it hard at first, start reading *The Go Programming Language Specification* at <https://golang.org/ref/spec>.

Exercises and web links

- Try to write a Go program that prints the invalid part or parts of an IPv4 address.
- Can you state the differences between `make` and `new` without looking at the chapter text?
- Using the code of `findIPv4.go`, write a Go program that prints the most popular IPv4 addresses found in a log file without processing the output with any UNIX utilities.
- Develop a Go program that finds the IPv4 addresses in a log file that generated a 404 HTML error message.
- Read a JSON file with 10 integer values, store it in a `struct` variable, increment each integer value by one, and write the updated JSON entry on disk. Now, do the same for an XML file.
- Develop a Go program that finds all the IPv4 addresses of a log file that downloaded ZIP files.
- Using the `math/big` standard Go package, write a Go program that calculates square roots with high precision – choose the algorithm on your own.
- Write a Go utility that finds a given date and time format in its input and returns just the time part of it.
- Do you remember the differences between a character, a byte, and a rune?
- Develop a Go utility that uses a regular expression in order to match integers from 200 to 400.
- Try to improve `keyValue.go` by adding logging to it.

Summary

In this chapter, we talked about many handy Go features, including creating and using structures, tuples, strings, and runes, and the functionality of the `unicode` standard Go package. Additionally, you learned about pattern matching, regular expressions, the processing of JSON and XML files, the `switch` statement, and the `strings` standard Go package.

Finally, we developed a key-value store in Go and you learned how to use the types of the `math/big` package to calculate Pi with the desired accuracy.

In the next chapter, you will learn how you can group and manipulate data using more advanced arrangements, such as binary trees, linked lists, doubly linked lists, queues, stacks, and hash tables. You will also explore the structures that can be found in the `container` standard Go package, how to perform matrix operations in Go, and how to verify Sudoku puzzles. The last topic of the next chapter will be random numbers and generating difficult-to-guess strings that can potentially be used as secure passwords.

How to Enhance Go Code with Data Structures

In the previous chapter, we discussed composite data types, which are constructed using the `struct` keyword, and JSON and XML processing in Go, as well as topics such as regular expressions, pattern matching, tuples, runes, strings, and the `unicode` and `strings` standard Go packages. Finally, we developed a simple key-value store in Go.

There are times, however, when the structures offered by a programming language will not fit a particular problem. In such cases, you will need to create your own data structures to store, search, and receive your data in explicit and specialized ways.

Consequently, this chapter is all about developing and using many famous data structures in Go, including **binary trees**, **linked lists**, **hash tables**, **stacks**, and **queues**, and learning about their advantages. As nothing describes a data structure better than an image, you will see many explanatory figures in this chapter.

The last parts of the chapter will talk about verifying Sudoku puzzles and performing calculations with matrices using slices.

In this chapter, you will learn about the following topics:

- **Graphs and nodes**
- Measuring the complexity of an algorithm
- Binary trees
- Hash tables
- Linked lists
- Doubly linked lists
- Working with queues in Go
- Stacks

- The data structures offered by the `container` standard Go package
- Performing matrix calculations
- Working with Sudoku puzzles
- Generating random numbers in Go
- Building random strings that can be used as difficult-to-crack passwords

About graphs and nodes

A graph ($G(V, E)$) is a finite, nonempty set of vertices (V) (or nodes) and a set of edges (E). There are two main types of graphs: **cyclic graphs** and **acyclic graphs**. A cyclic graph is a graph where all or a number of its vertices are connected in a closed chain. In acyclic graphs, there are not any closed chains.

A **directed graph** is a graph with edges that have a direction associated with them, while a **directed acyclic graph** is a directed graph with no cycles in it.



As a node may contain any kind of information, nodes are usually implemented using Go structures due to their versatility.

Algorithm complexity

The efficiency of an algorithm is judged by its computational complexity, which mostly has to do with the number of times the algorithm needs to access its input data to do its job. The **big O notation** is used in computer science for describing the complexity of an algorithm. Thus, an $O(n)$ algorithm, which needs to access its input only once, is considered better than an $O(n^2)$ algorithm, which is better than an $O(n^3)$ algorithm, and so on. The worst algorithms, however, are the ones with an $O(n!)$ running time, which makes them almost unusable for inputs with more than 300 elements.

Lastly, most Go lookup operations in built-in types, such as finding the value of a map key or accessing an array element, have a constant time, which is represented by $O(1)$. This means that built-in types are faster than custom types, and that you should generally favor using them, unless you want full control over what is going on behind the scenes.

Furthermore, not all data structures are created equal. Generally speaking, array operations are faster than map operations, which is the price you have to pay for the versatility of a map.



Although every algorithm has its drawbacks, if you do not have lots of data, the algorithm is not really important as long as it performs the desired job accurately.

Binary trees in Go

A binary tree is a data structure where underneath each node there exist *at most* two other nodes. This means that a node can be connected to one, two, or no other nodes. The **root of a tree** is the first node of the tree. The **depth of a tree**, which is also called the **height of a tree**, is defined as the longest path from the root to a node, whereas the **depth of a node** is the number of edges from the node to the root of the tree. A leaf is a node with no children.

A tree is considered **balanced** when the longest length from the root node to a leaf is at most one more than the shortest such length. An **unbalanced tree** is a tree that is not balanced. Balancing a tree might be a difficult and slow operation, so it is better to keep your tree balanced from the beginning rather than trying to balance it after you have created it, especially when your tree has a large number of nodes.

The next figure shows an unbalanced binary tree. Its root node is J, whereas nodes A, G, W, and D are leaves.

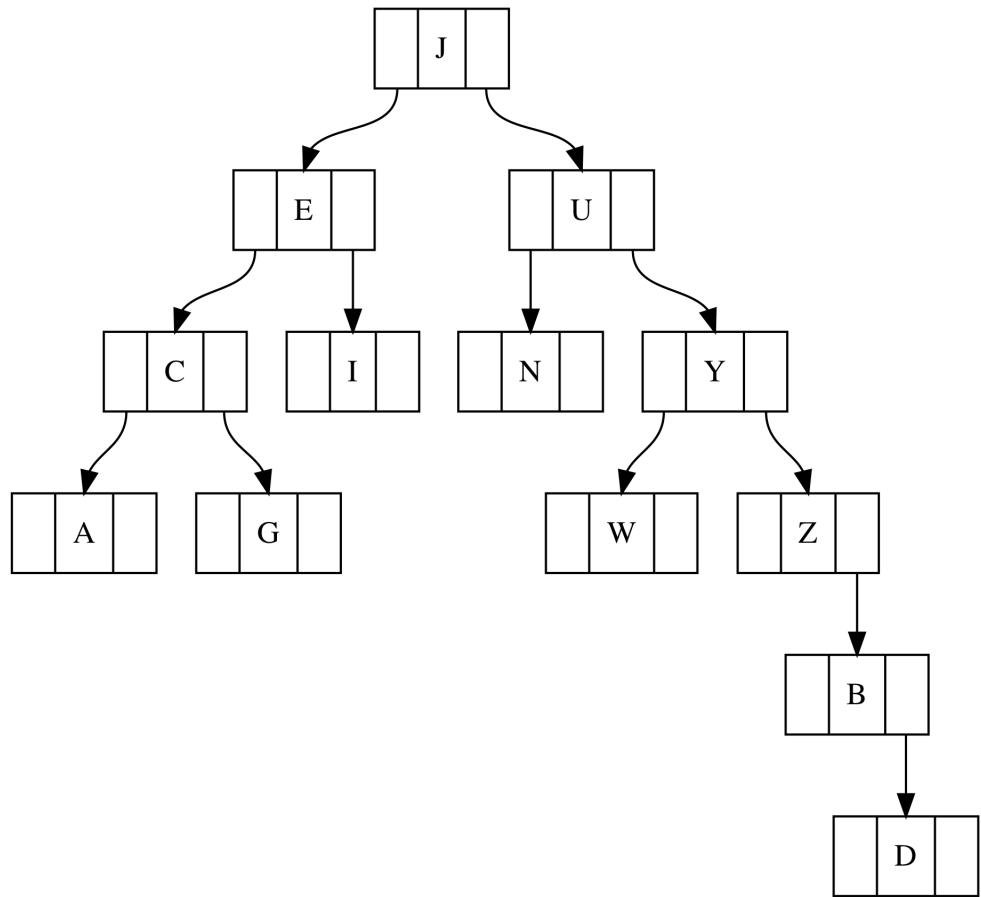


Figure 5.1: An unbalanced binary tree

Implementing a binary tree in Go

This section will illustrate how to implement a binary tree in Go using the source code found in `binTree.go` as an example. The contents of `binTree.go` will be presented in five parts. The first part is next:

```
package main

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

type Tree struct {
    Left *Tree
    Value int
    Right *Tree
}
```

What you see here is the definition of the node of the tree using a Go structure. The `math/rand` package is used for populating the tree with random numbers, as we do not have any real data.

The second code portion from `binTree.go` comes with the next Go code:

```
func traverse(t *Tree) {
    if t == nil {
        return
    }
    traverse(t.Left)
    fmt.Print(t.Value, " ")
    traverse(t.Right)
}
```

The `traverse()` function reveals how you can visit all of the nodes of a binary tree using recursion.

The third code segment is as follows:

```
func create(n int) *Tree {
    var t *Tree
    rand.Seed(time.Now().Unix())
    for i := 0; i < 2*n; i++ {
        temp := rand.Intn(n * 2)
        t = insert(t, temp)
```

```
    }
    return t
}
```

The `create()` function is only used for populating the binary tree with random integers.

The fourth part of the program is next:

```
func insert(t *Tree, v int) *Tree {
    if t == nil {
        return &Tree{nil, v, nil}
    }
    if v == t.Value {
        return t
    }
    if v < t.Value {
        t.Left = insert(t.Left, v)
        return t
    }
    t.Right = insert(t.Right, v)
    return t
}
```

The `insert()` function does many important things using `if` statements. The first `if` statement checks whether we are dealing with an empty tree or not. If it is indeed an empty tree, then the new node will be the root of the tree and will be created as `&Tree{nil, v, nil}`.

The second `if` statement determines whether the value you are trying to insert already exists in the binary tree or not. If it exists, the function returns without doing anything else.

The third `if` statement determines whether the value you are trying to insert will go on the left or on the right of the node that is currently being examined and acts accordingly. Please note that the presented implementation creates unbalanced binary trees.

The last part of `binTree.go` contains the following Go code:

```
func main() {
    tree := create(10)
    fmt.Println("The value of the root of the tree is",
tree.Value)
    traverse(tree)
    fmt.Println()
    tree = insert(tree, -10)
```

```
    tree = insert(tree, -2)
    traverse(tree)
    fmt.Println()
    fmt.Println("The value of the root of the tree is",
tree.Value)
}
```

Executing `binTree.go` will generate the next kind of output:

```
$ go run binTree.go
The value of the root of the tree is 18
0 3 4 5 7 8 9 10 11 14 16 17 18 19
-10 -2 0 3 4 5 7 8 9 10 11 14 16 17 18 19
The value of the root of the tree is 18
```

Advantages of binary trees

You cannot beat a tree when you need to represent hierarchical data. For that reason, trees are extensively used when the compiler of a programming language parses a computer program.

Additionally, trees are *ordered* by design, which means that you do not have to make any special effort to order them; putting an element into its correct place keeps them ordered. However, deleting an element from a tree is not always trivial because of the way that trees are constructed.

If a binary tree is balanced, its search, insert, and delete operations take about $\log(n)$ steps, where n is the total number of elements that the tree holds. Additionally, the height of a balanced binary tree is approximately $\log_2(n)$, which means that a balanced tree with 10,000 elements has a height of about 14, and that is remarkably small.

Similarly, the height of a balanced tree with 100,000 elements will be about 17, and the height of a balanced tree with 1,000,000 elements will be about 20. In other words, putting a significantly large number of elements into a balanced binary tree does not change the speed of the tree in an extreme way. Stated differently, you can reach any node of a tree with 1,000,000 nodes in less than 20 steps!

A major disadvantage of binary trees is that the shape of the tree depends on the order in which its elements were inserted. If the keys of a tree are long and complex, then inserting or searching for an element might be slow due to the large number of comparisons required. Finally, if a tree is not balanced, then the performance of the tree will be unpredictable.

Although you can create a linked list or an array faster than a binary tree, the flexibility that a binary tree offers in search operations might be worth the extra overhead and maintenance.

 *When searching for an element on a binary tree, you check whether the value of the element that you are looking for is bigger or smaller than the value of the current node*

and use that decision to choose which part of the tree you will go down next. Doing this saves a lot of time.

Hash tables in Go

Strictly speaking, a hash table is a data structure that stores one or more key-value pairs and uses a **hash function** to compute an index into an array of buckets or slots, from which the correct value can be discovered. Ideally, the hash function should assign each key to a unique bucket provided that you have the required number of buckets, which is usually the case.

A good hash function must be able to produce a uniform distribution of the hash values, because it is inefficient to have unused buckets or big differences in the cardinalities of the buckets. Additionally, the hash function should work consistently and output the same hash value for identical keys. Otherwise, it would be impossible to locate the information you want.

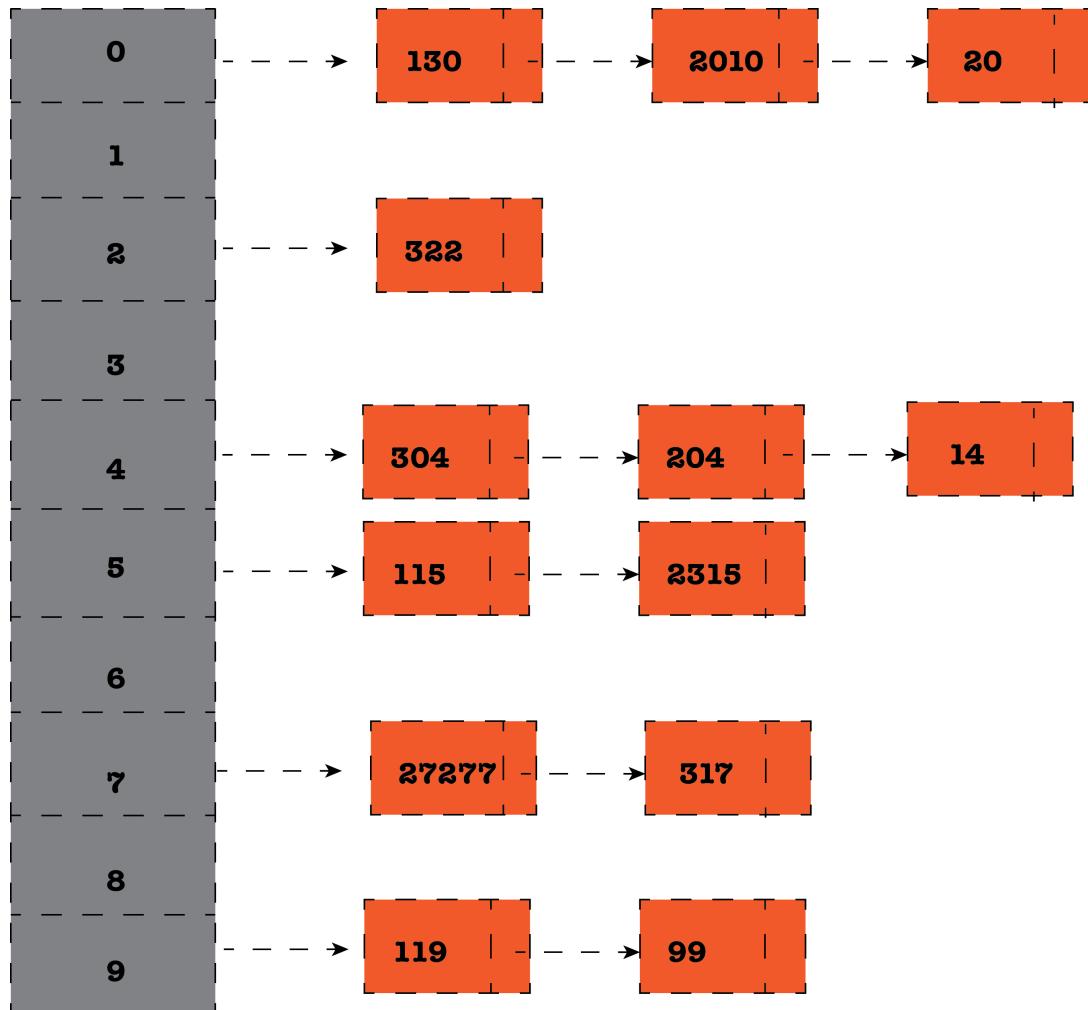


Figure 5.2: A hash table with 10 buckets

Implementing a hash table in Go

The Go code of `hashTable.go`, which will be presented in five parts, will help to clarify many things about hash tables.

The first part of `hashTable.go` is as follows:

```
package main

import (
    "fmt"
)

const SIZE = 15
type Node struct {
    Value int
    Next  *Node
}
```

In this part, you can see the definition of the node of the hash table, which, as expected, is defined using a Go structure. The `SIZE` constant variable holds the number of buckets of the hash table.

The second segment from `hashTable.go` is shown in the following Go code:

```
type HashTable struct {
    Table map[int]*Node
    Size  int
}

func hashFunction(i, size int) int {
    return (i % size)
}
```

In this code segment, you can see the implementation of the hash function used in this particular hash. The `hashFunction()` uses the **modulo operator**. The main reason for choosing the modulo operator is because this particular hash table has to cope with integer values. If you were dealing with strings or floating-point numbers, then you would use a different logic in your hash function.

The actual hash is stored in a `HashTable` structure that has two fields. The first field is a map that associates an integer with a linked list (`*Node`), and the second field is the size of the hash table. As a result, this hash table will have as many linked lists as the number of its buckets. This also means that the nodes of each bucket of the hash table will be stored in linked lists. You will learn more about linked lists in a little while.

The third code portion of `hashTable.go` is as follows:

```
func insert(hash *HashTable, value int) int {
    index := hashFunction(value, hash.Size)
    element := Node{Value: value, Next: hash.Table[index]}
    hash.Table[index] = &element
    return index
}
```

The `insert()` function is called for inserting elements into the hash table. Note that the current implementation of the `insert()` function does not check for duplicate values.

The fourth part of `hashTable.go` is next:

```
func traverse(hash *HashTable) {
    for k := range hash.Table {
        if hash.Table[k] != nil {
            t := hash.Table[k]
            for t != nil {
                fmt.Printf("%d -> ", t.value)
                t = t.Next
            }
            fmt.Println()
        }
    }
}
```

The `traverse()` function is used for printing all of the values in the hash table. The function visits each of the linked lists of the hash table and prints the stored values, linked list by linked list.

The last code portion of `hashTable.go` is as follows:

```
func main() {
    table := make(map[int]*Node, SIZE)
    hash := &HashTable{Table: table, Size: SIZE}
    fmt.Println("Number of spaces:", hash.Size)
    for i := 0; i < 120; i++ {
        insert(hash, i)
    }
}
```

```
    }
    traverse(hash)
}
```

In this part of the code, you create a new hash table named `hash` using the `table` variable, which is a map that holds the buckets of the hash table. As you already know, the slots of a hash table are implemented using linked lists.

The main reason for using a map to hold the linked lists of a hash table instead of a slice or an array is that the keys of a slice or an array can only be positive integers, while the keys of a map can be almost anything you need.

Executing `hashTable.go` will produce the following output:

```
$ go run hashTable.go
Number of spaces: 15
105 -> 90 -> 75 -> 60 -> 45 -> 30 -> 15 -> 0 ->
110 -> 95 -> 80 -> 65 -> 50 -> 35 -> 20 -> 5 ->
114 -> 99 -> 84 -> 69 -> 54 -> 39 -> 24 -> 9 ->
118 -> 103 -> 88 -> 73 -> 58 -> 43 -> 28 -> 13 ->
119 -> 104 -> 89 -> 74 -> 59 -> 44 -> 29 -> 14 ->
108 -> 93 -> 78 -> 63 -> 48 -> 33 -> 18 -> 3 ->
112 -> 97 -> 82 -> 67 -> 52 -> 37 -> 22 -> 7 ->
113 -> 98 -> 83 -> 68 -> 53 -> 38 -> 23 -> 8 ->
116 -> 101 -> 86 -> 71 -> 56 -> 41 -> 26 -> 11 ->
106 -> 91 -> 76 -> 61 -> 46 -> 31 -> 16 -> 1 ->
107 -> 92 -> 77 -> 62 -> 47 -> 32 -> 17 -> 2 ->
109 -> 94 -> 79 -> 64 -> 49 -> 34 -> 19 -> 4 ->
117 -> 102 -> 87 -> 72 -> 57 -> 42 -> 27 -> 12 ->
111 -> 96 -> 81 -> 66 -> 51 -> 36 -> 21 -> 6 ->
115 -> 100 -> 85 -> 70 -> 55 -> 40 -> 25 -> 10 ->
```

This particular hash table is perfectly balanced because it has to deal with continuous numbers that are placed in a slot according to the results of the modulo operator. Real-world problems might not generate such convenient results!



*The remainder of a **Euclidean division** between two natural numbers, a and b , can be calculated according to the $a = bq + r$ formula, where q is the quotient and r is the remainder. The values allowed for the remainder can be between 0 and $b-1$, which are the possible results of the modulo operator.*

Note that if you execute `hashTable.go` several times, you will most likely get an output where the lines are in a different order, because the way that Go

outputs the key-value pairs of a map is deliberately totally random, and therefore cannot be relied upon.

Implementing the lookup functionality

In this section, you are going to see an implantation of the `lookup()` function that allows you to determine whether a given element already exists in the hash table or not. The code of the `lookup()` function is based on that of the `traverse()` function, as follows:

```
func lookup(hash *HashTable, value int) bool {
    index := hashFunction(value, hash.Size)
    if hash.Table[index] != nil {
        t := hash.Table[index]
        for t != nil {
            if t.Value == value {
                return true
            }
            t = t.Next
        }
    }
    return false
}
```

You can find the preceding code in the `hashTableLookup.go` source file. Executing `hashTableLookup.go` will create the following output:

```
$ go run hashTableLookup.go
120 is not in the hash table!
121 is not in the hash table!
122 is not in the hash table!
123 is not in the hash table!
124 is not in the hash table!
```

The preceding output means that the `lookup()` function does its job pretty well.

Advantages of hash tables

If you are thinking that hash tables are not that useful, handy, or smart, consider the following: when a hash table has n keys and k buckets, the search speed for the n keys goes from $O(n)$ for a linear search down to $O(n/k)$. Although the improvement might look small, for a hash array with only 20 slots, the search time will be reduced by 20 times! This makes hash tables perfect for applications such as dictionaries or any other analogous application where you have to search large amounts of data.

Linked lists in Go

A linked list is a data structure with a finite set of elements where each element uses at least two memory locations: one for storing the actual data and the other for storing a pointer that links the current element to the next one, thus creating a sequence of elements that construct the linked list.

The first element of a linked list is called the **head**, whereas the last element is often called the **tail**. The first thing that you should do when defining a linked list is to keep the head of the list in a separate variable because the head is the only thing that you have to access the entire linked list. Note that if you lose the pointer to that first node of a singly linked list, there is no way to find it again.

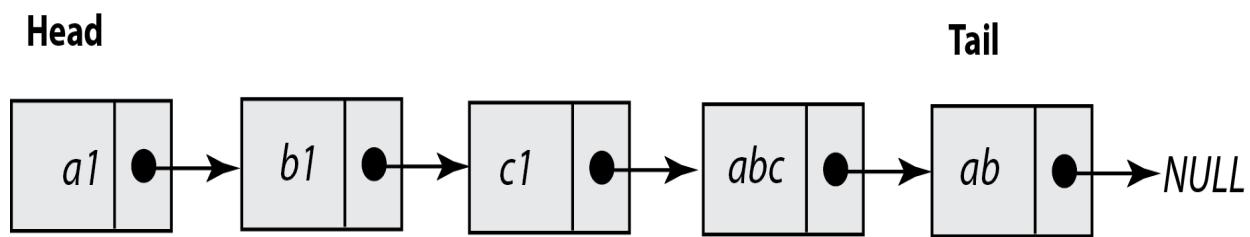


Figure 5.3: A linked list with five nodes

The next figure shows you how to remove an existing node from a linked list in order to better understand the steps that are involved in the process. The main thing that you will need to do is to arrange the pointer to the left node of the node that you are removing in order to point to the right node of the node that is being removed.

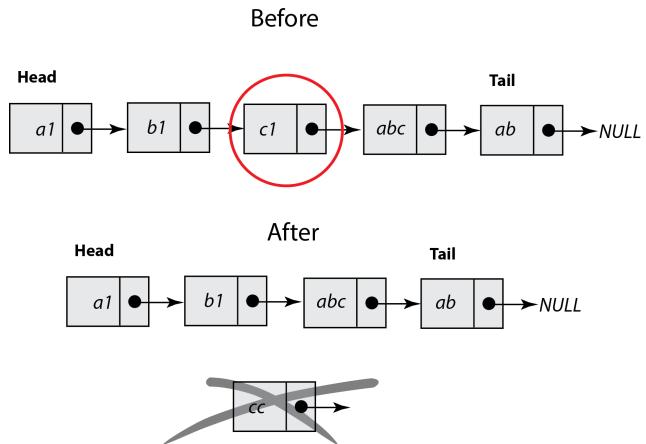


Figure 5.4: Removing a node from a linked list

The linked list implementation that follows is relatively simple and will not include the delete node functionality, which is left as an exercise for you to tackle.

Implementing a linked list in Go

The Go source file for the implementation of the linked list is called `linkedList.go`, and it will be presented in five parts.

The first code segment of `linkedList.go` is as follows:

```
package main

import (
    "fmt"
)

type Node struct {
    Value int
    Next  *Node
}

var root = new(Node)
```

In this part of the program, you define the `Node` structure type that will be used for the nodes of the linked list, as well as the `root` global variable that holds the first element of the linked list, which will be accessible everywhere in the code. Bear in mind that although using global variables is generally fine for smaller programs and example code, it might create bugs in larger programs.

The second part of `linkedList.go` is shown in the following Go code:

```
func addNode(t *Node, v int) int {
    if root == nil {
        t = &Node{v, nil}
        root = t
        return 0
    }

    if v == t.Value {
        fmt.Println("Node already exists:", v)
        return -1
    }

    if t.Next == nil {
        t.Next = &Node{v, nil}
        return -2
    }
}
```

```
|     return addNode(t.Next, v)
| }
```

Due to the way that linked lists work, they do not normally contain duplicate entries. Furthermore, new nodes are usually added at the end of a linked list when the linked list is not sorted. Thus, the `addNode()` function is used for adding new nodes to the linked list.

There are three distinct cases in the implementation that are examined using `if` statements. In the first case, you test whether you are dealing with an empty linked list or not. In the second case, you check whether the value that you want to add is already in the list. In the third case, you check whether you have reached the end of the linked list. In this case, you add a new node at the end of the list with the desired value using `t.Next = &Node{v, nil}`. If none of these conditions is true, you repeat the same process with the `addNode()` function for the next node of the linked list using `return addNode(t.Next, v)`.

The third code segment of the `linkedList.go` program contains the implementation of the `traverse()` function:

```
func traverse(t *Node) {
    if t == nil {
        fmt.Println("=> Empty list!")
        return
    }

    for t != nil {
        fmt.Printf("%d -> ", t.Value)
        t = t.Next
    }
    fmt.Println()
}
```

The fourth part of `linkedList.go` is as follows:

```
func lookupNode(t *Node, v int) bool {
    if root == nil {
        t = &Node{v, nil}
        root = t
        return false
    }

    if v == t.Value {
        return true
    }
}
```

```

    if t.Next == nil {
        return false
    }

    return lookupNode(t.Next, v)
}

func size(t *Node) int {
    if t == nil {
        fmt.Println("-> Empty list!")
        return 0
    }

    i := 0
    for t != nil {
        i++
        t = t.Next
    }
    return i
}

```

In this part, you see the implementation of two very handy functions: `lookupNode()` and `size()`. The former checks whether a given element exists in the linked list, while the latter returns the size of the linked list, which is the number of nodes in the linked list.

The logic behind the implementation of the `lookupNode()` function is easy to understand: you start accessing all of the elements of the singly linked list in order to search for the value you want. If you reach the tail of the linked list without having found the desired value, then you know that the linked list does not contain that value.

The last part of `linkedList.go` contains the implementation of the `main()` function:

```

func main() {
    fmt.Println(root)
    root = nil
    traverse(root)
    addNode(root, 1)
    addNode(root, -1)
    traverse(root)
    addNode(root, 10)
    addNode(root, 5)
    addNode(root, 45)
    addNode(root, 5)
    addNode(root, 5)
    traverse(root)
    addNode(root, 100)
    traverse(root)

    if lookupNode(root, 100) {

```

```
        fmt.Println("Node exists!")
    } else {
        fmt.Println("Node does not exist!")
    }

    if lookupNode(root, -100) {
        fmt.Println("Node exists!")
    } else {
        fmt.Println("Node does not exist!")
    }
}
```

Executing `linkedList.go` will generate the following output:

```
$ go run linkedList.go
&{0 <nil>}
-> Empty list!
1 -> -1 ->
Node already exists: 5
Node already exists: 5
1 -> -1 -> 10 -> 5 -> 45 ->
1 -> -1 -> 10 -> 5 -> 45 -> 100 ->
Node exists!
Node does not exist!
```

Advantages of linked lists

The greatest advantages of linked lists are that they are easy to understand and implement, and they are generic enough that they can be used in many different situations. This means that they can be used to model many different kinds of data, starting from single values and going up to complex data structures with many fields. Additionally, linked lists are really fast at sequential searching when used with pointers.

Linked lists not only help you to sort your data, but they can also assist you in keeping your data sorted even after inserting or deleting elements. Deleting a node from a sorted linked list is the same as in an unsorted linked list; however, inserting a new node into a sorted linked list is different because the new node has to go to the right place in order for the list to remain sorted. In practice, this means that if you have lots of data and you know that you will need to delete data all the time, using a linked list is a better choice than using a hash table or a binary tree.

Lastly, **sorted linked lists** allow you to use various optimization techniques when searching for or inserting a node. The most common technique is keeping a pointer at the center node of the sorted linked list and starting your lookups from there. This simple optimization can reduce the time of the lookup operation by half!

Doubly linked lists in Go

A doubly linked list is one where each node keeps a pointer to the previous element on the list, as well as the next element.

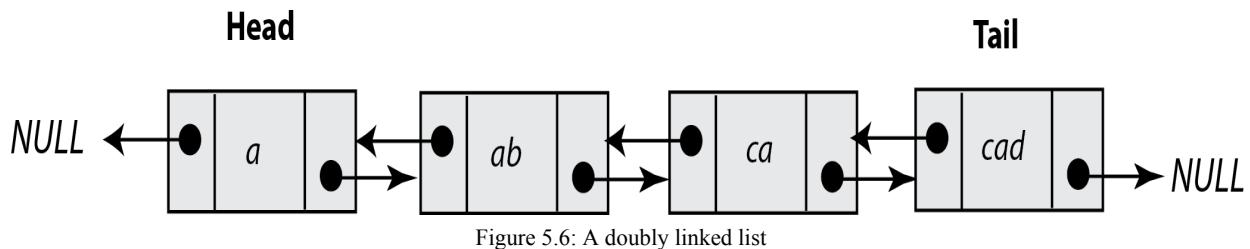


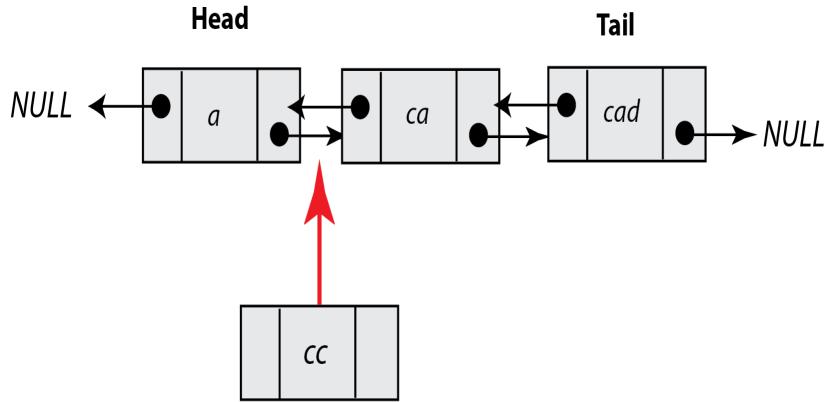
Figure 5.6: A doubly linked list

Thus, on a doubly linked list, the next link of the first node points to the second node, while its previous link points to `nil` (also called `NULL`).

Analogously, the next link of the last node points to `nil`, while its previous link points to the penultimate node of the doubly linked list.

The last figure of this chapter illustrates the addition of a node in a doubly linked list. As you can imagine, the main task that needs to be accomplished is dealing with the pointers of three nodes: the new node, the node that will be on the left of the new node, and the node that will be on the right of the new node.

Before



After

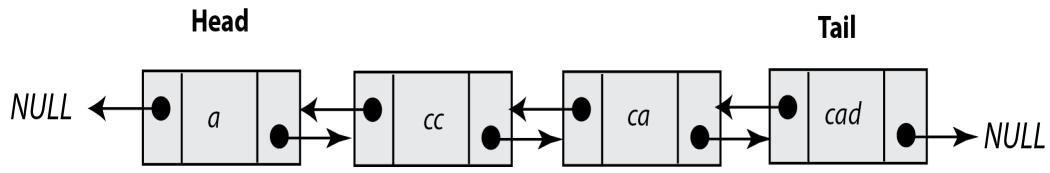


Figure 5.7: Inserting a new node into the middle of a doubly linked list

Thus, in reality, the main difference between a singly linked list and a doubly linked list is that the latter requires more housekeeping. This is the price that you will have to pay for being able to access your doubly linked list both ways.

Implementing a doubly linked list in Go

The name of the program with the Go implementation of a doubly linked list is `doublyLLList.go`, and it will be offered to you in five parts. The general idea behind a doubly linked list is the same as with a singly linked list, but you just have to do more housekeeping due to the presence of two pointers in each node of the list.

The first part of `doublyLLList.go` is as follows:

```
package main

import (
    "fmt"
)

type Node struct {
    Value    int
    Previous *Node
    Next     *Node
}
```

In this part, you can see the definition of the node of the doubly linked list using a Go structure. However, this time, the `struct` has two pointer fields for apparent reasons.

The second code portion of `doublyLLList.go` contains the following Go code:

```
func addNode(t *Node, v int) int {
    if root == nil {
        t = &Node{v, nil, nil}
        root = t
        return 0
    }

    if v == t.Value {
        fmt.Println("Node already exists:", v)
        return -1
    }

    if t.Next == nil {
        temp := t
```

```

        t.Next = &Node{v, temp, nil}
        return -2
    }

    return addNode(t.Next, v)
}

```

As happened in the case of the singly linked list, each new node is placed at the end of the current doubly linked list. However, this is not mandatory, as you can decide that you want to have a sorted doubly linked list.

The third part of `doublyLLList.go` is as follows:

```

func traverse(t *Node) {
    if t == nil {
        fmt.Println("=> Empty list!")
        return
    }

    for t != nil {
        fmt.Printf("%d -> ", t.Value)
        t = t.Next
    }
    fmt.Println()
}

func reverse(t *Node) {
    if t == nil {
        fmt.Println("=> Empty list!")
        return
    }

    temp := t
    for t != nil {
        temp = t
        t = t.Next
    }

    for temp.Previous != nil {
        fmt.Printf("%d -> ", temp.Value)
        temp = temp.Previous
    }
    fmt.Printf("%d -> ", temp.Value)
    fmt.Println()
}

```

Here you see the Go code for the `traverse()` and `reverse()` functions. The implementation of the `traverse()` function is the same as in the `linkedList.go` program. However, the logic behind the `reverse()` function is very interesting. As we do not keep a pointer to the tail of the doubly linked list, we need to go to the end of the doubly linked list before being able to access its nodes in reverse order.

Notice that Go allows you to write code such as `a, b = b, a` in order to swap the values of two variables without the need for a temporary variable.

The fourth part of `doublyLLList.go` contains the following Go code:

```
func size(t *Node) int {
    if t == nil {
        fmt.Println("=> Empty list!")
        return 0
    }

    n := 0
    for t != nil {
        n++
        t = t.Next
    }
    return n
}

func lookupNode(t *Node, v int) bool {
    if root == nil {
        return false
    }

    if v == t.Value {
        return true
    }

    if t.Next == nil {
        return false
    }

    return lookupNode(t.Next, v)
}
```

The last code segment of `doublyLLList.go` contains the following Go code:

```
var root = new(Node)

func main() {

    fmt.Println(root)
    root = nil
    traverse(root)
    addNode(root, 1)
    addNode(root, 1)
    traverse(root)
    addNode(root, 10)
    addNode(root, 5)
    addNode(root, 0)
    addNode(root, 0)
    traverse(root)
    addNode(root, 100)
    fmt.Println("Size:", size(root))
    traverse(root)
}
```

```
|     reverse(root)
| }
```

If you execute `doublyLLList.go`, you will get the following output:

```
$ go run doublyLLList.go
&{0 <nil> <nil>}
-> Empty list!
Node already exists: 1
1 ->
Node already exists: 0
1 -> 10 -> 5 -> 0 ->
Size: 5
1 -> 10 -> 5 -> 0 -> 100 ->
100 -> 0 -> 5 -> 10 -> 1 ->
```

As you can see, the `reverse()` function works just fine!

Advantages of doubly linked lists

Doubly linked lists are more versatile than singly linked lists because you can traverse them in any direction you want and also you can insert and delete elements from them more easily. Additionally, even if you lose the pointer to the head of a doubly linked list, you can still find the head node of that list. However, this versatility comes at a price: maintaining two pointers for each node. It is up to the developer to decide whether that extra complexity is justified or not. After all, your music player might be using a doubly linked list to represent your current list of songs and be able to go to the previous song as well as the next one.

Queues in Go

A queue is a special kind of linked list where each new element is inserted to the head and removed from the tail of the linked list. I do not need a figure to describe a queue; imagine going to a bank and waiting for the people that came before you to finish their transactions before you can talk to a bank teller.

The main advantage of queues is simplicity. You only need two functions to access a queue, which means that you have to worry about fewer things going wrong and you can implement a queue any way you want as long as you can offer support for those two functions.

Implementing a queue in Go

The program that will illustrate the Go implementation of a queue is called `queue.go`, and it will be presented in five parts. Note that a linked list is going to be used for the implementation of the queue. The `Push()` and `Pop()` functions are used for adding and removing nodes from the queue, respectively.

The first part of the code for `queue.go` is as follows:

```
package main

import (
    "fmt"
)

type Node struct {
    Value int
    Next  *Node
}

var size = 0
var queue = new(Node)
```

Having a variable (`size`) for keeping the number of nodes that you have on the queue is handy but not compulsory. However, the implementation presented here supports this functionality because it makes things simpler. In practice, you will probably want to keep these fields in your own structure.

The second code portion of `queue.go` contains the following Go code:

```
func Push(t *Node, v int) bool {
    if queue == nil {
        queue = &Node{v, nil}
        size++
        return true
    }

    t = &Node{v, nil}
    t.Next = queue
    queue = t
    size++
```

```
|     return true  
| }
```

This part displays the implementation of the `Push()` function, which is straightforward. If the queue is empty, then the new node will become the queue. If the queue is not empty, then you create a new node that is placed in front of the current queue. After that, the head of the queue becomes the node that was just created.

The third part of `queue.go` contains the following Go code:

```
func Pop(t *Node) (int, bool) {  
    if size == 0 {  
        return 0, false  
    }  
  
    if size == 1 {  
        queue = nil  
        size--  
        return t.Value, true  
    }  
  
    temp := t  
    for (t.Next) != nil {  
        temp = t  
        t = t.Next  
    }  
  
    v := (temp.Next).Value  
    temp.Next = nil  
  
    size--  
    return v, true  
}
```

The preceding code shows the implementation of the `Pop()` function, which removes the oldest element of the queue. If the queue is empty (`size == 0`), there is nothing to extract.

If the queue has only one node, then you extract the value of that node and the queue becomes empty. Otherwise, you extract the last element of the queue, remove the last node of the queue, and fix the required pointers before returning the desired value.

The fourth part of `queue.go` contains the following Go code:

```
func traverse(t *Node) {  
    if size == 0 {
```

```

        fmt.Println("Empty Queue!")
        return
    }
    for t != nil {
        fmt.Printf("%d -> ", t.Value)
        t = t.Next
    }
    fmt.Println()
}

```

Strictly speaking, the `traverse()` function is not necessary for the operation of a queue, but it gives you a practical way of looking at all of the nodes of the queue.

The last code segment of `queue.go` is shown in the following Go code:

```

func main() {
    queue = nil
    Push(queue, 10)
    fmt.Println("Size:", size)
    traverse(queue)

    v, b := Pop(queue)
    if b {
        fmt.Println("Pop:", v)
    }
    fmt.Println("Size:", size)

    for i := 0; i < 5; i++ {
        Push(queue, i)
    }
    traverse(queue)
    fmt.Println("Size:", size)

    v, b = Pop(queue)
    if b {
        fmt.Println("Pop:", v)
    }
    fmt.Println("Size:", size)

    v, b = Pop(queue)
    if b {
        fmt.Println("Pop:", v)
    }
    fmt.Println("Size:", size)
    traverse(queue)
}

```

Almost all of the Go code in `main()` is for checking the operation of the queue. The most important code in here is the two `if` statements, which let you know whether the `Pop()` function returned an actual value, or if the queue was empty and there is nothing to return.

Executing `queue.go` will produce the following type of output:

```
$ go run queue.go
Size: 1
10 ->
Pop: 10
Size: 0
4 -> 3 -> 2 -> 1 -> 0 ->
Size: 5
Pop: 0
Size: 4
Pop: 1
Size: 3
4 -> 3 -> 2 ->
```

Stacks in Go

A stack is a data structure that looks like a pile of plates. The last plate that goes on the top of the pile is the one that will be used first when you need to use a new plate.

Like a queue, the main advantage of a stack is its simplicity because you only have to worry about implementing two functions in order to be able to work with a stack: adding a new node to the stack and removing a node from the stack.

Implementing a stack in Go

It is now time to look at the implementation of a stack in Go. This will be illustrated in the `stack.go` source file. Once again, a linked list will be used for implementing the stack. As you know, you will need two functions: one function named `Push()` for putting things on the stack and another one named `Pop()` for removing things from the stack.

Although it is not necessary, it is useful to keep the number of elements that you have on the stack on a separate variable in order to be able to tell whether you are dealing with an empty stack or not, without having to access the linked list itself.

The source code of `stack.go` will be presented in four parts. The first part is as follows:

```
package main

import (
    "fmt"
)

type Node struct {
    Value int
    Next  *Node
}

var size = 0
var stack = new(Node)
```

The second part of `stack.go` contains the implementation of the `Push()` function:

```
func Push(v int) bool {
    if stack == nil {
        stack = &Node{v, nil}
        size = 1
        return true
    }

    temp := &Node{v, nil}
    temp.Next = stack
    stack = temp
    size++
```

```

    }
    return true
}

```

If the stack is not empty, then you create a new node (`t`), which is placed in front of the current stack. After that, this new node becomes the head of the stack. The current version of the `Push()` function always returns `true`, but if your stack does not have unlimited space, you might want to modify it and return `false` when you are about to exceed its capacity.

The third part contains the implementation of the `Pop()` function:

```

func Pop(t *Node) (int, bool) {
    if size == 0 {
        return 0, false
    }

    if size == 1 {
        size = 0
        stack = nil
        return t.Value, true
    }

    stack = stack.Next
    size--
    return t.Value, true
}

```

The fourth code segment of `stack.go` is as follows:

```

func traverse(t *Node) {
    if size == 0 {
        fmt.Println("Empty Stack!")
        return
    }

    for t != nil {
        fmt.Printf("%d -> ", t.Value)
        t = t.Next
    }
    fmt.Println()
}

```

As the stack is implemented using a linked list, it is traversed as such.

The last part of `stack.go` is shown in the following Go code:

```

func main() {
    stack = nil
    v, b := Pop(stack)
    if b {
        fmt.Print(v, " ")
    }
}

```

```

    } else {
        fmt.Println("Pop() failed!")
    }

Push(100)
traverse(stack)
Push(200)
traverse(stack)

for i := 0; i < 10; i++ {
    Push(i)
}

for i := 0; i < 15; i++ {
    v, b := Pop(stack)
    if b {
        fmt.Print(v, " ")
    } else {
        break
    }
}
fmt.Println()
traverse(stack)
}

```

As you just saw, the source code of `stack.go` is a little shorter than the Go code of `queue.go`, primarily because the idea behind a stack is simpler than the idea behind a queue.

Executing `stack.go` will generate the following type of output:

```

$ go run stack.go
Pop() failed!
100 ->
200 -> 100 ->
9 8 7 6 5 4 3 2 1 0 200 100
Empty Stack!

```



NOTE: So far, you have seen how a linked list is used in the implementation of a hash table, a queue, and a stack. These examples should help you to realize the usefulness and the importance of linked lists in programming and computer science in general.

The container package

In this section, I will explain the use of the `container` standard Go package. The `container` package supports three data structures: a **heap**, **list**, and **ring**. These data structures are implemented in `container/heap`, `container/list`, and `container/ring`, respectively.

If you are unfamiliar with rings, a ring is a **circular list**, which means that the last element of a ring points to its first element. In essence, this means that all of the nodes of a ring are equivalent and that a ring does not have a beginning and an end. As a result, each element of a ring can help you to traverse the entire ring.

The next three subsections will illustrate each one of the packages contained in the `container` package. The rational advice is that if the functionality of the `container` standard Go package suits your needs, use it; otherwise, you should implement and use your own data structures.

Using container/heap

In this subsection, you will see the functionality that the `container/heap` package offers. First of all, you should know that the `container/heap` package implements a heap, which is a tree where the value of each node of the tree is the smallest element in its subtree. Note that I am using the phrase *smallest element* instead of *minimum value* in order to make it clear that a heap does not only support numerical values.

However, as you can guess, in order to implement a heap tree in Go, you will have to develop a way to tell which of two elements is smaller than the other on your own. In such cases, Go uses **interfaces** because they allow you to define such a behavior.

This means that the `container/heap` package is more advanced than the other two packages found in `container`, and that you will have to define some things before being able to use the functionality of the `container/heap` package. Strictly speaking, the `container/heap` package requires that you implement `container/heap.Interface`, which is defined as follows:

```
type Interface interface {
    sort.Interface
    Push(x interface{}) // add x as element Len()
    Pop() interface{}  // remove and return element Len() - 1.
}
```

You will learn more about interfaces in [Chapter 7, *Reflection and Interfaces for All Seasons*](#). For now, just remember that compliance with a Go interface requires the implementation of one or more functions or other interfaces, which in this case is `sort.Interface`, as well as the `Push()` and `Pop()` functions.

`sort.Interface` requires that you implement the `Len()`, `Less()`, and `Swap()` functions, which makes perfect sense because you cannot perform any kind of sorting without being able to swap two elements, being able to calculate a value for the things that you want to sort, and being able to tell which

element between two elements is bigger than the other based on the value that you calculated previously. Although you might think that this is a lot of work, keep in mind that most of the time, the implementation of these functions is either trivial or rather simple.

Since the purpose of this section is to illustrate the use of `container/heap` and not to make your life difficult, the data type for the elements in this example will be `float32`.

The Go code of `conHeap.go` will be presented in five parts. The first part is as follows:

```
package main

import (
    "container/heap"
    "fmt"
)
type heapFloat32 []float32
```

The second part of `conHeap.go` is shown in the following Go code:

```
func (n *heapFloat32) Pop() interface{} {
    old := *n
    x := old[len(old)-1]
    new := old[0 : len(old)-1]
    *n = new
    return x
}

func (n *heapFloat32) Push(x interface{}) {
    *n = append(*n, x.(float32))
}
```

Although you define two functions named `Pop()` and `Push()` here, these two functions are used for interface compliance. In order to add and remove elements from the heap, you should call `heap.Push()` and `heap.Pop()`, respectively.

The third code segment of `conHeap.go` contains the following Go code:

```
func (n heapFloat32) Len() int {
    return len(n)
}

func (n heapFloat32) Less(a, b int) bool {
```

```

        return n[a] < n[b]
    }

func (n heapFloat32) Swap(a, b int) {
    n[a], n[b] = n[b], n[a]
}

```

This part implements the three functions needed by the `sort.Interface` interface.

The fourth part of `conHeap.go` is as follows:

```

func main() {
    myHeap := &heapFloat32{1.2, 2.1, 3.1, -100.1}
    heap.Init(myHeap)
    size := len(*myHeap)
    fmt.Printf("Heap size: %d\n", size)
    fmt.Printf("%v\n", myHeap)
}

```

The last code portion of `conHeap.go` is as follows:

```

myHeap.Push(float32(-100.2))
myHeap.Push(float32(0.2))
fmt.Printf("Heap size: %d\n", len(*myHeap))
fmt.Printf("%v\n", myHeap)
heap.Init(myHeap)
fmt.Printf("%v\n", myHeap)
}

```

In this last part of `conHeap.go`, you add two new elements to `myHeap` using `heap.Push()`. However, in order for the heap to get properly resorted, you will need to make another call to `heap.Init()`.

Executing `conHeap.go` will generate the following type of output:

```

$ go run conHeap.go
Heap size: 4
&[-100.1 1.2 3.1 2.1]
Heap size: 6
&[-100.1 1.2 3.1 2.1 -100.2 0.2]
&[-100.2 -100.1 0.2 2.1 1.2 3.1]

```

If you find it strange that the `2.1 1.2 3.1` triplet in the last line of the output is not sorted in the linear logic, remember that a heap is a tree-not a linear structure like an array or a slice.

Using container/list

This subsection will illustrate the operation of the `container/list` package using the Go code of `conList.go`, which will be presented in three parts.



NOTE: The `container/list` package implements a linked list.

The first part of `conList.go` contains the following Go code:

```
package main

import (
    "container/list"
    "fmt"
    "strconv"
)

func printList(l *list.List) {
    for t := l.Back(); t != nil; t = t.Prev() {
        fmt.Print(t.Value, " ")
    }
    fmt.Println()

    for t := l.Front(); t != nil; t = t.Next() {
        fmt.Print(t.Value, " ")
    }
    fmt.Println()
}
```

Here, you see a function named `printList()`, which allows you to print the contents of a `list.List` variable passed as a pointer. The Go code shows you how to print the elements of `list.List` starting from the first element and going to the last element, and vice versa. Usually, you will need to use only one of the two methods in your programs. The `Prev()` and `Next()` functions allow you to iterate over the elements of a list backward and forward.

The second code segment of `conList.go` is as follows:

```
func main() {
    values := list.New()

    e1 := values.PushBack("One")
    e2 := values.PushBack("Two")
```

```

values.PushFront("Three")
values.InsertBefore("Four", e1)
values.InsertAfter("Five", e2)
values.Remove(e2)
values.Remove(e2)
values.InsertAfter("FiveFive", e2)
values.PushBackList(values)

printList(values)

values.Init()

```

The `list.PushBack()` function allows you to insert an object at the back of a linked list, whereas the `list.PushFront()` function allows you to insert an object at the front of a list. The return value of both functions is the element inserted in the list.

If you want to insert a new element after a specific element, then you should use `list.InsertAfter()`. Similarly, if you want to insert a new element before a specific element, you should use the `list.InsertBefore()` function. If the element does not exist, then the list will not change. `list.PushBackList()` inserts a copy of an existing list at the end of another list, whereas the `list.PushFrontList()` function puts a copy of an existing list at the front of another list. The `list.Remove()` function removes a specific element from a list.

Note the use of the `values.Init()` function, which either empties an existing list or initializes a new list.

The last portion of the code of `conList.go` is shown in the following Go code:

```

fmt.Printf("After Init(): %v\n", values)

for i := 0; i < 20; i++ {
    values.PushFront(strconv.Itoa(i))
}

printList(values)
}

```

Here, you create a new list using a `for` loop. The `strconv.Itoa()` function converts an integer value into a string.

In summary, the use of the functions of the `container/list` package is straightforward and comes with no surprises.

Executing `conList.go` will generate the following type of output:

```
$ go run conList.go
Five One Four Three Five One Four Three
Three Four One Five Three Four One Five
After Init(): &{{0xc420074180 0xc420074180 <nil> <nil>} 0}
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

Using container/ring

This section will illustrate the use of the `container/ring` package using the Go code of `conRing.go`, which will be presented in four parts. Note that the `container/ring` package is much simpler than both `container/list` and `container/heap`, which means that it contains fewer functions than the other two packages.

The first code segment of `conRing.go` follows:

```
package main

import (
    "container/ring"
    "fmt"
)

var size int = 10
```

The `size` variable holds the size of the ring that is going to be created.

The second part of `conRing.go` contains the following Go code:

```
func main() {
    myRing := ring.New(size + 1)
    fmt.Println("Empty ring:", *myRing)

    for i := 0; i < myRing.Len()-1; i++ {
        myRing.Value = i
        myRing = myRing.Next()
    }

    myRing.Value = 2
```

Thus, a new ring is created with the help of the `ring.New()` function, which requires a single parameter: the size of the ring. The `myRing.Value = 2` statement at the end adds the value `2` to the ring. That value, however, already exists in the ring as it was added in the `for` loop. Lastly, the zero value of a ring is a ring with a single element whose value is `nil`.

The third part of `conRing.go` is shown in the following Go code:

```

    sum := 0
    myRing.Do(func(x interface{}) {
        t := x.(int)
        sum = sum + t
    })
    fmt.Println("Sum:", sum)
}

```

The `ring.Do()` function allows you to call a function for each element of a ring in chronological order. However, if that function makes any changes to the ring, then the behavior of `ring.Do()` is undefined.

The `x.(int)` statement is called **type assertion**. You will learn more about type assertions in [Chapter 7, *Reflection and Interfaces for All Seasons*](#). For now, just know that it shows that `x` is of type `int`.

The last part of the `conRing.go` program is as follows:

```

for i := 0; i < myRing.Len() + 2; i++ {
    myRing = myRing.Next()
    fmt.Print(myRing.Value, " ")
}
fmt.Println()
}

```

The only problem with rings is that you can keep calling `ring.Next()` indefinitely, so you will need to find a way to put a stop to that. In this case, this is accomplished with the help of the `ring.Len()` function. Personally, I prefer to use the `ring.Do()` function for iterating over all of the elements of a ring because it generates cleaner code, but using a `for` loop is just as good.

Executing `conRing.go` will generate the following type of output:

```

$ go run conRing.go
Empty ring: {0xc42000a080 0xc42000a1a0 <nil>}
Sum: 47
0 1 2 3 4 5 6 7 8 9 2 0 1

```

The output verifies that a ring can contain duplicate values, which means that unless you use the `ring.Len()` function, you have no safe way of knowing the size of a ring.

Generating random numbers

Random number generation is an art as well as a research area in computer science. This is because computers are purely logical machines, and it turns out that using them to generate random numbers is extremely difficult!

Go uses the `math/rand` package for generating pseudo-random numbers. It needs a **seed** to start producing the numbers. The seed is used for initializing the entire process, and it is extremely important because if you always start with the same seed, you will always get the same sequence of pseudo-random numbers. This means that everybody can regenerate that sequence, and that particular sequence will not be random after all.

The name of the utility that will help us to generate pseudo-random numbers is `randomNumbers.go`, and it will be presented in four parts. The utility takes various parameters, which are the lower and upper limits of the numbers that will be generated, as well as the amount of numbers to generate. If you use a fourth command parameter, the program will use that as the seed of the pseudo-random number generator, which will help you to regenerate the same number sequence – the main reason for doing so is for testing your code.

The first part of the utility is as follows:

```
package main

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

func random(min, max int) int {
    return rand.Intn(max-min) + min
}
```

The `random()` function does all of the job, which is generating pseudo-random numbers in the given range by calling `rand.Intn()`.

The second part of the command-line utility is as follows:

```
func main() {
    MIN := 0
    MAX := 100
    TOTAL := 100
    SEED := time.Now().Unix()

    arguments := os.Args
```

In this part, you initialize the variables that will be used in the program.

The third part of `randomNumbers.go` contains the following Go code:

```
switch len(arguments) {
case 2:
    fmt.Println("Usage: ./randomNumbers MIN MAX TOTAL")
    MIN, _ = strconv.Atoi(arguments[1])
    MAX = MIN + 100
case 3:
    fmt.Println("Usage: ./randomNumbers MIN MAX TOTAL SEED")
    MIN, _ = strconv.Atoi(arguments[1])
    MAX, _ = strconv.Atoi(arguments[2])
case 4:
    fmt.Println("Usage: ./randomNumbers MIN MAX TOTAL SEED")
    MIN, _ = strconv.Atoi(arguments[1])
    MAX, _ = strconv.Atoi(arguments[2])
    TOTAL, _ = strconv.Atoi(arguments[3])
case 5:
    MIN, _ = strconv.Atoi(arguments[1])
    MAX, _ = strconv.Atoi(arguments[2])
    TOTAL, _ = strconv.Atoi(arguments[3])
    SEED, _ = strconv.ParseInt(arguments[4], 10, 64)
default:
```

```
    fmt.Println("Using default values!")
}
```

The logic behind this `switch` block is relatively simple: depending on the number of command-line arguments you have, you use either the initial values of the missing arguments or the values given by the user. For reasons of simplicity, the `error` variables of the `strconv.Atoi()` and `strconv.ParseInt()` functions are being ignored using underscore characters. If this was a commercial program, the error variables of the `strconv.Atoi()` and `strconv.ParseInt()` functions would not have been ignored.

Lastly, the reason for using `strconv.ParseInt()` for setting a new value to the `SEED` variable is that the `rand.Seed()` function requires an `int64` parameter. The first parameter of `strconv.ParseInt()` is the string to parse, the second parameter is the base of the generated number, and the third parameter is the bit size of the generated number.

As we want to create a decimal integer that uses 64 bits, we are using `10` as the base and `64` as the bit size. Please note that had we wanted to parse an unsigned integer, we would have used the `strconv.ParseUint()` function instead.

The last part of `randomNumbers.go` is shown in the following Go code:

```
rand.Seed(SEED)
for i := 0; i < TOTAL; i++ {
    myrand := random(MIN, MAX)
    fmt.Print(myrand)
    fmt.Print(" ")
}
fmt.Println()
```



Instead of using the UNIX epoch time as the seed for the pseudo-random number generator, you can use the `/dev/random` system device. You will learn about reading from `/dev/random` in Chapter 8, Telling a UNIX System What to Do.

Executing `randomNumbers.go` will create the following type of output:

```
$ go run randomNumbers.go
Using default values!
75 69 15 75 62 67 64 8 73 1 83 92 7 34 8 70 22 58 38 8 54 34 91 65 1 50 76 5 82 61 90 10 38 40 63 6 28 51 54 49 27 52 92 76 35
$ go run randomNumbers.go 1 3 2
Usage: ./randomNumbers MIN MAX TOTAL SEED
1 1
$ go run randomNumbers.go 1 3 2
Usage: ./randomNumbers MIN MAX TOTAL SEED
2 2
$ go run randomNumbers.go 1 5 10 10
3 1 4 4 1 1 4 4 4 3
$ go run randomNumbers.go 1 5 10 10
3 1 4 4 1 1 4 4 4 3
```

If you are really interested in random number generation, you should start by reading the second volume of *The Art of Computer Programming* by Donald E. Knuth (Addison-Wesley Professional, 2011).

If you intend to use these pseudo-random numbers for security-related reasons, it is important that you use the `crypto/rand` package. This package implements a cryptographically secure pseudo-random number generator. It will be presented later in this chapter.

Generating random strings

Once you know how a computer represents single characters, it is not difficult to go from pseudo-random numbers to random strings. This section will present a technique for creating difficult-to-guess passwords based on the Go code of `randomNumbers.go`, which was presented in the previous section. The name of the Go program for this task will be `generatePassword.go`, and it is going to be presented in four parts. The utility requires just one command-line parameter, which is the length of the password that you want to generate.

The first part of `generatePassword.go` contains the following Go code:

```
package main

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

func random(min, max int) int {
    return rand.Intn(max-min) + min
}
```

The second code portion of `generatePassword.go` contains the following Go code:

```
func main() {
    MIN := 0
    MAX := 94
    var LENGTH int64 = 8

    arguments := os.Args
```

As we only want to get printable ASCII characters, we limit the range of pseudo-random numbers that can be generated. The number of printable characters in the ASCII table is 94. This means that the range of the pseudo-random numbers that the program can generate should be from 0 to 94, not including 94.

The third code segment of `generatePassword.go` is shown in the following Go code:

```
switch len(arguments) {
case 2:
    LENGTH, _ = strconv.ParseInt(os.Args[1], 10, 64)
default:
    fmt.Println("Using default values!")
}

SEED := time.Now().Unix()
rand.Seed(SEED)
```

The last part of `generatePassword.go` is as follows:

```
startChar := "!"
var i int64 = 1
for {
    myRand := random(MIN, MAX)
    newChar := string(startChar[0] + byte(myRand))
    fmt.Print(newChar)
    if i == LENGTH {
        break
    }
    i++
}
fmt.Println()
```

The `startChar` variable holds the first ASCII character that can be generated by the utility, which, in this case, is the exclamation mark, which has a decimal ASCII value of 33. Given that the program can generate pseudo-random numbers up to 94, the maximum ASCII value that can be generated is $93 + 33$, which is equal to 126, which is the ASCII value of ~. The following output shows the ASCII table with the corresponding decimal values for each character:

0 nul	1 soh	2 st	3 etx	4 eot	5 enq	6 ack	7 bel
8 bs	9 ht	10 nl	11 vt	12 np	13 cr	14 so	15 si
16 dle	17 dc1	18 dc	19 dc3	20 dc4	21 nak	22 syn	23 etb
24 can	25 em	26 su	27 esc	28 fs	29 gs	30 rs	31 us
32 sp	33 !	34 "	35 #	36 \$	37 %	38 &	39 '
40 (41)	42 *	43 +	44 ,	45 -	46 .	47 /
48 0	49 1	50 2	51 3	52 4	53 5	54 6	55 7
56 8	57 9	58 :	59 ;	60 <	61 =	62 >	63 ?
64 @	65 A	66 B	67 C	68 D	69 E	70 F	71 G
72 H	73 I	74 J	75 K	76 L	77 M	78 N	79 O
80 P	81 Q	82 R	83 S	84 T	85 U	86 V	87 W
88 X	89 Y	90 Z	91 [92 \	93]	94 ^	95 _
96 `	97 a	98 b	99 c	100 d	101 e	102 f	103 g
104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o

```
| 112  p  113  q  114  r  115  s  116  t  117  u  118  v  119  w
| 120  x  121  y  122  z  123  {  124  |  125  }  126  ~  127 del
```



Typing `man ascii` on your favorite UNIX shell will also generate the ASCII table in a readable form.

Executing `generatePassword.go` with the appropriate command-line parameters will create the following type of output:

```
$ go run generatePassword.go
Using default values!
ugs$5mv1
$ go run generatePassword.go
Using default values!
PA/8hA@?
$ go run generatePassword.go 20
HBR+=3\UA'B@ExT4QG|o
$ go run generatePassword.go 20
XLcr|R{*pX/::'t2u^T'
```

Generating secure random numbers

Should you wish to generate more secure pseudo-random numbers in Go, you should use the `crypto/rand` package, which implements a cryptographically secure pseudo-random number generator and is the subject of this section.

The use of the `crypto/rand` package will be illustrated using the Go code of `cryptoRand.go`, which is going to be presented in three parts.

The first part of `cryptoRand.go` is as follows:

```
package main

import (
    "crypto/rand"
    "encoding/base64"
    "fmt"
    "os"
    "strconv"
)

func generateBytes(n int64) ([]byte, error) {
    b := make([]byte, n)
    _, err := rand.Read(b)
    if err != nil {
        return nil, err
    }
    return b, nil
}
```

The second part of `cryptoRand.go` contains the following Go code:

```
func generatePass(s int64) (string, error) {
    b, err := generateBytes(s)
    return base64.URLEncoding.EncodeToString(b), err
}
```

The last part of `cryptoRand.go` is as follows:

```
func main() {
    var LENGTH int64 = 8
    arguments := os.Args
    switch len(arguments) {
    case 2:
        LENGTH, _ = strconv.ParseInt(os.Args[1], 10, 64)
        if LENGTH <= 0 {
            LENGTH = 8
        }
    default:
        fmt.Println("Using default values!")
    }

    myPass, err := generatePass(LENGTH)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(myPass[0:LENGTH])
}
```

Executing `cryptoRand.go` will generate the following kind of output:

```
$ go run cryptoRand.go
Using default values!
hIAFYuvW
$ go run cryptoRand.go 120
WTR15SIcjYQmaMKds01DfFturG27ovH_HZ6iAi_kOnJC88EDLdvNPcv1JjOd9DcF0r0S3q2itXZ801TNaNFpHkT-aMrsjeue6kUyHnx_EaL_vJHy9wL5RTr8
```

You can find more information about the `crypto/rand` package by visiting its documentation page at <https://golang.org/pkg/crypto/rand/>.

Performing matrix calculations

A **matrix** is an array with two dimensions. The easiest way to represent a matrix in Go is using a slice. However, if you know the dimensions of your array in advance, an array will also do the job just fine. If both dimensions of a matrix are the same, then the matrix is called a **square matrix**.

There are some rules that can tell you whether you can perform a calculation between two matrices or not. The rules are the following:

- In order to add or subtract two matrices, they should have exactly the same dimensions.
- In order to multiply matrix A with matrix B, the number of columns of matrix A should be equal to the number of rows of matrix B. Otherwise, the multiplication of matrices A and B is impossible.
- In order to divide matrix A with matrix B, two conditions must be met. Firstly, you will need to be able to calculate the inverse of matrix B and secondly, you should be able to multiply matrix A with the inverse of matrix B according to the previous rule. Only square matrices can have an inverse.

Adding and subtracting matrices

In this section, you are going to learn how to add and subtract matrices with the help of the `addMat.go` utility, which is going to be presented in three parts. The program uses slices to implement the required matrices.

The first part of `addMat.go` is as follows:

```
package main

import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
)

func random(min, max int) int {
    return rand.Intn(max-min) + min
}

func negativeMatrix(s [][]int) [][]int {
    for i, x := range s {
        for j, _ := range x {
            s[i][j] = -s[i][j]
        }
    }
    return s
}
```

The `negativeMatrix()` function gets a slice input and returns a new slice where each of the integer elements of the original size is replaced with its opposite integer. As you will soon see, the elements of the two initial matrices are generated using pseudo-random numbers, hence the need for the `random()` function.

The second part of `addMat.go` contains the following Go code:

```
func addMatrices(m1 [][]int, m2 [][]int) [][]int {
    result := make([][]int, len(m1))
    for i, x := range m1 {
        for j, _ := range x {
            result[i] = append(result[i], m1[i][j]+m2[i][j])
        }
    }
    return result
}
```

The `addMatrices()` function accesses the elements of both matrices in order to add them and create the result matrix.

The last part of `addMat.go` is as follows:

```
func main() {
    arguments := os.Args
    if len(arguments) != 3 {
        fmt.Println("Wrong number of arguments!")
        return
    }

    var row, col int
    row, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println("Need an integer: ", arguments[1])
        return
    }

    col, err = strconv.Atoi(arguments[2])
    if err != nil {
        fmt.Println("Need an integer: ", arguments[2])
        return
    }
    fmt.Printf("Using %dx%d arrays\n", row, col)

    if col <= 0 || row <= 0 {
        fmt.Println("Need positive matrix dimensions!")
        return
    }

    m1 := make([][]int, row)
    m2 := make([][]int, row)

    rand.Seed(time.Now().Unix())
    // Initialize m1 and m2 with random numbers
    for i := 0; i < row; i++ {
        for j := 0; j < col; j++ {
            m1[i] = append(m1[i], random(-1, i*j+rand.Intn(10)))
            m2[i] = append(m2[i], random(-1, i*j+rand.Intn(10)))
        }
    }
    fmt.Println("m1:", m1)
    fmt.Println("m2:", m2)

    // Adding
    r1 := addMatrices(m1, m2)
    // Subtracting
    r2 := addMatrices(m1, negativeMatrix(m2))
    fmt.Println("r1:", r1)
    fmt.Println("r2:", r2)
}
```

The `main()` function is the controller of the program. Among other things, it makes sure that the user has given the correct type of input, creates the

desired matrices, and populates them with pseudo-randomly generated numbers.

Executing `addMat.go` will create the following output:

```
$ go run addMat.go 2 3
Using 2x3 arrays
m1: [[0 -1 0] [1 1 1]]
m2: [[2 1 0] [7 4 9]]
r1: [[2 0 0] [8 5 10]]
r2: [[-2 -2 0] [-6 -3 -8]]
$ go run addMat.go 2 3
Using 2x3 arrays
m1: [[0 -1 0] [1 1 1]]
m2: [[2 1 0] [7 4 9]]
r1: [[2 0 0] [8 5 10]]
r2: [[-2 -2 0] [-6 -3 -8]]
$ go run addMat.go 3 2
Using 3x2 arrays
m1: [[0 -1] [0 0] [0 1]]
m2: [[2 1] [0 3] [1 9]]
r1: [[2 0] [0 3] [1 10]]
r2: [[-2 -2] [0 -3] [-1 -8]]
```

Multiplying matrices

As you already know, multiplying matrices is much more complex than adding or subtracting matrices, which is also shown in the number of command-line arguments that the presented utility, which is named `mulMat.go`, requires. The `mulMat.go` utility, which is going to be presented in four parts, requires four command-line arguments, which are the dimensions of the first and the second matrices, respectively.

The first part of `mulMat.go` is as follows:

```
package main

import (
    "errors"
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "time"
)

func random(min, max int) int {
    return rand.Intn(max-min) + min
}
```

The second part of `mulMat.go` contains the following Go code:

```
func multiplyMatrices(m1 [][]int, m2 [][]int) ([][]int, error) {
    if len(m1[0]) != len(m2) {
        return nil, errors.New("Cannot multiply the given matrices!")
    }

    result := make([][]int, len(m1))
    for i := 0; i < len(m1); i++ {
        result[i] = make([]int, len(m2[0]))
        for j := 0; j < len(m2[0]); j++ {
            for k := 0; k < len(m2); k++ {
                result[i][j] += m1[i][k] * m2[k][j]
            }
        }
    }
    return result, nil
}
```

The way matrices are multiplied is totally different from the way they are added, hence the way `multiplyMatrices()` is implemented. The `multiplyMatrices()`

function also returns its own custom error message in case the input matrices do not have the right dimensions that will allow them to get multiplied.

The third part of `mulMat.go` is as follows:

```
func createMatrix(row, col int) [][]int {
    r := make([][]int, row)
    for i := 0; i < row; i++ {
        for j := 0; j < col; j++ {
            r[i] = append(r[i], random(-5, i*j))
        }
    }
    return r
}

func main() {
    rand.Seed(time.Now().Unix())
    arguments := os.Args
    if len(arguments) != 5 {
        fmt.Println("Wrong number of arguments!")
        return
    }
}
```

The `createMatrix()` function creates a slice with the desired dimensions and populates it with integers that are randomly generated.

The last part of `mulMat.go` is as follows:

```
var row, col int
row, err := strconv.Atoi(arguments[1])
if err != nil {
    fmt.Println("Need an integer: ", arguments[1])
    return
}

col, err = strconv.Atoi(arguments[2])
if err != nil {
    fmt.Println("Need an integer: ", arguments[2])
    return
}

if col <= 0 || row <= 0 {
    fmt.Println("Need positive matrix dimensions!")
    return
}
fmt.Printf("m1 is a %dx%d matrix\n", row, col)
// Initialize m1 with random numbers
m1 := createMatrix(row, col)

row, err = strconv.Atoi(arguments[3])
if err != nil {
    fmt.Println("Need an integer: ", arguments[3])
    return
}
```

```

    }

    col, err = strconv.Atoi(arguments[4])
    if err != nil {
        fmt.Println("Need an integer: ", arguments[4])
        return
    }

    if col <= 0 || row <= 0 {
        fmt.Println("Need positive matrix dimensions!")
        return
    }
    fmt.Printf("m2 is a %dx%d matrix\n", row, col)
    // Initialize m2 with random numbers
    m2 := createMatrix(row, col)
    fmt.Println("m1:", m1)
    fmt.Println("m2:", m2)

    // Multiply
    r1, err := multiplyMatrices(m1, m2)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println("r1:", r1)
}
}

```

The `main()` function is the controller of the program that defines the way it operates and makes sure that the correct kind of command-line arguments were given.

Executing `mulMat.go` will create the following output:

```

$ go run mulMat.go 1 2 2 1
m1 is a 1x2 matrix
m2 is a 2x1 matrix
m1: [[-3 -1]]
m2: [[-2] [-1]]
r1: [[7]]
$ go run mulMat.go 5 2 2 1
m1 is a 5x2 matrix
m2 is a 2x1 matrix
m1: [[-1 -2] [-4 -4] [-4 -1] [-2 2] [-5 -5]]
m2: [[-5] [-3]]
r1: [[11] [32] [23] [4] [40]]
$ go run mulMat.go 1 2 3 4
m1 is a 1x2 matrix
m2 is a 3x4 matrix
m1: [[-3 -4]]
m2: [[-5 -2 -2 -3] [-1 -4 -3 -5] [-5 -2 3 3]]
Cannot multiply the given matrices!

```

Dividing matrices

In this subsection, you will learn how to divide two matrices using the Go code of `divMat.go`, which is going to be presented in five parts. The core function of `divMat.go` is called `inverseMatrix()`. What `inverseMatrix()` mainly implements is the calculation of the inverse matrix of a given matrix, which is a pretty complex task. There are ready-to-use Go packages that allow you to inverse a matrix, but I have decided to implement it from scratch.



*Not all matrices are invertible. Non-square matrices are not invertible. A **square matrix** that is not invertible is called **singular** or **degenerate** – this happens when the determinant of the square matrix is 0. Singular matrices are very rare.*

The `divMat.go` utility requires a single command-line argument that defines the dimensions of the used square matrices.

The first part of `divMat.go` is as follows:

```
package main

import (
    "errors"
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "time"
)

func random(min, max int) float64 {
    return float64(rand.Intn(max-min) + min)
}
```

This time the `random()` function generates a `float64`. The entire `divMat.go` operates using floating-point numbers, mainly because the inverse of a matrix with integer elements will most likely not be a matrix with integer elements.

The second part of `divMat.go` contains the following Go code:

```
func getMinor(A [][]float64, temp [][]float64, p int, q int, n int) {
    i := 0
```

```

j := 0

for row := 0; row < n; row++ {
    for col := 0; col < n; col++ {
        if row != p && col != q {
            temp[i][j] = A[row][col]
            j++
            if j == n-1 {
                j = 0
                i++
            }
        }
    }
}

func determinant(A [][]float64, n int) float64 {
    D := float64(0)
    if n == 1 {
        return A[0][0]
    }

    temp := createMatrix(n, n)
    sign := 1

    for f := 0; f < n; f++ {
        getCoFactor(A, temp, 0, f, n)
        D += float64(sign) * A[0][f] * determinant(temp, n-1)
        sign = -sign
    }

    return D
}

```

The `getCoFactor()` and `determinant()` functions calculate things that are necessary for inverting a matrix. If the determinant of a matrix is 0, then the matrix is singular.

The third part of `divMat.go` is as follows:

```

func adjoint(A [][]float64) ([][]float64, error) {
    N := len(A)
    adj := createMatrix(N, N)
    if N == 1 {
        adj[0][0] = 1
        return adj, nil
    }
    sign := 1
    var temp = createMatrix(N, N)

    for i := 0; i < N; i++ {
        for j := 0; j < N; j++ {
            getCoFactor(A, temp, i, j, N)
            if (i+j)%2 == 0 {
                sign = 1
            } else {
                sign = -1
            }
            adj[i][j] = sign * temp[i][j]
        }
    }
    return adj, nil
}

```

```

        }
        adj[j][i] = float64(sign) * (determinant(temp, N-1))
    }
}
return adj, nil
}

func inverseMatrix(A [][]float64) ([][]float64, error) {
    N := len(A)
    var inverse = createMatrix(N, N)
    det := determinant(A, N)
    if det == 0 {
        fmt.Println("Singular matrix, cannot find its inverse!")
        return nil, nil
    }

    adj, err := adjoint(A)
    if err != nil {
        fmt.Println(err)
        return nil, nil
    }

    fmt.Println("Determinant:", det)
    for i := 0; i < N; i++ {
        for j := 0; j < N; j++ {
            inverse[i][j] = float64(adj[i][j]) / float64(det)
        }
    }
}

return inverse, nil
}

```

The `adjoint()` function calculates the adjoint matrix of the given matrix. The `inverseMatrix()` function is what calculates the inverse of the given matrix.



The `divMat.go` program is a great example of Go code that needs to be extensively tested before putting it into production. You will learn more about testing in [Chapter 11, Code Testing, Optimization, and Profiling](#).

The fourth part of `divMat.go` contains the following Go code:

```

func multiplyMatrices(m1 [][]float64, m2 [][]float64) ([][]float64, error) {
    if len(m1[0]) != len(m2) {
        return nil, errors.New("Cannot multiply the given matrices!")
    }

    result := make([][]float64, len(m1))
    for i := 0; i < len(m1); i++ {
        result[i] = make([]float64, len(m2[0]))
        for j := 0; j < len(m2[0]); j++ {
            for k := 0; k < len(m2); k++ {
                result[i][j] += m1[i][k] * m2[k][j]
            }
        }
    }
    return result, nil
}

```

```

func createMatrix(row, col int) [][]float64 {
    r := make([][]float64, row)
    for i := 0; i < row; i++ {
        for j := 0; j < col; j++ {
            r[i] = append(r[i], random(-5, i*j))
        }
    }
    return r
}

```

The `multiplyMatrices()` function is needed because the division of a matrix with another is equal to the multiplication of the first matrix with the inverse of the second one.

The last part of `divMat.go` is as follows:

```

func main() {
    rand.Seed(time.Now().Unix())
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Wrong number of arguments!")
        return
    }

    var row int
    row, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println("Need an integer:", arguments[1])
        return
    }
    col := row
    if col <= 0 {
        fmt.Println("Need positive matrix dimensions!")
        return
    }

    m1 := createMatrix(row, col)
    m2 := createMatrix(row, col)
    fmt.Println("m1:", m1)
    fmt.Println("m2:", m2)

    inverse, err := inverseMatrix(m2)
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Println("\t\t\tPrinting inverse matrix!")
    for i := 0; i < len(inverse); i++ {
        for j := 0; j < len(inverse[0]); j++ {
            fmt.Printf("%.2f\t", inverse[i][j])
        }
        fmt.Println()
    }

    fmt.Println("\t\t\tPrinting result!")
}

```

```

    r1, err := multiplyMatrices(m1, inverse)
    if err != nil {
        fmt.Println(err)
        return
    }

    for i := 0; i < len(r1); i++ {
        for j := 0; j < len(r1[0]); j++ {
            fmt.Printf("%.3f\t", r1[i][j])
        }
        fmt.Println()
    }
}

```

Once again, the `main()` function orchestrates the flow of the program and makes the necessary checks on the user input before proceeding.

Executing `divMat.go` will create the following output:

```

$ go run divMat.go 2
m1: [[-3 -3] [-4 -4]]
m2: [[-3 -5] [-4 -1]]
Determinant: -17
    Printing inverse matrix!
0.06    -0.29
-0.24    0.18
    Printing result!
0.529    0.353
0.706    0.471
$ go run divMat.go 3
m1: [[-3 -5 -2] [-1 -4 1] [-2 -5 -1]]
m2: [[-2 -4 -5] [-1 0 -2] [-2 -2 1]]
Determinant: -22
    Printing inverse matrix!
0.18    -0.64    -0.36
-0.23    0.55    -0.05
-0.09    -0.18    0.18
    Printing result!
0.773    -0.455    0.955
0.636    -1.727    0.727
0.864    -1.273    0.773
$ go run divMat.go 2
m1: [[-3 -5] [-5 -5]]
m2: [[-5 -3] [-5 -3]]
Singular matrix, cannot find its inverse!
    Printing inverse matrix!
    Printing result!
Cannot multiply the given matrices!

```

A tip on finding out the dimensions of an array

In this section, I am going to present you with a way to find the dimensions of an array using the Go code found in `dimensions.go`. The same technique can be used for finding out the dimensions of a slice.

The Go code of `dimensions.go` is as follows:

```
package main

import (
    "fmt"
)

func main() {
    array := [12][4][7][10]float64{}
    x := len(array)
    y := len(array[0])
    z := len(array[0][0])
    w := len(array[0][0][0])
    fmt.Println("x:", x, "y:", y, "z:", z, "w:", w)
}
```

There is an array called `array` with four dimensions. The `len()` function allows you to find its dimensions when provided with the right arguments. Finding the first dimension requires a call to `len(array)`, whereas finding the second dimension requires a call to `len(array[0])`, and so on.

Executing `dimensions.go` will generate the following output:

```
$ go run dimensions.go
x: 12 y: 4 z: 7 w: 10
```

Solving Sudoku puzzles

The main purpose of this section is to help you to understand that you should always use the simplest data structure that does the job you want it to. In this case, that data structure will be a humble slice, which will be used for representing and verifying a Sudoku puzzle. Alternatively, we could have used an array because Sudoku puzzles have a predefined size.

A Sudoku is a logic-based, combinatorial, number-placement puzzle. Verifying a Sudoku puzzle means making sure that the Sudoku puzzle is correctly solved – this is a task that can be easily done by a computer program.

In order to be as generic as possible, the presented utility, which is named `sudoku.go` and will be presented in four parts, will load Sudoku puzzles from external files.

The first part of `sudoku.go` is as follows:

```
package main

import (
    "bufio"
    "errors"
    "fmt"
    "io"
    "os"
    "strconv"
    "strings"
)

func importFile(file string) ([][]int, error) {
    var err error
    var mySlice = make([][]int, 0)

    f, err := os.Open(file)
    if err != nil {
        return nil, err
    }
    defer f.Close()

    r := bufio.NewReader(f)
    for {
        line, err := r.ReadString('\n')
        fields := strings.Fields(line)
```

```

        temp := make([]int, 0)
        for _, v := range fields {
            n, err := strconv.Atoi(v)
            if err != nil {
                return nil, err
            }
            temp = append(temp, n)
        }
        if len(temp) != 0 {
            mySlice = append(mySlice, temp)
        }

        if err == io.EOF {
            break
        } else if err != nil {
            return nil, err
        }

        if len(temp) != len(mySlice[0]) {
            return nil, errors.New("Wrong number of elements!")
        }
    }
    return mySlice, nil
}

```

The `importFile()` function only checks whether it reads valid integer numbers or not. Put simply, `importFile()` will accept negative integers or integers bigger than 9 but it will not accept the value `a`, which is not a number or a float. The only other test that `importFile()` will perform is making sure that all the lines in the input file have the same number of integers. The first line of the input text file is the one that specifies the number of columns that should exist in each input line.

The second code portion of `sudoku.go` contains the following Go code:

```

func validPuzzle(sl [][]int) bool {
    for i := 0; i <= 2; i++ {
        for j := 0; j <= 2; j++ {
            iEl := i * 3
            jEl := j * 3
            mySlice := []int{0, 0, 0, 0, 0, 0, 0, 0, 0}
            for k := 0; k <= 2; k++ {
                for m := 0; m <= 2; m++ {
                    bigI := iEl + k
                    bigJ := jEl + m
                    val := sl[bigI][bigJ]
                    if val > 0 && val < 10 {
                        if mySlice[val-1] == 1 {
                            fmt.Println("Appeared 2 times:", val)
                            return false
                        } else {
                            mySlice[val-1] = 1
                        }
                    }
                }
            }
        }
    }
    return true
}

```

```
        } else {
            fmt.Println("Invalid value:", val)
            return false
        }
    }
}
```

In order to access all the elements of a Sudoku puzzle, `sudoku.go` uses four `for` loops. Although using four `for` loops for a 9x9 array might not be a performance issue, it would definitely be a problem if we had to work with much bigger arrays.

The third part of `sudoku.go` is as follows:

```

// Testing columns
for i := 0; i <= 8; i++ {
    sum := 0
    for j := 0; j <= 8; j++ {
        sum = sum + sl[i][j]
    }
    if sum != 45 {
        return false
    }
    sum = 0
}

// Testing rows
for i := 0; i <= 8; i++ {
    sum := 0
    for j := 0; j <= 8; j++ {
        sum = sum + sl[j][i]
    }
    if sum != 45 {
        return false
    }
    sum = 0
}

return true
}

```

The presented code makes sure that each column and row of the Sudoku puzzle contains all numbers.

The last part of `sudoku.go` is as follows:

```
func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Printf("usage: loadFile textFile size\n")
        return
```

```
    }

    file := arguments[1]

    mySlice, err := importFile(file)
    if err != nil {
        fmt.Println(err)
        return
    }

    if validPuzzle(mySlice) {
        fmt.Println("Correct Sudoku puzzle!")
    } else {
        fmt.Println("Incorrect Sudoku puzzle!")
    }
}
```

The `main()` function orchestrates the entire program.

Executing `sudoku.go` with various input files will generate the following kind of output:

```
$ go run sudoku.go OK.txt
Correct Sudoku puzzle!
$ go run sudoku.go noOK1.txt
Incorrect Sudoku puzzle!
```

Additional resources

You should find the following resources very useful:

- Examine the **Graphviz** utility website. This utility lets you draw graphs using its own language: <http://graphviz.org/>.
- Read the documentation page of the sub-packages of the `container` standard Go package by visiting <https://golang.org/pkg/container/>.
- Should you wish to learn more about data structures, you should read *The Design and Analysis of Computer Algorithms* by Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman (Addison-Wesley, 1974). It is an excellent book!
- You can learn more about hash functions by visiting https://en.wikipedia.org/wiki/Hash_function.
- Another really interesting book about algorithms and data structures is *Programming Pearls*, written by Jon Bentley (Addison-Wesley Professional, 1999) as well as *More Programming Pearls: Confessions of a Coder*, also by John Bentley (Addison-Wesley Professional, 1988). Reading both books will make you a better programmer.

Exercises

- Try to change the logic behind `generatePassword.go` by picking the password from a list of passwords found in a Go slice combined with the current system time or date.
- Make the necessary changes to the code of `queue.go` in order to store floating-point numbers instead of integers.
- Change the Go code of `stack.go` so that its nodes have three data fields of integer type, named `value`, `Number`, and `Seed`. Apart from the apparent changes to the definition of the `Nodestruct`, what is the main change that you will need to make to the rest of the program?
- Can you change the code in `linkedList.go` in order to keep the nodes of the linked list sorted?
- Similarly, can you change the Go code of `doublyLLlist.go` in order to keep the nodes of the list sorted? Can you develop a function for deleting existing nodes?
- Change the code of `hashTableLookup.go` so that you do not have duplicate values in your hash table. Use the `lookup()` function for that.
- Rewrite `sudoku.go` in order to use a Go map instead of a slice.
- Rewrite `sudoku.go` in order to use a linked list instead of a slice. Why is this difficult?
- Create a Go program that can calculate the powers of matrices. Are there any conditions that need to be met in order to find the power of a matrix?
- Implement the addition and subtraction of arrays with three dimensions.
- Try to represent matrices using Go structures. What are the main challenges?
- Can you modify the Go code of `generatePassword.go` in order to generate passwords that only contain uppercase letters?
- Try to change the code of `conHeap.go` in order to support a custom and more complex structure instead of just `float32` elements.
- Implement the delete node functionality that is missing from `linkedList.go`.

- Do you think that a doubly linked list would make the code of the `queue.go` program better? Try to implement a queue using a doubly linked list instead of a singly linked list.

Summary

This chapter talked about many interesting and practical topics, including implementing linked lists, doubly linked lists, hash tables, queues, and stacks in Go, as well as using the functionality of the `container` standard Go package, verifying Sudoku puzzles, and generating pseudo-random numbers, along with difficult-to-guess passwords, in Go.

What you should remember from this chapter is that the foundation of every data structure is the definition and the implementation of its node. Lastly, we talked about performing matrix calculations.

I am sure that you will find the next chapter to be one of the most interesting and valuable chapters of this book. The main topic is Go packages, along with information about how to define and use the various types of Go functions in your programs. Additionally, the chapter will talk about modules, which, put simply, are packages with versions.

What You Might Not Know About Go Packages and Functions

The previous chapter talked about developing and using custom data structures like linked lists, binary trees, and hash tables, as well as generating random numbers and difficult-to-guess passwords in Go and performing matrix operations.

The main focus of this chapter is Go **packages**, which are the Go way of organizing, delivering, and using code. The most common components of a Go package are **functions**, which are pretty flexible in Go. Additionally, this chapter will talk about Go **modules**, which are packages with versions. In the last part of this chapter, you will see some advanced packages that belong to the Go standard library in order to better understand that not all Go packages are created equal.

In this chapter, you will learn about the following topics:

- Developing functions in Go
- Anonymous functions
- Functions that return multiple values
- Giving names to the return values of a function
- Functions that return other functions
- Functions that get other functions as parameters
- Variadic functions
- Developing Go packages
- Developing and working with Go modules
- Private and public package objects
- The use of the `init()` function in packages
- The sophisticated `html/template` standard Go package
- The `text/template` standard package, which is another truly sophisticated Go package that has its own language
- The `go/scanner`, `go/parser`, and `go/token` advanced packages

- The `syscall` standard Go package, which is a low-level package that, although you might not use it directly, is extensively used by other Go packages

About Go packages

Everything in Go is delivered in the form of packages. A Go package is a Go source file that begins with the `package` keyword followed by the name of the package. Some packages have a structure. For example, the `net` package has several subdirectories, named `http`, `mail`, `rpc`, `smtp`, `textproto`, and `url`, which should be imported as `net/http`, `net/mail`, `net/rpc`, `net/smtp`, `net/textproto`, and `net/url`, respectively.

Apart from the packages of the Go standard library, there exist external packages that can be imported using their full address and that should be downloaded before their first use. One such example is `github.com/matryer/is`, which is stored in GitHub.

Packages are mainly used for grouping related functions, variables, and constants so that you can transfer them easily and use them in your own Go programs. Note that apart from the `main` package, Go packages are not autonomous programs and cannot be compiled into executable files. This means that they need to be called directly or indirectly from a `main` package in order to be used. As a result, if you try to execute a Go package as if it is an autonomous program, you will be disappointed:

```
$ go run aPackage.go
go run: cannot run non-main package
```

About Go functions

Functions are an important element of every programming language because they allow you to break big programs into smaller and more manageable parts. Functions must be as independent from each other as possible and must do one job and only one job well. So, if you find yourself writing functions that do multiple things, you might consider replacing them with multiple functions instead.

The single most popular Go function is `main()`, which is used in every independent Go program. You should already know that all function definitions begin with the `func` keyword.

Anonymous functions

Anonymous functions can be defined inline without the need for a name and they are usually used for implementing things that require a small amount of code. In Go, a function can return an anonymous function or take an anonymous function as one of its arguments. Additionally, anonymous functions can be attached to Go variables. Note that anonymous functions are also called **closures**, especially in functional programming terminology.



It is considered good practice for anonymous functions to have a small implementation and a local focus. If an anonymous function does not have a local focus, then you might need to consider making it a regular function.

When an anonymous function is suitable for a job it is extremely convenient and makes your life easier; just do not use too many anonymous functions in your programs without having a good reason. You will see anonymous functions in action in a while.

Functions that return multiple values

As you already know from functions such as `strconv.Atoi()`, Go functions can return multiple distinct values, which saves you from having to create a dedicated structure for returning and receiving multiple values from a function. You can declare a function that returns four values (two `int` values, one `float64` value, and one `string`) as follows:

```
| func aFunction() (int, int, float64, string) {  
| }
```

It is now time to illustrate anonymous functions and functions that return multiple values in more detail using the Go code of `functions.go` as an example. The relevant code will be presented in five parts.

The first code portion of `functions.go` is as follows:

```
| package main  
|  
| import (  
|     "fmt"  
|     "os"  
|     "strconv"  
| )
```

The second code segment from `functions.go` is shown in the following Go code:

```
| func doubleSquare(x int) (int, int) {  
|     return x * 2, x * x  
| }
```

Here you can see the definition and implementation of a function named `doubleSquare()`, which requires a single `int` parameter and returns two `int` values.

The third part of the `functions.go` program is as follows:

```

func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("The program needs 1 argument!")
        return
    }

    y, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println(err)
        return
    }
}

```

The preceding code deals with the command-line arguments of the program.

The fourth portion of the `functions.go` program contains the following Go code:

```

square := func(s int) int {
    return s * s
}
fmt.Println("The square of", y, "is", square(y))

double := func(s int) int {
    return s + s
}
fmt.Println("The double of", y, "is", double(y))

```

Each of the `square` and `double` variables holds an anonymous function. The bad part is that you are allowed to change the value of `square`, `double`, or any other variable that holds an anonymous function afterward, which means that the meaning of those variables can change and calculate something else instead.



It is not considered good programming practice to alter the code of variables that hold anonymous functions because this might be the root cause of nasty bugs.

The last part of `functions.go` is as follows:

```

fmt.Println(doubleSquare(y))
d, s := doubleSquare(y)
fmt.Println(d, s)
}

```

So, you can either print the return values of a function, such as `doubleSquare()`, or assign them to distinct variables.

Executing `functions.go` will generate the following kind of output:

```
$ go run functions.go 1 21
The program needs 1 argument!
$ go run functions.go 10.2
strconv.Atoi: parsing "10.2": invalid syntax
$ go run functions.go 10
The square of 10 is 100
The double of 10 is 20
20 100
20 100
```

The return values of a function can be named

Unlike C, Go allows you to name the return values of a Go function. Additionally, when such a function has a `return` statement without any arguments, then the function automatically returns the current value of each named return value in the order in which it was declared in the definition of the function.

The source code that illustrates Go functions that have named return values is `returnNames.go`, and it will be presented in three parts.

The first part of the `returnNames.go` program is as follows:

```
package main

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

func namedMinMax(x, y int) (min, max int) {
    if x > y {
        min = y
        max = x
    } else {
        min = x
        max = y
    }
    return
}
```

In this code segment, you can see the implementation of the `namedMinMax()` function, which uses named return parameters. However, there is a tricky point here: the `namedMinMax()` function does not explicitly return any variables or values in its `return` statement. Nevertheless, as this function has named return values in its signature, the `min` and `max` parameters are automatically returned in the order in which they were put into the function definition.

The second code segment from `returnNames.go` is as follows:

```

func minMax(x, y int) (min, max int) {
    if x > y {
        min = y
        max = x
    } else {
        min = x
        max = y
    }
    return min, max
}

```

The `minMax()` function also uses named return values, but its `return` statement specifically defines the order and the variables that are going to be returned.

The last code portion from `returnNames.go` is shown in the following Go code:

```

func main() {
    arguments := os.Args
    if len(arguments) < 3 {
        fmt.Println("The program needs at least 2 arguments!")
        return
    }

    a1, _ := strconv.Atoi(arguments[1])
    a2, _ := strconv.Atoi(arguments[2])

    fmt.Println(minMax(a1, a2))
    min, max := minMax(a1, a2)
    fmt.Println(min, max)

    fmt.Println(namedMinMax(a1, a2))
    min, max = namedMinMax(a1, a2)
    fmt.Println(min, max)
}

```

The purpose of the Go code in the `main()` function is to verify that all methods generate the same results.

Executing `returnNames.go` will produce the following output:

```

$ go run returnNames.go -20 1
-20 1
-20 1
-20 1
-20 1

```

Functions with pointer parameters

A function can take pointer parameters provided that its signature allows it. The Go code of `ptrFun.go` will illustrate the use of pointers as function parameters.

The first part of `ptrFun.go` is as follows:

```
package main

import (
    "fmt"
)

func getPtr(v *float64) float64 {
    return *v * *v
}
```

So, the `getPtr()` function accepts a pointer parameter that points to a `float64` value.

The second part of the program is shown in the following Go code:

```
func main() {
    x := 12.2
    fmt.Println(getPtr(&x))
    x = 12
    fmt.Println(getPtr(&x))
}
```

The tricky part here is that you need to pass the address of the variable to the `getPtr()` function because it requires a pointer parameter, which can be done by putting an ampersand in front of a variable (`&x`).

Executing `ptrFun.go` will generate the following kind of output:

```
$ go run ptrFun.go
148.8399999999997
144
```

If you try to pass a plain value, such as `12.12`, to `getPtr()` and call it, such as `getPtr(12.12)`, the compilation of the program will fail, as shown in the following error message:

```
$ go run ptrFun.go
# command-line-arguments
./ptrFun.go:15:21: cannot use 12.12 (type float64) as type *float64 in argument to getPtr
```

Functions that return pointers

As illustrated in `pointerStruct.go` from [Chapter 4](#), *The Uses of Composite Types*, it is considered good practice to create new structure variables using a separate function and return a pointer to them from that function. So, the scenario of functions returning pointers is very common. Generally speaking, such a function simplifies the structure of a program and allows the developer to concentrate on more important things instead of copying the same Go code all the time. This section will use a much simpler example, as found in the Go code of `returnPtr.go`.

The first part of `returnPtr.go` contains the following Go code:

```
package main

import (
    "fmt"
)

func returnPtr(x int) *int {
    y := x * x
    return &y
}
```

Apart from the expected preamble, this code portion defines a new function that returns a pointer to an `int` variable. The only thing to remember is to use `&y` in the `return` statement in order to return the memory address of the `y` variable.

The second part of `returnPtr.go` is as follows:

```
func main() {
    sq := returnPtr(10)
    fmt.Println("sq value:", *sq)
```

The `*` character dereferences a pointer variable, which means that it returns the actual value stored at the memory address instead of the memory address itself.

The last code segment from `returnPtr.go` is shown in the following Go code:

```
|     fmt.Println("sq memory address:", sq)  
| }
```

The preceding code will return the memory address of the `sq` variable, not the `int` value stored in it.

If you execute `returnPtr.go`, you will see the following output (the memory address will differ):

```
$ go run returnPtr.go  
sq value: 100  
sq memory address: 0xc00009a000
```

Functions that return other functions

In this section, you are going to learn how to implement a Go function that returns another function using the Go code of `returnFunction.go`, which will be presented in three segments. The first code segment of `returnFunction.go` is as follows:

```
package main

import (
    "fmt"
)

func funReturnFun() func() int {
    i := 0
    return func() int {
        i++
        return i * i
    }
}
```

As you can see from the implementation of `funReturnFun()`, its return value is an anonymous function (`func() int`).

The second code segment from `returnFunction.go` contains the following code:

```
func main() {
    i := funReturnFun()
    j := funReturnFun()
```

In this code, you call `funReturnFun()` two times and assign its return value, which is a function, to two separate variables named `i` and `j`. As you will see in the output of the program, the two variables are completely unrelated to each other.

The last code section of `returnFunction.go` is as follows:

```
fmt.Println("1:", i())
fmt.Println("2:", i())
fmt.Println("j1:", j())
fmt.Println("j2:", j())
```

```
|     fmt.Println("3:", i())  
| }
```

So, in this Go code, you use the `i` variable three times as `i()` and the `j` variable two times as `j()`. The important thing here is that although both `i` and `j` were created by calling `funReturnFun()`, they are totally independent from each other and have nothing in common.

Executing `returnFunction.go` will produce the following output:

```
$ go run returnFunction.go  
1: 1  
2: 4  
j1: 1  
j2: 4  
3: 9
```

As you can see from the output of `returnFunction.go`, the value of `i` in `funReturnFun()` keeps increasing and does not become `0` after each call either to `i()` or `j()`.

Functions that accept other functions as parameters

Go functions can accept other Go functions as parameters, which is a feature that adds versatility to what you can do with a Go function. The two most common uses of this functionality are functions for sorting elements and the `filepath.Walk()` function. However, in the example presented here, which is named `funFun.go`, we will implement a much simpler case that deals with integer values. The relevant code will be presented in three parts.

The first code segment of `funFun.go` is shown in the following Go code:

```
package main

import "fmt"

func function1(i int) int {
    return i + i
}

func function2(i int) int {
    return i * i
}
```

What we have here is two functions that both accept an `int` and return an `int`. These functions will be used as parameters to another function in a short while.

The second code segment of `funFun.go` contains the following code:

```
func funFun(f func(int) int, v int) int {
    return f(v)
}
```

The `funFun()` function accepts two parameters, a function parameter named `f` and an `int` value. The `f` parameter should be a function that takes one `int` argument and returns an `int` value.

The last code segment of `funFun.go` follows:

```
func main() {
    fmt.Println("function1:", funFun(function1, 123))
    fmt.Println("function2:", funFun(function2, 123))
    fmt.Println("Inline:", funFun(func(i int) int {return i * i
        *i}, 123))
}
```

The first `fmt.Println()` call uses `funFun()` with `function1`, without any parentheses, as its first parameter, whereas the second `fmt.Println()` call uses `funFun()` with `function2` as its first parameter.

In the last `fmt.Println()` statement, something magical happens: the implementation of the function parameter is defined inside the call to `funFun()`. Although this method works fine for simple and small function parameters, it might not work that well for functions with many lines of Go code.

Executing `funFun.go` will produce the next output:

```
$ go run funFun.go
function1: 246
function2: 15129
Inline: 1860867
```

Variadic functions

Go also supports **variadic functions**, which are functions that accept a variable number of arguments. The most popular variadic functions can be found in the `fmt` package. Variadic functions will be illustrated in `variadic.go`, which will be presented in three parts.

The first part of `variadic.go` is as follows:

```
package main

import (
    "fmt"
    "os"
)

func varFunc(input ...string) {
    fmt.Println(input)
}
```

This code part presents the implementation of a variadic function named `varFunc()` that accepts string arguments. The `input` function argument is a slice and will be handled as a slice inside the `varFunc()` function. The `...` operator used as `...type` is called the **pack operator**, whereas the **unpack operator** ends with `...` and begins with a slice. A variadic function cannot use the pack operator more than once.

The second part of `variadic.go` contains the following Go code:

```
func oneByOne(message string, s ...int) int {
    fmt.Println(message)
    sum := 0
    for i, a := range s {
        fmt.Println(i, a)
        sum = sum + a
    }
    s[0] = -1000
    return sum
}
```

Here you can see another variadic function named `oneByOne()` that accepts a single string and a variable number of integer arguments. The `s` function argument is a slice.

The last part of `variadic.go` is as follows:

```
func main() {
    arguments := os.Args
    varFunc(arguments...)
    sum := oneByOne("Adding numbers...", 1, 2, 3, 4, 5, -1, 10)
    fmt.Println("Sum:", sum)
    s := []int{1, 2, 3}
    sum = oneByOne("Adding numbers...", s...)
    fmt.Println(s)
}
```

The `main()` function is what calls and uses the two variadic functions. As the second call to `oneByOne()` uses a slice, any changes you make to the slice inside the variadic function will remain after the function exits.

Building and executing `variadic.go` will generate the following output:

```
$ ./variadic 1 2
[./variadic 1 2]
Adding numbers...
0 1
1 2
2 3
3 4
4 5
5 -1
6 10
Sum: 24
Adding numbers...
0 1
1 2
2 3
[-1000 2 3]
```

Developing your own Go packages

The source code of a Go package, which can contain multiple files and multiple directories, can be found within a single directory that is named after the package name, with the obvious exception of the `main` package, which can be located anywhere.

For the purposes of this section, a simple Go package named `aPackage` will be developed. The source file of the package is called `aPackage.go`, and its source code will be presented in two parts.

The first part of `aPackage.go` is shown in the following Go code:

```
package aPackage

import (
    "fmt"
)

func A() {
    fmt.Println("This is function A!")
}
```

Notice that using capital letters in Go package names is not considered a good practice – `aPackage` is only used here as an example.

The second code segment of `aPackage.go` follows:

```
func B() {
    fmt.Println("privateConstant:", privateConstant)
}

const MyConstant = 123
const privateConstant = 21
```

As you can see, developing a new Go package is pretty easy. Right now, you cannot use that package on its own and you need to create a package named `main` with a `main()` function in it in order to create an executable file. In this case, the name of the program that will use `aPackage` is `useAPackage.go`, and it is included in the following Go code:

```

package main

import (
    "aPackage"
    "fmt"
)

func main() {
    fmt.Println("Using aPackage!")
    aPackage.A()
    aPackage.B()
    fmt.Println(aPackage.MyConstant)
}

```

If you try to execute `useAPackage.go` right now, however, you will get an error message, which means that we are not done yet:

```

$ go run useAPackage.go
useAPackage.go:4:2: cannot find package "aPackage" in any of:
    /usr/local/Cellar/go/1.9.2/libexec/src/aPackage (from $GOROOT)
    /Users/mtsouk/go/src/aPackage (from $GOPATH)

```

There is another thing that you will need to handle. As you already know from [Chapter 1](#), *Go and the Operating System*, Go requires the execution of specific commands from the UNIX shell in order to install all external packages, which also includes packages that you have developed locally. Therefore, you will need to put the preceding package in the appropriate directory and make it available to the current UNIX user. Thus, installing one of your own packages involves the execution of the following commands from your favorite UNIX shell:

```

$ mkdir ~/go/src/aPackage
$ cp aPackage.go ~/go/src/aPackage/
$ go install aPackage
$ cd ~/go/pkg/darwin_amd64/
$ ls -l aPackage.a
-rw-r--r-- 1 mtsouk staff 4980 Dec 22 06:12 aPackage.a

```



If the `~/go` directory does not already exist, you will need to create it with the help of the `mkdir(1)` command. In that case, you will also need to do the same for the `~/go/src` directory.

Executing `useAPackage.go` will create the following output:

```

$ go run useAPackage.go
Using aPackage!
This is function A!
privateConstant: 21

```


Compiling a Go package

Although you cannot execute a Go package if it does not include a `main()` function, you are still allowed to compile it and create an object file, as follows:

```
$ go tool compile aPackage.go
$ ls -l aPackage.*
-rw-r--r--@ 1 mtsouk  staff    201 Jan 10 22:08 aPackage.go
-rw-r--r--  1 mtsouk  staff  16316 Mar  4 20:01 aPackage.o
```

Private variables and functions

What differentiates private variables and functions from public ones is that private ones can be strictly used and called internally in a package. Controlling which functions, constants, and variables are public or not is also known as **encapsulation**.

Go follows a simple rule that states that functions, variables, types, and so forth that begin with an uppercase letter are public, whereas functions, variables, types, and so on that begin with a lowercase letter are private. This is the reason that `fmt.Println()` is named `Println()` instead of `println()`. However, this rule does not affect package names that are allowed to begin with uppercase and lowercase letters.

The `init()` function

Every Go package can optionally have a private function named `init()` that is automatically executed at the beginning of the execution time.



The `init()` function is a private function by design, which means that it cannot be called from outside the package in which it is contained. Additionally, as the user of a package has no control over the `init()` function, you should think carefully before using an `init()` function in public packages or changing any global state in `init()`.

I will now present a code example with multiple `init()` functions from multiple Go packages. Examine the code of the following basic Go package, which is simply called `a`:

```
package a

import (
    "fmt"
)

func init() {
    fmt.Println("init() a")
}

func FromA() {
    fmt.Println("fromA()")
}
```

The `a` package implements an `init()` function and a public one named `FromA()`.

After that, you will need to execute the following commands from your UNIX shell so that the package becomes available to the current UNIX user:

```
$ mkdir ~/go/src/a
$ cp a.go ~/go/src/a/
$ go install a
```

Now, look at the code of the next Go code package, which is named `b`:

```
package b
```

```

import (
    "a"
    "fmt"
)

func init() {
    fmt.Println("init() b")
}

func FromB() {
    fmt.Println("fromB()")
    a.FromA()
}

```

What is happening here? Package `a` uses the `fmt` standard Go package. However, package `b` needs to import package `a`, as it uses `a.FromA()`. Both `a` and `b` have an `init()` function.

As before, you will need to install that package and make it available to the current UNIX user by executing the following commands from your UNIX shell:

```

$ mkdir ~/go/src/b
$ cp b.go ~/go/src/b
$ go install b

```

Thus, we currently have two Go packages that both have an `init()` function. Now try to guess the output that you will get from executing `manyInit.go`, which comes with the following code:

```

package main

import (
    "a"
    "b"
    "fmt"
)

func init() {
    fmt.Println("init() manyInit")
}

func main() {
    a.FromA()
    b.FromB()
}

```

The actual question could have been: how many times is the `init()` function of package `a` going to be executed? Executing `manyInit.go` will generate the

following output and shed some light on this question:

```
$ go run manyInit.go
init() a
init() b
init() manyInit
fromA()
fromB()
fromA()
```

The preceding output shows that the `init()` function of `a` is executed only once, despite the fact that the `a` package is imported two times by two different packages. Additionally, as the `import` block from `manyInit.go` is executed first, the `init()` functions of package `a` and package `b` are executed before the `init()` function of `manyInit.go`, which makes perfect sense. The main reason for this is that the `init()` function of `manyInit.go` is allowed to use an element from either `a` or `b`.

Notice that `init()` can be very useful when you want to set up some unexported internal variables. As an example, you might find the current time zone in `init()`. Finally, bear in mind that you can have many `init()` functions within one file; however, this Go feature is rarely used.

Go modules

Go modules were first introduced in Go version 1.11. At the time of writing, the latest Go version is 1.13. Although the general idea behind Go modules will remain the same, some of the presented details might change in future versions of Go.

A Go module is like a Go package with a version. Go uses **semantic versioning** for versioning modules. This means that versions begin with the letter `v` followed by the version number. Therefore, you can have versions such as `v1.0.0`, `v1.0.5`, and `v2.0.2`. The `v1`, `v2`, or `v3` part signifies the major version of a Go package that is usually not backwards compatible. This means that if your Go program works with `v1`, it will not necessarily work with `v2` or `v3` – it might work, but you cannot count on it.

The second number in a version is about features. Usually `v1.1.0` has more features than `v1.0.2` or `v1.0.0`, while being compatible with all older versions.

Lastly, the third number is just about bug fixes without having any new features. Note that semantic versioning is also used for Go versions.

Note that Go modules allow you to write things outside of `GOPATH`.

Creating and using a Go module

In this subsection, we are going to create the first version of a basic module. You will need to have a GitHub repository for storing your Go code. In my case, the GitHub repository will be <https://github.com/mactsouk/myModule>. We will begin with an empty GitHub repository – only `README.md` will be there. So, first we will need to execute the following command to get the contents of the GitHub repository:

```
$ git clone git@github.com:mactsouk/myModule.git
Cloning into 'myModule'...
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 7 (delta 1), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (7/7), done.
Resolving deltas: 100% (1/1), done.
```

If you execute the same command on your computer, you will get my GitHub repository, which will not be empty at the time you read this. If you want to create your own Go module from scratch, you will need to create your own empty GitHub repository.

Creating version v1.0.0

We will need to execute the following commands to create v1.0.0 of our basic Go module:

```
$ go mod init
go: creating new go.mod: module github.com/mactsouk/myModule
$ touch myModule.go
$ vi myModule.go
$ git add .
$ git commit -a -m "Initial version 1.0.0"
$ git push
$ git tag v1.0.0
$ git push -q origin v1.0.0
$ go list
github.com/mactsouk/myModule
$ go list -m
github.com/mactsouk/myModule
```

The contents of `myModule.go` will be as follows:

```
package myModule

import (
    "fmt"
)

func Version() {
    fmt.Println("Version 1.0.0")
}
```

The contents of `go.mod`, which was created previously, will be as follows:

```
$ cat go.mod
module github.com/mactsouk/myModule

go 1.12
```

Using version v1.0.0

In this section, you are going to learn how to use `v1.0.0` of the Go module we created earlier. In order to use our Go modules, we will need to create a Go program, which in this case will be called `useModule.go` and will contain the following Go code:

```
package main

import (
    v1 "github.com/mactsouk/myModule"
)

func main() {
    v1.Version()
}
```

You will need to include the path of the Go module (`github.com/mactsouk/myModule`) – in this case the Go module also has an alias (`v1`). Using aliases for packages is highly irregular in Go; however, in this example it makes the code easier to read. Nevertheless, this feature should not be used on production code without a really good reason.

If you just try to execute `useModule.go`, which in this case is going to be put in `/tmp`, it will fail because the required module is not present on your system:

```
$ pwd
/tmp
$ go run useModule.go
useModule.go:4:2: cannot find package "github.com/mactsouk/myModule" in any of:
/usr/local/Cellar/go/1.12/libexec/src/github.com/mactsouk/myModule (from $GOROOT)
/Users/mtsouk/go/src/github.com/mactsouk/myModule (from $GOPATH)
```

Therefore, you will need to execute the following commands to get the required Go modules and to successfully execute `useModule.go`:

```
$ export GO111MODULE=on
$ go run useModule.go
go: finding github.com/mactsouk/myModule v1.0.0
go: downloading github.com/mactsouk/myModule v1.0.0
go: extracting github.com/mactsouk/myModule v1.0.0
Version 1.0.0
```

So, `useModule.go` is correct and can be executed. Now it is time to make things even more official by giving `useModule.go` a name and building it:

```
$ go mod init hello
go: creating new go.mod: module hello
$ go build
```

The last command generates an executable file inside `/tmp`, as well as two additional files named `go.sum` and `go.mod`. The contents of `go.sum` will be as follows:

```
$ cat go.sum
github.com/mactsouk/myModule v1.0.0 h1:eTCn2JewnaJw0REKONrVhHmeDEJ0Q5TAZ0xsSbh8kFs=
github.com/mactsouk/myModule v1.0.0/go.mod h1:s3ziarTDDvaXaHWYYOf/ULi97aoBd6JfnvAkM8rSuzg=
```

The `go.sum` file keeps a checksum of all the modules it has downloaded.

The contents of `go.mod` will be as follows:

```
$ cat go.mod
module hello

go 1.12

require github.com/mactsouk/myModule v1.0.0
```



Please note that if the `go.mod` file found in your project specifies the use of version `v1.3.0` of a Go module, Go will use version `v1.3.0` even if a newer version of the Go module is available.

Creating version v1.1.0

In this subsection, we are going to create a new version of `myModule` using a different **tag**. However, this time there is no need to execute `go mod init`, as this was done before. You will just need to execute the following commands:

```
$ vi myModule.go
$ git commit -a -m "v1.1.0"
[master ddd0742] v1.1.0
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git push
$ git tag v1.1.0
$ git push -q origin v1.1.0
```

The contents of this version of `myModule.go` will be as follows:

```
package myModule

import (
    "fmt"
)

func Version() {
    fmt.Println("Version 1.1.0")
}
```

Using version v1.1.0

In this section, you are going to learn how to use v1.1.0 of the Go module we created. This time we are going to use a **Docker image** in order to be as independent from the machine we used for developing the module as possible. The command we will use to get the Docker image and going into its UNIX shell is the following:

```
$ docker run --rm -it golang:latest
root@884c0d188694:/go# cd /tmp
root@58c5688e3ee0:/tmp# go version
go version go1.13 linux/amd64
```

As you can see, the Docker image uses the latest version of Go, which at the time of writing is 1.13. In order to use one or more Go modules, you will need to create a Go program, which is called `useUpdatedModule.go` and contains the following Go code:

```
package main

import (
    v1 "github.com/mactsouk/myModule"
)

func main() {
    v1.Version()
}
```

The Go code of `useUpdatedModule.go` is the same as the Go code of `useModule.go`. The good thing is that you will automatically get the latest update of version `v1`.

After writing the program in the Docker image, you will need to do the following:

```
root@58c5688e3ee0:/tmp# ls -l
total 4
-rw-r--r-- 1 root root 91 Mar  2 19:59 useUpdatedModule.go
root@58c5688e3ee0:/tmp# export GO111MODULE=on
root@58c5688e3ee0:/tmp# go run useUpdatedModule.go
go: finding github.com/mactsouk/myModule v1.1.0
```

```
| go: downloading github.com/mactsouk/myModule v1.1.0
| go: extracting github.com/mactsouk/myModule v1.1.0
Version 1.1.0
```

This means that `useUpdatedModule.go` is automatically using the latest `v1` version of the Go module. It is critical that you execute `export GO111MODULE=on` to turn on module support.

If you try to execute `useModule.go`, which is located in the `/tmp` directory of your local machine, you will get the following:

```
$ ls -l go.mod go.sum useModule.go
-rw----- 1 mtsouk wheel 67 Mar 2 21:29 go.mod
-rw----- 1 mtsouk wheel 175 Mar 2 21:29 go.sum
-rw-r--r-- 1 mtsouk wheel 92 Mar 2 21:12 useModule.go
$ go run useModule.go
Version 1.0.0
```

This means that `useModule.go` is still using the older version of the Go module. If you want `useModule.go` to use the latest version of the Go module, you can do the following:

```
$ rm go.mod go.sum
$ go run useModule.go
go: finding github.com/mactsouk/myModule v1.1.0
go: downloading github.com/mactsouk/myModule v1.1.0
go: extracting github.com/mactsouk/myModule v1.1.0
Version 1.1.0
```

If you want to go back to using `v1.0.0` of the module, you can do the following:

```
$ go mod init hello
go: creating new go.mod: module hello
$ go build
$ go run useModule.go
Version 1.1.0
$ cat go.mod
module hello

go 1.12

require github.com/mactsouk/myModule v1.1.0
$ vi go.mod
$ cat go.mod
module hello

go 1.12
```

```
| require github.com/mactsouk/myModule v1.0.0
| $ go run useModule.go
Version 1.0.0
```

The next subsection will create a new major version of the Go module, which means that instead of using a different tag, we will need to use a different GitHub branch.

Creating version v2.0.0

In this subsection, we are going to create the second major version of `myModule`. Note that for major versions, you will need to be explicit in your `import` statements.

So `github.com/mactsouk/myModule` will become `github.com/mactsouk/myModule/v2` for version v2 and `github.com/mactsouk/myModule/v3` for v3.

The first thing to do is create a new GitHub branch:

```
$ git checkout -b v2
Switched to a new branch 'v2'
$ git push --set-upstream origin v2
```

Then, you should do the following:

```
$ vi go.mod
$ cat go.mod
module github.com/mactsouk/myModule/v2

go 1.12
$ git commit -a -m "Using 2.0.0"
[v2 5af2269] Using 2.0.0
 2 files changed, 2 insertions(+), 2 deletions(-)
$ git tag v2.0.0
$ git push --tags origin v2
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 441 bytes | 441.00 KiB/s, done.
Total 4 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:mactsouk/myModule.git
 * [new branch]      v2 -> v2
 * [new tag]         v2.0.0 -> v2.0.0
$ git --no-pager branch -a
  master
* v2
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/v2
```

The contents of this major version of `myModule.go` will be as follows:

```
package myModule

import (
    "fmt"
)

func Version() {
    fmt.Println("Version 2.0.0")
}
```

Using version v2.0.0

Once again, in order to use our Go modules, we will need to create a Go program, which is called `useV2.go` and contains the following Go code:

```
package main

import (
    v "github.com/mactsouk/myModule/v2"
)

func main() {
    v.Version()
}
```

We are going to use a Docker image. This is the most convenient way of playing with Go modules because we are starting with a clean Go installation:

```
$ docker run --rm -it golang:latest
root@191d84fc5571:/go# cd /tmp
root@191d84fc5571:/tmp# cat > useV2.go
package main

import (
    v "github.com/mactsouk/myModule/v2"
)

func main() {
    v.Version()
}
root@191d84fc5571:/tmp# export GO111MODULE=on
root@191d84fc5571:/tmp# go run useV2.go
go: finding github.com/mactsouk/myModule/v2 v2.0.0
go: downloading github.com/mactsouk/myModule/v2 v2.0.0
go: extracting github.com/mactsouk/myModule/v2 v2.0.0
Version 2.0.0
```

Everything is working fine as the Docker image is using version `v2.0.0` of `myModule`.

Creating version v2.1.0

We are now going to create an updated version of `myModule.go` that has to do with using a different GitHub tag. So, execute the following commands:

```
$ vi myModule.go
$ git commit -a -m "v2.1.0"
$ git push
$ git tag v2.1.0
$ git push -q origin v2.1.0
```

The updated contents of `myModule.go` will be as follows:

```
package myModule

import (
    "fmt"
)

func Version() {
    fmt.Println("Version 2.1.0")
}
```

Using version v2.1.0

As you already know, in order to use our Go modules, we will need to create a Go program, which will be called `useUpdatedV2.go` and contains the following Go code:

```
package main

import (
    v "github.com/mactsouk/myModule/v2"
)

func main() {
    v.Version()
}
```

Still, there is no need to declare that you want to use the latest `v2` version of the Go module because this is handled by Go, which is the main reason that `useUpdatedV2.go` and `useV2.go` are exactly the same.

Once again, a Docker image will be used for reasons of simplicity – the reason for using the `cat(1)` command to create `useUpdatedV2.go` is because that particular Docker image comes without `vi(1)` installed.

```
$ docker run --rm -it golang:1.12
root@ccfc675e333:/go# cd /tmp/
root@ccfc675e333:/tmp# cat > useUpdatedV2.go
package main

import (
    v "github.com/mactsouk/myModule/v2"
)

func main() {
    v.Version()
}
root@ccfc675e333:/tmp# ls -l
total 4
-rw-r--r-- 1 root root 92 Mar  2 20:34 useUpdatedV2.go
root@ccfc675e333:/tmp# go run useUpdatedV2.go
useUpdatedV2.go:4:2: cannot find package "github.com/mactsouk/myModule/v2" in any of:
    /usr/local/go/src/github.com/mactsouk/myModule/v2 (from $GOROOT)
    /go/src/github.com/mactsouk/myModule/v2 (from $GOPATH)
root@ccfc675e333:/tmp# export GO111MODULE=on
root@ccfc675e333:/tmp# go run useUpdatedV2.go
go: finding github.com/mactsouk/myModule/v2 v2.1.0
go: downloading github.com/mactsouk/myModule/v2 v2.1.0
go: extracting github.com/mactsouk/myModule/v2 v2.1.0
```

| Version 2.1.0



You will learn more about `git(1)` and GitHub in [Chapter 7](#), *Reflection and Interfaces for All Seasons*.

Using two different versions of the same Go module

In this subsection, you will see how to use two different major versions of the same Go module in a single Go program. The same technique can be used if you want to use more than two major versions of a Go module at the same time.

The name of the Go source file will be `useTwo.go` and it is as follows:

```
package main

import (
    v1 "github.com/mactsouk/myModule"
    v2 "github.com/mactsouk/myModule/v2"
)

func main() {
    v1.Version()
    v2.Version()
}
```

So, you just need to explicitly import the major versions of the Go module you want to use and give them different aliases.

Executing `useTwo.go` will generate the following kind of output:

```
$ export GO111MODULE=on
$ go run useTwo.go
go: creating new go.mod: module github.com/PacktPublishing/Mastering-Go-Second-Edition
go: finding github.com/mactsouk/myModule/v2 v2.1.0
go: downloading github.com/mactsouk/myModule/v2 v2.1.0
go: extracting github.com/mactsouk/myModule/v2 v2.1.0
Version 1.1.0
Version 2.1.0
```

Where Go stores Go modules

In this section, we are going to see where and how Go stores the code and the information about the Go modules we are using, using our Go module as an example. Here are the contents of the `~/go/pkg/mod/github.com/mactsouk` directory after using the presented Go module on my local macOS Mojave machine:

```
$ ls -lR ~/go/pkg/mod/github.com/mactsouk
total 0
drwxr-xr-x  3 mtsouk  staff    96B Mar  2 22:38 my!module
dr-x-----  6 mtsouk  staff   192B Mar  2 21:18 my!module@v1.0.0
dr-x-----  6 mtsouk  staff   192B Mar  2 22:07 my!module@v1.1.0

/Users/mtsouk/go/pkg/mod/github.com/mactsouk/my!module:
total 0
dr-x-----  6 mtsouk  staff   192B Mar  2 22:38 v2@v2.1.0

/Users/mtsouk/go/pkg/mod/github.com/mactsouk/my!module/v2@v2.1.0:
total 24
-r--r---  1 mtsouk  staff    28B Mar  2 22:38 README.md
-r--r---  1 mtsouk  staff    48B Mar  2 22:38 go.mod
-r--r---  1 mtsouk  staff    86B Mar  2 22:38 myModule.go

/Users/mtsouk/go/pkg/mod/github.com/mactsouk/my!module@v1.0.0:
total 24
-r--r---  1 mtsouk  staff    28B Mar  2 21:18 README.md
-r--r---  1 mtsouk  staff    45B Mar  2 21:18 go.mod
-r--r---  1 mtsouk  staff    86B Mar  2 21:18 myModule.go

/Users/mtsouk/go/pkg/mod/github.com/mactsouk/my!module@v1.1.0:
total 24
-r--r---  1 mtsouk  staff    28B Mar  2 22:07 README.md
-r--r---  1 mtsouk  staff    45B Mar  2 22:07 go.mod
-r--r---  1 mtsouk  staff    86B Mar  2 22:07 myModule.go
```



The best way to learn how to develop and use Go modules is to experiment. Go modules are here to stay, so start using them.

The go mod vendor command

There are times when you need to store all your dependencies in the same place and keep them close to the files of your project. In these situations, the `go mod vendor` command can help you to do exactly this:

```
$ cd useTwoVersions
$ go mod init useV1V2
go: creating new go.mod: module useV1V2
$ go mod vendor
$ ls -l
total 24
-rw----- 1 mtsouk  staff   114B Mar  2 22:43 go.mod
-rw----- 1 mtsouk  staff   356B Mar  2 22:43 go.sum
-rw-r--r--@ 1 mtsouk  staff   143B Mar  2 19:36 useTwo.go
drwxr-xr-x 4 mtsouk  staff   128B Mar  2 22:43 vendor
$ ls -l vendor/github.com/mactsovuk/myModule
total 24
-rw-r--r-- 1 mtsouk  staff    28B Mar  2 22:43 README.md
-rw-r--r-- 1 mtsouk  staff    45B Mar  2 22:43 go.mod
-rw-r--r-- 1 mtsouk  staff    86B Mar  2 22:43 myModule.go
drwxr-xr-x 6 mtsouk  staff   192B Mar  2 22:43 v2
$ ls -l vendor/github.com/mactsovuk/myModule/v2
total 24
-rw-r--r-- 1 mtsouk  staff    28B Mar  2 22:43 README.md
-rw-r--r-- 1 mtsouk  staff    48B Mar  2 22:43 go.mod
-rw-r--r-- 1 mtsouk  staff    86B Mar  2 22:43 myModule.go
```

The key point here is to execute `go mod init <package name>` before executing the `go mod vendor` command.

Creating good Go packages

This section will provide some handy advice that will help you to develop better Go packages. We have covered that Go packages are organized in directories and can contain public and private elements. Public elements can be used both internally and externally from other packages, whereas private elements can only be used internally in a package.

Here are several good rules to follow to create superior Go packages:

- The first unofficial rule of a successful package is that its elements must be related in some way. Thus, you can create a package for supporting cars, but it would not be a good idea to create a single package for supporting both cars and bicycles. Put simply, it is better to split the functionality of a package unnecessarily into multiple packages than to add too much functionality to a single Go package. Additionally, packages should be made simple and stylish-but not too simplistic and fragile.
- A second practical rule is that you should use your own packages first for a reasonable amount of time before giving them to the public. This will help you to discover silly bugs and make sure that your packages operate as expected. After that, give them to some fellow developers for additional testing before making them publicly available.
- Next, try to imagine the kinds of users who will use your packages happily and make sure that your packages will not create more problems to them than they can solve.
- Unless there is a very good reason for doing so, your packages should not export an endless list of functions. Packages that export a short list of functions are understood better and used more easily. After that, try to title your functions using descriptive but not very long names.
- **Interfaces** can improve the usefulness of your functions, so when you think it is appropriate, use an interface instead of a single type as a function parameter or return type.
- When updating one of your packages, try not to break things and create incompatibilities with older versions unless it is absolutely

necessary.

- When developing a new Go package, try to use multiple files in order to group similar tasks or concepts.
- Additionally, try to follow the rules that exist in the Go packages of the standard library. Reading the code of a Go package that belongs to the standard library will help you on this.
- Do not create a package that already exists from scratch. Make changes to the existing package and maybe create your own version of it.
- Nobody wants a Go package that prints logging information on the screen. It would be more professional to have a flag for turning on logging when needed.
- The Go code of your packages should be in harmony with the Go code of your programs. This means that if you look at a program that uses your packages and your function names stand out in the code in a bad way, it would be better to change the names of your functions. As the name of a package is used almost everywhere, try to use a concise and expressive package name.
- It is more convenient if you put new Go type definitions near the place that they will be used for the first time because nobody, including you, wants to search source files for definitions of new data types.
- Try to create test files for your packages, because packages with test files are considered more professional than ones without them; small details make all the difference and give people confidence that you are a serious developer! Notice that writing tests for your packages is not optional and that you should avoid using packages that do not include tests. You will learn more about testing in [Chapter 11, *Code Testing, Optimization, and Profiling*](#).
- Finally, do not write a Go package because you do not have anything better to do – in that case, find something better to do and do not waste your time!



*Always remember that apart from the fact that the actual Go code in a package should be bug-free, the next most important element of a successful package is its **documentation**, as well as some code examples that clarify its use and showcase the idiosyncrasies of the functions of the package.*

The syscall package

This section will present a small portion of the functionality of the `syscall` standard Go package. Note that the `syscall` package offers a plethora of functions and types related to low-level operating system primitives.

Additionally, the `syscall` package is extensively used by other Go packages, such as `os`, `net`, and `time`, which all provide a portable interface to the operating system. This means that the `syscall` package is not the most portable package in the Go library – that is not its job.

Although UNIX systems have many similarities, they also exhibit various differences, especially when we talk about their system internals. The job of the `syscall` package is to deal with all of these incompatibilities as gently as possible. The fact that this is not a secret and is well documented makes `syscall` a successful package.

Strictly speaking, a **system call** is a programmatic way for an application to request something from the kernel of an operating system. As a consequence, system calls are responsible for accessing and working with most UNIX low-level elements such as processes, storage devices, printing data, network interfaces, and all kinds of files. Put simply, you cannot work on a UNIX system without using system calls. You can inspect the system calls of a UNIX process using utilities such as `strace(1)` and `dtrace(1)`, which were presented in [Chapter 2, Understanding Go Internals](#).

The use of the `syscall` package will be illustrated in the `useSyscall.go` program, which will be presented in four parts.



You might not directly need to use the `syscall` package unless you are working on pretty low-level stuff. Not all Go packages are for everyone!

The first code portion of `useSyscall.go` is as follows:

```
package main  
import (
```

```
| "fmt"  
| "os"  
| "syscall"  
| )
```

This is the easy part of the program, where you just import the required Go packages.

The second part of `useSyscall.go` is shown in the following Go code:

```
| func main() {  
|     pid, _, _ := syscall.Syscall(39, 0, 0, 0)  
|     fmt.Println("My pid is", pid)  
|     uid, _, _ := syscall.Syscall(24, 0, 0, 0)  
|     fmt.Println("User ID:", uid)
```

In this part, you find out information about the process ID and the user ID using two `syscall.Syscall()` calls. The first parameter of the `syscall.Syscall()` call determines the information that you request.

The third code segment of `useSyscall.go` contains the following Go code:

```
|     message := []byte{'H', 'e', 'l', 'l', 'o', '!', '\n'}  
|     fd := 1  
|     syscall.Write(fd, message)
```

In this part, you print a message on the screen using `syscall.Write()`. The first parameter is the file descriptor to which you will write, and the second parameter is a byte slice that holds the actual message. The `syscall.Write()` function is portable.

The last part of the `useSyscall.go` program is as follows:

```
|     fmt.Println("Using syscall.Exec())")  
|     command := "/bin/ls"  
|     env := os.Environ()  
|     syscall.Exec(command, []string{"ls", "-a", "-x"}, env)  
| }
```

In the last part of the program, you can see how to use the `syscall.Exec()` function to execute an external command. However, you have no control over the output of the command, which is automatically printed on the screen.

Executing `useSyscall.go` on macOS Mojave will generate the following output:

```
$ go run useSyscall.go
My pid is 14602
User ID: 501
Hello!
Using syscall.Exec()
.
    ..          a.go
funFun.go      functions.go    html.gohtml
htmlT.db       htmlT.go       manyInit.go
ptrFun.go      returnFunction.go returnNames.go
returnPtr.go   text.gotext    textT.go
useAPackage.go useSyscall.go
```

Executing the same program on a Debian Linux machine will generate the following output:

```
$ go run useSyscall.go
My pid is 20853
User ID: 0
Hello!
Using syscall.Exec()
.
    ..          a.go
funFun.go      functions.go    html.gohtml
htmlT.db       htmlT.go       manyInit.go
ptrFun.go      returnFunction.go returnNames.go
returnPtr.go   text.gotext    textT.go
useAPackage.go useSyscall.go
```

So, although most of the output is the same as before, the `syscall.Syscall(39, 0, 0, 0)` call does not work on Linux because the user ID of the Linux user is not `0`, which means that this command is not portable.

If you want to discover which standard Go packages use the `syscall` package, you can execute the next command from your UNIX shell:

```
$ grep \"syscall\" `find /usr/local/Cellar/go/1.12/libexec/src -name \"*.go\"`
```

Please replace `/usr/local/Cellar/go/1.12/libexec/src` with the appropriate directory path.

Finding out how `fmt.Println()` really works

If you really want to grasp the usefulness of the `syscall` package, start by reading this subsection. The implementation of the `fmt.Println()` function, as found in <https://golang.org/src/fmt/print.go>, is as follows:

```
| func Println(a ...interface{}) (n int, err error) {
|     return Fprintln(os.Stdout, a...)
| }
```

This means that the `fmt.Println()` function calls `fmt.Fprintln()` to do its job. The implementation of `fmt.Fprintln()`, as found in the same file, is as follows:

```
| func Fprintln(w io.Writer, a ...interface{}) (n int, err error) {
|     p := newPrinter()
|     p.doPrintln(a)
|     n, err = w.Write(p.buf)
|     p.free()
|     return
| }
```

This means that the actual writing in `fmt.Fprintln()` is done by the `Write()` function of the `io.Writer` interface. In this case, the `io.Writer` interface is `os.Stdout`, which is defined as follows in <https://golang.org/src/os/file.go>:

```
| var (
|     Stdin  = NewFile(uintptr(syscall.Stdin), "/dev/stdin")
|     Stdout = NewFile(uintptr(syscall.Stdout), "/dev/stdout")
|    .Stderr = NewFile(uintptr(syscall.Stderr), "/dev/stderr")
| )
```

Now look at the implementation of `NewFile()`, which can be found inside [http://golang.org/src/os/file_plan9.go](https://golang.org/src/os/file_plan9.go):

```
| func NewFile(fd uintptr, name string) *File {
|     fdi := int(fd)
|     if fdi < 0 {
|         return nil
|     }
|     f := &File{&file{fd: fdi, name: name}}
|     runtime.SetFinalizer(f.file, (*file).close)
| }
```

```
|     return f  
| }
```

When you see a Go source file named `file_plan9.go`, you should suspect that it contains commands specific to a UNIX variant, which means that it contains code that is not portable.

What we have here is the `file` structure type that is embedded in the `File` type, which is the one that is being exported due to its name. So, start looking for functions inside https://golang.org/src/os/file_plan9.go that are applied to a `File` structure or to a pointer to a `File` structure, and that allow you to write data. As the function we are seeking is named `Write()` – look at the implementation of `Fprintln()` – we will have to search all of the source files of the `os` package to find it:

```
$ grep "func (f *File) Write(" *.go  
file.go:func (f *File) Write(b []byte) (n int, err error) {
```

The implementation of `Write()` as found in <https://golang.org/src/os/file.go> is as follows:

```
func (f *File) Write(b []byte) (n int, err error) {  
    if err := f.checkValid("write"); err != nil {  
        return 0, err  
    }  
    n, e := f.write(b)  
    if n < 0 {  
        n = 0  
    }  
    if n != len(b) {  
        err = io.ErrShortWrite  
    }  
  
    epipecheck(f, e)  
  
    if e != nil {  
        err = f.wrapErr("write", e)  
    }  
  
    return n, err  
}
```

This means that we now have to search for the `write()` function. Searching for the `write` string in https://golang.org/src/os/file_plan9.go reveals the following function inside https://golang.org/src/os/file_plan9.go:

```
| func (f *File) write(b []byte) (n int, err error) {
|     if len(b) == 0 {
|         return 0, nil
|     }
|     return fixCount(syscall.Write(f.fd, b))
| }
```

This tells us that a call to the `fmt.Println()` function is implemented using a call to `syscall.Write()`. This underscores how useful and necessary the `syscall` package is.

The go/scanner, go/parser, and go/token packages

This section will talk about the `go/scanner`, `go/parser`, and `go/token` packages, as well as the `go/ast` package. This is low-level information about how Go scans and parses Go code that will help you to understand how Go works. However, you might want to skip this section if low-level things frustrate you.

Parsing a language requires two phases. The first one is about breaking up the input into tokens (**lexical analysis**) and the second one is about feeding the parser with all these tokens in order to make sure that these tokens make sense and are in the right order (**semantic analysis**). Just combining English words does not always create valid sentences.

The `go/ast` package

An **abstract syntax tree (AST)** is a structured representation of the source code of a Go program. This tree is constructed according to some rules that are specified in the language specification. The `go/ast` package is used for declaring the data types required to represent ASTs in Go. If you want to find out more about an `ast.*` type, the `go/ast` package should be the best place for this kind of information.

The go/scanner package

A **scanner** is something, which in this case will be some Go code, that reads a program written in a programming language, which in this case is Go, and generates tokens.

The `go/scanner` package is used for reading Go programs and generating a series of tokens. The use of the `go/scanner` package will be illustrated in `goscaner.go`, which will be presented in three parts.

The first part of `goscaner.go` is as follows:

```
package main

import (
    "fmt"
    "go/scanner"
    "go/token"
    "io/ioutil"
    "os"
)

func main() {
    if len(os.Args) == 1 {
        fmt.Println("Not enough arguments!")
        return
    }
}
```

The `go/token` package defines constants that represent the lexical tokens of the Go programming language.

The second part of `goscaner.go` contains the following Go code:

```
for _, file := range os.Args[1:] {
    fmt.Println("Processing:", file)
    f, err := ioutil.ReadFile(file)
    if err != nil {
        fmt.Println(err)
        return
    }
    One := token.NewFileSet()
    files := one.AddFile(file, one.Base(), len(f))
```

The source file that is going to be tokenized is stored in the `file` variable, whereas its contents are stored in `f`.

The last part of `goScanner.go` is as follows:

```
    var myScanner scanner.Scanner
    myScanner.Init(files, f, nil, scanner.ScanComments)

    for {
        pos, tok, lit := myScanner.Scan()
        if tok == token.EOF {
            break
        }
        fmt.Printf("%s\t%s\t%q\n", one.Position(pos), tok, lit)
    }
}
```

The `for` loop is used for traversing the input file. The end of the source code file is indicated by `tok==EOF` – this will exit the `for` loop. The `scanner.Scan()` method returns the current file position, the token, and the literal. The use of `scanner.ScanComments` in `scanner.Init()` tells the scanner to return comments as `COMMENT` tokens. You can use `1` instead of `scanner.ScanComments` and you can put `0` if you do not want to see any `COMMENT` tokens in the output.

Building and executing `goScanner.go` will create the following output:

```
$ ./goScanner a.go
Processing: a.go
a.go:1:1      package "package"
a.go:1:9      IDENT   "a"
a.go:1:10     ;       "\n"
a.go:3:1      import  "import"
a.go:3:8      (
a.go:4:2      STRING  "\"fmt\""
a.go:4:7      ;
a.go:5:1      )
a.go:5:2      ;
a.go:7:1      func    "func"
a.go:7:6      IDENT   "init"
a.go:7:10     (
a.go:7:11     )
a.go:7:13     {
a.go:8:2      IDENT   "fmt"
a.go:8:5      .
a.go:8:6      IDENT   "Println"
a.go:8:13     (
a.go:8:14     STRING  "\"init() a\""
a.go:8:24     )
a.go:8:25     ;
a.go:9:1      }
a.go:9:2      ;       "\n"
```

```
a.go:11:1    func      "func"
a.go:11:6    IDENT     "FromA"
a.go:11:11   (
a.go:11:12   )
a.go:11:14   {
a.go:12:2    IDENT     "fmt"
a.go:12:5    .
a.go:12:6    IDENT     "Println"
a.go:12:13   (
a.go:12:14   STRING    "\"fromA()\""
a.go:12:23   )
a.go:12:24   ;
a.go:13:1    }
a.go:13:2    ;         "\n"
```

The output of `goScanner.go` is as simple as it can be. Note that `goScanner.go` can scan any type of file, even binary files. However, if you scan a binary file, you might get less readable output. As you can see from the output, the Go scanner adds semicolons automatically. Note that `IDENT` notifies an identifier, which is the most popular type of token.

The next subsection will deal with the parsing process.

The go/parser package

A parser reads the output of a scanner (tokens) in order to generate a structure from those tokens. Parsers use a grammar that describes a programming language to make sure that the given tokens compose a valid program. That structure is represented as a tree, which is the AST.

The use of the `go/parser` package that processes the output of `go/token` is illustrated in `goParser.go`, which is going to be presented in four parts.

The first part of `goParser.go` is as follows:

```
package main

import (
    "fmt"
    "go/ast"
    "go/parser"
    "go/token"
    "os"
    "strings"
)
type visitor int
```

The second part of `goParser.go` contains the following Go code:

```
func (v visitor) Visit(n ast.Node) ast.Visitor {
    if n == nil {
        return nil
    }
    fmt.Printf("%s%T\n", strings.Repeat("\t", int(v)), n)
    return v + 1
}
```

The `visit()` method will be called for every node of the AST.

The third part of `goParser.go` is as follows:

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Not enough arguments!")
        return
    }
}
```

The last part of `goParser.go` is as follows:

```
    for _, file := range os.Args[1:] {
        fmt.Println("Processing:", file)
        one := token.NewFileSet()
        var v visitor
        f, err := parser.ParseFile(one, file, nil,
parser.AllErrors)
        if err != nil {
            fmt.Println(err)
            return
        }
        ast.Walk(v, f)
    }
}
```

The `Walk()` function, which is called recursively, traverses an AST in depth-first order in order to visit all of its nodes.

Building and executing `goParser.go` to find the AST of a simple and small Go module will generate the following kind of output:

```
$ ./goParser a.go
Processing: a.go
*ast.File
    *ast.Ident
    *ast.GenDecl
        *ast.ImportSpec
            *ast.BasicLit
    *ast.FuncDecl
        *ast.Ident
        *ast.FuncType
            *ast.FieldList
    *ast.BlockStmt
        *ast.ExprStmt
            *ast.CallExpr
                *ast.SelectorExpr
                    *ast.Ident
                    *ast.Ident
                *ast.BasicLit
    *ast.FuncDecl
        *ast.Ident
        *ast.FuncType
            *ast.FieldList
    *ast.BlockStmt
        *ast.ExprStmt
            *ast.CallExpr
                *ast.SelectorExpr
                    *ast.Ident
                    *ast.Ident
                *ast.BasicLit
```

The output of `goParser.go` is as simple as it gets. However, it is totally different from the output of `goScanner.go`.

Now that you know how the outputs of the Go scanner and the Go parser look, you are ready to see some more practical examples.

A practical example

In this subsection, we are going to write a Go program that counts the number of times a keyword appears in the input files. In this case, the keyword that is going to be counted is `var`. The name of the utility will be `varTimes.go` and it is going to be presented in four parts. The first part of `varTimes.go` is as follows:

```
package main

import (
    "fmt"
    "go/scanner"
    "go/token"
    "io/ioutil"
    "os"
)

var KEYWORD = "var"
var COUNT = 0
```

You can search for any Go keyword you want – you can even set the value of the `KEYWORD` global variable at runtime if you modify `varTimes.go`.

The second part of `varTimes.go` contains the following Go code:

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Not enough arguments!")
        return
    }

    for _, file := range os.Args[1:] {
        fmt.Println("Processing:", file)
        f, err := ioutil.ReadFile(file)
        if err != nil {
            fmt.Println(err)
            return
        }
        one := token.NewFileSet()
        files := one.AddFile(file, one.Base(), len(f))
    }
}
```

The third part of `varTimes.go` is as follows:

```
var myScanner scanner.Scanner
myScanner.Init(files, f, nil, scanner.ScanComments)
```

```
localCount := 0
for {
    _, tok, lit := myScanner.Scan()
    if tok == token.EOF {
        break
    }
}
```

In this case, the position the token was found in is ignored as it does not matter. However, the `tok` variable is needed for finding out the end of the file.

The last part of `varTimes.go` is as follows:

```
if lit == KEYWORD {
    COUNT++
    localCount++
}
fmt.Printf("Found _%s_ %d times\n", KEYWORD, localCount)
}
fmt.Printf("Found _%s_ %d times in total\n", KEYWORD, COUNT)
}
```

Compiling and executing `varTimes.go` will create the following kind of output:

```
$ go build varTimes.go
$ ./varTimes varTimes.go variadic.go a.go
Processing: varTimes.go
Found _var_ 3 times
Processing: variadic.go
Found _var_ 0 times
Processing: a.go
Found _var_ 0 times
Found _var_ 3 times in total
```

Finding variable names with a given string length

This subsection will present another practical example that will be more advanced than the one presented in `varTimes.go`. You will see how to find the variable names with a given string length – you can use any string length you want. Additionally, the program will be able to differentiate between **global variables** and **local variables**.



A local variable is defined inside a function whereas a global variable is defined outside of a function. Global variables are also called package variables.

The name of the utility is `varSize.go` and it will be presented in four parts. The first part of `varSize.go` is as follows:

```
package main

import (
    "fmt"
    "go/ast"
    "go/parser"
    "go/token"
    "os"
    "strconv"
)

var SIZE = 2
var GLOBAL = 0
var LOCAL = 0

type visitor struct {
    Package map[*ast.GenDecl]bool
}

func makeVisitor(f *ast.File) visitor {
    k1 := make(map[*ast.GenDecl]bool)
    for _, aa := range f.Decls {
        v, ok := aa.(*ast.GenDecl)
        if ok {
            k1[v] = true
        }
    }
    return visitor{k1}
}
```

As we want to differentiate between local and global variables, we define two global variables named `GLOBAL` and `LOCAL` to keep these two counts. The use of the `visitor` structure will help us to differentiate between local and global variables, hence the `map` field defined in the `visitor` structure. The `makeVisitor()` method is used to initialize the active `visitor` structure according to the values of its parameter, which is a `File` node representing an entire file.

The second part of `varSize.go` contains the implementation of the `visit()` method:

```
func (v visitor) Visit(n ast.Node) ast.Visitor {
    if n == nil {
        return nil
    }

    switch d := n.(type) {
    case *ast.AssignStmt:
        if d.Tok != token.DEFINE {
            return v
        }

        for _, name := range d.Lhs {
            v.isItLocal(name)
        }

    case *ast.RangeStmt:
        v.isItLocal(d.Key)
        v.isItLocal(d.Value)
    case *ast.FuncDecl:
        if d.Recv != nil {
            v.CheckAll(d.Recv.List)
        }

        v.CheckAll(d.Type.Params.List)
        if d.Type.Results != nil {
            v.CheckAll(d.Type.Results.List)
        }

    case *ast.GenDecl:
        if d.Tok != token.VAR {
            return v
        }

        for _, spec := range d.Specs {
            value, ok := spec.(*ast.ValueSpec)
            if ok {
                for _, name := range value.Names {
                    if name.Name == "_" {
                        continue
                    }

                    if v.Package[d] {
                        if len(name.Name) == SIZE {
                            fmt.Printf("## %s\n", name.Name)
                            GLOBAL++
                        }
                    } else {
                }
            }
        }
    }
}
```

```
        if len(name.Name) == SIZE {
            fmt.Printf("* %s\n", name.Name)
            LOCAL++
        }
    }
}
}
}
return v
}
```

The main job of the `visit()` function is to determine the type of the node that it works with in order to act accordingly. This happens with the help of a `switch` statement.

The `ast.AssignStmt` node represents assignments or short variable declarations. The `ast.RangeStmt` node is a structure type for representing a `for` statement with a `range` clause – this is another place where new local variables are declared.

The `ast.FuncDecl` node is a structure type for representing function declarations – every variable that is defined inside a function is a local variable. Lastly, the `ast.GenDecl` node is a structure type for representing an import, constant, type, or variable declaration. However, we are only interested in `token.VAR` tokens.

The third part of `varSize.go` is as follows:

```
func (v visitor) isItLocal(n ast.Node) {
    identifier, ok := n.(*ast.Ident)
    if ok == false {
        return
    }

    if identifier.Name == "_" || identifier.Name == "" {
        return
    }

    if identifier.Obj != nil && identifier.Obj.Pos() == identifier.Pos() {
        if len(identifier.Name) == SIZE {
            fmt.Printf("* %s\n", identifier.Name)
            LOCAL++
        }
    }
}

func (v visitor) CheckAll(fs []*ast.Field) {
    for _, f := range fs {
        for _, name := range f.Names {
```

```

        v.isItLocal(name)
    }
}

```

These two functions are helper methods. The first one decides whether an identifier node is a local variable or not, and the second one visits an `ast.Field` node in order to examine its contents for local variables.

The last part of `varSize.go` is as follows:

```

func main() {
    if len(os.Args) <= 2 {
        fmt.Println("Not enough arguments!")
        return
    }

    temp, err := strconv.Atoi(os.Args[1])
    if err != nil {
        SIZE = 2
        fmt.Println("Using default SIZE:", SIZE)
    } else {
        SIZE = temp
    }

    var v visitor
    all := token.NewFileSet()
    for _, file := range os.Args[2:] {
        fmt.Println("Processing:", file)
        f, err := parser.ParseFile(all, file, nil, parser.AllErrors)
        if err != nil {
            fmt.Println(err)
            continue
        }

        v = makeVisitor(f)
        ast.Walk(v, f)
    }
    fmt.Printf("Local: %d, Global:%d with a length of %d.\n", LOCAL, GLOBAL, SIZE)
}

```

The program generates the AST of its input and processes that in order to extract the desired information. Apart from the `visit()` method, which is part of the interface, the rest of the magic in the `main()` function happens with the help of `ast.Walk()`, which automatically visits all the AST nodes of each file that is being processed.

Building and executing `varSize.go` will generate the following kind of output:

```

$ go build varSize.go
$ ./varSize
Not enough arguments!

```

```
$ ./varSize 2 varSize.go variadic.go
Processing: varSize.go
* k1
* aa
* ok
* ok
* ok
* fs
Processing: variadic.go
Local: 6, Global:0 with a length of 2.
$ ./varSize 3 varSize.go variadic.go
Processing: varSize.go
* err
* all
* err
Processing: variadic.go
* sum
* sum
Local: 5, Global:0 with a length of 3.
$ ./varSize 7 varSize.go variadic.go
Processing: varSize.go
Processing: variadic.go
* message
Local: 1, Global:0 with a length of 7.
```

You can remove the various `fmt.Println()` calls in `varSize.go` and have a less cluttered output.

You can do pretty ingenious things in Go once you know how to parse a Go program – you can even write your own parser for your own programming language if you want! If you are really into parsing, you should have a look at the documentation page of the `go/ast` package, as well as its source code, which can be found at <https://golang.org/pkg/go/ast/> and at <https://github.com/golang/go/tree/master/src/go/ast> respectively.

Text and HTML templates

The subject of this section will probably surprise you in a good way, because both presented packages give you so much flexibility that I am sure you will find many creative ways to use them. **Templates** are mainly used for separating the formatting part and the data part of the output. Please note that a Go template can be either a file or a string – the general idea is to use inline strings for smaller templates and external files for bigger ones.



You cannot import both `text/template` and `html/template` on the same Go program because these two packages share the same package name (`template`). If absolutely necessary, you should define an alias for one of them. See the `useStrings.go` code in Chapter 4, The Uses of Composite Types.

Text output is usually presented on your screen, whereas HTML output is seen with the help of a web browser. However, as text output is usually better than HTML output, if you think that you will need to process the output of a Go utility using other UNIX command-line utilities, you should use `text/template` instead of `html/template`.

Note that both the `text/template` and `html/template` packages are good examples of how sophisticated a Go package can be. As you will see shortly, both packages support their own kind of programming language – good software makes complex things look simple and elegant.

Generating text output

If you need to create plain text output, then using the `text/template` package is a good choice. The use of the `text/template` package will be illustrated in `textT.go`, which will be presented in five parts.

As templates are usually stored in external files, the example presented will use the `text.gotext` template file, which will be analyzed in three parts. Data is typically read from text files or from the Internet. However, for reasons of simplicity, the data for `textT.go` will be hardcoded in the program using a slice.

We will start by looking at the Go code of `textT.go`. The first code portion of `textT.go` is as follows:

```
package main

import (
    "fmt"
    "os"
    "text/template"
)
```

The second code segment from `textT.go` is as follows:

```
type Entry struct {
    Number int
    Square int
}
```

You will need to define a new data type for storing your data unless you are dealing with very simplistic data.

The third part of `textT.go` is shown in the following Go code:

```
func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Need the template file!")
        return
    }
}
```

```
|     tFile := arguments[1]
|     DATA := [][]int{{-1, 1}, {-2, 4}, {-3, 9}, {-4, 16}}
```

The `DATA` variable, which is a slice with two dimensions, holds the initial version of your data.

The fourth part of `textT.go` contains the following Go code:

```
|     var Entries []Entry
|
|     for _, i := range DATA {
|         if len(i) == 2 {
|             temp := Entry{Number: i[0], Square: i[1]}
|             Entries = append(Entries, temp)
|         }
|     }
```

The preceding code creates a **slice of structures** from the `DATA` variable.

The last code segment from `textT.go` is the following:

```
|     t := template.Must(template.ParseGlob(tFile))
|     t.Execute(os.Stdout, Entries)
| }
```

The `template.Must()` function is used for making the required initializations. Its return data type is `Template`, which is a structure that holds the representation of a parsed template. `template.ParseGlob()` reads the external template file. Note that I prefer to use the `.gohtml` extension for the template files, but you can use any extension that you want – just be consistent.

Lastly, the `template.Execute()` function does all the work, which includes processing the data and printing the output to the desired file, which in this case is `os.Stdout`.

Now it is time to look at the code of the template file. The first part of the text template file is as follows:

```
|Calculating the squares of some integers
```

Note that empty lines in a text template file are significant and will be shown as empty lines in the final output.

The second part of the template is as follows:

```
| {{ range . }} The square of {{ printf "%d" .Number}} is {{ printf  
" %d" .Square}}
```

There are many interesting things happening here. The `range` keyword allows you to iterate over the lines of the input, which is given as a slice of structures. Plain text is printed as such, whereas variables and dynamic text must begin with `{{` and end with `}}`. The fields of the structure are accessed as `.Number` and `.Square`. Note the dot character in front of the field name of the `Entry` data type. Lastly, the `printf` command is used to format the final output.

The third part of the `text.gotext` file is as follows:

```
| {{ end }}
```

A `{{ range }}` command is ended with `{{ end }}`. Accidentally putting `{{ end }}` in the wrong place will affect your output. Once again, keep in mind that empty lines in text template files are significant and will be shown in the final output.

Executing `textT.go` will generate the following type of output:

```
$ go run textT.go text.gotext  
Calculating the squares of some integers  
  
The square of -1 is 1  
The square of -2 is 4  
The square of -3 is 9  
The square of -4 is 16
```

Constructing HTML output

This section illustrates the use of the `html/template` package with an example named `htmlT.go`. It will be presented in six parts. The philosophy of the `html/template` package is the same as the `text/template` package. The main difference between these two packages is that `html/template` generates HTML output that is safe against code injection.



Although you can create HTML output with the `text/template` package – after all, HTML is just plain text – if you want to create HTML output, then you should use the `html/template` package instead.

For reasons of simplicity, the following presented will read data from an SQLite database, but you can use any database that you want, provided that you have or you can write the appropriate Go drivers. To make things even easier, the example will populate a database table before reading from it.

The first code portion from `htmlT.go` is as follows:

```
package main

import (
    "database/sql"
    "fmt"
    "github.com/matttn/go-sqlite3"
    "html/template"
    "net/http"
    "os"
)

type Entry struct {
    Number int
    Double int
    Square int
}

var DATA []Entry
var tFile string
```

You can see a new package named `net/http` in the `import` block, which is used for creating HTTP servers and clients in Go. You will learn more about network programming in Go and the use of the `net` and `net/http` standard Go packages in [Chapter 12, *The Foundations of Network Programming in Go*](#),

and [Chapter 13](#), *Network Programming – Building Your Own Servers and Clients*.

Apart from `net/http`, you can also see the definition of the `Entry` data type that will hold the records read from the SQLite3 table, as well as two global variables named `DATA` and `tFile`, which hold the data that is going to be passed to the template file and the filename of the template file, respectively.

Lastly, you can see the use of the <https://github.com/mattn/go-sqlite3> package for communicating with the SQLite3 database with the help of the `database/sql` interface.

The second part of `htmlT.go` is as follows:

```
func myHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Printf("Host: %s Path: %s\n", r.Host, r.URL.Path)
    myT := template.Must(template.ParseGlob(tFile))
    myT.ExecuteTemplate(w, tFile, DATA)
}
```

The simplicity and the effectiveness of the `myHandler()` function is phenomenal, especially if you consider the size of the function! The `template.ExecuteTemplate()` function does all the work for us. Its first parameter is the variable that holds the connection with the HTTP client, its second parameter is the template file that will be used for formatting the data, and its third parameter is the slice of structures with the data.

The third code segment from `htmlT.go` is shown in the following Go code:

```
func main() {
    arguments := os.Args
    if len(arguments) != 3 {
        fmt.Println("Need Database File + Template File!")
        return
    }

    database := arguments[1]
    tFile = arguments[2]
```

The fourth code portion from `htmlT.go` is where you start dealing with the database:

```

db, err := sql.Open("sqlite3", database)
if err != nil {
    fmt.Println(nil)
    return
}

fmt.Println("Emptying database table.")
_, err = db.Exec("DELETE FROM data")
if err != nil {
    fmt.Println(nil)
    return
}

fmt.Println("Populating", database)
stmt, _ := db.Prepare("INSERT INTO data(number, double,
square) values(?, ?, ?)")
for i := 20; i < 50; i++ {
    _, _ = stmt.Exec(i, 2*i, i*i)
}

```

The `sql.Open()` function opens the connection with the desired database. With `db.Exec()`, you can execute database commands without expecting any feedback from them. Lastly, the `db.Prepare()` function allows you to execute a database command multiple times by changing only its parameters and calling `Exec()` afterward.

The fifth part of `htmlT.go` contains the following Go code:

```

rows, err := db.Query("SELECT * FROM data")
if err != nil {
    fmt.Println(nil)
    return
}

var n int
var d int
var s int
for rows.Next() {
    err = rows.Scan(&n, &d, &s)
    temp := Entry{Number: n, Double: d, Square: s}
    DATA = append(DATA, temp)
}

```

In this part of the program, we read the data from the desired table using `db.Query()` and multiple calls to `Next()` and `Scan()`. While reading the data, you put it into a slice of structures and you are done dealing with the database.

The last part of the program is all about setting up the web server, and it contains the following Go code:

```
    http.HandleFunc("/", myHandler)
    err = http.ListenAndServe(":8080", nil)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

Here, the `http.HandleFunc()` function tells the web server embedded in the program which URLs will be supported and by which handler function (`myHandler()`). The current handler supports the `/` URL, which in Go matches all URLs. This saves you from having to create any extra static or dynamic pages.

The code of the `htmlT.go` program is divided into two virtual parts. The first part is about getting the data from the database and putting it into a slice of structures, whereas the second part, which is similar to `textT.go`, is about displaying your data in a web browser.



The two biggest advantages of SQLite are that you do not need to run a server process for the database server and that SQLite databases are stored in self-contained files, which means that single files hold entire SQLite databases.

Note that in order to reduce the Go code and be able to run the `htmlT.go` program multiple times, you will need to create the database table and the SQLite3 database manually, which is as simple as executing the following commands:

```
$ sqlite3 htmlT.db
SQLite version 3.19.3 2017-06-27 16:48:08
Enter ".help" for usage hints.
sqlite> CREATE TABLE data (
...>     number INTEGER PRIMARY KEY,
...>     double INTEGER,
...>     square INTEGER );
sqlite> ^D
$ ls -l htmlT.db
-rw-r--r-- 1 mtsouk  staff  8192 Dec 26 22:46 htmlT.db
```

The first command is executed from the UNIX shell and it is needed for creating the database file. The second command is executed from the SQLite3 shell and it has to do with creating a database table named `data`, which has three fields named `number`, `double`, and `square`.

Additionally, you are going to need an external template file, which will be named `html.gohtml`. It is going to be used for the generation of the output of the program.

The first part of `html.gohtml` is as follows:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
        <title>Doing Maths in Go!</title>
        <style>
            html {
                font-size: 14px;
            }
            table, th, td {
                border: 2px solid blue;
            }
        </style>
</head>
<body>
```

The HTML code that a web browser will get is based on the contents of `html.gohtml`. This means that you will need to create proper HTML output, hence the preceding HTML code, which also includes some inline CSS code for formatting the generated HTML table.

The second part of `html.gohtml` contains the following code:

```
<table>
    <thead>
        <tr>
            <th>Number</th>
            <th>Double</th>
            <th>Square</th>
        </tr>
    </thead>
    <tbody>
{{ range . }}
    <tr>
        <td> {{ .Number }} </td>
        <td> {{ .Double }} </td>
        <td> {{ .Square }} </td>
    </tr>
{{ end }}
    </tbody>
</table>
```

As you can see from the preceding code, you still have to use `{{ range }}` and `{{ end }}` in order to iterate over the elements of the slice of structures that

was passed to `template.ExecuteTemplate()`. However, this time the `html.gohtml` template file contains lots of HTML code in order to format the data in the slice of structures better.

The last part of the HTML template file is as follows:

```
|</body>
|</html>
```

The last part of `html.gohtml` is mainly used for properly ending the generated HTML code according to the HTML standards. Before being able to compile and execute `htmlT.go`, you will need to download the package that will help the Go programming language to communicate with SQLite3. You can do this by executing the following command:

```
| $ go get github.com/mattn/go-sqlite3
```

As you already know, you can find the source code of the downloaded package inside `~/go/src` and its compiled version inside `~/go/pkg/darwin_amd64` if you are on a macOS machine. Otherwise, check the contents of `~/go/pkg` to find out your own architecture. Note that the `~` character denotes the home directory of the current user.

Keep in mind that additional Go packages exist that can help you to communicate with an SQLite3 database. However, the one used here is the only one that currently supports the `database/sql` interface. Executing `htmlT.go` will produce the kind of output on a web browser that you can see in the following figure:

Doing Maths in Go!

localhost:8080

Number	Double	Square
20	40	400
21	42	441
22	44	484
23	46	529
24	48	576
25	50	625
26	52	676
27	54	729
28	56	784
29	58	841
30	60	900
31	62	961
32	64	1024
33	66	1089
34	68	1156
35	70	1225
36	72	1296
37	74	1369
38	76	1444
39	78	1521
40	80	1600
41	82	1681
42	84	1764
43	86	1849
44	88	1936
45	90	2025
46	92	2116
47	94	2209
48	96	2304
49	98	2401

Figure 6.1: The output of the htmlT.go program

Moreover, `htmlT.go` will generate the following type of output in your UNIX shell, which is mainly debugging information:

```
$ go run htmlT.go htmlT.db html.gohtml
Emptying database table.
Populating htmlT.db
Host: localhost:8080 Path: /
Host: localhost:8080 Path: /favicon.ico
Host: localhost:8080 Path: /123
```

If you want to see the HTML output of the program from the UNIX shell, you can use the `wget(1)` utility as follows:

```
$ wget -qO- http://localhost:8080
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
        <title>Doing Maths in Go!</title>
        <style>
            html {
                font-size: 14px;
            }
            table, th, td {
                border: 2px solid blue;
            }
        </style>
    </head>
    <body>
<table>
    <thead>
        <tr>
```

Both `text/template` and `html/template` are powerful packages that can save you a lot of time, so I suggest that you use them when they fit the requirements of your applications.

Additional resources

You will find the next resources very useful:

- Visit the documentation page of the `syscall` standard Go package at [http://golang.org/pkg/syscall/](https://golang.org/pkg/syscall/). This is one of the biggest Go documentation pages that I have ever seen!
- Visit the documentation page of the `text/template` package, which can be found at <https://golang.org/pkg/text/template/>.
- Similarly, go to <https://golang.org/pkg/html/template/> for the documentation of the `html/template` package.
- You can find out more about the `go/token` package at <https://golang.org/pkg/go/token/>.
- You can find out more about the `go/parser` package at <https://golang.org/pkg/go/parser/>.
- You can find out more about the `go/scanner` package at <https://golang.org/pkg/go/scanner/>.
- You can find out more about the `go/ast` package at <https://golang.org/pkg/go/ast/>.
- Visit the home page of SQLite3 at <https://www.sqlite.org/>.
- Watch the "Writing Beautiful Packages in Go" video by Mat Ryer at <https://www.youtube.com/watch?v=cAWlv2SeQus>.
- If you want to know about **Plan 9**, look at <https://plan9.io/plan9/>.
- Take the time to look at the `find(1)` command-line tool by visiting its man page (`man 1 find`).

Exercises

- Seek out more information about the actual implementation of the `fmt.Sprintf()` function.
- Can you write a function that sorts three `int` values? Try to write two versions of the function: one with named returned values and another without named return values. Which one do you think is better?
- Can you modify the Go code of `htmlT.go` in order to use `text/template` instead of `html/template`?
- Can you modify the Go code of `htmlT.go` in order to use either [https://git hub.com/feyeleanor/gosqlite3](https://github.com/feyeleanor/gosqlite3) or the <https://github.com/phf/go-sqlite3> package for communicating with the SQLite3 database?
- Create your own Go module and develop three major versions of it.
- Write a Go program like `htmlT.go` that reads data from a MySQL database. Write down the code changes that you made.

Summary

This chapter presented three primary topics: Go functions, Go packages, and Go modules. The main advantage of Go modules is that they record the exact dependency requirements, which makes creating reproducible builds easy and straightforward.

This chapter also offered you ample advice about developing good Go packages. It subsequently talked about the `text/template` and `html/template` packages, which allow you to create plain text and HTML output based on templates, as well as the `go/token`, `go/parser`, and `go/scanner` packages. Lastly, it talked about the `syscall` standard Go package that offers advanced features.

The next chapter will discuss two important Go features: interfaces and reflection. Additionally, it will talk about object-oriented programming in Go, debugging, and Go type methods. All of these topics are pretty advanced and you might find them difficult at first. However, learning more about them will unquestionably make you a better Go programmer.

Finally, the next chapter includes a quick introduction to the `git` utility, which was used in this chapter to create Go modules.

Reflection and Interfaces for All Seasons

The previous chapter talked about developing packages, modules, and functions in Go, as well as working with text and HTML templates with the help of the `text/template` and `html/template` packages. The chapter also explained the use of the `syscall` package.

This chapter is going to teach you three very interesting, handy, and somewhat advanced Go concepts: **reflection**, **interfaces**, and **type** methods. Although Go interfaces are used all the time, reflection is not, mainly because it is not usually necessary for your programs to use it. Furthermore, you will learn about type assertions, the Delve debugger, and **object-oriented programming** in Go. Lastly, this chapter will present a short introduction to **git** and **GitHub**.

Therefore, in this chapter, you will learn about:

- Type methods
- Go interfaces
- Type assertions
- Developing and using your own interfaces
- GitHub and git
- An introduction to the Delve debugger
- Object-oriented programming in Go
- Reflection and the `reflect` standard Go package
- Reflection and the `reflectwalk` library

Type methods

A Go type method is a function with a special receiver argument. You declare methods as ordinary functions with an additional parameter that appears in front of the function name. This particular parameter connects the function to the type of that extra parameter. As a result, that parameter is called the **receiver of the method**.

The following Go code is the implementation of the `Close()` function as found in https://golang.org/src/os/file_plan9.go:

```
func (f *File) Close() error {
    if err := f.checkValid("close"); err != nil {
        return err
    }
    return f.file.close()
}
```

The `Close()` function is a type method because there is that `(f *File)` parameter in front of its name and after the `func` keyword. The `f` parameter is called the receiver of the method. In object-oriented programming terminology this process can be described as sending a message to an **object**. In Go, the receiver of a method is defined using a regular variable name without the need to use a dedicated keyword such as `this` or `self`.

Now let me offer you a complete example using the Go code of the `methods.go` file, which will be presented in four parts.

The first part of `methods.go` comes with the following Go code:

```
package main

import (
    "fmt"
)

type twoInts struct {
    X int64
    Y int64
}
```

In the preceding Go code, you can see the definition of a new structure with two fields named `twoInts`.

The second code segment of `methods.go` is next:

```
| func regularFunction(a, b twoInts) twoInts {  
|     temp := twoInts{X: a.X + b.X, Y: a.Y + b.Y}  
|     return temp  
| }
```

In this part, you define a new function named `regularFunction()` that accepts two parameters of type `twoInts` and returns just one `twoInts` value.

The third part of the program contains the next Go code:

```
| func (a twoInts) method(b twoInts) twoInts {  
|     temp := twoInts{X: a.X + b.X, Y: a.Y + b.Y}  
|     return temp  
| }
```

The `method()` function is equivalent to the `regularFunction()` function defined in the previous part of `methods.go`. However, the `method()` function is a type method and you are going to learn a different way of calling it in a moment.



The really interesting thing here is that the implementation of `method()` is exactly the same as the implementation of `regularFunction()`!

The last code segment of `methods.go` is the following:

```
| func main() {  
|     i := twoInts{X: 1, Y: 2}  
|     j := twoInts{X: -5, Y: -2}  
|     fmt.Println(regularFunction(i, j))  
|     fmt.Println(i.method(j))  
| }
```

As you can see, the way you call a type method (`i.method(j)`) is different from the way you call a conventional function (`regularFunction(i, j)`).

Executing `methods.go` will create the next output:

```
$ go run methods.go  
{-4 0}  
{-4 0}
```

Notice that type methods are also associated with interfaces, which will be the subject of the next section. As a result, you will see more type methods later.

Go interfaces

Strictly speaking, a Go interface type defines the behavior of other types by specifying a set of methods that need to be implemented. For a type to satisfy an interface, it needs to implement all the methods required by that interface, which are usually not too many.

Putting it simply, interfaces are **abstract types** that define a set of functions that need to be implemented so that a type can be considered an instance of the interface. When this happens, we say that the type satisfies this interface. So, an interface is two things: a set of methods and a type, and it is used to define the behavior of other types.

The biggest advantage you get from having and using an interface is that you can pass a variable of a type that implements that particular interface to any function that expects a parameter of that specific interface. Without that amazing capability, interfaces would have been only a formality without any practical or real benefit.



Please note that if you find yourself defining an interface and its implementation in the same Go package, you might need to rethink your approach. This is not because this is technically wrong, but because it looks logically wrong.

Two very common Go interfaces are `io.Reader` and `io.Writer` and they are used in file input and output operations. More specifically, `io.Reader` is used for reading from a file, whereas `io.Writer` is used for writing to a file of any type.

The definition of `io.Reader` as found in <https://golang.org/src/io/io.go> is the following:

```
| type Reader interface {  
|     Read(p []byte) (n int, err error)  
| }
```

So, in order for a type to satisfy the `io.Reader` interface, you will need to implement the `Read()` method as described in the interface definition.

Similarly, the definition of `io.Writer` as found in <https://golang.org/src/io/io.go> is next:

```
| type Writer interface {  
|     Write(p []byte) (n int, err error)  
| }
```

To satisfy the `io.Writer` interface, you will just need to implement a single method named `Write()`.

Each of the `io.Reader` and `io.Writer` interfaces requires the implementation of just one method. Yet both interfaces are very powerful – most likely their power comes from their simplicity. Generally speaking, most interfaces are fairly simple.

In the next subsections, you will learn how to define an interface on your own and how to use it in other Go packages. Notice that it is not necessary for an interface to be fancy or impressive as long as it does what you want it to do.



Putting it simply, interfaces should be utilized when there is a need for making sure that certain conditions will be met and certain behaviors will be anticipated from a Go element.

About type assertions

A type assertion is the `x. (T)` notation, where `x` is an interface type and `T` is a type. Additionally, the actual value stored in `x` is of type `T` and `T` must satisfy the interface type of `x`. The following paragraphs, as well as the code example, will help you to clarify this relatively eccentric definition of a type assertion.

Type assertions help you to do *two things*. The first thing is checking whether an interface value keeps a particular type. When used this way, a type assertion returns two values: the underlying value and a `bool` value. Although the underlying value is what you might want to use, the Boolean value tells you whether the type assertion was successful or not.

The second thing that type assertions help with is allowing you to use the concrete value stored in an interface or assign it to a new variable. This means that if there is an `int` variable in an interface, you can get that value using a type assertion.

However, if a type assertion is not successful and you do not handle that failure on your own, your program will panic. Look at the Go code of the `assertion.go` program, which will be presented in two parts. The first part contains the following Go code:

```
package main

import (
    "fmt"
)

func main() {
    var myInt interface{} = 123

    k, ok := myInt.(int)
    if ok {
        fmt.Println("Success:", k)
    }

    v, ok := myInt.(float64)
    if ok {
        fmt.Println(v)
    } else {
        fmt.Println("Failed without panicking!")
    }
}
```

First, you declare the `myInt` variable that has a dynamic type `int` and value `123`. Then, you use a type assertion twice to testing the interface of the `myInt` variable – once for `int` and once for `float64`.

As the `myInt` variable does not contain a `float64` value, the `myInt.(float64)` type assertion will fail unless handled properly. Fortunately, in this case, the correct use of the `ok` variable will save your program from panicking.

The second part comes with the next Go code:

```
i := myInt.(int)
fmt.Println("No checking:", i)

j := myInt.(bool)
fmt.Println(j)
}
```

There are two type assertions taking place here. The first type assertion is successful, so there will be no problem with that. But let me talk a little bit more about this particular type assertion. The type of variable `i` will be `int` and its value will be `123`, which is the value stored in `myInt`. So, as `int` satisfies the `myInt` interface, which in this case happens because the `myInt` interface requires no functions to be implemented, the value of `myInt.(int)` is an `int` value.

However, the second type assertion, which is `myInt.(bool)`, will trigger a panic because the underlying value of `myInt` is not Boolean (`bool`).

Therefore, executing `assertion.go` will generate the following output:

```
$ go run assertion.go
Success: 123
Failed without panicking!
No cheking: 123
panic: interface conversion: interface {} is int, not bool

goroutine 1 [running]:
main.main()
    /Users/mtsouk/Desktop/mGo2nd/Mastering-Go-Second-Edition/ch07/assertion.go:25 +0x1c1
exit status 2
```

Go states pretty clearly the reason for panicking: `interface {} is int, not bool`.

Generally speaking, when using interfaces, expect to use type assertions as well. You will see more type assertions in the `useInterface.go` program coming up.

Writing your own interfaces

In this section, you will learn how to develop your own interfaces, which is a relatively easy process as long as you know what you want to develop.

The technique is going to be illustrated using the Go code of `myInterface.go`, which will be presented shortly. The interface that is going to be created will help you to work with geometric shapes of the plane.

The Go code of `myInterface.go` is next:

```
package myInterface

type Shape interface {
    Area() float64
    Perimeter() float64
}
```

The definition of the `shape` interface is truly straightforward as it requires that you implement just two functions, named `Area()` and `Perimeter()`, that both return a `float64` value. The first function will be used to calculate the area of a shape in the plane and the second one to calculate the perimeter of a shape in the plane.

After that, you will need to install the `myInterface.go` package and make it available to the current user. As you already know, the installation process involves the execution of the following UNIX commands:

```
$ mkdir ~/go/src/myInterface
$ cp myInterface.go ~/go/src/myInterface
$ go install myInterface
```

Using a Go interface

This subsection will teach you how to use the interface defined in `myInterface.go` in a Go program named `useInterface.go`, which will be presented in five parts.

The first part of `useInterface.go` comes with the next Go code:

```
package main

import (
    "fmt"
    "math"
    "myInterface"
)

type square struct {
    X float64
}

type circle struct {
    R float64
}
```

As the desired interface is defined in its own package, it should come as no surprise that you are importing the `myInterface` package.

The second code portion from `useInterface.go` contains the following code:

```
func (s square) Area() float64 {
    return s.X * s.X
}

func (s square) Perimeter() float64 {
    return 4 * s.X
}
```

In this part, you implement the `shape` interface for the `square` type.

The third part contains the next Go code:

```
func (s circle) Area() float64 {
    return s.R * s.R * math.Pi
}

func (s circle) Perimeter() float64 {
```

```
|     return 2 * s.R * math.Pi  
| }
```

In this part, you implement the `shape` interface for the `circle` type.

The fourth part of `useInterface.go` comes with the following Go code:

```
func Calculate(x myInterface.Shape) {  
    _, ok := x.(circle)  
    if ok {  
        fmt.Println("Is a circle!")  
    }  
  
    v, ok := x.(square)  
    if ok {  
        fmt.Println("Is a square:", v)  
    }  
  
    fmt.Println(x.Area())  
    fmt.Println(x.Perimeter())  
}
```

So, in the preceding code, you implement one function that requires a single `shape` parameter (`myInterface.Shape`). The magic here should be more obvious once you understand that it requires any `shape` parameter, which is any parameter whose type implements the `shape` interface.

The code at the beginning of the function shows how you can differentiate between the various data types that implement the desired interface. In the second block, you can see how you can find out the values stored in a `square` parameter – you can use this technique for any type that implements the `myInterface.Shape` interface.

The last code segment includes the next code:

```
func main() {  
    x := square{X: 10}  
    fmt.Println("Perimeter:", x.Perimeter())  
    Calculate(x)  
    y := circle{R: 5}  
    Calculate(y)  
}
```

In this part, you can see how you can use both `circle` and `square` variables as parameters to the `Calculate()` function you implemented earlier.

If you execute `useInterface.go`, you will get the next output:

```
$ go run useInterface.go
Perimeter: 40
Is a square: {10}
100
40
Is a circle!
78.53981633974483
31.41592653589793
```

Using switch with interface and data types

In this subsection, you will learn how to use the `switch` statement to differentiate between different data types using the Go code of `switch.go`, which will be presented in four parts. The Go code of `switch.go` is partially based on `useInterface.go`, but it will add another type named `rectangle` and will not need to implement the methods of any interface.

The first part of the program is next:

```
package main
import (
    "fmt"
)
```

As the code in `switch.go` will not work with the interface defined in `myInterface.go`, there is no need to import the `myInterface` package.

The second part is where you define the three new data types that will be used in the program:

```
type square struct {
    X float64
}

type circle struct {
    R float64
}

type rectangle struct {
    X float64
    Y float64
}
```

All three types are pretty simple.

The third code segment from `switch.go` comes with the following Go code:

```
func tellInterface(x interface{}) {
    switch v := x.(type) {
```

```

    case square:
        fmt.Println("This is a square!")
    case circle:
        fmt.Printf("%v is a circle!\n", v)
    case rectangle:
        fmt.Println("This is a rectangle!")
    default:
        fmt.Printf("Unknown type %T!\n", v)
    }
}

```

Here you can see the implementation of a function named `tellInterface()` with a single parameter named `x` and `type interface{}`.

This trick will help you to differentiate between the different data types of the `x` parameter. All the magic is performed with the use of the `x.(type)` statement that returns the type of the `x` element. The `%v` verb used in `fmt.Printf()` allows you to acquire the value of the type.

The last part of `switch.go` contains the implementation of the `main()` function:

```

func main() {
    x := circle{R: 10}
    tellInterface(x)
    y := rectangle{X: 4, Y: 1}
    tellInterface(y)
    z := square{X: 4}
    tellInterface(z)
    tellInterface(10)
}

```

Executing `switch.go` will generate the next kind of output:

```

$ go run switch.go
{10} is a circle!
This is a rectangle!
This is a square!
Unknown type int!

```

Reflection

Reflection is an advanced Go feature that allows you to dynamically learn the type of an arbitrary object, as well as information about its structure. Go offers the `reflect` package for working with reflection. What you should remember is that you will most likely not need to use reflection in every Go program. So, the first two questions are: why is reflection necessary and when should you use it?

Reflection is necessary for the implementation of packages such as `fmt`, `text/template`, and `html/template`. In the `fmt` package, reflection saves you from having to explicitly deal with every data type that exists. However, even if you had the patience to write code to work with every data type that you know of, you would still not be able to work with all possible types! In this case, reflection makes it possible for the methods of the `fmt` package to find the structure and to work with new types.

Therefore, you might need to use reflection when you want to be as generic as possible or when you want to make sure that you will be able to deal with data types that do not exist at the time of writing your code but might exist in the future. Additionally, reflection is handy when working with values of types that do not implement a common interface.



As you can see, reflection helps you to work with unknown types and unknown values of types. However, that flexibility comes at a cost.

The stars of the `reflect` package are two types named `reflect.Value` and `reflect.Type`. The former type is used for storing values of any type, whereas the latter type is used for representing Go types.

A simple reflection example

This section will present you with a relatively simple reflection example in order to help you to feel comfortable with this advanced Go feature.

The name of the Go source file is `reflection.go` and it will be presented to you in four parts. The purpose of `reflection.go` is to examine an "unknown" structure variable and find out more about it at runtime. In order to be more interesting, the program will define two new `struct` types. Based on these two types, it will also define two new variables; however, it will only examine one of them.

If the program has no command-line arguments, it will examine the first one, otherwise it will explore the second one – practically, this means that the program will not know in advance (at runtime) the kind of `struct` variable it will have to process.

The first part of `reflection.go` contains the next Go code:

```
package main

import (
    "fmt"
    "os"
    "reflect"
)

type a struct {
    X int
    Y float64
    Z string
}

type b struct {
    F int
    G int
    H string
    I float64
}
```

In this part, you can see the definition of the `struct` data types that will be used in the program.

The second code segment from `reflection.go` is the following:

```
func main() {
    x := 100
    xRefl := reflect.ValueOf(&x).Elem()
    xType := xRefl.Type()
    fmt.Printf("The type of x is %s.\n", xType)
```

The preceding Go code presents a small and naive reflection example. First you declare a variable named `x` and then you call the `reflect.ValueOf(&x).Elem()` function. Then you call `xRefl.Type()` in order to get the type of the variable, which is stored in `xType`. These three lines of code illustrate how you can get the data type of a variable using reflection. However, if all you care about is the data type of a variable, you can just call `reflect.TypeOf(x)` instead.

The third code portion from `reflection.go` contains the next Go code:

```
A := a{100, 200.12, "Struct a"}
B := b{1, 2, "Struct b", -1.2}
var r reflect.Value

arguments := os.Args
if len(arguments) == 1 {
    r = reflect.ValueOf(&A).Elem()
} else {
    r = reflect.ValueOf(&B).Elem()
}
```

In this part, you declare two variables named `A` and `B`. The type of the `A` variable is `a` and the type of the `B` variable is `b`. The type of the `r` variable should be `reflect.Value` because this is what the `reflect.ValueOf()` function returns. The `Elem()` method returns the value contained in the reflection interface (`reflect.Value`).

The last part of `reflection.go` is next:

```
iType := r.Type()
fmt.Printf("i Type: %s\n", iType)
fmt.Printf("The %d fields of %s are:\n", r.NumField(), iType)

for i := 0; i < r.NumField(); i++ {
    fmt.Printf("Field name: %s ", iType.Field(i).Name)
    fmt.Printf("with type: %s ", r.Field(i).Type())
    fmt.Printf("and value %v\n", r.Field(i).Interface())
}
```

In this part of the program, you use the appropriate functions of the `reflect` package in order to obtain the desired information. The `NumField()` method returns the number of fields in a `reflect.Value` structure, whereas the `Field()` function returns the field of the structure that is specified by its parameter. The `Interface()` function returns the value of a field of the `reflect.Value` structure as an interface.

Executing `reflection.go` twice will generate the following output:

```
$ go run reflection.go 1
The type of x is int.
i Type: main.b
The 4 fields of main.b are:
Field name: F with type: int and value 1
Field name: G with type: int and value 2
Field name: H with type: string and value Struct b
Field name: I with type: float64 and value -1.2
$ go run reflection.go
The type of x is int.
i Type: main.a
The 3 fields of main.a are:
Field name: X with type: int and value 100
Field name: Y with type: float64 and value 200.12
Field name: Z with type: string and value Struct a
```

It is important to note that Go uses its internal representation to print the data types of variables `A` and `B`, which are `main.a` and `main.b`, respectively. However, this is not the case with variable `x`, which is an `int`.

A more advanced reflection example

In this section, we are going to see more advanced uses of reflection illustrated in relatively small code blocks using the Go code of `advRef1.go`.

The `advRef1.go` program will be presented in five parts – its first part is next:

```
package main

import (
    "fmt"
    "os"
    "reflect"
)

type t1 int
type t2 int
```

Note that although both `t1` and `t2` types are based on `int`, and therefore are essentially the same type as `int`, Go treats them as totally different types. Their internal representation after Go parses the code of the program will be `main.t1` and `main.t2`, respectively.

The second code portion from `advRef1.go` is the following:

```
type a struct {
    X    int
    Y    float64
    Text string
}

func (a1 a) compareStruct(a2 a) bool {
    r1 := reflect.ValueOf(&a1).Elem()
    r2 := reflect.ValueOf(&a2).Elem()

    for i := 0; i < r1.NumField(); i++ {
        if r1.Field(i).Interface() != r2.Field(i).Interface() {
            return false
        }
    }
    return true
}
```

In this code, you define a Go structure type named `a` and implement a Go function named `compareStruct()`. The purpose of this function is to find out whether two variables of the `a` type are exactly the same or not. As you can see, `compareStruct()` uses Go code from `reflection.go` to perform its task.

The third code segment of `advRefl.go` comes with the following Go code:

```
func printMethods(i interface{}) {
    r := reflect.ValueOf(i)
    t := r.Type()
    fmt.Printf("Type to examine: %s\n", t)

    for j := 0; j < r.NumMethod(); j++ {
        m := r.Method(j).Type()
        fmt.Println(t.Method(j).Name, "-->", m)
    }
}
```

The `printMethods()` function prints the methods of a variable. The variable type that will be used in `advRefl.go` to illustrate `printMethods()` will be `os.File`.

The fourth code segment from `advRefl.go` contains the following Go code:

```
func main() {
    x1 := t1(100)
    x2 := t2(100)
    fmt.Printf("The type of x1 is %s\n", reflect.TypeOf(x1))
    fmt.Printf("The type of x2 is %s\n", reflect.TypeOf(x2))

    var p struct{}
    r := reflect.New(reflect.ValueOf(&p).Type()).Elem()
    fmt.Printf("The type of r is %s\n", reflect.TypeOf(r))
```

The last code portion of `advRefl.go` is as follows:

```
a1 := a{1, 2.1, "A1"}
a2 := a{1, -2, "A2"}

if a1.compareStruct(a1) {
    fmt.Println("Equal!")
}

if !a1.compareStruct(a2) {
    fmt.Println("Not Equal!")
}

var f *os.File
printMethods(f)
```

As you will see later, the `a1.CompareStruct(a1)` call returns `true` because we are comparing `a1` with itself, whereas the `a1.CompareStruct(a2)` call will return `false` because the `a1` and `a2` variables have different values.

Executing `advRefl.go` will create the following output:

```
$ go run advRefl.go
The type of x1 is main.t1
The type of x2 is main.t2
The type of r is reflect.Value
Equal!
Not Equal!
Type to examine: *os.File
Chdir --> func() error
Chmod --> func(os.Mode) error
Chown --> func(int, int) error
Close --> func() error
Fd --> func() uintptr
Name --> func() string
Read --> func([]uint8) (int, error)
ReadAt --> func([]uint8, int64) (int, error)
Readdir --> func(int) ([]os.FileInfo, error)
Readdirnames --> func(int) ([]string, error)
Seek --> func(int64, int) (int64, error)
Stat --> func() (os.FileInfo, error)
Sync --> func() error
Truncate --> func(int64) error
Write --> func([]uint8) (int, error)
WriteAt --> func([]uint8, int64) (int, error)
WriteString --> func(string) (int, error)
```

You can see that the type of the `r` variable, which is returned by `reflect.New()`, will be `reflect.Value`. Additionally, the output of the `printMethods()` method tells us that the `*os.File` type supports a plethora of methods, such as `Chdir()` and `Chmod()`.

The three disadvantages of reflection

Without a doubt, reflection is a powerful Go feature. However, as with all tools, reflection should be used rationally for three main reasons. The first reason is that extensive use of reflection will make your programs hard to read and maintain. A potential solution to this problem is good documentation, but developers are famous for not having the time to write the required documentation.

The second reason is that the Go code that uses reflection will make your programs slower. Generally speaking, Go code that is made to work with a particular data type will always be faster than Go code that uses reflection to dynamically work with any Go data type. Additionally, such dynamic code will make it difficult for tools to refactor or analyze your code.

The last reason is that reflection errors cannot be caught at build time and are reported at runtime as panics, which means that reflection errors can potentially crash your programs.

This can happen months or even years after the development of a Go program! One solution to this problem is extensive testing before a dangerous function call. However, this will add even more Go code to your programs, which will make them even slower.

The reflectwalk library

The `reflectwalk` library allows you to walk complex values in Go using reflection in a way that is similar to the way you walk a filesystem. The following example, which is going to walk a structure, is called `walkRef.go` and it is going to be presented in five parts.

The first part of `walkRef.go` is as follows:

```
package main

import (
    "fmt"
    "github.com/mitchellh/reflectwalk"
    "reflect"
)

type Values struct {
    Extra map[string]string
}
```

As `reflectwalk` is not a standard Go package, you will need to call it using its full address.

The second part of `walkRef.go` is as follows:

```
type WalkMap struct {
    MapVal reflect.Value
    Keys    map[string]bool
    Values  map[string]bool
}

func (t *WalkMap) Map(m reflect.Value) error {
    t.MapVal = m
    return nil
}
```

The `Map()` function is required by an interface that is defined in `reflectwalk` and used for searching maps.

The third part of `walkRef.go` is as follows:

```
func (t *WalkMap) MapElem(m, k, v reflect.Value) error {
    if t.Keys == nil {
        t.Keys = make(map[string]bool)
    }
    t.Values[k] = v.String()
}
```

```

        t.Values = make(map[string]bool)
    }

    t.Keys[k.Interface().(string)] = true
    t.Values[v.Interface().(string)] = true
    return nil
}

```

The fourth part of `walkRef.go` is as follows:

```

func main() {
    w := new(WalkMap)

    type S struct {
        Map map[string]string
    }

    data := &S{
        Map: map[string]string{
            "V1": "v1v",
            "V2": "v2v",
            "V3": "v3v",
            "V4": "v4v",
        },
    }

    err := reflectwalk.Walk(data, w)
    if err != nil {
        fmt.Println(err)
        return
    }
}

```

Here, you define a new variable named `data` that holds the map and you call `reflectwalk.Walk()` to learn more about it.

The last part of `walkRef.go` is as follows:

```

    r := w.MapVal
    fmt.Println("MapVal:", r)
    rType := r.Type()
    fmt.Printf("Type of r: %s\n", rType)

    for _, key := range r.MapKeys() {
        fmt.Println("key:", key, "value:", r.MapIndex(key))
    }
}

```

The last part of `walkRef.go` shows how to use reflection to print the contents of the `MapVal` field of the `WalkMap` structure. The `MapKeys()` method returns a slice of `reflect.Values` – each value holds a single map key. The `MapIndex()` method allows you to print the value of a key. The `MapKeys()` and `MapIndex()` methods

only work with the `reflect.Map` type and allow you to iterate over a map – the order of the returned map elements will be random.

Before using the `reflectwalk` library for the first time, you will need to download it, which can be done as follows:

```
$ go get github.com/mitchellh/reflectwalk
```



If you decide to use Go modules, the process of downloading the `reflectwalk` library will be much simpler and automated.

Executing `walkRef.go` will generate the following output:

```
$ go run walkRef.go
MapVal: map[V1:v1v V2:v2v V3:v3v V4:v4v]
Type of r: map[string]string
key: V2 value: v2v
key: V3 value: v3v
key: V4 value: v4v
key: V1 value: v1v
```

Object-oriented programming in Go

You should know by now that Go does not use inheritance; instead, it supports composition. Go interfaces provide a kind of polymorphism. So, although Go is not an object-oriented programming language, it has some features that allow us to mimic object-oriented programming.



If you really want to develop applications using the object-oriented methodology, then choosing Go might not be your best option. As I am not really into Java, I would suggest looking at C++ or Python instead.

First, let me explain to you the two techniques that will be used in the Go program of this section. The first technique uses methods in order to associate a function with a type, which means that in some ways, the function and the type construct an object. In the second technique, you embed a type into a new structure type in order to create a kind of hierarchy.

There is also a third technique where you use a Go interface to make two or more elements *objects of the same class*. This technique is not going to be illustrated in this section as it was shown earlier in this chapter.

The key point here is that a Go interface allows you to define a common behavior between different elements such that all these different elements share the characteristics of an object. This might permit you to say that these different elements are objects of the same class; however, objects and classes of an actual object-oriented programming language can do many more things.

The first two techniques will be illustrated in `ooo.go`, which will be presented in four parts. The first code segment from `ooo.go` contains the next Go code:

```
package main  
  
import (  
    "fmt"
```

```

)
type a struct {
    XX int
    YY int
}

type b struct {
    AA string
    XX int
}

```

The second part of the program is the following:

```

type c struct {
    A a
    B b
}

```

So, composition allows you to create a structure in your Go elements using multiple `struct` types. In this case, data type `c` groups an `a` variable and a `b` variable.

The third portion of `ooo.go` comes with the next Go code:

```

func (A a) A() {
    fmt.Println("Function A() for A")
}

func (B b) A() {
    fmt.Println("Function A() for B")
}

```

The two methods defined here can have the same name (`A()`) because they have different function headers – the first one works with `a` variables, whereas the second one works with `b` variables. This technique allows you to share the same function name between multiple types.

The last part of `ooo.go` is next:

```

func main() {
    var i c
    i.A.A()
    i.B.A()
}

```

All the Go code in `ooo.go` is pretty simplistic compared to the code of an object-oriented programming language that would implement abstract

classes and inheritance. However, it is more than adequate for generating types and elements with a structure in them, as well as for having different data types with the same method names.

Executing `ooo.go` will generate the following output:

```
$ go run ooo.go
Function A() for A
Function A() for B
```

Nevertheless, as the following code illustrates, composition is not inheritance and the `first` type knows nothing about the changes made to the `shared()` function by the `second` type:

```
package main

import (
    "fmt"
)

type first struct{}

func (a first) F() {
    a.shared()
}

func (a first) shared() {
    fmt.Println("This is shared() from first!")
}

type second struct {
    first
}

func (a second) shared() {
    fmt.Println("This is shared() from second!")
}

func main() {
    first{}.F()
    second{}.shared()
    i := second{}
    j := i.first
    j.F()
}
```

Please note that the `second` type embeds the `first` type and that the two types share a function named `shared()`.

Saving the former Go code as `goCoIn.go` and executing it will generate the following output:

```
$ go run goCoIn.go
This is shared() from first!
This is shared() from second!
This is shared() from first!
```

Although the calls to `first{}.F()` and `second{}.shared()` generate the expected results, the call to `j.F()` still calls `first.shared()` instead of the `second.shared()` function despite the fact that the `second` type changes the implementation of the `shared()` function. This is called **method overriding** in object-oriented terminology.

Note that the `j.F()` call can be written as `(i.first).F()` or as `(second{}.first).F()` without the need to define too many variables. Breaking it into three lines of code makes it a little easier to understand.

An introduction to git and GitHub

GitHub is a website and service for storing and building software. You can work on GitHub using its graphical user interface or using command-line utilities. On the other hand, `git(1)` is a command-line utility that can do many things, including working with GitHub repositories.



*An alternative to GitHub is **GitLab**. Most, if not all, of the presented `git(1)` commands and options will work for communicating with GitLab without any modifications.*

This section will offer you a quick introduction to `git(1)` and its most common and frequently used commands.

Using git

Note that `git(1)` has a huge number of commands and options that you do not need to use on a daily basis. In this subsection, I am going to present you with the most useful and popular `git(1)` commands in my experience and based on the way I work.

Please note that in order to get an existing GitHub repository onto your local computer, you will need to use the `git clone` command followed by the URL of the repository:

```
$ git clone git@github.com:mactsouk/go-kafka.git
Cloning into 'go-kafka'...
remote: Enumerating objects: 13, done.
remote: Counting objects: 100% (13/13), done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 13 (delta 4), reused 10 (delta 4), pack-reused 0
Receiving objects: 100% (13/13), done.
Resolving deltas: 100% (4/4), done.
```

The git status command

The `git status` command shows the status of the working tree. If everything is in sync, `git status` will return an output similar to the following:

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

If there are changes, the output of `git status` will look similar to the following:

```
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   main.go

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    newFile.go

no changes added to commit (use "git add" and/or "git commit -a")
```

The git pull command

The `git pull` command is used to get updates from the remote repository. This is especially useful when multiple people are working on the same repository or if you are working from multiple machines.

The git commit command

The `git commit` command is for recording changes to the repository. After a `git commit` command, you will most likely need to issue a `git push` command to send the changes to the remote repository. A very common way to execute the `git commit` command is the following:

```
| $ git commit -a -m "Commit message"
```

The `-m` option specifies that the message that will go with the commit, whereas the `-a` option tells `git commit` to automatically include all modified files. Please note that this will exclude new files that need to be added first using `git add`.

The git push command

For local changes to be transferred to the GitHub repository, you will need to execute the `git push` command. The output of the `git push` command is similar to the following:

```
$ touch a_file.go
$ git add a_file.go
$ git commit -a -m "Adding a new file"
[master 782c4da] Adding a new file
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 ch07/a_file.go
$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 337 bytes | 337.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To github.com:PacktPublishing/Mastering-Go-Second-Edition.git
 98f8a77..782c4da master -> master
```

Working with branches

A branch offers a way to manage your workflow and separate changes from the main branch. Each repository has a default branch, which is usually called the `master` branch, and potentially multiple other branches.

You can create a new branch named `new_branch` on your local machine and go to it as follows:

```
$ git checkout -b new_branch
Switched to a new branch 'new_branch'
```

If you want to connect that branch with GitHub, you should execute the following command:

```
$ git push --set-upstream origin new_branch
Total 0 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'new_branch' on GitHub by visiting:
remote:     https://github.com/PacktPublishing/Mastering-Go-Second-Edition/pull/new/new_branch
remote:
To github.com:PacktPublishing/Mastering-Go-Second-Edition.git
 * [new branch]      new_branch -> new_branch
Branch 'new_branch' set up to track remote branch 'new_branch' from 'origin'.
```

If you want to change your current branch and go back to the `master` branch, you can execute the following command:

```
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
```

If you want to delete a local branch, `new_branch` in this case, you can execute the `git branch -D` command:

```
$ git --no-pager branch -a
* master
new_branch
remotes/origin/HEAD -> origin/master
remotes/origin/master
remotes/origin/new_branch
$ git branch -D new_branch
Deleted branch new_branch (was 98f8a77).
$ git --no-pager branch -a
* master
remotes/origin/HEAD -> origin/master
remotes/origin/master
remotes/origin/new_branch
```

Working with files

When you add or remove one or more files from a repository, `git(1)` should know about it from you. You can delete a file named `a_file.go` as follows:

```
$ rm a_file.go  
$ git rm a_file.go  
rm 'ch07/a_file.go'
```

Executing `git status` at this point will generate the following kind of output:

```
$ git status  
On branch master  
Your branch is up to date with 'origin/master'.  
  
Changes to be committed:  
  (use "git reset HEAD <file>..." to unstage)  
  
        deleted:    a_file.go
```

For changes to take effect you will need to `git commit` first and `git push` afterwards:

```
$ git commit -a -m "Deleting a_file.go"  
[master 1b06700] Deleting a_file.go  
1 file changed, 0 insertions(+), 0 deletions(-)  
delete mode 100644 ch07/a_file.go  
$ git push  
Enumerating objects: 5, done.  
Counting objects: 100% (5/5), done.  
Delta compression using up to 8 threads  
Compressing objects: 100% (3/3), done.  
Writing objects: 100% (3/3), 296 bytes | 296.00 KiB/s, done.  
Total 3 (delta 2), reused 0 (delta 0)  
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.  
To github.com:PacktPublishing/Mastering-Go-Second-Edition.git  
  782c4da..1b06700  master -> master
```

The `.gitignore` file

The `.gitignore` file is used to list files or directories that you want to ignore when committing to GitHub. The contents of a sample `.gitignore` file might look as follows:

```
$ cat .gitignore
public/
.DS_Store
*.swp
```

Please note that after being created for the first time, `.gitignore` should be added to the current branch using `git add`.

Using git diff

The `git diff` command shows differences between commits and a working repository or branch, and so on.

The following command will show the changes between your files and the files that are on GitHub (before the last `git push`). These will be the changes that will be added to the version that is on GitHub after a `git push` command:

```
$ git diff
diff --git a/content/blog/Stats.md b/content/blog/Stats.md
index 0f36b60..af64ec3 100644
--- a/content/blog/Stats.md
+++ b/content/blog/Stats.md
@@ -16,6 +16,8 @@ title: Statistical analysis of random numbers

## Developing a Kafka producer in Go

+Please note that the format of the first record that is written to Kafka
+specifies the format of the subsequent records

### Viewing the data in Lenses
```

Working with tags

A **tag** is a way of identifying specific release versions of your code. You can think of a tag as a branch that never changes.

You can create a new **lightweight tag** as follows:

```
$ git tag c7.0
```

You can find information about a specific tag as follows:

```
$ git --no-pager show v1.0.0
commit f415872e62bd71a004b680d50fa089c139359533 (tag: v1.0.0)
Author: Mihalis Tsoukalos <mihalitsoukalos@gmail.com>
Date:   Sat Mar 2 20:33:58 2019 +0200

    Initial version 1.0.0

diff --git a/go.mod b/go.mod
new file mode 100644
index 0000000..c4928c5
--- /dev/null
+++ b/go.mod
@@ -0,0 +1,3 @@
+module github.com/mactsouk/myModule
+
+go 1.12
diff --git a/myModule.go b/myModule.go
index e69de29..fa6b0fe 100644
--- a/myModule.go
+++ b/myModule.go
@@ -0,0 +1,9 @@
+package myModule
+
+import (
+    "fmt"
+)
+
+func Version() {
+    fmt.Println("Version 1.0.0")
+}
```

You can list all available tags using the `git tag` command:

```
$ git --no-pager tag
c7.0
```

You can push a tag to GitHub as follows:

```
$ git push origin c7.0
Total 0 (delta 0), reused 0 (delta 0)
To github.com:PacktPublishing/Mastering-Go-Second-Edition.git
 * [new tag]           c7.0 -> c7.0
```

You can delete an existing tag from localhost as follows:

```
$ git tag -d c7.0
Deleted tag 'c7.0' (was 1b06700)
```

You can delete an existing tag from remote, which is the GitHub server, as follows:

```
$ git push origin :refs/tags/c7.0
To github.com:PacktPublishing/Mastering-Go-Second-Edition.git
 - [deleted]           c7.0
```

The git cherry-pick command

The `git cherry-pick` command is an advanced command that should be used with care as it applies the changes introduced by some existing commits to the current branch. The following command will apply commit `4226f2c4` to the current branch:

```
| $ git cherry-pick 4226f2c4
```

The following command will apply all commits from `4226f2c4` to `0d820a87` to the current branch without including the `4226f2c4` commit:

```
| $ git cherry-pick 4226f2c4..0d820a87
```

The following command will apply all commits from `4226f2c4` to `0d820a87` to the current branch, including the `4226f2c4` commit:

```
| $ git cherry-pick 4226f2c4^..0d820a87
```



Although the presented list of `git(1)` commands and options is far from complete, they will allow you to work with `git(1)` and GitHub, and will come in handy when creating Go modules.

Debugging with Delve

Delve is a text-based **debugger** for Go programs written in Go. On macOS Mojave, you can download Delve as follows:

```
$ go get -u github.com/go-delve/delve/cmd/dlv
$ ls -l ~/go/bin/dlv
-rwxr-xr-x 1 mtsouk  staff   16M Mar  7 09:04 /Users/mtsouk/go/bin/dlv
```

As Delve depends on a lot of Go modules and packages, the installation process is going to take a while. The Delve binary is installed on `~/go/bin`. Executing `dlv version` will reveal information about its version:

```
$ ~/go/bin/dlv version
Delve Debugger
Version: 1.2.0
Build: $Id: 068e2451004e95d0b042e5257e34f0f08ce01466 $
```

Note that Delve also works on Linux and Microsoft Windows machines. Also note that Delve is an external program, which means that you do not need to include any packages in your Go programs for Delve to work.

As this section is a quick introduction to Delve, the following subsection will present a small example to help you to get started with the Delve debugger. The presented information shows the general ideas behind Delve and almost every other debugger.

A debugging example

If `~/go/bin` is in your `PATH` environment variable, then you can call Delve from everywhere as `dlv`. Otherwise, you will need to provide its full path. I am going to use the full path in this subsection.

The first Delve command that you should know is `debug`. This command will tell Delve to compile the `main` package in the current working directory and begin to debug it. If there is no `main` package in the current working directory, you will get the following error message:

```
$ ~/go/bin/dlv debug
go: cannot find main module; see 'go help modules'
exit status 1
```

So, let us go to the `./ch07/debug` directory and debug a real program that is stored in `main.go`. The Go code of `main.go` is as follows:

```
package main

import (
    "fmt"
    "os"
)

func function(i int) int {
    return i * i
}

func main() {
    if len(os.Args) == 1 {
        fmt.Println("Need at least one argument.")
        return
    }

    i := 5
    fmt.Println("Debugging with Delve")
    fmt.Println(i)
    fmt.Println(function(i))

    for arg, _ := range os.Args[1:] {
        fmt.Println(arg)
    }
}
```

In order to pass some command-line arguments to the program, you should execute Delve as follows:

```
| $ ~/go/bin/dlv debug -- arg1 arg2 arg3
```

We are going to execute Delve as follows:

```
| $ ~/go/bin/dlv debug -- 1 2 3
| Type 'help' for list of commands.
| (dlv)
```

The prompt of Delve is `(dlv)`, which is where you give your Delve commands. If you just press `c` (for `continue`) at this point, your Go program will be executed as if you were issuing the `go run` command from your shell:

```
| (dlv) c
| Debugging with Delve
| 5
| 25
| 0
| 1
| 2
| Process 57252 has exited with status 0
| (dlv)
```

If you press `c` or type `continue` again, you will get the following:

```
| (dlv) c
| Process 57252 has exited with status 0
```

This happens because your program has ended and cannot continue at this point – you will have to restart it in order to be able to debug it. You can restart a program by typing `r` or `restart`, which is equal to executing Delve from the UNIX shell:

```
| (dlv) r
| Process restarted with PID 57257
| (dlv)
```

At this point, we will have to try something different. What we are going to do is set two breakpoints, one for the `main()` function and another one for `function()`:

```
$ ~/go/bin/dlv debug -- 1 2 3
Type 'help' for list of commands.
(dlv)
(dlv) break main.main
Breakpoint 1 set at 0x10b501b for main.main() ./main.go:12
(dlv) break function
Breakpoint 2 set at 0x10b4fe0 for main.function() ./main.go:8
(dlv)
```

If you press continue at this point, the debugger will stop the program at either `main()` or `function()`, whatever comes first. As this is an executable program, the `main()` function will be first:

```
(dlv) c
> main.main() ./main.go:12 (hits goroutine(1):1 total:1) (PC: 0x10b501b)
    7:
    8:     func function(i int) int {
    9:         return i * i
   10:    }
   11:
=> 12:    func main() {
   13:        if len(os.Args) == 1 {
   14:            fmt.Println("Need at least one argument.")
   15:            return
   16:        }
   17:
(dlv)
```

The `=>` arrow shows the line of the source code that the break happened in. Typing `next` will take us to the next Go statement – we can type `next` as many times as we want until we reach the end of the program:

```
(dlv) next
> main.main() ./main.go:13 (PC: 0x10b5032)
    8:     func function(i int) int {
    9:         return i * i
   10:    }
   11:
   12:    func main() {
=> 13:        if len(os.Args) == 1 {
   14:            fmt.Println("Need at least one argument.")
   15:            return
   16:        }
   17:
   18:        i := 5
(dlv) next
> main.main() ./main.go:18 (PC: 0x10b50d0)
    13:        if len(os.Args) == 1 {
    14:            fmt.Println("Need at least one argument.")
    15:            return
    16:        }
    17:
=> 18:        i := 5
```

```

19:         fmt.Println("Debugging with Delve")
20:         fmt.Println(i)
21:         fmt.Println(function(i))
22:
23:         for arg, _ := range os.Args[1:] {
(dlv) next
> main.main() ./main.go:19 (PC: 0x10b50db)
14:             fmt.Println("Need at least one argument.")
15:             return
16:         }
17:
18:         i := 5
=> 19:         fmt.Println("Debugging with Delve")
20:         fmt.Println(i)
21:         fmt.Println(function(i))
22:
23:         for arg, _ := range os.Args[1:] {
24:             fmt.Println(arg)
(dlv) print i
5

```

The last command (`print i`) prints the value of variable `i`. Typing `continue` will take us to the next break point, if there is one, or to the end of the program:

```

(dlv) c
Debugging with Delve
5
> main.function() ./main.go:8 (hits goroutine(1):1 total:1) (PC: 0x10b4fe0)
3:     import (
4:         "fmt"
5:         "os"
6:     )
7:
=> 8:     func function(i int) int {
9:         return i * i
10:    }
11:
12:     func main() {
13:         if len(os.Args) == 1 {

```

In this case, the next break point is the `function()` function, as defined earlier.

Please note that for debugging Go tests, you should use the `dlv test` command and Delve will take care of the rest.

Additional resources

You will find the next resources very handy:

- Visit the documentation page of the `reflect` Go standard package, which can be found at <https://golang.org/pkg/reflect/>. This package has many more capabilities than the ones presented in this chapter.
- GitHub: <https://github.com/>.
- GitLab: <https://gitlab.com/>.
- You can find out more about Delve at <https://github.com/go-delve/delve>.
- You can find out more about the `reflectwalk` library by Mitchell Hashimoto at <https://github.com/mitchellh/reflectwalk>. Studying its code will help you to learn more about reflection.

Exercises

- Write your own interface and use it in another Go program. Then state why your interface is useful.
- Write an interface for calculating the volume of shapes with three dimensions, such as cubes and spheres.
- Write an interface for calculating the length of line segments and the distance between two points in the plane.
- Explore reflection using your own example.
- How does reflection work on Go maps?
- If you are good at mathematics, try to write an interface that implements the four basic mathematical operations for both real numbers and complex numbers. Do not use the `complex64` and `complex128` standard Go types – define your own structure for supporting complex numbers.

Summary

In this chapter, you learned about debugging, `git(1)`, GitHub, and interfaces, which are like contracts, and also about type methods, type assertion, and reflection in Go. Although reflection is a very powerful Go feature, it might slow down your Go programs because it adds a layer of complexity at runtime. Furthermore, your Go programs could crash if you use reflection carelessly.

You additionally learned about creating Go code that follows the principles of object-oriented programming. If you are going to remember just one thing from this chapter, it should be that Go is not an object-oriented programming language, but it can mimic some of the functionality offered by object-programming languages, such as Java and C++. This means that if you plan to develop software using the object-oriented paradigm all of the time, it would be best to choose a programming language other than Go. Nevertheless, object-oriented programming is not a panacea, and you might create a better, cleaner, and more robust design if you choose a programming language such as Go!

Although there may have been more theory in this chapter than you expected, the next chapter will reward your patience; it will address systems programming in Go. File I/O, working with UNIX system files, handling UNIX signals, and supporting UNIX pipes will be discussed.

The next chapter will also talk about using the `flag` and `viper` packages to support multiple command-line arguments and options in your command-line tools, as well as the `cobra` package, UNIX file permissions, and some advanced uses of the functionality offered by the `syscall` standard Go package. If you are really into systems programming with Go, my book on the subject elaborates on this (*Go Systems Programming* (Packt Publishing, 2017)).

Telling a UNIX System What to Do

In the previous chapter, we talked about two advanced, but somewhat theoretical, Go topics: interfaces and reflection. The Go code that you will find in this chapter is anything but theoretical!

The subject of this chapter is **systems programming** because, after all, Go is a mature systems programming language that was born out of frustration. Its spiritual fathers were unsatisfied with the programming language choices they had for creating systems software, so they decided to create a new programming language.



This chapter contains some interesting and somewhat advanced topics that are not included in Go Systems Programming (Packt Publishing, 2017).

This chapter focuses on the following topics:

- UNIX processes
- The `flag` package
- The `viper` package
- The `cobra` package
- The use of the `io.Reader` and `io.Writer` interfaces
- Handling UNIX **signals** in Go with the help of the `os/signal` package
- Supporting UNIX **pipes** in your UNIX system utilities
- Creating Go clients for Docker
- Reading text files
- Reading CSV files
- Writing to files
- The `bytes` package
- Advanced uses of the `syscall` package
- UNIX file permissions

About UNIX processes

Strictly speaking, a **process** is an execution environment that contains instructions, user data and system data parts, and other types of resources that are obtained during runtime. On the other hand, a **program** is a binary file that contains instructions and data that are used for initializing the instruction and user data parts of a process. Each running UNIX process is uniquely identified by an unsigned integer, which is called the **process ID** of the process.

There are three categories of processes: **user processes**, **daemon processes**, and **kernel processes**. User processes run in user space and usually have no special access rights. Daemon processes are programs that can be found in the user space and run in the background without the need for a terminal. Kernel processes are executed in kernel space only and can fully access all kernel data structures.



The C way of creating new processes involves calling the `fork()` system call. The return value of `fork()` allows the programmer to differentiate between the parent and the child process. In contrast, Go does not support a similar functionality but offers goroutines.

The flag package

Flags are specially-formatted strings that are passed into a program to control its behavior. Dealing with flags on your own might become very difficult if you want to support multiple flags. Thus, if you are developing UNIX system command-line utilities, you will find the `flag` package very interesting and useful.

The `flag` package makes no assumptions about the order of command-line arguments and options, and it prints helpful messages in case there was an error in the way the command-line utility was executed.



The biggest advantage of the `flag` package is that it is part of the standard Go library, which means that it has been extensively tested and debugged.

I will present two Go programs that use the `flag` package: a simple one and a more advanced one. The first one, named `simpleFlag.go`, will be offered in four parts. The `simpleFlag.go` program will recognize two command-line options: the first one will be a Boolean option and the second one will require an integer value.

The first part of `simpleFlag.go` contains the following Go code:

```
package main  
  
import (  
    "flag"  
    "fmt"  
)
```

The second code portion from `simpleFlag.go` is as follows:

```
func main() {  
    minusK := flag.Bool("k", true, "k flag")  
    minusO := flag.Int("O", 1, "O")  
    flag.Parse()
```

The `flag.Bool("k", true, "k flag")` statement defines a Boolean command-line option named `k` with the default value of `true`. The last parameter of the statement is the usage string that will be displayed with the usage information of the program. Similarly, the `flag.Int()` function adds support for an integer command-line option.



You always need to call `flag.Parse()` after defining the command-line options that you want to support.

The third part of the `simpleFlag.go` program contains the following Go code:

```
| valueK := *minusK
| valueO := *minusO
| valueO++
| }
```

In the preceding Go code, you can see how you can obtain the values of your options. The good thing here is that the `flag` package automatically converts the input associated with the `flag.Int()` flag to an integer value. This means that you do not have to do that on your own. Additionally, the `flag` package makes sure that it was given an acceptable integer value.

The remaining Go code from `simpleFlag.go` follows:

```
|     fmt.Println("-k:", valueK)
|     fmt.Println("-O:", valueO)
| }
```

After getting the values of the desired parameters, you are now ready to use them.

Interacting with `simpleFlag.go` will create the following type of output:

```
$ go run simpleFlag.go -O 100
-k: true
-O: 101
$ go run simpleFlag.go -O=100
-k: true
-O: 101
$ go run simpleFlag.go -O=100 -k
-k: true
-O: 101
$ go run simpleFlag.go -O=100 -k=false
-k: true
-O: 101
$ go run simpleFlag.go -O=100 -k=false
-k: false
-O: 101
```

If there is an error in the way `simpleFlag.go` was executed, you will get the following type of error message from the `flag` package:

```
$ go run simpleFlag.go -O=notAnInteger
invalid value "notAnInteger" for flag -O: parse error
Usage of /var/folders/sk/1tk8cnw501zdtr2hxcj5sv2m0000gn/T/go-build593534621/b001/exe/simpleFlag:
  -O int
    O (default 1)
  -k    flag (default true)
exit status 2
```

Notice the convenient usage message that is automatically printed when there is an error in the command-line options given to your program.

Now it is time to present a more realistic and advanced program that uses the `flag` package. Its name is `funWithFlag.go`, and it will be presented in five parts. The `funWithFlag.go` utility will recognize various kinds of options, including one that accepts

multiple values that are separated by commas. Additionally, it will illustrate how you can access the command-line arguments that are located at the end of the executable and do not belong to any option.

The `flag.Var()` function used in `funWithFlag.go` creates a flag of any type that satisfies the `flag.Value` interface, which is defined as follows:

```
type Value interface {
    String() string
    Set(string) error
}
```

The first part of `funWithFlag.go` contains the following Go code:

```
package main

import (
    "flag"
    "fmt"
    "strings"
)

type NamesFlag struct {
    Names []string
}
```

The `NamesFlag` structure will be used in a short while for the `flag.Value` interface.

The second part of `funWithFlag.go` is as follows:

```
func (s *NamesFlag) GetNames() []string {
    return s.Names
}

func (s *NamesFlag) String() string {
    return fmt.Sprint(s.Names)
}
```

The third code portion from `funWithFlag.go` contains the following code:

```
func (s *NamesFlag) Set(v string) error {
    if len(s.Names) > 0 {
        return fmt.Errorf("Cannot use names flag more than once!")
    }

    names := strings.Split(v, ",")
    for _, item := range names {
        s.Names = append(s.Names, item)
    }
    return nil
}
```

First, the `Set()` method makes sure that the related command-line option is not already set. After that, it gets the input and separates its arguments using the `strings.Split()` function. Then, the arguments are saved in the `Names` field of the `NamesFlag` structure.

The fourth code segment from `funWithFlag.go` is shown in the following Go code:

```

func main() {
    var manyNames NamesFlag
    minusK := flag.Int("k", 0, "An int")
    minusO := flag.String("o", "Mihalis", "The name")
    flag.Var(&manyNames, "names", "Comma-separated list")

    flag.Parse()
    fmt.Println("-k:", *minusK)
    fmt.Println("-o:", *minusO)
}

```

The last part of the `funWithFlag.go` utility follows:

```

for i, item := range manyNames.GetNames() {
    fmt.Println(i, item)
}

fmt.Println("Remaining command line arguments:")
for index, val := range flag.Args() {
    fmt.Println(index, ":", val)
}
}

```

The `flag.Args()` slice keeps the command-line arguments that are left, while the `manyNames` variable holds the values from the `flag.Var()` command-line option.

Executing `funWithFlag.go` will create the following type of output:

```

$ go run funWithFlag.go -names=Mihalis,Jim,Athina 1 two Three
-k: 0
-o: Mihalis
0 Mihalis
1 Jim
2 Athina
Remaining command line arguments:
0 : 1
1 : two
2 : Three
$ go run funWithFlag.go -Invalid=Marietta 1 two Three
flag provided but not defined: -Invalid
Usage of funWithFlag:
-k int
        An int
-names value
        Comma-separated list
-o string
        The name (default "Mihalis")
exit status 2
$ go run funWithFlag.go -names=Marietta -names=Mihalis
invalid value "Mihalis" for flag -names: Cannot use names flag more than once!
Usage of funWithFlag:
-k int
        An int
-names value
        Comma-separated list
-o string
        The name (default "Mihalis")
exit status 2

```



Unless you are developing a trivial command-line utility that requires no command-line options, you will most likely need to use a Go package to process the command-line arguments of your program.

The viper package

`viper` is a powerful Go package that supports a plethora of options. All `viper` projects follow a pattern. First, you initialize `viper` and then you define the elements that interest you. After that, you get these elements and read their values in order to use them. Notice that the `viper` package can entirely replace the `flag` package.

The desired values can be either taken directly, as happens when you are using the `flag` package of the standard Go library, or indirectly using configuration files. When using formatted configuration files in the JSON, YAML, TOML, HCL, or Java properties format, `viper` does all the parsing for you, which saves you from having to write and debug lots of Go code. `viper` also allows you to extract and save values in Go structures. However, this also requires the fields of the Go structure to match the keys of the configuration file.

The home page of `viper` is on GitHub (<https://github.com/spf13/viper>). Please note that you are not obliged to use every capability of `viper` in your tools – just the features that you want. The general rule is to use the features of `viper` that simplify your code. Put simply, if your command-line utility requires too many command-line parameters and flags, then it would be better to use a configuration file instead.

A simple viper example

Before going into more advanced examples, I will present some sample Go code that uses `viper`. The name of the program is `usingViper.go` and it will be presented in three parts.

The first part of `useViper.go` is as follows:

```
package main

import (
    "fmt"
    "github.com/spf13/viper"
)
```

The second part of `useViper.go` is as follows:

```
func main() {
    viper.BindEnv("GOMAXPROCS")
    val := viper.Get("GOMAXPROCS")
    fmt.Println("GOMAXPROCS:", val)
    viper.Set("GOMAXPROCS", 10)
    val = viper.Get("GOMAXPROCS")
    fmt.Println("GOMAXPROCS:", val)
```

The last part of `useViper.go` contains the following Go code:

```
viper.BindEnv("NEW_VARIABLE")
val = viper.Get("NEW_VARIABLE")
if val == nil {
    fmt.Println("NEW_VARIABLE not defined.")
    return
}
fmt.Println(val)
```

The purpose of this program is to illustrate how you can read and modify environment variables using `viper`. The `flag` package does not offer such functionality but the `os` standard Go package does, although not as easily as the `viper` package.

The first time you are going to use `viper`, you will need to download it. If you are not using Go modules, this can be done as follows:

```
| $ go get -u github.com/spf13/viper
```

If you are using Go modules, Go will automatically download `viper` the first time you try to execute a Go program that uses it.

Executing `useViper.go` will generate the following kind of output:

```
| $ go run useViper.go
| GOMAXPROCS: <nil>
| GOMAXPROCS: 10
| NEW_VARIABLE not defined.
```

From flag to viper

There is a chance that you have a Go program that already uses the `flag` package and that you want to convert to using the `viper` package. Let us say that we have the following Go code that uses the `flag` package:

```
package main

import (
    "flag"
    "fmt"
)

func main() {
    minusI := flag.Int("i", 100, "i parameter")
    flag.Parse()
    i := *minusI
    fmt.Println(i)
}
```

The new version of that program, which is going to be saved in `flagToViper.go`, is going to use the `viper` package and will be presented in three parts. The first part of `flagToViper.go` is as follows:

```
package main

import (
    "flag"
    "fmt"
    "github.com/spf13/pflag"
    "github.com/spf13/viper"
)
```

You will need to import the `pflag` package in order to work with command-line arguments in `viper`.

The second part of `flagToViper.go` contains the following Go code:

```
func main() {
    flag.Int("i", 100, "i parameter")
    pflag.CommandLine.AddGoFlagSet(flag.CommandLine)
    pflag.Parse()
```

So, you still use `flag.Int()` in order to change as little code as possible, but for the parsing, you call `pflag.Parse()`. However, all the magic happens with

the call to `pflag.CommandLine.AddGoFlagSet(flag.CommandLine)` because this call imports the data from the `flag` package to the `pflag` package.

The last part of `flagToViper.go` is the following:

```
viper.BindPFlags(pflag.CommandLine)
i := viper.GetInt("i")
fmt.Println(i)
}
```

The last function call that you have to make is `viper.BindPFlags()`. After that, you can get the value of an integer command-line parameter using `viper.GetInt()`. For other kinds of data types, you will have to call different `viper` functions.

At this point, you might need to download the `pflag` Go package for `flagToViper.go` to work, which can be done as follows:

```
$ go get -u github.com/spf13/pflag
```

Executing `flagToViper.go` will generate the following kind of output:

```
$ go run flagToViper.go
100
$ go build flagToViper.go
$ ./flagToViper -i 0
0
$ ./flagToViper -i abcd
invalid argument "abcd" for "-i, --i" flag: parse error
Usage of ./flagToViper:
  -i, --i int  i parameter
invalid argument "abcd" for "-i, --i" flag: parse error
```

If, for some reason, you give an unknown command-line parameter to `flagToViper.go`, `viper` will complain about it:

```
$ ./flagToViper -j 200
unknown shorthand flag: 'j' in -j
Usage of ./flagToViper:
  -i, --i int  i parameter (default 100)
unknown shorthand flag: 'j' in -j
exit status 2
```

Reading JSON configuration files

In this subsection, you will learn how to read JSON configuration files with the `viper` package. The name of the utility is `readJSON.go` and it is going to be presented in three parts. The first part of `readJSON.go` is as follows:

```
package main

import (
    "fmt"
    "github.com/spf13/viper"
)
```

The second part of `readJSON.go` contains the following code:

```
func main() {
    viper.SetConfigType("json")
    viper.SetConfigFile("./myJSONConfig.json")
    fmt.Printf("Using config: %s\n", viper.ConfigFileUsed())
    viper.ReadInConfig()
```

This is where the parsing of the configuration file takes place. Please note that the filename of the JSON configuration file is hardcoded inside `readJSON.go` using the `viper.SetConfigFile("./myJSONConfig.json")` function call.

The last part of `readJSON.go` is as follows:

```
if viper.IsSet("item1.key1") {
    fmt.Println("item1.key1:", viper.Get("item1.key1"))
} else {
    fmt.Println("item1.key1 not set!")
}

if viper.IsSet("item2.key3") {
    fmt.Println("item2.key3:", viper.Get("item2.key3"))
} else {
    fmt.Println("item2.key3 is not set!")
}

if !viper.IsSet("item3.key1") {
    fmt.Println("item3.key1 is not set!")
}
```

This is where the values of the JSON configuration file are examined by the program in order to find out whether the desired keys exist or not.

The contents of the `myJSONConfig.json` file are as follows:

```
{  
    "item1": {  
        "key1": "val1",  
        "key2": false,  
        "key3": "val3"  
    },  
    "item2": {  
        "key1": "val1",  
        "key2": true,  
        "key3": "val3"  
    }  
}
```

Executing `readJSON.go` will generate the following kind of output:

```
$ go run readJSON.go  
Using config: ./myJSONConfig.json  
item1.key1: val1  
item2.key3: val3  
item3.key1 is not set!
```

If `myJSONConfig.json` cannot be located, the program will not complain and will act as if it has read an empty JSON configuration file:

```
$ mv myJSONConfig.json ..  
$ go run readJSON.go  
Using config: ./myJSONConfig.json  
item1.key1 not set!  
item2.key3 is not set!  
item3.key1 is not set!
```

Reading YAML configuration files

YAML is another popular text-based format that is used for configuration files. In this subsection, you will learn how to read YAML configuration files with the `viper` package.

However, this time, the filename of the YAML configuration file will be given as a command-line argument to the utility. Additionally, the utility will use the `viper.AddConfigPath()` function to add three search paths, which are places where `viper` will automatically look for configuration files. The name of the utility, which will be presented in four parts, is `readYAML.go`.

The first part of `readYAML.go` is as follows:

```
package main

import (
    "fmt"
    "flag"
    "github.com/spf13/pflag"
    "github.com/spf13/viper"
    "os"
)
```

In the preamble of the program, we define an alias (`flag`) for the `pflag` Go package.

The second part of `readYAML.go` contains the following Go code:

```
func main() {
    var configFile *string = flag.String("c", "myConfig", "Setting the configuration file")
    flag.Parse()

    _, err := os.Stat(*configFile)

    if err == nil {
        fmt.Println("Using User Specified Configuration file!")
        viper.SetConfigFile(*configFile)
    } else {
        viper.SetConfigName(*configFile)
        viper.AddConfigPath("/tmp")
        viper.AddConfigPath("$HOME")
        viper.AddConfigPath(".")
    }
}
```

The code checks whether the value of the config flag (`--c`) exists using a call to `os.Stat()`. If it exists, the provided file will be used; otherwise, the default config filename (`myConfig`) will be used. Notice that we are not explicitly specifying that we want to use a YAML configuration file - the program will look for all supported file formats, provided that the filename without the file extension is `myConfig`, because this

is the way `viper` works. Three paths will be searched for configuration files: `/tmp`, the home directory of the current user, and the current working directory, in that order.

Using the `/tmp` directory to keep your configuration files is not recommended, mainly because the contents of `/tmp` are automatically deleted after each system reboot - it is only used here for reasons of simplicity.

The use of `viper.ConfigFileUsed()` makes perfect sense because there is no hardcoded configuration file, which means that we will have to define it on our own.

The third part of `readyYAML.go` is as follows:

```
err = viper.ReadInConfig()
if err != nil {
    fmt.Printf("%v\n", err)
    return
}
fmt.Printf("Using config: %s\n", viper.ConfigFileUsed())
```

The YAML file is read and parsed using a call to `viper.ReadInConfig()`.

The last part of `readyYAML.go` is as follows:

```
if viper.IsSet("item1.k1") {
    fmt.Println("item1.val1:", viper.Get("item1.k1"))
} else {
    fmt.Println("item1.k1 not set!")
}

if viper.IsSet("item1.k2") {
    fmt.Println("item1.val2:", viper.Get("item1.k2"))
} else {
    fmt.Println("item1.k2 not set!")
}

if !viper.IsSet("item3.k1") {
    fmt.Println("item3.k1 is not set!")
}
```

This is where the contents of the parsed configuration file are examined by the program in order to find out whether the desired keys exist or not.

The contents of `myConfig.yaml` are as follows:

```
item1:
  k1:
    - true
  k2:
    - myValue
```

Executing `readyYAML.go` will generate the following kind of output:

```
$ go build readyYAML.go
$ ./readyYAML
```

```
|Using config: /Users/mtsouk/Desktop/mGo2nd/Mastering-Go-Second-Edition/ch08/viper/myConfig.yaml
item1.val1: [true]
item1.val2: [myValue]
item3.k1 is not set!
$ mv myConfig.yaml /tmp
$ ./readYAML
Using config: /tmp/myConfig.yaml
item1.val1: [true]
item1.val2: [myValue]
item3.k1 is not set!
```

The cobra package

`cobra` is a very handy and popular Go package that allows you to develop command-line utilities with commands, subcommands, and aliases. If you have ever used `hugo`, `docker`, or `kubectl` you will understand immediately what Cobra does, as all these tools were developed using `cobra`.

As you will see in this section, commands in `cobra` can have one or more aliases, which is very handy when you want to please both amateur and experienced users. `cobra` also supports Persistent Flags and Local Flags, which are flags that are available to all commands and flags that are available to a given command only, respectively. Also, by default, `cobra` uses `viper` for parsing its command-line arguments.

All `cobra` projects follow the same development pattern. You use the `cobra` tool, then you create commands, and then you make the desired changes to the generated Go source code files in order to implement the desired functionality. Depending on the complexity of your utility, you might need to make lots of changes to the created files. Although `cobra` saves you lots of time, you will still have to write the code that implements the desired functionality.

If you want to install your utility in binary version, you can always execute `go install` from anywhere in the `cobra` project directory. Unless otherwise specified, the binary executable will be placed in `~/go/bin`.

The home page of `cobra` is on GitHub (<https://github.com/spf13/cobra>).

A simple cobra example

In this section, we are going to implement a simple command-line utility using `cobra` and the `~/go/bin/cobra` tool that comes with the package. If you execute `~/go/bin/cobra` without any command-line arguments, you will get the following output:

```
$ ~/go/bin/cobra
Cobra is a CLI library for Go that empowers applications.
This application is a tool to generate the needed files
to quickly create a Cobra application.

Usage:
  cobra [command]

Available Commands:
  add      Add a command to a Cobra Application
  help     Help about any command
  init     Initialize a Cobra Application

Flags:
  -a, --author string   author name for copyright attribution (default "YOUR NAME")
  --config string        config file (default is $HOME/.cobra.yaml)
  -h, --help              help for cobra
  -l, --license string   name of license for the project
  --viper                use Viper for configuration (default true)

Use "cobra [command] --help" for more information about a command.
```

With that information in mind, we are going to create a new `cobra` project, as follows:

```
$ ~/go/bin/cobra init cli
Your Cobra application is ready at
/Users/mtsouk/go/src/cli

Give it a try by going there and running `go run main.go`.
Add commands to it by running `cobra add [cmdname]`.
$ cd ~/go/src/cli
$ ls -l
total 32
-rw-r--r--  1 mtsouk  staff  11358 Mar 13 09:51 LICENSE
drwxr-xr-x  3 mtsouk  staff    96 Mar 13 09:51 cmd
-rw-r--r--  1 mtsouk  staff    669 Mar 13 09:51 main.go
$ ~/go/bin/cobra add cmdOne
cmdOne created at /Users/mtsouk/go/src/cli/cmd/cmdOne.go
$ ~/go/bin/cobra add cmdTwo
cmdTwo created at /Users/mtsouk/go/src/cli/cmd/cmdTwo.go
```

The `cobra init` command creates a new `cobra` project inside `~/go/src`, named after its last parameter (`cli`). The `cobra add` command adds a new command to the command-line tool and creates all necessary files and Go functions to support that command.

Therefore, after each execution of the `cobra add` command, Cobra does most of the dirty work for us. However, you will still need to implement the functionality of the commands you just added - in this case, the commands are called `cmdOne` and `cmdTwo`. The `cmdOne` command will accept a local command-line flag named `number` - you will also need to write some extra code for this feature to work.

At this point, if you execute `go run main.go`, you will get the following output:

```
A longer description that spans multiple lines and likely contains examples and usage of using your application. For example:  
  
Cobra is a CLI library for Go that empowers applications.  
This application is a tool to generate the needed files  
to quickly create a Cobra application.  
  
Usage:  
  cli [command]  
  
Available Commands:  
  cmdOne      A brief description of your command  
  cmdTwo      A brief description of your command  
  help        Help about any command  
  
Flags:  
  --config string  config file (default is $HOME/.cli.yaml)  
  -h, --help       help for cli  
  -t, --toggle     Help message for toggle  
  
Use "cli [command] --help" for more information about a command.
```

The Go code for the `cmdOne` command can be found in `./cmd/cmdOne.go` and for the `cmdTwo` command in `./cmd/cmdTwo.go`.

The final version of `./cmd/cmdOne.go` is as follows:

```
package cmd  
  
import (  
    "fmt"  
    "github.com/spf13/cobra"  
)  
  
// cmdOneCmd represents the cmdOne command  
var cmdOneCmd = &cobra.Command{  
    Use:   "cmdOne",
```

```

Short: "A brief description of your command",
Long: `A longer description that spans multiple lines and likely contains examples
and usage of using your command. For example:

Cobra is a CLI library for Go that empowers applications.
This application is a tool to generate the needed files
to quickly create a Cobra application.`,
Run: func(cmd *cobra.Command, args []string) {
    fmt.Println("cmdOne called!")
    number, _ := cmd.Flags().GetInt("number")
    fmt.Println("Going to use number", number)
    fmt.Sprintf("Square: %d\n", number*number)
},
}

func init() {
    rootCmd.AddCommand(cmdOneCmd)
    cmdOneCmd.Flags().Int("number", 0, "A help for number")
}

```

The previous Go code comes without the comments that are automatically generated by `cobra`. The following line of code in the `init()` function is what defines the new local command-line flag:

```
|cmdOneCmd.Flags().Int("number", 0, "A help for number")
```

That flag, which is called `number`, is used in the `cobra.Command` block as follows:

```
|number, _ := cmd.Flags().GetInt("number")
```

After that, you can use the `number` variable any way you want.

The Go code of the final version of `./cmd/cmdTwo.go`, which includes comments and license information, is as follows:

```

// Copyright © 2019 NAME HERE <EMAIL ADDRESS>
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

package cmd

import (
    "fmt"
    "github.com/spf13/cobra"
)

```

```

// cmdTwoCmd represents the cmdTwo command
var cmdTwoCmd = &cobra.Command{
    Use:   "cmdTwo",
    Short: "A brief description of your command",
    Long:  `A longer description that spans multiple lines and likely contains examples
and usage of using your command. For example:

Cobra is a CLI library for Go that empowers applications.
This application is a tool to generate the needed files
to quickly create a Cobra application.`,
    Run: func(cmd *cobra.Command, args []string) {
        fmt.Println("cmdTwo called!")
    },
}

func init() {
    rootCmd.AddCommand(cmdTwoCmd)

    // Here you will define your flags and configuration settings.

    // Cobra supports Persistent Flags which will work for this command
    // and all subcommands, e.g.:
    // cmdTwoCmd.PersistentFlags().String("foo", "", "A help for foo")

    // Cobra supports local flags which will only run when this command
    // is called directly, e.g.:
    // cmdTwoCmd.Flags().BoolP("toggle", "t", false, "Help message for toggle")
}

```

This is the default implementation of the `cmdTwo` command generated by `cobra`.

Executing the `cli` tool will generate the following kind of output:

```

$ go run main.go cmdOne
cmdOne called!
Going to use number 0
Square: 0
$ go run main.go cmdOne --number -20
cmdOne called!
Going to use number -20
Square: 400
$ go run main.go cmdTwo
cmdTwo called!

```

If you give the `cli` the tool wrong input, it will generate the following kind of error messages:

```

$ go run main.go cmdThree
Error: unknown command "cmdThree" for "cli"
Run 'cli --help' for usage.
unknown command "cmdThree" for "cli"
exit status 1
$ go run main.go cmdOne --n -20
Error: unknown flag: --n
Usage:
  cli cmdOne [flags]

Flags:

```

```
-h, --help      help for cmdOne
--number int   A help for number

Global Flags:
  --config string   config file (default is $HOME/.cli.yaml)

unknown flag: --n
exit status 1
```

The help screen for the `cmdOne` command, which is automatically generated, is as follows:

```
$ go run main.go cmdOne --help
A longer description that spans multiple lines and likely contains examples
and usage of using your command. For example:

Cobra is a CLI library for Go that empowers applications.
This application is a tool to generate the needed files
to quickly create a Cobra application.

Usage:
  cli cmdOne [flags]

Flags:
  -h, --help      help for cmdOne
  --number int   A help for number

Global Flags:
  --config string   config file (default is $HOME/.cli.yaml)
```

The directory structure and the files of the final version of the utility can be displayed with the help of the `tree(1)` command:

```
$ tree
.
├── LICENSE
└── cmd
    ├── cmdOne.go
    ├── cmdTwo.go
    └── root.go
└── main.go

1 directory, 5 files
```

Creating command aliases

In this section, you will learn how to create aliases for existing commands with `cobra`.

As before, you will first need to create a new `cobra` project, which in this case is going to be called `aliases`, along with the desired commands. This can be done as follows:

```
$ ~/go/bin/cobra init aliases
$ cd ~/go/src/aliases
$ ~/go/bin/cobra add initialization
initialization created at /Users/mtsouk/go/src/aliases/cmd/initialization.go
$ ~/go/bin/cobra add preferences
preferences created at /Users/mtsouk/go/src/aliases/cmd/preferences.go
```

So far, you have a command-line utility that supports two commands, named `initialization` and `preferences`.

Each alias of an existing `cobra` command needs to be explicitly specified in the Go code. For the `initialization` command, you will need the following line of code in the appropriate place in `./cmd/initialization.go`:

```
|     Aliases: []string{"initialize", "init"},
```

The previous statement creates two aliases for the `initialization` command, named `initialize` and `init`. The contents of the final version of `./cmd/initialization.go`, without any comments, will be as follows:

```
package cmd

import (
    "fmt"
    "github.com/spf13/cobra"
)

var initializationCmd = &cobra.Command{
    Use:      "initialization",
    Aliases:  []string{"initialize", "init"},
    Short:    "A brief description of your command",
    Long:     `A longer description of your command`,
    Run: func(cmd *cobra.Command, args []string) {
        fmt.Println("initialization called")
```

```

        },
    }

func init() {
    rootCmd.AddCommand(initializationCmd)
}

```

Similarly, for the `preferences` command, you will need to include the following line of Go code in the appropriate place in `./cmd/preferences.go`:

```
Aliases: []string{"prefer", "pref", "prf"},
```

The aforementioned statement creates three aliases for the `preferences` command, named `prefer`, `pref`, and `prf`.

The contents of the final version of `./cmd/preferences.go`, without any comments, are as follows:

```

package cmd

import (
    "fmt"
    "github.com/spf13/cobra"
)

var preferencesCmd = &cobra.Command{
    Use:      "preferences",
    Aliases:  []string{"prefer", "pref", "prf"},
    Short:    "A brief description of your command",
    Long:     `A longer description of your command`,
    Run: func(cmd *cobra.Command, args []string) {
        fmt.Println("preferences called")
    },
}

func init() {
    rootCmd.AddCommand(preferencesCmd)
}

```

Executing the `aliases` command-line utility will generate the following kind of output:

```

$ go run main.go prefer
preferences called
$ go run main.go prf
preferences called
$ go run main.go init
initialization called

```

If you give erroneous commands to `aliases`, then it will generate the following kind of error messages:

```
$ go run main.go initS
Error: unknown command "initS" for "aliases"
Run 'aliases --help' for usage.
unknown command "initS" for "aliases"
exit status 1
$ go run main.go preferr
Error: unknown command "preferr" for "aliases"

Did you mean this?
    preferences

Run 'aliases --help' for usage.
unknown command "preferr" for "aliases"

Did you mean this?
    preferences

exit status 1
```

The `tree(1)` command, which does not come with most UNIX systems and must be installed separately, can help us to get an idea of the directory structure and the files of the generated `cobra` utility:

```
$ tree
.
├── LICENSE
└── cmd
    ├── initialization.go
    ├── preferences.go
    └── root.go
└── main.go

1 directory, 5 files
```



Both `viper` and `cobra` have more features and capabilities than the ones presented here.

The `io.Reader` and `io.Writer` Interfaces

As stated in the previous chapter, compliance with the `io.Reader` interface requires the implementation of the `Read()` method, whereas if you want to satisfy the `io.Writer` interface guidelines, you will need to implement the `Write()` method. Both these interfaces are very popular in Go and we will put them to use in a little while.

Buffered and unbuffered file input and output

Buffered file input and output happens when there is a buffer for temporarily storing data before reading data or writing data. Thus, instead of reading a file byte by byte, you read many bytes at once. You put the data in a buffer and wait for someone to read it in the desired way. Unbuffered file input and output happens when there is no buffer to temporarily store data before actually reading or writing it.

The next question that you might ask is how to decide when to use buffered and when to use unbuffered file input and output. When dealing with critical data, unbuffered file input and output is generally a better choice because buffered reads might result in out-of-date data and buffered writes might result in data loss when the power of your computer is interrupted. However, most of the time, there is no definitive answer to that question. This means that you can use whatever makes your tasks easier to implement.

The `bufio` package

As the name suggests, the `bufio` package is about buffered input and output. However, the `bufio` package still uses (internally) the `io.Reader` and `io.Writer` objects, which it wraps in order to create the `bufio.Reader` and `bufio.Writer` objects, respectively. As you will see in the forthcoming sections, the `bufio` package is very popular for reading text files.

Reading text files

A text file is the most common kind of file that you can find on a UNIX system. In this section, you will learn how to read text files in three ways: line by line, word by word, and character by character. As you will see, reading a text file line by line is the easiest method to access a text file, while reading a text file word by word is the most difficult method of all.

If you look closely at the `byLine.go`, `byWord.go`, and `byCharacter.go` programs, you will see many similarities in their Go code. Firstly, all three utilities read the input file line by line. Secondly, all three utilities have the same `main()` function, with the exception of the function that is called in the `for` loop of the `main()` function. Lastly, the three functions that process the input text files are almost identical, except for the part that implements the actual functionality of the function.

Reading a text file line by line

Going line by line is the most common method of reading a text file. This is the main reason that it is being shown first. The Go code of `byLine.go`, which will be presented in three parts, will help you to understand the technique.

The first code segment from `byLine.go` is shown in the following Go code:

```
package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "os"
)
```

As you can see by the presence of the `bufio` package, we will use buffered input.

The second part of `byLine.go` contains the following Go code:

```
func lineByLine(file string) error {
    var err error

    f, err := os.Open(file)
    if err != nil {
        return err
    }
    defer f.Close()

    r := bufio.NewReader(f)
    for {
        line, err := r.ReadString('\n')
        if err == io.EOF {
            break
        } else if err != nil {
            fmt.Printf("error reading file %s", err)
            break
        }
        fmt.Print(line)
    }
    return nil
}
```

All the magic happens in the `lineByLine()` function. After making sure that you can open the given filename for reading, you create a new reader using `bufio.NewReader()`. Then, you use that reader with `bufio.ReadString()` in order to read the input file line by line. The trick is done by the parameter of `bufio.ReadString()`, which is a character that tells `bufio.ReadString()` to keep reading until that character is found. Constantly calling `bufio.ReadString()` when that parameter is the newline character results in reading the input file line by line. The use of `fmt.Print()` instead of `fmt.Println()` for printing the input line shows that the newline character is included in each input line.

The third part of `byLine.go` follows:

```
func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: byLine <file1> [<file2> ...]\n")
        return
    }

    for _, file := range flag.Args() {
        err := lineByLine(file)
        if err != nil {
            fmt.Println(err)
        }
    }
}
```

Executing `byLine.go` and processing its output with `wc(1)` will generate the following type of output:

```
$ go run byLine.go /tmp/swtag.log /tmp/adobegc.log | wc
 4761   88521  568402
```

The following command will verify the accuracy of the preceding output:

```
$ wc /tmp/swtag.log /tmp/adobegc.log
 131      693     8440 /tmp/swtag.log
 4630    87828  559962 /tmp/adobegc.log
 4761   88521  568402 total
```

Reading a text file word by word

The technique presented in this subsection will be demonstrated by the `byWord.go` file and shown in four parts. As you will see in the Go code, separating the words of a line can be tricky.

The first part of this utility is as follows:

```
package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "os"
    "regexp"
)
```

The second code portion of `byWord.go` is shown in the following Go code:

```
func wordByWord(file string) error {
    var err error
    f, err := os.Open(file)
    if err != nil {
        return err
    }
    defer f.Close()

    r := bufio.NewReader(f)
    for {
        line, err := r.ReadString('\n')
        if err == io.EOF {
            break
        } else if err != nil {
            fmt.Printf("error reading file %s", err)
            return err
        }
    }
}
```

This part of the `wordByWord()` function is the same as the `lineByLine()` function of the `byLine.go` utility.

The third part of `byWord.go` is as follows:

```
r := regexp.MustCompile("[^\\s]+")
words := r.FindAllString(line, -1)
for i := 0; i < len(words); i++ {
```

```

        fmt.Println(words[i])
    }
}
return nil
}
}

```

The remaining code of the `wordByWord()` function is totally new and it uses regular expressions to separate the words found in each line of the input file. The regular expression defined in the `regexp.MustCompile("[^\s]+")` statement states that empty characters will separate one word from another.

The last code segment of `byWord.go` is as follows:

```

func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: byWord <file1> [<file2> ...]\n")
        return
    }

    for _, file := range flag.Args() {
        err := wordByWord(file)
        if err != nil {
            fmt.Println(err)
        }
    }
}

```

Executing `byWord.go` will produce the following type of output:

```

$ go run byWord.go /tmp/adobegc.log
01/08/18
20:25:09:669
|
[INFO]

```

You can verify the validity of `byWord.go` with the help of the `wc(1)` utility:

```

$ go run byWord.go /tmp/adobegc.log | wc
    91591    91591   559005
$ wc /tmp/adobegc.log
    4831    91591   583454 /tmp/adobegc.log

```

As you can see, the number of words calculated by `wc(1)` is the same as the number of lines and words that you took from the execution of `byWord.go`.

Reading a text file character by character

In this section, you will learn how to read a text file character by character, which is a pretty rare requirement unless you want to develop a text editor. The relevant Go code will be saved as `byCharacter.go`, which will be presented in four parts.

The first part of `byCharacter.go` is shown in the following Go code:

```
package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "os"
)
```

As you can see, you will not need to use regular expressions for this task.

The second code segment from `byCharacter.go` is as follows:

```
func charByChar(file string) error {
    var err error
    f, err := os.Open(file)
    if err != nil {
        return err
    }
    defer f.Close()

    r := bufio.NewReader(f)
    for {
        line, err := r.ReadString('\n')
        if err == io.EOF {
            break
        } else if err != nil {
            fmt.Printf("error reading file %s", err)
            return err
        }
    }
}
```

The third part of `byCharacter.go` is where the logic of the program is found:

```

        for _, x := range line {
            fmt.Println(string(x))
        }
    return nil
}

```

Here, you take each line that you read and split it using `range`, which returns two values. You discard the first, which is the location of the current character in the `line` variable, and you use the second. However, that value is not a character - that is the reason you have to convert it into a character using the `string()` function.

Note that, due to the `fmt.Println(string(x))` statement, each character is printed in a distinct line, which means that the output of the program will be large. If you want a more compressed output, you should use the `fmt.Print()` function instead.

The last part of `byCharacter.go` contains the following Go code:

```

func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: byChar <file1> [<file2> ...]\n")
        return
    }

    for _, file := range flag.Args() {
        err := charByChar(file)
        if err != nil {
            fmt.Println(err)
        }
    }
}

```

The execution of `byCharacter.go` will generate the following type of output:

```

$ go run byCharacter.go /tmp/adobegc.log
0
1
/
0
8
/
1
8

```

Note that the Go code presented here can be used for counting the number of characters found in the input file, which can help you to implement a Go version of the handy `wc(1)` command-line utility.

Reading from /dev/random

In this section, you will learn how to read from the `/dev/random` system device. The purpose of the `/dev/random` system device is to generate random data, which you might use for testing your programs or, in this case, as the seed for a random number generator. Getting data from `/dev/random` can be a little bit tricky, and this is the main reason for specifically discussing it here.

On a macOS Mojave machine, the `/dev/random` file has the following permissions:

```
$ ls -l /dev/random
crw-rw-rw- 1 root wheel 14,   0 Mar 12 20:24 /dev/random
```

Similarly, on a Debian Linux machine, the `/dev/random` system device has the following UNIX file permissions:

```
$ ls -l /dev/random
crw-rw-rw- 1 root root 1, 8 Oct 13 12:19 /dev/random
```

This means that the `/dev/random` file has analogous file permissions on both UNIX variants. The only difference between these two UNIX variants is the UNIX group that owns the file, which is `wheel` on macOS and `root` on Debian Linux.

The name of the program for this topic is `devRandom.go`, and it will be presented in three parts. The first part of the program is as follows:

```
package main

import (
    "encoding/binary"
    "fmt"
    "os"
)
```

In order to read from `/dev/random`, you will need to import the `encoding/binary` standard Go package, because `/dev/random` returns binary data that needs to be

decoded. The second code portion of `devRandom.go` follows:

```
func main() {
    f, err := os.Open("/dev/random")
    defer f.Close()

    if err != nil {
        fmt.Println(err)
        return
    }
```

You open `/dev/random` as usual because everything in UNIX is a file.

The last code segment of `devRandom.go` is shown in the following Go code:

```
var seed int64
binary.Read(f, binary.LittleEndian, &seed)
fmt.Println("Seed:", seed)
```

You need the `binary.Read()` function, which requires three parameters, in order to read from the `/dev/random` system device. The value of the second parameter (`binary.LittleEndian`) specifies that you want to use the **little endian** byte order. The other option is `binary.BigEndian`, which is used when your computer is using the **big endian** byte order.

Executing `devRandom.go` will generate the following type of output:

```
$ go run devRandom.go
Seed: -2044736418491485077
$ go run devRandom.go
Seed: -5174854372517490328
$ go run devRandom.go
Seed: 7702177874251412774
```

Reading a specific amount of data

In this section, you will learn how to read exactly the amount of data you want. This technique is particularly useful when reading binary files, where you have to decode the data you read in a particular way. Nevertheless, this technique still works with text files.

The logic behind this technique is as follows: you create a **byte slice** with the size you need and use that byte slice for reading. To make this more interesting, this functionality is going to be implemented as a function with two parameters. One parameter will be used to specify the amount of data that you want to read, and the other parameter, which will have the `*os.File` type, will be used to access the desired file. The return value of that function will be the data you have read.

The name of the Go program for this topic will be `readsize.go` and it will be presented in four parts. The utility will accept a single parameter, which will be the size of the byte slice.



This particular program, when used with the presented technique, can help you to copy any file using the buffer size you want.

The first part of `readSize.go` has the expected preamble:

```
package main

import (
    "fmt"
    "io"
    "os"
    "strconv"
)
```

The second part of `readSize.go` contains the following Go code:

```
func readSize(f *os.File, size int) []byte {
    buffer := make([]byte, size)

    n, err := f.Read(buffer)
    if err == io.EOF {
        return nil
    }
}
```

```

    }

    if err != nil {
        fmt.Println(err)
        return nil
    }

    return buffer[0:n]
}

```

This is the function discussed earlier. Although the code is straightforward, there is one point that needs an explanation. The `io.Reader.Read()` method returns two parameters: the number of bytes read and an `error` variable. The `readSize()` function uses the former return value of `io.Read()` in order to return a byte slice of that size. Although this is a tiny detail, and it is only significant when you reach the end of the file, it ensures that the output of the utility will be same as the input and that it will not contain any extra characters.

Finally, there is code that checks for `io.EOF`, which is an error that signifies that you have reached the end of a file. When that kind of error occurs, the function returns. Dave Cheney, a project member for Go and open-source contributor, calls these errors "sentinel errors" because they signify that an error did not occur.

The third code portion of this utility is as follows:

```

func main() {
    arguments := os.Args
    if len(arguments) != 3 {
        fmt.Println("<buffer size> <filename>")
        return
    }

    bufferSize, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }

    file := os.Args[2]
    f, err := os.Open(file)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer f.Close()
}

```

The last code segment of `readSize.go` is as follows:

```
| for {
|     readData := readSize(f, bufferSize)
|     if readData != nil {
|         fmt.Println(string(readData))
|     } else {
|         break
|     }
| }
```

Here, you keep reading your input file until `readSize()` returns an error or `nil`.

Executing `readSize.go` by telling it to process a binary file and handling its output with `wc(1)` will validate the correctness of the program:

```
| $ go run readSize.go 1000 /bin/ls | wc
|     80      1032    38688
| $ wc /bin/ls
|     80      1032    38688 /bin/ls
```

The advantages of binary formats

In the previous section, the `readsize.go` utility illustrated how you can read a file byte by byte, which is a technique that best applies to binary files. So, you might ask, why read data in binary format when text formats are so much easier to understand? The main reason is space reduction. Imagine that you want to store the number 20 as a string to a file. It is easy to understand that you will need two bytes to store 20 using ASCII characters: one for storing 2 and another for storing 0.

Storing 20 in binary format requires just one byte, since 20 can be represented as 00010100 in binary or as 0x14 in hexadecimal.

This difference might look insignificant when you are dealing with small amounts of data, but it could be pretty substantial when dealing with data found in applications such as database servers.

Reading CSV files

CSV files are plain text files with a format. In this section, you will learn how to read a text file that contains points of a plane, which means that each line will contain a pair of coordinates. Additionally, you are also going to use an external Go library named **Glot**, which will help you to create a plot of the points that you read from the CSV file. Note that Glot uses **Gnuplot**, which means that you will need to install Gnuplot on your UNIX machine in order to use Glot.

The name of the source file for this topic is `csvplot.go`, and it is going to be presented in five parts. The first code segment is as follows:

```
package main

import (
    "encoding/csv"
    "fmt"
    "github.com/Arafatk/glot"
    "os"
    "strconv"
)
```

The second part of `csvplot.go` is shown in the following Go code:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Println("Need a data file!")
        return
    }

    file := os.Args[1]
    _, err := os.Stat(file)
    if err != nil {
        fmt.Println("Cannot stat", file)
        return
    }
}
```

In this part, you can see a technique for checking whether a file already exists or not using the powerful `os.Stat()` function.

The third part of `csvplot.go` is as follows:

```

f, err := os.Open(file)
if err != nil {
    fmt.Println("Cannot open", file)
    fmt.Println(err)
    return
}
defer f.Close()

reader := csv.NewReader(f)
reader.FieldsPerRecord = -1
allRecords, err := reader.ReadAll()
if err != nil {
    fmt.Println(err)
    return
}

```

The fourth code segment of `csvplot.go` is shown in the following Go code:

```

xP := []float64{}
yP := []float64{}
for _, rec := range allRecords {
    x, _ := strconv.ParseFloat(rec[0], 64)
    y, _ := strconv.ParseFloat(rec[1], 64)
    xP = append(xP, x)
    yP = append(yP, y)
}

points := [][]float64{}
points = append(points, xP)
points = append(points, yP)
fmt.Println(points)

```

Here, you convert the string values you read into numbers and put them into a slice with two dimensions named `points`.

The last part of `csvplot.go` contains the following Go code:

```

dimensions := 2
persist := true
debug := false
plot, _ := glot.NewPlot(dimensions, persist, debug)

plotSetTitle("Using Glot with CSV data")
plotSetXLabel("X-Axis")
plotSetYLabel("Y-Axis")
style := "circle"
plotAddPointGroup("Circle:", style, points)
plotSavePlot("output.png")
}

```

In the preceding Go code, you saw how you can create a PNG file with the help of the Glot library and its `glot.SavePlot()` function.

As you might guess, you will need to download the GLOT library before being able to compile and execute the `csvplot.go` source code, which requires the execution of the following command from your favorite UNIX shell:

```
| $ go get github.com/Arafatk/glot
```

The CSV data file containing the points that will be plotted has the following format:

```
| $ cat /tmp/dataFile
| 1,2
| 2,3
| 3,3
| 4,4
| 5,8
| 6,5
| -1,12
| -2,10
| -3,10
| -4,10
```

Executing `csvplot.go` will generate the following kind of output:

```
| $ go run CSVplot.go /tmp/doesNotExist
| Cannot stat /tmp/doesNotExist
| $ go run CSVplot.go /tmp/dataFile
| [[1 2 3 4 5 6 -1 -2 -3 -4] [2 3 3 4 8 5 12 10 10 10]]
```

You can see the results of `csvplot.go` in a much better format in the following figure:

Using GLOT with CSV data

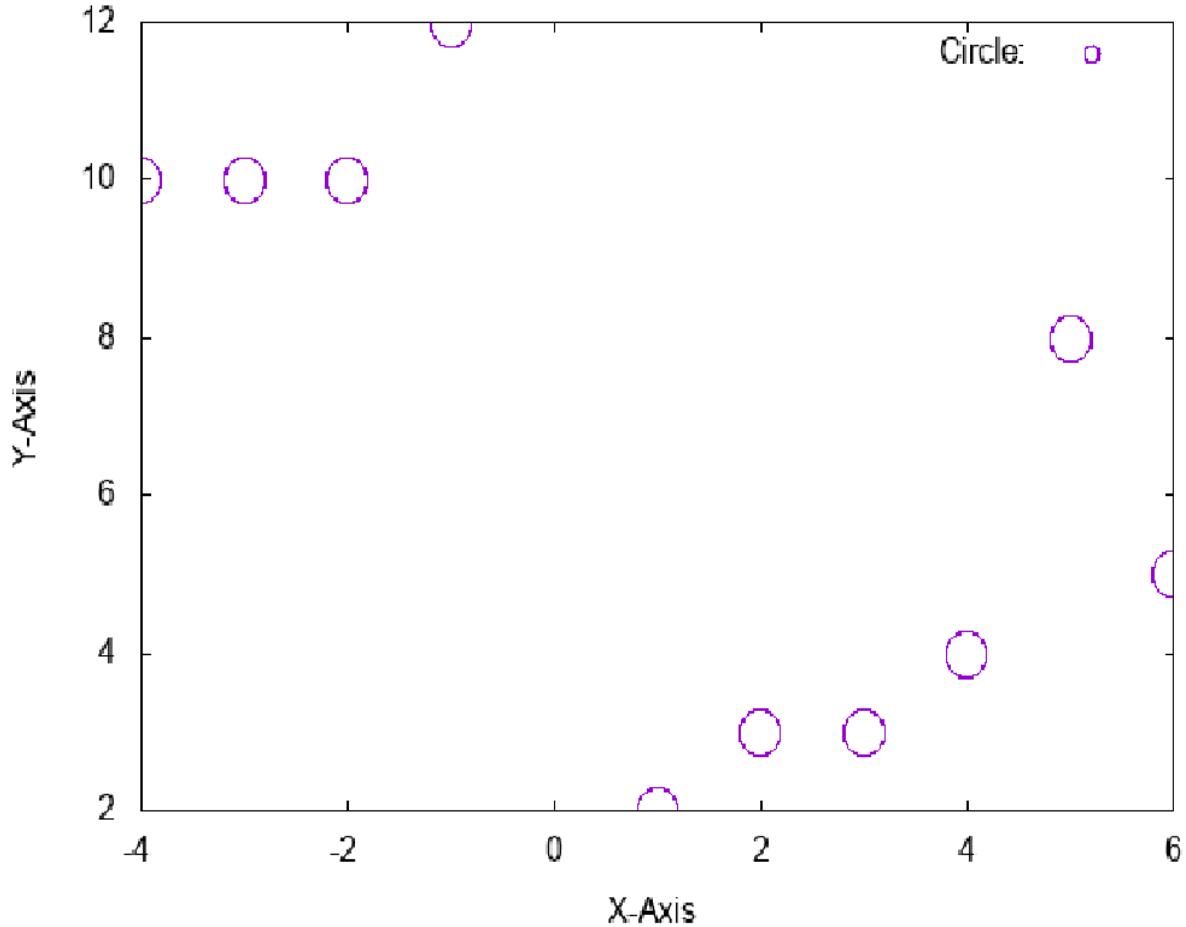


Figure 8.1: The type of graphical output you can get from GLOT

Writing to a file

Generally speaking, you can use the functionality of the `io.Writer` interface for writing data to files on a disk. Nevertheless, the Go code of `save.go` will show you five ways to write data to a file. The `save.go` program will be presented in six parts.

The first part of `save.go` is as follows:

```
package main

import (
    "bufio"
    "fmt"
    "io"
    "io/ioutil"
    "os"
)
```

The second code portion of `save.go` is shown in the following Go code:

```
func main() {
    s := []byte("Data to write\n")

    f1, err := os.Create("f1.txt")
    if err != nil {
        fmt.Println("Cannot create file", err)
        return
    }
    defer f1.Close()
    fmt.Fprintf(f1, string(s))
```

Notice that the `s` byte slice will be used in every line that involves writing presented in this Go program. Additionally, the `fmt.Fprintf()` function used here can help you to write data to your own log files using the format you want. In this case, `fmt.Fprintf()` writes your data to the file identified by `f1`.

The third part of `save.go` contains the following Go code:

```
f2, err := os.Create("f2.txt")
if err != nil {
    fmt.Println("Cannot create file", err)
    return
}
defer f2.Close()
```

```
|     n, err := f2.WriteString(string(s))
|     fmt.Printf("wrote %d bytes\n", n)
```

In this case, `f2.WriteString()` is used for writing your data to a file.

The fourth code segment of `save.go` is next:

```
|     f3, err := os.Create("f3.txt")
|     if err != nil {
|         fmt.Println(err)
|         return
|     }
|     w := bufio.NewWriter(f3)
|     n, err = w.WriteString(string(s))

|     fmt.Printf("wrote %d bytes\n", n)
|     w.Flush()
```

In this case, `bufio.NewWriter()` opens a file for writing and `bufio.WriteString()` writes the data.

The fifth part of `save.go` will teach you another method for writing to a file:

```
|     f4 := "f4.txt"
|     err = ioutil.WriteFile(f4, s, 0644)
|     if err != nil {
|         fmt.Println(err)
|         return
|     }
```

This method needs just a single function call named `ioutil.WriteFile()` for writing your data, and it does not require the use of `os.Create()`.

The last code segment of `save.go` is as follows:

```
|     f5, err := os.Create("f5.txt")
|     if err != nil {
|         fmt.Println(err)
|         return
|     }
|     n, err = io.WriteString(f5, string(s))
|     if err != nil {
|         fmt.Println(err)
|         return
|     }
|     fmt.Printf("wrote %d bytes\n", n)
| }
```

The last technique uses `io.WriteString()` to write the desired data to a file.

Executing `save.go` will create the following type of output:

```
$ go run save.go
wrote 14 bytes
wrote 14 bytes
wrote 14 bytes
$ ls -l f?.txt
-rw-r--r-- 1 mtsouk staff 14 Jan 23 20:30 f1.txt
-rw-r--r-- 1 mtsouk staff 14 Jan 23 20:30 f2.txt
-rw-r--r-- 1 mtsouk staff 14 Jan 23 20:30 f3.txt
-rw-r--r-- 1 mtsouk staff 14 Jan 23 20:30 f4.txt
-rw-r--r-- 1 mtsouk staff 14 Jan 23 20:30 f5.txt
$ cat f?.txt
Data to write
```

The next section will show you how to save data to a file with the help of a specialized function of a package that is in the standard Go library.

Loading and saving data on disk

Do you remember the `keyValue.go` application from [Chapter 4](#), *The Uses of Composite Types*? Well, it was far from perfect, so in this section you will learn how to save the data of a **key-value store** on disk and how to load it back into memory when you next start your application.

We are going to create two new functions, one named `save()` for saving data to disk and another named `load()` for retrieving the data from disk. Thus, we will only present the code differences between `keyValue.go` and `kvSaveLoad.go` using the `diff(1)` UNIX command-line utility.



The `diff(1)` UNIX command-line utility can be very convenient when you want to spot the differences between two text files. You can learn more about it by executing `man 1 diff` at the command line of your favorite UNIX shell.

If you think about the task to be implemented here, you will recognize that what you really need is an easy way to save the contents of a Go map to disk, as well as a way to load the data from a file and put it into a Go map.

The process of converting your data into a stream of bytes is called **serialization**. The process of reading a data file and converting it into an object is called **deserialization**. The `encoding/gob` standard Go package will be used for the `kvSaveLoad.go` program. It will help the serialization and deserialization process. The `encoding/gob` package uses the **gob format** to store its data. The official name for such formats is **stream encoding**. The good thing about the gob format is that Go does all of the dirty work, so you do not have to worry about the encoding and decoding stages.

Other Go packages that can help you to serialize and deserialize data are `encoding/xml`, which uses the **XML format**, and `encoding/json`, which uses the **JSON format**.

The following output reveals the code changes between `kvSaveLoad.go` and `keyValue.go` without including the implementations of the `save()` and `load()`

functions, which will be fully presented shortly:

```
$ diff keyValue.go kvSaveLoad.go
4a5
>     "encoding/gob"
16a18,55
> var DATAFILE = "/tmp/dataFile.gob"
> func save() error {
>
>     return nil
> }
>
> func load() error {
>
> }
59a99,104
>
>     err := load()
>     if err != nil {
>         fmt.Println(err)
>     }
>
88a134,137
>             err = save()
>             if err != nil {
>                 fmt.Println(err)
>             }
```

An important part of the `diff(1)` output is the definition of the `DATAFILE` global variable, which holds the path to the file that is used by the key-value store. Apart from this, you can see where the `load()` function is called, as well as the point where the `save()` function is called. The `load()` function is used first in the `main()` function, while the `save()` function is executed when the user issues the `STOP` command.

Each time you execute `kvSaveLoad.go`, the program checks whether there is data to be read by trying to read the default data file. If there is no data file to read, you will start with an empty key-value store. When the program is about to terminate, it writes all of its data on the disk using the `save()` function.

The `save()` function has the following implementation:

```
func save() error {
    fmt.Println("Saving", DATAFILE)
    err := os.Remove(DATAFILE)
    if err != nil {
        fmt.Println(err)
    }
```

```

    saveTo, err := os.Create(DATAFILE)
    if err != nil {
        fmt.Println("Cannot create", DATAFILE)
        return err
    }
    defer saveTo.Close()

    encoder := gob.NewEncoder(saveTo)
    err = encoder.Encode(DATA)
    if err != nil {
        fmt.Println("Cannot save to", DATAFILE)
        return err
    }
    return nil
}

```

Note that the first thing the `save()` function does is to delete the existing data file using the `os.Remove()` function in order to create it later on.

One of the most critical tasks the `save()` function does is to make sure that you can actually create and write to the desired file. Although there are many ways to do this, the `save()` function uses the simplest way, which is checking the `error` value returned by the `os.Create()` function. If that value is not `nil`, then there is a problem and the `save()` function finishes without saving any data.

The `load()` function is implemented as follows:

```

func load() error {
    fmt.Println("Loading", DATAFILE)
    loadFrom, err := os.Open(DATAFILE)
    defer loadFrom.Close()
    if err != nil {
        fmt.Println("Empty key/value store!")
        return err
    }

    decoder := gob.NewDecoder(loadFrom)
    decoder.Decode(&DATA)
    return nil
}

```

One of the tasks of the `load()` function is to make sure that the file that you are trying to read is actually there and that you can read it without any problems.

Once again, the `load()` function uses the simplest approach, which is to look at the return value of the `os.Open()` function. If the `error` value returned is

equal to `nil`, then everything is fine.

It is also important to close the file after reading the data from it, as it will be overwritten by the `save()` function later on. The release of the file is accomplished by the `defer loadFrom.Close()` statement.

Executing `kvSaveLoad.go` will generate the following type of output:

```
$ go run kvSaveLoad.go
Loading /tmp/dataFile.gob
Empty key/value store!
open /tmp/dataFile.gob: no such file or directory
ADD 1 2 3
ADD 4 5 6
STOP
Saving /tmp/dataFile.gob
remove /tmp/dataFile.gob: no such file or directory
$ go run kvSaveLoad.go
Loading /tmp/dataFile.gob
PRINT
key: 1 value: {2 3 }
key: 4 value: {5 6 }
DELETE 1
PRINT
key: 4 value: {5 6 }
STOP
Saving /tmp/dataFile.gob
rMacBook:code mtsouk$ go run kvSaveLoad.go
Loading /tmp/dataFile.gob
PRINT
key: 4 value: {5 6 }
STOP
Saving /tmp/dataFile.gob
$ ls -l /tmp/dataFile.gob
-rw-r--r-- 1 mtsouk wheel 80 Jan 22 11:22 /tmp/dataFile.gob
$ file /tmp/dataFile.gob
/tmp/dataFile.gob: data
```

In [Chapter 13](#), *Network Programming – Building Your Own Servers and Clients*, you are going to see the final version of the key-value store, which will be able to operate over a TCP/IP connection and will serve multiple network clients using **goroutines**.

The strings package revisited

We first talked about the handy `strings` package back in [Chapter 4, *The Uses of Composite Types*](#). This section will address the functions of the `strings` package that are related to file input and output.

The first part of `str.go` is shown in the following Go code:

```
package main

import (
    "fmt"
    "io"
    "os"
    "strings"
)
```

The second code segment of `str.go` is as follows:

```
func main() {
    r := strings.NewReader("test")
    fmt.Println("r length:", r.Len())
```

The `strings.NewReader()` function creates a read-only `Reader` from a string. The `strings.Reader` object implements the `io.Reader`, `io.ReaderAt`, `io.Seeker`, `io.WriterTo`, `io.ByteScanner`, and `io.RuneScanner` interfaces.

The third part of `str.go` follows:

```
b := make([]byte, 1)
for {
    n, err := r.Read(b)
    if err == io.EOF {
        break
    }

    if err != nil {
        fmt.Println(err)
        continue
    }
    fmt.Printf("Read %s Bytes: %d\n", b, n)
}
```

Here, you can see how to use `strings.Reader` as an `io.Reader` in order to read a string byte by byte using the `Read()` function.

The last code portion of `str.go` contains the following Go code:

```
s := strings.NewReader("This is an error!\n")
fmt.Println("r length:", s.Len())
n, err := s.WriteTo(os.Stderr)

if err != nil {
    fmt.Println(err)
    return
}
fmt.Printf("Wrote %d bytes to os.Stderr\n", n)
```

In this code segment, you can see how you can write to standard error with the help of the `strings` package.

Executing `str.go` will generate the following output:

```
$ go run str.go
r length: 4
Read t Bytes: 1
Read e Bytes: 1
Read s Bytes: 1
Read t Bytes: 1
r length: 18
This is an error!
Wrote 18 bytes to os.Stderr
$ go run str.go 2>/dev/null
r length: 4
Read t Bytes: 1
Read e Bytes: 1
Read s Bytes: 1
Read t Bytes: 1
r length: 18
Wrote 18 bytes to os.Stderr
```

About the bytes package

The `bytes` standard Go package contains functions for working with **byte slices** in the same way that the `strings` standard Go package helps you to work with **strings**. The name of the Go source code file is `bytes.go`, and it will be presented in three code portions.

The first part of `bytes.go` follows:

```
package main

import (
    "bytes"
    "fmt"
    "io"
    "os"
)
```

The second code portion of `bytes.go` contains the following Go code:

```
func main() {
    var buffer bytes.Buffer
    buffer.Write([]byte("This is"))
    fmt.Fprintf(&buffer, " a string!\n")
    buffer.WriteTo(os.Stdout)
    buffer.WriteTo(os.Stdout)
```

First, you create a new `bytes.Buffer` variable and you put data into it using `buffer.Write()` and `fmt.Fprintf()`. Then, you call `buffer.WriteTo()` twice.

The first `buffer.WriteTo()` call will print the contents of the `buffer` variable. However, the second call to `buffer.WriteTo()` has nothing to print because the `buffer` variable will be empty after the first `buffer.WriteTo()` call.

The last part of `bytes.go` is as follows:

```
buffer.Reset()
buffer.Write([]byte("Mastering Go!"))
r := bytes.NewReader([]byte(buffer.String()))
fmt.Println(buffer.String())
for {
    b := make([]byte, 3)
    n, err := r.Read(b)
    if err == io.EOF {
```

```
        break
    }

    if err != nil {
        fmt.Println(err)
        continue
    }
    fmt.Printf("Read %s Bytes: %d\n", b, n)
}
}
```

The `Reset()` method resets the `buffer` variable and the `Write()` method puts some data into it again. Then, you create a new reader using `bytes.NewReader()`, and after that you use the `Read()` method of the `io.Reader` interface to read the data found in the `buffer` variable.

Executing `bytes.go` will create the following type of output:

```
$ go run bytes.go
This is a string!
Mastering Go!
Read Mas Bytes: 3
Read ter Bytes: 3
Read ing Bytes: 3
Read Go Bytes: 3
Read ! Bytes: 1
```

File permissions

A popular topic in UNIX systems programming is UNIX file permissions. In this section, you will learn how to print the permissions of any file, provided that you have the required UNIX permission to do so. The name of the program is `permissions.go`, and it will be presented in three parts.

The first part of `permissions.go` contains the following Go code:

```
package main  
import (  
    "fmt"  
    "os"  
)
```

The second code segment of `permissions.go` is shown in the following Go code:

```
func main() {  
    arguments := os.Args  
    if len(arguments) == 1 {  
        fmt.Printf("usage: permissions filename\n")  
        return  
    }
```

The last part of this utility follows:

```
    filename := arguments[1]  
    info, _ := os.Stat(filename)  
    mode := info.Mode()  
    fmt.Println(filename, "mode is", mode.String()[1:10])  
}
```

The call to `os.Stat(filename)` returns a big structure with lots of data. As we are only interested in the permissions of the file, we call the `Mode()` method and print its output. Actually, we are printing a part of the output denoted by `mode.String()[1:10]` because this is where the data that interests us is found.

Executing `permissions.go` will create the following type of output:

```
$ go run permissions.go /tmp/adobegc.log  
/tmp/adobegc.log mode is rw-rw-rw-
```

```
| $ go run permissions.go /dev/random  
| /dev/random mode is crw-rw-rw
```

The output of the `ls(1)` utility verifies the correctness of `permissions.go`:

```
| $ ls -l /dev/random /tmp/adobegc.log  
| crw-rw-rw- 1 root wheel 14, 0 Jan 8 20:24 /dev/random  
| -rw-rw-rw- 1 root wheel 583454 Jan 16 19:12 /tmp/adobegc.log
```

Handling UNIX signals

Go provides the `os/signal` package to help developers to work with signals. This section will show you how to use it for UNIX signal handling.

First, let me present some useful information about UNIX signals. Have you ever pressed Ctrl+C in order to stop a running program? If your answer is "yes," then you are already familiar with signals because Ctrl+C sends the **SIGINT signal** to a program. Strictly speaking, **UNIX signals** are software interrupts that can be accessed either by name or by number, and they offer a way to handle asynchronous events on a UNIX system.

Generally speaking, it is safer to send a signal by name because you are less likely to send the wrong signal accidentally.

A program cannot handle all of the available signals. Some signals cannot be caught, but nor can they be ignored. The `SIGKILL` and `SIGSTOP` signals cannot be caught, blocked, or ignored. The reason for this is that they provide the kernel and the root user with a way of stopping any process that they desire. The `SIGKILL` signal, which is also known by the number 9, is usually called in extreme conditions where you need to act fast. Thus, it is the only signal that is usually called by number, simply because it is quicker to do so.



signal.SIGINFO is not available on Linux machines, which means that if you find it in a Go program that you want to run on a Linux machine, you need to replace it with another signal, or your Go program will not be able to compile and execute.

The most common way to send a signal to a process is by using the `kill(1)` utility. By default, `kill(1)` sends the `SIGTERM` signal. If you want to find all of the supported signals on your UNIX machine, you should execute the `kill -1` command.

If you try to send a signal to a process without having the required permissions, `kill(1)` will not do the job and you will get an error message similar to the following:

```
| $ kill 1210  
| -bash: kill: (1210) - Operation not permitted
```

Handling two signals

In this subsection, you will learn how to handle two signals in a Go program using the code found in `handleTwo.go`, which will be presented in four parts. The signals that will be handled by `handleTwo.go` are `SIGINFO` and `SIGINT`, which in Go are named `syscall.SIGINFO` and `os.Interrupt`, respectively.



If you look at the documentation of the `os` package, you will find that the only two signals that are guaranteed to be present on all systems are `syscall.SIGKILL` and `syscall.SIGINT`, which in Go are also defined as `os.Kill` and `os.Interrupt`, respectively.

The first part of `handleTwo.go` contains the following Go code:

```
package main

import (
    "fmt"
    "os"
    "os/signal"
    "syscall"
    "time"
)
```

The second part of the `handleTwo.go` program follows:

```
func handleSignal(signal os.Signal) {
    fmt.Println("handleSignal() Caught:", signal)
}
```

The `handleSignal()` function will be used for handling the `syscall.SIGINFO` signal, while the `os.Interrupt` signal will be handled inline.

The third code segment of `handleTwo.go` is shown in the following Go code:

```
func main() {
    sigs := make(chan os.Signal, 1)
    signal.Notify(sigs, os.Interrupt, syscall.SIGINFO)
    go func() {
        for {
            sig := <-sigs
            switch sig {
            case os.Interrupt:
                fmt.Println("Caught:", sig)
            case syscall.SIGINFO:
                handleSignal(sig)
            }
        }
    }()
}
```

```
|           return  
|     }  
|   }()  
| }
```

This technique works as follows: first, you define a channel named `sigs` that helps you to pass data around. Then, you call `signal.Notify()` in order to state the signals that interest you. Next, you implement an **anonymous function** that runs as a goroutine in order to act when you receive any one of the signals you care about. You will have to wait for [Chapter 9, *Concurrency in Go – Goroutines, Channels, and Pipelines*](#), to learn more about goroutines and channels.

The last portion of the `handleTwo.go` program is as follows:

```
| for {  
|     fmt.Printf(".")  
|     time.Sleep(20 * time.Second)  
| }  
| }
```

The `time.Sleep()` call is used to prohibit the program from terminating, as it has no real work to do. In an actual application, there would be no need to use similar code.

As we need the process ID of a program in order to send signals to it using the `kill(1)` utility, we will first compile `handleTwo.go` and run the executable file instead of using `go run handleTwo.go`. Working with `handleTwo` will generate the following type of output:

```
$ go build handleTwo.go  
$ ls -l handleTwo  
-rwxr-xr-x 1 mtsouk  staff  2005200 Jan 18 07:49 handleTwo  
$ ./handleTwo  
.^CCaught: interrupt  
.Caught: interrupt  
handleSignal() Caught: information request  
.Killed: 9
```

Note that you will need an additional terminal in order to interact with `handleTwo.go` and obtain the preceding output. You will execute the following commands in that terminal:

```
$ ps ax | grep ./handleTwo | grep -v grep  
47988 s003 S+    0:00.00 ./handleTwo  
$ kill -s INT 47988  
$ kill -s INFO 47988  
$ kill -s USR1 47988  
$ kill -9 47988
```

The first command is used for finding the process ID of the `handleTwo` executable, while the remaining commands are used for sending the desired signals to that process. The `SIGUSR1` signal is ignored and it does not appear in the output.

The problem with `handleTwo.go` is that if it gets a signal that it is not programmed to handle, it will ignore it. Thus, in the next section, you will see a technique that uses a relatively different approach in order to handle signals in a more efficient way.

Handling all signals

In this subsection, you will learn how to handle all signals but respond only to the ones that really interest you. This is a much better and safer technique than the one presented in the previous subsection. The technique will be illustrated using the Go code of `handleAll.go`, which will be presented in four parts.

The first part of `handleAll.go` contains the following Go code:

```
package main

import (
    "fmt"
    "os"
    "os/signal"
    "syscall"
    "time"
)

func handle(signal os.Signal) {
    fmt.Println("Received:", signal)
}
```

The second code segment from `handleAll.go` is shown in the following Go code:

```
func main() {
    sigs := make(chan os.Signal, 1)
    signal.Notify(sigs)
```

So, all the magic happens due to the `signal.Notify(sigs)` statement. As no signals are specified, all incoming signals will be handled.



You are allowed to call `signal.Notify()` multiple times in the same program using different channels and the same signals. In that case, each relevant channel will receive a copy of the signals that it was programmed to handle.

The third code portion of the `handleAll.go` utility follows:

```
go func() {
    for {
        sig := <-sigs
        switch sig {
```

```

        case os.Interrupt:
    handle(sig)
    case syscall.SIGTERM:
        handle(sig)
        os.Exit(0)
    case syscall.SIGUSR2:
        fmt.Println("Handling syscall.SIGUSR2!")
    default:
        fmt.Println("Ignoring:", sig)
    }
}
}()

```

It is very convenient to use one of the signals in order to exit your program. This gives you the opportunity to do some housekeeping in your program when needed. In this case, the `syscall.SIGTERM` signal is used for that purpose. This does not prevent you from using `SIGKILL` to kill a program, though.

The remaining Go code for `handleAll.go` follows:

```

for {
    fmt.Printf(".")
    time.Sleep(30 * time.Second)
}

```

You still need to call `time.Sleep()` to prevent your program from terminating immediately.

Again, it would be better to build an executable file for `handleAll.go` using the `go build` tool. Executing `handleAll` and interacting with it from another terminal will generate the following type of output:

```

$ go build handleAll.go
$ ls -l handleAll
-rwxr-xr-x 1 mtsouk staff 2005216 Jan 18 08:25 handleAll
$ ./handleAll
.Ignoring: hangup
Handling syscall.SIGUSR2!
Ignoring: user defined signal 1
Received: interrupt
^CReceived: interrupt
Received: terminated

```

The commands issued from the second terminal are as follows:

```

$ ps ax | grep ./handleAll | grep -v grep
49902 s003 S+    0:00.00 ./handleAll
$ kill -s HUP 49902

```

```
| $ kill -s USR2 49902  
| $ kill -s USR1 49902  
| $ kill -s INT 49902  
| $ kill -s TERM 49902
```

Programming UNIX pipes in Go

According to the UNIX philosophy, UNIX command-line utilities should do one job and perform that job well. In practice, this means that instead of developing huge utilities that do lots of jobs, you should develop multiple smaller programs, which, when combined, should perform the desired job. The most common way for two or more UNIX command-line utilities to communicate is by using pipes. In a **UNIX pipe**, the output of a command-line utility becomes the input of another command-line utility. This process may involve more than two programs. The symbol that is used for UNIX pipes is the `|` character.

Pipes have two serious limitations: firstly, they usually communicate in one direction, and secondly, they can only be used between processes that have a common ancestor. The general idea behind the implementation of UNIX pipes is that if you do not have a file to process, you should wait to get your input from standard input.

Similarly, if you are not told to save your output to a file, you should write your output to standard output, either for the user to see it or for another program to process it.

In [Chapter 1](#), *Go and the Operating System*, you learned how to read from standard input and how to write to standard output. If you have doubts about these two operations, it would be a good time to review the Go code of `stdOUT.go` and `stdIN.go`.

Implementing the cat(1) utility in Go

In this section, you will see a Go version of the `cat(1)` utility. You will most likely be surprised by the length of the program. The source code of `cat.go` will be presented in three parts. The first part of `cat.go` follows:

```
package main

import (
    "bufio"
    "fmt"
    "io"
    "os"
)
```

The second code segment of `cat.go` contains the following Go code:

```
func printFile(filename string) error {
    f, err := os.Open(filename)
    if err != nil {
        return err
    }
    defer f.Close()
    scanner := bufio.NewScanner(f)
    for scanner.Scan() {
        io.WriteString(os.Stdout, scanner.Text())
        io.WriteString(os.Stdout, "\n")
    }
    return nil
}
```

In this part, you can see the implementation of a function whose purpose it is to print the contents of a file in the standard output.

The last part of `cat.go` is as follows:

```
func main() {
    filename := ""
    arguments := os.Args
    if len(arguments) == 1 {
        io.Copy(os.Stdout, os.Stdin)
        return
    }

    for i := 1; i < len(arguments); i++ {
```

```

    filename = arguments[i]
    err := printFile(filename)
    if err != nil {
        fmt.Println(err)
    }
}

```

The preceding code contains all of the magic of `cat.go`, because this is where you define how the program will behave. First of all, if you execute `cat.go` without any command-line arguments, then the program will just copy standard input to standard output as defined by the `io.Copy(os.Stdout, os.Stdin)` statement. However, if there are command-line arguments, then the program will process them all in the same order that they were given.

Executing `cat.go` will create the following type of output:

```

$ go run cat.go
Mastering Go!
Mastering Go!
1 2 3 4
1 2 3 4

```

However, things get really interesting if you execute `cat.go` using UNIX pipes:

```

$ go run cat.go /tmp/1.log /tmp/2.log | wc
 2367   44464  279292
$ go run cat.go /tmp/1.log /tmp/2.log | go run cat.go | wc
 2367   44464  279292

```

`cat.go` is also able to print multiple files on your screen:

```

$ go run cat.go 1.txt 1 1.txt
 2367   44464  279292
 2367   44464  279292
open 1: no such file or directory
 2367   44464  279292
 2367   44464  279292

```

Please note that if you try to execute `cat.go` as `go run cat.go cat.go` and expect that you will get the contents of `cat.go` on your screen, the process will fail and you will get the following error message instead:

```
| package main: case-insensitive file name collision: "cat.go" and "Cat.go"
```

The reason for this is that Go does not understand that the second `cat.go` should be used as a command-line argument to the `go run cat.go` command. Instead, `go run` tries to compile `cat.go` twice, which causes the error message. The solution to this problem is to execute `go build cat.go` first, and then use `cat.go` or any other Go source file as the argument to the generated binary executable file.

About `syscall.PtraceRegs`

You might have assumed that you are done dealing with the `syscall` standard Go package, but you are mistaken! In this section, we will work with `syscall.PtraceRegs`, which is a structure that holds information about the state of the registers.

You will now learn how to print the values of all of the following registers on your screen using the Go code of `ptraceRegs.go`, which will be presented in four parts. The star of the `ptraceRegs.go` utility is the `syscall.PtraceGetRegs()` function - there are also the `syscall.PtraceSetRegs()`, `syscall.PtraceAttach()`, `syscall.PtracePeekData()`, and `syscall.PtracePokeData()` functions that can help you to work with registers, but these functions will not be used in `ptraceRegs.go`.

The first part of the `ptraceRegs.go` utility follows:

```
package main

import (
    "fmt"
    "os"
    "os/exec"
    "syscall"
    "time"
)
```

The second code portion of `ptraceRegs.go` is shown in the following Go code:

```
func main() {
    var r syscall.PtraceRegs
    cmd := exec.Command(os.Args[1], os.Args[2:]...)

    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr
```

The last two statements redirect the standard output and standard error from the executed command to the UNIX standard output and standard error, respectively.

The third part of `ptraceRegs.go` contains the following Go code:

```
cmd.SysProcAttr = &syscall.SysProcAttr{Ptrace: true}
err := cmd.Start()
if err != nil {
    fmt.Println("Start:", err)
    return
}

err = cmd.Wait()
fmt.Printf("State: %v\n", err)
wpid := cmd.Process.Pid
```

In the preceding Go code, you call an external command, which is specified in the command-line arguments of the program, and you find its process ID, which will be used in the `syscall.PtraceGetRegs()` call. The `&syscall.SysProcAttr{Ptrace: true}` statement specifies that you want to use `ptrace` on the child process.

The last code segment of `ptraceRegs.go` follows:

```
err = syscall.PtraceGetRegs(wpid, &r)
if err != nil {
    fmt.Println("PtraceGetRegs:", err)
    return
}
fmt.Printf("Registers: %#v\n", r)
fmt.Printf("R15=%d, Gs=%d\n", r.R15, r.Gs)

time.Sleep(2 * time.Second)
```

Here, you call `syscall.PtraceGetRegs()` and you print the results that are stored in the `r` variable, which should be passed as a pointer.

Executing `ptraceRegs.go` on a macOS Mojave machine will generate the following output:

```
$ go run ptraceRegs.go
# command-line-arguments
./ptraceRegs.go:11:8: undefined: syscall.PtraceRegs
./ptraceRegs.go:14:9: undefined: syscall.PtraceGetRegs
```

This means that this program will not work on machines running macOS and Mac OS X.

Executing `ptraceRegs.go` on a Debian Linux machine will create the following output:

```
$ go version
go version go1.7.4 linux/amd64
$ go run ptraceRegs.go echo "Mastering Go!"
State: stop signal: trace/breakpoint trap
Registers: syscall.PtraceRegs{R15:0x0, R14:0x0, R13:0x0, R12:0x0, Rbp:0x0, Rbx:0x0, R11:0x0, R10:0x0, R9:0x0, R8:0x0, Rax:0x0,
R15=0, Gs=0
Mastering Go!
```

You can also find the list of registers on the documentation page of the `syscall` package.

Tracing system calls

This section will present a pretty advanced technique that uses the `syscall` package and allows you to monitor the system calls executed in a Go program.

The name of the Go utility is `traceSyscall.go`, and it is going to be presented in five code segments. The first part of `traceSyscall.go` follows:

```
package main

import (
    "bufio"
    "fmt"
    "os"
    "os/exec"
    "strings"
    "syscall"
)
var maxSyscalls = 0
const SYSCALLFILE = "SYSCALLS"
```

You will learn more about the purpose of the `SYSCALLFILE` variable in a short while.

The second code segment from `traceSyscall.go` is the following:

```
func main() {
    var SYSTEMCALLS []string
    f, err := os.Open(SYSCALLFILE)
    defer f.Close()
    if err != nil {
        fmt.Println(err)
        return
    }

    scanner := bufio.NewScanner(f)
    for scanner.Scan() {
        line := scanner.Text()
        line = strings.Replace(line, " ", "", -1)
        line = strings.Replace(line, "SYS_", "", -1)
        temp := strings.ToLower(strings.Split(line, "=")[0])
        SYSTEMCALLS = append(SYSTEMCALLS, temp)
        maxSyscalls++
    }
}
```

Note that the information of the `SYSCALLS` text file is taken from the documentation of the `syscall` package, and it associates each system call with a number, which is the internal Go representation of the system call. This file is mainly used for printing the names of the system calls used by the program that is being traced.

The format of the `SYSCALLS` text file is as follows:

```
SYS_READ = 0  
SYS_WRITE = 1  
SYS_OPEN = 2  
SYS_CLOSE = 3  
SYS_STAT = 4
```

After reading the text file, the program creates a slice named `SYSTEMCALLS` for storing that information.

The third part of `traceSyscall.go` is as follows:

```
COUNTER := make([]int, maxSyscalls)
var regs syscall.PtraceRegs
cmd := exec.Command(os.Args[1], os.Args[2:]...)

cmd.Stdin = os.Stdin
cmd.Stdout = os.Stdout
cmd.Stderr = os.Stderr
cmd.SysProcAttr = &syscall.SysProcAttr{Ptrace: true}

err = cmd.Start()
err = cmd.Wait()
if err != nil {
    fmt.Println("Wait:", err)
}

pid := cmd.Process.Pid
fmt.Println("Process ID:", pid)
```

The `COUNTER` slice stores the number of times each system call is found in the program that is being traced.

The fourth code segment of `traceSyscall.go` contains the following Go code:

```
    before := true
forCount := 0
for {
    if before {
        err := syscall.PtraceGetRegs(pid, &regs)
        if err != nil {
            break
```

```

        }
        if regs.Orig_rax > uint64(maxSyscalls) {
            fmt.Println("Unknown:", regs.Orig_rax)
            return
        }

        COUNTER[regs.Orig_rax]++
        forCount++
    }

    err = syscall.PtraceSyscall(pid, 0)
    if err != nil {
        fmt.Println("PtraceSyscall:", err)
        return
    }

    _, err = syscall.Wait4(pid, nil, 0, nil)
    if err != nil {
        fmt.Println("Wait4:", err)
        return
    }
    before = !before
}

```

The `syscall.PtraceSyscall()` function tells Go to continue the execution of the program that is being traced, but to stop when that program enters or exits a system call, which is exactly what we want! As each system call is traced before being called, right after it has finished its job, we use the `before` variable in order to count each system call only once.

The last part of `traceSyscall.go` follows:

```

    for i, x := range COUNTER {
        if x != 0 {
            fmt.Println(SYSTEMCALLS[i], "->", x)
        }
    }
    fmt.Println("Total System Calls:", forCount)
}

```

In this part, we print the contents of the `COUNTER` slice. The `SYSTEMCALLS` slice is used here for finding out the name of a system call when we know its numerical Go representation.

Executing `traceSyscall.go` on a macOS Mojave machine will create the following output:

```

$ go run traceSyscall.go
# command-line-arguments
./traceSyscall.go:36:11: undefined: syscall.PtraceRegs
./traceSyscall.go:57:11: undefined: syscall.PtraceGetRegs

```

```
| ./traceSyscall.go:70:9: undefined: syscall.PtraceSyscall
```

Once again, the `traceSyscall.go` utility will not run on macOS and Mac OS X.

Executing the same program on a Debian Linux machine will create the following output:

```
$ go run traceSyscall.go ls /tmp/
Wait: stop signal: trace/breakpoint trap
Process ID: 5657
go-build084836422 test.go upload_progress_cache
read -> 11
write -> 1
open -> 37
close -> 27
stat -> 1
fstat -> 25
mmap -> 39
mprotect -> 16
munmap -> 4
brk -> 3
rt_sigaction -> 2
rt_sigprocmask -> 1
ioctl -> 2
access -> 9
execve -> 1
getdents -> 2
getrlimit -> 1
statfs -> 2
arch_prctl -> 1
futex -> 1
set_tid_address -> 1
openat -> 1
set_robust_list -> 1
Total System Calls: 189
```

At the end of the program, `traceSyscall.go` prints the number of times each system call was called in the program. The correctness of `traceSyscall.go` is verified by the output of the `strace -c` utility:

```
$ strace -c ls /tmp
test.go upload_progress_cache
% time      seconds   usecs/call     calls    errors syscall
----- -----
 0.00  0.000000       0          11      read
 0.00  0.000000       0           1      write
 0.00  0.000000       0          37      13 open
 0.00  0.000000       0          27      close
 0.00  0.000000       0           1      stat
 0.00  0.000000       0          25      fstat
 0.00  0.000000       0          39      mmap
 0.00  0.000000       0          16      mprotect
 0.00  0.000000       0            4      munmap
```

0.00	0.000000	0	3	brk
0.00	0.000000	0	2	rt_sigaction
0.00	0.000000	0	1	rt_sigprocmask
0.00	0.000000	0	2	ioctl
0.00	0.000000	0	9	9 access
0.00	0.000000	0	1	execve
0.00	0.000000	0	2	getdents
0.00	0.000000	0	1	getrlimit
0.00	0.000000	0	2	2 statfs
0.00	0.000000	0	1	arch_prctl
0.00	0.000000	0	1	futex
0.00	0.000000	0	1	set_tid_address
0.00	0.000000	0	1	openat
0.00	0.000000	0	1	set_robust_list
<hr/>				
100.00	0.000000	189	24	total

User ID and group ID

In this section, you will learn how to find the user ID of the current user, as well as the group IDs to which the current user belongs. Both the user ID and group IDs are positive integers kept in UNIX system files.

The name of the utility is `ids.go`, and it will be presented in two parts. The first part of the utility follows:

```
package main

import (
    "fmt"
    "os"
    "os/user"
)

func main() {
    fmt.Println("User id:", os.Getuid())
```

Finding the user ID of the current user is as simple as calling the `os.Getuid()` function.

The second part of `ids.go` is as follows:

```
var u *user.User
u, _ = user.Current()
fmt.Print("Group ids: ")
groupIDs, _ := u.GroupIds()
for _, i := range groupIDs {
    fmt.Print(i, " ")
}
fmt.Println()
```

On the other hand, finding the group IDs to which a user belongs is a much trickier task.

Executing `ids.go` will generate the following type of output:

```
$ go run ids.go
User id: 501
Group ids: 20 701 12 61 79 80 81 98 33 100 204 250 395 398 399
```

The Docker API and Go

If you work with Docker, you will find this section particularly handy as it will teach you how to communicate with Docker using Go and the **Docker API**.

The `dockerAPI.go` utility, which will be presented in four parts, implements the `docker ps` and the `docker image ls` commands. The first command lists all running containers, whereas the second command lists all available images on the local machine.

The first part of `dockerAPI.go` is as follows:

```
package main

import (
    "fmt"
    "github.com/docker/docker/api/types"
    "github.com/docker/docker/client"
    "golang.org/x/net/context"
)
```

As `dockerAPI.go` requires lots of external packages, it would be a good idea to execute it using Go modules; therefore, execute `export GO111MODULE=on` before running `dockerAPI.go` for the first time. This will also save you from having to manually download all of the required packages.

The second part of `dockerAPI.go` contains the following Go code:

```
func listContainers() error {
    cli, err := client.NewEnvClient()
    if err != nil {
        return (err)
    }

    containers, err := cli.ContainerList(context.Background(), types.ContainerListOptions{})
    if err != nil {
        return (err)
    }

    for _, container := range containers {
        fmt.Println("Images:", container.Image, "with ID:", container.ID)
    }
    return nil
}
```

The definition of the `types.Container` (<https://godoc.org/github.com/docker/docker/api/types#Container>) structure, which is returned by `ContainerList()`, is as follows:

```
type Container struct {
    ID      string `json:"Id"`
    Names   []string
    Image   string
    ImageID string
    Command string
    Created int64
    Ports   []Port
    SizeRw  int64 `json:",omitempty"`
    SizeRootFs int64 `json:",omitempty"`
    Labels  map[string]string
    State   string
    Status  string
    HostConfig struct {
```

```

        NetworkMode string `json:",omitempty"`
    }
    NetworkSettings *SummaryNetworkSettings
    Mounts          []MountPoint
}

```

Should you wish to find any other information about the list of running Docker images (containers), you should make use of the other fields of the `types.Container` structure.

The third part of `dockerAPI.go` is as follows:

```

func listImages() error {
    cli, err := client.NewEnvClient()
    if err != nil {
        return (err)
    }

    images, err := cli.ImageList(context.Background(),
        types.ImageListOptions{})
    if err != nil {
        return (err)
    }

    for _, image := range images {
        fmt.Printf("Images %s with size %d\n", image.RepoTags,
            image.Size)
    }
    return nil
}

```

The definition of the `types.ImageSummary` (<https://godoc.org/github.com/docker/docker/api/types#ImageSummary>) structure, which is the data type of the slice that is returned by `ImageList()`, is as follows:

```

type ImageSummary struct {
    Containers int64 `json:"Containers"`
    Created int64 `json:"Created"`
    ID string `json:"Id"`
    Labels map[string]string `json:"Labels"`
    ParentID string `json:"ParentId"`
    RepoDigests []string `json:"RepoDigests"`
    RepoTags []string `json:"RepoTags"`
    SharedSize int64 `json:"SharedSize"`
    Size int64 `json:"Size"`
    VirtualSize int64 `json:"VirtualSize"`
}

```

The last part of `dockerAPI.go` is the following:

```

func main() {
    fmt.Println("The available images are:")
    err := listImages()
    if err != nil {
        fmt.Println(err)
    }

    fmt.Println("The running Containers are:")
    err = listContainers()
    if err != nil {
        fmt.Println(err)
    }
}

```

Executing `dockerAPI.go` on my macOS Mojave machine will generate the following kind of output:

```

$ go run dockerAPI.go
The available images are:
Images [golang:1.12] with size 772436547
Images [landoop/kafka-lenses-dev:latest] with size 1379088343
Images [confluentinc/cp-kafka:latest] with size 568503378

```

```
| Images [landoop/fast-data-dev:latest] with size 1052076831
| The running Containers are:
| Images: landoop/kafka-lenses-dev with ID: 90e1caaab43297810341290137186425878ef5891c787f6707c03be617862db5
```

If Docker is not available or if it is not running, you will get the following error message:

```
| $ go run dockerAPI.go
| The available images are:
| Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker daemon running?
| The running Containers are:
| Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker daemon running?
```

Additional resources

You will find the following web links very useful:

- Read the documentation page of the `io` package, which can be found at <https://golang.org/pkg/io/>.
- Hear from Dave Cheney on error handling: <https://dave.cheney.net/2016/04/27/dont-just-check-errors-handle-them-gracefully>.
- You can learn more about the **Glot** plotting library by visiting its official web page at <https://github.com/Arafatk/glot>.
- You can find more examples of the use of the Docker API in Go, Python, and HTTP at <https://docs.docker.com/develop/sdk/examples/>.
- You can learn more about the `encoding/binary` standard package by visiting <https://golang.org/pkg/encoding/binary/>.
- Check out the documentation page of the `encoding/gob` package, which can be found at <https://golang.org/pkg/encoding/gob/>.
- You can also watch <https://www.youtube.com/watch?v=JRFNIKUROPE> and <https://www.youtube.com/watch?v=w8nFRoFJ6EQ>.
- You can learn about **Endianness** in many places, including <https://en.wikipedia.org/wiki/Endianness>.
- Visit the documentation page of the `flag` package, which can be found at <https://golang.org/pkg/flag/>.

Exercises

- Write a Go program that takes three arguments: the name of a text file, and two strings. This utility should then replace every occurrence of the first string in the file with the second string. For reasons of security, the final output will be printed on the screen, which means that the original text file will remain intact.
- Use the `encoding/gob` package to serialize and deserialize a Go map, as well as a slice of structures.
- Create a Go program that handles any three signals you choose.
- Create a utility in Go that replaces all tab characters found in a text file with a given number of spaces, specified as a command-line parameter to the program. Once again, the output will be printed on the screen.
- Develop a utility that reads a text file line by line and removes the space characters from each line using the `strings.TrimSpace()` function.
- Modify `kvSaveLoad.go` in order to support a single command-line argument, which is the filename that you will use both to load and save your data.
- Can you create a Go version of the `wc(1)` utility? Look at the manual page of `wc(1)` to find out about the command-line options that it supports.
- Can you write a program that uses **Glot** to plot a function?
- Modify `traceSyscall.go` in order to display each system call at the time it is being traced.
- Modify `cat.go` just to do `io.Copy(os.Stdout, f)` in order to copy the contents of a file straight out, instead of scanning it all.
- Use the Docker API to write a utility that terminates all containers that begin with a given string.
- The `cobra` package also supports subcommands, which are commands associated with specific commands like `go run main.go command` list. Try to implement a utility with subcommands.
- You can also use `bufio.NewScanner()` and `bufio.ScanWords` to read a line word by word. Find out how and create a new version of the `byWord.go` utility.

Summary

This invaluable chapter talked about many interesting topics, including reading files, writing to files, using the Docker API, and using the `flag`, `cobra`, and `viper` packages. Nevertheless, there are many more topics related to systems programming not mentioned in this chapter, such as working with directories; copying, deleting, and renaming files; dealing with UNIX users, groups, and UNIX processes; changing UNIX file permissions; generating **sparse files**; file locking and creating; and using and rotating your own log files, as well as the information found in the structure returned by the `os.Stat()` call.

At the end of this chapter, I presented two advanced utilities written in Go. The first one allowed you to inspect the state of the registers, while the second one showed you a technique that allows you to trace the system calls of any program.

The next chapter will talk about goroutines, channels, and pipelines, which are unique and powerful Go features.

Concurrency in Go – Goroutines, Channels, and Pipelines

The previous chapter discussed systems programming in Go, including the Go functions and techniques that allow you to communicate with your operating system. Two of the areas of systems programming that were not covered in the previous chapter are concurrent programming and how to create and manage multiple threads. Both of these topics will be addressed in this chapter and the next one.

Go offers its own unique and innovative way of achieving concurrency, which comes in the form of **goroutines** and **channels**. Goroutines are the smallest Go entities that can be executed on their own in a Go program. Channels can get data from goroutines in a concurrent and efficient way. This allows goroutines to have a point of reference and they can communicate with each other. Everything in Go is executed using goroutines, which makes perfect sense since Go is a concurrent programming language by design. Therefore, when a Go program starts its execution, its single goroutine calls the `main()` function, which starts the actual program execution.

The contents and the code of this chapter will be pretty simple, and you should have no problem following and understanding them. I left the more advanced parts of goroutines and channels for [Chapter 10, *Concurrency in Go – Advanced Topics*](#).

In this chapter, you will learn about the following topics:

- The differences between processes, threads, and goroutines
- The Go scheduler
- Concurrency versus parallelism
- The concurrency models of Erlang and Rust
- Creating goroutines

- Creating channels
- Reading or receiving data from a channel
- Writing or sending data to a channel
- Creating pipelines
- Waiting for your goroutines to finish

About processes, threads, and goroutines

A **process** is an execution environment that contains instructions, user data, and system data parts, as well as other types of resources that are obtained during runtime, whereas a **program** is a file that contains instructions and data that are used for initializing the instruction and user-data parts of a process.

A **thread** is a smaller and lighter entity than a process or a program. Threads are created by processes and have their own flow of control and stack. A quick and simplistic way to differentiate a thread from a process is to consider a process as the running binary file and a thread as a subset of a process.

A **goroutine** is the minimum Go entity that can be executed concurrently. The use of the word "minimum" is very important here, as goroutines are not autonomous entities like UNIX processes – goroutines live in UNIX threads that live in UNIX processes. The main advantage of goroutines is that they are extremely lightweight and running thousands or hundreds of thousands of them on a single machine is not a problem.

The good thing is that goroutines are lighter than threads, which, in turn, are lighter than processes. In practice, this means that a process can have multiple threads as well as lots of goroutines, whereas a goroutine needs the environment of a process in order to exist. So, in order to create a goroutine, you will need to have a process with at least one thread – UNIX takes care of the process and thread management, while Go and the developer need to take care of the goroutines.

Now that you know the basics of processes, programs, threads, and goroutines, let us talk a little bit about the **Go scheduler**.

The Go scheduler

The UNIX kernel scheduler is responsible for the execution of the threads of a program. On the other hand, the Go runtime has its own scheduler, which is responsible for the execution of the goroutines using a technique known as **m:n scheduling**, where m goroutines are executed using n operating system threads using multiplexing. The Go scheduler is the Go component responsible for the way and the order in which the goroutines of a Go program get executed. This makes the Go scheduler a really important part of the Go programming language, as everything in a Go program is executed as a goroutine.

Be aware that as the Go scheduler only deals with the goroutines of a single program, its operation is much simpler, cheaper, and faster than the operation of the kernel scheduler.

[chapter 10](#), *Concurrency in Go – Advanced Topics*, will talk about the way the Go scheduler operates in much more detail.

Concurrency and parallelism

It is a very common misconception that **concurrency** is the same thing as **parallelism** – this is just not true! Parallelism is the simultaneous execution of multiple entities of some kind, whereas concurrency is a way of structuring your components so that they can be executed independently when possible.

It is only when you build software components concurrently that you can safely execute them in parallel, when and if your operating system and your hardware permit it. The Erlang programming language did this a long time ago – long before CPUs had multiple cores and computers had lots of RAM.

In a valid concurrent design, adding concurrent entities makes the whole system run faster because more things can be executed in parallel. So, the desired parallelism comes from a better concurrent expression and implementation of the problem. The developer is responsible for taking concurrency into account during the design phase of a system and will benefit from a potential parallel execution of the components of the system. So, the developer should not think about parallelism but about breaking things into independent components that solve the initial problem when combined.

Even if you cannot run your functions in parallel on a UNIX machine, a valid concurrent design will still improve the design and the maintainability of your programs. In other words, concurrency is better than parallelism!

Goroutines

You can define, create, and execute a new goroutine using the `go` keyword followed by a function name or the full definition of an **anonymous function**. The `go` keyword makes the function call return immediately, while the function starts running in the background as a goroutine and the rest of the program continues its execution.

However, as you will see in a moment, you cannot control or make any assumptions about the order in which your goroutines are going to be executed because this depends on the scheduler of the operating system, the Go scheduler, and the load of the operating system.

Creating a goroutine

In this subsection, you will learn two ways of creating goroutines. The first one is by using regular functions, while the second method is by using anonymous functions – these two ways are equivalent.

The name of the program covered in this section is `simple.go`, and it is presented in three parts.

The first part of `simple.go` is the following Go code:

```
package main

import (
    "fmt"
    "time"
)

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

Apart from the `import` block, the preceding code defines a function named `function()` that will be used in a short while.

The function name `function()` is nothing special – you can give it any valid function name you want.

The second part of `simple.go` is as follows:

```
func main() {
    go function()
```

The preceding code starts by executing `function()` as a goroutine. After that, the program continues its execution, while `function()` begins to run in the background.

The last code portion of `simple.go` is shown in the following Go code:

```
go func() {
    for i := 10; i < 20; i++ {
        fmt.Println(i, " ")
    }
}()

time.Sleep(1 * time.Second)
fmt.Println()
```

With this code, you create a goroutine using an anonymous function. This method works best for relatively small functions. However, if you have lots of code, it is considered a better practice to create a regular function and execute it using the `go` keyword.

As you will see in the next section, you can create multiple goroutines any way you desire, including using a `for` loop.

Executing `simple.go` three times will generate the following type of output:

```
$ go run simple.go
10 11 12 13 14 15 16 17 18 19 0123456789
$ go run simple.go
10 11 12 13 14 15 16 0117 2345678918 19
$ go run simple.go
10 11 12 012345678913 14 15 16 17 18 19
```

Although what you really want from your programs is to generate the same output for the same input, the output you get from `simple.go` is not always the same. The preceding output supports the fact that you cannot control the order in which your goroutines will be executed without taking extra care. This means writing extra code specifically for this to occur. In [Chapter 10, *Concurrency in Go – Advanced Topics*](#), you will learn how to control the order in which your goroutines are executed, as well as how to print the results of one goroutine before printing the results of the following one.

Creating multiple goroutines

In this subsection, you will learn how to create a variable number of goroutines. The program reviewed in this section is called `create.go`. It is going to be presented in four parts, and it will allow you to create a dynamic number of goroutines. The number of goroutines will be given as a command-line argument to the program, which uses the `flag` package to process its command-line argument.

The first code part of `create.go` is as follows:

```
package main  
import (  
    "flag"  
    "fmt"  
    "time"  
)
```

The second code segment from `create.go` contains the following Go code:

```
func main() {  
    n := flag.Int("n", 10, "Number of goroutines")  
    flag.Parse()  
  
    count := *n  
    fmt.Printf("Going to create %d goroutines.\n", count)
```

The preceding code reads the value of the `n` command-line option, which determines the number of goroutines that are going to be created. If there is no `n` command-line option, the value of the `n` variable will be `10`.

The third code portion of `create.go` is as follows:

```
    for i := 0; i < count; i++ {  
        go func(x int) {  
            fmt.Printf("%d ", x)  
        }(i)  
    }
```

A `for` loop is used to create the desired number of goroutines. Once again, you should remember that you cannot make any assumptions about the order in which they are going to be created and executed.

The last part of the Go code from `create.go` is the following:

```
    time.Sleep(time.Second)  
    fmt.Println("\nExiting...")  
}
```

The purpose of the `time.Sleep()` statement is to give the goroutines enough time to finish their jobs so that their output can be seen on the screen. In a real program, you will not need a `time.Sleep()` statement, as you will want to finish as soon as possible and, moreover, you will learn a better technique of making your program wait for the various goroutines to finish before the `main()` function returns.

Executing `create.go` multiple times will generate the following type of output:

```
$ go run create.go -n 100  
Going to create 100 goroutines.  
5 3 2 4 19 9 0 1 7 11 10 12 13 14 15 31 16 20 17 22 8 18 28 29 21 52 30 45 25 24 49 38 41 46 6 56 57 54 23 26 53 27 59 47 69 61  
Exiting...  
$ go run create.go -n 100  
Going to create 100 goroutines.  
2 5 3 16 6 7 8 9 1 22 10 12 13 17 11 18 15 14 19 20 31 23 26 21 29 24 30 25 37 32 36 38 35 33 45 41 43 42 40 39 34 44 48 46 47  
Exiting...
```

Once again, you can see that the output is non-deterministic and messy in the sense that you will have to search the output to find what you are looking for. Additionally, if you do not use a suitable delay in the `time.Sleep()` call, you

will not be able to see the output of the goroutines. `time.Second` might be fine for now, but this kind of code can cause nasty and unpredictable bugs further down the road.

In the next section, you will learn how to give your goroutines enough time to finish what they are doing before your program ends, without the need to call `time.Sleep()`.

Waiting for your goroutines to finish

This section will present a way to prevent the `main()` function from ending while it is waiting for its goroutines to finish using the `sync` package. The logic of the `syncGo.go` program will be based on `create.go`, which was presented in the previous section.

The first part of `syncGo.go` is as follows:

```
package main  
import (  
    "flag"  
    "fmt"  
    "sync"  
)
```

As you can see, there is no need to import and use the `time` package, as we will use the functionality of the `sync` package and wait for as long as necessary for all the goroutines to end.



In [Chapter 10](#), Concurrency in Go – Advanced Topics, you will study two techniques for timing out goroutines when they are taking longer than desired.

The second code segment of `syncGo.go` is shown in the following Go code:

```
func main() {  
    n := flag.Int("n", 20, "Number of goroutines")  
    flag.Parse()  
    count := *n  
    fmt.Printf("Going to create %d goroutines.\n", count)  
  
    var waitGroup sync.WaitGroup
```

In the preceding Go code, you define a `sync.WaitGroup` variable. If you look at the source code of the `sync` Go package, and more specifically at the `waitgroup.go` file that is located inside the `sync` directory, you will see that the `sync.WaitGroup` type is nothing more than a structure with three fields:

```
type WaitGroup struct {  
    noCopy  
    state1 [12]byte  
    sema    uint32  
}
```

The output of `syncGo.go` will reveal more information about the way `sync.WaitGroup` variables work. The number of goroutines that belong to a `sync.WaitGroup` group is defined by one or multiple calls to the `sync.Add()` function.

The third part of `syncGo.go` contains the following Go code:

```
    fmt.Printf("%#v\n", waitGroup)  
    for i := 0; i < count; i++ {  
        waitGroup.Add(1)  
        go func(x int) {  
            defer waitGroup.Done()  
            fmt.Printf("%d ", x)  
        }(i)  
    }
```

Here, you create the desired number of goroutines using a `for` loop. (You could use multiple sequential Go statements instead.)

Each call to `sync.Add()` increases a counter in a `sync.WaitGroup` variable. Notice that it is really important to call `sync.Add(1)` before the `go` statement in order to prevent any **race conditions**. When each goroutine finishes its job, the `sync.Done()` function will be executed, which will decrease the same counter.

The last code portion of `syncGo.go` is as follows:

```
    fmt.Printf("%#v\n", waitGroup)
    waitGroup.Wait()
    fmt.Println("\nExiting...")  
}
```

The `sync.Wait()` call blocks until the counter in the relevant `sync.WaitGroup` variable is zero, giving your goroutines time to finish.

Executing `syncGo.go` will create the following type of output:

```
$ go run syncGo.go
Going to create 20 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0}, sema:0x0}
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x14, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
19 7 8 9 10 11 12 13 14 15 16 17 0 1 2 5 18 4 6 3
Exiting...
$ go run syncGo.go -n 30
Going to create 30 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0}, sema:0x0}
1 0 4 5 17 7 8 9 10 11 12 13 2 sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x17, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
29 15 6 27 24 25 16 22 14 23 18 26 3 19 20 28 21
Exiting...
$ go run syncGo.go -n 30
Going to create 30 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0}, sema:0x0}
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x1e, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
29 1 7 8 2 9 10 11 12 4 13 15 0 6 5 22 25 23 16 28 26 20 19 24 21 14 3 17 18 27
Exiting...
```

The output of `syncGo.go` still varies from execution to execution, especially if you are dealing with a large number of goroutines. Most of the time, this is acceptable; however, there are times when this is not the desired behavior. Additionally, when the number of goroutines is 30, some of the goroutines have finished their job before the second `fmt.Printf("%#v\n", waitGroup)` statement. Finally, notice that one of the elements of the `state1` field in `sync.WaitGroup` is the one that holds the counter, which increases and decreases according to the `sync.Add()` and `sync.Done()` calls.

What if the number of Add() and Done() calls do not agree?

When the number of `sync.Add()` calls and `sync.Done()` calls are equal, everything will be fine in your programs. However, this section will tell you what will happen when these two numbers do not agree with each other.

If you have executed more `sync.Add()` calls than `sync.Done()` calls, in this case by adding a `waitGroup.Add(1)` statement before the first `fmt.Printf("%#v\n", waitGroup)` statement of the `syncGo.go` program, then the output of the `go run` command will be similar to the following:

```
$ go run syncGo.go
Going to create 20 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0, 0x1, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x15, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
19 10 11 12 13 17 18 8 5 4 6 14 1 0 7 3 2 15 9 16 fatal error: all goroutines are asleep - deadlock!

goroutine 1 [semacquire]:
sync.runtime_Semacquire(0xc4200120bc)
    /usr/local/Cellar/go/1.9.3/libexec/src/runtime/sema.go:56 +0x39
sync.(*WaitGroup).Wait(0xc4200120b0)
    /usr/local/Cellar/go/1.9.3/libexec/src/sync/waitgroup.go:131 +0x72
main.main()
    /Users/mtsouk/Desktop/masterGo/ch/ch9/code/syncGo.go:28 +0x2d7
exit status 2
```

The error message is pretty clear: `fatal error: all goroutines are asleep - deadlock!` This happened because you told your program to wait for $n+1$ goroutines by calling the `sync.Add(1)` function $n+1$ times while only n `sync.Done()` statements were executed by your n goroutines. As a result, the `sync.Wait()` call will wait indefinitely for one or more calls to `sync.Done()` without any luck, which is obviously a deadlock situation.

If you have made fewer `sync.Add()` calls than `sync.Done()` calls, which can be emulated by adding a `waitGroup.Done()` statement after the `for` loop of the `syncGo.go` program, then the `go run` output will be similar to the following:

```
$ go run syncGo.go
Going to create 20 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0}, sema:0x0}
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x12, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
19 6 1 2 9 7 8 15 13 0 14 16 17 3 11 4 5 12 18 10 panic: sync: negative WaitGroup counter

goroutine 22 [running]:
sync.(*WaitGroup).Add(0xc4200120b0, 0xffffffffffffffff)
    /usr/local/Cellar/go/1.9.3/libexec/src/sync/waitgroup.go:75 +0x134
sync.(*WaitGroup).Done(0xc4200120b0)
    /usr/local/Cellar/go/1.9.3/libexec/src/sync/waitgroup.go:100 +0x34
main.main.func1(0xc4200120b0, 0x11)
    /Users/mtsouk/Desktop/masterGo/ch/ch9/code/syncGo.go:25 +0xd8
created by main.main
    /Users/mtsouk/Desktop/masterGo/ch/ch9/code/syncGo.go:21 +0x206
exit status 2
```

Once again, the root of the problem is stated pretty clearly: `panic: sync: negative WaitGroup counter`.

Although the error messages are very descriptive in both cases and will help you to solve the real problem, you should be very careful with the number of `sync.Add()` and `sync.Done()` calls that you put into your programs. Additionally, notice that in the second error case (`panic: sync: negative WaitGroup counter`), the problem might not appear all of the time.

Channels

A channel is a communication mechanism that allows goroutines to exchange data, among other things.

However, there are some specific rules. Firstly, each channel allows the exchange of a particular data type, which is also called the **element type** of the channel, and secondly, for a channel to operate properly, you will need someone to receive what is sent via the channel. You should declare a new channel using the `chan` keyword, and you can close a channel using the `close()` function.

Finally, a very important detail: when you are using a channel as a function parameter, you can specify its direction; that is, whether it is going to be used for sending or receiving. In my opinion, if you know the purpose of a channel in advance, you should use this capability because it will make your programs more robust, as well as safer. You will not be able to send data accidentally to a channel from which you should only receive data, or receive data from a channel to which you should only be sending data. As a result, if you declare that a channel function parameter will be used for reading only and you try to write to it, you will get an error message that will most likely save you from nasty bugs. We will talk about this later on in this chapter.



Although you will learn many things about channels in this chapter, you will have to wait for [chapter 10](#), Concurrency in Go – Advanced Topics, to fully understand the power and flexibility that channels offer to the Go developer.

Writing to a channel

The code in this subsection will teach you how to write to a channel. Writing the value `x` to channel `c` is as easy as writing `c <- x`. The arrow shows the direction of the value, and you will have no problem with this statement as long as both `x` and `c` have the same type. The example code in this section is saved in `writeCh.go`, and it will be presented in three parts.

The first code segment from `writeCh.go` is as follows:

```
package main

import (
    "fmt"
    "time"
)

func writeToChannel(c chan int, x int) {
    fmt.Println(x)
    c <- x
    close(c)

    fmt.Println(x)
}
```

The `chan` keyword is used for declaring that the `c` function parameter will be a channel, and it should be followed by the type of the channel (`int`). The `c <- x` statement allows you to write the value `x` to channel `c`, and the `close()` function closes the channel; that is, it makes writing to it impossible.

The second part of `writeCh.go` contains the following Go code:

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

In the preceding code, you can find the definition of a channel variable, which is named `c`, and for the first time in this chapter you are using the `make()` function as well as the `chan` keyword. All channels have a type associated with them, which in this case is `int`.

The remaining code from `writeCh.go` is as follows:

```
|     go writeToChannel(c, 10)
|     time.Sleep(1 * time.Second)
| }
```

Here, you execute the `writeToChannel()` function as a goroutine and call `time.Sleep()` in order to give enough time to the `writeToChannel()` function to execute.

Executing `writeCh.go` will create the following output:

```
$ go run writeCh.go
10
```

The strange thing here is that the `writeToChannel()` function prints the given value only once. The cause of this unexpected output is that the second `fmt.Println(x)` statement is never executed. The reason for this is pretty simple once you understand how channels work: the `c <- x` statement is blocking the execution of the rest of the `writeToChannel()` function because nobody is reading what was written to the `c` channel. Therefore, when the `time.Sleep(1 * time.Second)` statement finishes, the program terminates without waiting for `writeToChannel()`.

The next section will illustrate how to read data from a channel.

Reading from a channel

In this subsection, you will learn how to read from a channel. You can read a single value from a channel named `c` by executing `<-c`. In this case, the direction is from the channel to the outer world.

The name of the program that will be used to help you understand how to read from a channel is `readCh.go`, and it will be presented in three parts.

The first code segment from `readCh.go` is shown in the following Go code:

```
package main

import (
    "fmt"
    "time"
)

func writeToChannel(c chan int, x int) {
    fmt.Println("1", x)
    c <- x
    close(c)
    fmt.Println("2", x)
}
```

The implementation of the `writeToChannel()` function is the same as before.

The second part of `readCh.go` follows:

```
func main() {
    c := make(chan int)
    go writeToChannel(c, 10)
    time.Sleep(1 * time.Second)
    fmt.Println("Read:", <-c)
    time.Sleep(1 * time.Second)
```

In the preceding code, you read from channel `c` using the `<-c` notation. If you want to store that value to a variable named `k` instead of just printing it, you can use a `k := <-c` statement. The second `time.Sleep(1 * time.Second)` statement gives you the time to read from the channel.

The last code portion of `readCh.go` contains the following Go code:

```
|_, ok := <-c
|if ok {
|    fmt.Println("Channel is open!")
|} else {
|    fmt.Println("Channel is closed!")
|}
```

In the preceding code, you can see a technique for determining whether a given channel is open or not. The current Go code works fine when the channel is closed; however, if the channel was open, the Go code presented here would have discarded the read value of the channel because of the use of the `_` character in the `_, ok := <-c` statement. Use a proper variable name instead of `_` if you also want to store the value found in the channel in case it is open.

Executing `readCh.go` will generate the following output:

```
$ go run readCh.go
1 10
Read: 10
2 10
Channel is closed!
$ go run readCh.go
1 10
2 10
Read: 10
Channel is closed!
```

Although the output is still not deterministic, both the `fmt.Println(x)` statements of the `writeToChannel()` function are executed because the channel is unblocked when you read from it.

Receiving from a closed channel

In this subsection, you will learn what happens when you try to read from a closed channel using the Go code found in `readClose.go`, which is going to be presented in two parts.

The first part of `readClose.go` is as follows:

```
package main

import (
    "fmt"
)

func main() {
    willClose := make(chan int, 10)

    willClose <- -1
    willClose <- 0
    willClose <- 2

    <-willClose
    <-willClose
    <-willClose
```

In this part of the program, we create a new `int` channel named `willClose`, we write data to it, and we read all that data without doing anything with it.

The second part of `readClose.go` contains the following code:

```
    close(willClose)
    read := <-willClose
    fmt.Println(read)
}
```

In this part, we close the `willClose` channel and we try to read from the `willClose` channel, which we emptied in the previous part.

Executing `readClose.go` will generate the following output:

```
$ go run readClose.go
0
```

This means that reading from a closed channel returns the zero value of its data type, which in this case is `0`.

Channels as function parameters

Although neither `readCh.go` nor `writeCh.go` used this feature, Go allows you to specify the direction of a channel when used as a function parameter; that is, whether it will be used for reading or writing. These two types of channels are called unidirectional channels, whereas, by default, channels are bidirectional.

Examine the Go code of the following two functions:

```
| func f1(c chan int, x int) {
|   fmt.Println(x)
|   c <- x
| }
| func f2(c chan<- int, x int) {
|   fmt.Println(x)
|   c <- x
| }
```

Although both functions implement the same functionality, their definitions are slightly different. The difference is created by the `<-` symbol found on the right of the `chan` keyword in the definition of the `f2()` function. This denotes that the `c` channel can be used for writing only. If the code of a Go function attempts to read from a write-only channel (**send-only channel**) parameter, the Go compiler will generate the following kind of error message:

```
| # command-line-arguments
| a.go:19:11: invalid operation: range in (receive from send-only type chan<- int)
```

Similarly, you can have the following function definitions:

```
| func f1(out chan<- int64, in <-chan int64) {
|   fmt.Println(x)
|   c <- x
| }
| func f2(out chan int64, in chan int64) {
|   fmt.Println(x)
|   c <- x
| }
```

The definition of `f2()` combines a read-only channel named `in` with a write-only channel named `out`. If you accidentally try to write and close a read-only channel (**receive-only channel**) parameter of a function, you will get the following kind of error message:

```
# command-line-arguments
a.go:13:7: invalid operation: out <- i (send to receive-only type <-chan int)
a.go:15:7: invalid operation: close(out) (cannot close receive-only channel)
```

Pipelines

A **pipeline** is a virtual method for connecting goroutines and channels so that the output of one goroutine becomes the input of another goroutine using channels to transfer your data.

One of the benefits that you get from using pipelines is that there is a constant data flow in your program, as no goroutine and channel have to wait for everything to be completed in order to start their execution. Additionally, you use fewer variables and therefore less memory space because you do not have to save everything as a variable. Finally, the use of pipelines simplifies the design of the program and improves its maintainability.

Pipelines are going to be illustrated using the code of `pipeline.go`. This program will be presented in six parts. The task performed by the `pipeline.go` program is to generate random numbers in a given range and stop when any number in the random sequence appears a second time. However, before terminating, the program will print the sum of all random numbers that appeared up to the point where the first random number appeared a second time. You will need three functions to connect the channels of the program. The logic of the program is found in these three functions, but the data flows in the channels of the pipeline.

This program will have two channels. The first channel (channel A) will be used to get the random numbers from the first function and send them to the second function. The second channel (channel B) will be used by the second function to send the acceptable random numbers to the third function. The third function will be responsible for getting the data from channel B, calculating it, and presenting the results.

The first code segment of `pipeline.go` contains the following Go code:

```
| package main
```

```

import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "time"
)
var CLOSEA = false
var DATA = make(map[int]bool)

```

As the `second()` function will need a way to tell the `first()` function to close the first channel, I will use a global variable named `CLOSEA` for that. The `CLOSEA` variable is only checked by the `first()` function, and it can only be altered by the `second()` function.

The second part of `pipeline.go` is shown in the following Go code:

```

func random(min, max int) int {
    return rand.Intn(max-min) + min
}

func first(min, max int, out chan<- int) {
    for {
        if CLOSEA {
            close(out)
            return
        }
        out <- random(min, max)
    }
}

```

The preceding code presents the implementation of two functions named `random()` and `first()`. You are already familiar with the `random()` function that generates random numbers in a given range. However, the `first()` function is really interesting as it keeps running using a `for` loop until a Boolean variable (`CLOSEA`) becomes `true`. In that case, it will close its `out` channel.

The third code segment of `pipeline.go` is as follows:

```

func second(out chan<- int, in <-chan int) {
    for x := range in {
        fmt.Println(x, " ")
        _, ok := DATA[x]
        if ok {
            CLOSEA = true
        } else {
            DATA[x] = true
            out <- x
        }
    }
}

```

```

    }
    fmt.Println()
    close(out)
}

```

The `second()` function receives data from the `in` channel and keeps sending it to the `out` channel. However, as soon as the `second()` function finds a random number that already exists in the `DATA` map, it sets the `CLOSEA` global variable to `true` and stops sending any more numbers to the `out` channel. After that, it closes the `out` channel.



range loops over channels and will automatically exit when the channel is closed.

The fourth code portion of `pipeline.go` is shown in the following Go code:

```

func third(in <-chan int) {
    var sum int
    sum = 0
    for x2 := range in {
        sum = sum + x2
    }
    fmt.Printf("The sum of the random numbers is %d\n", sum)
}

```

The `third()` function keeps reading from the `in` function parameter channel. When that channel is closed by the `second()` function, the `for` loop will stop getting any more data and the function will display its output. At this point, it should become clear that the `second()` function controls many things.

The fifth code segment of `pipeline.go` is as follows:

```

func main() {
    if len(os.Args) != 3 {
        fmt.Println("Need two integer parameters!")
        return
    }

    n1, _ := strconv.Atoi(os.Args[1])
    n2, _ := strconv.Atoi(os.Args[2])

    if n1 > n2 {
        fmt.Printf("%d should be smaller than %d\n", n1, n2)
        return
    }
}

```

The previous code is used for working with the command-line arguments of the program.

The last part of the `pipeline.go` program is as follows:

```
    rand.Seed(time.Now().UnixNano())
    A := make(chan int)
    B := make(chan int)

    go first(n1, n2, A)
    go second(B, A)
    third(B)
}
```

Here you define the required channels, and you execute two goroutines and one function. The `third()` function is what prevents `main()` from returning immediately, because it is not executed as a goroutine.

Executing `pipeline.go` will produce the following type of output:

```
$ go run pipeline.go 1 10
2 2
The sum of the random numbers is 2
$ go run pipeline.go 1 10
9 7 8 4 3 3
The sum of the random numbers is 31
$ go run pipeline.go 1 10
1 6 9 7 1
The sum of the random numbers is 23
$ go run pipeline.go 10 20
16 19 16
The sum of the random numbers is 35
$ go run pipeline.go 10 20
10 16 17 11 15 10
The sum of the random numbers is 69
$ go run pipeline.go 10 20
12 11 14 15 10 15
The sum of the random numbers is 62
```

The important point here is that although the `first()` function keeps generating random numbers at its own pace and the `second()` function will print all of them on your screen, the unwanted random numbers, which are the random numbers that have already appeared, will not be sent to the `third()` function and therefore will not be included in the final sum.

Race conditions

The code of `pipeline.go` is not perfect and contains a logical error, which in concurrent terminology is called a race condition. This can be revealed by executing the following command:

```
$ go run -race pipeline.go 1 10
2 2 =====
WARNING: DATA RACE
Write at 0x00000122bae8 by goroutine 7:
  main.second()
    /Users/mtsouk/ch09/pipeline.go:34 +0x15c

Previous read at 0x00000122bae8 by goroutine 6:
  main.first()
    /Users/mtsouk/ch09/pipeline.go:21 +0xa3

Goroutine 7 (running) created at:
  main.main()
    /Users/mtsouk/ch09/pipeline.go:72 +0x2a1

Goroutine 6 (running) created at:
  main.main()
    /Users/mtsouk/ch09/pipeline.go:71 +0x275
=====
2
The sum of the random numbers is 2.
Found 1 data race(s)
exit status 66
```

The problem here is that the goroutine that executes the `second()` function might change the value of the `CLOSEA` variable while the `first()` function reads the `CLOSEA` variable. As what will happen first and what will happen second is not deterministic, it is considered a race condition. In order to correct this race condition, we will need to use a signal channel and the `select` keyword.



You will learn more about race conditions, signal channels, and the `select` keyword in [chapter 10](#), Concurrency in Go – Advanced Topics.

The output of the `diff(1)` command will reveal the changes made to `pipeline.go` – the new version is called `p1NoRace.go`:

```
$ diff pipeline.go p1NoRace.go
14a15,16
> var signal chan struct{}
```

```
>
21c23,24
<        if CLOSEA {
---
>        select {
>        case <- signal:
23a27
>        case out <- random(min, max):
25d28
<        out <- random(min, max)
31d33
<        fmt.Println(x, " ")
34c36
<        CLOSEA = true
---
>        signal <- struct{}{}
35a38
>        fmt.Println(x, " ")
61d63
<
66a69,70
>        signal = make(chan struct{})
>
```

The logical correctness of `plNoRace.go` can be verified by the output of the next command:

```
$ go run -race plNoRace.go 1 10
8 1 4 9 3
The sum of the random numbers is 25.
```

Comparing Go and Rust concurrency models

Rust is a very popular systems programming language that also supports concurrent programming. Briefly speaking, some characteristics of Rust and the Rust concurrency model are as follows:

- Rust threads are UNIX threads, which means that they are heavy but can do many things.
- Rust supports both **message-passing** and **shared-state** concurrency like Go does with channels, mutexes, and shared variables.
- Based on its strict type and ownership system, Rust provides a safe thread mutable state. The rules are enforced by the Rust compiler.
- There are Rust structures that allow you to share state.
- If a thread starts misbehaving, the system will not crash. This situation can be handled and controlled.
- The Rust programming language is under constant development, which might discourage some people from using it as they might need to make changes to their existing code all the time.

So, Rust has a flexible concurrency model that is even more flexible than the concurrency model of Go. However, the price you will have to pay for this flexibility is having to live with Rust and its idiosyncrasies.

Comparing Go and Erlang concurrency models

Erlang is a very popular concurrent functional programming language that was designed with high availability in mind. Briefly speaking, the main characteristics of Erlang and the Erlang concurrency model are as follows:

- Erlang is a mature and tested programming language – this also applies to its concurrency model.
- If you do not like the way Erlang code works, you can always try **Elixir**, which is based on Erlang and uses the Erlang VM, but its code is more pleasant.
- Erlang uses asynchronous communication and nothing else.
- Erlang uses error handling for developing robust concurrent systems.
- Erlang processes can crash but if that crashing is handled properly, the system can continue working without problems.
- Just like goroutines, Erlang processes are isolated and there is no shared state between them. The one and only way for Erlang processes to communicate with each other is through message passing.
- Erlang threads are lightweight, just like Go goroutines. This means that you will be able to create as many processes as you want.

In summary, both Erlang and Elixir are established choices for reliable and highly available systems, as long as you are willing to work with the Erlang concurrency approach.

Additional resources

Visit the following useful resources:

- Visit the documentation page of the `sync` package, which can be found at <https://golang.org/pkg/sync/>.
- Visit the Rust web site at <https://www.rust-lang.org/>.
- Visit the Erlang web site at <https://www.erlang.org/>.
- Visit the documentation page of the `sync` package once more. Pay close attention to the `sync.Mutex` and `sync.RWMutex` types that will appear in the next chapter.

Exercises

- Create a pipeline that reads text files, finds the number of occurrences of a given phrase in each text file, and calculate the total number of occurrences of the phrase in all files.
- Create a pipeline for calculating the sum of the squares of all of the natural numbers in a given range.
- Remove the `time.Sleep(1 * time.Second)` statement from the `simple.go` program and see what happens. Why is that?
- Modify the Go code of `pipeline.go` in order to create a pipeline with five functions and the appropriate number of channels.
- Modify the Go code of `pipeline.go` in order to find out what will happen when you forget to close the `out` channel of the `first()` function.

Summary

In this chapter, you studied many unique Go features including goroutines, channels, and pipelines. Additionally, you found out how to give your goroutines enough time to finish their jobs using the functionality offered by the `sync` package. Finally, you learned that channels can be used as parameters to Go functions. This allows developers to create pipelines where data can flow.

The next chapter will continue talking about Go concurrency by introducing the formidable `select` keyword. This keyword helps Go channels to perform many interesting jobs, and I think that you will be truly amazed by its power.

After that, you will see two techniques that allow you to time out one or more goroutines that are stalled for some reason. Afterward, you will learn about nil channels, signal channels, channel of channels, and buffered channels, as well as the `context` package.

You will also learn about **shared memory**, which is the traditional way of sharing information among the threads of the same UNIX process that also applies to goroutines. Nevertheless, shared memory is not that popular among Go programmers because Go offers better, safer, and faster ways for goroutines to exchange data.

Concurrency in Go – Advanced Topics

The previous chapter introduced goroutines, which are the most important feature of Go, channels, and pipelines. This chapter will continue from the point where the previous one left off in order to help you to learn more about goroutines, channels, and the `select` keyword, before discussing shared variables, as well as the `sync.Mutex` and `sync.RWMutex` types.

This chapter also includes code examples that demonstrate the use of **signal channels**, **buffered channels**, **nil channels**, and **channels of channels**. Additionally, early on in this chapter, you will learn two techniques for timing out a goroutine after a given amount of time, because nobody can guarantee that all goroutines will finish before a desired time.

The chapter will end by examining the `atomic` package, race conditions, the `context` standard Go package, and worker pools.

In this chapter, you will learn about the following topics:

- The `select` keyword
- How the Go scheduler works
- Two techniques that allow you to time out a goroutine that takes longer than expected to finish
- Signal channels
- Buffered channels
- Nil channels
- Monitor goroutines
- Shared memory and mutexes
- The `sync.Mutex` and `sync.RWMutex` types
- The `context` package and its advanced functionality
- The `atomic` package
- Worker pools
- Detecting race conditions

The Go scheduler revisited

A **scheduler** is responsible for distributing the amount of work that needs to be done over the available resources in an efficient way. In this section, we will examine the way that the Go scheduler operates in much greater depth than in the previous chapter. As you already know, Go works using the **m:n scheduler** (or **M:N scheduler**). It schedules goroutines, which are lighter than OS threads, using OS threads. First, though, let us review the necessary theory and define some useful terms.

Go uses the **fork-join concurrency** model. The fork part of the model states that a child branch can be created at any point of a program. Analogously, the join part of the Go concurrency model is where the child branch ends and joins with its parent. Among other things, both `sync.Wait()` statements and channels that collect the results of goroutines are join points, whereas each new goroutine creates a child branch.



The fork phase of the fork-join model and the `fork(2)` C system call are two totally different things.

The **fair scheduling strategy**, which is pretty straightforward and has a simple implementation, shares all load evenly among the available processors. At first, this might look like the perfect strategy because it does not have to take many things into consideration while keeping all processors equally occupied. However, it turns out that this is not exactly the case because most distributed tasks usually depend on other tasks. Therefore, some processors are underutilized, or equivalently, some processors are utilized more than others.

A goroutine in Go is a **task**, whereas everything after the calling statement of a goroutine is a **continuation**. In the **work-stealing strategy** used by the Go scheduler, a (logical) processor that is underutilized looks for additional work from other processors. When it finds such jobs, it steals them from the other processor or processors, hence the name. Additionally, the work-

stealing algorithm of Go queues and steals continuations. A **stalling join**, as is suggested by its name, is a point where a thread of execution stalls at a join and starts looking for other work to do.

Although both task stealing and continuation stealing have stalling joins, continuations happen more often than tasks; therefore, the Go algorithm works with continuations rather than tasks.

The main disadvantage of continuation stealing is that it requires extra work from the compiler of the programming language. Fortunately, Go provides that extra help and therefore uses **continuation stealing** in its work-stealing algorithm.

One of the benefits of continuation stealing is that you get the same results when using just functions instead of goroutines or a single thread with multiple goroutines. This makes perfect sense, as only one thing is executed at any given time in both cases.

Now, let us return to the $m:n$ scheduling algorithm used in Go. Strictly speaking, at any time, you have m goroutines that are executed, and therefore scheduled to run, on n OS threads using, at most, `GOMAXPROCS` number of logical processors. You will learn about `GOMAXPROCS` shortly.

The Go scheduler works using three main kinds of entities: OS threads (**M**), which are related to the operating system in use, goroutines (**G**), and **logical processors (P)**. The number of processors that can be used by a Go program is specified by the value of the **GOMAXPROCS environment variable** – at any given time, there are at most `GOMAXPROCS` processors.

The following figure illustrates this point:

Global queue

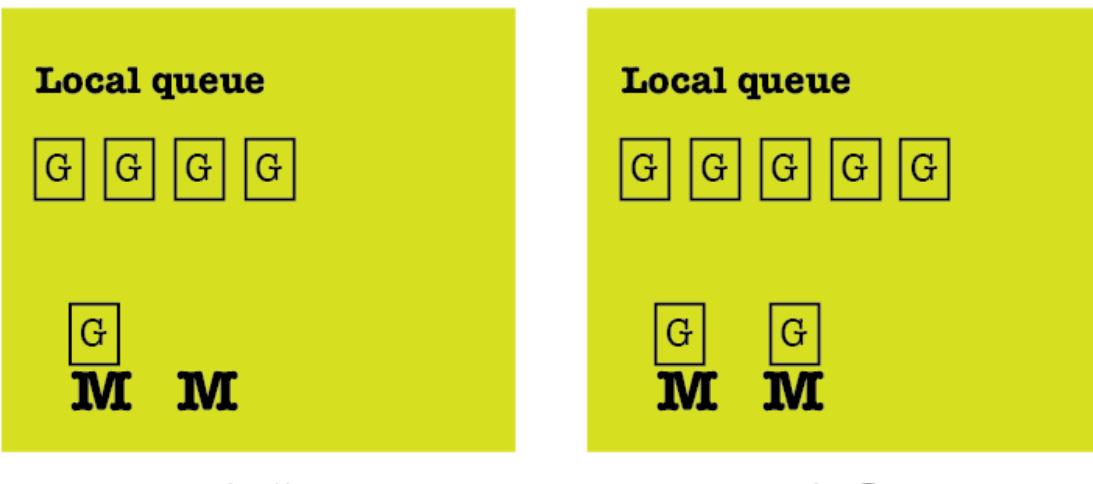


Figure 10.1: How the Go scheduler works

What the figure tells us is that there are two different kinds of queues: a global queue and a local queue attached to each logical processor.

Goroutines from the global queue are assigned to the queue of a logical processor in order to be executed. As a result, the Go scheduler needs to check the global queue in order to avoid executing goroutines that are only located at the local queue of each logical processor. However, the global queue is not checked all of the time, which means that it does not have an advantage over the local queue.

Additionally, each logical processor can have multiple threads, and the stealing occurs between the local queues of the available logical processors. Finally, keep in mind that the Go scheduler is allowed to create more OS threads when needed. OS threads are pretty expensive, however, which means that dealing too much with OS threads might slow down your Go applications.

Remember that using more goroutines in a program is not a panacea for performance, as more goroutines, in addition to the various calls to

`sync.Add()`, `sync.Wait()`, and `sync.Done()`, might slow down your program due to the extra housekeeping that needs to be done by the Go scheduler.



The Go scheduler, as well as most Go components, is always evolving, which means that the people who work on the Go scheduler constantly try to improve its performance by making small changes to the way it works. The core principles, however, remain the same.

You do not need to know all of this information in order to write Go code that uses goroutines. But knowing what occurs behind the scenes can definitely help you when strange things start happening or if you are curious about how the Go scheduler works. It will certainly make you a better developer!

The `GOMAXPROCS` environment variable

The `GOMAXPROCS` environment variable (and Go function) allows you to limit the number of operating system threads that can execute user-level Go code simultaneously. Starting with Go version 1.5, the default value of `GOMAXPROCS` should be the number of logical cores available in your UNIX machine.

If you decide to assign a value to `GOMAXPROCS` that is less than the number of the cores in your UNIX machine, you might affect the performance of your program. However, using a `GOMAXPROCS` value that is larger than the number of the available cores will not necessarily make your Go programs run faster.

You can programmatically discover the value of the `GOMAXPROCS` environment variable; the relevant code can be found in the following program, which is named `maxprocs.go`:

```
package main

import (
    "fmt"
    "runtime"
)

func getGOMAXPROCS() int {
    return runtime.GOMAXPROCS(0)
}

func main() {
    fmt.Printf("GOMAXPROCS: %d\n", getGOMAXPROCS())
}
```

Executing `maxprocs.go` on a machine with an Intel i7 processor will produce the following output:

```
$ go run maxprocs.go
GOMAXPROCS: 8
```

You can modify the previous output by changing the value of the `GOMAXPROCS` environment variable prior to the execution of the program, however. The following commands are executed in the bash(1) UNIX shell:

```
$ go version
go version go1.12.3 darwin/amd64
$ export GOMAXPROCS=800; go run maxprocs.go
GOMAXPROCS: 800
$ export GOMAXPROCS=4; go run maxprocs.go
GOMAXPROCS: 4
```

The select keyword

As you will learn in a short while, the `select` keyword is pretty powerful and can do many things in a variety of situations. The `select` statement in Go looks like a `switch` statement but for channels. In practice, this means that `select` allows a goroutine to wait on multiple communication operations. Therefore, the main benefit that you receive from `select` is that it gives you the power to work with multiple channels using a single `select` block. As a consequence, you can have nonblocking operations on channels, provided that you have appropriate `select` blocks.



*The biggest problem when using multiple channels and the `select` keyword is **deadlocks**. This means that you should be extra careful during the design and the development process in order to avoid such deadlocks.*

The Go code of `select.go` will clarify the use of the `select` keyword. This program will be presented in five parts. The first part of `select.go` is shown in the following Go code:

```
package main

import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "time"
)
```

The second code portion from `select.go` is as follows:

```
func gen(min, max int, createNumber chan int, end chan bool) {
    for {
        select {
        case createNumber <- rand.Intn(max-min) + min:
        case <-end:
            close(end)
            return
        case <-time.After(4 * time.Second):
            fmt.Println("\ntime.After()!")
        }
    }
}
```

So, what is really happening in the code of this `select` block? This particular `select` statement has three cases. Notice that `select` statements do not require a `default` branch. You can consider the third branch of the `select` statement of the aforementioned code as a clever `default` branch. This happens because `time.After()` waits for the specified duration to elapse and then sends the current time on the returned channel – this will unblock the `select` statement in case all of the other channels are blocked for some reason.

A `select` statement is not evaluated sequentially, as all of its channels are examined simultaneously. If none of the channels in a `select` statement are ready, the `select` statement will block until one of the channels is ready. If multiple channels of a `select` statement are ready, then the Go runtime will make a random selection from the set of these ready channels.

The Go runtime tries to make this random selection between these ready channels as uniformly and as fairly as possible.

The third part of `select.go` is as follows:

```
func main() {
    rand.Seed(time.Now().Unix())
    createNumber := make(chan int)
    end := make(chan bool)

    if len(os.Args) != 2 {
        fmt.Println("Please give me an integer!")
        return
    }
```

The fourth code portion of the `select.go` program contains the following Go code:

```
n, _ := strconv.Atoi(os.Args[1])
fmt.Printf("Going to create %d random numbers.\n", n)
go gen(0, 2*n, createNumber, end)

for i := 0; i < n; i++ {
    fmt.Printf("%d ", <-createNumber)
}
```



The reason for not examining the `error` value returned by `strconv.Atoi()` is to save some space. You should never do this in real applications.

The remaining Go code of the `select.go` program is as follows:

```
|     time.Sleep(5 * time.Second)
|     fmt.Println("Exiting...")
| end <- true
| }
```

The main purpose of the `time.Sleep(5 * time.Second)` statement is to give the `time.After()` function of `gen()` enough time to return and therefore activate the relevant branch of the `select` statement.

The last statement of the `main()` function is what terminates the program by activating the `case <-end` branch of the `select` statement in `gen()` and executing the related Go code.

Executing `select.go` will generate the following output:

```
$ go run select.go 10
Going to create 10 random numbers.
13 17 8 14 19 9 2 0 19 5
time.After()!
Exiting...
```



The biggest advantage of `select` is that it can connect, orchestrate, and manage multiple channels. As channels connect goroutines, `select` connects channels that connect goroutines. Therefore, the `select` statement is one of the most important, if not the single most important, part of the Go concurrency model.

Timing out a goroutine

This section presents two very important techniques that will help you to time out goroutines. Put simply, these two techniques will save you from having to wait forever for a goroutine to finish its job, and they will give you full control over the amount of time that you want to wait for a goroutine to end. Both techniques use the capabilities of the handy `select` keyword combined with the `time.After()` function that you experienced in the previous section.

Timing out a goroutine – take 1

The source code of the first technique will be saved in `timeOut1.go`, and it will be presented in four parts.

The first part of `timeOut1.go` is shown in the following Go code:

```
package main  
import (  
    "fmt"  
    "time"  
)
```

The second code segment from `timeOut1.go` is as follows:

```
func main() {  
    c1 := make(chan string)  
    go func() {  
        time.Sleep(time.Second * 3)  
        c1 <- "c1 OK"  
    }()  
}
```

The `time.Sleep()` call is used to emulate the time it will normally take for the function to finish its job. In this case, the anonymous function that is executed as a goroutine will take about three seconds (`time.Second * 3`) before writing a message to the `c1` channel.

The third code segment from `timeOut1.go` contains the following Go code:

```
select {  
case res := <-c1:  
    fmt.Println(res)  
case <-time.After(time.Second * 1):  
    fmt.Println("timeout c1")  
}
```

The purpose of the `time.After()` function call is to wait for the chosen time. In this case, you are not interested in the actual value returned by `time.After()` but in the fact that the `time.After()` function call has ended, which means that the available waiting time has passed. In this case, as the value passed to the `time.After()` function is smaller than the value used in the

`time.Sleep()` call that was executed as the goroutine in the previous code segment, you will most likely get a timeout message.

The remaining code from `timeOut1.go` is as follows:

```
c2 := make(chan string)
go func() {
    time.Sleep(3 * time.Second)
    c2 <- "c2 OK"
}()

select {
case res := <-c2:
    fmt.Println(res)
case <-time.After(4 * time.Second):
    fmt.Println("timeout c2")
}
```

The preceding code both executes a goroutine that will take about three seconds to execute because of the `time.Sleep()` call and defines a timeout period of four seconds using `time.After(4 * time.Second)`. If the `time.After(4 * time.Second)` call returns after you get a value from the `c2` channel found in the first case of the `select` block, then there will not be any timeout; otherwise, you will get a timeout! However, in this case, the value of the `time.After()` call provides enough time for the `time.Sleep()` call to return, so you will most likely not get a timeout message here.

Executing `timeOut1.go` will generate the following type of output:

```
$ go run timeOut1.go
timeout c1
c2 OK
```

As expected, the first goroutine did not finish its job, whereas the second goroutine had enough time to finish.

Timing out a goroutine – take 2

The source code of the second technique will be saved in `timeOut2.go`, and it will be presented in five parts. This time, the timeout period is provided as a command-line argument to the program.

The first part of `timeOut2.go` is as follows:

```
package main

import (
    "fmt"
    "os"
    "strconv"
    "sync"
    "time"
)
```

The second code segment of `timeOut2.go` is shown in the following Go code:

```
func timeout(w *sync.WaitGroup, t time.Duration) bool {
    temp := make(chan int)
    go func() {
        defer close(temp)
        time.Sleep(5 * time.Second)

        w.Wait()
    }()

    select {
    case <-temp:
        return false
    case <-time.After(t):
        return true
    }
}
```

In the preceding code, the time duration that will be used in the `time.After()` call is a parameter to the `timeout()` function, which means that it can vary.

Once again, the `select` block implements the logic of the time out.

Additionally, the `w.Wait()` call will make the `timeout()` function wait for a matching `sync.Done()` function indefinitely in order to end. When the `w.Wait()` call returns, the first branch of the `select` statement will be executed.

The third code portion from `timeOut2.go` is as follows:

```

func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Need a time duration!")
        return
    }

    var w sync.WaitGroup
    w.Add(1)

    t, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println(err)
        return
    }
}

```

The fourth part of the `timeOut2.go` program is as follows:

```

duration := time.Duration(int32(t)) * time.Millisecond
fmt.Printf("Timeout period is %s\n", duration)

if timeout(&w, duration) {
    fmt.Println("Timed out!")
} else {
    fmt.Println("OK!")
}

```

The `time.Duration()` function converts an integer value into a `time.Duration` variable that you can use afterward.

The remaining Go code from `timeOut2.go` is as follows:

```

w.Done()
if timeout(&w, duration) {
    fmt.Println("Timed out!")
} else {
    fmt.Println("OK!")
}
}

```

Once the `w.Done()` call is executed, the previous `timeout()` function will return. However, the second call to `timeout()` has no `sync.Done()` statement to wait for.

Executing `timeOut2.go` will generate the following type of output:

```

$ go run timeOut2.go 10000
Timeout period is 10s
Timed out!
OK!

```

In this execution of `timeOut2.go`, the timeout period is longer than the `time.Sleep(5 * time.Second)` call of the anonymous goroutine. However, without the required call to `w.Done()`, the anonymous goroutine cannot return and therefore the `time.After(t)` call will end first, so the `timeout()` function of the first `if` statement will return `true`. In the second `if` statement, the anonymous function does not have to wait for anything, so the `timeout()` function will return `false` because `time.Sleep(5 * time.Second)` will finish before `time.After(t)`.

```
$ go run timeOut2.go 100
Timeout period is 100ms
Timed out!
Timed out!
```

In the second program execution, however, the timeout period is too small, so both executions of `timeout()` will not have enough time to finish; therefore, both will be timed out. So, when defining a timeout period, make sure that you choose an appropriate value, or your results might not be what you expect.

Go channels revisited

Once the `select` keyword comes into play, Go channels can be used in several unique ways to do many more things than those you experienced in [Chapter 9, Concurrency in Go – Goroutines, Channels, and Pipelines](#). This section will reveal the many uses of Go channels.

It helps to remember that the zero value of the channel type is `nil`, and that if you send a message to a closed channel, the program will panic. However, if you try to read from a closed channel, you will get the zero value of the type of that channel. So, after closing a channel, you can no longer write to it, but you can still read from it.

In order to be able to close a channel, the channel must not be receive-only. Additionally, a `nil` channel always blocks, which means that trying to read or write from a `nil` channel will block. This property of channels can be very useful when you want to disable a branch of a `select` statement by assigning the `nil` value to a channel variable.

Finally, if you try to close a `nil` channel, your program will panic. This is best illustrated in the `closeNilChannel.go` program, which is presented next:

```
package main

func main() {
    var c chan string
    close(c)
}
```

Executing `closeNilChannel.go` will generate the following output:

```
$ go run closeNilChannel.go
panic: close of nil channel

goroutine 1 [running]:
main.main()
    /Users/mtsouk/closeNilChannel.go:5 +0x2a
exit status 2
```

Signal channels

A signal channel is one that is used just for signaling. Put simply, you can use a signal channel when you want to inform another goroutine about something. Signal channels should not be used for the transferring of data.



*You should not confuse signal channels with **UNIX signal handling**, which was discussed in [Chapter 8](#), *Telling a UNIX System What to Do*, because they are totally different things.*

You will see a code example that uses signal channels in the section called *Specifying the order of execution for your goroutines* later in this chapter.

Buffered channels

The topic of this subsection is buffered channels. These are channels that allow the Go scheduler to put jobs in the queue quickly in order to be able to deal with more requests.

Moreover, you can use buffered channels as **semaphores** in order to limit the throughput of your application.

The technique presented here works as follows: all incoming requests are forwarded to a channel, which processes them one by one. When the channel is done processing a request, it sends a message to the original caller saying that it is ready to process a new one. So, the capacity of the buffer of the channel restricts the number of simultaneous requests that it can keep.

The technique will be presented with the help of the code found in `bufChannel.go`, which is broken down into four parts.

The first part of the code of `bufChannel.go` is as follows:

```
package main
|
import (
    "fmt"
)
```

The second code segment of `bufChannel.go` contains the following Go code:

```
func main() {
    numbers := make(chan int, 5)
    counter := 10
```

The definition presented of the `numbers` channel gives it a place to store up to five integers.

The third part of the code of `bufChannel.go` is shown in the following Go code:

```
for i := 0; i < counter; i++ {
    select {
        case numbers <- i:
        default:
            fmt.Println("Not enough space for", i)
    }
}
```

In the preceding code, we tried to put 10 integers in the `numbers` channel. However, as the `numbers` channel has room for only five integers, we will not be able to store all 10 integers in it.

The remaining Go code of `bufChannel.go` follows:

```
for i := 0; i < counter+5; i++ {  
    select {  
  
        case num := <-numbers:  
            fmt.Println(num)  
        default:  
            fmt.Println("Nothing more to be done!")  
            break  
    }  
}  
}
```

In the preceding Go code, we tried to read the contents of the `numbers` channel using a `for` loop and a `select` statement. As long as there is something to read from the `numbers` channel, the first branch of the `select` statement will get executed. As long as the `numbers` channel is empty, the `default` branch will be executed.

Executing `bufChannel.go` will create the following type of output:

```
$ go run bufChannel.go
Not enough space for 5
Not enough space for 6
Not enough space for 7
Not enough space for 8
Not enough space for 9
0
1
2
3
4
Nothing more to be done!
```

Nothing more to be done!
Nothing more to be done!
Nothing more to be done!

Nil channels

In this subsection, you will learn about nil channels. These are a special kind of channel because they always block. They are illustrated in `nilChannel.go`, which will be presented in four code segments.

The first part of `nilChannel.go` is as follows:

```
package main

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

The second code portion of `nilChannel.go` is shown in the following Go code:

```
func add(c chan int) {
    sum := 0
    t := time.NewTimer(time.Second)

    for {
        select {
        case input := <-c:
            sum = sum + input
        case <-t.C:
            c = nil
            fmt.Println(sum)
        }
    }
}
```

The `add()` function demonstrates how a nil channel is used. The `<-t.c` statement blocks the `c` channel of the `t` timer for the time that is specified in the `time.NewTimer()` call. Do not confuse channel `c`, which is the parameter of the function, with channel `t.c`, which belongs to timer `t`. When the time expires, the timer sends a value to the `t.c` channel. This will trigger the execution of the relevant branch of the `select` statement, which will assign the value `nil` to channel `c` and print the `sum` variable.

The third code segment of `nilChannel.go` is as follows:

```
func send(c chan int) {
    for {
        c <- rand.Intn(10)
    }
}
```

The purpose of the `send()` function is to generate random numbers and continue sending them to a channel for as long as the channel is open.

The remaining Go code of `nilChannel.go` is as follows:

```
func main() {
    c := make(chan int)
    go add(c)
    go send(c)

    time.Sleep(3 * time.Second)
}
```

The `time.Sleep()` function is used to give enough time to the two goroutines to operate.

Executing `nilChannel.go` will generate the following output:

```
$ go run nilChannel.go
13167523
$ go run nilChannel.go
12988362
```

Since the number of times that the first branch of the `select` statement in the `add()` function will be executed is not fixed, you get different results from executing `nilChannel.go`.

Channels of channels

A channel of channels is a special kind of channel variable that works with channels instead of other types of variables. Nevertheless, you still have to declare a data type for a channel of channels. You can define a channel of channels using the `chan` keyword twice in a row, as shown in the following statement:

```
| c1 := make(chan chan int)
```



The other types of channels presented in this chapter are far more popular and useful than a channel of channels.

The use of channels of channels is illustrated using the code found in `chSquare.go`, which will be presented in four parts.

The first part of `chSquare.go` is as follows:

```
package main

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

var times int
```

The second code portion from `chSquare.go` is shown in the following Go code:

```
func f1(cc chan chan int, f chan bool) {
    c := make(chan int)
    cc <- c
    defer close(c)

    sum := 0
    select {
    case x := <-c:
        for i := 0; i <= x; i++ {
            sum = sum + i
        }
        c <- sum
    case <-f:
```

```
    }
}
```

After declaring a regular `int` channel, you send that to the `channel of channels` variable. Then, you use a `select` statement in order to be able to read data from the regular `int` channel or exit your function using the `f` signal channel.

Once you read a single value from the `c` channel, you start a `for` loop that calculates the sum of all integers from `0` up to the integer value that you just read. Next, you send the calculated value to the `c int` channel and you are done.

The third part of `chSquare.go` contains the following Go code:

```
func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Need just one integer argument!")
        return
    }

    times, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println(err)
        return
    }

    cc := make(chan chan int)
```

The last statement in the preceding code is where you declare a `channel of channels` variable named `cc`, which is the star of the program because everything depends on that variable. The `cc` variable is passed to the `f1()` function, and it will be used in the `for` loop that is coming next.

The remaining Go code of `chSquare.go` is as follows:

```
for i := 1; i < times+1; i++ {
    f := make(chan bool)
    go f1(cc, f)
    ch := <-cc
    ch <- i
    for sum := range ch {
        fmt.Print("Sum(", i, ")=", sum)
    }
    fmt.Println()
    time.Sleep(time.Second)
```

```
|           close(f)
| }
}
```

The `f` channel is a signal channel used to end the goroutine when the real work is finished. The `ch := <-cc` statement allows you to get a regular channel from the channel of channels variable in order to be able to send an `int` value to it using `ch <- i`. After that, you start reading from it using a `for` loop. Although the `f1()` function is programmed to send a single value back, you can also read multiple values. Notice that each value of `i` is served by a different goroutine.

The type of a signal channel can be anything you want, including `bool`, which is used in the preceding code, and `struct{}`, which will be used in the signal channel in the next section. The main advantage of a `struct{}` signal channel is that no data can be sent to it, which can save you from bugs and misconceptions.

Executing `chSquare.go` will generate the following type of output:

```
$ go run chSquare.go 4
Sum(1)=1
Sum(2)=3
Sum(3)=6
Sum(4)=10
$ go run chSquare.go 7
Sum(1)=1
Sum(2)=3
Sum(3)=6
Sum(4)=10
Sum(5)=15
Sum(6)=21
Sum(7)=28
```

Specifying the order of execution for your goroutines

Although you should not make any assumptions about the order in which your goroutines will be executed, there are times when you need to be able to control this order. This subsection illustrates such a technique using signal channels.



You might ask, "Why choose to execute goroutines in a given order when simple functions could do the same job much more easily?" The answer is simple: goroutines are able to operate concurrently and wait for other goroutines to end, whereas functions executed in sequence cannot do that.

The name of the Go program for this topic is `defineOrder.go`, and it will be presented in five parts. The first part of `defineOrder.go` follows:

```
package main

import (
    "fmt"
    "time"
)

func A(a, b chan struct{}) {
    <-a
    fmt.Println("A()!")
    time.Sleep(time.Second)
    close(b)
}
```

The `A()` function is blocked by the channel stored in the `a` parameter. Once that channel is unblocked in the `main()` function, the `A()` function will start working. Finally, it will close channel `b`, which will unblock another function – in this case, function `B()`.

The second code portion of `defineOrder.go` is shown in the following Go code:

```
func B(a, b chan struct{}) {
    <-a
    fmt.Println("B()!")
```

```
|     close(b)  
| }
```

The logic in `B()` is the same as in function `A()`. The function is blocked until channel `a` is closed. Then, it does its job and closes channel `b`. Notice that channels `a` and `b` refer to the names of the parameters of the function.

The third code segment of `defineOrder.go` follows:

```
| func C(a chan struct{}) {  
|     <-a  
|     fmt.Println("C()!")  
| }
```

Function `C()` is blocked and waits for channel `a` to close in order to start working.

The fourth part of `defineOrder.go` contains the following code:

```
| func main() {  
|     x := make(chan struct{})  
|     y := make(chan struct{})  
|     z := make(chan struct{})
```

These three channels will be the parameters to the three functions.

The last code segment of `defineOrder.go` contains the following Go code:

```
| go C(z)  
| go A(x, y)  
| go C(z)  
| go B(y, z)  
| go C(z)  
  
| close(x)  
| time.Sleep(3 * time.Second)  
| }
```

In this part, we execute the desired goroutines before closing the `x` channel and sleeping for three seconds.

Executing `defineOrder.go` will generate the desired output even though the `C()` function is called multiple times:

```
$ go run defineOrder.go  
A()  
B()
```

```
| C() !
| C() !
| C() !
```

Calling the `C()` function multiple times as goroutines will work just fine because `C()` does not close any channels. However, if you call `A()` or `B()` more than once, you will most likely get an error message such as the following:

```
$ go run defineOrder.go
A() !
A() !
B() !
C() !
C() !
C() !
panic: close of closed channel

goroutine 7 [running]:
main.A(0xc420072060, 0xc4200720c0)
    /Users/mtsouk/Desktop/defineOrder.go:12 +0x9d
created by main.main
    /Users/mtsouk/Desktop/defineOrder.go:33 +0xfa
exit status 2
```

As you can see from the output, function `A()` was called twice. However, as function `A()` closes a channel, one of its goroutines will find that channel already closed and generate a panic situation when it tries to close it again. You will get a similar panic situation if you call `B()` more than once.

How not to use goroutines

In this section, you are going to see a naive way to sort natural numbers using goroutines. The name of the program is `sillySort.go` and it will be presented in two parts. The first part of `sillySort.go` is the following:

```
package main

import (
    "fmt"
    "os"
    "strconv"
    "sync"
    "time"
)

func main() {
    arguments := os.Args

    if len(arguments) == 1 {
        fmt.Println(os.Args[0], "n1, n2, [n]")
        return
    }

    var wg sync.WaitGroup
    for _, arg := range arguments[1:] {
        n, err := strconv.Atoi(arg)
        if err != nil || n < 0 {
            fmt.Print(". ")
            continue
        }
    }
}
```

The second part of `sillySort.go` contains the following Go code:

```
    wg.Add(1)
    go func(n int) {
        defer wg.Done()
        time.Sleep(time.Duration(n) * time.Second)
        fmt.Print(n, " ")
    }(n)
}

wg.Wait()
fmt.Println()
```

The sorting takes place with the help of the `time.Sleep()` call – the bigger the natural number, the bigger the wait for the execution of the `fmt.Println()` statement!

Executing `sillySort.go` will generate the following kind of output:

```
$ go run sillySort.go a -1 1 2 3 5 0 100 20 60
. . 0 1 2 3 5 20 60 100
$ go run sillySort.go a -1 1 2 3 5 0 100 -1 a 20 hello 60
. . . . 0 1 2 3 5 20 60 100
$ go run sillySort.go 0 0 10 2 30 3 4 30
0 0 2 3 4 10 30 30
```

Shared memory and shared variables

Shared memory and **shared variables** are the most common ways for UNIX threads to communicate with each other.

A **mutex** variable, which is an abbreviation of **mutual exclusion** variable, is mainly used for thread synchronization and for protecting shared data when multiple writes can occur at the same time. A mutex works like a buffered channel with a capacity of one, which allows at most one goroutine to access a shared variable at any given time. This means that there is no way for two or more goroutines to try to update that variable simultaneously.

A **critical section** of a concurrent program is the code that cannot be executed simultaneously by all processes, threads, or, in this case, goroutines. It is the code that needs to be protected by mutexes. Therefore, identifying the critical sections of your code will make the whole programming process so much simpler that you should pay particular attention to this task.



A critical section cannot be embedded into another critical section when both critical sections use the same `sync.Mutex` or `sync.RWMutex` variable. Put simply, avoid at almost any cost spreading mutexes across functions because that makes it really hard to see whether you are embedding or not.

The next two subsections will illustrate the use of the `sync.Mutex` and `sync.RWMutex` types.

The sync.Mutex type

The `sync.Mutex` type is the Go implementation of a mutex. Its definition, which can be found in the `mutex.go` file of the `sync` directory, is as follows:

```
// A Mutex is a mutual exclusion lock.  
// The zero value for a Mutex is an unlocked mutex.  
//  
// A Mutex must not be copied after first use.  
type Mutex struct {  
    state int32  
    sema  uint32  
}
```



If you are interested in what code in the standard library is doing, remember that Go is completely open source and you can go and read it.

The definition of the `sync.Mutex` type is nothing extraordinary. All of the interesting work is done by the `sync.Lock()` and `sync.Unlock()` functions, which can lock and unlock a `sync.Mutex` mutex, respectively. Locking a mutex means that nobody else can lock it until it has been released using the `sync.Unlock()` function.

The `mutex.go` program, which is going to be presented in five parts, illustrates the use of the `sync.Mutex` type.

The first code segment of `mutex.go` is as follows:

```
package main  
  
import (  
    "fmt"  
    "os"  
    "strconv"  
    "sync"  
    "time"  
)  
  
var (  
    m  sync.Mutex  
    v1 int  
)
```

The second part of `mutex.go` is shown in the following Go code:

```
func change(i int) {  
    m.Lock()  
    time.Sleep(time.Second)  
    v1 = v1 + 1  
    if v1%10 == 0 {  
        v1 = v1 - 10*i  
    }  
    m.Unlock()  
}
```

The critical section of this function is the Go code between the `m.Lock()` and `m.Unlock()` statements.

The third part of `mutex.go` contains the following Go code:

```
func read() int {  
    m.Lock()  
    a := v1  
    m.Unlock()  
    return a  
}
```

Similarly, the critical section of this function is defined by the `m.Lock()` and `m.Unlock()` statements.

The fourth code segment of `mutex.go` is as follows:

```
func main() {  
    if len(os.Args) != 2 {  
        fmt.Println("Please give me an integer!")  
        return  
    }
```

```

numGR, err := strconv.Atoi(os.Args[1])
if err != nil {
    fmt.Println(err)
    return
}
var waitGroup sync.WaitGroup

```

The last part of `mutex.go` is shown in the following Go code:

```

fmt.Printf("%d ", read())
for i := 0; i < numGR; i++ {
    waitGroup.Add(1)
    go func(i int) {
        defer waitGroup.Done()
        change(i)
        fmt.Printf("-> %d", read())
    }(i)
}
waitGroup.Wait()
fmt.Printf("-> %d\n", read())
}

```

Executing `mutex.go` will generate the following type of output:

```

$ go run mutex.go 21
0 -> 1-> 2-> 3-> 4-> 5-> 6-> 7-> 8-> 9-> -30-> -29-> -28-> -27-> -26-> -25-> -24-> -23-> -22-> -21-> -210-> -209-> -209
$ go run mutex.go 21
0 -> 1-> 2-> 3-> 4-> 5-> 6-> 7-> 8-> 9-> -130-> -129-> -128-> -127-> -126-> -125-> -124-> -123-> -122-> -121-> -220-> -219-> -;
$ go run mutex.go 21
0 -> 1-> 2-> 3-> 4-> 5-> 6-> 7-> 8-> 9-> -100-> -99-> -98-> -97-> -96-> -95-> -94-> -93-> -92-> -91-> -260-> -259-> -259

```

If you remove the `m.Lock()` and `m.Unlock()` statements from the `change()` function, the program will generate output similar to the following:

```

$ go run mutex.go 21
0 -> 1-> 6-> 7-> 5-> -60-> -59-> 9-> 2-> -58-> 3-> -52-> 4-> -57-> 8-> -55-> -90-> -54-> -89-> -53-> -56-> -51-> -89
$ go run mutex.go 21
0 -> 1-> 7-> 8-> 9-> 5-> -99-> 4-> 2-> -97-> -96-> 3-> -98-> -95-> -100-> -93-> -94-> -92-> -91-> -230-> 6-> -229-> -229
$ go run mutex.go 21
0 -> 3-> 7-> 8-> 9-> -120-> -119-> -118-> -117-> 1-> -115-> -114-> -116-> 4-> 6-> -112-> 2-> -111-> 5-> -260-> -113-> -259-> -;

```

The reason for such a change in the output is that all goroutines are simultaneously changing the shared variable, which is the main reason that the output appears randomly generated.

What happens if you forget to unlock a mutex?

In this section, you will see what happens if you forget to unlock a `sync.Mutex`. You will do this using the Go code of `forgetMutex.go`, which will be presented in two parts.

The first part of `forgetMutex.go` is shown in the following Go code:

```
package main

import (
    "fmt"
    "sync"
)

var m sync.Mutex

func function() {
    m.Lock()
    fmt.Println("Locked!")
}
```

All of the problems in this program are caused because the developer forgot to release the lock to `m sync.Mutex`. However, if your program is going to call `function()` only once, then everything will look just fine.

The second part of `forgetMutex.go` is as follows:

```
func main() {
    var w sync.WaitGroup

    go func() {
        defer w.Done()
        function()
    }()
    w.Add(1)

    go func() {
        defer w.Done()
        function()
    }()
    w.Add(1)

    w.Wait()
}
```

There is nothing wrong with the `main()` function, which generates just two goroutines and waits for them to finish.

Executing `forgetMutex.go` will produce the following output:

```
$ go run forgetMutex.go
Locked!
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [semacquire]:
sync.runtime_Semacquire(0xc42001209c)
    /usr/local/Cellar/go/1.12.3/libexec/src/runtime/sema.go:56 +0x39
sync.(*WaitGroup).Wait(0xc420012090)
    /usr/local/Cellar/go/1.12.3/libexec/src/sync/waitgroup.go:131 +0x72
main.main()
    /Users/mtsouk/forgetMutex.go:30 +0xb6

goroutine 5 [semacquire]:
sync.runtime_SemacquireMutex(0x115c6fc, 0x0)
    /usr/local/Cellar/go/1.12.3/libexec/src/runtime/sema.go:71 +0x3d
sync.(*Mutex).Lock(0x115c6f8)
    /usr/local/Cellar/go/1.12.3/libexec/src/sync/mutex.go:134 +0xee
main.function()
    /Users/mtsouk/forgetMutex.go:11 +0x2d
main.main.func1(0xc420012090)
    /Users/mtsouk/forgetMutex.go:20 +0x48
created by main.main
    /Users/mtsouk/forgetMutex.go:18 +0x58
exit status 2
```

So, forgetting to unlock a `sync.Mutex` mutex will create a panic situation even in the simplest kind of program. The same applies to the `sync.RWMutex` type of mutex, which you are going to work with in the next section.

The sync.RWMutex type

The `sync.RWMutex` type is another kind of mutex – actually, it is an improved version of `sync.Mutex`, which is defined in the `rwmutex.go` file of the `sync` directory as follows:

```
| type RWMutex struct {  
|     w           Mutex // held if there are pending writers  
|     writerSem   uint32 // semaphore for writers to wait for completing readers  
|     readerSem   uint32 // semaphore for readers to wait for completing writers  
|     readerCount int32 // number of pending readers  
|     readerWait  int32 // number of departing readers  
| }
```

In other words, `sync.RWMutex` is based on `sync.Mutex` with the necessary additions and improvements.

Now, let us talk about how `sync.RWMutex` improves `sync.Mutex`. Although only one function is allowed to perform write operations using a `sync.RWMutex` mutex, you can have multiple readers owning a `sync.RWMutex` mutex. However, there is one thing that you should be aware of: until all of the readers of a `sync.RWMutex` mutex unlock that mutex, you cannot lock it for writing, which is the small price you have to pay for allowing multiple readers.

The functions that can help you to work with a `sync.RWMutex` mutex are `RLock()` and `RUnlock()`, which are used for locking and unlocking the mutex for reading purposes, respectively. The `Lock()` and `Unlock()` functions used in a `sync.Mutex` mutex should still be used when you want to lock and unlock a `sync.RWMutex` mutex for writing purposes. Hence, an `RLock()` function call that locks for reading purposes should be paired with an `RUnlock()` function call. Finally, it should be apparent that you should not make changes to any shared variables inside an `RLock()` and `RUnlock()` block of code.

The Go code found in `rwmutex.go` illustrates the use and usefulness of the `sync.RWMutex` type. The program will be presented in six parts, and it contains two slightly different versions of the same function. The first one uses a

`sync.RWMutex` mutex for reading, and the second one uses a `sync.Mutex` mutex for reading. The performance difference between these two functions will help you to better understand the benefits of the `sync.RWMutex` mutex when used for reading purposes.

The first part of `rwMutex.go` contains the following Go code:

```
package main

import (
    "fmt"
    "os"
    "sync"
    "time"
)

var Password = secret{password: "myPassword"}

type secret struct {
    RWM      sync.RWMutex
    M        sync.Mutex
    password string
}
```

The `secret` structure holds a shared variable, a `sync.RWMutex` mutex, and a `sync.Mutex` mutex.

The second code portion of `rwMutex.go` is shown in the following code:

```
func Change(c *secret, pass string) {
    c.RWM.Lock()
    fmt.Println("LChange")
    time.Sleep(10 * time.Second)
    c.password = pass
    c.RWM.Unlock()
}
```

The `Change()` function modifies a shared variable, which means that you need to use an exclusive lock, which is the reason for using the `Lock()` and `Unlock()` functions. You cannot avoid using exclusive locks when changing things!

The third part of `rwMutex.go` is as follows:

```
func show(c *secret) string {
    c.RWM.RLock()
    fmt.Print("show")
    time.Sleep(3 * time.Second)
    defer c.RWM.RUnlock()
```

```

    |     return c.password
  }
```

The `show()` function uses the `RLock()` and `RUnlock()` functions because its critical section is used for reading a shared variable. So, although many goroutines can read the shared variable, no one can change it without using the `Lock()` and `Unlock()` functions. However, the `Lock()` function will be blocked for as long as there is someone reading that shared variable using the mutex.

The fourth code segment of `rwMutex.go` contains the following Go code:

```

func showWithLock(c *secret) string {
    c.RWM.Lock()
    fmt.Println("showWithLock")
    time.Sleep(3 * time.Second)
    defer c.RWM.Unlock()
    return c.password
}
```

The only difference between the code of the `showWithLock()` function and the code of the `show()` function is that the `showWithLock()` function uses an exclusive lock for reading, which means that only one `showWithLock()` function can read the `password` field of the `secret` structure.

The fifth part of `rwMutex.go` contains the following Go code:

```

func main() {
    var showFunction = func(c *secret) string { return "" }
    if len(os.Args) != 2 {
        fmt.Println("Using sync.RWMutex!")
        showFunction = show
    } else {
        fmt.Println("Using sync.Mutex!")
        showFunction = showWithLock
    }

    var waitGroup sync.WaitGroup
    fmt.Println("Pass:", showFunction(&Password))
```

The remaining code of `rwMutex.go` follows:

```

for i := 0; i < 15; i++ {
    waitGroup.Add(1)
    go func() {
        defer waitGroup.Done()
        fmt.Println("Go Pass:", showFunction(&Password))
    }()
}
```

```
    }

    go func() {
        waitGroup.Add(1)
        defer waitGroup.Done()
        Change(&Password, "123456")
    }()
}

waitGroup.Wait()
fmt.Println("Pass:", showFunction(&Password))
}
```

Executing `rwMutex.go` twice and using the `time(1)` command-line utility to benchmark the two versions of the program will generate the following type of output:

```
$ time go run rwMutex.go 10 >/dev/null

real      0m51.206s
user      0m0.130s
sys       0m0.074s
$ time go run rwMutex.go >/dev/null

real      0m22.191s
user      0m0.135s
sys       0m0.071s
```

Note that `> /dev/null` at the end of the preceding commands is for omitting the output of the two commands. Hence, the version that uses the `sync.RWMutex` mutex is much faster than the version that uses `sync.Mutex`.

The atomic package

An **atomic operation** is an operation that is completed in a single step relative to other threads or, in this case, to other goroutines. This means that an atomic operation cannot be interrupted in the middle of it.

The Go standard library offers the `atomic` package, which, in some cases, can help you to avoid using a mutex. However, mutexes are more versatile than atomic operations. Using the `atomic` package, you can have atomic counters accessed by multiple goroutines without synchronization issues and race conditions.

Notice that, if you have an atomic variable, all reading and writing must be done using the atomic functions provided by the `atomic` package. The use of the `atomic` package will be illustrated with the `atom.go` program, which will be presented in three parts.

The first part of `atom.go` is as follows:

```
package main

import (
    "flag"
    "fmt"
    "sync"
    "sync/atomic"
)

type atomCounter struct {
    val int64
}

func (c *atomCounter) Value() int64 {
    return atomic.LoadInt64(&c.val)
}
```

The second part of `atom.go` contains the following code:

```
func main() {
    minusX := flag.Int("x", 100, "Goroutines")
    minusY := flag.Int("y", 200, "Value")
    flag.Parse()
    X := *minusX
```

```
Y := *minusY

var waitGroup sync.WaitGroup
counter := atomCounter{}
```

The last part of `atom.go` is as follows:

```
for i := 0; i < X; i++ {
    waitGroup.Add(1)
    go func(no int) {
        defer waitGroup.Done()
        for i := 0; i < Y; i++ {
            atomic.AddInt64(&counter.val, 1)
        }
    }(i)
}

waitGroup.Wait()
fmt.Println(counter.Value())
}
```

The desired variable changes using `atomic.AddInt64()`.

Executing `atom.go` will generate the following output:

```
$ go run atom.go
20000
$ go run atom.go -x 4000 -y 10
40000
```

The output of `atom.go` proves that the counter used in the program is safe. It would be a very interesting exercise to modify the `atom.go` program in order to modify the counter variable using regular arithmetic (`counter.val++`) instead of `atomic.AddInt64()`. In that case, the output of the program would have been similar to the following:

```
$ go run atom.go -x 4000 -y 10
37613
$ go run atom.go
15247
```

The previous output shows that the use of `counter.val++` makes the program thread unsafe.

In [Chapter 12](#), *The Foundations of Network Programming in Go*, you are going to see a similar example using an HTTP server written in Go.

Sharing memory using goroutines

The last subsection of this topic illustrates how you can share data using a dedicated goroutine. Although shared memory is the traditional way that threads communicate with each other, Go comes with built-in synchronization features that allow a single goroutine to own a shared piece of data. This means that other goroutines must send messages to this single goroutine that owns the shared data, which prevents the corruption of the data. Such a goroutine is called a **monitor goroutine**. In Go terminology, this is *sharing by communicating instead of communicating by sharing*.

The technique will be illustrated using the `monitor.go` source file, which will be presented in five parts. The `monitor.go` program generates random numbers using a monitor goroutine.

The first part of `monitor.go` is as follows:

```
package main

import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "sync"
    "time"
)

var readValue = make(chan int)
var writeValue = make(chan int)
```

The `readValue` channel is used for reading random numbers, whereas the `writeValue` channel is used for getting new random numbers.

The second code portion of `monitor.go` is shown in the following code:

```
func set(newValue int) {
    writeValue <- newValue
}

func read() int {
    return <-readValue
}
```

The purpose of the `set()` function is to set the value of the shared variable, whereas the purpose of the `read()` function is to read the value of the saved variable.

The third code segment of the `monitor.go` program is as follows:

```
func monitor() {
    var value int
    for {
        select {
        case newValue := <-writeValue:
            value = newValue
            fmt.Printf("%d ", value)
        case readValue <- value:
        }
    }
}
```

All of the logic of the program can be found in the implementation of the `monitor()` function. More specifically, the `select` statement orchestrates the operation of the entire program.

When you have a read request, the `read()` function attempts to read from the `readValue` channel, which is controlled by the `monitor()` function. This returns the current value that is kept in the `value` variable. On the other hand, when you want to change the stored value, you call `set()`. This writes to the `writeValue` channel, which is also handled by the `select` statement. As a result, no one can deal with the `value` shared variable without using the `monitor()` function.

The fourth code segment of `monitor.go` is as follows:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Println("Please give an integer!")
        return
    }
    n, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Printf("Going to create %d random numbers.\n", n)
    rand.Seed(time.Now().Unix())
    go monitor()
```

The last part of `monitor.go` contains the following Go code:

```
var w sync.WaitGroup

for r := 0; r < n; r++ {
    w.Add(1)
    go func() {
        defer w.Done()
        set(rand.Intn(10 * n))
    }()
}
w.Wait()
fmt.Printf("\nLast value: %d\n", read())
}
```

Executing `monitor.go` generates the following output:

```
$ go run monitor.go 20

Going to create 20 random numbers.
89 88 166 42 149 89 20 84 44 178 184 28 52 121 62 91 31 117 140 106
Last value: 106
$ go run monitor.go 10
Going to create 10 random numbers.
30 16 66 70 65 45 31 57 62 26
Last value: 26
```



Personally, I prefer to use a monitor goroutine instead of traditional shared memory techniques because the implementation that uses the monitor goroutine is safer, closer to the Go philosophy, and much cleaner.

Revisiting the go statement

Although goroutines are fast and you can execute thousands of goroutines on your machine, this comes at a price. In this section, we are going to talk about the `go` statement, its behavior, and what happens when you start new goroutines in your Go programs.

Notice that closed variables in goroutines are evaluated when the goroutine actually runs and when the `go` statement is executed in order to create a new goroutine. This means that closed variables are going to be replaced by their values when the Go scheduler decides to execute the relevant code. This is illustrated in the following Go code, which is saved as `cloGo.go`:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    for i := 0; i <= 20; i++ {
        go func() {
            fmt.Println(i, " ")
        }()
    }
    time.Sleep(time.Second)
    fmt.Println()
}
```

It would be a good idea to stop reading for a moment and try to guess what the output of that code will be.

Executing `cloGo.go` multiple times will reveal the problem we were talking about:

```
$ go run cloGo.go
9 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21
$ go run cloGo.go
4 21 21 21 21 21 21 21 6 21 21 21 21 21 21 21 21 21 21 21 21
$ go run cloGo.go
6 21 6 6 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 6 21
```

The program mostly prints the number `21`, which is the last value of the variable of the `for` loop and not the other numbers. As `i` is a **closed**
variable, it is evaluated at the time of execution. As the goroutines begin but wait for the Go scheduler to allow them to get executed, the `for` loop ends, so the value of `i` that is being used is `21`. Lastly, the same issue also applies to Go channels, so be careful.

There is a pretty funny and unexpected way to solve this problem and it involves some idiomatic Go. The solution can be seen in `cloGoCorrect.go`, which contains the following code:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    for i := 0; i <= 20; i++ {
        i := i
        go func() {
            fmt.Println(i, " ")
        }()
    }
    time.Sleep(time.Second)
    fmt.Println()
}
```

The valid yet bizarre `i := i` statement creates a new instance of the variable for the goroutine, which makes the output of `cloGoCorrect.go` similar to the following:

```
$ go run cloGoCorrect.go
1 5 4 3 6 0 13 7 8 9 10 11 12 17 14 15 16 19 18 20 2
$ go run cloGoCorrect.go
5 2 20 13 6 7 1 9 10 11 0 3 17 14 15 16 4 19 18 8 12
```

You are now going to see another strange case that uses the `go` statement. Look at the following Go code, which is saved as `endlessComp.go`:

```
package main

import (
    "fmt"
    "runtime"
)
```

```
func main() {
    var i byte
    go func() {
        for i = 0; i <= 255; i++ {
        }
    }()
    fmt.Println("Leaving goroutine!")
    runtime.Gosched()
    runtime.GC()

    fmt.Println("Good bye!")
}
```

Executing `endlessComp.go` will surprise you as the program never ends because it blocks indefinitely and therefore has to be stopped manually:

```
$ go run endlessComp.go
Leaving goroutine!
^Csignal: interrupt
```

As you might have guessed, the root of the problem is related to the Go garbage collector and the way it works. The call to the `runtime.Gosched()` function asks the scheduler to execute another goroutine and then we invoke the Go garbage collector, which is trying to do its job.

First, the garbage collector needs all goroutines to go to sleep before doing its job. The problem is that the `for` loop will never end because the type of the `for` loop variable is `byte`. This means that the `for` loop prevents the system from doing anything else because the goroutine with that `for` loop will never go to sleep. This unfortunate event will happen even if your machine has multiple cores.

Notice that if the `for` loop was not empty, then the program would have been executed and ended just fine because the garbage collector would have had a place to stop.

Lastly, keep in mind that goroutines need to be explicitly signaled in order to end. You can easily end a goroutine using a simple `return` statement.

Catching race conditions

A **data race condition** is a situation where two or more running elements, such as threads and goroutines, try to take control of or modify a shared resource or a variable of a program. Strictly speaking, a data race occurs when two or more instructions access the same memory address, where at least one of them performs a write operation. If all operations are read operations, then there is no race condition.

Using the `-race` flag when running or building a Go source file will turn on the Go **race detector**, which will make the compiler create a modified version of a typical executable file. This modified version can record all access to shared variables as well as all synchronization events that take place, including calls to `sync.Mutex` and `sync.WaitGroup`. After analyzing the relevant events, the race detector prints a report that can help you to identify potential problems so that you can correct them.

Look at the following Go code, which is saved as `raceC.go`. This program is presented in three parts.

The first part of `raceC.go` is as follows:

```
package main

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

func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Give me a natural number!")
        return
    }
    numGR, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

The second part of `raceC.go` contains the following Go code:

```
var waitGroup sync.WaitGroup
var i int

k := make(map[int]int)
k[1] = 12

for i = 0; i < numGR; i++ {
    waitGroup.Add(1)
    go func() {
        defer waitGroup.Done()
        k[i] = i
    }()
}
```

The remaining Go code of `raceC.go` is as follows:

```
    k[2] = 10
    waitGroup.Wait()
    fmt.Printf("k = %#v\n", k)
}
```

As if it was not enough that many goroutines accessed the `k` map at the same time, I added another statement that accesses the `k` map before calling the `sync.Wait()` function.

If you execute `raceC.go`, you will get the following type of output without any warning or error messages:

```
$ go run raceC.go 10
k = map[int]int{7:10, 2:10, 10:10, 1:12}
```

```

$ go run raceC.go 10
k = map[int]int{2:10, 10:10, 1:12, 8:8, 9:9}
$ go run raceC.go 10
k = map[int]int{10:10, 1:12, 6:7, 7:7, 2:10}

```

If you execute `raceC.go` only once, then everything will look normal despite the fact that you do not get what you would expect when printing the contents of the `k` map. However, executing `raceC.go` multiple times tells us that there is something wrong here, mainly because each execution generates a different output.

There are many more things that we can get from `raceC.go` and its unexpected output – if we decide to use the Go race detector to analyze it:

```

$ go run -race raceC.go 10
=====
WARNING: DATA RACE
Read at 0x00c00001a0a8 by goroutine 6:
  main.main.func1()
    /Users/mtsouk/Desktop/mGo2nd/raceC.go:32 +0x66

Previous write at 0x00c00001a0a8 by main goroutine:
  main.main()
    /Users/mtsouk/Desktop/mGo2nd/raceC.go:28 +0x23f

Goroutine 6 (running) created at:
  main.main()
    /Users/mtsouk/Desktop/mGo2nd/raceC.go:30 +0x215
=====

=====
WARNING: DATA RACE
Write at 0x00c0000bc000 by goroutine 7:
  runtime.mapassign_fast64()
    /usr/local/Cellar/go/1.12.3/libexec/src/runtime/map_fast64.go:92 +0x0
  main.main.func1()
    /Users/mtsouk/Desktop/mGo2nd/raceC.go:32 +0x8d

Previous write at 0x00c0000bc000 by goroutine 6:
  runtime.mapassign_fast64()
    /usr/local/Cellar/go/1.12.3/libexec/src/runtime/map_fast64.go:92 +0x0
  main.main.func1()
    /Users/mtsouk/Desktop/mGo2nd/raceC.go:32 +0x8d

Goroutine 7 (running) created at:
  main.main()
    /Users/mtsouk/Desktop/mGo2nd/raceC.go:30 +0x215

Goroutine 6 (finished) created at:
  main.main()
    /Users/mtsouk/Desktop/mGo2nd/raceC.go:30 +0x215
=====

=====
WARNING: DATA RACE
Write at 0x00c0000bc000 by goroutine 8:
  runtime.mapassign_fast64()
    /usr/local/Cellar/go/1.12.3/libexec/src/runtime/map_fast64.go:92 +0x0
  main.main.func1()
    /Users/mtsouk/Desktop/mGo2nd/raceC.go:32 +0x8d

Previous write at 0x00c0000bc000 by goroutine 6:
  runtime.mapassign_fast64()
    /usr/local/Cellar/go/1.12.3/libexec/src/runtime/map_fast64.go:92 +0x0
  main.main.func1()
    /Users/mtsouk/Desktop/mGo2nd/raceC.go:32 +0x8d

Goroutine 8 (running) created at:
  main.main()
    /Users/mtsouk/Desktop/mGo2nd/raceC.go:30 +0x215

Goroutine 6 (finished) created at:
  main.main()
    /Users/mtsouk/Desktop/mGo2nd/raceC.go:30 +0x215
=====

k = map[int]int{1:1, 2:10, 3:3, 4:4, 5:5, 6:6, 7:7, 8:8, 9:9, 10:10}
Found 3 data race(s)
exit status 66

```

So, the race detector found three data races. Each one begins with the `WARNING: DATA RACE` message in its output.

The first data race happens inside `main.main.func1()` at line 32, which is called in line 28 by the `for` loop, which is called by a goroutine created at line 30. The problem here is signified by the `Previous write` message. After

examining the related code, it is easy to see that the actual problem is that the anonymous function takes no parameters, which means that the value of `i` that is used in the `for` loop cannot be deterministically discerned, as it keeps changing due to the `for` loop, which is a write operation.

The message of the second data race is `Write at 0x00c0000bc000 by goroutine 7.` If you read the relevant output, you will see that the data race is related to a write operation, and it happens on a Go map at line 32 by at least two goroutines that started at line 30. As the two goroutines have the same name (`main.main.func1()`), this is an indication that we are talking about the same goroutine. Two goroutines trying to modify the same variable is a data race condition. The third data race is similar to the second one.



The `main.main.func1()` notation is used by Go in order to name an anonymous function internally. If you have different anonymous functions, their names will be different as well.

You might ask, "What can I do now in order to correct the problems coming from the two data races?"

Well, you can rewrite the `main()` function of `raceC.go`, as follows:

```
func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Give me a natural number!")
        return
    }
    numGR, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }

    var waitGroup sync.WaitGroup
    var i int

    k := make(map[int]int)
    k[1] = 12

    for i = 0; i < numGR; i++ {
        waitGroup.Add(1)
        go func(j int) {
            defer waitGroup.Done()
            aMutex.Lock()
            k[j] = j
            aMutex.Unlock()
        }(i)
    }

    waitGroup.Wait()
    k[2] = 10
    fmt.Printf("k = %#v\n", k)
}
```

The `aMutex` variable is a global `sync.Mutex` variable defined outside the `main()` function that is accessible from everywhere in the program. Although this is not required, having such a global variable can save you from having to pass it to your functions all of the time.

Saving the new version of `raceC.go` as `noRaceC.go` and executing it will generate the expected output:

```
$ go run noRaceC.go 10
k = map[int]int{1:1, 0:0, 5:5, 3:3, 6:6, 9:9, 2:10, 4:4, 7:7, 8:8}
```

Processing `noRaceC.go` with the Go race detector will generate the following output:

```
$ go run -race noRaceC.go 10
k = map[int]int{5:5, 7:7, 9:9, 1:1, 0:0, 4:4, 6:6, 8:8, 2:10, 3:3}
```

Note that you need to use a locking mechanism while accessing the `k` map. If you do not use such a mechanism and just change the implementation of the anonymous function that is executed as a goroutine, you will get the following output from `go run noRaceC.go`:

```
$ go run noRaceC.go 10
fatal error: concurrent map writes

goroutine 10 [running]:
runtime.throw(0x10ca0bd, 0x15)
    /usr/local/Cellar/go/1.9.3/libexec/src/runtime/panic.go:605 +0x95 fp=0xc420024738 sp=0xc420024718 pc=0x10276b5
runtime.mapassign_fast64(0x10ae680, 0xc420074180, 0x5, 0x0)
    /usr/local/Cellar/go/1.9.3/libexec/src/runtime/hashmap_fast.go:607 +0x3d2 fp=0xc420024798 sp=0xc420024738 pc=0x100b582
main.main.func1(0xc420010090, 0xc420074180, 0x5)
    /Users/mtsouk/ch10/code/noRaceC.go:35 +0x6b fp=0xc4200247c8 sp=0xc420024798 pc=0x1096f5b
runtime.goexit()
    /usr/local/Cellar/go/1.9.3/libexec/src/runtime/asm_amd64.s:2337 +0x1 fp=0xc4200247d0 sp=0xc4200247c8 pc=0x1050c21
created by main.main
    /Users/mtsouk/ch10/code/noRaceC.go:32 +0x15a

goroutine 1 [semacquire]:
sync.runtime_Semacquire(0xc42001009c)
    /usr/local/Cellar/go/1.9.3/libexec/src/runtime/sema.go:56 +0x39
sync.(*WaitGroup).Wait(0xc420010090)
    /usr/local/Cellar/go/1.9.3/libexec/src/sync/waitgroup.go:131 +0x72
main.main()
    /Users/mtsouk/ch10/code/noRaceC.go:40 +0x17a

goroutine 12 [runnable]:
sync.(*WaitGroup).Done(0xc420010090)
    /usr/local/Cellar/go/1.9.3/libexec/src/sync/waitgroup.go:99 +0x43
main.main.func1(0xc420010090, 0xc420074180, 0x7)
    /Users/mtsouk/ch10/code/noRaceC.go:37 +0x79
created by main.main
    /Users/mtsouk/ch10/code/noRaceC.go:32 +0x15a
exit status 2
```

The root of the problem can be seen clearly: concurrent map writes.

The context package

The main purpose of the `context` package is to define the `Context` type and support **cancellation**. Yes, you heard that right; there are times when, for some reason, you want to abandon what you are doing. However, it would be very helpful to be able to include some extra information about your cancellation decisions. The `context` package allows you to do exactly that.

If you take a look at the source code of the `context` package, you will realize that its implementation is pretty simple – even the implementation of the `Context` type is pretty simple, yet the `context` package is very important.



The `context` package existed for a while as an external Go package; it first appeared as a standard Go package in Go version 1.7. So, if you have an older Go version, you will not be able to follow this section without first downloading the `context` package or installing a newer Go version.

The `Context` type is an interface with four methods named `Deadline()`, `Done()`, `Err()`, and `Value()`. The good news is that you do not need to implement all of these functions of the `Context` interface – you just need to modify a `Context` variable using functions such as `context.WithCancel()`, `context.WithDeadline()`, and `context.WithTimeout()`.

All three of these functions return a derived `Context` (the child) and a `cancelFunc` function. Calling the `cancelFunc` function removes the parent's reference to the child and stops any associated timers. This means that the Go garbage collector is free to garbage collect the child goroutines that no longer have associated parent goroutines. For garbage collection to work correctly, the parent goroutine needs to keep a reference to each child goroutine. If a child goroutine ends without the parent knowing about it, then a memory leak occurs until the parent is canceled as well.

The following is a simple use of the `context` package using the Go code of the `simpleContext.go` file, which will be presented in six parts.

The first code segment of `simpleContext.go` contains the following code:

```
package main

import (
    "context"
    "fmt"
    "os"
    "strconv"
    "time"
)
```

The second part of `simpleContext.go` is as follows:

```
func f1(t int) {
    c1 := context.Background()
    c1, cancel := context.WithCancel(c1)
    defer cancel()

    go func() {
        time.Sleep(4 * time.Second)
        cancel()
    }()
}
```

The `f1()` function requires just one parameter, which is the time delay, because everything else is defined inside the function. Notice that the type of the `cancel` variable is `context.CancelFunc`.

You need to call the `context.Background()` function in order to initialize an empty `Context`. The `context.WithCancel()` function uses an existing `Context` and creates a child of it with cancellation. The `context.WithCancel()` function also creates a `Done` channel that can be closed, either when the `cancel()` function is called, as shown in the preceding code, or when the `Done` channel of the parent context is closed.

The third code portion of `simpleContext.go` contains the rest of the code of the `f1()` function:

```
select {
case <-c1.Done():
    fmt.Println("f1():", c1.Err())
    return
case r := <-time.After(time.Duration(t) * time.Second):
    fmt.Println("f1():", r)
}
return
}
```

Here, you see the use of the `Done()` function of a `Context` variable. When this function is called, you have a cancellation. The return value of `Context.Done()` is a channel because, otherwise, you would have not been able to use it in a `select` statement.

The fourth part of `simpleContext.go` contains the following Go code:

```
func f2(t int) {
    c2 := context.Background()
    c2, cancel := context.WithTimeout(c2, time.Duration(t)*time.Second)
    defer cancel()

    go func() {
        time.Sleep(4 * time.Second)
        cancel()
    }()

    select {
    case <-c2.Done():
        fmt.Println("f2():", c2.Err())
        return
    case r := <-time.After(time.Duration(t) * time.Second):
        fmt.Println("f2():", r)
    }
    return
}
```

This part showcases the use of the `context.WithTimeout()` function, which requires two parameters: a `Context` parameter and a `time.Duration` parameter. When the timeout period expires, the `cancel()` function is automatically called.

The fifth part of `simpleContext.go` is as follows:

```
func f3(t int) {
    c3 := context.Background()
    deadline := time.Now().Add(time.Duration(2*t) * time.Second)
    c3, cancel := context.WithDeadline(c3, deadline)
    defer cancel()

    go func() {
        time.Sleep(4 * time.Second)
        cancel()
    }()

    select {
    case <-c3.Done():
        fmt.Println("f3():", c3.Err())
        return
    case r := <-time.After(time.Duration(t) * time.Second):
        fmt.Println("f3():", r)
    }
}
```

```
|     return  
| }
```

The preceding Go code illustrates the use of the `context.WithDeadline()` function, which requires two parameters: a `Context` variable and a time in the future that signifies the deadline of the operation. When the deadline passes, the `cancel()` function is automatically called.

The remaining Go code of `simpleContext.go` is as follows:

```
func main() {  
    if len(os.Args) != 2 {  
        fmt.Println("Need a delay!")  
        return  
    }  
  
    delay, err := strconv.Atoi(os.Args[1])  
    if err != nil {  
        fmt.Println(err)  
        return  
    }  
    fmt.Println("Delay:", delay)  
  
    f1(delay)  
    f2(delay)  
    f3(delay)  
}
```

The purpose of the `main()` function is to initialize things.

Executing `simpleContext.go` will generate the following type of output:

```
$ go run simpleContext.go 4  
Delay: 4  
f1(): context canceled  
f2(): 2019-04-11 18:18:43.327345 +0300 EEST m==+8.004588898  
f3(): 2019-04-11 18:18:47.328073 +0300 EEST m==+12.005483099  
$ go run simpleContext.go 2  
Delay: 2  
f1(): 2019-04-11 18:18:53.972045 +0300 EEST m==+2.005231943  
f2(): context deadline exceeded  
f3(): 2019-04-11 18:18:57.974337 +0300 EEST m==+6.007690061  
$ go run simpleContext.go 10  
Delay: 10  
f1(): context canceled  
f2(): context canceled  
f3(): context canceled
```

The long lines of the output are the return values of the `time.After()` function calls. They denote the normal operation of the program. The point here is

that the operation of the program is canceled when there are delays in its execution.

This is as simple as it gets with the use of the `context` package, as the code presented did not do any serious work with the `Context` interface. However, the Go code included in the next section will present a more realistic example.

An advanced example of the context package

The functionality of the `context` package will be illustrated much better and in greater depth by using the Go code of the `useContext.go` program, which is presented in five parts. In this example, you will create an HTTP client that does not want to wait too long for the response of the HTTP server, which is not an unusual scenario. In fact, as almost all HTTP clients support such functionality, you will study another technique for timing out an HTTP request in [Chapter 12, *The Foundations of Network Programming in Go*](#).

The `useContext.go` program requires two command-line arguments: the URL of the server to which it is going to connect and the time for which the presented utility should wait. If the program has only one command-line argument, then the delay will be five seconds.

The first code segment of `useContext.go` follows:

```
package main

import (
    "context"
    "fmt"
    "io/ioutil"
    "net/http"
    "os"
    "strconv"
    "sync"
    "time"
)

var (
    myUrl string
    delay int = 5
    w      sync.WaitGroup
)

type myData struct {
    r    *http.Response
    err error
}
```

Both `myURL` and `delay` are global variables, so they can be accessed by anything in the code. Additionally, there is a `sync.WaitGroup` variable named `w`, which also has a global scope, and the definition of a structure named `myData` for keeping together the response of the web server, along with an `error` variable in case there is an error somewhere. The second part of `useContext.go` is shown in the following Go code:

```
func connect(c context.Context) error {
    defer w.Done()
    data := make(chan myData, 1)

    tr := &http.Transport{}
    httpClient := &http.Client{Transport: tr}

    req, _ := http.NewRequest("GET", myUrl, nil)
```

The preceding Go code deals with the HTTP connection.



You will learn more about developing HTTP servers and clients in Go in [Chapter 12, The Foundations of Network Programming in Go](#).

The third code portion of `useContext.go` contains the following Go code:

```
go func() {
    response, err := httpClient.Do(req)
    if err != nil {
        fmt.Println(err)
        data <- myData{nil, err}
        return
    } else {
        pack := myData{response, err}
        data <- pack
    }
}()
```

The fourth code segment of `useContext.go` is shown in the following Go code:

```
select {
case <-c.Done():
    tr.CancelRequest(req)
    <-data
    fmt.Println("The request was cancelled!")
    return c.Err()
case ok := <-data:
    err := ok.err
    resp := ok.r
    if err != nil {
        fmt.Println("Error select:", err)
        return err
    }
    defer resp.Body.Close()
```

```

    realHTTPData, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        fmt.Println("Error select:", err)
        return err
    }
    fmt.Printf("Server Response: %s\n", realHTTPData)

}
return nil
}

```

The remaining code of `useContext.go`, which is the implementation of the `main()` function, is as follows:

```

func main() {
    if len(os.Args) == 1 {
        fmt.Println("Need a URL and a delay!")
        return
    }

    myUrl = os.Args[1]
    if len(os.Args) == 3 {
        t, err := strconv.Atoi(os.Args[2])
        if err != nil {
            fmt.Println(err)
            return
        }
        delay = t
    }

    fmt.Println("Delay:", delay)
    c := context.Background()
    c, cancel := context.WithTimeout(c, time.Duration(delay)*time.Second)
    defer cancel()

    fmt.Printf("Connecting to %s \n", myUrl)
    w.Add(1)
    go connect(c)
    w.Wait()
    fmt.Println("Exiting...")
}

```

The timeout period is defined by the `context.WithTimeout()` method. The `connect()` function that is executed as a goroutine will either terminate normally or when the `cancel()` function is executed. Notice that it is considered good practice to use `context.Background()` in the `main()` function or the `init()` function of a package or in tests.

Although it is not necessary to know about the server side of the operation, it is good to see how a Go version of a web server can be slow in a random way. In this case, a random number generator decides how slow your web

server will be – real web servers might be too busy to answer or there might be network issues that cause the delay. The name of the source file is `slowWWW.go`, and its contents are as follows:

```
package main

import (
    "fmt"
    "math/rand"
    "net/http"
    "os"
    "time"
)

func random(min, max int) int {
    return rand.Intn(max-min) + min
}

func myHandler(w http.ResponseWriter, r *http.Request) {
    delay := random(0, 15)
    time.Sleep(time.Duration(delay) * time.Second)

    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Fprintf(w, "Delay: %d\n", delay)
    fmt.Printf("Served: %s\n", r.Host)
}

func main() {
    seed := time.Now().Unix()
    rand.Seed(seed)

    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Using default port number: ", PORT)
    } else {
        PORT = ":" + arguments[1]
    }

    http.HandleFunc("/", myHandler)
    err := http.ListenAndServe(PORT, nil)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

As you can see, you do not need to use the `context` package in the `slowWWW.go` file because it is the job of the web client to decide how much time it can wait for a response.

The code of the `myHandler()` function is responsible for the slowness of the web server program. The delay can be from 0 seconds to 14 seconds, as

introduced by the `random(0, 15)` function call.

If you try to use the `slowWWW.go` web server using a tool such as `wget(1)`, you will receive the following type of output:

```
$ wget -qO- http://localhost:8001/
Serving: /
Delay: 4
$ wget -qO- http://localhost:8001/
Serving: /
Delay: 13
```

This happens because the default timeout value of `wget(1)` is larger. Executing `useContext.go` while `slowWWW.go` is already running in another UNIX shell will create the following type of output when processed with the handy `time(1)` utility:

```
$ time go run useContext.go http://localhost:8001/ 1
Delay: 1
Connecting to http://localhost:8001/
Get http://localhost:8001/: net/http: request canceled
The request was cancelled!
Exiting...

real      0m1.374s
user      0m0.304s
sys       0m0.117s
$ time go run useContext.go http://localhost:8001/ 10
Delay: 10
Connecting to http://localhost:8001/
Get http://localhost:8001/: net/http: request canceled
The request was cancelled!
Exiting...

real      0m10.381s
user      0m0.314s
sys       0m0.125s
$ time go run useContext.go http://localhost:8001/ 15
Delay: 15
Connecting to http://localhost:8001/
Server Response: Serving: /
Delay: 13

Exiting...

real      0m13.379s
user      0m0.309s
sys       0m0.118s
```

The output shows that only the third command actually got an answer from the HTTP server – the first two commands timed out.

Another example of the context package

In this section, you are going to learn even more about the `context` package because it is such a powerful and unique package of the Go standard library. This time, we are going to create a context that uses the `context.TODO()` function instead of the `context.Background()` function. Although both functions return a non-nil, empty `Context`, their purposes differ. We are also going to illustrate the use of the `context.WithValue()` function. The name of the program that will be developed in this section is `moreContext.go`, and it will be presented in four parts.

The first part of `moreContext.go` is as follows:

```
package main

import (
    "context"
    "fmt"
)
type aKey string
```

The second part of `moreContext.go` contains the following Go code:

```
func searchKey(ctx context.Context, k aKey) {
    v := ctx.Value(k)
    if v != nil {
        fmt.Println("found value:", v)
        return
    } else {
        fmt.Println("key not found:", k)
    }
}
```

This is a function that retrieves a value from a context and checks whether that value exists or not.

The third part of `moreContext.go` contains the following Go code:

```
func main() {
    myKey := aKey("mySecretValue")
    ctx := contextWithValue(context.Background(), myKey, "mySecretValue")

    searchKey(ctx, myKey)
```

The `contextWithValue()` function offers a way of associating a value with a Context.

Notice that contexts should be not stored in structures – they should be passed as separate parameters to functions. It is considered good practice to pass them as the first parameter of a function.

The last part of `moreContext.go` is as follows:

```
    searchKey(ctx, aKey("notThere"))
    emptyCtx := context.TODO()
    searchKey(emptyCtx, aKey("notThere"))

}
```

In this case, we declare that although we intend to use an operation context, we are not sure about it yet – this is signified by the use of the `context.TODO()` function. The good thing is that `TODO()` is recognized by static analysis tools, which allows them to determine whether Contexts are propagated correctly in a program or not.

Executing `moreContext.go` will generate the following output:

```
$ go run moreContext.go
found value: mySecretValue
key not found: notThere
key not found: notThere
```

Remember that you should never pass a `nil` context – use the `context.TODO()` function to create a suitable context – and remember that the `context.TODO()` function should be used when you are not sure about the `context` that you want to use.

Worker pools

Generally speaking, a **worker pool** is a set of threads that are about to process jobs assigned to them. The Apache web server and the `net/http` package of Go more or less work this way: the main process accepts all incoming requests, which are forwarded to the worker processes to get served. Once a worker process has finished its job, it is ready to serve a new client.

Nevertheless, there is a central difference here because our worker pool is going to use goroutines instead of threads. Additionally, threads do not usually die after serving a request because the cost of ending a thread and creating a new one is too high, whereas goroutines do die after finishing their job. As you will see shortly, worker pools in Go are implemented with the help of buffered channels, because they allow you to limit the number of goroutines running at the same time.

The next program, `workerPool.go`, will be presented in five parts. The program will implement a simple task: it will process integers and print their square values using a single goroutine to serve each request. Despite the simplicity of `workerPool.go`, the Go code of the program can be easily used as a template for implementing much more difficult tasks.



This is an advanced technique that can help you to create server processes in Go that can accept and serve multiple clients using goroutines.

The first part of `workerPool.go` follows:

```
package main

import (
    "fmt"
    "os"
    "strconv"
    "sync"
    "time"
)

type Client struct {
    id      int
```

```

    integer int
}

type Data struct {
    job    Client
    square int
}

```

Here, you can see a technique that uses the `Client` structure to assign a unique ID to each request that you are going to process. The `Data` structure is used to group the data of a `Client` with the actual results generated by the program. Put simply, the `Client` structure holds the input data of each request, whereas the `Data` structure holds the results of a request.

The second code portion of `workerPool.go` is shown in the following Go code:

```

var (
    size      = 10
    clients   = make(chan Client, size)
    data      = make(chan Data, size)
)

func worker(w *sync.WaitGroup) {
    for c := range clients {
        square := c.integer * c.integer
        output := Data{c, square}
        data <- output
        time.Sleep(time.Second)
    }
    w.Done()
}

```

The preceding code has two interesting parts. The first part creates three global variables. The `clients` and `data` buffered channels are used to get new client requests and write the results, respectively. If you want your program to run faster, you can increase the value of the `size` parameter.

The second part is the implementation of the `worker()` function, which reads the `clients` channel in order to get new requests to serve. Once the processing is complete, the result is written to the `data` channel. The delay that is introduced using the `time.Sleep(time.Second)` statement is not necessary, but it gives you a better sense of the way that the generated output will be printed.

Finally, remember to use a pointer for the `sync.WaitGroup` parameter in the `worker()` function because, otherwise, the `sync.WaitGroup` variable is copied,

which means that it will be useless.

The third part of `workerPool.go` contains the following Go code:

```
func makeWP(n int) {
    var w sync.WaitGroup
    for i := 0; i < n; i++ {
        w.Add(1)
        go worker(&w)
    }
    w.Wait()
    close(data)
}

func create(n int) {
    for i := 0; i < n; i++ {
        c := Client{i, i}
        clients <- c
    }
    close(clients)
}
```

The preceding code implements two functions, named `makeWP()` and `create()`. The purpose of the `makeWP()` function is to generate the required number of `worker()` goroutines to process all requests. Although the `w.Add(1)` function is called in `makeWP()`, `w.Done()` is called in the `worker()` function once a worker has finished its job. The purpose of the `create()` function is to create all requests properly using the `Client` type and then write them to the `clients` channel for processing. Note that the `clients` channel is read by the `worker()` function.

The fourth code segment of `workerPool.go` is as follows:

```
func main() {
    fmt.Println("Capacity of clients:", cap(clients))
    fmt.Println("Capacity of data:", cap(data))

    if len(os.Args) != 3 {
        fmt.Println("Need #jobs and #workers!")
        os.Exit(1)
    }

    nJobs, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }

    nWorkers, err := strconv.Atoi(os.Args[2])
    if err != nil {
        fmt.Println(err)
        return
    }
```

In the preceding code, you read your command-line parameters. First, however, you see that you can use the `cap()` function to find the capacity of a channel.

If the number of workers is greater than the size of the `clients` buffered channel, then the number of goroutines that are going to be created will be equal to the size of the `clients` channel. Similarly, if the number of jobs is greater than the number of workers, the jobs will be served in smaller sets.

The program allows you to define the number of workers and the number of jobs using its command-line arguments. However, in order to change the size of the `clients` and `data` channels, you will need to make changes to the source code of the program.

The remaining code of `workerPool.go` follows:

```
go create(nJobs)
finished := make(chan interface{})
go func() {
    for d := range data {
        fmt.Printf("Client ID: %d\tint: ", d.job.id)
        fmt.Printf("%d\tsquare: %d\n", d.job.integer, d.square)
    }
    finished <- true
}()

makeWP(nWorkers)
fmt.Printf(": %v\n", <-finished)
```

First, you call the `create()` function to mimic the client requests that you will have to process. An anonymous goroutine is used to read the `data` channel and print the output to the screen. The `finished` channel is used to block the program until the anonymous goroutine is done reading the `data` channel. Therefore, the `finished` channel needs no particular type.

Next, you call the `makeWP()` function to actually process the requests. The `<-finished` statement in `fmt.Printf()` blocks means that it does not allow the program to end until somebody writes something to the `finished` channel. That somebody is the anonymous goroutine of the `main()` function. Additionally, although the anonymous function writes the `true` value to the

`finished` channel, you could have written `false` to it and had the same result, which is unblocking the `main()` function. Try it on your own!

Executing `workerPool.go` will generate the following type of output:

```
$ go run workerPool.go 15 5
Capacity of clients: 10
Capacity of data: 10
Client ID: 0    int: 0    square: 0
Client ID: 4    int: 4    square: 16
Client ID: 1    int: 1    square: 1
Client ID: 3    int: 3    square: 9
Client ID: 2    int: 2    square: 4
Client ID: 5    int: 5    square: 25
Client ID: 6    int: 6    square: 36
Client ID: 7    int: 7    square: 49
Client ID: 8    int: 8    square: 64
Client ID: 9    int: 9    square: 81
Client ID: 10   int: 10   square: 100
Client ID: 11   int: 11   square: 121
Client ID: 12   int: 12   square: 144
Client ID: 13   int: 13   square: 169
Client ID: 14   int: 14   square: 196
: true
```

When you want to serve each individual request without expecting an answer from it in the `main()` function, as happened with `workerPool.go`, you have fewer things to worry about. A simple way to use goroutines for processing your requests and to get an answer from them in the `main()` function is by using shared memory or a monitor process that will collect the data instead of just printing it on the screen.

Finally, the work of the `workerPool.go` program is much simpler because the `worker()` function cannot fail. This will not be the case when you have to work over computer networks or with other kinds of resources that can fail.

Additional resources

The following are very useful resources:

- Visit the documentation page of the `sync` package, which can be found at <https://golang.org/pkg/sync/>.
- Visit the documentation page of the `context` package, which can be found at <https://golang.org/pkg/context/>.
- You can learn more about the implementation of the Go scheduler by visiting <https://golang.org/src/runtime/proc.go>.
- You can find the documentation page of the atomic package at <https://golang.org/pkg/sync/atomic/>.
- You can view the design document of the Go scheduler at <https://golang.org/s/gol11sched>.

Exercises

- Try to implement a concurrent version of `wc(1)` that uses a buffered channel.
- Next, try to implement a concurrent version of `wc(1)` that uses shared memory.
- Finally, try to implement a concurrent version of `wc(1)` that uses a monitor goroutine.
- Modify the Go code of `workerPool.go` in order to save the results in a file. Use a mutex and a critical section while dealing with the file or a monitor goroutine that will keep writing your data on the disk.
- What will happen to the `workerPool.go` program when the value of the `size` global variable becomes `1`? Why?
- Modify the Go code of `workerPool.go` in order to implement the functionality of the `wc(1)` command-line utility.
- Modify the Go code of `workerPool.go` so that the size of the `clients` and `data` buffered channels can be defined using command-line arguments.
- Try to write a concurrent version of the `find(1)` command-line utility that uses a monitor goroutine.
- Modify the code of `simpleContext.go` so that the anonymous function used in all `f1()`, `f2()`, and `f3()` functions becomes a separate function. What is the main challenge of this code change?
- Modify the Go code of `simpleContext.go` so that all `f1()`, `f2()`, and `f3()` functions use an externally-created `context` variable instead of defining their own.
- Modify the Go code of `useContext.go` in order to use either `context.WithDeadline()` or `context.WithCancel()` instead of `context.WithTimeout()`.
- Finally, try to implement a concurrent version of the `find(1)` command-line utility using a `sync.Mutex` mutex.

Summary

This chapter addressed many important topics related to goroutines and channels. Mainly, however, it clarified the power of the `select` statement. Due to the capabilities of the `select` statement, channels are the preferred Go way for interconnecting the components of a concurrent Go program that utilizes multiple goroutines. Additionally, the chapter demonstrated the use of the `context` standard Go package, which, when needed, is irreplaceable.

There are many rules in concurrent programming; however, the most important rule is that you should avoid sharing things unless you have a pretty important reason to do so! Shared data is the root of all nasty bugs in concurrent programming.

What you must remember from this chapter is that, although shared memory used to be the only way of exchanging data over the threads of the same process, Go offers better ways for goroutines to communicate with each other, so think in Go terms before deciding to use shared memory in your Go code. Nonetheless, if you really have to use shared memory, you might want to use a monitor goroutine instead.

The primary subjects of the next chapter will be code testing, code optimization, and code profiling with Go. Apart from these topics, you will learn about benchmarking Go code, cross-compilation, and finding unreachable Go code.

At the end of the next chapter, you will also learn how to document your Go code and how to generate HTML output using the `godoc` utility.

Code Testing, Optimization, and Profiling

The previous chapter discussed concurrency in Go, mutexes, the `atomic` package, the various types of channels, race conditions, and how the `select` statement allows you to use channels as glue to control goroutines and allow them to communicate.

The Go topics in this chapter are very practical and important, especially if you are interested in improving the performance of your Go programs and discovering bugs quickly. This chapter primarily addresses code optimization, code testing, code documentation, and code profiling.

Code optimization is a process where one or more developers try to make certain parts of a program run faster, be more efficient, or use fewer resources. Put simply, code optimization is about eliminating the bottlenecks of a program.

Code testing is about making sure that your code does what you want it to do. In this chapter, you will experience the Go way of testing code. The best time to write code to test your programs is during the development phase, as this can help to reveal bugs in your code as early as possible.

Code profiling relates to measuring certain aspects of a program in order to get a detailed understanding of the way the code works. The results of code profiling may help you to decide which parts of your code need to change.

I hope that you already recognize the importance of documenting your code in order to describe the decisions you made while developing the implementation of your program. In this chapter, you will see how Go can help you to generate documentation for the modules that you implement.



Documentation is so important that some developers write the documentation first and the code afterward! However, what is really important is that the documentation and

 *the functionality of the program say and do the same thing, respectively.*

In this chapter, you will learn about the following topics:

- Profiling Go code
- The `go tool pprof` utility
- Using the web interface of the Go profiler
- Testing Go code
- The `go test` command
- The `go tool trace` utility
- The handy `testing/quick` package
- Benchmarking Go code
- Cross-compilation
- Testing code coverage
- Generating documentation for your Go code
- Creating example functions
- Finding unreachable Go code in your programs

About optimization

Code optimization is both an art and a science. This means that there is no deterministic way to help you optimize your Go code, or any other code in any programming language, and that you should use your brain and try many things if you want to make your code run faster.



You should make sure that you optimize code that does not contain any bugs, because there is no point in optimizing a bug. If you have any bugs in your program, you should debug it first.

If you are really into code optimization, you might want to read *Compilers: Principles, Techniques, and Tools* by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman (Pearson Education Limited, 2014), which focuses on compiler construction. Additionally, all volumes in *The Art of Computer Programming* series by Donald Knuth (Addison-Wesley Professional, 1998) are great resources for all aspects of programming.

Always remember what Knuth said about optimization:

"The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming."

Also, remember what Joe Armstrong, one of the developers of Erlang, said about optimization:

"Make it work, then make it beautiful, then if you really, really have to, make it fast. 90 percent of the time, if you make it beautiful, it will already be fast. So really, just make it beautiful!"

Additionally, generally speaking, only a small percentage of a program needs to be optimized. In such cases, the assembly programming language, which can be used to implement certain Go functions, is a really good candidate and will have a huge impact on the performance of your programs.

Optimizing Go code

As mentioned, **code optimization** is the process where you try to discover the parts of your code that have a big impact on the performance of the entire program in order to make them run faster or use fewer resources.

The *benchmarking* section that appears later in this chapter will greatly help you to understand what is going on with your code behind the scenes and which parameters of your program have the greatest impact on the performance of your program. However, do not underestimate the importance of common sense. Put simply, if one of your functions is executed 10,000 times more than the rest of the functions of a program, try to optimize that function first.



The general advice for optimization is that you must optimize bug-free code only. This means that you must optimize working code only. Therefore, you should first try to write correct code even if that code is slow. Finally, the single most frequent mistake that programmers make is trying to optimize the first version of their code, which is the root of most bugs!

Again, code optimization is both an art and a science, which means that it is a pretty difficult task. The next section about profiling Go code will definitely help you with code optimization because the main purpose of profiling is to find the bottlenecks in your code in order to optimize the most important parts of your program.

Profiling Go code

Profiling is a process of dynamic program analysis that measures various values related to program execution in order to give you a better understanding of the behavior of your program. In this section, you are going to learn how to profile Go code in order to understand your code better and improve its performance. Sometimes, code profiling can even reveal bugs!

First, we are going to use the command-line interface of the Go profiler. Then, we will use the web interface of the Go profiler.

The single most important thing to remember is that if you want to profile Go code, you will need to import the `runtime/pprof` standard Go package, either directly or indirectly. You can find the help page of the `pprof` tool by executing the `go tool pprof -help` command, which will generate lots of output.

The net/http/pprof standard Go package

Although Go comes with the low-level `runtime/pprof` standard Go package, there is also the high-level `net/http/pprof` package, which should be used when you want to profile a web application written in Go. As this chapter will not talk about creating HTTP servers in Go, you will learn more about the `net/http/pprof` package in [chapter 12](#), *The Foundations of Network Programming in Go*.

A simple profiling example

Go supports two kinds of profiling: **CPU profiling** and **memory profiling**. It is not recommended that you profile an application for both kinds at the same time, because these two different kinds of profiling do not work well with each other. The `profileMe.go` application is an exception, however, because it is used to illustrate the two techniques.

The Go code to be profiled is saved as `profileMe.go`, and it will be presented in five parts. The first part of `profileMe.go` is shown in the following Go code:

```
package main
import (
    "fmt"
    "math"
    "os"
    "runtime"
    "runtime/pprof"
    "time"
)
func fibo1(n int) int64 {
    if n == 0 || n == 1 {
        return int64(n)
    }
    time.Sleep(time.Millisecond)
    return int64(fibo2(n-1)) + int64(fibo2(n-2))
}
```

Notice that it is compulsory to import the `runtime/pprof` package directly or indirectly for your program to create profiling data. The reason for calling `time.Sleep()` in the `fibo1()` function is to slow it down a bit. You will learn why near the end of this section.

The second code segment of `profileMe.go` follows:

```
func fibo2(n int) int {
    fn := make(map[int]int)
    for i := 0; i <= n; i++ {
        var f int
        if i <= 2 {
            f = 1
        } else {
            f = fn[i-1] + fn[i-2]
        }
        fn[i] = f
    }
    time.Sleep(50 * time.Millisecond)
    return fn[n]
}
```

The preceding code contains the implementation of another Go function that uses a different algorithm for calculating numbers of the Fibonacci sequence.

The third part of `profileMe.go` contains the following Go code:

```
func N1(n int) bool {
    k := math.Floor(float64(n/2 + 1))
    for i := 2; i < int(k); i++ {
        if (n % i) == 0 {
            return false
        }
    }
    return true
}

func N2(n int) bool {
    for i := 2; i < n; i++ {
        if (n % i) == 0 {
            return false
        }
    }
    return true
}
```

Both the `N1()` and `N2()` functions are used to find out whether a given integer is a prime number or not. The first function is optimized because its `for` loop iterates over approximately half the numbers used in the `for` loop of `N2()`.

As both functions are relatively slow, there is no need for a call to `time.Sleep()` here.

The fourth code segment of `profileMe.go` is as follows:

```
func main() {
    cpuFile, err := os.Create("/tmp/cpuProfile.out")
    if err != nil {
        fmt.Println(err)
        return
    }
    pprof.StartCPUProfile(cpuFile)
    defer pprof.StopCPUProfile()
    total := 0
    for i := 2; i < 100000; i++ {
        n := N1(i)
        if n {
            total = total + 1
        }
    }
    fmt.Println("Total primes:", total)
    total = 0
    for i := 2; i < 100000; i++ {
        n := N2(i)
        if n {
            total = total + 1
        }
    }
    fmt.Println("Total primes:", total)
    for i := 1; i < 90; i++ {
        n := fibol(i)
        fmt.Print(n, " ")
    }
    fmt.Println()
    for i := 1; i < 90; i++ {
        n := fibo2(i)
        fmt.Print(n, " ")
    }
    fmt.Println()
    runtime.GC()
```

The call to `os.Create()` is used to have a file to which to write the profiling data. The `pprof.StartCPUProfile()` call begins the **CPU profiling** of the program, and the call to `pprof.StopCPUProfile()` stops it.

If you want to create and use temporary files and directories multiple times, then you should have a look at `ioutil.TempFile()` and `ioutil.TempDir()` respectively.

The last part of `profileMe.go` follows:

```
// Memory profiling!
memory, err := os.Create("/tmp/memoryProfile.out")
if err != nil {
    fmt.Println(err)
    return
}
defer memory.Close()
for i := 0; i < 10; i++ {
    s := make([]byte, 50000000)
    if s == nil {
        fmt.Println("Operation failed!")
    }
    time.Sleep(50 * time.Millisecond)
}
err = pprof.WriteHeapProfile(memory)
if err != nil {
    fmt.Println(err)
    return
}
```

In the last part, you can see how the **memory profiling** technique works. It is pretty similar to CPU profiling, and once again, you will need a file to write out the profiling data.

Executing `profileMe.go` will generate the following output:

```
$ go run profileMe.go
Total primes: 9592
Total primes: 9592
1 2 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 8:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 8:
```

Apart from the output, the program will also collect the profiling data in two files:

```
$ cd /tmp
$ ls -l *Profile*
-rw-r--r-- 1 mtsouk wheel 1557 Apr 24 16:37 cpuProfile.out
-rw-r--r-- 1 mtsouk wheel 438 Apr 24 16:37 memoryProfile.out
```

It is only after collecting the profiling data that you can start to inspect it. Thus, you can now start the command-line profiler to examine the CPU data as follows:

```
$ go tool pprof /tmp/cpuProfile.out
Type: cpu
Time: Apr 24, 2019 at 4:37pm (EEST)
Duration: 19.59s, Total samples = 4.46s (22.77%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof)
```

Pressing `help` while in the profiler shell will generate the following output:

```
(pprof) help
Commands:
  callgrind      Outputs a graph in callgrind format
  comments       Output all profile comments
  disasm        Output assembly listings annotated with samples
  dot           Outputs a graph in DOT format
  eog           Visualize graph through eog
  evince        Visualize graph through evince
  gif            Outputs a graph image in GIF format
  gv             Visualize graph through gv
  kcachegrind   Visualize report in KCachegrind
  list          Output annotated source for functions matching regexp
  pdf           Outputs a graph in PDF format
  peek          Output callers/callees of functions matching regexp
  png           Outputs a graph image in PNG format
  proto         Outputs the profile in compressed protobuf format
  ps             Outputs a graph in PS format
  raw            Outputs a text representation of the raw profile
  svg            Outputs a graph in SVG format
  tags           Outputs all tags in the profile
  text           Outputs top entries in text form
  top            Outputs top entries in text form
  topproto       Outputs top entries in compressed protobuf format
  traces         Outputs all profile samples in text form
  tree           Outputs a text rendering of call graph
  web            Visualize graph through web browser
  weblist        Display annotated source in a web browser
  o/options      List options and their current values
  quit/exit/^D  Exit pprof

Options:
  call_tree      Create a context-sensitive call tree
  compact_labels Show minimal headers
  divide_by     Ratio to divide all samples before visualization
  drop_negative Ignore negative differences
  edgefraction  Hide edges below <f>*total
  focus          Restricts to samples going through a node matching regexp
  hide           Skips nodes matching regexp
  ignore         Skips paths going through any nodes matching regexp
  mean           Average sample value over first value (count)
  nodecount     Max number of nodes to show
  nodefraction  Hide nodes below <f>*total
  noinline      Ignore inlines.
  normalize     Scales profile based on the base profile.
  output         filename for file-based outputs
  prune_from    Drops any functions below the matched frame.
  relative_percentages Show percentages relative to focused subgraph
  sample_index   Sample value to report (0-based index or name)
  show           Only show nodes matching regexp
  show_from     Drops functions above the highest matched frame.
  source_path   Search path for source files
  tagfocus      Restricts to samples with tags in range or matched by regexp
  taghide       Skip tags matching this regexp
  tagignore     Discard samples with tags in range or matched by regexp
  tagshow       Only consider tags matching this regexp
  trim          Honor nodefraction/edgefraction/nodecount defaults
  trim_path     Path to trim from source paths before search
  unit          Measurement units to display

Option groups (only set one per group):
  cumulative
    cum           Sort entries based on cumulative weight
    flat          Sort entries based on own weight
  granularity
    addresses    Aggregate at the address level.
    filefunctions Aggregate at the function level.
```

```

files      Aggregate at the file level.
functions   Aggregate at the function level.
lines      Aggregate at the source code line level.
:  Clear focus/ignore/hide/tagfocus/tagignore

type "help <cmd|option>" for more information
(pprof)

```



Do find the time to try out all of the commands of the `go tool pprof` utility and familiarize yourself with them.

The `top` command returns the top 10 entries in text form:

```

(pprof) top
Showing nodes accounting for 4.42s, 99.10% of 4.46s total
Dropped 14 nodes (cum <= 0.02s)
Showing top 10 nodes out of 19
      flat  flat%  sum%  cum    cum%
2.69s 60.31% 60.31% 2.69s 60.31% main.N2
1.41s 31.61% 91.93% 1.41s 31.61% main.N1
0.19s 4.26% 96.19% 0.19s 4.26% runtime.nanotime
0.10s 2.24% 98.43% 0.10s 2.24% runtime.usleep
0.03s 0.67% 99.10% 0.03s 0.67% runtime.memclrNoHeapPointers
0     0% 99.10% 4.14s 92.83% main.main
0     0% 99.10% 0.03s 0.67% runtime.(*mheap).alloc
0     0% 99.10% 0.03s 0.67% runtime.largeAlloc
0     0% 99.10% 4.14s 92.83% runtime.main
0     0% 99.10% 0.03s 0.67% runtime.makeslice

```

As the first line of the output states, the functions presented are responsible for `99.10%` of the total execution time of the program.

The `main.N2` function in particular is responsible for `60.31%` of the execution time of the program.

The `top10 --cum` command returns the cumulative time for each function:

```

(pprof) top10 --cum
Showing nodes accounting for 4390ms, 98.43% of 4460ms total
Dropped 14 nodes (cum <= 22.30ms)
Showing top 10 nodes out of 19
      flat  flat%  sum%  cum    cum%
0     0% 0% 4140ms 92.83% main.main
0     0% 0% 4140ms 92.83% runtime.main
2690ms 60.31% 60.31% 2690ms 60.31% main.N2
1410ms 31.61% 91.93% 1410ms 31.61% main.N1
0     0% 91.93% 290ms 6.50% runtime.mstart
0     0% 91.93% 270ms 6.05% runtime.mstart1
0     0% 91.93% 270ms 6.05% runtime.sysmon
190ms 4.26% 96.19% 190ms 4.26% runtime.nanotime
100ms 2.24% 98.43% 100ms 2.24% runtime.usleep
0     0% 98.43% 50ms 1.12% runtime.systemstack

```

What if you want to find out what is happening with a particular function? You can use the `list` command followed by the function name, combined with the package name, and you'll get more detailed information about the performance of that function:

```

(pprof) list main.N1
Total: 4.18s
ROUTINE ===== main.N1 in /Users/mtsouk/ch11/code/profileMe.go
 1.41s   1.41s (flat, cum) 31.61% of Total
  .       .           32:     return fn[n]
  .       .           33: }
  .       .           34:
  .       .           35: func N1(n int) bool {
  .       .           36:     k := math.Floor(float64(n/2 + 1))
  60ms   60ms   37:     for i := 2; i < int(k); i++ {
  1.35s   1.35s   38:         if (n % i) == 0 {
  .       .             39:             return false
  .       .         }
  .       .         40:     }
  .       .         41:   }
  .       .         42:   return true
  .       .         43: }

(pprof)

```

The output shows that the `for` loop of `main.N1` is responsible for almost all of the execution time of the entire function. Specifically, the `if (n % i) == 0` statement is responsible for `1.35s` out of `1.41s` of the execution time of the

entire function.

You can also create PDF output of the profiling data from the shell of the Go profiler using the `pdf` command:

```
| (pprof) pdf  
Generating report in profile001.pdf
```

Note that you will need **Graphviz** in order to generate a PDF file that can be viewed using your favorite PDF reader.

Finally, a warning: if your program executes too quickly, then the profiler will not have enough time to get its required samples and you might see the `Total samples = 0` output when you load the data file. In that case, you will not be able to get any useful information from the profiling process. That is the reason for using the `time.Sleep()` function in some of the functions of `profileMe.go`:

```
| $ go tool pprof /tmp/cpuProfile.out  
Type: cpu  
Time: Apr 24, 2019 at 4:37pm (EEST)  
Duration: 19.59s, Total samples = 4.46s (22.77%)  
Entering interactive mode (type "help" for commands, "o" for options)  
(pprof)
```

A convenient external package for profiling

In this subsection, you will see the use of an external Go package that sets up the profiling environment much more conveniently than by using the `runtime/pprof` standard Go package. This point is illustrated in `betterProfile.go`, which will be presented in three parts.

The first part of `betterProfile.go` is as follows:

```
package main
import (
    "fmt"
    "github.com/pkg/profile"
)
var VARIABLE int
func N1(n int) bool {
    for i := 2; i < n; i++ {
        if (n % i) == 0 {
            return false
        }
    }
    return true
}
```

In the preceding code, you can see the use of an external Go package that can be found at github.com/pkg/profile. You can download it with the help of the `go get` command, as follows:

```
$ go get github.com/pkg/profile
```

The second code segment of `betterProfile.go` contains the following Go code:

```
func Multiply(a, b int) int {
    if a == 1 {
        return b
    }
    if a == 0 || b == 0 {
        return 0
    }
    if a < 0 {
        return -Multiply(-a, b)
    }
    return b + Multiply(a-1, b)
```

```
| }  
| func main() {  
|     defer profile.Start(profile.ProfilePath("/tmp")).Stop()
```

The github.com/pkg/profile package by Dave Cheney requires that you insert just a single statement in order to enable **CPU profiling** in your Go application. If you want to enable **memory profiling**, you should insert the following statement instead:

```
| defer profile.Start(profile.MemProfile).Stop()
```

The remaining Go code of the program is as follows:

```
total := 0  
for i := 2; i < 200000; i++ {  
    n := N1(i)  
    if n {  
        total++  
    }  
}  
fmt.Println("Total: ", total)  
total = 0  
for i := 0; i < 5000; i++ {  
    for j := 0; j < 400; j++ {  
        k := Multiply(i, j)  
        VARIABLE = k  
        total++  
    }  
}  
fmt.Println("Total: ", total)
```

Executing `betterProfile.go` generates the following output:

```
$ go run betterProfile.go  
2019/04/24 16:44:05 profile: cpu profiling enabled, /tmp/cpu.pprof  
Total: 17984  
Total: 2000000  
2019/04/24 16:44:33 profile: cpu profiling disabled, /tmp/cpu.pprof
```



The github.com/pkg/profile package will help you with the data capturing portion; the processing part is the same as before.

```
$ go tool pprof /tmp/cpu.pprof  
Type: cpu  
Time: Apr 24, 2019 at 4:44pm (EEST)  
Duration: 27.40s, Total samples = 25.10s (91.59%)  
Entering interactive mode (type "help" for commands, "o" for options)  
(pprof)
```

The web interface of the Go profiler

The good news is that with Go version 1.10, the `go tool pprof` command comes with a web user interface.



*For the web user interface feature to work, you will need to have **Graphviz** installed and your web browser must support JavaScript. If you want to play it safe, use either **Chrome** or **Firefox**.*

You can start the interactive Go profiler as follows:

```
| $ go tool pprof -http=[host]:[port] aProfile
```

A profiling example that uses the web interface

We will use the data captured from the execution of `profileMe.go` to study the web interface of the Go profiler, as there is no need to create a specific code example to do this. As you learned in the previous subsection, you will first need to execute the following command:

```
$ go tool pprof -http=localhost:8080 /tmp/cpuProfile.out
Main binary filename not available.
```

The following figure shows the initial screen of the web user interface of the Go profiler after executing the preceding command:

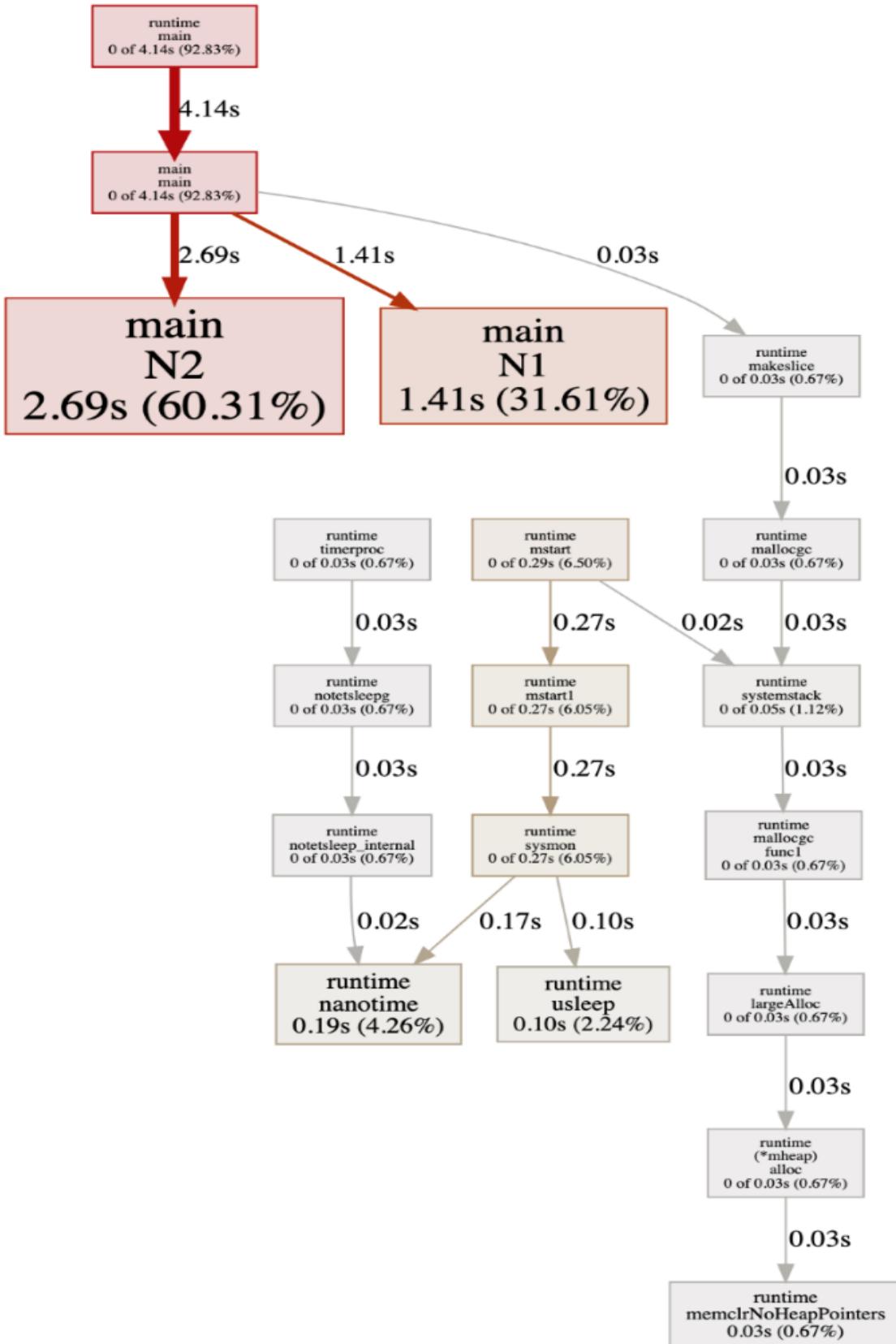


Figure 11.1: The web interface of the Go profiler in action

Similarly, the following figure shows the <http://localhost:8080/ui/source> URL of the Go profiler, which displays analytical information for each function of the program:

unknown cpu

localhost:8080/ui/source

pprof VIEW SAMPLE REFINE Search regexp unknown cpu

main.N2

```
/Users/mtsouk/Desktop/mGo2nd/Mastering-Go-Second-Edition/ch11/profileMe.go
```

Total:	2.69s	2.69s (flat, cum)	60.31%
41	.	.	}
42	.	.	return true
43	.	.	}
44	.	.	
45	.	.	func N2(n int) bool {
46	90ms	90ms	for i := 2; i < n; i++ {
47	2.60s	2.60s	if (n % i) == 0 {
48	.	.	return false
49	.	.	}
50	.	.	}
51	.	.	return true
52	.	.	}

main.N1

```
/Users/mtsouk/Desktop/mGo2nd/Mastering-Go-Second-Edition/ch11/profileMe.go
```

Total:	1.41s	1.41s (flat, cum)	31.61%
32	.	.	return fn[n]
33	.	.	}
34	.	.	
35	.	.	func N1(n int) bool {
36	.	.	k := math.Floor(float64(n/2 + 1))
37	60ms	60ms	for i := 2; i < int(k); i++ {
38	1.35s	1.35s	if (n % i) == 0 {
39	.	.	return false
40	.	.	}
41	.	.	}
42	.	.	return true
43	.	.	}

runtime.nanotime

```
/usr/local/Cellar/go/1.12.4/libexec/src/runtime/sys_darwin.go
```

Total:	190ms	190ms (flat, cum)	4.26%
222	*	*	func open_trampoline()
223	*	*	
224	*	*	//go:nosplit
225	*	*	//go:cgo_unsafe_args
226	*	*	func nanotime() int64 {
227	190ms	190ms	var r struct {
228	*	*	t int64 // raw timer
229	*	*	numer, denom uint32 // conversion factors. nanoseconds = t * numer / denom.
230	*	*	}
231	*	*	libcCall(unsafe.Pointer(funcPC(nanotime_trampoline)), unsafe.Pointer(&r))
232	*	*	// Note: Apple seems unconcerned about overflow here. See

runtime.usleep

```
/usr/local/Cellar/go/1.12.4/libexec/src/runtime/sys_darwin.go
```

Total:	100ms	100ms (flat, cum)	2.24%
202	*	*	
203	*	*	//go:nosplit
204	*	*	//go:cgo_unsafe_args
205	*	*	func usleep(usec uint32) {

Figure 11.2: Using the /source URL of the Go profiler



As I cannot possibly show every single page of the Go profiler web interface, you should start to familiarize yourself with it on your own, as it is a great tool for examining the operations of your programs.

A quick introduction to Graphviz

Graphviz is a very handy compilation of utilities and a computer language that allows you to draw complex graphs. Strictly speaking, Graphviz is a collection of tools for manipulating both directed and undirected **graph** structures and generating graph layouts. Graphviz has its own language, named **DOT**, which is simple, elegant, and powerful. The good thing about Graphviz is that you can write its code using a simple plain text editor. A wonderful side effect of this feature is that you can easily develop scripts that generate Graphviz code. Also, most programming languages, including **Python**, **Ruby**, **C++**, and **Perl**, provide their own interfaces for creating Graphviz files using native code.



You do not need to know all of these things about Graphviz in order to use the web interface of the Go profiler. It is just useful to know how Graphviz works and what its code looks like.

The following Graphviz code, which is saved as `graph.dot`, briefly illustrates the way that Graphviz works and the look of the Graphviz language:

```
digraph G
{
    graph [dpi = 300, bgcolor = "gray"];
    rankdir = LR;
    node [shape=record, width=.2, height=.2, color="white" ];

    node0 [label = "<p0>; |<p1>|<p2>|<p3>|<p4>| | ", height = 3];
    node[ width=2 ];
    node1 [label = "{<e> r0 | 123 | <p> }", color="gray" ];
    node2 [label = "{<e> r10 | 13 | <p> }" ];
    node3 [label = "{<e> r11 | 23 | <p> }" ];
    node4 [label = "{<e> r12 | 326 | <p> }" ];
    node5 [label = "{<e> r13 | 1f3 | <p> }" ];
    node6 [label = "{<e> r20 | 143 | <p> }" ];
    node7 [label = "{<e> r40 | b23 | <p> }" ];

    node0:p0 -> node1:e [dir=both color="red:blue"];
    node0:p1 -> node2:e [dir=back arrowhead=diamond];
    node2:p -> node3:e;
    node3:p -> node4:e [dir=both arrowtail=box color="red"];
    node4:p -> node5:e [dir=forward];
    node0:p2 -> node6:e [dir=none color="orange"];
    node0:p4 -> node7:e;
}
```

The `color` attribute changes the color of a node, whereas the `shape` attribute changes the shape of a node. Additionally, the `dir` attribute, which can be applied to edges, defines whether an edge is going to have two arrows, one, or none. Furthermore, the style of the arrowhead can be specified using the `arrowhead` and `arrowtail` attributes.

Compiling the preceding code using one of the Graphviz command-line tools in order to create a PNG image requires the execution of the following command in your favorite UNIX shell:

```
$ dot -T png graph.dot -o graph.png
$ ls -l graph.png
-rw-r--r--@ 1 mtsouk staff 94862 Apr 24 16:48 graph.png
```

The next figure shows the graphics file generated from the execution of the preceding command:

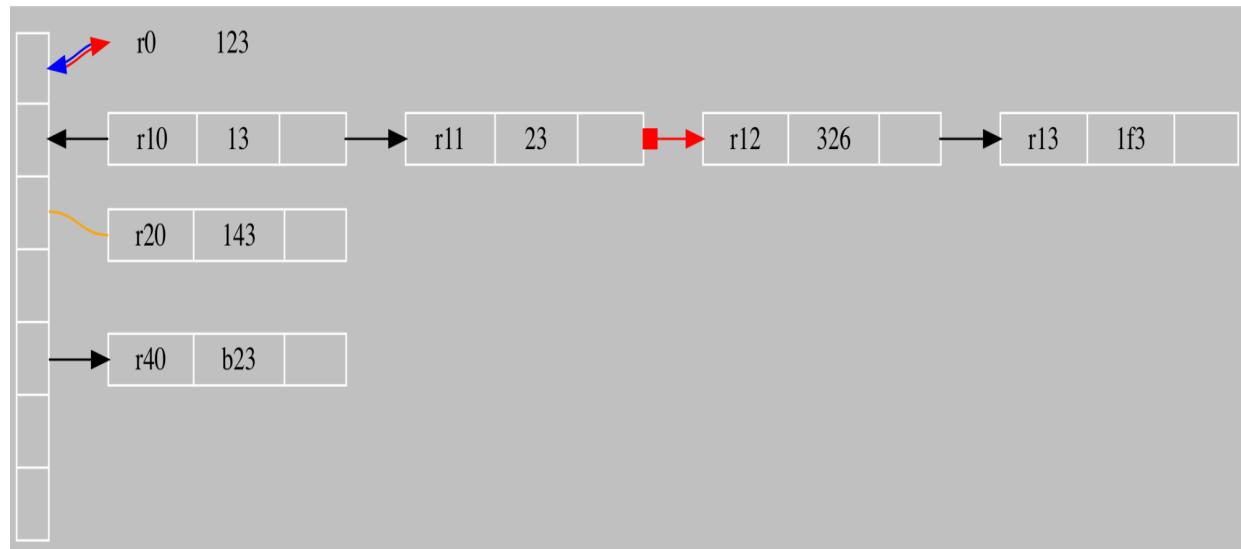


Figure 11.3: Using Graphviz to create graphs

Thus, if you want to visualize any kind of structure, you should definitely consider using Graphviz and its tools, especially if you want to automate things with your own scripts.

The go tool trace utility

The `go tool trace` utility is a tool for viewing trace files that can be generated in any one of the following three ways:

- Using the `runtime/trace` package
- Using the `net/http/pprof` package
- Executing the `go test -trace` command

This section will use the first technique only. The output of the following command will greatly help you to understand what the **Go execution tracer** does:

```
$ go doc runtime/trace
package trace // import "runtime/trace"

Package trace contains facilities for programs to generate traces for the Go
execution tracer.

Tracing runtime activities

The execution trace captures a wide range of execution events such as
goroutine creation/blocking/unblocking, syscall enter/exit/block, GC-related
events, changes of heap size, processor start/stop, etc. A precise
nanosecond-precision timestamp and a stack trace is captured for most
events. The generated trace can be interpreted using `go tool trace`.

The trace tool computes the latency of a task by measuring the time between the task creation and the task end and provides

func IsEnabled() bool
func Log(ctx context.Context, category, message string)
func Logf(ctx context.Context, category, format string, args ...interface{})
func Start(w io.Writer) error
func Stop()
func WithRegion(ctx context.Context, regionType string, fn func())
type Region struct{ ... }
    func StartRegion(ctx context.Context, regionType string) *Region
type Task struct{ ... }
    func NewTask(pctx context.Context, taskType string) (ctx context.Context, task *Task)
```

In [Chapter 2](#), *Understanding Go Internals*, we talked about the **Go garbage collector** and presented a Go utility, `gColl.go`, which allowed us to see some of the variables of the Go garbage collector. In this section, we are going to gather even more information about the operation of `gColl.go` using the `go tool trace` utility.

First, let's examine the modified version of the `gColl.go` program, which tells Go to collect performance data. It is saved as `goGC.go`, and it will be presented in three parts.

The first part of `goGC.go` is as follows:

```
package main
import (
    "fmt"
    "os"
    "runtime"
    "runtime/trace"
    "time"
)

func printStats(mem runtime.MemStats) {
    runtime.ReadMemStats(&mem)
    fmt.Println("mem.Alloc:", mem.Alloc)
    fmt.Println("mem.TotalAlloc:", mem.TotalAlloc)
    fmt.Println("mem.HeapAlloc:", mem.HeapAlloc)
    fmt.Println("mem.NumGC:", mem.NumGC)
    fmt.Println("----")
```

As you already know, you first need to import the `runtime/trace` standard Go package in order to collect data for the `go tool trace` utility.

The second code segment of `goGC.go` is shown in the following Go code:

```

func main() {
    f, err := os.Create("/tmp/traceFile.out")
    if err != nil {
        panic(err)
    }
    defer f.Close()
    err = trace.Start(f)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer trace.Stop()
}

```

This part is all about acquiring data for the `go tool trace` utility, and it has nothing to do with the functionality of the actual program. First, you create a new file that will hold the tracing data for the `go tool trace` utility. Then, you start the tracing process using `trace.Start()`. When you are done, you call the `trace.Stop()` function. The `defer` call to this function means that you want to terminate tracing when your program ends.



Using the `go tool trace` utility is a process with two phases that requires extra Go code. First, you collect the data, and then you display and process it.

The remaining Go code is as follows:

```

var mem runtime.MemStats
printStats(mem)
for i := 0; i < 3; i++ {
    s := make([]byte, 50000000)
    if s == nil {
        fmt.Println("Operation failed!")
    }
}
printStats(mem)
for i := 0; i < 5; i++ {
    s := make([]byte, 100000000)
    if s == nil {
        fmt.Println("Operation failed!")
    }
    time.Sleep(time.Millisecond)
}
printStats(mem)
}

```

Executing `goGC.go` produces the following output, as well as a new file named `/tmp/traceFile.out` with the tracing information:

```

$ go run goGC.go
mem.Alloc: 108592
mem.TotalAlloc: 108592
mem.HeapAlloc: 108592
mem.NumGC: 0
-----
mem.Alloc: 109736
mem.TotalAlloc: 150127000
mem.HeapAlloc: 109736
mem.NumGC: 3
-----
mem.Alloc: 114672
mem.TotalAlloc: 650172952
mem.HeapAlloc: 114672
mem.NumGC: 8
-----
$ cd /tmp
$ ls -l traceFile.out
-rw-r--r-- 1 mtsouk wheel 10108 Apr 24 16:51 /tmp/traceFile.out
$ file /tmp/traceFile.out
/tmp/traceFile.out: data

```

The `go tool trace` utility uses a web interface that starts automatically when you execute the next command:

```

$ go tool trace /tmp/traceFile.out
2019/04/24 16:52:06 Parsing trace...
2019/04/24 16:52:06 Splitting trace...
2019/04/24 16:52:06 Opening browser. Trace viewer is listening on http://127.0.0.1:50383

```

The figure that follows shows the initial image of the web interface of the `go tool trace` utility when examining the `/tmp/traceFile.out` trace file.

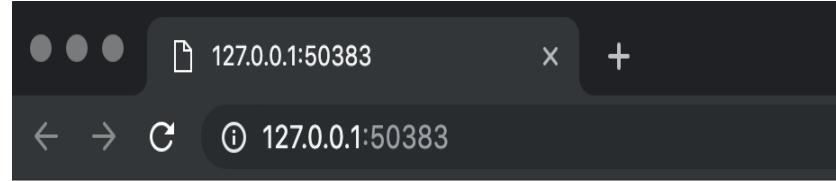


Figure 11.4: The initial screen of the web interface of the go tool trace utility

You should now select the `View trace` link. This will take you to the next figure, which shows you another view of the web interface of the `go tool trace` utility that uses the data from `/tmp/traceFile.out`.

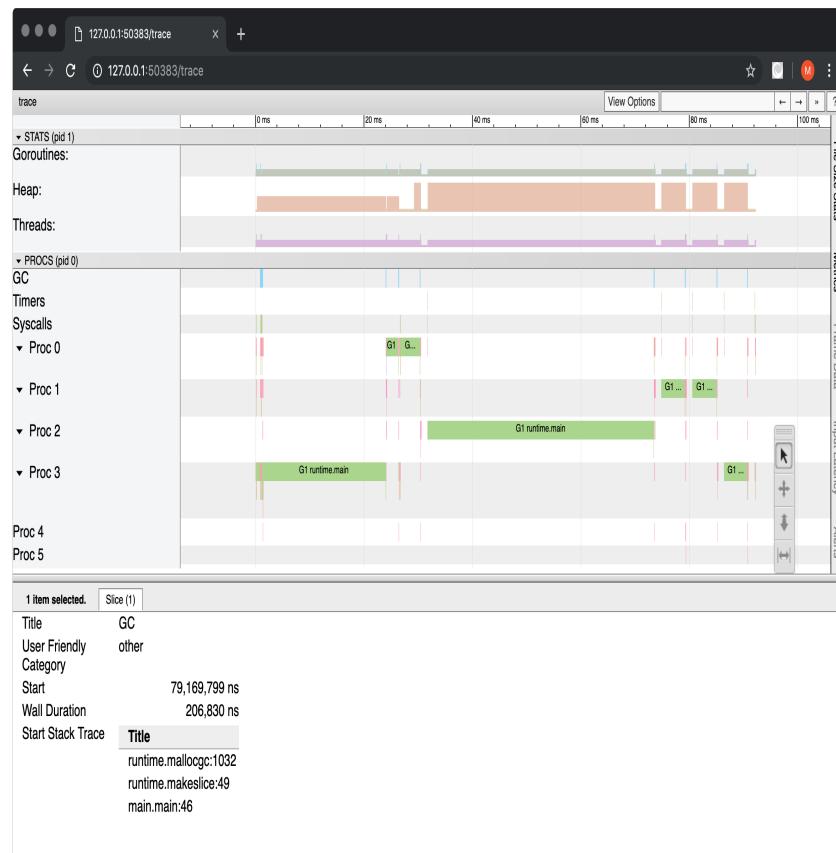


Figure 11.5: Examining the operation of the Go garbage collector using go tool trace

In the preceding figure, you can see that the Go garbage collector runs on its own goroutine, but that it does not run all of the time. Additionally, you can see the number of goroutines used by the program. You can learn more about this by selecting certain parts of the interactive view. As we are interested in the operation of the garbage collector, a useful piece of information that is displayed is how often and for how long the garbage collector runs.

Notice that although `go tool trace` is a very handy and powerful utility, it cannot solve every kind of performance problem. There are times when `go tool pprof` is more appropriate, especially when you want to reveal where your program spends most of its time by individual function.

Testing Go code

Software testing is a very large subject and cannot be covered in a single section of a chapter in a book. So, this brief section will try to present as much practical information as possible.

Go allows you to write tests for your Go code in order to detect bugs. Strictly speaking, this section is about **automated testing**, which involves writing extra code to verify whether the real code – that is, the production code – works as expected or not. Thus, the result of a test function is either `PASS` or `FAIL`. You will see how this works shortly.

Although the Go approach to testing might look simple at first, especially if you compare it with the testing practices of other programming languages, it is very efficient and effective because it does not require too much of the developer's time.

Go follows certain conventions regarding testing. First of all, testing functions should be included in Go source files that end with `_test.go`. So, if you have a package named `aGoPackage.go`, then your tests should be placed in a file named `aGoPackage_test.go`. A test function begins with `Test`, and it checks the correctness of the behavior of a function of the production package. Finally, you will need to import the `testing` standard Go package for the `go test` subcommand to work correctly. As you will soon see, this import requirement also applies to two additional cases.

Once the testing code is correct, the `go test` subcommand does all the dirty work for you, which includes scanning all `*_test.go` files for special functions, generating a temporary `main` package properly, calling these special functions, getting the results, and generating the final output.



Always put the testing code in a different source file. There is no need to create a huge source file that is hard to read and maintain.

Writing tests for existing Go code

In this section, you will learn how to write tests for an existing Go application that includes two functions: one for calculating numbers of the Fibonacci sequence and one for finding out the length of a string. The main reason for using these two functions that implement such relatively trivial tasks is simplicity. The tricky point here is that each function will have two different implementations: one that works well and another one that has some issues.

The Go package in this example is named `testMe` and it is saved as `testMe.go`. The code of this package will be presented in three parts.

The first part of `testMe.go` includes the following Go code:

```
package testMe
func f1(n int) int {
    if n == 0 {
        return 0
    }
    if n == 1 {
        return 1
    }
    return f1(n-1) + f1(n-2)
}
```

In the preceding code, you can see the definition of a function named `f1()`, which calculates natural numbers of the Fibonacci sequence.

The second part of `testMe.go` is shown in the following Go code:

```
func f2(n int) int {
    if n == 0 {
        return 0
    }
    if n == 1 {
        return 2
    }
    return f2(n-1) + f2(n-2)
}
```

In this code, you can see the implementation of another function that calculates the numbers of the Fibonacci sequence, named `f2()`. However, this function contains a bug because it does not return `1` when the value of `n` is `1`, which destroys the entire functionality of the function.

The remaining code of `testMe.go` is shown in the following Go code:

```
func s1(s string) int {
    if s == "" {
        return 0
    }
    n := 1
    for range s {
        n++
    }
    return n
}

func s2(s string) int {
    return len(s)
}
```

In this part, we implement two functions named `s1()` and `s2()`, which work on strings. Both of these functions find the length of a string. However, the implementation of `s1()` is incorrect because the initial value of `n` is `1` instead of `0`.

It is now time to start thinking about tests and test cases. First of all, you should create a `testMe_test.go` file, which will be used to store your testing functions. Next, it is important to realize that you do not need to make any code changes to `testMe.go`. Finally, remember that you should try to write as many tests as required to cover all potential inputs and outputs.

The first part of `testMe_test.go` is shown in the following Go code:

```
package testMe

import "testing"

func TestS1(t *testing.T) {
    if s1("123456789") != 9 {
        t.Error(`s1("123456789") != 9`)
    }
    if s1("") != 0 {
        t.Error(`s1("") != 0`)
    }
}
```

The preceding function performs two tests on the `s1()` function: one using `"123456789"` as input and another one using `""` as input.

The second part of `testMe_test.go` is as follows:

```
func TestS2(t *testing.T) {
    if s2("123456789") != 9 {
        t.Error(`s2("123456789") != 9`)
    }
    if s2("") != 0 {
        t.Error(`s2("") != 0`)
    }
}
```

The preceding testing code performs the same two tests on the `s2()` function.

The remaining code of `testMe_test.go` comes next:

```
func TestF1(t *testing.T) {
    if f1(0) != 0 {
        t.Error(`f1(0) != 0`)
    }
    if f1(1) != 1 {
        t.Error(`f1(1) != 1`)
    }
    if f1(2) != 1 {
        t.Error(`f1(2) != 1`)
    }
    if f1(10) != 55 {
        t.Error(`f1(10) != 55`)
    }
}

func TestF2(t *testing.T) {
    if f2(0) != 0 {
        t.Error(`f2(0) != 0`)
    }
    if f2(1) != 1 {
        t.Error(`f2(1) != 1`)
    }
    if f2(2) != 1 {
        t.Error(`f2(2) != 1`)
    }
    if f2(10) != 55 {
        t.Error(`f2(10) != 55`)
    }
}
```

The previous code tests the operation of the `f1()` and `f2()` functions.

Executing the tests will generate the following type of output:

```
$ go test testMe.go testMe_test.go -v
==== RUN TestS1
--- FAIL: TestS1 (0.00s)
    testMe_test.go:7: s1("123456789") != 9
==== RUN TestS2
--- PASS: TestS2 (0.00s)
==== RUN TestF1
--- PASS: TestF1 (0.00s)
==== RUN TestF2
--- FAIL: TestF2 (0.00s)
    testMe_test.go:50: f2(1) != 1
    testMe_test.go:54: f2(2) != 1
    testMe_test.go:58: f2(10) != 55
FAIL
FAIL  command-line-arguments 0.005s
```

If you do not include the `-v` parameter, which produces richer output, you will get the following output:

```
$ go test testMe.go testMe_test.go
--- FAIL: TestS1 (0.00s)
    testMe_test.go:7: s1("123456789") != 9
--- FAIL: TestF2 (0.00s)
    testMe_test.go:50: f2(1) != 1
    testMe_test.go:54: f2(2) != 1
    testMe_test.go:58: f2(10) != 55
FAIL
FAIL  command-line-arguments 0.005s
```

If you want to run a test multiple times in succession, you can use the `-count` option as follows:

```
$ go test testMe.go testMe_test.go -count 2
--- FAIL: TestS1 (0.00s)
    testMe_test.go:7: s1("123456789") != 9
--- FAIL: TestF2 (0.00s)
    testMe_test.go:50: f2(1) != 1
    testMe_test.go:54: f2(2) != 1
    testMe_test.go:58: f2(10) != 55
--- FAIL: TestS1 (0.00s)
    testMe_test.go:7: s1("123456789") != 9
--- FAIL: TestF2 (0.00s)
    testMe_test.go:50: f2(1) != 1
    testMe_test.go:54: f2(2) != 1
    testMe_test.go:58: f2(10) != 55
FAIL
FAIL  command-line-arguments 0.005s
```

Should you wish to execute specific tests, you should use the `-run` command-line option, which accepts a regular expression and executes all tests that have a function name that matches the given regular expression:

```
$ go test testMe.go testMe_test.go -run='F2' -v
==== RUN TestF2
--- FAIL: TestF2 (0.00s)
    testMe_test.go:50: f2(1) != 1
    testMe_test.go:54: f2(2) != 1
    testMe_test.go:58: f2(10) != 55
FAIL

FAIL command-line-arguments 0.005s
$ go test testMe.go testMe_test.go -run='F1'
ok    command-line-arguments (cached)
```

The last command verifies that the `go test` command used caching.



CAUTION: Software testing can only show the presence of one or more bugs, not the absence of bugs. This means that you can never be absolutely sure that your code has no bugs!

Test code coverage

In this section, you will learn how to find more information about the code coverage of your programs. There are times when seeing the code coverage of your programs can reveal issues and bugs with your code, so do not underestimate its usefulness.

The Go code of `codeCover.go` is as follows:

```
package codeCover

func fibo1(n int) int {
    if n == 0 {
        return 0
    } else if n == 1 {
        return 1
    } else {
        return fibo1(n-1) + fibo1(n-2)
    }
}

func fibo2(n int) int {
    if n >= 0 {
        return 0
    } else if n == 1 {
        return 1
    } else {
        return fibo1(n-1) + fibo1(n-2)
    }
}
```

The implementation of `fibo2()` is erroneous because it returns `0` all the time. A result of this bug is that most of the Go code of `fibo2()` will never get executed.

The test code that can be found in `codeCover_test.go` is as follows:

```
package codeCover

import (
    "testing"
)

func TestFibol(t *testing.T) {
    if fibo1(1) != 1 {
        t.Errorf("Error fibo1(1): %d\n", fibo1(1))
    }
}
```

```

    }

func TestFibo2(t *testing.T) {
    if fibo2(0) != 0 {
        t.Errorf("Error fibo2(0): %d\n", fibo1(0))
    }
}

func TestFibol_10(t *testing.T) {
    if fibol(10) == 1 {
        t.Errorf("Error fibol(1): %d\n", fibol(1))
    }
}

func TestFibo2_10(t *testing.T) {
    if fibo2(10) != 0 {
        t.Errorf("Error fibo2(0): %d\n", fibo1(0))
    }
}

```

The implementations of these functions are pretty naive because their purpose is to illustrate the use of the code coverage tool and not the generation of test functions.

Now we are going to check the coverage of the previous code. The main way to do this is by executing `go test` with the `-cover` parameter:

```

$ go test -cover -v
--- RUN    TestFibol
--- PASS: TestFibol (0.00s)
--- RUN    TestFibo2
--- PASS: TestFibo2 (0.00s)
--- RUN    TestFibol_10
--- PASS: TestFibol_10 (0.00s)
--- RUN    TestFibo2_10
--- PASS: TestFibo2_10 (0.00s)

PASS
coverage: 70.0% of statements
ok      _/Users/mtsouk/cover    0.005s

```

So, the code coverage is 70.0%, which means that there might be a problem somewhere. Notice that the `-v` flag is not required for `-cover` to work.

However, there is a variation of the main command for test coverage that can generate a coverage profile:

```

$ go test -coverprofile=coverage.out
PASS
coverage: 70.0% of statements

```

```
| ok      _/Users/mtsouk/cover      0.005s
```

After generating that special file, you can analyze it as follows:

```
$ go tool cover -func=coverage.out
/Users/mtsouk/cover/codeCover.go:3: fibo1    100.0%
/Users/mtsouk/cover/codeCover.go:13: fibo2    40.0%
total: (statements)    70.0%
```

Additionally, you can use a web browser to analyze that code coverage file as follows:

```
| $ go tool cover -html=coverage.out
```

In the browser window that automatically opens, you will see lines of code colored either in green or in red. Red-colored lines of code are not covered by tests, which means that you should either cover them with more test cases or that there is something wrong with the Go code that is being tested. One way or another, you should look into these lines of code and reveal the problem.

Lastly, you can save that HTML output as follows:

```
| $ o tool cover -html=coverage.out -o output.html
```

Testing an HTTP server with a database backend

In this section, you are going to see how to test a database server, which in this case is going to be a **PostgreSQL** server that works with an HTTP server written in Go. This is a special situation because you will have to get your database data in order to be able to test the correctness of the HTTP server.

For the purposes of this section, I am going to show two Go files named `webServer.go` and `webServer_test.go`. Before executing the code, you will need to download the Go package that will help you to work with PostgreSQL from Go, which requires the execution of the following command:

```
$ go get github.com/lib/pq
```

The Go code of `webServer.go` is as follows:

```
package main

import (
    "database/sql"
    "fmt"
    "github.com/lib/pq"
    "net/http"
    "os"
    "time"
)

func myHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served: %s\n", r.Host)
}

func timeHandler(w http.ResponseWriter, r *http.Request) {
    t := time.Now().Format(time.RFC1123)
    Body := "The current time is:"
    fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>", Body)
    fmt.Fprintf(w, "<h2 align=\"center\">%s</h2>\n", t)
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served time for: %s\n", r.Host)
}

func getData(w http.ResponseWriter, r *http.Request) {
    fmt.Printf("Serving: %s\n", r.URL.Path)
    fmt.Printf("Served: %s\n", r.Host)

    connStr := "user=postgres dbname=s2 sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>", err)
        return
    }
}
```

```

rows, err := db.Query("SELECT * FROM users")
if err != nil {
    fmt.Fprintf(w, "<h3 align=\"center\">%s</h3>\n", err)
    return
}
defer rows.Close()

for rows.Next() {
    var id int
    var firstName string
    var lastName string
    err = rows.Scan(&id, &firstName, &lastName)
    if err != nil {
        fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>\n", err)
        return
    }
    fmt.Fprintf(w, "<h3 align=\"center\">%d, %s, %s</h3>\n", id, firstName, lastName)
}

err = rows.Err()
if err != nil {
    fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>", err)
    return
}
}

func main() {
    PORT := ":8001"
    arguments := os.Args
    if len(arguments) != 1 {
        PORT = ":" + arguments[1]
    }
    fmt.Println("Using port number: ", PORT)
    http.HandleFunc("/time", timeHandler)
    http.HandleFunc("/getdata", getData)
    http.HandleFunc("/", myHandler)

    err := http.ListenAndServe(PORT, nil)
    if err != nil {
        fmt.Println(err)
        return
    }
}

```

The Go code of `webServer_test.go` will be presented in six parts. The thing with testing a web server that gets its data from a database is that you will need to write lots of utility code to support the testing of the database.

The first part of `webServer_test.go` is as follows:

```

package main

import (
    "database/sql"
    "fmt"
    "github.com/lib/pq"
    "net/http"
    "net/http/httpptest"
    "testing"
)

```

The second part of `webServer_test.go` contains the following Go code:

```

func create_table() {
    connStr := "user=postgres dbname=s2 sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        fmt.Println(err)
    }

    const query = `CREATE TABLE IF NOT EXISTS users (
        id SERIAL PRIMARY KEY,
        first_name TEXT,
        last_name TEXT
    )`,

    _, err = db.Exec(query)
    if err != nil {
        fmt.Println(err)
        return
    }
    db.Close()
}

```

This function communicates with PostgreSQL and creates a table named `users` inside the database, which will be used for testing purposes only.

The third part of `webServer_test.go` is as follows:

```

func drop_table() {
    connStr := "user=postgres dbname=s2 sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        fmt.Println(err)
        return
    }

    _, err = db.Exec("DROP TABLE IF EXISTS users")
    if err != nil {
        fmt.Println(err)
        return
    }
    db.Close()
}

func insert_record(query string) {
    connStr := "user=postgres dbname=s2 sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        fmt.Println(err)
        return
    }

    _, err = db.Exec(query)
    if err != nil {
        fmt.Println(err)
        return
    }
    db.Close()
}

```

These are two helper functions. The first one deletes the table that is created by `create_table()`, whereas the second one inserts a record in PostgreSQL.

The fourth part of `webServer_test.go` is the following:

```

func Test_count(t *testing.T) {
    var count int
    create_table()

    insert_record("INSERT INTO users (first_name, last_name) VALUES ('Epifanios', 'Doe')")
    insert_record("INSERT INTO users (first_name, last_name) VALUES ('Mihalis', 'Tsoukalos')")
    insert_record("INSERT INTO users (first_name, last_name) VALUES ('Mihalis', 'Unknown')")

    connStr := "user=postgres dbname=s2 sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        fmt.Println(err)
        return
    }

    row := db.QueryRow("SELECT COUNT(*) FROM users")
    err = row.Scan(&count)
    db.Close()

    if count != 3 {
        t.Errorf("Select query returned %d", count)
    }
    drop_table()
}

```

This is the first test function of the package and it operates in two stages. Firstly, it inserts three records into a database table. Secondly, it verifies that the database table has exactly three records in it.

The fifth part of `webServer_test.go` contains the following Go code:

```

func Test_queryDB(t *testing.T) {
    create_table()

    connStr := "user=postgres dbname=s2 sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        fmt.Println(err)
        return
    }

    query := "INSERT INTO users (first_name, last_name) VALUES ('Random Text', '123456')"
    insert_record(query)

    rows, err := db.Query(`SELECT * FROM users WHERE last_name=$1`, `123456`)
    if err != nil {
        fmt.Println(err)
        return
    }
    var col1 int
    var col2 string
    var col3 string
    for rows.Next() {
        rows.Scan(&col1, &col2, &col3)
    }
    if col2 != "Random Text" {
        t.Errorf("first_name returned %s", col2)
    }

    if col3 != "123456" {
        t.Errorf("last_name returned %s", col3)
    }

    db.Close()
    drop_table()
}

```

This is another test function that inserts a record into a database table and verifies that the data was written correctly.

The last part of `webServer_test.go` is as follows:

```
func Test_record(t *testing.T) {
    create_table()
    insert_record("INSERT INTO users (first_name, last_name) VALUES ('John', 'Doe')")

    req, err := http.NewRequest("GET", "/getdata", nil)
    if err != nil {
        fmt.Println(err)
        return
    }
    rr := httptest.NewRecorder()
    handler := http.HandlerFunc(getData)
    handler.ServeHTTP(rr, req)

    status := rr.Code
    if status != http.StatusOK {
        t.Errorf("Handler returned %v", status)
    }

    if rr.Body.String() != "<h3 align=\"center\">1, John, Doe</h3>\n" {
        t.Errorf("Wrong server response!")
    }
    drop_table()
}
```

The last test function of the package interacts with the web server and visits the `/getdata` URL. Then, it verifies that the return value is the expected one.

At this point, you should create a PostgreSQL database named `s2` because this is the one that is used by the testing code. This can be done as follows:

```
$ psql -p 5432 -h localhost -U postgres -c "CREATE DATABASE s2"
CREATE DATABASE
```

If everything is OK, you will get the following kind of output from `go test`:

```
$ go test webServer* -v
===[ RUN   Test_count
--- PASS: Test_count (0.05s)
===[ RUN   Test_queryDB
--- PASS: Test_queryDB (0.04s)
===[ RUN   Test_record
Serving: /getdata
Served:
--- PASS: Test_record (0.04s)
PASS
ok    command-line-arguments      0.138s
```

Omitting the `-v` option will generate a shorter output:

```
$ go test webServer*
ok    command-line-arguments      0.160s
```

Notice that you should not start the web server before the `go test` command, as it is automatically started by `go test`.

If the PostgreSQL server is not running, the test will fail with the following error messages:

```
$ go test webServer* -v
===[REDACTED] RUN  Test_count
dial tcp [::1]:5432: connect: connection refused
--- FAIL: Test_count (0.01s)
    webServer_test.go:85: Select query returned 0
===[REDACTED] RUN  Test_queryDB
dial tcp [::1]:5432: connect: connection refused
dial tcp [::1]:5432: connect: connection refused
dial tcp [::1]:5432: connect: connection refused
--- PASS: Test_queryDB (0.00s)
===[REDACTED] RUN  Test_record
dial tcp [::1]:5432: connect: connection refused
dial tcp [::1]:5432: connect: connection refused
Serving: /getdata
Served:
dial tcp [::1]:5432: connect: connection refused
--- FAIL: Test_record (0.00s)
    webServer_test.go:145: Wrong server response!
FAIL
FAIL    command-line-arguments      0.024s
```

In [Chapter 12](#), *The Foundations of Network Programming in Go*, you are going to see an example where you test the handlers and the HTTP code of an HTTP server written in Go.

The testing/quick package

The Go standard library offers the `testing/quick` package, which can be used for **black-box testing** and is somewhat related to the `QuickCheck` package found in the **Haskell** programming language – both packages implement utility functions to help you with black-box testing. Go can generate random values of built-in types. The `testing/quick` package allows you to use these random values for testing, which saves you from having to generate all these values on your own.

A small program, which is saved as `randomBuiltIn.go` and illustrates how Go can generate random values, is the following:

```
package main

import (
    "fmt"
    "math/rand"
    "reflect"
    "testing/quick"
    "time"
)

func main() {
    type point3D struct {
        X, Y, Z int8
        S         float32
    }
    ran := rand.New(rand.NewSource(time.Now().Unix()))

    myValues := reflect.TypeOf(point3D{})
    _, _ := quick.Value(myValues, ran)
    fmt.Println(x)
}
```

First, we create a new random generator using `rand.New()` and then we use **reflection** to get information about the type of `point3D`. After that, we call `quick.Value()` from the `testing/quick` package with a type descriptor and a random number generator in order to put some random data into the `myValues` variable. Notice that in order to generate arbitrary values for structures, all the fields of the structure must be exported.

Executing `randomBuiltIn.go` will generate the following kind of output:

```
$ go run randomBuiltIn.go
{65 8 75 -3.3435536e+38}
$ go run randomBuiltIn.go
{-38 33 36 3.2604468e+38}
```

Notice that if you decide to create random strings, you will end up having Unicode strings with strange characters.

Now that you know how to create random values for built-in types, let us continue with the `testing/quick` package. For the purposes of this subsection, we are going to use two Go source files named `quick.go` and `quick_test.go`.

The Go code of `quick.go` is as follows:

```
package main

import (
    "fmt"
)

func Add(x, y uint16) uint16 {
    var i uint16
    for i = 0; i < x; i++ {
        y++
    }
    return y
}

func main() {
    fmt.Println(Add(0, 0))
}
```

The `add()` function in `quick.go` implements the addition of unsigned integers (`uint16`) using a `for` loop and it is used to illustrate black-box testing using the `testing/quick` package.

The code of `quick_test.go` will be the following:

```
package main

import (
    "testing"
    "testing/quick"
)

var N = 1000000

func TestWithSystem(t *testing.T) {
    condition := func(a, b uint16) bool {
        return Add(a, b) == (b + a)
    }
}
```

```

        err := quick.Check(condition, &quick.Config{MaxCount: N})
        if err != nil {
            t.Errorf("Error: %v", err)
        }
    }

func TestWithItself(t *testing.T) {
    condition := func(a, b uint16) bool {
        return Add(a, b) == Add(b, a)
    }

    err := quick.Check(condition, &quick.Config{MaxCount: N})
    if err != nil {
        t.Errorf("Error: %v", err)
    }
}

```

The two calls to `quick.Check()` automatically generate random numbers based on the signature of their first argument, which is a function defined earlier. There is no need to create these random input numbers on your own, which makes the code easy to read and write. The actual tests happen in the `condition` function in both cases.

Executing the tests will generate the following output:

```

$ go test -v quick*
--- RUN  TestWithSystem
--- PASS: TestWithSystem (8.36s)
--- RUN  TestWithItself
--- PASS: TestWithItself (17.41s)
PASS
ok    command-line-arguments      (cached)

```

If you do not want to use cached testing, you should execute `go test` as follows:

```

$ go test -v quick* -count=1
--- RUN  TestWithSystem
--- PASS: TestWithSystem (8.15s)
--- RUN  TestWithItself
--- PASS: TestWithItself (15.95s)
PASS
ok    command-line-arguments      24.104s

```

 *The idiomatic way to bypass test caching is to use `-count=1` with the `go test` command because `GOCACHE=off` no longer works with Go 1.12.*

TIP *The error message you will get if you try to use `GOCACHE=off` is `build cache is disabled by GOCACHE=off`, but required as of Go 1.12. You can learn more about it by executing `go help`*

What if testing takes too long or never finishes?

If the `go test` tool takes too much time to finish or for some reason it never ends, there is the `-timeout` parameter. In order to illustrate that, we are going to create a new Go program along with its tests.

The Go code of the `main` package, which is saved as `too_long.go`, is the following:

```
package main

import (
    "time"
)

func sleep_with_me() {
    time.Sleep(5 * time.Second)
}

func get_one() int {
    return 1
}

func get_two() int {
    return 2
}

func main() {
```

The testing functions, which are saved in `too_long_test.go`, will be as follows:

```
package main

import (
    "testing"
)

func Test_test_one(t *testing.T) {
    sleep_with_me()
    value := get_one()
    if value != 1 {
        t.Errorf("Function returned %v", value)
    }
    sleep_with_me()
}

func Test_test_two(t *testing.T) {
    sleep_with_me()
    value := get_two()
    if value != 2 {
        t.Errorf("Function returned %v", value)
    }
}
```

```

    }

func Test_that_will_fail(t *testing.T) {
    value := get_one()
    if value != 2 {
        t.Errorf("Function returned %v", value)
    }
}

```

The `Test_that_will_fail()` test function will always fail, whereas the other two functions are correct but slow.

Executing `go test` with and without the `-timeout` parameter will generate the following kinds of output:

```

$ go test too_long* -v
== RUN Test_test_one
--- PASS: Test_test_one (10.01s)
== RUN Test_test_two
--- PASS: Test_test_two (5.00s)
== RUN Test_that_will_fail
--- FAIL: Test_that_will_fail (0.00s)
    too_long_test.go:27: Function returned 1
FAIL
FAIL command-line-arguments 15.019s
$ go test too_long* -v -timeout 20s
== RUN Test_test_one
--- PASS: Test_test_one (10.01s)
== RUN Test_test_two
--- PASS: Test_test_two (5.01s)
== RUN Test_that_will_fail
--- FAIL: Test_that_will_fail (0.00s)
    too_long_test.go:27: Function returned 1
FAIL
FAIL command-line-arguments 15.021s
$ go test too_long* -v -timeout 15s
== RUN Test_test_one
--- PASS: Test_test_one (10.01s)
== RUN Test_test_two
panic: test timed out after 15s

goroutine 34 [running]:
testing.(*M).startAlarm.func1()
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:1334 +0xd
created by time.goFunc
    /usr/local/Cellar/go/1.12.4/libexec/src/time/sleep.go:169 +0x44

goroutine 1 [chan receive]:
testing.(*T).Run(0xc0000dc000, 0x113a46d, 0xd, 0x1141a08, 0x1069b01)
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:917 +0x381
testing.runTests.func1(0xc0000c0000)
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:1157 +0x78
testing.tRunner(0xc0000c0000, 0xc00009fe30)
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:865 +0xc0
testing.runTests(0xc0000ba000, 0x1230280, 0x3, 0x3, 0x0)
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:1155 +0x2a9
testing.(*M).Run(0xc0000a8000, 0x0)
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:1072 +0x162
main.main()
    _testmain.go:46 +0x13e

goroutine 5 [runnable]:
runtime.goparkunlock(...)

```

```
    /usr/local/Cellar/go/1.12.4/libexec/src/runtime/proc.go:307
time.Sleep(0x12a05f200)
    /usr/local/Cellar/go/1.12.4/libexec/src/runtime/time.go:105 +0x159
command-line-arguments.sleep_with_me(...)
    /Users/mtsouk/Desktop/mGo2nd/Mastering-Go-Second-Edition/ch11/long.go:8
command-line-arguments.Test_test_two(0xc0000dc000)
    /Users/mtsouk/Desktop/mGo2nd/Mastering-Go-Second-Edition/ch11/long_test.go:17 +0x31
testing.tRunner(0xc0000dc000, 0x1141a08)
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:865 +0xc0
created by testing.(*T).Run
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:916 +0x35a
FAIL    command-line-arguments    15.015s
```

Only the third `go test` command fails because it takes longer than 15 seconds to finish.

Benchmarking Go code

Benchmarking measures the performance of a function or program, allowing you to compare implementations and to understand the impact of changes you make to your code.

Using that information, you can easily reveal the part of the Go code that needs to be rewritten in order to improve its performance.



Never benchmark your Go code on a busy UNIX machine that is currently being used for other, more important, purposes unless you have a very good reason to do so! Otherwise, you will interfere with the benchmarking process and get inaccurate results.

Go follows certain conventions regarding benchmarking. The most important convention is that the name of a benchmark function must begin with `Benchmark`.

Once again, the `go test` subcommand is responsible for benchmarking a program. As a result, you still need to import the `testing` standard Go package and include benchmarking functions in Go files that end with `_test.go`.

A simple benchmarking example

In this section, I will show you a basic benchmarking example that will measure the performance of three algorithms that generate numbers belonging to the Fibonacci sequence. The good news is that such algorithms require lots of mathematical calculations, which makes them perfect candidates for benchmarking.

For the purposes of this section, I will create a new `main` package, which will be saved as `benchmarkMe.go` and presented in three parts.

The first part of `benchmarkMe.go` is as follows:

```
package main

import (
    "fmt"
)

func fibo1(n int) int {
    if n == 0 {
        return 0
    } else if n == 1 {
        return 1
    } else {
        return fibo1(n-1) + fibo1(n-2)
    }
}
```

The preceding code contains the implementation of the `fibo1()` function, which uses recursion in order to calculate numbers of the Fibonacci sequence. Although the algorithm works fine, this is a relatively simple and somewhat slow approach.

The second code segment of `benchmarkMe.go` is shown in the following Go code:

```
func fibo2(n int) int {
    if n == 0 || n == 1 {
        return n
    }
    return fibo2(n-1) + fibo2(n-2)
}
```

In this part, you see the implementation of the `fibo2()` function, which is almost identical to the `fibo1()` function that you saw earlier. However, it will be interesting to see whether a small code change – a single `if` statement as opposed to an `if else if` block – has any impact on the performance of the function.

The third code portion of `benchmarkMe.go` contains yet another implementation of a function that calculates numbers that belong to the Fibonacci sequence:

```
func fibo3(n int) int {
    fn := make(map[int]int)
    for i := 0; i <= n; i++ {
        var f int
        if i <= 2 {
            f = 1
        } else {
            f = fn[i-1] + fn[i-2]
        }
        fn[i] = f
    }
    return fn[n]
}
```

The `fibo3()` function presented here uses a totally new approach that requires a Go **map** and has a `for` loop. It remains to be seen whether this approach is indeed faster than the other two implementations. The algorithm presented in `fibo3()` will also be used in [Chapter 13, Network Programming – Building Your Own Servers and Clients](#), where it will be explained in greater detail. As you will see in a while, choosing an efficient algorithm can save you a lot of trouble!

The remaining code of `benchmarkMe.go` follows:

```
func main() {
    fmt.Println(fibo1(40))
    fmt.Println(fibo2(40))
    fmt.Println(fibo3(40))
}
```

Executing `benchmarkMe.go` will generate the following output:

```
$ go run benchmarkMe.go
102334155
102334155
102334155
```

The good news is that all three implementations returned the same number. Now it is time to add some benchmarks to `benchmarkMe.go` in order to understand the efficiency of each one of the three algorithms.

As the Go rules require, the version of `benchmarkMe.go` containing the benchmark functions is going to be saved as `benchmarkMe_test.go`. This program is presented in five parts. The first code segment of `benchmarkMe_test.go` contains the following Go code:

```
package main
import (
    "testing"
)

var result int

func benchmarkfibol(b *testing.B, n int) {
    var r int
    for i := 0; i < b.N; i++ {
        r = fibol(n)
    }
    result = r
}
```

In the preceding code, you can see the implementation of a function with a name that begins with the `benchmark` string instead of the `Benchmark` string. As a result, this function will not run automatically because it begins with a lowercase `b` instead of an uppercase `B`.

The reason for storing the result of `fibol(n)` in a variable named `r` and using another global variable named `result` afterward is tricky: this technique is used to prevent the compiler from performing any optimizations that will exclude the function that you want to measure from being executed because its results are never used. The same technique will be applied to the `benchmarkfibob()` and `benchmarkfiboc()` functions, which will be presented next.

The second part of `benchmarkMe_test.go` is shown in the following Go code:

```
func benchmarkfibob(b *testing.B, n int) {
    var r int
    for i := 0; i < b.N; i++ {
        r = fibob(n)
    }
    result = r
}
```

```

func benchmarkfibo3(b *testing.B, n int) {
    var r int
    for i := 0; i < b.N; i++ {
        r = fibo3(n)
    }
    result = r
}

```

The preceding code defines two more benchmark functions that will not run automatically because they begin with a lowercase `b` instead of an uppercase `B`.

Now, I will tell you a big secret: even if these three functions were named `BenchmarkFibol()`, `BenchmarkFibo2()`, and `BenchmarkFibo3()`, they would not have been invoked automatically by the `go test` command because their signature is not `func(*testing.B)`. So, that is the reason for naming them with a lowercase `b`. However, there is nothing that prevents you from invoking them from other benchmark functions afterward, as you will see shortly.

The third part of `benchmarkMe_test.go` follows:

```

func Benchmark30fibol(b *testing.B) {
    benchmarkfibol(b, 30)
}

```

This is a correct benchmark function with the correct name and the correct signature, which means that it will be executed by `go tool`.

Notice that although `Benchmark30fibol()` is a valid benchmark function name, `BenchmarkfibolII()` is not because there is no uppercase letter or a number after the `Benchmark` string. This is very important because a benchmark function with an incorrect name will not get executed automatically. The same rule applies to test functions.

The fourth code segment of `benchmarkMe_test.go` contains the following Go code:

```

func Benchmark30fibo2(b *testing.B) {
    benchmarkfibo2(b, 30)
}

func Benchmark30fibo3(b *testing.B) {
    benchmarkfibo3(b, 30)
}

```

Both the `Benchmark30fibo2()` and `Benchmark30fibo3()` benchmark functions are similar to `Benchmark30fibo1()`.

The last part of `benchmarkMe_test.go` is as follows:

```
func Benchmark50fibo1(b *testing.B) {
    benchmarkfibo1(b, 50)
}
func Benchmark50fibo2(b *testing.B) {
    benchmarkfibo2(b, 50)
}
func Benchmark50fibo3(b *testing.B) {
    benchmarkfibo3(b, 50)
}
```

In this part, you see three additional benchmark functions that calculate the 50th number in the Fibonacci sequence.



Remember that each benchmark is executed for at least one second by default. If the benchmark function returns in a time that is less than one second, the value of `b.N` is increased, and the function is run again. The first time the value of `b.N` is one, then it becomes two, then five, then 10, then 20, then 50, and so on. This happens because the faster the function is, the more times you need to run it to get accurate results.

Executing `benchmarkMe_test.go` will generate the following output:

```
$ go test -bench=. benchmarkMe.go benchmarkMe_test.go
goos: darwin
goarch: amd64
Benchmark30fibo1-8      300      4494213 ns/op
Benchmark30fibo2-8      300      4463607 ns/op
Benchmark30fibo3-8     500000      2829 ns/op
Benchmark50fibo1-8       1      67272089954 ns/op
Benchmark50fibo2-8       1      67300080137 ns/op
Benchmark50fibo3-8     300000      4138 ns/op
PASS
ok   command-line-arguments   145.827s
```

There are two important points here: first, the value of the `-bench` parameter specifies the benchmark functions that are going to be executed. The `.` value used is a regular expression that matches all valid benchmark functions. The second point is that if you omit the `-bench` parameter, no benchmark function will be executed.

So, what does this output tell us? First of all, the `-8` at the end of each benchmark function (`Benchmark10fibo1-8`) signifies the number of goroutines

used during its execution, which is essentially the value of the `GOMAXPROCS` environment variable. You will recall that we talked about the `GOMAXPROCS` environment variable back in [Chapter 10, Concurrency in Go – Advanced Topics](#). Similarly, you can see the values of `GOOS` and `GOARCH`, which show the operating system and the architecture of your machine.

The second column in the output displays the number of times that the relevant function was executed. Faster functions are executed more times than slower functions. As an example, the `Benchmark30fib03()` function was executed 500,000 times, while the `Benchmark50fib02()` function was executed only once! The third column in the output shows the average time of each run.

As you can see, the `fib01()` and `fib02()` functions are really slow compared to the `fib03()` function. Should you wish to include memory allocation statistics in the output, you can execute the following command:

```
$ go test -benchmem -bench=. benchmarkMe.go benchmarkMe_test.go
goos: darwin
goarch: amd64
Benchmark30fib01-8 300      4413791 ns/op      0 B/op      0 allocs/op
Benchmark30fib02-8 300      4430097 ns/op      0 B/op      0 allocs/op
Benchmark30fib03-8 500000    2774 ns/op      2236 B/op      6 allocs/op
Benchmark50fib01-8 1        71534648696 ns/op     0 B/op      0 allocs/op
Benchmark50fib02-8 1        72551120174 ns/op     0 B/op      0 allocs/op
Benchmark50fib03-8 300000    4612 ns/op      2481 B/op     10 allocs/op
PASS
ok   command-line-arguments   150.500s
```

The preceding output is similar to the one without the `-benchmem` command-line parameter, but it includes two more columns in its output. The fourth column shows the amount of memory that was allocated on average in each execution of the benchmark function. The fifth column shows the number of allocations used to allocate the memory value of the fourth column. So, `Benchmark50fib03()` allocated 2,481 bytes in 10 allocations on average.

As you can understand, the `fib01()` and `fib02()` functions do not need any special kind of memory apart from what is expected, because neither of them use any kind of data structure, which is not the case with `fib03()`,

which uses a map variable; hence the larger-than-zero values in both the fourth and fifth columns of the output of `Benchmark10fibo3-8`.

Wrongly defined benchmark functions

Look at the Go code of the following benchmark function:

```
func BenchmarkFiboI(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = fibo1(i)
    }
}
```

The `BenchmarkFibo()` function has a valid name and the correct signature. The bad news, however, is that this benchmark function is wrong, and you will not get any output from it after executing the `go test` command.

The reason for this is that as the `b.N` value grows in the way described earlier, the runtime of the benchmark function will also increase because of the `for` loop. This fact prevents `BenchmarkFiboI()` from converging to a stable number, which prevents the function from completing and therefore returning.

For analogous reasons, the next benchmark function is also wrongly implemented:

```
func BenchmarkfiboII(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = fibo2(b.N)
    }
}
```

On the other hand, there is nothing wrong with the implementation of the following two benchmark functions:

```
func BenchmarkFiboIV(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = fibo3(10)
    }
}

func BenchmarkFiboIII(b *testing.B) {
```

```
| }     - = fibo3(b.N)
```

Benchmarking buffered writing

In this section, we will explore how the size of the write buffer affects the performance of the entire writing operation using the Go code of `writingBU.go`, which will be presented in five parts.

The `writingBU.go` program generates dummy files with randomly generated data. The variables of the program are the size of the buffer and the size of the output file.

The first part of `writingBU.go` is as follows:

```
package main

import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
)

var BUFSIZE int
var FILESIZE int

func random(min, max int) int {
    return rand.Intn(max-min) + min
}
```

The second code portion of `writingBU.go` contains the following Go code:

```
func createBuffer(buf *[]byte, count int) {
    *buf = make([]byte, count)
    if count == 0 {
        return
    }
    for i := 0; i < count; i++ {
        intByte := byte(random(0, 100))
        if len(*buf) > count {
            return
        }
        *buf = append(*buf, intByte)
    }
}
```

The third part of `writingBU.go` is shown in the following Go code:

```

func Create(dst string, b, f int) error {
    _, err := os.Stat(dst)
    if err == nil {
        return fmt.Errorf("File %s already exists.", dst)
    }
    destination, err := os.Create(dst)
    if err != nil {
        return err
    }
    defer destination.Close()
    if err != nil {
        panic(err)
    }
    buf := make([]byte, 0)
    for {
        createBuffer(&buf, b)
        buf = buf[:b]
        if _, err := destination.Write(buf); err != nil {
            return err
        }
        if f < 0 {
            break
        }
        f = f - len(buf)
    }
    return err
}

```

The `Create()` function does all of the work in the program, and it is this function that needs to be benchmarked.

Note that if the buffer size and the file size were not part of the signature of the `Create()` function, you would have a problem writing a benchmark function for `Create()` because you would be required to use the `BUFFERSIZE` and `FILESIZE` global variables, which are both initialized in the `main()` function of `writingBU.go`. This would be difficult to do in the `writingBU_test.go` file. This means that in order to create a benchmark for a function, you should think about it when you are writing your code.

The fourth code segment of `writingBU.go` is as follows:

```

func main() {
    if len(os.Args) != 3 {
        fmt.Println("Need BUFFERSIZE FILESIZE!")
        return
    }

    output := "/tmp/randomFile"
    BUFFERSIZE, _ = strconv.Atoi(os.Args[1])
    FILESIZE, _ = strconv.Atoi(os.Args[2])
    err := Create(output, BUFFERSIZE, FILESIZE)
    if err != nil {

```

```
    }  
    fmt.Println(err)  
}
```

The remaining Go code of `writingBU.go` follows:

```
    err = os.Remove(output)  
    if err != nil {  
        fmt.Println(err)  
    }  
}
```

Although the `os.Remove()` call that deletes the temporary file is inside of the `main()` function, which is not called by the benchmark functions, it is easy to call `os.Remove()` from the benchmark functions, so there is no problem here.

Executing `writingBU.go` twice on a macOS Mojave machine with an SSD hard disk using the `time(1)` utility to check the speed of the program generates the following output:

```
$ time go run writingBU.go 1 100000  
real 0m1.193s  
user 0m0.349s  
sys 0m0.809s  
$ time go run writingBU.go 10 100000  
real 0m0.283s  
user 0m0.195s  
sys 0m0.228s
```

So, although it is obvious that the size of the write buffer plays a key role in the performance of the program, we need to be much more specific and accurate. Therefore, we will write benchmark functions that will be stored in `writingBU_test.go`.

The first part of `writingBU_test.go` is shown in the following Go code:

```
package main  
import (  
    "fmt"  
    "os"  
    "testing"  
)  
var ERR error  
  
func benchmarkCreate(b *testing.B, buffer, filesize int) {  
    var err error  
    for i := 0; i < b.N; i++ {  
        err = Create("/tmp/random", buffer, filesize)  
    }
```

```

    ERR = err
    err = os.Remove("/tmp/random")
    if err != nil {
        fmt.Println(err)
    }
}

```

As you will recall, this is not a valid benchmark function.

The second code segment of `writingBU_test.go` is as follows:

```

func Benchmark1Create(b *testing.B) {
    benchmarkCreate(b, 1, 1000000)
}

func Benchmark2Create(b *testing.B) {
    benchmarkCreate(b, 2, 1000000)
}

```

The remaining code of `writingBU_test.go` is as follows:

```

func Benchmark4Create(b *testing.B) {
    benchmarkCreate(b, 4, 1000000)
}

func Benchmark10Create(b *testing.B) {
    benchmarkCreate(b, 10, 1000000)
}

func Benchmark1000Create(b *testing.B) {
    benchmarkCreate(b, 1000, 1000000)
}

```

Here, we created five benchmark functions that are going to check the performance of the `benchmarkCreate()` function, which checks the performance of the `Create()` function for various write buffer sizes.

Executing `go test` on both the `writingBU.go` and `writingBU_test.go` files will generate the following type of output:

```

$ go test -bench=. writingBU.go writingBU_test.go
goos: darwin
goarch: amd64
Benchmark1Create-8          1    6001864841 ns/op
Benchmark2Create-8          1    3063250578 ns/op
Benchmark4Create-8          1    1557464132 ns/op
Benchmark10Create-8         1000000      11136 ns/op
Benchmark1000Create-8       2000000      5532 ns/op
PASS
ok   command-line-arguments  21.847s

```

The following output also checks the memory allocations of the benchmark functions:

```
$ go test -bench=. writingBU.go writingBU_test.go -benchmem
goos: darwin
goarch: amd64
Benchmark1Create-8 1    6209493161 ns/op    16000840 B/op    2000017 allocs/op
Benchmark2Create-8 1    3177139645 ns/op    8000584 B/op    1000013 allocs/op
Benchmark4Create 1     1632772604 ns/op    4000424 B/op    500011 allocs/op
Benchmark10Create-8 100000  11238 ns/op    336 B/op      7 allocs/op
Benchmark1000Create-8 200000 5122 ns/op    303 B/op      5 allocs/op
PASS
ok    command-line-arguments    24.031s
```

It is now time to interpret the output of the two `go test` commands.

It is extremely obvious that using a write buffer with a size of 1 byte is totally inefficient and slows everything down. Additionally, such a buffer size requires too many memory operations, which slows down the program even more.

Using a write buffer with 2 bytes makes the entire program twice as fast, which is a good thing. However, it is still very slow. The same applies to a write buffer with a size of 4 bytes. Where things get much faster is when we decide to use a write buffer with a size of 10 bytes. Finally, the results show that using a write buffer with a size of 1,000 bytes does not make things 100 times faster than when using a buffer size of 10 bytes, which means that the sweet spot between speed and write buffer size is somewhere between these two buffer size values.

Finding unreachable Go code

Go code that cannot be executed is a logical error, and therefore it is pretty difficult to reveal it with developers or a normal execution of the Go compiler. Put simply, there is nothing wrong with unreachable code, apart from the fact that there is no way for this code to get executed.

Take a look at the following Go code, which is saved as `cannotReach.go`:

```
package main
import (
    "fmt"
)

func f1() int {
    fmt.Println("Entering f1()")
    return -10
    fmt.Println("Exiting f1()")
    return -1
}

func f2() int {
    if true {
        return 10
    }
    fmt.Println("Exiting f2()")
    return 0
}

func main() {
    fmt.Println(f1())
    fmt.Println("Exiting program...")
}
```

There is nothing syntactically incorrect with the Go code of `cannotReach.go`. As a result, you can execute `cannotReach.go` without getting any error messages from the compiler:

```
$ go run cannotReach.go
Entering f1()
-1
Exiting program...
```

Note that `f2()` is never used in the program. However, it is obvious that the following Go code of `f2()` never gets executed because the condition in the

preceding `if` is always `true`:

```
|     fmt.Println("Exiting f2()")  
|     return 0
```

So, what can you do about it? You can execute `go vet` as follows:

```
| $ go vet cannotReach.go  
| # command-line-arguments  
| ./cannotReach.go:10:2: unreachable code
```

The output tells us that there is unreachable code in line 10 of the program. Now let's remove the `return -10` statement from the `f1()` function and rerun `go vet`:

```
| $ go vet cannotReach.go
```

Here, there are no new error messages despite the fact that there is still unreachable code in the `f2()` function. This means that `go vet` cannot catch every possible type of logical error.

Cross-compilation

Cross-compilation is the process of generating a binary executable file for a different architecture than the one on which you are working.

The main benefit that you receive from cross-compilation is that you do not need a second or third machine to create executable files for different architectures. This means that you basically need just a single machine for your development. Fortunately, Go has built-in support for cross-compilation.

For the purpose of this section, we are going to use the Go code of `xCompile.go` to illustrate the cross-compilation process. The Go code of `xCompile.go` is as follows:

```
package main

import (
    "fmt"
    "runtime"
)

func main() {
    fmt.Println("You are using ", runtime.Compiler, " ")
    fmt.Println("on a", runtime.GOARCH, "machine")
    fmt.Println("with Go version", runtime.Version())
}
```

Running `xCompile.go` on a macOS Mojave machine generates the following output:

```
$ go run xCompile.go
You are using gc on a amd64 machine
with Go version go1.12.4
```

In order to cross compile a Go source file, you will need to set the `GOOS` and `GOARCH` environment variables to the target operating system and architecture, respectively, which is not as difficult as it sounds.

So, the cross-compilation process goes like this:

```
$ env GOOS=linux GOARCH=arm go build xCompile.go
$ file xCompile
xCompile: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically linked, not stripped
$ ./xCompile
-bash: ./xCompile: cannot execute binary file
```

The first command generates a binary file that works on Linux machines that use the ARM architecture, while the output of `file(1)` verifies that the generated binary file is indeed for a different architecture.

As the Debian Linux machine that will be used for this example has an Intel processor, we will have to execute the `go build` command once more using the correct `GOARCH` value:

```
$ env GOOS=linux GOARCH=386 go build xCompile.go
$ file xCompile
xCompile: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, with debug_info, not stripped
```

Executing the generated binary executable file on a Linux machine will produce the following expected output:

```
$ ./xCompile
You are using gc on a 386 machine
with Go version go1.12.4
$ go version
```

```
| go version go1.3.3 linux/amd64
| $ go run xCompile.go
| You are using gc on a amd64 machine
| with Go version go1.3.3
```

One thing to notice here is that the cross-compiled binary file of `xCompile.go` prints the Go version of the machine used for compiling it. The second thing to notice is that the architecture of the Linux machine is actually `amd64` instead of `386`, which was used in the cross-compilation process.



You can find a list of the available values for the `GOOS` and `GOARCH` environment variables at <https://golang.org/doc/install/source>. Keep in mind, however, that not all `GOOS` and `GOARCH` pairs are valid.

Creating example functions

Part of the documentation process is generating example code that showcases the use of some or all of the functions and types of a package. Example functions have many benefits, including the fact that they are executable tests that are executed by `go test`. Therefore, if an example function contains an `// Output:` line, the `go test` tool will check whether the calculated output matches the values found after the `// Output:` line.

Additionally, examples are really useful when seen in the documentation of the package, which is the subject of the next section. Finally, example functions that are presented on the Go documentation server (https://golang.org/pkg/io/#example_Copy) allow the reader of the documentation to experiment with the example code. The Go playground at <https://play.golang.org/> also supports this functionality.

As the `go test` subcommand is responsible for the examples of a program, you will need to import the `testing` standard Go package and include example functions in Go files that end with `_test.go`. Moreover, the name of each example function must begin with `Example`. Lastly, **example functions** take no input parameters and return no results.

Now let's create some example functions for the following package, which is saved as `ex.go`:

```
package ex
func F1(n int) int {
    if n == 0 {
        return 0
    }
    if n == 1 || n == 2 {
        return 1
    }
    return F1(n-1) + F1(n-2)
}
func S1(s string) int {
    return len(s)
}
```

The `ex.go` source file contains the implementation of two functions named `F1()` and `S1()`.

Notice that `ex.go` does not need to import the `fmt` package.

As you know, the example functions will be included in the `ex_test.go` file, which will be presented in three parts.

The first part of `ex_test.go` is as follows:

```
package ex
import (
    "fmt"
)
```

The second code portion of `ex_test.go` is shown in the following Go code:

```
func ExampleF1() {
    fmt.Println(F1(10))
    fmt.Println(F1(2))
    // Output:
    // 55
    // 1
}
```

The remaining Go code of `ex_test.go` follows:

```
func ExampleS1() {
    fmt.Println(S1("123456789"))
    fmt.Println(S1(""))
    // Output:
    // 8
    // 0
}
```

Executing the `go test` command on the `ex.go` package will generate the following type of output:

```
$ go test ex.go ex_test.go -v
=== RUN ExampleF1
--- PASS: ExampleF1 (0.00s)
=== RUN ExampleS1
--- FAIL: ExampleS1 (0.00s)
got:
9
0
want:
8
0
```

FAIL		
FAIL	command-line-arguments	0.006s

You will observe that the preceding output tells us that there is something wrong with the `s1()` function based on the data after the `// output: comment`.

From Go code to machine code

In this section, you will learn how Go code is converted into machine code in much more detail. The Go program that is going to be used in this example is named `machineCode.go` and it is going to be presented in two parts.

The first part of `machineCode.go` is as follows:

```
package main

import (
    "fmt"
)

func hello() {
    fmt.Println("Hello!")
}
```

The second and final part of `machineCode.go` is as follows:

```
func main() {
    hello()
}
```

After that, you can see `machineCode.go` translated into machine code as follows:

```
$ GOSSAFUNC=main GOOS=linux GOARCH=amd64 go build -gcflags "-S" machineCode.go
# runtime
dumped SSA to /usr/local/Cellar/go/1.12.4/libexec/src/runtime/ssa.html
# command-line-arguments
dumped SSA to ./ssa.html
os.(*File).close STEXT dupok nosplit size=26 args=0x18 locals=0x0
    0x0000 00000 (<autogenerated>:1) TEXT os.(*File).close(SB), DUPOK|NOSPLIT|ABIInternal, $0-24
    0x0000 00000 (<autogenerated>:1) FUNCDATA $0, glocals e6397a44f8e1b6e77d0f200b4fba5269(SB)
    0x0000 00000 (<autogenerated>:1) FUNCDATA $1, glocals 69c1753bd5f81501d95132d08af04464(SB)
    0x0000 00000 (<autogenerated>:1) FUNCDATA $3, glocals 9fb7f0986f647f17cb5
...
...
```

The first lines of the output tell us that there are two files that contain all the generated output: `/usr/local/Cellar/go/1.12.4/libexec/src/runtime/ssa.html` and `./ssa.html`. Notice that if you have a different Go version or a different Go installation, the first file will be located at a different place. **Static Single Assignment (SSA)** form is a method for describing low-level operations in a way that is pretty close to machine instructions. However, notice that SSA acts as if it has an infinite number of registers, which is not the case with machine code.

Opening that file on your favorite web browser will present a plethora of useful yet low-level information about the program. Lastly, notice that the parameter to `GOSSAFUNC` is the Go function that you want to **disassemble**.

Unfortunately, talking more about SSA is beyond the scope of this book.

Using assembly with Go

In this section, we are going to talk about **assembly** and Go, and how you can use assembly to implement Go functions. We will begin with the following Go program, which is saved as `add_me.go`:

```
package main

import (
    "fmt"
)

func add(x, y int64) int64 {
    return x + y
}

func main() {
    fmt.Println(add(1, 2))
}
```

Executing the following command will reveal the assembly implementation of the `add()` function, which is the following:

```
$ GOOS=darwin GOARCH=amd64 go tool compile -S add_me.go
"".add STEXT nosplit size=19 args=0x18 locals=0x0
    0x0000 00000 (add_me.go:7)    TEXT    """.add(SB), NOSPLIT|ABIInternal, $0-24
    0x0000 00000 (add_me.go:7)    FUNCDATA   $0, glocals 33cdeccccebe80329f1fdbbee7f5874cb(SB)
    0x0000 00000 (add_me.go:7)    FUNCDATA   $1, glocals 33cdeccccebe80329f1fdbbee7f5874cb(SB)
    0x0000 00000 (add_me.go:7)    FUNCDATA   $3, glocals 33cdeccccebe80329f1fdbbee7f5874cb(SB)
    0x0000 00000 (add_me.go:8)    PCDATA    $2, $0
    0x0000 00000 (add_me.go:8)    PCDATA    $0, $0
    0x0000 00000 (add_me.go:8)    MOVQ     """.y+16(SP), AX
    0x0005 00005 (add_me.go:8)    MOVQ     """.x+8(SP), CX
    0x000a 00010 (add_me.go:8)    ADDQ     CX, AX
    0x000d 00013 (add_me.go:8)    MOVQ     AX, """.~r2+24(SP)
    0x0012 00018 (add_me.go:8)    RET
    0x0000 48 8b 44 24 10 48 8b 4c 24 08 48 01 c8 48 89 44 H.D$.H.L$.H..H.D
        0x0010 24 18 c3
    "".main STEXT size=150 args=0x0 locals=0x58
$..
```

The last line is not part of the assembly implementation of the `add()` function. Additionally, the `FUNCDATA` lines have nothing to do with the assembly implementation of the function and are added by the Go compiler.

Now, we are going to make some changes to the previous assembly code and make it look as follows:

```
TEXT add(SB),$0
    MOVQ x+0(FP), BX
    MOVQ y+8(FP), BP
    ADDQ BP, BX
    MOVQ BX, ret+16(FP)
    RET
```

The assembly code will be saved in a file named `add_amd64.s` in order to be used as the implementation of the `add()` function.

The new version of `add_me.go` will be as follows:

```
package main

import (
    "fmt"
)

func add(x, y int64) int64

func main() {
    fmt.Println(add(1, 2))
}
```

This means that `add_me.go` will use the assembly implementation of the `add()` function. Using the assembly implementation of the `add()` function is as simple as this:

```
$ go build
$ ls -l
total 4136
-rw-r--r--@ 1 mtsouk  staff      93 Apr 18 22:49 add_amd64.s
-rw-r--r--@ 1 mtsouk  staff     101 Apr 18 22:59 add_me.go
-rwxr-xr-x  1 mtsouk  staff  2108072 Apr 18 23:00 asm
$ ./asm
3
$ file asm
asm: Mach-O 64-bit executable x86_64
```

The only tricky point here is that the assembly code is not portable. Unfortunately, talking more about assembly and Go is beyond the scope of the book – you should look at the assembly reference of your own CPU and architecture.

Generating documentation

Go offers the `godoc` tool, which allows you to view the documentation of your packages – provided that you have included some extra information in your files.



The general advice is that you should try to document everything but leave out obvious things. Put simply, do not say, "Here, I create a new `int` variable." It would be better to state the use of that `int` variable! Nevertheless, really good code does not usually need documentation.

The rule about writing documentation in Go is pretty simple and straightforward: in order to document something, you have to put one or more regular comment lines that start with `//` just before its declaration. This convention can be used to document functions, variables, constants, or even packages.

Additionally, you will notice that the first line of the documentation of a package of any size will appear in the package list of `godoc`, as happens in [http://golang.org/pkg/](https://golang.org/pkg/). This means that your description should be pretty expressive and complete.

Keep in mind that comments that begin with `BUG(something)` will appear in the `Bugs` section of the documentation of a package, even if they do not precede a declaration.

If you are looking for such an example, you can visit the source code and the documentation page of the `bytes` package, which can be found at: <https://golang.org/src/bytes/bytes.go> and <https://golang.org/pkg/bytes/>, respectively. Lastly, all comments that are not related to a top-level declaration are omitted from the output that is generated by the `godoc` utility.

Take a look at the following Go code, which is saved as `documentMe.go`:

```
// This package is for showcasing the documentation capabilities of Go
// It is a naive package!
package documentMe
// Pie is a global variable
```

```

// This is a silly comment!
const Pie = 3.1415912
// The S1() function finds the length of a string
// It iterates over the string using range
func S1(s string) int {
    if s == "" {
        return 0
    }
    n := 0
    for range s {
        n++
    }
    return n
}
// The F1() function returns the double value of its input integer
// A better function name would have been Double()!
func F1(n int) int {
    return 2 * n
}

```

As discussed in the previous section, we will need to create a `documentMe_test.go` file in order to develop example functions for it. The contents of `documentMe_test.go` follow:

```

package documentMe
import (
    "fmt"
)
func ExampleS1() {
    fmt.Println(S1("123456789"))
    fmt.Println(S1(""))
    // Output:
    // 9
    // 0
}

func ExampleF1() {
    fmt.Println(F1(10))
    fmt.Println(F1(2))
    // Output:
    // 1
    // 55
}

```

In order to be able to see the documentation of `documentMe.go`, you will need to install the package on your local machine as you learned back in [Chapter 6, *What You Might Not Know About Go Packages and Functions*](#). This will require the execution of the following commands from your favorite UNIX shell:

```

$ mkdir ~/go/src/documentMe
$ cp documentMe* ~/go/src/documentMe/
$ ls -l ~/go/src/documentMe/
total 16

```

```
-rw-r--r--@ 1 mtsouk staff 542 Mar 6 21:11 documentMe.go
-rw-r--r--@ 1 mtsouk staff 223 Mar 6 21:11 documentMe_test.go
$ go install documentMe
$ cd ~/go/pkg/darwin_amd64
$ ls -l documentMe.a
-rw-r--r-- 1 mtsouk staff 1626 Mar 6 21:11 documentMe.a
```

Next, you should execute the `godoc` utility as follows:

```
$ godoc -http=:8080"
```

Note that if the port is already in use and the user has root privileges, the error message you will get will be the following:

```
$ godoc -http=:22"
2019/08/19 15:18:21 ListenAndServe :22: listen tcp :22: bind: address already in use
```

On the other hand, if the user does not have root privileges, the error message will be the following (even if the port is already in use):

```
$ godoc -http=:22"
2019/03/06 21:03:05 ListenAndServe :22: listen tcp :22: bind: permission denied
```

After taking care of that, you will be able to browse the HTML documentation created using your favorite web browser. The URL that will take you to the documentation is <http://localhost:8080/pkg/>.

The following figure shows the root directory of the `godoc` server that we just started. There, you can see the `documentMe` package that you created in `documentMe.go` among the other Go packages:

Packages - The Go Programming Language

localhost:8080/pkg/ Mihalis

pkix Package pkix contains shared, low level structures used for ASN.1 parsing and serialization of X.509 certificates, CRL and OCSP.

database

sql Package sql provides a generic interface around SQL (or SQL-like) databases.

driver Package driver defines interfaces to be implemented by database drivers as used by package sql.

debug

dwarf Package dwarf provides access to DWARF debugging information loaded from executable files, as defined in the DWARF 2.0 Standard at <http://dwarfstd.org/doc/dwarf-2.0.0.pdf>

elf Package elf implements access to ELF object files.

gosym Package gosym implements access to the Go symbol and line number tables embedded in Go binaries generated by the gc compilers.

macho Package macho implements access to Mach-O object files.

pe Package pe implements access to PE (Microsoft Windows Portable Executable) files.

plan9obj Package plan9obj implements access to Plan 9 a.out object files.

documentMe This package is for showcasing the documentation capabilities of Go. It is a naive package!

encoding Package encoding defines interfaces shared by other packages that convert data to and from byte-level and textual representations.

ascii85 Package ascii85 implements the ascii85 data encoding as used in the btoa tool and Adobe's PostScript and PDF document formats.

asn1 Package asn1 implements parsing of DER-encoded ASN.1 data structures, as defined in ITU-T Rec X.690.

base32 Package base32 implements base32 encoding as specified by RFC 4648.

base64 Package base64 implements base64 encoding as specified by RFC 4648.

binary Package binary implements simple translation between numbers and byte sequences and encoding and decoding of varints.

csv Package csv reads and writes comma-separated values (CSV) files.

gob Package gob manages streams of gob - binary values exchanged between an Encoder (transmitter) and a Decoder (receiver).

hex Package hex implements hexadecimal encoding and decoding.

json Package json implements encoding and decoding of JSON as defined in RFC 7159.

pem Package pem implements the PEM data encoding, which originated in Privacy Enhanced Mail.

xml Package xml implements a simple XML 1.0 parser that understands XML name spaces.

errors Package errors implements functions to manipulate errors.

expvar Package expvar provides a standardized interface to public variables, such as operation counters in servers.

flag Package flag implements command-line flag parsing.

fmt Package fmt implements formatted I/O with functions analogous to C's printf and scanf.

github.com

Arafatk

glot

mattn

go-sqlite3 Package sqlite3 provides interface to SQLite3 databases.

tool

go

ast Package ast declares the types used to represent syntax trees for Go packages

Figure 11.6: The root directory of the godoc server

The following figure shows the root directory of the documentation of the `documentMe` package implemented in the `documentMe.go` source file:

The Go Programming Language

Documents Packages The Project Help Blog Search

Package documentMe

```
import "documentMe"
```

Overview Index Examples

Overview ▾

This package is for showcasing the documentation capabilities of Go. It is a naive package!

Index ▾

Constants

```
func F1(n int) int
func S1(s string) int
```

Examples

F1
S1

Package files

documentMe.go

Constants

Pie is a global variable. This is a silly comment!

```
const Pie = 3.1415912
```

func F1

```
func F1(n int) int
```

Figure 11.7: The root page of the documentMe.go file

Similarly, the following figure shows the documentation of the `s1()` function of the `documentMe.go` package in greater detail, which also includes the example code:

The screenshot shows a web browser window titled "documentMe - The Go Program". The address bar displays "localhost:8080/pkg/documentMe/". The main content area shows the Go code for two functions:

```
func F1(n int) int
```

The F1() function returns the double value of its input integer A better function name would have been Double()!

▼ Example

Code:

```
fmt.Println(F1(10))
fmt.Println(F1(2))
```

Output:

```
1
55
```

func S1

```
func S1(s string) int
```

The S1() function finds the length of a string It iterates over the string using range

▼ Example

Code:

```
fmt.Println(S1("123456789"))
fmt.Println(S1(""))
```

Output:

```
9
0
```

Build version go1.10.

Except as noted, the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Figure 11.8: The documentation page and the example of the S1() function

Executing the `go test` command will generate the following output, which might reveal potential problems and bugs in our code:

```
$ go test -v documentMe*
==== RUN ExampleS1
--- PASS: ExampleS1 (0.00s)
==== RUN ExampleF1
--- FAIL: ExampleF1 (0.00s)
got:
20
4
want:
1
55
FAIL
FAIL    command-line-arguments    0.005s
```

Using Docker images

In this section, you will learn how to create a Docker image that contains both PostgreSQL and Go. The image will not be perfect, but it will illustrate how this can be done. As you already know, everything should begin with a `Dockerfile`, which in this case will be the following:

```
FROM ubuntu:18.04

RUN apt-get update && apt-get install -y gnupg
RUN apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys B97B0AFCAA1A47F044F244A07FCC7D46ACCC4CF8
RUN echo "deb http://apt.postgresql.org/pub/repos/apt/ precise-pgdg main" > /etc/apt/sources.list.d/pgdg.list
RUN apt-get update && apt-get install -y software-properties-common postgresql-9.3 postgresql-client-9.3 postgresql-contrib-9.3
RUN apt-get update && apt-get install -y git golang vim

USER postgres
RUN /etc/init.d/postgresql start &&
    psql --command "CREATE USER docker WITH SUPERUSER PASSWORD 'docker';" &&
    createdb -O docker docker

RUN echo "host all all 0.0.0.0/0 md5" >> /etc/postgresql/9.3/main/pg_hba.conf
RUN echo "listen_addresses='*'" >> /etc/postgresql/9.3/main/postgresql.conf

USER root
RUN mkdir files
COPY webServer.go files
WORKDIR files
RUN go get github.com/lib/pq
RUN go build webServer.go
RUN ls -l

USER postgres
CMD ["/usr/lib/postgresql/9.3/bin/postgres", "-D", "/var/lib/postgresql/9.3/main", "-c", "config_file=/etc/postgresql/9.3/main/pg_hba.conf"]
```

This is a relatively complex `Dockerfile` that begins with a base Docker image from Docker Hub and downloads additional software – this is more or less the way to build a custom Docker image that fits your needs.

Executing `Dockerfile` will generate lots of output, including the following:

```
$ docker build -t go_postgres .
Sending build context to Docker daemon 9.216kB
Step 1/19 : FROM ubuntu:18.04
--> 94e814e2efa8
Step 2/19 : RUN apt-get update && apt-get install -y gnupg
...
Step 15/19 : RUN go get github.com/lib/pq
--> Running in 17aedelc97d8
Removing intermediate container 17aedelc97d8
--> 1878408c6e06
Step 16/19 : RUN go build webServer.go
--> Running in 39cf8aaf63d5
Removing intermediate container 39cf8aaf63d5
--> 1b4d638242ae
...
Removing intermediate container 9af6a391c1e4
--> c24937079367
Successfully built c24937079367
Successfully tagged go_postgres:latest
```

In the last part of the `Dockerfile`, we also build the executable file from `webServer.go` using the operating system of the Docker image, which is Ubuntu Linux.

Executing `docker images` will generate the following kind of output:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
go_postgres	latest	c24937079367	34 seconds ago	831MB

Now we can execute that Docker image and connect to its `bash(1)` shell as follows:

```
$ docker run --name=my_go -dt go_postgres:latest
23de1ab0d3f5517d5dbf8c599a68075574a8ed9217aa3cb4899ea2f92412a833
$ docker exec -it my_go bash
postgres@23de1ab0d3f5:/files$
```

Additional resources

Visit the following web links:

- Visit the web site of Graphviz at <http://graphviz.org>.
- Visit the documentation page of the `testing` package, which can be found at <https://golang.org/pkg/testing/>.
- If you are not familiar with Donald Knuth and his work, you can learn more about him at https://en.wikipedia.org/wiki/Donald_Knuth.
- You can learn more about the `godoc` utility by visiting its documentation page at <https://godoc.org/golang.org/x/tools/cmd/godoc>.
- Visit the documentation page of the `runtime/pprof` standard Go package, which can be found at <https://golang.org/pkg/runtime/pprof/>.
- You can view the Go code of the `net/http/pprof` package by visiting [http://golang.org/src/net/http/pprof/pprof.go](https://golang.org/src/net/http/pprof/pprof.go).
- You can find the documentation page of the `net/http/pprof` package at <https://golang.org/pkg/net/http/pprof/>.
- You can learn more about the `pprof` tool by visiting its development page at <https://github.com/google/pprof>.
- Watch the "Advanced Testing with Go" video from GopherCon 2017 by Mitchell Hashimoto: <https://www.youtube.com/watch?v=8hQG7Q1cLBk>.
- You can find the source code of the `testing` package at <https://golang.org/src/testing/testing.go>.
- You can find more information about the `testing/quick` package by visiting <https://golang.org/pkg/testing/quick/>.
- You can learn more about the `profile` package by visiting its web page at <https://github.com/pkg/profile>.
- A "Manual for the Plan 9 assembler": <https://9p.io/sys/doc/asm.html>.
- The documentation page of the `arm64` package can be found at <https://golang.org/pkg/cmd/internal/obj/arm64/>.
- You can learn more about Go and SSA by visiting <https://golang.org/src/cmd/compile/internal/ssa/>.
- SSA bibliography: <http://www.dcs.gla.ac.uk/~jsinger/ssa.html>.
- You can learn more about the `go fix` tool by visiting its web page at [http://golang.org/cmd/fix/](https://golang.org/cmd/fix/).

Exercises

- Write test functions for the `byWord.go` program that we developed in [Chapter 8, Telling a UNIX System What to Do](#).
- Write benchmark functions for the `readSize.go` program that we developed in [Chapter 8, Telling a UNIX System What to Do](#).
- Try to fix the problems in the Go code of `documentMe.go` and `documentMe_test.go`.
- Use the text interface of the `go tool pprof` utility to examine the `memoryProfile.out` file generated by `profileMe.go`.
- Modify `webServer.go` and `webServer_test.go` in order to work with other databases such as MySQL and SQLite3.
- Modify the `Dockerfile` file from the last section in order to include `webServer_test.go`.
- Modify `Dockerfile` in order to be able to execute `go test -v` for the web server. The changes have to do with creating the right PostgreSQL users and databases.
- Use the web interface of the `go tool pprof` utility to examine the `memoryProfile.out` file generated by `profileMe.go`.

Summary

In this chapter, we talked about code testing, code optimization, and code profiling. Near the end of the chapter, you learned how to find unreachable code and how to cross-compile Go code. The `go test` command is used to test and benchmark Go code, as well as offering extra documentation with the use of example functions.

Although the discussion of the Go profiler and `go tool trace` is far from complete, you should understand that with topics such as profiling and code tracing, nothing can replace experimenting and trying new techniques on your own!

In the next chapter, [Chapter 12, *The Foundations of Network Programming in Go*](#), we will start talking about network programming in Go, which involves programming applications that work over TCP/IP computer networks, which includes the Internet. Some of the subjects in the next chapter are the `net/http` package, creating web clients and web servers in Go, the `http.Response` and `http.Request` structures, profiling HTTP servers, gRPC, and timing out network connections.

Additionally, the next chapter will discuss the **IPv4** and **IPv6** protocols, as well as the Wireshark and tshark tools, which are used to capture and analyze network traffic.

The Foundations of Network Programming in Go

The previous chapter discussed benchmarking Go code using benchmark functions, testing in Go, example functions, code coverage, cross-compilation, and profiling Go code, as well as generating documentation in Go and creating Docker images that contain the software that you want.

This chapter is all about network and web programming, which means that you will learn how to create web applications that work over computer networks and the Internet. However, you will have to wait for [Chapter 13](#), *Network Programming – Building Your Own Servers and Clients*, to learn how to develop TCP and UDP applications.

Please note that in order to follow this chapter and the next one successfully, you will need to know some basic information about HTTP, networking, and how computer networks work.

In this chapter, you will learn about the following topics:

- What TCP/IP is and why it is important
- The IPv4 and IPv6 protocols
- The **netcat** command-line utility
- Performing **DNS** lookups in Go
- The `net/http` package
- The `http.Response`, `http.Request`, and `http.Transport` structures
- Creating web servers in Go
- Programming web clients in Go
- Creating websites in Go
- **gRPC** and Go
- The `http.NewServeMux` type
- **Wireshark** and **tshark**
- Timing out HTTP connections that take too long to finish either on the server or client end

About `net/http`, `net`, and `http.RoundTripper`

The star of this chapter will be the `net/http` package, which offers functions that allow you to develop powerful web servers and web clients. The `http.Set()` and `http.Get()` methods can be used to make HTTP and HTTPS requests, whereas the `http.ListenAndServe()` function can be used to create web servers by specifying the IP address and the TCP port to which the server will listen, as well as the functions that will handle incoming requests.

Apart from `net/http`, we will use the `net` package in some of the programs presented in this chapter. The functionality of the `net` package, however, will be exposed in greater detail in [Chapter 13, *Network Programming – Building Your Own Servers and Clients*](#).

Finally, it will come in handy to know that an `http.RoundTripper` is an **interface** that is used to make sure that a Go element is capable of executing HTTP transactions. Put simply, this means that a Go element can get an `http.Response` for a given `http.Request`. You will learn about `http.Response` and `http.Request` shortly.

The http.Response type

The definition of the `http.Response` structure, which can be found in the [http
s://golang.org/src/net/http/response.go](https://golang.org/src/net/http/response.go) file, is as follows:

```
type Response struct {
    Status      string // e.g. "200 OK"
    StatusCode int    // e.g. 200
    Proto       string // e.g. "HTTP/1.0"
    ProtoMajor  int    // e.g. 1
    ProtoMinor int    // e.g. 0
    Header      Header
    Body        io.ReadCloser
    ContentLength int64
    TransferEncoding []string
    Close       bool
    Uncompressed bool
    Trailer     Header
    Request    *Request
    TLS         *tls.ConnectionState
}
```

The goal of this pretty complex `http.Response` type is to represent the response of an HTTP request. The source file contains more information about the purpose of each field of the structure, which is the case with most `struct` types found in the standard Go library.

The http.Request type

The purpose of the `http.Request` type is to represent an HTTP request as received by a server, or as about to be sent to a server by an HTTP client.

The `http.Request` structure type as defined in <https://golang.org/src/net/http/request.go> is as follows:

```
type Request struct {
    Method string
    URL *url.URL
    Proto     string // "HTTP/1.0"
    ProtoMajor int    // 1
    ProtoMinor int    // 0
    Header Header
    Body io.ReadCloser
    GetBody func() (io.ReadCloser, error)
    ContentLength int64
    TransferEncoding []string
    Close bool
    Host string
    Form url.Values
    PostForm url.Values
    MultipartForm *multipart.Form
    Trailer Header
    RemoteAddr string
    RequestURI string
    TLS *tls.ConnectionState
    Cancel <-chan struct{}
    Response *Response
    ctx context.Context
}
```

The http.Transport type

The definition of the `http.Transport` structure, which can be found in <https://golang.org/src/net/http/transport.go>, is as follows:

```
type Transport struct {
    idleMu      sync.Mutex
    wantIdle   bool
    idleConn   map[connectMethodKey][]*persistConn
    idleConnCh map[connectMethodKey]chan *persistConn
    idleLRU     connLRU
    reqMu       sync.Mutex
    reqCanceler map[*Request]func(error)
    altMu       sync.Mutex
    altProto    atomic.Value
    Proxy func(*Request) (*url.URL, error)
    DialContext func(ctx context.Context, network, addr string) (net.Conn, error)
    Dial func(network, addr string) (net.Conn, error)
    DialTLS func(network, addr string) (net.Conn, error)
    TLSClientConfig *tls.Config
    TLSHandshakeTimeout time.Duration
    DisableKeepAlives bool
    DisableCompression bool
    MaxIdleConns int
    MaxIdleConnsPerHost int
    IdleConnTimeout time.Duration
    ResponseHeaderTimeout time.Duration
    ExpectContinueTimeout time.Duration
    TLSNextProto map[string]func(authority string, c *tls.Conn) RoundTripper
    ProxyConnectHeader Header
    MaxResponseHeaderBytes int64
    nextProtoOnce sync.Once
    h2transport   *http2Transport
}
```

As you can see, the `http.Transport` structure is pretty complex and contains a very large number of fields. The good news is that you will not need to use the `http.Transport` structure in all of your programs and that you are not required to deal with all of its fields each time that you use it.

The `http.Transport` structure implements the `http.RoundTripper` interface and supports HTTP, HTTPS, and HTTP proxies. Note that `http.Transport` is pretty low-level, whereas the `http.Client` structure, which is also used in this chapter, implements a high-level HTTP client.

About TCP/IP

TCP/IP is a family of protocols that help the internet to operate. Its name comes from its two most well-known protocols: **TCP** and **IP**.

TCP stands for **Transmission Control Protocol**. TCP software transmits data between machines using segments, which are also called **TCP packets**. The main characteristic of TCP is that it is a reliable protocol, which means that it makes sure that a packet was delivered without needing any extra code from the programmer. If there is no proof of packet delivery, TCP resends that particular packet. Among other things, a TCP packet can be used for establishing connections, transferring data, sending acknowledgements, and closing connections.

When a TCP connection is established between two machines, a full duplex virtual circuit, similar to a telephone call, is created between those two machines. The two machines constantly communicate to make sure that data is sent and received correctly. If the connection fails for some reason, the two machines try to find the problem and report to the relevant application.

IP stands for **Internet Protocol**. The main characteristic of IP is that it is not a reliable protocol by nature. IP encapsulates the data that travels over a TCP/IP network because it is responsible for delivering packets from the source host to the destination host according to the IP addresses. IP has to find an addressing method to send a packet to its destination effectively. Although there are dedicated devices, called routers, that perform IP routing, every TCP/IP device has to perform some basic routing.

The **UDP (User Datagram Protocol)** protocol is based on IP, which means that it is also unreliable. Generally speaking, the UDP protocol is simpler than the TCP protocol, mainly because UDP is not reliable by design. As a result, UDP messages can be lost, duplicated, or arrive out of order.

Furthermore, packets can arrive faster than the recipient can process them. So, UDP is used when speed is more important than reliability.

About IPv4 and IPv6

The first version of the **IP protocol** is now called **IPv4** in order to differentiate it from the latest version of the IP protocol, which is called **IPv6**.

The main problem with IPv4 is that it is about to run out of IP addresses, which is the main reason for creating the IPv6 protocol. This happened because an IPv4 address is represented using 32 bits only, which allows a total number of 2^{32} (4,294,967,296) different IP addresses. On the other hand, IPv6 uses 128 bits to define each one of its addresses.

The format of an IPv4 address is 10.20.32.245 (four parts separated by dots), while the format of an IPv6 address is 3fce:1706:4523:3:150:f8ff:fe21:56cf (eight parts separated by colons).

The nc(1) command-line utility

The `nc(1)` utility, which is also called `netcat(1)`, comes in very handy when you want to test TCP/IP servers and clients. This section will present some of its more common uses.

You can use `nc(1)` as a client for a TCP service that runs on a machine with the `10.10.1.123` IP address and listens to port number `1234`, as follows:

```
| $ nc 10.10.1.123 1234
```

By default, `nc(1)` uses the TCP protocol. However, if you execute `nc(1)` with the `-u` flag, then `nc(1)` will use the UDP protocol.

The `-l` option tells `netcat(1)` to act as a server, which means that `netcat(1)` will start listening for connections at the given port number.

Finally, the `-v` and `-vv` options tell `netcat(1)` to generate verbose output, which can come in handy when you want to troubleshoot network connections.

Although `netcat(1)` can help you to test HTTP applications, it will be niftier in [Chapter 13](#), *Network Programming – Building Your Own Servers and Clients*, when we will develop our own TCP and UDP clients and servers. As a result, the `netcat(1)` utility will be used only once in this chapter.

Reading the configuration of network interfaces

There are four core elements in a network configuration: the IP address of the interface, the network mask of the interface, the DNS servers of the machine, and the default gateway or default router of the machine. However, there is a problem here: you cannot find every piece of information using native, portable Go code. This means that there is no portable way to discover the DNS configuration and the default gateway information of a UNIX machine.

As a result, in this section, you will learn how to read the configuration of the network interfaces of a UNIX machine with Go. For that purpose, I will present two portable utilities that allow you to find out information about your network interfaces.

The source code of the first utility, which is called `netConfig.go`, is presented in three parts. The first part of `netConfig.go` is shown in the following Go code:

```
package main

import (
    "fmt"
    "net"
)

func main() {
    interfaces, err := net.Interfaces()
    if err != nil {
        fmt.Println(err)
        return
    }
```

The `net.Interfaces()` function returns all of the interfaces of the current machine as a slice with elements of the `net.Interface` type. This slice will be used shortly to acquire the desired information.

The second code portion of `netConfig.go` contains the following Go code:

```
|     for _, i := range interfaces {
|         fmt.Printf("Interface: %v\n", i.Name)
|         byName, err := net.InterfaceByName(i.Name)
|         if err != nil {
|             fmt.Println(err)
|         }
|     }
```

In the preceding code, you visit each element of the slice with the `net.Interface` elements to retrieve the desired information.

The remaining Go code of `netConfig.go` is as follows:

```
|     addresses, err := byName.Addrs()
|     for k, v := range addresses {
|         fmt.Printf("Interface Address #%v: %v\n", k, v.String())
|     }
|     fmt.Println()
| }
```

Executing `netConfig.go` on a macOS Mojave machine with Go version 1.12.4 generates the following output:

```
$ go run netConfig.go
Interface: lo0
Interface Address #0 : 127.0.0.1/8
Interface Address #1 : ::1/128
Interface Address #2 : fe80::1/64

Interface: gif0

Interface: stf0

Interface: XHC20

Interface: en0
Interface Address #0 : fe80::1435:19cd:ece8:f532/64
Interface Address #1 : 10.67.93.23/24

Interface: p2p0

Interface: awd10
Interface Address #0 : fe80::888:68ff:fe01:99c/64

Interface: en1

Interface: en2

Interface: bridge0

Interface: utun0
Interface Address #0 : fe80::7fd3:e1ba:a4b1:fe22/64
```

As you can see, the `netConfig.go` utility returns a rather large output, because today's computers tend to have lots of network interfaces, and the program supports both the IPv4 and IPv6 protocols.

Executing `netConfig.go` on a Debian Linux machine with Go version 1.7.4 will generate the following output:

```
$ go run netConfig.go
Interface: lo
Interface Address #0: 127.0.0.1/8
Interface Address #1: ::1/128

Interface: dummy0

Interface: eth0
Interface Address #0: 10.74.193.253/24
Interface Address #1: 2a01:7e00::f03c:91ff:fe69:1381/64
Interface Address #2: fe80::f03c:91ff:fe69:1381/64

Interface: teq10

Interface: tun10

Interface: gre0

Interface: gretap0

Interface: erspan0

Interface: ip_vti0

Interface: ip6_vti0

Interface: sit0

Interface: ip6tn10

Interface: ip6gre0
```

Please note that the main reason why a network interface may not have a network address is that it is down, which essentially means that it is not currently configured.



*Not all of the listed interfaces have a real hardware network device attached to them. A more representative example is the `lo` interface, which is a loopback device. The **loopback device** is a special, virtual network interface that is used by your computer in order to communicate with itself over a network.*

The Go code for the next utility, `netCapabilities.go`, is also presented in three parts. The purpose of the `netCapabilities.go` utility is to reveal the capabilities

of each network interface found on your UNIX system.

The `netCapabilities.go` utility uses the fields of the `net.Interface` structure, which is defined as follows:

```
type Interface struct {
    Index      int
    MTU       int
    Name      string
    HardwareAddr HardwareAddr
    Flags      Flags
}
```

The first part of the Go code for `netCapabilities.go` is as follows:

```
package main

import (
    "fmt"
    "net"
)
```

The second code portion of `netCapabilities.go` contains the following Go code:

```
func main() {
    interfaces, err := net.Interfaces()

    if err != nil {
        fmt.Println(err)
        return
    }
```

The last part of `netCapabilities.go` comes in the following Go code:

```
    for _, i := range interfaces {
        fmt.Printf("Name: %v\n", i.Name)
        fmt.Println("Interface Flags:", i.Flags.String())
        fmt.Println("Interface MTU:", i.MTU)
        fmt.Println("Interface Hardware Address:", i.HardwareAddr)

        fmt.Println()
    }
}
```

Running `netCapabilities.go` on a macOS Mojave machine will generate the following output:

```
$ go run netCapabilities.go
Name : lo0
Interface Flags: up|loopback|multicast
```

```
Interface MTU: 16384
Interface Hardware Address:

Name : gif0
Interface Flags: pointtopoint|multicast
Interface MTU: 1280
Interface Hardware Address:

Name : stf0
Interface Flags: 0
Interface MTU: 1280
Interface Hardware Address:

Name : XHC20
Interface Flags: 0
Interface MTU: 0
Interface Hardware Address:

Name : en0
Interface Flags: up|broadcast|multicast
Interface MTU: 1500
Interface Hardware Address: b8:e8:56:34:a1:c8

Name : p2p0
Interface Flags: up|broadcast|multicast
Interface MTU: 2304
Interface Hardware Address: 0a:e8:56:34:a1:c8

Name : awdl0
Interface Flags: up|broadcast|multicast
Interface MTU: 1484
Interface Hardware Address: 0a:88:68:01:09:9c

Name : en1
Interface Flags: up|broadcast|multicast
Interface MTU: 1500
Interface Hardware Address: 72:00:00:9d:b2:b0

Name : en2
Interface Flags: up|broadcast|multicast
Interface MTU: 1500
Interface Hardware Address: 72:00:00:9d:b2:b1

Name : bridge0
Interface Flags: up|broadcast|multicast
Interface MTU: 1500
Interface Hardware Address: 72:00:00:9d:b2:b0

Name : utun0
Interface Flags: up|pointtopoint|multicast
Interface MTU: 2000
Interface Hardware Address:
```

Executing `netCapabilities.go` on a Debian Linux machine will generate similar output.

Finally, if you are really interested in finding out the default gateway of the machine, you can execute the `netstat -nr` command either externally or using `exec.Command()`, taking its output using a pipe or `exec.CombinedOutput()` and processing it as text using Go. This solution, however, is neither elegant nor perfect!

Performing DNS lookups

DNS stands for **Domain Name System**, which relates to the way an IP address is translated into a name, such as `packt.com`, and vice versa. The logic behind the `DNS.go` utility, which will be developed in this section, is pretty simple: if the given command-line argument is a valid IP address, the program will process it as an IP address; otherwise, it will assume that it is dealing with a hostname that needs to be translated into one or more IP addresses.

The code for the `DNS.go` utility will be presented in three parts. The first part of the program contains the following Go code:

```
package main

import (
    "fmt"
    "net"
    "os"
)

func lookIP(address string) ([]string, error) {
    hosts, err := net.LookupAddr(address)
    if err != nil {
        return nil, err
    }
    return hosts, nil
}

func lookHostname(hostname string) ([]string, error) {
    IPs, err := net.LookupHost(hostname)
    if err != nil {
        return nil, err
    }
    return IPs, nil
}
```

The `lookIP()` function gets an IP address as input and returns a list of names that match that IP address with the help of the `net.LookupAddr()` function.

On the other hand, the `lookHostname()` function gets a hostname as input and returns a list with the associated IP addresses using the `net.LookupHost()` function.

The second part of `DNS.go` is the following Go code:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide an argument!")
        return
    }

    input := arguments[1]
    IPaddress := net.ParseIP(input)
```

The `net.ParseIP()` function parses a string as an IPv4 or an IPv6 address. If the IP address is not valid, `net.ParseIP()` returns `nil`.

The remaining Go code for the `DNS.go` utility is as follows:

```
if IPaddress == nil {
    IPs, err := lookHostname(input)
    if err == nil {
        for _, singleIP := range IPs {
            fmt.Println(singleIP)
        }
    }
} else {
    hosts, err := lookIP(input)
    if err == nil {
        for _, hostname := range hosts {
            fmt.Println(hostname)
        }
    }
}
```

Executing `DNS.go` with various kinds of input will generate the following output:

```
$ go run DNS.go 127.0.0.1
localhost
$ go run DNS.go 192.168.1.1
cisco
$ go run DNS.go packtpub.com
83.166.169.231
$ go run DNS.go google.com
2a00:1450:4001:816::200e
216.58.210.14
$ go run DNS.go www.google.com
2a00:1450:4001:816::2004
216.58.214.36
$ go run DNS.go cnn.com
2a04:4e42::323
2a04:4e42:600::323
2a04:4e42:400::323
2a04:4e42:200::323
```

```
| 151.101.193.67  
| 151.101.1.67  
| 151.101.129.67  
| 151.101.65.67
```

Please note that the output of the `go run DNS.go 192.168.1.1` command is taken from my `/etc/hosts` file, because the `cisco` hostname is an alias for the `192.168.1.1` IP address in my `/etc/hosts` file.

The output of the last command shows that, sometimes, a single hostname (`cnn.com`) might have multiple public IP addresses. Please pay special attention to the word *public* here, because although `www.google.com` has multiple IP addresses, it uses just a single public IP address (`216.58.214.36`).

Getting the NS records of a domain

A very popular DNS request has to do with finding out the **name servers** of a domain, which are stored in the **NS records** of that domain. This functionality will be illustrated in the code for `NSrecords.go`.

The code for `NSrecords.go` will be presented in two parts.

The first part of `NSrecords.go` is as follows:

```
package main

import (
    "fmt"
    "net"
    "os"
)

func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Need a domain name!")
        return
    }
}
```

In this part, you check whether you have at least one command-line argument in order to have something with which you can work.

The remaining Go code for `NSrecords.go` is as follows:

```
domain := arguments[1]
NSs, err := net.LookupNS(domain)
if err != nil {
    fmt.Println(err)
    return
}

for _, NS := range NSs {
    fmt.Println(NS.Host)
}
}
```

All the work is done by the `net.LookupNS()` function, which returns the NS records of a domain as a slice variable of the `net.NS` type. This is the reason

for printing the `Host` field of each `net.NS` element of the slice. Executing `NSrecords.go` will generate the following type of output:

```
$ go run NSrecords.go mtsoukalos.eu
ns5.linode.com.
ns4.linode.com.
ns1.linode.com.
ns2.linode.com.
ns3.linode.com.
$ go run NSrecords.go www.mtsoukalos.eu
lookup www.mtsoukalos.eu on 8.8.8.8:53: no such host
```

You can verify the correctness of the preceding output with the help of the `host(1)` utility:

```
$ host -t ns www.mtsoukalos.eu
www.mtsoukalos.eu has no NS record
$ host -t ns mtsoukalos.eu
mt soukalos.eu name server ns3.linode.com.
mt soukalos.eu name server ns1.linode.com.
mt soukalos.eu name server ns4.linode.com.
mt soukalos.eu name server ns2.linode.com.
mt soukalos.eu name server ns5.linode.com.
```

Getting the MX records of a domain

Another very popular DNS request has to do with getting the **MX records** of a domain. The MX records specify the mail servers of a domain. The code for the `MXrecords.go` utility will perform this task with Go. The first part of the `MXrecords.go` utility is shown in the following Go code:

```
package main

import (
    "fmt"
    "net"
    "os"
)

func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Need a domain name!")
        return
    }
}
```

The second part of `MXrecords.go` contains the following Go code:

```
    domain := arguments[1]
    MXs, err := net.LookupMX(domain)
    if err != nil {
        fmt.Println(err)
        return
    }

    for _, MX := range MXs {
        fmt.Println(MX.Host)
    }
}
```

The code for `MXrecords.go` works in a similar way to the code for `NXrecords.go`, presented in the previous section. The only difference is that `MXrecords.go` uses the `net.LookupMX()` function instead of the `net.LookupNS()` function.

Executing `MXrecords.go` will generate the following type of output:

```
$ go run MXrecords.go golang.com
aspmx.l.google.com.
```

```
| alt3.aspmx.1.google.com.  
| alt1.aspmx.1.google.com.  
| alt2.aspmx.1.google.com.  
$ go run MXrecords.go www.mtsoukalos.eu  
lookup www.mtsoukalos.eu on 8.8.8.8:53: no such host
```

Once again, you can verify the validity of the preceding output with the help of the `host(1)` utility:

```
| $ host -t mx golang.com  
golang.com mail is handled by 2 alt3.aspmx.1.google.com.  
golang.com mail is handled by 1 aspmx.1.google.com.  
golang.com mail is handled by 2 alt1.aspmx.1.google.com.  
golang.com mail is handled by 2 alt2.aspmx.1.google.com.  
$ host -t mx www.mtsoukalos.eu  
www.mtsoukalos.eu has no MX record
```

Creating a web server in Go

Go allows you to create web servers on your own using some of the functions of its standard library.



Although a web server programmed in Go can do many things efficiently and securely, if what you really need is a powerful web server that will support modules, multiple websites, and virtual hosts, then you would be better off using a web server such as Apache, Nginx or Candy, which is written in Go.

The name of the Go program for this example will be `www.go`, and it will be presented in five parts. The first part of `www.go` contains the expected `import` statements:

```
package main

import (
    "fmt"
    "net/http"
    "os"
    "time"
)
```

The `time` package is not necessary for a web server to operate. However, it is needed in this case because the server is going to send the time and date to its clients.

The second code segment of `www.go` is shown in the following Go code:

```
func myHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served: %s\n", r.Host)
}
```

This is the implementation of the first **handler function** of the program. A handler function serves one or more URLs depending on the configuration that is stated in the Go code – you can have as many handler functions as you want.

The third part of `www.go` contains the following Go code:

```

func timeHandler(w http.ResponseWriter, r *http.Request) {
    t := time.Now().Format(time.RFC1123)
    Body := "The current time is:"
    fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>", Body)
    fmt.Fprintf(w, "<h2 align=\"center\">%s</h2>\n", t)
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served time for: %s\n", r.Host)
}

```

In the preceding Go code, you can see the implementation of the program's second handler function. This function generates dynamic content.

The fourth section of code for our web server deals with the command-line arguments and the definition of the supported URLs:

```

func main() {
    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Using default port number: ", PORT)
    } else {
        PORT = ":" + arguments[1]
    }

    http.HandleFunc("/time", timeHandler)
    http.HandleFunc("/", myHandler)
}

```

The `http.HandleFunc()` function associates a URL with a handler function. What is really important here is that all URLs apart from `/time` are served by the `myHandler()` function because its first argument, which is `/`, matches every URL not matched by another handler.

The last part of the `www.go` program is as follows:

```

    err := http.ListenAndServe(PORT, nil)
    if err != nil {
        fmt.Println(err)
        return
    }
}

```

You should start the web server with the help of the `http.ListenAndServe()` function using the desired port number.

Executing `www.go` and connecting to its web server will generate the following type of output:

```

$ go run www.go
Using default port number:  :8001

```

```
Served: localhost:8001
Served: localhost:8001
Served time for: localhost:8001
Served: localhost:8001
Served time for: localhost:8001
Served: localhost:8001
Served time for: localhost:8001
Served: localhost:8001
Served: localhost:8001
Served: localhost:8001
```

Although the output of the program provides some handy material, I think that you would prefer to see the real output of the program using your favorite web browser. The next screenshot shows the output of the `myHandler()` function of our web server as displayed in **Google Chrome**:

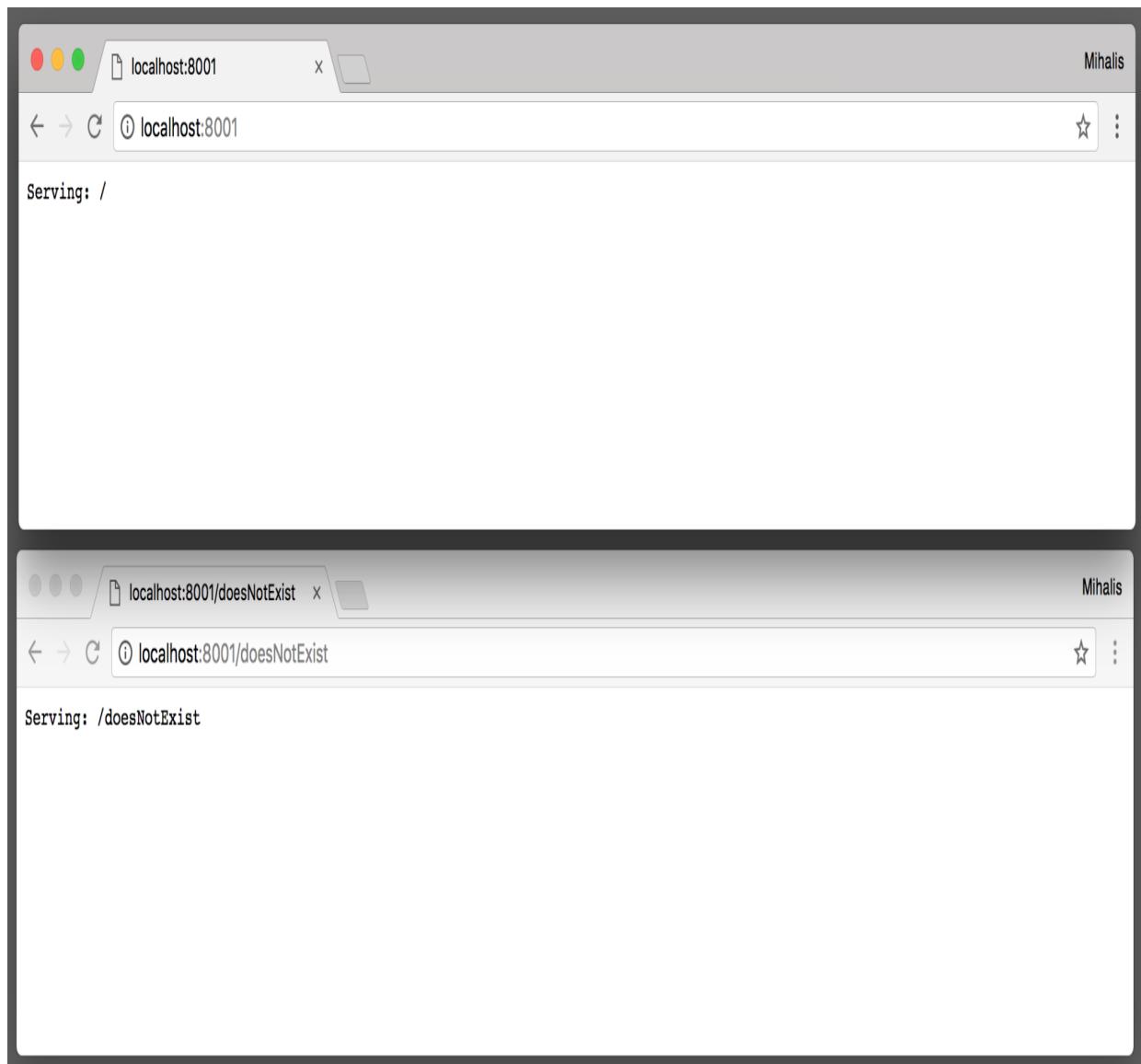


Figure 12.1: The home page of the `www.go` web server

The following screenshot shows that `www.go` can generate dynamic pages as well. In this case, it is a web page registered in `/time`, which displays the current date and time:

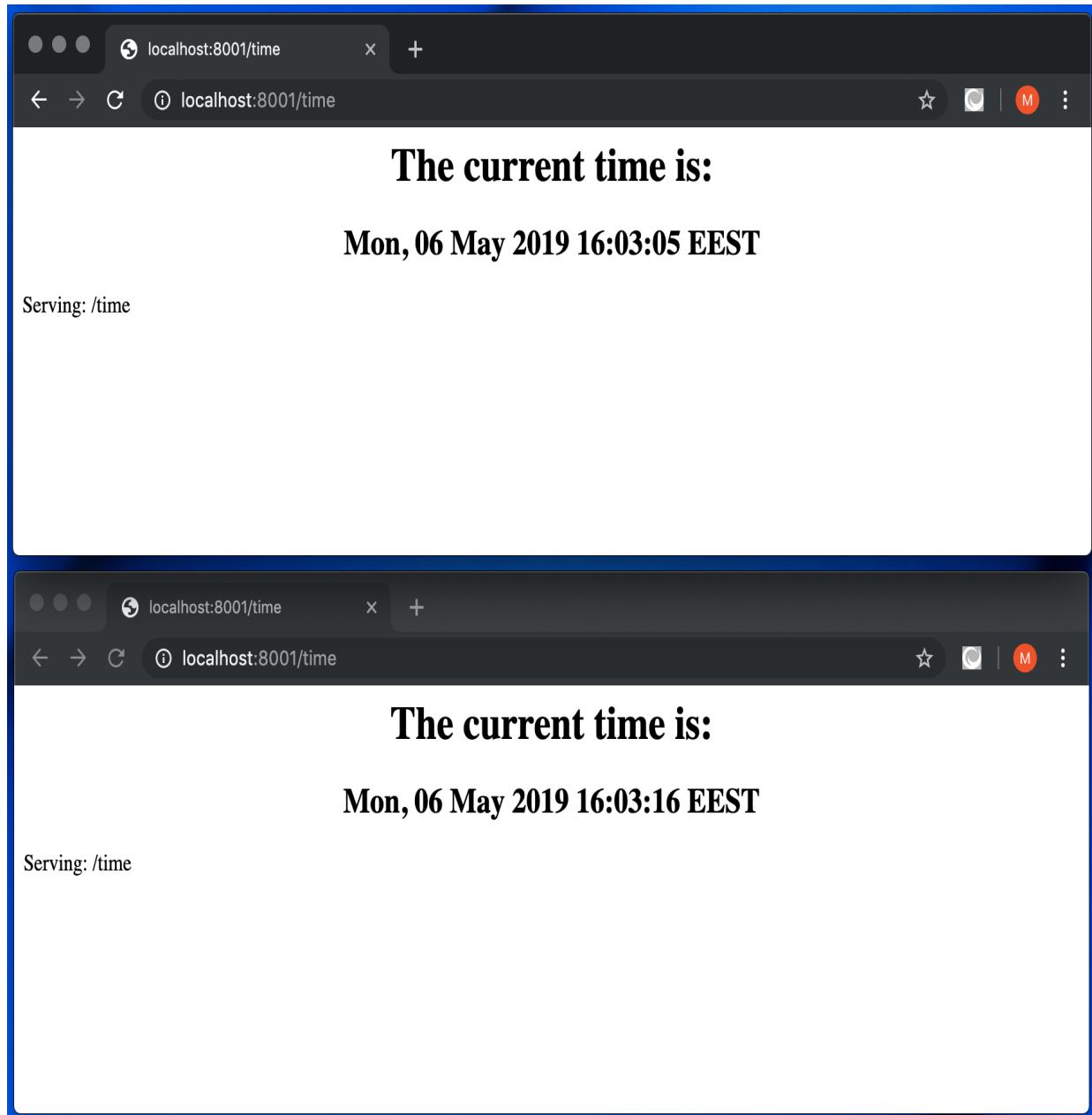


Figure 12.2: Getting the current date and time from the www.go web server

Using the atomic package

In this section, you will learn how to use the `atomic` package in an HTTP server environment. The name of the program is `atomWWW.go` and it will be presented in three parts.

The first part of `atomWWW.go` is as follows:

```
package main

import (
    "fmt"
    "net/http"
    "runtime"
    "sync/atomic"
)
var count int32
```

The variable that will be used by the `atomic` package is a global one, in order to be accessible from anywhere in the code.

The second part of `atomWWW.go` contains the following Go code:

```
func handleAll(w http.ResponseWriter, r *http.Request) {
    atomic.AddInt32(&count, 1)
}

func getCounter(w http.ResponseWriter, r *http.Request) {
    temp := atomic.LoadInt32(&count)
    fmt.Println("Count:", temp)
    fmt.Fprintf(w, "<h1 align=\"center\">%d</h1>", count)
}
```

The atomic counter used in this program is associated with the `count` global variable and helps us to count the total number of clients that the web server has served so far.

The last part of `atomWWW.go` is as follows:

```
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU() - 1)
    http.HandleFunc("/getCounter", getCounter)
    http.HandleFunc("/", handleAll)
```

```
|     http.ListenAndServe(":8080", nil)
| }
```

Testing `atomWWW.go` using the `ab(1)` utility will verify how the `atomic` package helps, because it will reveal that the `count` variable can be accessed by all clients without any problems. First, you will need to execute `atomWWW.go`:

```
$ go run atomWWW.go
Count: 1500
```

Then, you will need to execute `ab(1)` as follows:

```
$ ab -n 1500 -c 100 http://localhost:8080/
This is ApacheBench, Version 2.3 <$Revision: 1826891 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)
Completed 150 requests
Completed 300 requests
Completed 450 requests
Completed 600 requests
Completed 750 requests
Completed 900 requests
Completed 1050 requests
Completed 1200 requests
Completed 1350 requests
Completed 1500 requests
Finished 1500 requests

Server Software:
Server Hostname:      localhost
Server Port:          8080

Document Path:         /
Document Length:      0 bytes

Concurrency Level:    100
Time taken for tests: 0.098 seconds
Complete requests:   1500
Failed requests:      0
Total transferred:   112500 bytes
HTML transferred:    0 bytes
Requests per second: 15238.64 #[/sec] (mean)
Time per request:    6.562 [ms] (mean)
Time per request:    0.066 [ms] (mean, across all concurrent requests)
Transfer rate:        1116.11 [Kbytes/sec] received

Connection Times (ms)
              min  mean [+/-sd] median   max
Connect:       0    3   0.5      3      5
Processing:    2    3   0.6      3      6
Waiting:       0    3   0.6      3      5
Total:        4    6   0.6      6      9
```

```
| Percentage of the requests served within a certain time (ms)
| 50%      6
| 66%      6
| 75%      7
| 80%      7
| 90%      7
| 95%      7
| 98%      8
| 99%      8
| 100%     9 (longest request)
```

The previous `ab(1)` command sends 1,500 requests, with the number of concurrent requests being 100.

After the `ab(1)` part, you should visit the `/getCounter` address in order to get the current value of the `count` variable:

```
| $ wget -qO- http://localhost:8080/getCounter
| <h1 align="center">1500</h1>%
```

Profiling an HTTP server

As you learned in [Chapter 11, *Code Testing, Optimization, and Profiling*](#), there is a standard Go package named `net/http/pprof`, which should be used when you want to profile a Go application with its own HTTP server. To that end, importing `net/http/pprof` will install various handlers under the `/debug/pprof/` URL. You will see more on this in a short while. For now, it is enough to remember that the `net/http/pprof` package should be used to profile web applications with an HTTP server, whereas the `runtime/pprof` standard Go package should be used to profile all other kinds of applications.

Note that if your profiler works using the `http://localhost:8080` address, you will automatically get support for the following web links:

- `http://localhost:8080/debug/pprof/goroutine`
- `http://localhost:8080/debug/pprof/heap`
- `http://localhost:8080/debug/pprof/threadcreate`
- `http://localhost:8080/debug/pprof/block`
- `http://localhost:8080/debug/pprof/mutex`
- `http://localhost:8080/debug/pprof/profile`
- `http://localhost:8080/debug/pprof/trace?seconds=5`

The next program to be presented uses `www.go` as its starting point and adds the necessary Go code to allow you to profile it.

The name of the new program is `wwwProfile.go`, and it will be presented in four parts.

Note that `wwwProfile.go` uses the `http.NewServeMux` variable to register the program's supported paths. The main reason for doing so is that the use of `http.NewServeMux` requires defining the HTTP endpoints manually. Also note that you are allowed to define a subset of the supported HTTP endpoints. If you decide not to use `http.NewServeMux`, then the HTTP endpoints will be

registered automatically, which will also mean that you will have to import the `net/http/pprof` package using the `_` character in front of it.

The first part of `wwwProfile.go` contains the following Go code:

```
package main

import (
    "fmt"
    "net/http"
    "net/http/pprof"
    "os"
    "time"
)

func myHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served: %s\n", r.Host)
}

func timeHandler(w http.ResponseWriter, r *http.Request) {
    t := time.Now().Format(time.RFC1123)
    Body := "The current time is:"
    fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>", Body)
    fmt.Fprintf(w, "<h2 align=\"center\">%s</h2>\n", t)
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served time for: %s\n", r.Host)
}
```

The implementations of these two handler functions are exactly the same as before.

The second code segment of `wwwProfile.go` is as follows:

```
func main() {
    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Using default port number: ", PORT)
    } else {
        PORT = ":" + arguments[1]
        fmt.Println("Using port number: ", PORT)
    }

    r := http.NewServeMux()
    r.HandleFunc("/time", timeHandler)
    r.HandleFunc("/", myHandler)
}
```

In the preceding Go code, you define the URLs that will be supported by your web server using `http.NewServeMux()` and `HandleFunc()`.

The third code portion for `wwwProfile.go` is shown in the following Go code:

```
|     r.HandleFunc("/debug/pprof/", pprof.Index)
|     r.HandleFunc("/debug/pprof/cmdline", pprof.Cmdline)
|     r.HandleFunc("/debug/pprof/profile", pprof.Profile)
|     r.HandleFunc("/debug/pprof/symbol", pprof.Symbol)
|     r.HandleFunc("/debug/pprof/trace", pprof.Trace)
```

The preceding Go code defines the HTTP endpoints related to profiling. Without them, you will not be able to profile your web application.

The remaining Go code is as follows:

```
|     err := http.ListenAndServe(PORT, r)
|     if err != nil {
|         fmt.Println(err)
|         return
|     }
| }
```

This code begins the Go web server, and this allows it to serve connections from HTTP clients. You will notice that the second parameter to `http.ListenAndServe()` is no longer `nil`.

As you can see, `wwwProfile.go` does not define the `/debug/pprof/goroutine` HTTP endpoint, which makes perfect sense as `wwwProfile.go` does not use any goroutines.

Executing `wwwProfile.go` will generate the following type of output:

```
$ go run wwwProfile.go 1234
Using port number:  :1234
Served time for: localhost:1234
```

Using the Go profiler to get data is a pretty simple task that requires the execution of the following command, which will take you to the shell of the Go profiler automatically:

```
$ go tool pprof http://localhost:1234/debug/pprof/profile
Fetching profile over HTTP from http://localhost:1234/debug/pprof/profile
Saved profile in /Users/mtsouk/pprof/pprof.samples.cpu.003.pb.gz
Type: cpu
Time: Mar 27, 2018 at 10:04pm (EEST)
Duration: 30s, Total samples = 21.04s (70.13%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top
Showing nodes accounting for 19.94s, 94.77% of 21.04s total
Dropped 159 nodes (cum <= 0.11s)
Showing top 10 nodes out of 75
```

```

      flat  flat%  sum%          cum   cum%
13.73s 65.26% 65.26%    13.73s 65.26%  syscall.Syscall
 1.58s 7.51% 72.77%    1.58s 7.51%  runtime.kevent
 1.36s 6.46% 79.23%    1.36s 6.46%  runtime.mach_semaphore_signal
 1.02s 4.85% 84.08%    1.02s 4.85%  runtime.usleep
 0.80s 3.80% 87.88%    0.80s 3.80%  runtime.mach_semaphore_wait
 0.53s 2.52% 90.40%    2.11s 10.03% runtime.netpoll
 0.44s 2.09% 92.49%    0.44s 2.09%  internal/poll.convertErr
 0.26s 1.24% 93.73%    0.26s 1.24%  net.(*TCPConn).Read
 0.18s 0.86% 94.58%    0.18s 0.86%  runtime.freedefer
 0.04s 0.19% 94.77%    1.05s 4.99%  runtime.rungsteal
(pprof)

```

You can now use the profiling data and analyze it using `go tool pprof`, as you learned to do in [Chapter 11, *Code Testing, Optimization, and Profiling*](#).



You can visit <http://HOSTNAME:PORTNUMBER/debug/pprof/> and see the profiling results there. When the `HOSTNAME` value is `localhost` and the `PORTNUMBER` value is `1234`, you should visit <http://localhost:1234/debug/pprof/>.

Should you wish to test the performance of your web server application, you can use the `ab(1)` utility, which is more widely known as the **Apache HTTP server benchmarking tool**, in order to create some traffic and benchmark `wwwProfile.go`. This will also allow `go tool pprof` to collect more accurate data – you can execute `ab(1)` as follows:

```
$ ab -k -c 10 -n 100000 "http://127.0.0.1:1234/time"
This is ApacheBench, Version 2.3 <$Revision: 1807734 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 127.0.0.1 (be patient)
Completed 10000 requests
Completed 20000 requests
Completed 30000 requests
Completed 40000 requests
Completed 50000 requests
Completed 60000 requests
Completed 70000 requests
Completed 80000 requests
Completed 90000 requests
Completed 100000 requests
Finished 100000 requests

Server Software:
Server Hostname:      127.0.0.1
Server Port:           1234

Document Path:         /time
Document Length:       114 bytes

Concurrency Level:     10
Time taken for tests: 2.114 seconds
```

```

Complete requests: 100000
Failed requests: 0
Keep-Alive requests: 100000
Total transferred: 25500000 bytes
HTML transferred: 11400000 bytes
Requests per second: 47295.75 [#/sec] (mean)
Time per request: 0.211 [ms] (mean)
Time per request: 0.021 [ms] (mean, across all concurrent requests)
Transfer rate: 11777.75 [Kbytes/sec] received

Connection Times (ms)
      min   mean[+/-sd] median    max
Connect:      0     0     0.0     0     0
Processing:   0     0     0.7     0     13
Waiting:     0     0     0.7     0     13
Total:       0     0     0.7     0     13

Percentage of the requests served within a certain time (ms)
 50%      0
 66%      0
 75%      0
 80%      0
 90%      0
 95%      0
 98%      0
 99%      0
100%    13 (longest request)

```

 *Can you use the `net/http/pprof` package to profile command-line applications? Yes! However, the `net/http/pprof` package is particularly useful when you want to profile a running web application and capture live data, which is the main reason that it is presented in this chapter.*

Creating a website in Go

Do you remember the `keyValue.go` application from [Chapter 4, The Uses of Composite Types](#), and `kvSaveLoad.go` from [Chapter 8, Telling a UNIX System What to Do?](#) In this section, you will learn how to create a web interface for the `keyValue.go` application using the capabilities of the standard Go library. The name of the new Go source code file is `kvWeb.go`, and it is going to be presented in six parts.

The first difference between the Go code for `kvWeb.go` and `www.go`, developed earlier in this chapter, is that `kvWeb.go` uses the `http.NewServeMux` type to deal with HTTP requests because it is much more versatile for non-trivial web applications.

The first part of `kvWeb.go` is as follows:

```
package main

import (
    "encoding/gob"
    "fmt"
    "html/template"
    "net/http"
    "os"
)

type myElement struct {
    Name      string
    Surname   string
    Id        string
}

var DATA = make(map[string]myElement)
var DATAFILE = "/tmp/dataFile.gob"
```

You have already seen this code in `kvSaveLoad.go` in [Chapter 8, Telling a UNIX System What to Do.](#)

The second part of `kvWeb.go` is shown in the following Go code:

```
func save() error {
    fmt.Println("Saving", DATAFILE)
    err := os.Remove(DATAFILE)
    if err != nil {
        fmt.Println(err)
    }

    saveTo, err := os.Create(DATAFILE)
    if err != nil {
        fmt.Println("Cannot create", DATAFILE)
        return err
    }
    defer saveTo.Close()

    encoder := gob.NewEncoder(saveTo)
    err = encoder.Encode(DATA)
```

```

    if err != nil {
        fmt.Println("Cannot save to", DATAFILE)
        return err
    }
    return nil
}

func load() error {
    fmt.Println("Loading", DATAFILE)
    loadFrom, err := os.Open(DATAFILE)
    defer loadFrom.Close()
    if err != nil {
        fmt.Println("Empty key/value store!")
        return err
    }

    decoder := gob.NewDecoder(loadFrom)
    decoder.Decode(&DATA)
    return nil
}

func ADD(k string, n myElement) bool {
    if k == "" {
        return false
    }

    if LOOKUP(k) == nil {
        DATA[k] = n
        return true
    }
    return false
}

func DELETE(k string) bool {
    if LOOKUP(k) != nil {
        delete(DATA, k)
        return true
    }
    return false
}

func LOOKUP(k string) *myElement {
    _, ok := DATA[k]
    if ok {
        n := DATA[k]
        return &n
    } else {
        return nil
    }
}

func CHANGE(k string, n myElement) bool {
    DATA[k] = n
    return true
}

func PRINT() {
    for k, d := range DATA {
        fmt.Printf("key: %s value: %v\n", k, d)
    }
}

```

You should also be familiar with the preceding Go code, as it first appeared in `kvSaveLoad.go` in [Chapter 8](#), *Telling a UNIX System What to Do*.

The third code portion of `kvWeb.go` is as follows:

```

func homePage(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Serving", r.Host, "for", r.URL.Path)
    myT := template.Must(template.ParseGlob("home.gohtml"))
    myT.ExecuteTemplate(w, "home.gohtml", nil)
}

func listAll(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Listing the contents of the KV store!")

    fmt.Fprintf(w, "<a href=\"/\" style=\"margin-right: 20px;\">Home sweet home!</a>")
    fmt.Fprintf(w, "<a href=\"/list\" style=\"margin-right: 20px;\">List all elements!</a>")
    fmt.Fprintf(w, "<a href=\"/change\" style=\"margin-right: 20px;\">Change an element!</a>")
    fmt.Fprintf(w, "<a href=\"/insert\" style=\"margin-right: 20px;\">Insert new element!</a>")

    fmt.Fprintf(w, "<h1>The contents of the KV store are:</h1>")
    fmt.Fprintf(w, "<ul>")
    for k, v := range DATA {
        fmt.Fprintf(w, "<li>")
        fmt.Fprintf(w, "<strong>%s</strong> with value: %v\n", k, v)
        fmt.Fprintf(w, "</li>")
    }
    fmt.Fprintf(w, "</ul>")
}

```

The `listAll()` function does not use any Go templates to generate its dynamic output. Instead, its output is generated on the fly using Go. You may consider this an exception, as web applications usually work better with HTML templates and the `html/template` standard Go package.

The fourth part of `kvWeb.go` contains the following Go code:

```

func changeElement(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Changing an element of the KV store!")
    tmpl := template.Must(template.ParseFiles("update.gohtml"))
    if r.Method != http.MethodPost {
        tmpl.Execute(w, nil)
        return
    }

    key := r.FormValue("key")
    n := myElement{
        Name:   r.FormValue("name"),
        Surname: r.FormValue("surname"),
        Id:     r.FormValue("id"),
    }

    if !CHANGE(key, n) {
        fmt.Println("Update operation failed!")
    } else {
        err := save()
        if err != nil {
            fmt.Println(err)
            return
        }
        tmpl.Execute(w, struct{ Success bool }{true})
    }
}

```

In the preceding Go code, you can see how to read the values from the fields of an HTML form with the help of the `FormValue()` function. `template.Must()` is a helper function that makes sure that the template file provided contains no errors.

The fifth code segment of `kvWeb.go` is contained in the following Go code:

```
func insertElement(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Inserting an element to the KV store!")
    tmpl := template.Must(template.ParseFiles("insert.gohtml"))
    if r.Method != http.MethodPost {
        tmpl.Execute(w, nil)
        return
    }

    key := r.FormValue("key")
    n := myElement{
        Name:     r.FormValue("name"),
        Surname: r.FormValue("surname"),
        Id:       r.FormValue("id"),
    }

    if !ADD(key, n) {
        fmt.Println("Add operation failed!")
    } else {
        err := save()
        if err != nil {
            fmt.Println(err)
            return
        }
        tmpl.Execute(w, struct{ Success bool }{true})
    }
}
```

The remaining Go code is as follows:

```
func main() {
    err := load()
    if err != nil {
        fmt.Println(err)
    }

    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Using default port number: ", PORT)
    } else {
        PORT = ":" + arguments[1]
    }

    http.HandleFunc("/", homePage)
    http.HandleFunc("/change", changeElement)
    http.HandleFunc("/list", listAll)
    http.HandleFunc("/insert", insertElement)
    err = http.ListenAndServe(PORT, nil)
    if err != nil {
        fmt.Println(err)
    }
}
```

The `main()` function of `kvWeb.go` is much simpler than the `main()` function of `kvSaveLoad.go` from [Chapter 8](#), *Telling a UNIX System What to Do*, because these two programs have totally different designs.

It is now time to look at the `gohtml` files used for this project, starting with `home.gohtml`, which is as follows:

```

<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>A Key Value Store!</title>
</head>
<body>

<a href="/" style="margin-right: 20px;">Home sweet home!</a>
<a href="/list" style="margin-right: 20px;">List all elements!</a>
<a href="/change" style="margin-right: 20px;">Change an element!</a>
<a href="/insert" style="margin-right: 20px;">Insert new element!</a>

<h2>Welcome to the Go KV store!</h2>

</body>
</html>

```

The `home.gohtml` file is static, which means that its contents do not change. However, the remaining `gohtml` files display information dynamically.

The contents of `update.gohtml` are as follows:

```

<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>A Key Value Store!</title>
</head>
<body>

<a href="/" style="margin-right: 20px;">Home sweet home!</a>
<a href="/list" style="margin-right: 20px;">List all elements!</a>
<a href="/change" style="margin-right: 20px;">Change an element!</a>
<a href="/insert" style="margin-right: 20px;">Insert new element!</a>

{{if .Success}}
    <h1>Element updated!</h1>
{{else}}
<h1>Please fill in the fields:</h1>
<form method="POST">
    <label>Key:</label><br />
    <input type="text" name="key"><br />
    <label>Name:</label><br />
    <input type="text" name="name"><br />
    <label>Surname:</label><br />
    <input type="text" name="surname"><br />
    <label>Id:</label><br />
    <input type="text" name="id"><br />
    <input type="submit">
</form>
{{end}}
```

```

</body>
</html>
```

The preceding code is mainly HTML code. Its most interesting part is the `if` statement, which specifies whether you should see the form or the `Element updated!` message.

Finally, the contents of `insert.gohtml` are as follows:

```

<!doctype html>
<html lang="en">
<head>
```

```

<meta charset="UTF-8">
<title>A Key Value Store!</title>
</head>
<body>

<a href="/" style="margin-right: 20px;">Home sweet home!</a>
<a href="/list" style="margin-right: 20px;">List all elements!</a>
<a href="/change" style="margin-right: 20px;">Change an element!</a>
<a href="/insert" style="margin-right: 20px;">Insert new element!</a>

{{if .Success}}
    <h1>Element inserted!</h1>
{{else}}
    <h1>Please fill in the fields:</h1>
    <form method="POST">
        <label>Key:</label><br />
        <input type="text" name="key"><br />
        <label>Name:</label><br />
        <input type="text" name="name"><br />
        <label>Surname:</label><br />
        <input type="text" name="surname"><br />
        <label>Id:</label><br />
        <input type="text" name="id"><br />
        <input type="submit">
    </form>
{{end}}


</body>
</html>

```

As you can see, `insert.gohtml` and `update.gohtml` are identical apart from the text in the `<title>` tag.

Executing `kvWeb.go` will generate the following output on a UNIX shell:

```

$ go run kvWeb.go
Loading /tmp/dataFile.gob
Using default port number: :8001
Serving localhost:8001 for /
Serving localhost:8001 for /favicon.ico
Listing the contents of the KV store!
Serving localhost:8001 for /favicon.ico
Inserting an element to the KV store!
Serving localhost:8001 for /favicon.ico
Inserting an element to the KV store!
Add operation failed!
Serving localhost:8001 for /favicon.ico
Inserting an element to the KV store!
Serving localhost:8001 for /favicon.ico
Inserting an element to the KV store!
Saving /tmp/dataFile.gob
Serving localhost:8001 for /favicon.ico
Inserting an element to the KV store!
Serving localhost:8001 for /favicon.ico
Changing an element of the KV store!
Serving localhost:8001 for /favicon.ico

```

Additionally, what is really interesting is how you can interact with `kvWeb.go` from a web browser.

The home page of the website, as defined in `home.gohtml`, can be seen in the following figure:

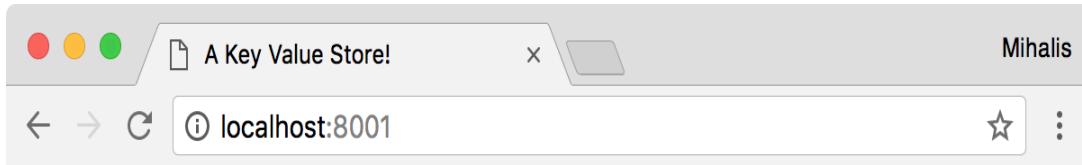
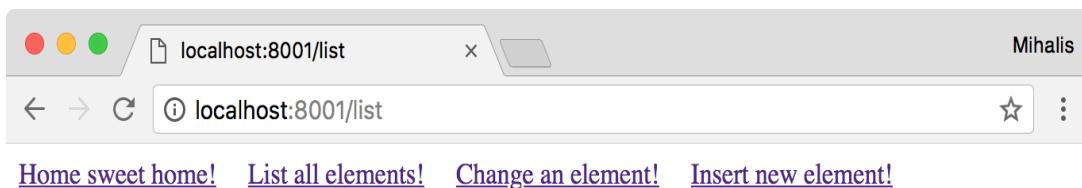


Figure 12.3: The static home page of our web application

The next figure presents the contents of the key-value store:



The contents of the KV store are:

- 12 with value: {Mihalis Tsoukalos 123}
- 21 with value: {Dimitris Tsoukalos 123}
- 123 with value: {Vassilis Tsoukalos 123456}

Figure 12.4: The contents of the key-value store

The next figure shows the appearance of the web page that allows you to add new data to the key-value store using the web interface of the `kvWeb.go` web application:

Figure 12.5: Adding a new entry to the key-value store

The next figure shows you how to update the value of an existing key using the web interface of the `kvWeb.go` web application:

Figure 12.6: Updating the value of a key in the key-value store

The `kvWeb.go` web application is far from perfect, so feel free to improve it as an exercise!



This section illustrated how you can develop entire websites and web applications in Go. Although your requirements will undoubtedly vary, the techniques are the same as the ones used for `kvWeb.go`. Notice that custom-made sites are considered more secure than sites created using some popular content management systems.

HTTP tracing

Go supports **HTTP tracing** with the help of the `net/http/httptrace` standard package. That package allows you to trace the phases of an HTTP request. So, the use of the `net/http/httptrace` standard Go package will be illustrated in `httpTrace.go`, which is going to be presented in five parts.

The first part of `httpTrace.go` is as follows:

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "net/http/httptrace"
    "os"
)
```

As you might expect, you need to import the `net/http/httptrace` package in order to enable HTTP tracing.

The second part of `httpTrace.go` contains the following Go code:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: URL\n")
        return
    }

    URL := os.Args[1]
    client := http.Client{}
```

In this part, we read the command-line arguments and create a new `http.Client` variable.

We will talk a little bit more about the `http.Client` object. The `http.Client` object offers a way to send a request to a server and get a reply.

Its `Transport` field permits you to set various HTTP details instead of using the default values.

Notice that you should never use the default values of the `http.Client` object in production software because those values do not specify request timeouts, which can jeopardize the performance of your programs and your goroutines. Additionally, by design, `http.Client` objects can be safely used in concurrent programs.

The third code segment of `httpTrace.go` is contained in the following Go code:

```
req, _ := http.NewRequest("GET", URL, nil)
trace := &httptrace.ClientTrace{
    GotFirstResponseByte: func() {
        fmt.Println("First response byte!")
    },
    GotConn: func(connInfo httptrace.GotConnInfo) {
        fmt.Printf("Got Conn: %v\n", connInfo)
    },
    DNSDone: func(dnsInfo httptrace.DNSDoneInfo) {
        fmt.Printf("DNS Info: %v\n", dnsInfo)
    },
    ConnectStart: func(network, addr string) {
        fmt.Println("Dial start")
    },
    ConnectDone: func(network, addr string, err error) {
        fmt.Println("Dial done")
    },
    WroteHeaders: func() {
        fmt.Println("Wrote headers")
    },
}
```

The preceding code is all about tracing HTTP requests. The `httptrace.ClientTrace` object defines the events that interest us. When such an event occurs, the relevant code is executed. You can find more information about supported events and their purpose in the documentation of the `net/http/httptrace` package.

The fourth part of the `httpTrace.go` utility is as follows:

```
req = req.WithContext(httptrace.WithClientTrace(req.Context(), trace))
fmt.Println("Requesting data from server!")
_, err := http.DefaultTransport.RoundTrip(req)
if err != nil {
    fmt.Println(err)
    return
}
```

The `httptrace.WithClientTrace()` function returns a new context based on the given parent context; while the `http.DefaultTransport.RoundTrip()` method wraps `http.DefaultTransport.RoundTrip` in order to tell it to keep track of the current request.

Notice that Go HTTP tracing has been designed to trace the events of a single `http.Transport.RoundTrip`. However, as you may have multiple URL redirects when serving a single HTTP request, you need to be able to identify the current request.

The remaining Go code for `httpTrace.go` is as follows:

```
    response, err := client.Do(req)
    if err != nil {
        fmt.Println(err)
        return
    }

    io.Copy(os.Stdout, response.Body)
}
```

This last part is about performing the actual request to the web server using `Do()`, getting the HTTP data, and displaying it on the screen.

Executing `httpTrace.go` will generate the following very informative output:

```
$ go run httpTrace.go http://localhost:8001/
Requesting data from server!
DNS Info: {Addrs:[{IP:::1 Zone:} {IP:127.0.0.1 Zone:}] Err:<nil> Coalesced:false}
Dial start
Dial done
Got Conn: {Conn:0xc420142000 Reused:false WasIdle:false IdleTime:0s}
Wrote headers
First response byte!
DNS Info: {Addrs:[{IP:::1 Zone:} {IP:127.0.0.1 Zone:}] Err:<nil> Coalesced:false}
Dial start
Dial done
Got Conn: {Conn:0xc420142008 Reused:false WasIdle:false IdleTime:0s}
Wrote headers
First response byte!
Serving: /
```

Be aware that since `httpTrace.go` prints the full HTML response from the HTTP server, you might get lots of output when you test it on a real web server, which is the main reason for using the web server developed in `www.go` here.

If you have the time to look at the source code for the `net/http/httptrace` package at <https://golang.org/src/net/http/httptrace/trace.go>, you will immediately realize that



`net/http/httptrace` is a pretty low-level package that uses the `context` package, the `reflect` package, and the `internal/nettrace` package to implement its functionality. Remember that you can do this for any code in the standard library because Go is a completely open-source project.

Testing HTTP handlers

In this section, we are going to learn how to test HTTP handlers in Go. We will begin with the Go code for `www.go` and modify it where needed.

The new version of `www.go` is called `testwww.go`, and it will be presented in three parts. The first code portion of `testwww.go` is as follows:

```
package main

import (
    "fmt"
    "net/http"
    "os"
)

func CheckStatusOK(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusOK)
    fmt.Fprintf(w, `Fine!`)
}
```

The second part of `testwww.go` is contained in the following Go code:

```
func StatusNotFound(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusNotFound)
}

func MyHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served: %s\n", r.Host)
}
```

The remaining Go code for `testwww.go` is as follows:

```
func main() {
    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Using default port number: ", PORT)
    } else {
        PORT = ":" + arguments[1]
    }

    http.HandleFunc("/CheckStatusOK", CheckStatusOK)
    http.HandleFunc("/StatusNotFound", StatusNotFound)
    http.HandleFunc("/", MyHandler)

    err := http.ListenAndServe(PORT, nil)
    if err != nil {
```

```

        fmt.Println(err)
    return
}
}

```

We now need to start testing `testWWW.go`, which means that we should create a `testWWW_test.go` file. The contents of that file will be presented in four parts.

The first part of `testWWW_test.go` is contained in the following Go code:

```

package main

import (
    "fmt"
    "net/http"
    "net/http/httptest"
    "testing"
)

```

Notice that you need to import the `net/http/httptest` standard Go package in order to test web applications in Go.

The second code portion of `testWWW_test.go` contains the following code:

```

func TestCheckStatusOK(t *testing.T) {
    req, err := http.NewRequest("GET", "/CheckStatusOK", nil)
    if err != nil {
        fmt.Println(err)
        return
    }

    rr := httptest.NewRecorder()
    handler := http.HandlerFunc(CheckStatusOK)
    handler.ServeHTTP(rr, req)
}

```

The `httptest.NewRecorder()` function returns an `httptest.ResponseRecorder` object, and it is used to record the HTTP response.

The third part of `testWWW_test.go` is as follows:

```

    status := rr.Code
    if status != http.StatusOK {
        t.Errorf("handler returned %v", status)
    }

    expect := `Fine!
    if rr.Body.String() != expect {
        t.Errorf("handler returned %v", rr.Body.String())
    }
}

```

You first check that the response code is as expected, and then you verify that the body of the response is also what you expected.

The remaining Go code for `testWWW_test.go` is as follows:

```
func TestStatusNotFound(t *testing.T) {
    req, err := http.NewRequest("GET", "/StatusNotFound", nil)
    if err != nil {
        fmt.Println(err)
        return
    }

    rr := httptest.NewRecorder()
    handler := http.HandlerFunc(StatusNotFound)
    handler.ServeHTTP(rr, req)

    status := rr.Code
    if status != http.StatusNotFound {
        t.Errorf("handler returned %v", status)
    }
}
```

This test function verifies that the `StatusNotFound()` function of the `main` package works as expected.

Executing the two test functions of `testWWW_test.go` will generate the following output:

```
$ go test testWWW.go testWWW_test.go -v --count=1
== RUN TestCheckStatusOK
--- PASS: TestCheckStatusOK (0.00s)
== RUN TestStatusNotFound
--- PASS: TestStatusNotFound (0.00s)
PASS
ok    command-line-arguments      (cached)
```

Creating a web client in Go

In this section, you will learn more about developing web clients in Go. The name of the web client utility is `webClient.go`, and it is going to be presented in four parts.

The first part of `webClient.go` contains the following Go code:

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "os"
    "path/filepath"
)
```

The second part of `webClient.go` is where you read the desired URL as a command-line argument:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: %s URL\n", filepath.Base(os.Args[0]))
        return
    }

    URL := os.Args[1]
```

The third code portion of `webClient.go` is where the real action takes place:

```
    data, err := http.Get(URL)

    if err != nil {
        fmt.Println(err)
        return
    }
```

All of the work is done by the `http.Get()` call, which is pretty convenient when you do not want to deal with parameters and options. However, this type of call gives you no flexibility about the process. Notice that `http.Get()` returns an `http.Response` variable.

The remaining Go code is as follows:

```
    } else {
        defer data.Body.Close()
        _, err := io.Copy(os.Stdout, data.Body)
        if err != nil {
            fmt.Println(err)
            return
        }
    }
}
```

What the previous code does is copy the contents of the `Body` field of the `http.Response` structure to standard output.

Executing `webClient.go` will generate the following type of output (note that only a small portion of the output is presented here):

```
$ go run webClient.go http://www.mtsoukalos.eu/ | head -20
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML+RDFa 1.0//EN"
"http://www.w3.org/MarkUp/DTD/xhtml-rdfa-1.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" version="XHTML+RDFa 1.0" dir="ltr"
  xmlns:content="http://purl.org/rss/1.0/modules/content/"
  xmlns:dc="http://purl.org/dc/terms/"
  xmlns:foaf="http://xmlns.com/foaf/0.1/"
  xmlns:og="http://ogp.me/ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:sioc="http://rdfs.org/sioc/ns#"
  xmlns:siocTypes="http://rdfs.org/sioc/types#"
  xmlns:skos="http://www.w3.org/2004/02/skos/core#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#">

<head profile="http://www.w3.org/1999/xhtml/vocab">
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <link rel="shortcut icon" href="http://www.mtsoukalos.eu/misc/favicon.ico" type="image/vnd.microsoft.icon" />
  <meta name="HandheldFriendly" content="true" />
  <meta name="MobileOptimized" content="width" />
  <meta name="Generator" content="Drupal 7 (http://drupal.org)" />
```

The main problem with `webClient.go` is that it gives you almost no control over the process – you either get the entire HTML output or nothing at all!

Making your Go web client more advanced

As the web client of the previous section is relatively simplistic and does not give you any flexibility, in this section, you will learn how to read a URL more elegantly without using the `http.Get()` function, and with more options. The name of the utility is `advancedWebClient.go` and it is going to be presented in five parts.

The first code segment of `advancedWebClient.go` contains the following Go code:

```
package main

import (
    "fmt"
    "net/http"
    "net/http/httputil"
    "net/url"
    "os"
    "path/filepath"
    "strings"
    "time"
)
```

The second part of `advancedWebClient.go` is as follows:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: %s URL\n", filepath.Base(os.Args[0]))
        return
    }

    URL, err := url.Parse(os.Args[1])
    if err != nil {
        fmt.Println("Error in parsing:", err)
        return
    }
}
```

The third code portion of `advancedWebClient.go` contains the following Go code:

```
c := &http.Client{
    Timeout: 15 * time.Second,
}
request, err := http.NewRequest("GET", URL.String(), nil)
if err != nil {
    fmt.Println("Get:", err)
    return
}

httpData, err := c.Do(request)
if err != nil {
    fmt.Println("Error in Do():", err)
    return
}
```

The `http.NewRequest()` function returns an `http.Request` object given a method, a URL, and an optional body. The `http.Do()` function sends an HTTP request (`http.Request`) using an `http.Client` and gets an HTTP response (`http.Response`). So, `http.Do()` does the job of `http.Get()` in a more comprehensive way.

The "GET" string used in `http.NewRequest()` can be replaced by `http.MethodGet`.

The fourth part of `advancedWebClient.go` is contained in the following Go code:

```
fmt.Println("Status code:", httpData.Status)
header, _ := httputil.DumpResponse(httpData, false)
fmt.Print(string(header))

contentType := httpData.Header.Get("Content-Type")
characterSet := strings.SplitAfter(contentType, "charset=")
if len(characterSet) > 1 {
    fmt.Println("Character Set:", characterSet[1])
}

if httpData.ContentLength == -1 {
    fmt.Println("ContentLength is unknown!")
} else {
    fmt.Println("ContentLength:", httpData.ContentLength)
}
```

In the preceding code, you can see how to start searching the server response in order to find what you want.

The last part of the `advancedWebClient.go` utility is as follows:

```
length := 0
var buffer [1024]byte
r := httpData.Body
for {
    n, err := r.Read(buffer[0:])
    if err != nil {
        fmt.Println(err)
        break
    }
    length = length + n
}
fmt.Println("Calculated response data length:", length)
```

In the preceding code, you can see a technique for discovering the size of the server HTTP response. If you want to display the HTML output on your screen, you can print the contents of the `r` buffer variable.

Using `advancedWebClient.go` to visit a web page will generate the following type of output, which is much richer than before:

```
$ go run advancedWebClient.go http://www.mtsoukalos.eu
Status code: 200 OK
HTTP/1.1 200 OK
Accept-Ranges: bytes
Age: 0
Cache-Control: no-cache, must-revalidate
Connection: keep-alive
Content-Language: en
Content-Type: text/html; charset=utf-8
Date: Sat, 24 Mar 2018 18:52:17 GMT
Expires: Sun, 19 Nov 1978 05:00:00 GMT
Server: Apache/2.4.25 (Debian) PHP/5.6.33-0+deb8u1 mod_wsgi/4.5.11 Python/2.7
Vary: Accept-Encoding
Via: 1.1 varnish (Varnish/5.0)
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
X-Generator: Drupal 7 (http://drupal.org)
X-Powered-By: PHP/5.6.33-0+deb8u1
X-Varnish: 886025

Character Set: utf-8
ContentLength is unknown!
EOF
Calculated response data length: 50176
```

Executing `advancedWebClient.go` to visit a different URL will return a slightly different output:

```
$ go run advancedWebClient.go http://www.google.com
Status code: 200 OK
HTTP/1.1 200 OK
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-7
Date: Sat, 24 Mar 2018 18:52:38 GMT
Expires: -1
P3p: CP="This is not a P3P policy! See g.co/p3phelp for more info."
Server: gws
Set-Cookie: 1P_JAR=2018-03-24-18; expires=Mon, 23-Apr-2018 18:52:38 GMT; path=/; domain=.google.gr
Set-Cookie: NID=126=csX1_koD30SJcC_ljAfcM2V8kTfRkppmAamLjINLfclracMxuk6JGe4glc0Pjs8uD00bqGaxkSW-J-ZNDJexG2ZX9pNB9E_dRc2y1KZ05V
X-Frame-Options: SAMEORIGIN
X-Xss-Protection: 1; mode=block

Character Set: ISO-8859-7
ContentLength is unknown!
EOF
Calculated response data length: 10240
```

If you try to fetch an erroneous URL with `advancedWebClient.go`, you will get the following type of output:

```
$ go run advancedWebClient.go http://www.google
Error in Do(): Get http://www.google: dial tcp: lookup www.google: no such host
$ go run advancedWebClient.go www.google.com
Error in Do(): Get www.google.com: unsupported protocol scheme ""
```

Feel free to make any changes you want to `advancedWebClient.go` in order to make the output match your requirements!

Timing out HTTP connections

This section will present a technique for timing out network connections that take too long to finish. Remember that you already know such a technique from [Chapter 10](#), *Concurrency in Go – Advanced Topics*, when we talked about the `context` standard Go package. This technique was presented in the `useContext.go` source code file.

The method that is going to be presented in this section is much easier to implement. The relevant code is saved in `clientTimeout.go` and it is going to be presented in four parts. The utility accepts two command-line arguments, which are the URL and the timeout period in seconds. Please notice that the second parameter is optional.

The first part of the code portion for `clientTimeout.go` is as follows:

```
package main

import (
    "fmt"
    "io"
    "net"
    "net/http"
    "os"
    "path/filepath"
    "strconv"
    "time"
)
var timeout = time.Duration(time.Second)
```

The second code segment for `clientTimeout.go` contains the following Go code:

```
func Timeout(network, host string) (net.Conn, error) {
    conn, err := net.DialTimeout(network, host, timeout)
    if err != nil {
        return nil, err
    }
    conn.SetDeadline(time.Now().Add(timeout))
    return conn, nil
}
```

You will learn more about the functionality of `SetDeadline()` in the next subsection. The `Timeout()` function will be used by the `Dial` field of an `http.Transport` variable.

The third code portion for `clientTimeOut.go` is contained in the following code:

```
func main() {
    if len(os.Args) == 1 {
        fmt.Printf("Usage: %s URL TIMEOUT\n", filepath.Base(os.Args[0]))
        return
    }

    if len(os.Args) == 3 {
        temp, err := strconv.Atoi(os.Args[2])
        if err != nil {
            fmt.Println("Using Default Timeout!")
        } else {
            timeout = time.Duration(time.Duration(temp) * time.Second)
        }
    }

    URL := os.Args[1]
    t := http.Transport{           Dial: Timeout,       }
```

The remaining Go code for the `clientTimeOut.go` utility is as follows:

```
client := http.Client{
    Transport: &t,
}

data, err := client.Get(URL)
if err != nil {
    fmt.Println(err)
    return
} else {
    defer data.Body.Close()
    _, err := io.Copy(os.Stdout, data.Body)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

The `clientTimeOut.go` web client is going to be tested using the `slowWWW.go` web server developed in [Chapter 10, Concurrency in Go – Advanced Topics](#).

Executing `clientTimeOut.go` twice will generate the following type of output:

```
$ go run clientTimeOut.go http://localhost:8001
Serving: /
Delay: 0
$ go run clientTimeOut.go http://localhost:8001
```

```
| Get http://localhost:8001: read tcp [::1]:57397->[::1]:8001: i/o timeout
```

As you can see from the generated output, the first request had no problem connecting to the desired web server. However, the second `http.Get()` request took longer than expected and therefore timed out.

More information about SetDeadline

The `SetDeadline()` function is used by the `net` package to set the read and write deadlines of a given network connection. Due to the way the `SetDeadline()` function works, you will need to call `SetDeadline()` before any read or write operation. Keep in mind that Go uses deadlines to implement timeouts, so you do not need to reset the timeout every time your application receives or sends any data.

Setting the timeout period on the server side

In this subsection, you will learn how to time out a connection on the server side. You need to do this because there are times when clients take much longer than expected to end an HTTP connection. This usually happens for two reasons. The first reason is bugs in the client software, and the second reason is when a server process is experiencing an attack!

This technique is going to be implemented in the `serverTimeOut.go` source code file, which will be presented in four parts. The first part of `serverTimeOut.go` is as follows:

```
package main

import (
    "fmt"
    "net/http"
    "os"
    "time"
)

func myHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served: %s\n", r.Host)
}
```

The second code portion of `serverTimeOut.go` contains the following Go code:

```
func timeHandler(w http.ResponseWriter, r *http.Request) {
    t := time.Now().Format(time.RFC1123)
    Body := "The current time is:"
    fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>", Body)
    fmt.Fprintf(w, "<h2 align=\"center\">%s</h2>\n", t)
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served time for: %s\n", r.Host)
}
```

The third code segment of `serverTimeOut.go` is shown in the following Go code:

```
func main() {
    PORT := ":8001"
```

```

arguments := os.Args
if len(arguments) == 1 {
    fmt.Printf("Listening on http://0.0.0.0:%s\n", PORT)
} else {
    PORT = ":" + arguments[1]
    fmt.Printf("Listening on http://0.0.0.0:%s\n", PORT)
}

m := http.NewServeMux()
srv := &http.Server{
    Addr:           PORT,
    Handler:        m,
    ReadTimeout:   3 * time.Second,
    WriteTimeout:  3 * time.Second,
}

```

In this case, we are using an `http.Server` structure that supports two kinds of timeouts with its fields. The first one is called `ReadTimeout`, and the second one is called `WriteTimeout`. The value of the `ReadTimeout` field specifies the maximum duration allowed to read the entire request, including the body.

The value of the `WriteTimeout` field specifies the maximum time duration before timing out the writing of the response. Put simply, this is the time from the end of the request header read to the end of the response write.

The remaining Go code for `serverTimeOut.go` is as follows:

```

m.HandleFunc("/time", timeHandler)
m.HandleFunc("/", myHandler)

err := srv.ListenAndServe()
if err != nil {
    fmt.Println(err)
    return
}

```

We are now going to execute `serverTimeOut.go` in order to interact with it using `nc(1)`:

```
$ go run serverTimeOut.go
Listening on http://0.0.0.0:8001
```

For the `nc(1)` part, which in this case acts as a dummy HTTP client, you should issue the following command to connect to `serverTimeOut.go`:

```
$ time nc localhost 8001
```

real	0m3.012s
user	0m0.001s
sys	0m0.002s

As we did not issue any commands, the HTTP server ended the connection. The output of the `time(1)` utility verifies the time it took the server to close the connection.

Yet another way to time out

This subsection will present yet another way to timeout an HTTP connection from the client side. As you will see, this is the simplest form of timeout, because you just need to use an `http.Client` object and set its `Timeout` field to the desired timeout value.

The name of the utility that showcases this last type of timeout is `anotherTimeOut.go`, and it is going to be presented in four parts.

The first part of `anotherTimeOut.go` is as follows:

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "os"
    "strconv"
    "time"
)
var timeout = time.Duration(time.Second)
```

The second part of `anotherTimeOut.go` contains the following Go code:

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Please provide a URL")
        return
    }

    if len(os.Args) == 3 {
        temp, err := strconv.Atoi(os.Args[2])
        if err != nil {
            fmt.Println("Using Default Timeout!")
        } else {
            timeout = time.Duration(time.Duration(temp) * time.Second)
        }
    }

    URL := os.Args[1]
```

The third code segment for `anotherTimeOut.go` contains the following Go code:

```
client := http.Client{
    Timeout: timeout,
}
client.Get(URL)
```

This is the place where the timeout period is defined using the `Timeout` field of the `http.Client` variable.

The last code portion for `anotherTimeOut.go` is as follows:

```
data, err := client.Get(URL)
if err != nil {
```

```
        fmt.Println(err)
        return
    } else {
        defer data.Body.Close()
        _, err := io.Copy(os.Stdout, data.Body)
        if err != nil {
            fmt.Println(err)
            return
        }
    }
}
```

Executing `anotherTimeOut.go` and interacting with the `slowWWW.go` web server developed in [Chapter 10, Concurrency in Go – Advanced Topics](#), will generate the following type of output:

```
$ go run anotherTimeOut.go http://localhost:8001
Get http://localhost:8001: net/http: request canceled (Client.Timeout exceeded while awaiting headers)
$ go run anotherTimeOut.go http://localhost:8001 15
Serving: /
Delay: 8
```

The Wireshark and tshark tools

This section will briefly mention the powerful **Wireshark** and **tshark** utilities. Wireshark, which is a graphical application, is the dominant tool for analyzing network traffic of almost any kind. Although Wireshark is very powerful, there may be times when you need something lighter that you can execute remotely without a graphical user interface. In such situations, you can use tshark, which is the command-line version of Wireshark.

Unfortunately, talking more about Wireshark and tshark is beyond the scope of this chapter.

gRPC and Go

Strictly speaking, **gRPC** is a protocol built on HTTP/2 that allows you to create services easily. gRPC can use protocol buffers to specify an **interface definition language**, as well as to specify the format of the interchanged messages. gRPC clients and servers can be written in any programming language without the need to have clients written in the same programming language as their servers.

The process that will be presented here has three steps. The first step is creating the interface definition file, the second step is the development of the gRPC client, and the third step is the development of the gRPC server that will work with the gRPC client.

Defining the interface definition file

As you learned in the previous section, before we begin developing the gRPC client and server for our service, we need to define some data structures and protocols that will be used by them.

Protocol Buffers (protobuf) is a method for serializing structured data. As protobuf uses the binary format, it takes up less space than plain text serialization formats such as JSON and XML. However, it needs to be encoded and decoded in order to be machine-readable and human-readable, respectively.

As a result, in order to be able to use protobuf in your applications, you will need to download the necessary tools that allow you to work with it – the funny thing is that most protobuf tools are written in Go because Go is great at creating command-line tools!

On a macOS machine, the necessary tools can be downloaded using **Homebrew**, as follows:

```
| $ brew install protobuf
```

There is an extra step that needs to be completed in order to get protobuf support for Go, because Go is not supported by default. This step requires the execution of the following command:

```
| $ go get -u github.com/golang/protobuf/protoc-gen-go
```

Among other things, the previous command downloads the `protoc-gen-go` executable and puts it in `~/go/bin`, which is the value of `$GOPATH/bin` on my machine. However, for the `protoc` compiler to find it, you will need to include that directory in your `PATH` environment variable, which on `bash(1)` and `zsh(1)` can be done as follows:

```
| $ export PATH=$PATH:~/go/bin
```

Once you have the necessary tools, you will need to define the structures and the functions that will be used between the gRPC client and the gRPC server. In our case, the interface definition file, which is saved in `api.proto`, will be the following:

```
syntax = "proto3";

package message_service;

message Request {
    string text      = 1;
    string subtext   = 2;
}

message Response {
    string text      = 1;
    string subtext   = 2;
}

service MessageService {
    rpc SayIt (Request) returns (Response);
}
```

The gRPC server and the gRPC client that are going to be developed will support that protocol, which more or less defines a simple, basic messaging service with two basic types named `Request` and `Response`, and a single function named `SayIt()`.

However, we are not done yet, as `api.proto` needs to be processed and compiled by the protobuf tool, which in this case is the protobuf compiler that can be found at `/usr/local/bin/protoc`. This should be done as follows:

```
$ protoc -I . --go_out=plugins=grpc:. api.proto
```

After executing these commands, you will have an extra file on your computer named `api.pb.go`:

```
$ ls -l api.pb.go
-rw-r--r-- 1 mtsouk  staff  7320 May  4 18:31 api.pb.go
```

So, for Go, the protobuf compiler generates `.pb.go` files that, among other things, contain a type for each message type in your interface definition file and a Go interface that needs to be implemented in the gRPC server. The `.pb.go` file extension will be different for other programming languages.

The first lines of `api.pb.go` are the following:

```
// Code generated by protoc-gen-go. DO NOT EDIT.  
// source: api.proto  
  
package message_service  
  
import (  
    context "context"  
    fmt "fmt"  
    proto "github.com/golang/protobuf/proto"  
    grpc "google.golang.org/grpc"  
    codes "google.golang.org/grpc/codes"  
    status "google.golang.org/grpc/status"  
    math "math"  
)
```

These lines say that you should not edit `api.pb.go` on your own and that the name of the package is `message_service`. For your Go programs to be able to find all protobuf-related files, it is advisable to put them in their own GitHub repository. For the purposes of this example, that repository will be <https://github.com/mactsouk/protobuf>. This also means that you will need to get that repository using the following command:

```
$ go get github.com/mactsouk/protobuf
```



If you ever update `api.proto` or any other similar file on your local machine, you should remember to do two things: first, update the GitHub repository, and second, execute `go get -u -v` followed by the address of the remote GitHub repository to get the updates on your local machine.

Now we are ready to continue with developing the Go code for the gRPC client and server.

Note that if the required Go package is not present on your computer, you will get the following error message when you try to compile the interface definition file:

```
$ protoc -I . --go_out=plugins=grpc:. api.proto  
protoc-gen-go: program not found or is not executable  
--go_out: protoc-gen-go: Plugin failed with status code 1.
```

The gRPC client

In this subsection, we are going to develop a gRPC client in Go, which is going to be saved in the `gClient.go` file, which will be presented in three parts.

The first part of `gClient.go` is as follows:

```
package main

import (
    "fmt"
    p "github.com/mactsouk/protobuf"
    "golang.org/x/net/context"
    "google.golang.org/grpc"
)
var port = ":8080"
```

Notice that you should not need to download the required external Go packages because you already downloaded them when you executed the following command in order to compile the interface definition language file and create the Go-related output files:

```
$ go get -u github.com/golang/protobuf/protoc-gen-go
```

Keep in mind that `-u` tells `go get` to update the named packages along with their dependencies. If you use `-v`, you will get a better understanding of what actually happens, as this makes `go get` generate extra debugging information.

The second part of `gClient.go` contains the following Go code:

```
func AboutToSayIt(ctx context.Context, m p.MessageServiceClient, text string) (*p.Response, error) {
    request := &p.Request{
        Text:   text,
        Subtext: "New Message!",
    }
    r, err := m.SayIt(ctx, request)
    if err != nil {
        return nil, err
    }
    return r, nil
}
```

The way you name the `AboutToSayIt()` function is totally up to you. However, the signature of the function must contain a `context.Context` parameter, as well as a `MessageServiceClient` parameter in order to be able to call the `SayIt()` function later on. Notice that, for the client, you don't need to implement any functions of the interface definition language – you just have to call them.

The last part of `gClient.go` is as follows:

```
func main() {
    conn, err := grpc.Dial(port, grpc.WithInsecure())
    if err != nil {
        fmt.Println("Dial:", err)
        return
    }

    client := p.NewMessageServiceClient(conn)
    r, err := AboutToSayIt(context.Background(), client, "My Message!")
    if err != nil {
        fmt.Println(err)
    }

    fmt.Println("Response Text:", r.Text)
    fmt.Println("Response SubText:", r.Subtext)
}
```

You need to call `grpc.Dial()` in order to connect to the gRPC server and create a new client using `NewMessageServiceClient()`. The name of the latter function depends on the value of the `package` statement found in

the `api.proto` file. The message that is sent to the gRPC server is hardcoded in the code for `gClient.go`.

There is no point in executing the gRPC client without having a gRPC server, so you should wait for the next section. However, if you really want to find out the generated error message, you can still try:

```
$ go run gClient.go
rpc error: code = Unavailable desc = all SubConns are in TransientFailure, latest connection error: connection error: desc = "!
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x8 pc=0x13d8afe]

goroutine 1 [running]:
main.main()
    /Users/mtsouk/ch12/gRPC/gClient.go:41 +0x22e
exit status 2
```

The gRPC server

In this subsection, you will learn how to develop a gRPC server in Go. The name of the program will be `gServer.go` and it is going to be presented in four parts.

The first part of `gServer.go` is as follows:

```
package main

import (
    "fmt"
    p "github.com/mactsouk/protobuf"
    "golang.org/x/net/context"
    "google.golang.org/grpc"
    "net"
)
```

As the name of the Go package with the interface definition language is `message_service`, which is pretty long, I've introduced an alias for it here, which is pretty short and convenient (`p`).

The `golang.org/x/net/context` and `google.golang.org/grpc` packages were downloaded earlier along with the other dependencies of the `github.com/golang/protobuf/protoc-gen-go` package, which means that you will not need to download any of them.



As of Go 1.7, the `golang.org/x/net/context` package is available in the standard library under the name `context`. Therefore, you are free to replace it with `context` if you want.

The second part of `gServer.go` contains the following Go code:

```
type MessageServer struct {
}

var port = ":8080"
```

You need that empty structure in order to be able to create the gRPC server later on in your Go code.

The third part of `gServer.go` is where you implement the interface:

```

func (MessageServer) SayIt(ctx context.Context, r *p.Request) (*p.Response, error) {
    fmt.Println("Request Text:", r.Text)
    fmt.Println("Request SubText:", r.Subtext)

    response := &p.Response{
        Text:      r.Text,
        Subtext:   "Got it!",
    }

    return response, nil
}

```

The signature of the `SayIt()` function depends on the data in the interface definition language file and can be found in `api.pb.go`.

What the `SayIt()` function does is send the contents of the `Text` field back to the client while changing the contents of the `Subtext` field.

The last part of `gServer.go` is as follows:

```

func main() {
    server := grpc.NewServer()
    var messageServer MessageServer
    p.RegisterMessageServiceServer(server, messageServer)
    listen, err := net.Listen("tcp", port)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println("Serving requests...")
    server.Serve(listen)
}

```

In order to test the connection, you will need to begin by executing `gServer.go`:

```

$ go run gServer.go
Serving requests...

```

Executing `gClient.go` while `gServer.go` is running will generate the following kind of output:

```

$ go run gClient.go
Response Text: My Message!
Response SubText: Got it!

```

Once you execute `gClient.go`, you will see the following output in `gServer.go`:

```

Request Text: My Message!
Request SubText: New Message!

```

Although `gClient.go` will automatically end, you will need to manually terminate `gServer.go`.

Additional resources

The following resources will definitely broaden your horizons, so please find some time to read them:

- The official web page for the Apache web server is located at <http://httpd.apache.org/>.
- The official web page for the Nginx web server is located at <http://nginx.org/>.
- Should you wish to learn more about the Internet or TCP/IP, and its various services, you should start by reading the **RFC** documents. One of the places that you can find such documents is at <http://www.rfc-archive.org/>.
- Visit the website of both Wireshark and tshark at <https://www.wireshark.org/> to learn more about them.
- Visit the documentation page for the `net` standard Go package, which can be found at <https://golang.org/pkg/net/>. This is one of the largest documentation pages in the official Go documentation.
- Visit the documentation page for the `net/http` Go package at <https://golang.org/pkg/net/http/>.
- If you want to create a website without writing any Go code, you can try the **Hugo utility**, which is written in Go. You can learn more about the Hugo framework at <https://gohugo.io/>. However, what would really be interesting and educational for every Go programmer is to look at its Go code, which can be found at <https://github.com/gohugoio/hugo>.
- You can visit the documentation page for the `net/http/httptrace` package at <https://golang.org/pkg/net/http/httptrace/>.
- You can find the documentation page for the `net/http/pprof` package at <https://golang.org/pkg/net/http/pprof/>.
- Visit the manual page for the `nc(1)` command-line utility to learn more about its capabilities and its various command-line options.
- The `httpstat` utility, which was developed by Dave Cheney, can be found at <https://github.com/davecheney/httpstat>. It is a good example of the use of the `net/http/httptrace` Go package for HTTP tracing.

- You can find more information about the Candy web server at <https://github.com/caddyserver/cadd>.
- You can learn more about protobuf by visiting <https://opensource.google.com/projects/protobuf> and <https://developers.google.com/protocol-buffers/>.
- You can find the Protocol Buffers Language Guide documentation at <https://developers.google.com/protocol-buffers/docs/proto3>.
- The documentation page for the `protobuf` Go package can be found at <https://github.com/golang/protobuf>.
- You can find more information about `ab(1)` by visiting its manual page at <https://httpd.apache.org/docs/2.4/programs/ab.html>.

Exercises

- Write a web client in Go on your own without looking at the code in this chapter.
- Merge the code for `MXrecords.go` and `NSrecords.go` in order to create a single utility that does both jobs based on its command-line arguments.
- Modify the code for `MXrecords.go` and `NSrecords.go` to also accept IP addresses as input.
- Create a version of `MXrecords.go` and `NSrecords.go` using the `cobra` and `viper` packages.
- Create your own gRPC application using your own interface definition language.
- Make the necessary code changes to `gServer.go` in order to use goroutines and keep the number of clients that it has served.
- Modify `advancedWebClient.go` in order to save the HTML output in an external file.
- Try to implement a simple version of `ab(1)` on your own in Go using goroutines and channels.
- Modify `kvWeb.go` in order to support the `DELETE` and `LOOKUP` operations found in the original version of the key-value store.
- Modify `httpTrace.go` in order to have a flag that disables the execution of the `io.Copy(os.Stdout, response.Body)` statement.

Summary

This chapter presented Go code for programming web clients and web servers, as well as creating a website in Go. You also learned about the `http.Response`, `http.Request`, and `http.Transport` structures, which allow you to define the parameters of an HTTP connection.

Additionally, you learned how to develop gRPC applications, how to get the network configuration of a UNIX machine using Go code, and how to perform DNS lookups in a Go program, including getting the NS and MX records of a domain.

Finally, I talked about Wireshark and tshark, which are two very popular utilities that allow you to capture and, most importantly, analyze network traffic. At the beginning of this chapter, I also mentioned the `nc(1)` utility.

In the next chapter of this book, we will continue our discussion of network programming in Go. However, this time, I will present lower-level Go code that will allow you to develop TCP clients and servers, as well as UDP client and server processes. Additionally, you will learn about creating RCP clients and servers.

Network Programming – Building Your Own Servers and Clients

The previous chapter discussed topics related to network programming, including developing HTTP clients, HTTP servers, and web applications; performing DNS lookups; and timing out HTTP connections.

This chapter will take you to the next level by showing you how to work with HTTPS and how to program your own TCP clients and servers, as well as your own UDP clients and servers.

Additionally, the chapter will demonstrate how you can program a concurrent TCP server using two examples. The first example will be relatively simple, as the concurrent TCP server will just calculate and return numbers in the Fibonacci sequence. However, the second example will use the code of the `keyValue.go` application from [Chapter 4](#), *The Uses of Composite Types*, as its foundation and convert the **key-value store** into a concurrent TCP application that can operate without the need for a web browser. In this chapter, you will learn about the following topics:

- Working with HTTPS traffic
- The `net` standard Go package
- Developing TCP clients and TCP servers
- Developing a concurrent TCP server
- Developing UDP clients and UDP servers
- Modifying `kvSaveLoad.go` from [Chapter 8](#), *Telling a UNIX System What to Do*, so that it can serve requests through TCP connections
- Executing a TCP/IP server on a Docker image
- Creating an RCP client and RCP server

Working with HTTPS traffic

Before you create TCP/IP servers with Go, this chapter will teach you how to work with the HTTPS protocol in Go, which is the secure version of the HTTP protocol. Notice that the default TCP port for HTTPS is 443, but you can use any port number you want as long as you put it in the URL.

Creating certificates

In order to be able to follow and execute the code examples of this section, you will first need to create certificates because HTTPS requires them. On a macOS Mojave machine, you will need to execute the following commands:

```
$ openssl genrsa -out server.key 2048
Generating RSA private key, 2048 bit long modulus
.....+++
.....+++
e is 65537 (0x10001)
$ openssl ecparam -genkey -name secp384r1 -out server.key
$ openssl req -new -x509 -sha256 -key server.key -out server.crt -days 3650
```

The last command will ask you some questions that are not displayed here. It does not really matter what information you put in, which means that you are free to leave most of the answers blank. What you will have after executing these commands are the following two files:

```
$ ls -l server.crt
-rw-r--r-- 1 mtsouk staff 501 May 16 09:42 server.crt
$ ls -l server.key
-rw-r--r-- 1 mtsouk staff 359 May 16 09:42 server.key
```

Notice that if a certificate is self-signed, as the one that we have just generated is, you will need to use the `InsecureSkipVerify: true` option in the `http.Transport` structure for the HTTPS client to work – this will be illustrated shortly.

Now, you should create a certificate for the client, which requires the execution of a single command:

```
$ openssl req -x509 -nodes -newkey rsa:2048 -keyout client.key -out client.crt -days 3650 -subj "/"
Generating a 2048 bit RSA private key
.....+++
.....+++
writing new private key to 'client.key'
-----
```

The previous command will generate two new files:

```
$ ls -l client.*
-rw-r--r-- 1 mtsouk staff 924 May 16 22:17 client.crt
-rw-r--r-- 1 mtsouk staff 1704 May 16 22:17 client.key
```

We are now ready to continue and talk about creating an HTTPS client.

An HTTPS client

Nowadays, most web sites work using HTTPS instead of HTTP. Therefore, in this subsection, you will learn how to create an HTTPS client. The name of the program will be `httpsClient.go` and it is going to be presented in three parts.



*Depending on the deployment architecture, Go programs might just speak HTTP and some other service (such as the Nginx web server or a cloud-supplied service) might provide the **Secure Sockets Layer (SSL)** part.*

The first part of `httpsClient.go` contains the following code:

```
package main

import (
    "crypto/tls"
    "fmt"
    "io/ioutil"
    "net/http"
    "os"
    "path/filepath"
    "strings"
)
```

The most important package is `crypto/tls`, which, according to its documentation, partially implements **Transport Layer Security (TLS)** 1.2, as specified in RFC 5246, and TLS 1.3, as specified in RFC 8446.

The second part of `httpsClient.go` is as follows:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: %s URL\n", filepath.Base(os.Args[0]))
        return
    }
    URL := os.Args[1]

    tr := &http.Transport{
        TLSClientConfig: &tls.Config{},
    }
    client := &http.Client{Transport: tr}
    response, err := client.Get(URL)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer response.Body.Close()
```

The `http.Transport` structure is configured for TLS using `TLSClientConfig`, which holds another structure named `tls.Config` that, at this point, uses its default values.

The last part of the `httpsClient.go` HTTPS client contains the following code for reading the HTTPS server response and printing it on the screen:

```
    content, _ := ioutil.ReadAll(response.Body)
    s := strings.TrimSpace(string(content))

    fmt.Println(s)
}
```

Executing `httpsClient.go` in order to read a secure web site will generate the following kind of output:

```
$ go run httpsClient.go https://www.google.com
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="el"><head><meta content="text/html; charset=UTF-8"
.
.
.
```

However, `httpsClient.go` might fail in some cases, depending on the server certificate:

```
$ go run httpsClient.go https://www.mtsoukalos.eu/
Get https://www.mtsoukalos.eu/: x509: certificate signed by unknown authority
```

|

The solution to this problem is the use of the `InsecureSkipVerify: true` option in the initialization of `http.Transport`.

You can try that now on your own or wait a little bit for `TLSclient.go`.

A simple HTTPS server

In this subsection, you are going to see the Go code of a HTTPS server. The implementation of the simple HTTPS server is saved in `https.go` and it is going to be presented in three parts.

The first part of `https.go` is as follows:

```
package main

import (
    "fmt"
    "net/http"
)
var PORT = ":1443"
```

The second part of `https.go` contains the following code:

```
func Default(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintf(w, "This is an example HTTPS server!\n")
}
```

This function will handle all incoming HTTPS connections.

The last part of `https.go` is as follows:

```
func main() {
    http.HandleFunc("/", Default)
    fmt.Println("Listening to port number", PORT)

    err := http.ListenAndServeTLS(PORT, "server.crt", "server.key", nil)
    if err != nil {
        fmt.Println("ListenAndServeTLS: ", err)
        return
    }
}
```

The `ListenAndServeTLS()` function is similar to the `ListenAndServe()` function used in the previous chapter – their main difference is that `ListenAndServeTLS()` expects HTTPS connections, whereas `ListenAndServe()` cannot serve HTTPS clients. Additionally, `ListenAndServeTLS()` requires more parameters than `ListenAndServe()` because it uses a certificate file as well as a key file.

Executing `https.go` and connecting to it using the `httpsClient.go` client will generate the following kind of output to the `httpsClient.go` client:

```
$ go run httpsClient.go https://localhost:1443
Get https://localhost:1443: x509: certificate is not valid for any names, but wanted to match localhost
```

Once again, the use of a self-signed certificate will not allow `httpsClient.go` to connect to our HTTPS server, which is an issue with the implementations of both the client and the server. In this case, the output of `https.go` will be as follows:

```
| $ go run https.go  
| Listening to port number :1443  
| 2019/05/17 10:11:21 http: TLS handshake error from [::1]:56716: remote error: tls: bad certificate
```

The HTTPS server developed in this section uses HTTPS via SSL, which is not the most secure option. A better option is TLS – a Go implementation of an HTTPS server that uses TLS will be presented in the next section.

Developing a TLS server and client

In this section, we are going to implement a TLS server named `TLSserver.go`, which will be presented in four parts. This time, the HTTPS server will be better than the `https.go` server implemented in the previous section.

The first part of `TLSserver.go` is as follows:

```
package main

import (
    "crypto/tls"
    "crypto/x509"
    "fmt"
    "io/ioutil"
    "net/http"
)

var PORT = ":1443"

type handler struct {
```

The second part of `TLSserver.go` contains the following code:

```
func (h *handler) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    w.Write([]byte("Hello world!\n"))
}
```

This is the handler function of the web server that handles all client connections.

The third part of `TLSserver.go` is as follows:

```
func main() {
    caCert, err := ioutil.ReadFile("client.crt")
    if err != nil {
        fmt.Println(err)
        return
    }

    caCertPool := x509.NewCertPool()
    caCertPool.AppendCertsFromPEM(caCert)
    cfg := &tls.Config{
        ClientAuth: tls.RequireAndVerifyClientCert,
        ClientCAs:  caCertPool,
    }
}
```

The `x509` Go package parses **X.509**-encoded keys and certificates. You can find more information about it at <https://golang.org/pkg/crypto/x509/>.

The last part of `TLSserver.go` contains the following code:

```
    srv := &http.Server{
        Addr:      PORT,
        Handler:   &handler{},
        TLSConfig: cfg,
    }
    fmt.Println("Listening to port number", PORT)
```

```
|     fmt.Println(srv.ListenAndServeTLS("server.crt", "server.key"))
| }
```

The `ListenAndServeTLS()` call starts the HTTPS server – its full configuration is held in the `http.Server` structure.

If you try to use `httpsClient.go` with `TLSserver.go`, you will get the following kind of output from the client:

```
$ go run httpsClient.go https://localhost:1443
Get https://localhost:1443: x509: certificate is not valid for any names, but wanted to match localhost
```

In this case, the server will generate the following output:

```
| $ go run TLSserver.go
| Listening to port number :1443
| 2019/05/17 10:05:11 http: TLS handshake error from [::1]:56569: remote error: tls: bad certificate
```

As mentioned earlier, in order for the `httpsClient.go` HTTPS client to communicate successfully with `TLSserver.go`, which uses a self-signed certificate, you will need to add the `InsecureSkipVerify: true` option to `http.Transport`. The version of `httpsClient.go` that works with `TLSserver.go` and contains the `InsecureSkipVerify: true` option is saved as `TLSclient.go` and will be presented in four parts.

The first part of `TLSclient.go` is as follows:

```
package main

import (
    "crypto/tls"
    "crypto/x509"
    "fmt"
    "io/ioutil"
    "net/http"
    "os"
    "path/filepath"
)
```

The second part of `TLSclient.go` contains the following code:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: %s URL\n", filepath.Base(os.Args[0]))
        return
    }
    URL := os.Args[1]

    caCert, err := ioutil.ReadFile("server.crt")
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

The third part of `TLSclient.go` is the following:

```
|     caCertPool := x509.NewCertPool()
|
|     caCertPool.AppendCertsFromPEM(caCert)
|     cert, err := tls.LoadX509KeyPair("client.crt", "client.key")
```

```

    if err != nil {
        fmt.Println(err)
        return
    }

    client := &http.Client{
        Transport: &http.Transport{
            TLSClientConfig: &tls.Config{
                RootCAs:           caCertPool,
                InsecureSkipVerify: true,
                Certificates:      []tls.Certificate{cert},
            },
        },
    }

    resp, err := client.Get(URL)
    if err != nil {
        fmt.Println(err)
        return
    }
}

```

In this part, you can see the use of `InsecureSkipVerify`.

The last part of `TLSclient.go` contains the following Go code:

```

htmlData, err := ioutil.ReadAll(resp.Body)
if err != nil {
    fmt.Println(err)
    return
}

defer resp.Body.Close()
fmt.Printf("%v\n", resp.Status)
fmt.Printf(string(htmlData))
}

```

If you try to connect `TLSclient.go` to `TLSserver.go`, you will get the expected output, which is the following:

```

$ go run TLSclient.go https://localhost:1443
200 OK
Hello world!

```

If you try to connect `TLSclient.go` to `https.go`, you will get the expected output, which means that `TLSclient.go` is a pretty good HTTPS client implementation:

```

$ go run TLSclient.go https://localhost:1443
200 OK
This is an example HTTPS server!

```

The net standard Go package

That is enough with HTTPS. It is time to begin talking about the core protocols of TCP/IP, which are TCP, IP, and UDP.

You cannot create a TCP or UDP client or server in Go without using the functionality offered by the `net` package. The `net.Dial()` function is used to connect to a network as a client, whereas the `net.Listen()` function is used to tell a Go program to accept network connections and thus act as a server. The return value of both the `net.Dial()` function and the `net.Listen()` function is of the `net.Conn` type, which implements the `io.Reader` and `io.Writer` interfaces. The first parameter of both functions is the network type, but this is where their similarities end.

A TCP client

As you already know from the previous chapter, TCP's principal characteristic is that it is a reliable protocol. The TCP header of each packet includes the **source port** and **destination port** fields. These two fields, plus the source and destination IP addresses, are combined to uniquely identify every single TCP connection. The name of the TCP client that will be developed in this section is `TCPclient.go`, and it will be presented in four parts. The first part of `TCPclient.go` is shown in the following Go code:

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strings"
)
```

The second code segment of `TCPclient.go` is as follows:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide host:port.")
        return
    }

    CONNECT := arguments[1]
    c, err := net.Dial("tcp", CONNECT)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

The `net.Dial()` function is used to connect to the remote server. The first parameter of the `net.Dial()` function defines the network that will be used, while the second parameter defines the server address, which must also include the port number. Valid values for the first parameter are `tcp`, `tcp4` (IPv4-only), `tcp6` (IPv6-only), `udp`, `udp4` (IPv4-only), `udp6` (IPv6-only), `ip`, `ip4` (IPv4-only), `ip6` (IPv6-only), `unix` (UNIX sockets), `unixgram`, and `unixpacket`.

The third part of `TCPclient.go` contains the following code:

```
| for {
|     reader := bufio.NewReader(os.Stdin)
|     fmt.Print(">> ")
|     text, _ := reader.ReadString('\n')
|     fmt.Fprintf(c, text+"\n")
```

The preceding code is for getting user input, which can be verified by the use of the `os.Stdin` file for reading. Ignoring the `error` value returned by `reader.ReadString()` is not good practice, but it saves some space here. Certainly, you should never do that in production software.

The last part of `TCPclient.go` follows:

```
| message, _ := bufio.NewReader(c).ReadString('\n')
| fmt.Print("->: " + message)
| if strings.TrimSpace(string(text)) == "STOP" {
|     fmt.Println("TCP client exiting...")
|     return
| }
| }
```

For testing purposes, `TCPclient.go` will connect to a TCP server that is implemented using `netcat(1)`, and will create the following type of output:

```
$ go run TCPclient.go 8001
dial tcp: address 8001: missing port in address
$ go run TCPclient.go localhost:8001
>> Hello from TCPclient.go!
->: Hi from nc!
->: STOP
->:
TCP client exiting...
```

The output of the first command illustrates what will happen if you do not include a host name in the command-line arguments of `TCPclient.go`. The output of the `netcat(1)` TCP server, which should be executed first, is as follows:

```
$ nc -l 127.0.0.1 8001
Hello from TCPclient.go!
Hi from nc!
STOP
```



Notice that a client for a given protocol, such as TCP and UDP, can be reasonably generic in nature, which means that it can talk to many kinds of servers that support its protocol. As you will soon see, this is not the case with server applications, which must implement a prearranged functionality using a prearranged protocol.

A slightly different version of the TCP client

Go offers a different family of functions that also allows you to develop TCP clients and servers. In this section, you will learn how to program a TCP client using these functions.

The name of the TCP client will be `otherTCPclient.go`, and it is going to be presented in four parts. The first code segment from `otherTCPclient.go` is as follows:

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strings"
)
```

The second code portion from `otherTCPclient.go` contains the following code:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a server:port string!")
        return
    }

    CONNECT := arguments[1]

    tcpAddr, err := net.ResolveTCPAddr("tcp4", CONNECT)
    if err != nil {
        fmt.Println("ResolveTCPAddr:", err.Error())
        return
    }
}
```

The `net.ResolveTCPAddr()` function returns the address of a TCP endpoint (`type TCPAddr`) and can only be used for TCP networks.

The third part of `otherTCPclient.go` contains the following code:

```

    conn, err := net.DialTCP("tcp4", nil, tcpAddr)
    if err != nil {
        fmt.Println("DialTCP:", err.Error())
        return
    }
}

```

The `net.DialTCP()` function is equivalent to `net.Dial()` but it is for TCP networks only.

The remaining code of `otherTCPclient.go` is as follows:

```

for {
    reader := bufio.NewReader(os.Stdin)
    fmt.Print(">> ")
    text, _ := reader.ReadString('\n')
    fmt.Fprintf(conn, text+"\n")

    message, _ := bufio.NewReader(conn).ReadString('\n')
    fmt.Print("->: " + message)
    if strings.TrimSpace(string(text)) == "STOP" {
        fmt.Println("TCP client exiting...")
        conn.Close()
        return
    }
}
}

```

Executing `otherTCPclient.go` and interacting with a TCP server will generate the following type of output:

```

$ go run otherTCPclient.go localhost:8001
>> Hello from otherTCPclient.go!
->: Hi from netcat!
>> STOP
->:
TCP client exiting...

```

For this example, the TCP server is supported by the `netcat(1)` utility, which is executed as follows:

```

$ nc -l 127.0.0.1 8001
Hello from otherTCPclient.go!

Hi from netcat!
STOP

```

A TCP server

The TCP server that is going to be developed in this section will return the current date and time to the client in a single network packet. In practice, this means that after accepting a client connection, the server will get the time and date from the UNIX system and send that data back to the client.

The name of the utility is `TCPserver.go`, and it will be presented in four parts.

The first part of `TCPserver.go` is as follows:

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strings"
    "time"
)
```

The second code portion of `TCPserver.go` contains the following Go code:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide port number")
        return
    }

    PORT := ":" + arguments[1]
    l, err := net.Listen("tcp", PORT)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer l.Close()
```

The `net.Listen()` function listens for connections. If the second parameter does not contain an IP address but only a port number, `net.Listen()` will listen on all available IP addresses of the local system.

The third segment from `TCPserver.go` is as follows:

```

    c, err := l.Accept()
    if err != nil {
        fmt.Println(err)
        return
    }
}

```

The `Accept()` function waits for the next connection and returns a generic `conn` variable. The only thing that is wrong with this particular TCP server is that it can only serve the first TCP client that is going to connect to it because the `Accept()` call is outside the `for` loop that is coming next. The remaining Go code of `TCPserver.go` is as follows:

```

for {
    netData, err := bufio.NewReader(c).ReadString('\n')
    if err != nil {
        fmt.Println(err)
        return
    }
    if strings.TrimSpace(string(netData)) == "STOP" {
        fmt.Println("Exiting TCP server!")
        return
    }

    fmt.Print("-> ", string(netData))
    t := time.Now()
    myTime := t.Format(time.RFC3339) + "\n"
    c.Write([]byte(myTime))
}
}

```

Executing `TCPserver.go` and interacting with it using a TCP client application will generate the following type of output:

```

$ go run TCPserver.go 8001
-> HELLO
Exiting TCP server!

```

On the client side, you will see the following output:

```

$ nc 127.0.0.1 8001
HELLO
2019-05-18T22:50:31+03:00
STOP

```

If the `TCPserver.go` utility attempts to use a TCP port that is already in use by another UNIX process, you will get the following type of error message:

```

$ go run TCPserver.go 9000
listen tcp :9000: bind: address already in use

```

|

Finally, if the `TCPserver.go` utility attempts to use a TCP port in the 1-1024 range that requires root privileges on UNIX systems, you will get the following type of error message:

```
$ go run TCPserver.go 80
listen tcp :80: bind: permission denied
```

A slightly different version of the TCP server

In this section, you are going to see an alternative implementation of a TCP server in Go. This time, the TCP server implements the **Echo service**, which basically returns to the client, the data that the client sent. The program is called `otherTCPserver.go`, and it will be presented in four parts.

The first part of `otherTCPserver.go` is as follows:

```
package main

import (
    "fmt"
    "net"
    "os"
    "strings"
)
```

The second portion of `otherTCPserver.go` contains the following Go code:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a port number!")
        return
    }

    SERVER := "localhost" + ":" + arguments[1]

    s, err := net.ResolveTCPAddr("tcp", SERVER)
    if err != nil {
        fmt.Println(err)
        return
    }

    l, err := net.ListenTCP("tcp", s)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

The `net.ListenTCP()` function is equivalent to `net.Listen()` for TCP networks.

The third segment of `otherTCPserver.go` is as follows:

```
buffer := make([]byte, 1024)
conn, err := l.Accept()
if err != nil {
    fmt.Println(err)
}
```

```
|     return  
| }
```

The remaining code of `otherTCPserver.go` is as follows:

```
for {  
    n, err := conn.Read(buffer)  
    if err != nil {  
        fmt.Println(err)  
        return  
    }  
  
    if strings.TrimSpace(string(buffer[0:n])) == "STOP" {  
        fmt.Println("Exiting TCP server!")  
        conn.Close()  
        return  
    }  
  
    fmt.Print("> ", string(buffer[0:n-1]))  
    _, err = conn.Write(buffer)  
    if err != nil {  
        fmt.Println(err)  
        return  
    }  
}  
}
```

Executing `otherTCPserver.go` and using a client for interacting with it will generate the following type of output:

```
$ go run otherTCPserver.go 8001  
> 1  
> 2  
> Hello!  
> Exiting TCP server!
```

On the client side, which in this case is going to be `otherTCPclient.go`, you will see the following type of output:

```
$ go run otherTCPclient.go localhost:8001  
>> 1  
->: 1  
>> 2  
->: 2  
>> Hello!  
->: Hello!  
>> ->:  
>> STOP  
->: TCP client exiting...
```

Finally, I will present a way of finding out the name of the process that listens to a given TCP or UDP port on a UNIX machine. So, if you want to discover which process uses TCP port number `8001`, you should execute the following command:

```
$ sudo lsof -n -i :8001
COMMAND      PID  USER    FD   TYPE             DEVICE SIZE/OFF NODE NAME
TCPserver  86775 mtsouk    3u  IPv6  0x98d55014e6c9360f      0t0    TCP *:vcom-tunnel (LISTEN)
```

A UDP client

If you know how to develop a TCP client, then you should find developing a UDP client much easier due to the simplicity of the UDP protocol.



The biggest difference between UDP and TCP is that UDP is not reliable by design. This also means that, in general, UDP is simpler than TCP because UDP does not need to keep the state of a UDP connection. Put simply, UDP is like "fire and forget," which in some cases is perfect.

The name of the utility presented for this topic is `UDPclient.go`, and it will be presented in four code segments. The first part of `UDPclient.go` is as follows:

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strings"
)
```

The second segment of `UDPclient.go` is as follows:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a host:port string")
        return
    }
    CONNECT := arguments[1]

    s, err := net.ResolveUDPAddr("udp4", CONNECT)
    c, err := net.DialUDP("udp4", nil, s)

    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Printf("The UDP server is %s\n", c.RemoteAddr().String())
    defer c.Close()
```

The `net.ResolveUDPAddr()` function returns an address of a UDP endpoint as defined by its second parameter. The first parameter (`udp4`) specifies that the program will support the IPv4 protocol only.

The `net.DialUDP()` function used is like `net.Dial()` for UDP networks.

The third segment of `UDPclient.go` contains the following code:

```
for {
    reader := bufio.NewReader(os.Stdin)
    fmt.Print(">> ")
    text, _ := reader.ReadString('\n')
    data := []byte(text + "\n")
    _, err = c.Write(data)
    if strings.TrimSpace(string(data)) == "STOP" {
        fmt.Println("Exiting UDP client!")
        return
    }
}
```

The preceding code requires that the user types some text, which is sent to the UDP server. The user text is read from UNIX standard input using `bufio.NewReader(os.Stdin)`. The `Write(data)` method sends the data over the UDP network connection.

The rest of the Go code is as follows:

```
if err != nil {
    fmt.Println(err)
    return
}

buffer := make([]byte, 1024)
n, _, err := c.ReadFromUDP(buffer)
if err != nil {
    fmt.Println(err)
    return
}
fmt.Printf("Reply: %s\n", string(buffer[0:n]))
}
```

Once the client data is sent, you must wait for the data that the UDP server has to send, which is read using `ReadFromUDP()`. Executing `UDPclient.go` and interacting with the `netcat(1)` utility that acts as a UDP server will generate the following type of output:

```
$ go run UDPclient.go localhost:8001
The UDP server is 127.0.0.1:8001
>> Hello!
Reply: Hi there!

>> Have to leave - bye!
Reply: OK.
```

```
|>> STOP  
|Exiting UDP client!
```

On the UDP server side, the output will be as follows:

```
$ nc -v -u -l 127.0.0.1 8001  
Hello!  
  
Hi there!  
Have to leave - bye!  
  
OK.  
STOP  
  
^C
```

The reason for typing *Ctrl+C* in order to terminate `nc(1)` is that `nc(1)` does not have any code that tells it to terminate when it receives the `STOP` string as input.

Developing a UDP server

The purpose of the UDP server that is going to be developed in this section is returning random numbers between 1 and 1,000 to its UDP clients. The name of the program is `UDPserver.go`, and it is presented in four segments.

The first part of `UDPserver.go` is as follows:

```
package main

import (
    "fmt"
    "math/rand"
    "net"
    "os"
    "strconv"
    "strings"
    "time"
)

func random(min, max int) int {
    return rand.Intn(max-min) + min
}
```

The second segment of `UDPserver.go` is as follows:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a port number!")
        return
    }
    PORT := ":" + arguments[1]

    s, err := net.ResolveUDPAddr("udp4", PORT)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

The third segment of `UDPserver.go` contains the following code:

```
connection, err := net.ListenUDP("udp4", s)
if err != nil {
    fmt.Println(err)
    return
}

defer connection.Close()
```

```
    buffer := make([]byte, 1024)
    rand.Seed(time.Now().Unix())
```

The `net.ListenUDP()` function acts like `net.ListenTCP()` for UDP networks.

The remaining Go code of `UDPServer.go` is as follows:

```
for {
    n, addr, err := connection.ReadFromUDP(buffer)
    fmt.Print("-> ", string(buffer[0:n-1]))

    if strings.TrimSpace(string(buffer[0:n])) == "STOP" {
        fmt.Println("Exiting UDP server!")
        return
    }

    data := []byte(strconv.Itoa(random(1, 1001)))
    fmt.Printf("data: %s\n", string(data))
    _, err = connection.WriteToUDP(data, addr)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

The `ReadFromUDP()` function allows you to read data from a UDP connection using a buffer, which, as expected, is a **byte slice**.

Executing `UDPServer.go` and connecting to it with `UDPclient.go` will generate the following type of output:

```
$ go run UDPServer.go 8001
-> Hello!
data: 156
-> Another random number please :)
data: 944
-> Leaving...
data: 491
-> STOP
Exiting UDP server!
On the client side, the output will be as follows:
$ go run UDPclient.go localhost:8001
The UDP server is 127.0.0.1:8001
>> Hello!
Reply: 156
>> Another random number please :)
Reply: 944
>> Leaving...
Reply: 491
>> STOP
Exiting UDP client!
```

A concurrent TCP server

In this section, you will learn how to develop a **concurrent TCP server** using goroutines. For each incoming connection to the TCP server, the program will start a new goroutine to handle that request. This allows it to accept more requests, which means that a concurrent TCP server can serve multiple clients simultaneously.

The job of the TCP concurrent server is to accept a positive integer and return a natural number from the Fibonacci sequence. If there is an error in the input, the return value will be `-1`. As the calculation of numbers in the Fibonacci sequence can be slow, we are going to use an algorithm that was first presented in [Chapter 11, *Code Testing, Optimization, and Profiling*](#), and was included in `benchmarkMe.go`. Additionally, this time, the algorithm used will be explained a little more.

The name of the program is `fibotCP.go`, and its code is presented in five parts. As it is considered good practice to be able to define the port number of a web service as a command-line parameter, `fibotCP.go` will do exactly that.

The first part of `fibotCP.go` contains the following Go code:

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strconv"
    "strings"
    "time"
)
```

The second code portion of `fibotCP.go` contains the following Go code:

```
func f(n int) int {
    fn := make(map[int]int)
    for i := 0; i <= n; i++ {
        var f int
        if i <= 2 {
```

```

        f = 1
    } else {
        f = fn[i-1] + fn[i-2]
    }
    fn[i] = f
}
return fn[n]
}

```

In the preceding code, you can see the implementation of the `f()` function, which generates natural numbers that belong to the Fibonacci sequence. The algorithm used is difficult to understand at first, but it is very efficient and therefore fast.

Firstly, the `f()` function uses a Go map named `fn`, which is pretty unusual when calculating numbers of the Fibonacci sequence. Secondly, the `f()` function uses a `for` loop, which is also fairly unusual. Finally, the `f()` function does not use recursion, which is the main reason for the speed of its operation.

The idea behind the algorithm used in `f()`, which uses a **dynamic programming** technique, is that whenever a Fibonacci number is computed, it is put into the `fn` map so that it will not be computed again. This simple idea saves a lot of time, especially when large Fibonacci numbers need to be calculated, because then you do not have to calculate the same Fibonacci number multiple times.

The third code segment of `fibotCP.go` is as follows:

```

func handleConnection(c net.Conn) {
    for {
        netData, err := bufio.NewReader(c).ReadString('\n')
        if err != nil {
            fmt.Println(err)
            os.Exit(100)
        }

        temp := strings.TrimSpace(string(netData))
        if temp == "STOP" {
            break
        }

        fibo := "-1\n"
        n, err := strconv.Atoi(temp)
        if err == nil {
            fibo = strconv.Itoa(f(n)) + "\n"
        }
        c.Write([]byte(string(fibo)))
    }
}

```

```

        }
        time.Sleep(5 * time.Second)
        c.Close()
    }
}

```

The `handleConnection()` function deals with each client of the concurrent TCP server.

The fourth part of `fibotcp.go` is as follows:

```

func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a port number!")
        return
    }

    PORT := ":" + arguments[1]
    l, err := net.Listen("tcp4", PORT)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer l.Close()
}

```

The remaining Go code of `fibotcp.go` is as follows:

```

    for {
        c, err := l.Accept()
        if err != nil {
            fmt.Println(err)
            return
        }
        go handleConnection(c)
    }
}

```

The concurrency of the program is implemented by the `go handleConnection(c)` statement, which commences a new goroutine each time a new TCP client comes from the internet or a local network. The goroutine is executed concurrently, which gives the server the opportunity to serve even more clients.

Executing `fibotcp.go` and interacting with it using both `netcat(1)` and `TCPclient.go` on two different terminals will generate the following type of output:

```

$ go run fibotcp.go 9000
n: 10

```

```
fibo: 55
n: 0
fibo: 1
n: -1
fibo: 0
n: 100
fibo: 3736710778780434371
n: 12
fibo: 144
n: 12
fibo: 144
```

The output will be as follows on the `TCPclient.go` side:

```
$ go run TCPclient.go localhost:9000
>> 12
->: 144
>> a
->: -1
>> STOP
->: TCP client exiting...
```

On the `netcat(1)` side, the output will be as follows:

```
$ nc localhost 9000
10
55
0
1
-1
0
100
3736710778780434371
ads
-1
STOP
```

When you send the `STOP` string to the server process, the goroutine that serves that particular TCP client will terminate, which will cause the connection to end.

Finally, the impressive thing here is that both clients are served at the same time, which can be verified by the output of the following command:

```
$ netstat -anp TCP | grep 9000
tcp4      0 0 127.0.0.1.9000  127.0.0.1.57309    ESTABLISHED
tcp4      0 0 127.0.0.1.57309 127.0.0.1.9000    ESTABLISHED
tcp4      0 0 127.0.0.1.9000  127.0.0.1.57305    ESTABLISHED
tcp4      0 0 127.0.0.1.57305 127.0.0.1.9000    ESTABLISHED
```

tcp4	0 0 * . 9000	*.*	LISTEN
------	--------------	-----	--------

The last line of the output of the preceding command tells us that there is a process

that listens to port 9000, which means that you can still connect to port 9000.

The first

two lines of the output say that there is a client that uses port 57309 to talk to the server process. The third and fourth lines of the preceding output verify that there is another client that communicates with the server that listens to port 9000. That client uses TCP port 57305.

A handy concurrent TCP server

Although the concurrent TCP server from the previous section works fine, it does not serve a practical application. Therefore, in this section, you will learn how to convert the `keyValue.go` application from [Chapter 4, *The Uses of Composite Types*](#), into a fully-featured concurrent TCP application.

What we are going to do is create our own kind of TCP protocol in order to be able to interact with the key-value store from a network. You will need a keyword for each one of the functions of the key-value store. For reasons of simplicity, each keyword will be followed by the relevant data. The result of most commands will be a success or failure message.



Designing your own TCP- or UDP-based protocols is not an easy job. This means that you should be particularly specific and careful when designing a new protocol.

The name of the utility shown in this topic is `kvTCP.go`, and it is presented in six parts.

The first part of `kvTCP.go` is as follows:

```
package main

import (
    "bufio"
    "encoding/gob"
    "fmt"
    "net"
    "os"
    "strings"
)

type myElement struct {
    Name      string
    Surname   string
    Id        string
}

const welcome = "Welcome to the Key Value store!\n"

var DATA = make(map[string]myElement)
var DATAFILE = "/tmp/dataFile.gob"
```

The second part of `kvTCP.go` contains the following Go code:

```
func handleConnection(c net.Conn) {
    c.Write([]byte(welcome))
    for {
        netData, err := bufio.NewReader(c).ReadString('\n')
        if err != nil {
            fmt.Println(err)
            return
        }

        command := strings.TrimSpace(string(netData))
        tokens := strings.Fields(command)
        switch len(tokens) {
        case 0:
            continue
        case 1:
            tokens = append(tokens, "")
            tokens = append(tokens, "")
            tokens = append(tokens, "")
            tokens = append(tokens, "")
        case 2:
            tokens = append(tokens, "")
            tokens = append(tokens, "")
            tokens = append(tokens, "")
        case 3:
            tokens = append(tokens, "")
            tokens = append(tokens, "")
        case 4:
            tokens = append(tokens, "")

        switch tokens[0] {
        case "STOP":
            err = save()
            if err != nil {
                fmt.Println(err)
            }
            c.Close()
            return
        case "PRINT":
            PRINT(c)
        case "DELETE":
            if !DELETE(tokens[1]) {
                netData := "Delete operation failed!\n"
                c.Write([]byte(netData))
            } else {
                netData := "Delete operation successful!\n"
                c.Write([]byte(netData))
            }
        case "ADD":
            n := myElement{tokens[2], tokens[3], tokens[4]}
            if !ADD(tokens[1], n) {
                netData := "Add operation failed!\n"
                c.Write([]byte(netData))
            } else {
                netData := "Add operation successful!\n"
                c.Write([]byte(netData))
            }
        err = save()
```

```

        if err != nil {
            fmt.Println(err)
        }
    case "LOOKUP":
        n := LOOKUP(tokens[1])
        if n != nil {
            netData := fmt.Sprintf("%v\n", *n)
            c.Write([]byte(netData))
        } else {
            netData := "Did not find key!\n"
            c.Write([]byte(netData))
        }
    case "CHANGE":
        n := myElement(tokens[2], tokens[3], tokens[4])
        if !CHANGE(tokens[1], n) {
            netData := "Update operation failed!\n"
            c.Write([]byte(netData))
        } else {
            netData := "Update operation successful!\n"
            c.Write([]byte(netData))
        }
        err = save()
        if err != nil {
            fmt.Println(err)
        }
    default:
        netData := "Unknown command - please try again!\n"
        c.Write([]byte(netData))
    }
}
}

```

The `handleConnection()` function communicates with each TCP client and interprets the client input.

The third segment of `kvTCP.go` contains the following Go code:

```

func save() error {
    fmt.Println("Saving", DATAFILE)
    err := os.Remove(DATAFILE)
    if err != nil {
        fmt.Println(err)
    }

    saveTo, err := os.Create(DATAFILE)
    if err != nil {
        fmt.Println("Cannot create", DATAFILE)
        return err
    }
    defer saveTo.Close()

    encoder := gob.NewEncoder(saveTo)
    err = encoder.Encode(DATA)
    if err != nil {
        fmt.Println("Cannot save to", DATAFILE)
        return err
    }
    return nil
}

```

```

    }

func load() error {
    fmt.Println("Loading", DATAFILE)
    loadFrom, err := os.Open(DATAFILE)
    defer loadFrom.Close()
    if err != nil {
        fmt.Println("Empty key/value store!")
        return err
    }

    decoder := gob.NewDecoder(loadFrom)
    decoder.Decode(&DATA)
    return nil
}

```

The fourth segment of `kvTCP.go` is as follows:

```

func ADD(k string, n myElement) bool {
    if k == "" {
        return false
    }

    if LOOKUP(k) == nil {
        DATA[k] = n
        return true
    }
    return false
}

func DELETE(k string) bool {
    if LOOKUP(k) != nil {
        delete(DATA, k)
        return true
    }
    return false
}

func LOOKUP(k string) *myElement {
    _, ok := DATA[k]
    if ok {
        n := DATA[k]
        return &n
    } else {
        return nil
    }
}

func CHANGE(k string, n myElement) bool {
    DATA[k] = n
    return true
}

```

The implementation of the preceding functions is the same as in `keyValue.go`. None of them talk directly to a TCP client.

The fifth part of `kvTCP.go` contains the following Go code:

```

func PRINT(c net.Conn) {
    for k, d := range DATA {
        netData := fmt.Sprintf("key: %s value: %v\n", k, d)
        c.Write([]byte(netData))
    }
}

```

The `PRINT()` function sends data to a TCP client directly, one line at a time.

The remaining Go code of the program is as follows:

```

func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a port number!")
        return
    }

    PORT := ":" + arguments[1]
    l, err := net.Listen("tcp", PORT)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer l.Close()

    err = load()
    if err != nil {
        fmt.Println(err)
    }

    for {
        c, err := l.Accept()
        if err != nil {
            fmt.Println(err)
            os.Exit(100)
        }
        go handleConnection(c)
    }
}

```

Executing `kvTCP.go` will generate the following type of output:

```

$ go run kvTCP.go 9000
Loading /tmp/dataFile.gob
Empty key/value store!
open /tmp/dataFile.gob: no such file or directory
Saving /tmp/dataFile.gob
remove /tmp/dataFile.gob: no such file or directory
Saving /tmp/dataFile.gob
Saving /tmp/dataFile.gob

```

For the purposes of this section, the `netcat(1)` utility will act as the TCP client of `kvTCP.go`:

```
$ nc localhost 9000
Welcome to the Key Value store!
PRINT
LOOKUP 1
Did not find key!
ADD 1 2 3 4
Add operation successful!
LOOKUP 1
{2 3 4}
ADD 4 -1 -2 -3
Add operation successful!
PRINT
key: 1 value: {2 3 4}
key: 4 value: {-1 -2 -3}
STOP
```

`kvTCP.go` is a concurrent application that uses goroutines and can serve multiple TCP clients simultaneously. However, all of these TCP clients will share the same data.

Creating a Docker image for a Go TCP/IP server

In this section, you will learn how to put `kvTCP.go` into a Docker image and use it from there, which is a pretty convenient way of using a TCP/IP application because a Docker image can be easily transferred to other machines or deployed in **Kubernetes**.

As you might expect, everything will begin with a `Dockerfile`, which will have the following contents:

```
FROM golang:latest
RUN mkdir /files
COPY kvTCP.go /files
WORKDIR /files

RUN go build -o /files/kvTCP kvTCP.go
ENTRYPOINT ["/files/kvTCP","80"]
```

After that, you will need to build the Docker image, as follows:

```
$ docker build -t kvtcp:latest .
Sending build context to Docker daemon 6.656kB
Step 1/6 : FROM golang:latest
--> 7ced090ee82e
Step 2/6 : RUN mkdir /files
--> Running in bbbbada6271f
Removing intermediate container bbbbada6271f
--> 5b0a621eee29
Step 3/6 : COPY kvTCP.go /files
--> 4aab441b14c2
Step 4/6 : WORKDIR /files
--> Running in 7185606bed2e
Removing intermediate container 7185606bed2e
--> 744e9800fdbba
Step 5/6 : RUN go build -o /files/kvTCP kvTCP.go
--> Running in f44fcbe8951b
Removing intermediate container f44fcbe8951b
--> a8d00c7ead13
Step 6/6 : ENTRYPOINT ["/files/kvTCP","80"]
--> Running in ec3227170e09
Removing intermediate container ec3227170e09
--> b65ba728849a
Successfully built b65ba728849a
Successfully tagged kvtcp:latest
```

Executing `docker images` will verify that you have created the desired Docker image:

```
$ docker images
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
kvtcp               latest   b65ba728849a  26 minutes ago  777MB
landoop/kafka-lenses-dev  latest   289093ceee7b  2 days ago    1.37GB
golang               latest   7ced090ee82e  9 days ago    774MB
```

Then, you will need to execute that Docker image. This time, you will need to specify some extra parameters to expose the port numbers that you want to make available to the outer world:

```
$ docker run -d -p 5801:80 kvtcp:latest
af939992a25cb5ccf405b1b97b9c813fb4cb3f3a2e5b13942db637709c8cee2
```

The last command exposes port number `80` from the Docker image to the local machine using port number `5801` on the local machine.

From now on, you will be able to use the TCP server that is located in the Docker image, as follows:

```
$ nc 127.0.0.1 5801
Welcome to the Key Value store!
```

Notice that, until you save your data in a location outside of the Docker image, your data will be lost when you terminate that Docker image.

This capability of Docker gives you the opportunity to use any port number that you want to access the TCP server in the Docker image. However, if you try to map port number `80` again, you will get the following kind of error message:

```
$ docker run -d -p 5801:80 kvtcp:latest
709d44be8668284b101d7dfc253938d13e6797d812821838aa5ab18ea48527ec
docker: Error response from daemon: driver failed programming external connectivity on endpoint eager_nobel (fa9d43d3c12973457)
```

As the error message says, the port is already allocated and therefore cannot be used again. However, there is a workaround to this limitation, which can be illustrated by the following command:

```
$ docker run -d -p 5801:80 -p 2000:80 kvtcp:latest
5cbb17c5bbf720eaa5ce0f1e11cc73dbe5bef3cc925b7936ae1b97e245d54ae8
```

The previous command maps port number `80` from the Docker image to two external TCP ports (`5801` and `2000`) in a single `docker run` command. Notice that there is a single `kvTCP` server running and that there is a single copy of the data even if that data can be accessed using multiple ports.

This can also be verified by the output of the `docker ps` command:

```
$ docker ps
CONTAINER ID        IMAGE           COMMAND          CREATED         STATUS          PORTS
5cbb17c5bbf7        kvtcp:latest   "/files/kvTCP 80"   3 seconds ago   Up 2 seconds   0.0.0.0:2000->80/tcp, 0.0.0.0:48527->80/tcp
```

Remote Procedure Call (RPC)

RPC is a client-server mechanism for interprocess communication that uses TCP/IP. Both the RPC client and the RPC server to be developed will use the following package, which is named `sharedRPC.go`:

```
package sharedRPC

type MyFloats struct {
    A1, A2 float64
}

type MyInterface interface {
    Multiply(arguments *MyFloats, reply *float64) error
    Power(arguments *MyFloats, reply *float64) error
}
```

The `sharedRPC` package defines one interface called `MyInterface` and one structure called `MyFloats`, which will be used by both the client and the server. However, only the RPC server will need to implement that interface.

After that, you will need to install the `sharedRPC.go` package by executing the following commands:

```
$ mkdir -p ~/go/src/sharedRPC
$ cp sharedRPC.go ~/go/src/sharedRPC/
$ go install sharedRPC
```

The RPC client

In this section, you are going to see the Go code of the RPC client, which will be saved as `RPCclient.go` and presented in four parts.

The first part of `RPCclient.go` is as follows:

```
package main

import (
    "fmt"
    "net/rpc"
    "os"
    "sharedRPC"
)
```

The second segment of `RPCclient.go` contains the following Go code:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a host:port string!")
        return
    }

    CONNECT := arguments[1]
    c, err := rpc.Dial("tcp", CONNECT)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

Observe the use of the `rpc.Dial()` function for connecting to the RPC server instead of the `net.Dial()` function, even though the RCP server uses TCP.

The third segment of `RPCclient.go` is as follows:

```
args := sharedRPC.MyFloats{16, -0.5}
var reply float64

err = c.Call("MyInterface.Multiply", args, &reply)
if err != nil {
    fmt.Println(err)
    return
}
fmt.Printf("Reply (Multiply): %f\n", reply)
```

What is being exchanged between the RPC client and the RPC server with the help of the `Call()` function are function names, their arguments, and the results of the function calls, as the RPC client knows nothing about the actual implementation of the functions.

The remaining Go code of `RPCclient.go` is as follows:

```
    err = c.Call("MyInterface.Power", args, &reply)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Printf("Reply (Power): %f\n", reply)
}
```

If you try to execute `RPCclient.go` without having an RPC server running, you will get the following type of error message:

```
$ go run RPCclient.go localhost:1234
dial tcp [::1]:1234: connect: connection refused
```

The RPC server

The RPC server will be saved as `RPCserver.go`, and it is presented in five parts.

The first part of `RPCserver.go` is as follows:

```
package main

import (
    "fmt"
    "math"
    "net"
    "net/rpc"
    "os"
    "sharedRPC"
)
```

The second segment of `RPCserver.go` contains the following Go code:

```
type MyInterface struct{}

func Power(x, y float64) float64 {
    return math.Pow(x, y)
}

func (t *MyInterface) Multiply(arguments *sharedRPC.MyFloats, reply *float64) error {
    *reply = arguments.A1 * arguments.A2
    return nil
}

func (t *MyInterface) Power(arguments *sharedRPC.MyFloats, reply *float64) error {
    *reply = Power(arguments.A1, arguments.A2)
    return nil
}
```

In the preceding Go code, the RPC server implements the desired interface, as well as a helper function named `Power()`.

The third segment of `RPCserver.go` is as follows:

```
func main() {
    PORT := ":1234"
    arguments := os.Args
    if len(arguments) != 1 {
        PORT = ":" + arguments[1]
    }
}
```

The fourth part of `RPCserver.go` contains the following code:

```

myInterface := new(MyInterface)
rpc.Register(myInterface)
t, err := net.ResolveTCPAddr("tcp4", PORT)
if err != nil {
    fmt.Println(err)
    return
}
l, err := net.ListenTCP("tcp4", t)
if err != nil {
    fmt.Println(err)
    return
}

```

What makes this program an RPC server is the use of the `rpc.Register()` function. However, as the RPC server uses TCP, it still needs to make function calls to `net.ResolveTCPAddr()` and `net.ListenTCP()`.

The rest of the Go code of `RPCserver.go` is as follows:

```

for {
    c, err := l.Accept()
    if err != nil {
        continue
    }
    fmt.Printf("%s\n", c.RemoteAddr())
    rpc.ServeConn(c)
}

```

The `RemoteAddr()` function returns the IP address and the port number used for communicating with the RPC client. The `rpc.ServeConn()` function serves the RPC client.

Executing `RPCserver.go` and waiting for `RPCclient.go` will create the following type of output:

```
$ go run RPCserver.go
127.0.0.1:52289
```

Executing `RPCclient.go` will create the following type of output:

```
$ go run RPCclient.go localhost:1234
Reply (Multiply): -8.000000
Reply (Power): 0.250000
```

Doing low-level network programming

Although the `http.Transport` structure allows you to modify the various low-level parameters of a network connection, you can write Go code that permits you to read the raw data of network packets.

There are two tricky points here. Firstly, network packets come in binary format, which requires you to look for specific kinds of network packets and not just any type of network packet. Put simply, when reading network packets, you should specify the protocol or protocols that you are going to support in your applications in advance. Secondly, in order to send a network packet, you will have to construct it on your own.

The next utility to be shown is called `lowLevel.go`, and it will be presented in three parts. Notice that `lowLevel.go` captures **Internet Control Message Protocol (ICMP)** packets, which use the IPv4 protocol and print their contents. Also, note that working with raw network data requires root privileges for security reasons.

The first segment of `lowLevel.go` is as follows:

```
package main  
import (  
    "fmt"  
    "net"  
)
```

The second part of `lowLevel.go` contains the following Go code:

```
func main() {  
    netaddr, err := net.ResolveIPAddr("ip4", "127.0.0.1")  
    if err != nil {  
        fmt.Println(err)  
        return  
    }  
    conn, err := net.ListenIP("ip4:icmp", netaddr)  
    if err != nil {  
        fmt.Println(err)  
        return  
    }
```

The ICMP protocol is specified in the second part of the first parameter (`ip4:icmp`) of `net.ListenIP()`. Moreover, the `ip4` part tells the utility to capture IPv4 traffic only.

The remaining part of `lowLevel.go` contains the following Go code:

```
    buffer := make([]byte, 1024)  
    n, _, err := conn.ReadFrom(buffer)  
    if err != nil {  
        fmt.Println(err)  
        return  
    }  
    fmt.Printf("% X\n", buffer[0:n])  
}
```

The preceding code tells `lowLevel.go` to read just a single network packet because there is no `for` loop.

The ICMP protocol is used by the `ping(1)` and `traceroute(1)` utilities, so one way to create ICMP traffic is to use one of these two tools. The network traffic will be generated using the following commands on both UNIX machines while `lowLevel.go` is already running:

```
$ ping -c 5 localhost  
PING localhost (127.0.0.1): 56 data bytes  
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.037 ms  
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.038 ms  
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.117 ms  
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.052 ms  
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.049 ms  
  
--- localhost ping statistics ---  
5 packets transmitted, 5 packets received, 0.0% packet loss  
round-trip min/avg/max/stddev = 0.037/0.059/0.117/0.030 ms
```

```
$ traceroute localhost
traceroute to localhost (127.0.0.1), 64 hops max, 52 byte packets
 1  localhost (127.0.0.1)  0.255 ms  0.048 ms  0.067 ms
```

Executing `lowLevel.go` on a macOS Mojave machine with root privileges will produce the following type of output:

```
$ sudo go run lowLevel.go
03 03 CD DA 00 00 00 00 45 00 34 00 B4 0F 00 00 01 11 00 00 7F 00 00 01 7F 00 00 01 B4 0E 82 9B 00 20 00 00
$ sudo go run lowLevel.go
00 00 0B 3B 20 34 00 00 5A CB 5C 15 00 04 32 A9 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 :
```

The first output example is generated by the `ping(1)` command, whereas the second output example is generated by the `traceroute(1)` command.

Running `lowLevel.go` on a Debian Linux machine will generate the following type of output:

```
$ uname -a
Linux mail 4.14.12-x86_64-linode92 #1 SMP Fri Jan 5 15:34:44 UTC 2018 x86_64 GNU/Linux
# go run lowLevel.go
08 00 61 DD 3F BA 00 01 9A 5D CB 5A 00 00 00 00 26 DC 0B 00 00 00 00 00 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 :
# go run lowLevel.go
03 03 BB B8 00 00 00 00 45 00 00 3C CD 8D 00 00 01 11 EE 21 7F 00 00 01 7F 00 00 01 CB 40 82 9A 00 28 FE 3B 40 41 42 43 44 45 :
```

The output of the `uname(1)` command prints useful information about the Linux system. Note that, on modern Linux machines, you should execute `ping(1)` with the `-4` flag in order to tell it to use the IPv4 protocol.

Grabbing raw ICMP network data

In this subsection, you will learn how to use the `syscall` package to capture raw ICMP network data and `syscall.SetsockoptInt()` in order to set the options of a socket.

Keep in mind that sending raw ICMP data is much more difficult, as you will have to construct the raw network packets on your own. The name of the utility is `syscallNet.go`, and it will be shown in four parts.

The first part of `syscallNet.go` is as follows:

```
package main

import (
    "fmt"
    "os"
    "syscall"
)
```

The second segment of `syscallNet.go` contains the following Go code:

```
func main() {
    fd, err := syscall.Socket(syscall.AF_INET, syscall.SOCK_RAW, syscall.IPPROTO_ICMP)
    if err != nil {
        fmt.Println("Error in syscall.Socket:", err)
        return
    }

    f := os.NewFile(uintptr(fd), "captureICMP")
    if f == nil {
        fmt.Println("Error in os.NewFile:", err)
        return
    }
}
```

The `syscall.AF_INET` parameter tells `syscall.Socket()` that you want to work with IPv4. The `syscall.SOCK_RAW` parameter is what makes the generated socket a raw socket. The last parameter, which is `syscall.IPPROTO_ICMP`, tells `syscall.Socket()` that you are interested in ICMP traffic only.

The third part of `syscallNet.go` is as follows:

```
    err = syscall.SetsockoptInt(fd, syscall.SOL_SOCKET, syscall.SO_RCVBUF, 256)
    if err != nil {
        fmt.Println("Error in syscall.Socket:", err)
        return
    }
}
```

The call to `syscall.SetsockoptInt()` sets the size of the receive buffer of the socket to 256. The `syscall.SOL_SOCKET` parameter is for stating that you want to work on the socket layer level.

The remaining Go code of `syscallNet.go` is as follows:

```
    for {
        buf := make([]byte, 1024)
        numRead, err := f.Read(buf)
        if err != nil {
            fmt.Println(err)
        }
        fmt.Printf("% X\n", buf[:numRead])
    }
}
```

Due to the `for` loop, `syscallNet.go` will keep capturing ICMP network packets until you terminate it manually.

Executing `syscallNet.go` on a macOS High Sierra machine will produce the following type of output:

```
$ sudo go run syscallNet.go
45 00 40 00 BC B6 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 00 00 3F 36 71 45 00 00 5A CB 6A 90 00 0B 9F 1A 08 09 0A 0B 0C 0D (
45 00 40 00 62 FB 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 00 00 31 EF 71 45 00 01 5A CB 6A 91 00 0B AC 5F 08 09 0A 0B 0C 0D (
45 00 40 00 9A 5F 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 00 00 1D D6 71 45 00 02 5A CB 6A 92 00 0B C0 76 08 09 0A 0B 0C 0D (
45 00 40 00 6E 0D 00 00 40 01 00 00 7F 00 00 01 7F 00 00 00 00 09 CF 71 45 00 03 5A CB 6A 93 00 0B D4 7B 08 09 0A 0B 0C 0D (
45 00 40 00 3A 07 00 00 40 01 00 00 7F 00 00 01 7F 00 00 00 FE 9C 71 45 00 04 5A CB 6A 94 00 0B DF AB 08 09 0A 0B 0C 0D (
```

```

45 00 24 00 45 55 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 03 03 AB 12 00 00 00 00 45 00 34 00 C5 73 00 00 01 11 00 00 7F 00 (
45 00 24 00 E8 1E 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 03 03 AB 10 00 00 00 00 45 00 34 00 C5 74 00 00 01 11 00 00 7F 00 (
45 00 24 00 2A 4B 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 03 03 AB 0E 00 00 00 00 45 00 34 00 C5 75 00 00 01 11 00 00 7F 00 (

```

Running `syscallNet.go` on a Debian Linux machine will generate the following type of output:

```

# go run syscallNet.go
45 00 00 54 7F E9 40 00 40 01 BC BD 7F 00 00 01 7F 00 00 01 08 00 6F 07 53 E3 00 01 FA 6A CB 5A 00 00 00 00 AA 7B 06 00 00 00 (
45 00 00 54 7F EA 00 00 40 01 FC BC 7F 00 00 01 7F 00 00 01 00 00 77 07 53 E3 00 01 FA 6A CB 5A 00 00 00 00 AA 7B 06 00 00 00 (
45 C0 00 44 68 54 00 00 34 01 8B 8E 67 DC 57 6D 4A C1 FD 03 0A 8F 27 00 00 00 00 45 00 00 28 40 4F 40 00 34 06 74 6A 6D 4A (
45 00 00 54 80 4E 40 00 40 01 BC 58 7F 00 00 01 7F 00 00 01 08 00 7E 01 53 E3 00 02 FB 6A CB 5A 00 00 00 00 9A 80 06 00 00 00 (
45 00 00 54 80 4F 00 00 40 01 FC 57 7F 00 00 01 7F 00 00 01 00 00 86 01 53 E3 00 02 FB 6A CB 5A 00 00 00 00 9A 80 06 00 00 00 (
45 00 00 54 80 9B 40 00 40 01 BC 0B 7F 00 00 01 7F 00 00 01 08 00 93 EC 53 E3 00 03 FC 6A CB 5A 00 00 00 00 83 94 06 00 00 00 00 (
45 00 00 54 80 9C 00 00 40 01 FC 0A 7F 00 00 01 7F 00 00 01 00 00 98 EC 53 E3 00 03 FC 6A CB 5A 00 00 00 00 83 94 06 00 00 00 00 (
45 C0 00 44 68 55 00 00 34 01 8B 8D 86 77 DC 57 6D 4A C1 FD 03 0A 8F 27 00 00 00 00 45 00 00 28 40 D1 40 00 34 06 73 E8 6D 4A (
45 00 00 54 80 F8 40 00 40 01 BB AE 7F 00 00 01 7F 00 00 01 08 00 F2 E7 53 E3 00 04 FD 6A CB 5A 00 00 00 00 23 98 06 00 00 00 00 (
45 00 00 54 80 F9 00 00 40 01 FB AD 7F 00 00 01 7F 00 00 01 00 00 FA E7 53 E3 00 04 FD 6A CB 5A 00 00 00 00 23 98 06 00 00 00 00 (
45 00 00 54 82 0D 40 00 40 01 BA 99 7F 00 00 01 7F 00 00 01 08 00 4A 82 53 E3 00 05 FE 6A CB 5A 00 00 00 00 CA FC 06 00 00 00 00 (
45 00 00 54 82 0E 00 00 40 01 FA 98 7F 00 00 01 7F 00 00 01 00 00 52 82 53 E3 00 05 FE 6A CB 5A 00 00 00 00 CA FC 06 00 00 00 00 (
45 C0 00 44 68 56 00 00 34 01 8B 8C 86 77 DC 57 6D 4A C1 FD 03 0A 8F 27 00 00 00 00 45 00 00 28 41 74 40 00 34 06 73 45 6D 4A (
45 C0 00 44 68 57 00 00 34 01 8B 8B 86 77 DC 57 6D 4A C1 FD 03 0A 8F 27 00 00 00 00 45 00 00 28 44 27 40 00 33 06 71 92 6D 4A (
45 C0 00 58 94 B4 00 00 40 01 E7 2E 7F 00 00 01 7F 00 00 01 03 03 F1 DA 00 00 00 00 45 00 00 3C 85 E1 00 00 01 11 35 CE 7F 00 (
45 C0 00 58 94 B5 00 00 40 01 E7 2D 7F 00 00 01 7F 00 00 01 03 03 F9 EA 00 00 00 00 45 00 00 3C 85 E2 00 00 01 11 35 CD 7F 00 (
45 C0 00 58 94 B6 00 00 40 01 E7 2C 7F 00 00 01 7F 00 00 01 03 03 D2 EB 00 00 00 00 45 00 00 3C 85 E3 00 00 01 11 35 CC 7F 00 (
45 C0 00 58 94 B7 00 00 40 01 E7 2B 7F 00 00 01 7F 00 00 01 03 03 D6 AC 00 00 00 00 45 00 00 3C 85 E4 00 00 02 11 34 CB 7F 00 (
45 C0 00 58 94 B8 00 00 40 01 E7 2A 7F 00 00 01 7F 00 00 01 03 03 F1 B4 00 00 00 00 45 00 00 3C 85 E5 00 00 02 11 34 CA 7F 00 (
45 C0 00 58 94 B9 00 00 40 01 E7 29 7F 00 00 01 7F 00 00 01 03 03 CD 43 00 00 00 00 45 00 00 3C 85 E6 00 00 02 11 34 C9 7F 00 (
45 C0 00 58 94 BA 00 00 40 01 E7 28 7F 00 00 01 7F 00 00 01 03 03 9D 8F 00 00 00 00 45 00 00 3C 85 E7 00 00 03 11 33 C8 7F 00 (
45 C0 00 58 94 BB 00 00 40 01 E7 27 7F 00 00 01 7F 00 00 01 03 03 A3 13 00 00 00 00 45 00 00 3C 85 E8 00 00 03 11 33 C7 7F 00 (
45 C0 00 58 94 BC 00 00 40 01 E7 26 7F 00 00 01 7F 00 00 01 03 03 D4 66 00 00 00 00 45 00 00 3C 85 E9 00 00 03 11 33 C6 7F 00 (
45 C0 00 58 94 BD 00 00 40 01 E7 25 7F 00 00 01 7F 00 00 01 03 03 A8 8D 00 00 00 00 45 00 00 3C 85 EA 00 00 04 11 32 C5 7F 00 (
45 C0 00 58 94 BE 00 00 40 01 E7 24 7F 00 00 01 7F 00 00 01 03 03 F1 C6 00 00 00 00 45 00 00 3C 85 EB 00 00 04 11 32 C4 7F 00 (
45 C0 00 58 94 BF 00 00 40 01 E7 23 7F 00 00 01 7F 00 00 01 03 03 A3 FE 00 00 00 00 45 00 00 3C 85 EC 00 00 04 11 32 C3 7F 00 (
45 C0 00 58 94 C0 00 00 40 01 E7 22 7F 00 00 01 7F 00 00 01 03 03 B9 AA 00 00 00 00 45 00 00 3C 85 ED 00 00 05 11 31 C2 7F 00 (
45 C0 00 58 94 C1 00 00 40 01 E7 21 7F 00 00 01 7F 00 00 01 03 03 B3 B7 00 00 00 00 45 00 00 3C 85 EE 00 00 05 11 31 C1 7F 00 (
45 C0 00 58 94 C2 00 00 40 01 E7 20 7F 00 00 01 7F 00 00 01 03 03 F2 62 00 00 00 00 45 00 00 3C 85 EF 00 00 05 11 31 C0 7F 00 (
45 C0 00 58 94 C3 00 00 40 01 E7 1F 7F 00 00 01 7F 00 00 01 03 03 DD BE 00 00 00 00 45 00 00 3C 85 F0 00 00 06 11 30 BF 7F 00 (

```

Additional resources

Take a look at the following resources:

- Visit the documentation of the `net` standard Go package, which can be found at <https://golang.org/pkg/net/>. This is one of the biggest documentation pages found in the Go documentation.
- You can learn more about the `crypto/tls` package of the standard Go library at <https://golang.org/pkg/crypto/tls/>.
- Visit <https://golang.org/pkg/crypto/x509/> to learn more about the `crypto/x509` package.
- The ICMP protocol for IPv4 is defined in RFC 792. It can be found in many places, including <https://tools.ietf.org/html/rfc792>.
- **WebSocket** is a protocol for two-way communication between a client and a remote host. There is a WebSocket implementation for Go at <https://github.com/gorilla/websocket>. You can learn more about WebSocket at <http://www.rfc-editor.org/rfc/rfc6455.txt>.
- If you are really into network programming and you want to be able to work with raw TCP packets, you might find interesting and helpful information and tools in the `gopacket` library, which can be found at <https://github.com/google/gopacket>.
- The `raw` package, which is located at <https://github.com/mdlayher/raw>, allows you to read and write data at the device driver level for a network device.

Exercises

- Develop a **File Transfer Protocol (FTP)** client in Go.
- Next, try to develop an FTP server in Go. Is it more difficult to implement the FTP client or the FTP server? Why?
- Try to implement a Go version of the `nc(1)` utility. The secret when programming such fairly complex utilities is to start with a version that implements the basic functionality of the desired utility, before trying to support every possible option.
- Modify `TCPserver.go` so that it returns the date in one network packet and the time in another.
- Modify `TCPserver.go` so that it can serve multiple clients in a sequential way. Notice that this is not the same as being able to serve multiple requests concurrently. Put simply, use a `for` loop so that the `Accept()` call can be executed multiple times.
- TCP servers, such as `fibotcp.go`, tend to terminate when they receive a given signal, so add signal handling code to `fibotcp.go`, as you learned in [Chapter 8, *Telling a UNIX System What to Do*](#).
- Modify `kvTCP.go` so that the `save()` function is protected using a `sync.Mutex`. Is this required?
- Try to connect to `https.go` and `TLSserver.go` using your favorite web browser.
- Try to put `https.go` into a Docker image and use it from there.
- Develop your own small web server in Go using a plain TCP implementation instead of using the `http.ListenAndServe()` function.

Summary

This chapter talked about many interesting things including developing UDP and TCP clients and servers, which are applications that work over TCP/IP computer networks.

The next chapter, which will be the last, will talk about machine learning and Go. It will include topics such as regression, classification, anomaly detection, and neural networks.

Machine Learning in Go

The previous two chapters discussed topics related to network programming, TCP/IP, HTTPS, RPC, and the `net` package. This chapter will talk about **machine learning** in Go, including many interesting topics such as calculating statistical properties, classification, regression, clustering, anomaly detection, neural networks, outlier analysis, and working with Apache Kafka. However, as all these are huge topics that deserve a book on their own, this chapter will only scratch the surface and give you a quick introduction to them, as well as introduce you to some handy Go packages that can help you to do the job.

Notice that each machine learning technique has some theory behind it – knowing the theory, the parameters, and the limitations of the techniques you are trying to use is essential for the success of your work. Additionally, visualizing your data can be helpful from time to time, as it allows you to get a good sense of your data quickly.



If you are really into machine learning with Go, then I suggest that you begin by getting Machine Learning with Go by Daniel Whitenack (Packt Publishing, 2017). If you want to learn about the theory behind machine learning, then you can begin by reading An Introduction to Statistical Learning, by Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani (Springer, 2013), and The Elements of Statistical Learning, 2nd Edition, by Trevor Hastie, Robert Tibshirani, and Jerome Friedman (Springer, 2009).

Therefore, in this chapter, you will learn about the following:

- Calculating simple statistical properties
- Regression
- Classification
- Anomaly detection
- Clustering
- Neural networks
- Working with TensorFlow
- **Outlier analysis**
- Working with Apache Kafka

Calculating simple statistical properties

Statistics is an area of mathematics that deals with the collection, analysis, interpretation, organization, and presentation of data. The field of statistics is divided into two main areas: the area of descriptive statistics, which tries to describe an already existing group of values, and the area of inferential statistics, which tries to predict upcoming values based on the information found in the current set of values.

Statistical learning is a branch of applied statistics that is related to machine learning. **Machine learning**, which is closely related to computational statistics, is an area of computer science that tries to learn from data and make predictions about it without being specifically programmed to do so.



Statistical models try to interpret data as accurately as possible. However, the accuracy of a model might depend on external factors that might affect the data. So, you might have a weather forecasting model that could become totally inaccurate when there is a hurricane nearby.

In this section, you are going to learn how to calculate basic statistical properties such as the mean value, the minimum and the maximum values of your sample, the **median** value, and the **variance** of the sample. These values give you a good overview of your sample without going into too much detail. However, generic values that try to describe your sample can easily trick you by making you believe that you know your sample well without this being true.

All these statistical properties will be computed in `stats.go`, which will be presented in five parts. Each line of the input file contains a single number, which means that the input file is read line by line. Invalid input will be ignored without any warning messages.

Notice that input will be stored in a slice in order to use a separate function for calculating each property. Also, as you will see shortly, the values of the slice will be sorted before processing them.

The first part of `stats.go` is as follows:

```
package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "math"
    "os"
    "sort"
    "strconv"
    "strings"
)

func min(x []float64) float64 {
    return x[0]
}

func max(x []float64) float64 {
    return x[len(x)-1]
}
```

As the slice is sorted, it is easy to find the minimum and the maximum values in it. However, if the elements of the slice are not numeric, you might need to calculate the minimum and the maximum values in a different way that is related to your data.

The second part of `stats.go` contains the following Go code:

```
func meanValue(x []float64) float64 {
    sum := float64(0)
    for _, v := range x {
        sum = sum + v
    }
    return sum / float64(len(x))
}
```

This function calculates the mean value of your numeric data.

The third part of `stats.go` is as follows:

```
func medianValue(x []float64) float64 {
    length := len(x)
    if length%2 == 1 {
```

```

        // Odd
        return x[(length-1)/2]
    } else {
        // Even
        return (x[length/2] + x[(length/2)-1]) / 2
    }
    return 0
}

func variance(x []float64) float64 {
    mean := meanValue(x)
    sum := float64(0)
    for _, v := range x {
        sum = sum + (v-mean)*(v-mean)
    }
    return sum / float64(len(x))
}

```

In order to be able to calculate the median value of a set, the set needs to be sorted.

The fourth part of `stats.go` contains the following code:

```

func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: stats filename\n")
        return
    }

    data := make([]float64, 0)

    file := flag.Args()[0]
    f, err := os.Open(file)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer f.Close()

```

The final part of `stats.go` is the following:

```

r := bufio.NewReader(f)
for {
    line, err := r.ReadString('\n')
    if err == io.EOF {
        break
    } else if err != nil {
        fmt.Printf("error reading file %s", err)
        break
    }
    line = strings.TrimSpace(line, "\r\n")
    value, err := strconv.ParseFloat(line, 64)
    if err == nil {
        data = append(data, value)
    }
}

```

```

    }

    sort.Float64s(data)

    fmt.Println("Min:", min(data))
    fmt.Println("Max:", max(data))
    fmt.Println("Mean:", meanValue(data))
    fmt.Println("Median:", medianValue(data))
    fmt.Println("Variance:", variance(data))
    fmt.Println("Standard Deviation:", math.Sqrt(variance(data)))
}

```

In this part, you will start reading the input file. There are many ways to get your input – feel free to refer to [Chapter 8, Telling a UNIX System What to Do](#), in order to learn more about working with files in Go.

Notice that although it is not obligatory to sort the slice with the data before processing it, it saves you time in some calculations, so it is performed inside `main()`.

The last block of `fmt.Println()` statements prints the calculated statistical properties.

Executing `stats.go` will generate the following kind of output:

```

$ go run stats.go data.txt
Min: -2
Max: 3
Mean: 1.04
Median: 1.2
Variance: 2.8064
Standard Deviation: 1.6752313273097539

```

Notice that the contents of `data.txt` are as follows:

```

$ cat data.txt
1.2
-2
1
2.0
not valid
3

```

As you can see, each line contains a single value, which means that we are working with the simplest form of data. Although the manipulation of more complex data might be slightly different, the general idea remains the same.

The program of this section calculated every statistical property without using any external Go packages related to statistics. Most of the Go code that follows will use existing Go packages and will not implement everything from scratch.

Regression

Regression is a statistical method for calculating relationships among variables. This section will implement **linear regression**, which is the most popular and simplest regression technique and a very good way to understand your data. Note that regression techniques are not 100% accurate, even if you use higher-order (nonlinear) polynomials. The key with regression, as with most machine learning techniques, is to find a good enough technique and not the perfect technique and model.

Linear regression

The idea behind linear regression is simple: you are trying to model your data using a first-degree equation. A first-degree equation can be represented as $y = a \cdot x + b$.

There are many methods that allow you to find out that first-degree equation that will model your data – all techniques calculate a and b .

Implementing linear regression

The Go code of this section will be saved in `regression.go`, which is going to be presented in three parts. The output of the program will be two floating-point numbers that define `a` and `b` in the first-degree equation.

The first part of `regression.go` contains the following code:

```
package main

import (
    "encoding/csv"
    "flag"
    "fmt"
    "gonum.org/v1/gonum/stat"
    "os"
    "strconv"
)

type xy struct {
    x []float64
    y []float64
}
```

The `xy` structure is used to hold the data and should change according to your data format and values.

The second part of `regression.go` is as follows:

```
func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: regression filename\n")
        return
    }

    filename := flag.Args()[0]
    file, err := os.Open(filename)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer file.Close()

    r := csv.NewReader(file)

    records, err := r.ReadAll()
    if err != nil {
        fmt.Println(err)
```

```

        return
    }
    size := len(records)

    data := xy{
        x: make([]float64, size),
        y: make([]float64, size),
    }
}

```

The final part of `regression.go` is as follows:

```

for i, v := range records {
    if len(v) != 2 {
        fmt.Println("Expected two elements")
        continue
    }

    if s, err := strconv.ParseFloat(v[0], 64); err == nil {
        data.y[i] = s
    }

    if s, err := strconv.ParseFloat(v[1], 64); err == nil {
        data.x[i] = s
    }
}

b, a := stat.LinearRegression(data.x, data.y, nil, false)
fmt.Printf("%.4v x + %.4v\n", a, b)
fmt.Printf("a = %.4v b = %.4v\n", a, b)
}

```

The data from the data file is read into the `data` variable. The function that implements the linear regression is `stat.LinearRegression()` and it returns two numbers, which are `b` and `a`, in that particular order.

At this point, it would be a good time to download the `gonum` package:

```
$ go get -u gonum.org/v1/gonum/stat
```

Executing `regression.go` with the input data stored in `reg_data.txt` will generate the following output:

```
$ go run regression.go reg_data.txt
0.9463 x + -0.3985
a = 0.9463 b = -0.3985
```

The two numbers returned are `a` and `b` from the $y = a x + b$ formula.

The contents of `reg_data.txt` are as follows:

```
$ cat reg_data.txt  
1,2  
3,4.0  
2.1,3  
4,4.2  
5,5.1  
-5,-5.1
```

Plotting data

It is now time to plot the results and the dataset in order to test how accurate the results from the linear regression technique are. For that purpose, we are going to use the Go code of `plotLR.go`, which will be presented in four parts. `plotLR.go` requires three command-line arguments, which are `a` and `b` from the $y = a \cdot x + b$ formula, and the file that contains the data points. The fact that `plotLR.go` does not calculate `a` and `b` on its own gives you the opportunity to experiment with `a` and `b` using your own values or values that were calculated by another utility.

The first part of `plotLR.go` is as follows:

```
package main

import (
    "encoding/csv"
    "flag"
    "fmt"
    "gonum.org/v1/plot"
    "gonum.org/v1/plot/plotter"
    "gonum.org/v1/plot/vg"
    "image/color"
    "os"
    "strconv"
)

type xy struct {
    x []float64
    y []float64
}

func (d xy) Len() int {
    return len(d.x)
}

func (d xy) XY(i int) (x, y float64) {
    x = d.x[i]
    y = d.y[i]
    return
}
```

The `Len()` and `XY()` methods are needed for the plotting part, whereas the `image/color` package is needed to change the colors in the output.

The second part of `plotLR.go` contains the following code:

```

func main() {
    flag.Parse()
    if len(flag.Args()) < 3 {
        fmt.Printf("usage: plotLR filename a b\n")
        return
    }

    filename := flag.Args()[0]
    file, err := os.Open(filename)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer file.Close()

    r := csv.NewReader(file)

    a, err := strconv.ParseFloat(flag.Args()[1], 64)
    if err != nil {
        fmt.Println(a, "not a valid float!")
        return
    }

    b, err := strconv.ParseFloat(flag.Args()[2], 64)
    if err != nil {
        fmt.Println(b, "not a valid float!")
        return
    }

    records, err := r.ReadAll()
    if err != nil {
        fmt.Println(err)
        return
    }
}

```

This part of the program works with the command-line arguments and the reading of the data.

The third part of `plotLR.go` is as follows:

```

size := len(records)

data := xy{
    x: make([]float64, size),
    y: make([]float64, size),
}

for i, v := range records {
    if len(v) != 2 {
        fmt.Println("Expected two elements per line!")
        return
    }

    s, err := strconv.ParseFloat(v[0], 64)
    if err == nil {
        data.y[i] = s
    }
}

```

```

    s, err = strconv.ParseFloat(v[1], 64)
    if err == nil {
        data.x[i] = s
    }
}

```

The final part of `plotLR.go` is as follows:

```

line := plotter.NewFunction(func(x float64) float64 { return a*x + b })
line.Color = color.RGBA{B: 255, A: 255}

p, err := plot.New()
if err != nil {
    fmt.Println(err)
    return
}

plotter.DefaultLineStyle.Width = vg.Points(1)
plotter.DefaultGlyphStyle.Radius = vg.Points(2)

scatter, err := plotter.NewScatter(data)
if err != nil {
    fmt.Println(err)
    return
}
scatter.GlyphStyle.Color = color.RGBA{R: 255, B: 128, A: 255}

p.Add(scatter, line)

w, err := p.WriterTo(300, 300, "svg")
if err != nil {
    fmt.Println(err)
    return
}

_, err = w.WriteTo(os.Stdout)
if err != nil {
    fmt.Println(err)
    return
}
}

```

The function that is going to be plotted is defined using the `plotter.NewFunction()` method.

At this point, you should download some external packages by executing the following commands:

```

$ go get -u gonum.org/v1/plot
$ go get -u gonum.org/v1/plot/plotter
$ go get -u gonum.org/v1/plot/vg

```

Executing `plotLR.go` will generate the following kind of output:

```
$ go run plotLR.go reg_data.txt
usage: plotLR filename a b
$ go run plotLR.go reg_data.txt 0.9463 -0.3985
<?xml version="1.0"?>
<!-- Generated by SVGo and Plotinum VG -->
<svg width="300pt" height="300pt" viewBox="0 0 300 300"
      xmlns="http://www.w3.org/2000/svg"
      xmlns:xlink="http://www.w3.org/1999/xlink">
<g transform="scale(1, -1) translate(0, -300)">
  .
  .
  .

```

Therefore, you should save the generated output in a file before using it:

```
$ go run plotLR.go reg_data.txt 0.9463 -0.3985 > output.svg
```

As the output is in the **Scalable Vector Graphics (SVG)** format, you should load it into a web browser in order to see the results. The results from our data can be seen in the following figure.

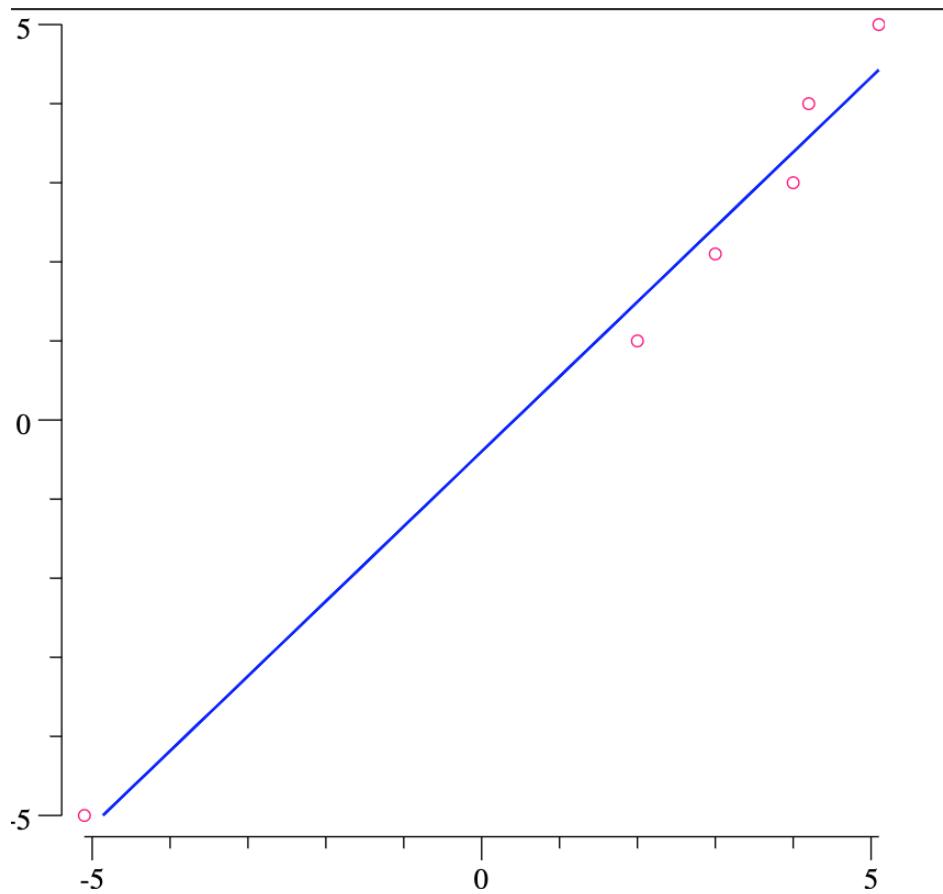


Figure 14.1: The output of the plotLR.go program

The output of the image will also show how accurately the data can be modeled using a linear equation.

Classification

In statistics and machine learning, **classification** is the process of putting elements into existing sets that are called categories. In machine learning, classification is considered a **supervised learning** technique, which is where a set that is considered to contain correctly identified observations is used for training before working with the actual data.

A very popular and easy-to-implement classification method is called **k-nearest neighbors (k-NN)**. The idea behind k-NN is that we can classify data items based on their similarity with other items. The k in k-NN denotes the number of neighbors that are going to be included in the decision, which means that k is a positive integer that is usually pretty small.

The input of the algorithm consists of the k -closest training examples in the feature space. An object is classified by a plurality vote of its neighbors, with the object being assigned to the class that is the most common among its k -NN. If the value of k is 1, then the element is simply assigned to the class that is the nearest neighbor according to the **distance metric** used. The distance metric depends on the data you are dealing with. As an example, you will need a different distance metric when working with complex numbers and another when working with points in three-dimensional space.

The Go code that will illustrate classification in Go can be found in `classify.go`. The file is going to be presented in three parts.

The first part of `classify.go` is as follows:

```
package main

import (
    "flag"
    "fmt"
    "strconv"

    "github.com/sjwhitworth/golearn/base"
    "github.com/sjwhitworth/golearn/evaluation"
    "github.com/sjwhitworth/golearn/knn"
)
```

The second part of `classify.go` contains the following Go code:

```
func main() {
    flag.Parse()
    if len(flag.Args()) < 2 {
        fmt.Printf("usage: classify filename k\n")
        return
    }

    dataset := flag.Args()[0]
    rawData, err := base.ParseCSVToInstances(dataset, false)
    if err != nil {
        fmt.Println(err)
        return
    }

    k, err := strconv.Atoi(flag.Args()[1])
    if err != nil {
        fmt.Println(err)
```

```

        return
    }

    cls := knn.NewKnnClassifier("euclidean", "linear", k)
}

```

The `knn.NewKnnClassifier()` method returns a new **classifier**. The last parameter of the function is the number of neighbors that the classifier will have.

The final part of `classify.go` is as follows:

```

train, test := base.InstancesTrainTestSplit(rawData, 0.50)
cls.Fit(train)

p, err := cls.Predict(test)
if err != nil {
    fmt.Println(err)
    return
}

confusionMat, err := evaluation.GetConfusionMatrix(test, p)
if err != nil {
    fmt.Println(err)
    return
}

fmt.Println(evaluation.GetSummary(confusionMat))
}

```

The `Fit()` method stores the training data in case you want to use it later on, whereas the `Predict()` method returns a classification for the input based on the training data using the k-NN algorithm. Finally, the `evaluation.GetSummary()` method returns a table of precision, recall, true positive, false positive, and true negative values for each class for the given `ConfusionMatrix`, which was calculated using the call to `evaluation.GetConfusionMatrix()`.



As `base.InstancesTrainTestSplit()` will not return the same values all the time, the training and test data will be different each time you execute `classify.go`, which means that you will get different results.

At this point, you should install the github.com/sjwhitworth/golearn package as follows:

```

$ go get -t -u -v github.com/sjwhitworth/golearn
github.com/sjwhitworth/golearn (download)
github.com/sjwhitworth/golearn
$ cd ~/go/src/github.com/sjwhitworth/golearn
$ go get -t -u -v ./...

```

Now, you should return back to your previous directory in order to execute `classify.go` from there. Executing `classify.go` will create the following kind of output:

```

$ go run classify.go class_data.txt 2
Reference ClassnTrue Positives False Positives True Negatives      Precision      Recall      F1 Score
-----
Iris-versicolor   25       0        41     1.0000      0.9259      0.9615
Iris-virginica    5       2        61     0.7143      1.0000      0.8333
Iris-setosa       36       0        32     1.0000      1.0000      1.0000
Overall accuracy: 0.9706
$ go run classify.go class_data.txt 30
Reference ClassnTrue Positives False Positives True Negatives      Precision      Recall      F1 Score
-----
Iris-versicolor   27       5        36     0.8438      1.0000      0.9153
Iris-virginica    0       0        63      NaN        0.0000      NaN
Iris-setosa       36       0        32     1.0000      1.0000      1.0000
Overall accuracy: 0.9265

```

The contents of `class_data.txt`, which are pretty simple, have the following format:

```
$ head -4 class_data.txt
6.7,3.1,5.6,2.4,Iris-virginica
6.9,3.1,5.1,2.3,Iris-virginica
5.8,2.7,5.1,1.9,Iris-virginica
6.8,3.2,5.9,2.3,Iris-virginica
```

The data is taken from the **Iris dataset**.

The precision of the algorithm is high because the number of elements in `class_data.txt` is pretty small. As `golearn` includes some sample data, I can try using the entire Iris dataset on my personal macOS Mojave machine as follows:

```
$ go run classify.go ~/go/src/github.com/sjwhitworth/golearn/examples/datasets/iris.csv 2
Reference ClassnTrue Positives False Positives True Negatives      Precision      Recall      F1 Score
-----  -----  -----  -----  -----  -----  -----  -----  -----
Iris-setosa    30      0      58      1.0000      1.0000      1.0000
Iris-virginica 28      3      56      0.9032      0.9655      0.9333
Iris-versicolor 26      1      58      0.9630      0.8966      0.9286
Overall accuracy: 0.9545
$ go run classify.go ~/go/src/github.com/sjwhitworth/golearn/examples/datasets/iris.csv 50
Reference ClassnTrue Positives False Positives True Negatives      Precision      Recall      F1 Score
-----  -----  -----  -----  -----  -----  -----  -----  -----
Iris-setosa    0      0      58      NaN      0.0000      NaN
Iris-virginica 4      5      54      0.4444      0.1379      0.2105
Iris-versicolor 24     55      4      0.3038      0.8276      0.4444
Overall accuracy: 0.3182
```

If you work with the Iris dataset and try different values for the number of neighbors, you will find out that the larger the number of neighbors you want to use, the smaller the accuracy of the results. Unfortunately, further discussion of the k-NN algorithm is beyond the scope of this book.

Clustering

Clustering is the unsupervised version of classification where the grouping of data into categories is based on some metric of similarity or distance. This section will use **k-means clustering**, which is the most famous clustering technique and one that is also easy to implement. Once again, we are going to use an external library that can be found at <https://github.com/mash/gokmeans>.

The utility that showcases clustering in Go is called `cluster.go`, and it is going to be presented in three parts. The utility requires one command-line argument, which is the number of clusters that are going to be created.

The first part of `cluster.go` is as follows:

```
package main

import (
    "flag"
    "fmt"
    "github.com/mash/gokmeans"
    "strconv"
)

var observations []gokmeans.Node = []gokmeans.Node{
    gokmeans.Node{4},
    gokmeans.Node{5},
    gokmeans.Node{6},
    gokmeans.Node{8},
    gokmeans.Node{10},
    gokmeans.Node{12},
    gokmeans.Node{15},
    gokmeans.Node{0},
    gokmeans.Node{-1},
}
```

This time, the data is included in the program for reasons of simplicity. However, nothing prohibits you from reading it from one or more external files.

The second part of `cluster.go` contains the following Go code:

```

func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: cluster k\n")
        return
    }

    k, err := strconv.Atoi(flag.Args()[0])
    if err != nil {
        fmt.Println(err)
        return
    }
}

```

The final part of `cluster.go` is as follows:

```

if success, centroids := gokmeans.Train(observations, k, 50); success {
    fmt.Println("The centroids are the following:")
    for _, centroid := range centroids {
        fmt.Println(centroid)
    }

    fmt.Println("The clusters are the following:")
    for _, observation := range observations {
        index := gokmeans.Nearest(observation, centroids)
        fmt.Println(observation, "belongs in cluster", index+1, ".")
    }
}
}

```

At this point, you will need to get the external Go library by executing the following command:

```

$ go get -v -u github.com/mash/gokmeans
github.com/mash/gokmeans (download)
github.com/mash/gokmeans

```

Executing `cluster.go` when you want to have a single cluster will produce the following kind of output:

```

$ go run cluster.go 1
The centroids are the following:
[6.55555555555555]
The clusters are the following:
[4] belongs in cluster 1 .
[5] belongs in cluster 1 .
[6] belongs in cluster 1 .
[8] belongs in cluster 1 .
[10] belongs in cluster 1 .
[12] belongs in cluster 1 .
[15] belongs in cluster 1 .
[0] belongs in cluster 1 .
[-1] belongs in cluster 1 .

```

As you have a single cluster, each item will belong to that cluster, hence the output returned by `cluster.go`.

If you change the value of `k`, you will get the following output:

```
$ go run cluster.go 5
The centroids are the following:
[5]
[-0.5]
[13.5]
[10]
[8]
The clusters are the following:
[4] belongs in cluster 1 .
[5] belongs in cluster 1 .
[6] belongs in cluster 1 .
[8] belongs in cluster 5 .
[10] belongs in cluster 4 .
[12] belongs in cluster 3 .
[15] belongs in cluster 3 .
[0] belongs in cluster 2 .
[-1] belongs in cluster 2 .
$ go run cluster.go 8
The centroids are the following:
[0]
[4.5]
[-1]
[9]
[-1]
[6]
[12]
[15]
The clusters are the following:
[4] belongs in cluster 2 .
[5] belongs in cluster 2 .
[6] belongs in cluster 6 .
[8] belongs in cluster 4 .
[10] belongs in cluster 4 .
[12] belongs in cluster 7 .
[15] belongs in cluster 8 .
[0] belongs in cluster 1 .
[-1] belongs in cluster 3 .
```

Anomaly detection

Anomaly detection techniques try to find the probability that a given set contains anomalous behavior, which can be unusual values or patterns.

The utility that is going to be developed in this section is called `anomaly.go`, and it is going to be presented in three parts. The utility uses probabilistic anomaly detection with the help of the `Anomalyzer` package and calculates the probability that the given set of numeric values contains anomalous behavior.

The first part of `anomaly.go` is as follows:

```
package main

import (
    "flag"
    "fmt"
    "math/rand"
    "strconv"
    "time"

    "github.com/lytics/anomalyzer"
)

func random(min, max int) int {
    return rand.Intn(max-min) + min
}
```

The second part of `anomaly.go` is as follows:

```
func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Println("usage: anomaly MAX\n")
        return
    }

    MAX, err := strconv.Atoi(flag.Args()[0])
    if err != nil {
        fmt.Println(err)
        return
    }

    conf := &anomalyzer.AnomalyzerConf{
        Sensitivity: 0.1,
        UpperBound: 5,
        LowerBound: anomalyzer.NA,
        ActiveSize: 1,
        NSeasons: 4,
        Methods: []string{"diff", "fence", "magnitude", "ks"},
    }
}
```

The `anomalyzer` package supports the cdf, diff, high rank, low rank, magnitude, fence, and bootstrap ks algorithmic tests. If you decide to use some or all of them as the value of `Methods`, then `anomalyzer` will execute all of them. As each one of them returns a probability of anomalous behavior, the `anomalyzer` package will compute a weighted mean in order to establish whether the current dataset has anomalous behavior.

The `Sensitivity` field is for the magnitude test – its default value is `0.1`. The `UpperBound` and `LowerBound` fields are used by the fence test. On the other hand, the presence of the `ActiveSize` field is compulsory – its value should be at least `1`. Lastly, the `NSeasons` field has a default value of `4` if it is not defined.

The third part of `anomaly.go` contains the following Go code:

```
data := []float64{}
SEED := time.Now().Unix()
rand.Seed(SEED)

for i := 0; i < MAX; i++ {
    data = append(data, float64(random(0, MAX)))
}
fmt.Println("data:", data)
```

The `anomaly.go` code generates random data on its own. The number of elements is defined as a command-line argument to the program.

The final part of `anomaly.go` is as follows:

```
| anom, _ := anomalyzer.NewAnomalyzer(conf, data)
| prob := anom.Push(8.0)
| fmt.Println("Anomalous Probability:", prob)
| }
```

At this point, you should download the external Go package as follows:

```
| $ go get -v -u github.com/lytics/anomalyzer
| github.com/lytics/anomalyzer (download)
| github.com/drewlanenga/govector (download)
| github.com/drewlanenga/govector
| github.com/lytics/anomalyzer
```

Executing `anomaly.go` will generate the following kind of output:

```
| $ go run anomaly.go 20
| data: [18 3 2 19 2 16 5 15 3 14 2 9 11 10 2 17 17 14 19 1]
| Anomalous Probability: 0.8612730015082957
| $ go run anomaly.go 20
| data: [17 8 19 10 0 14 12 7 7 13 2 5 18 1 15 4 0 14 13 9]
| Anomalous Probability: 0.7885470085470085
| $ go run anomaly.go 100
| data: [85 5 64 32 69 55 0 67 11 96 75 92 25 54 2 49 58 6 16 38 55 11 93 90 90 47 66 97 37 61 85 92 15 45 33 43 61 44 73 18 10 8
| Anomalous Probability: 0.8977395577395577
```

Neural networks

Neural networks, which try to work like the human brain, learn to perform tasks based on given examples. Neural network have layers, and the smallest neural network must have at least two layers: input and output. During the training phase, data flows through the layers of the neural network. The actual output values of the training data are used to correct the calculated output values of the training data so that the next iteration will be more precise.

The utility that will be developed in this section is named `neural.go`, and it will implement a really simple neural network. This is going to be presented in four parts.

The first part of `neural.go` is as follows:

```
package main

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

    "github.com/goml/gobrain"
)
```

The new line in the `import` list tells the `gofmt` tool to sort the package names in blocks that are separated by new lines. The second part of `neural.go` contains the following Go code:

```
func main() {
    seed := time.Now().Unix()
    rand.Seed(seed)

    patterns := [][][]float64{
        {{0, 0, 0, 0}, {0}},
        {{0, 1, 0, 1}, {1}},
        {{1, 0, 1, 0}, {1}},
        {{1, 1, 1, 1}, {1}},
    }
```

The `patterns` slice holds the training data that will be used later. The `rand.Seed()` function initializes a new random-number generator that is automatically used by the github.com/goml/gobrain package.

The third part of `neural.go` is as follows:

```
    ff := &gobrain.FeedForward{}
    ff.Init(4, 2, 1)
    ff.Train(patterns, 1000, 0.6, 0.4, false)
```

In this part, we initialize the neural network. The first parameter of the `Init()` method is the number of inputs, the second is the number of hidden nodes, and the third is the number of outputs. The dimension and the data of the `patterns` slice should be in compliance with the values in `Init()`, and vice versa.

The final part of `neural.go` is as follows:

```
    in := []float64{1, 1, 0, 1}
    out := ff.Update(in)
    fmt.Println(out)

    in = []float64{0, 0, 0, 0}
    out = ff.Update(in)
    fmt.Println(out)
}
```

There are two tests happening here. For the first test, the input is `{1, 1, 0, 1}`, whereas for the second test, the input is `{0, 0, 0, 0}`.

You should know by now which Go package you will need to download using `go get`, so please download it before trying to run `neural.go`.

Executing `neural.go` will generate the following kind of output:

```
$ go run neural.go
[0.9918648920317314]
[0.02826477691747802]
```

The first value is close to 1, whereas the second value is close to 0.

As there is a kind of randomness introduced, executing `neural.go` multiple times will generate slightly different results:

```
$ go run neural.go  
[0.9920127780655835]  
[0.028029429851140687]  
go run neural.go  
[0.9913803776914417]  
[0.028875009295811015]
```

Outlier analysis

Outlier analysis is about finding the values that look like they do not belong with the rest of the values. Put simply, outliers are extreme values that greatly differ from the other observations. A good book on outlier analysis is *Outlier Analysis, 2nd Edition*, by Charu C. Aggarwal (Springer, 2017).

The outlier technique that is going to be implemented in `outlier.go`, which is going to be presented in four parts, is based on standard deviation. You will learn more about this technique later.

The first part of `outlier.go` is as follows:

```
package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "math"
    "os"
    "sort"
    "strconv"
    "strings"
)
```

The technique that is going to be implemented requires packages of the standard Go library only.

The second part of `outlier.go` contains the following Go code:

```
func variance(x []float64) float64 {
    mean := meanValue(x)
    sum := float64(0)
    for _, v := range x {
        sum = sum + (v-mean)*(v-mean)
    }
    return sum / float64(len(x))
}

func meanValue(x []float64) float64 {
    sum := float64(0)
    for _, v := range x {
```

```

        sum = sum + v
    }
    return sum / float64(len(x))
}

```

These two functions were used previously in `stats.go`. If you find yourself using the same functions all the time, it would be good to create one or more Go libraries and group your Go code.

The third part of `outlier.go` contains the following Go code:

```

func outliers(x []float64, limit float64) []float64 {
    deviation := math.Sqrt(variance(x))
    mean := meanValue(x)
    anomaly := deviation * limit
    lower_limit := mean - anomaly
    upper_limit := mean + anomaly
    fmt.Println(lower_limit, upper_limit)

    y := make([]float64, 0)
    for _, val := range x {
        if val < lower_limit || val > upper_limit {
            y = append(y, val)
        }
    }
    return y
}

```

This function holds the logic of the program. It calculates the standard deviation and the mean value of the sample in order to compute the upper and lower limits. Everything outside these two limits will be considered an outlier.

The final part of `outlier.go` is as follows:

```

func main() {
    flag.Parse()
    if len(flag.Args()) != 2 {
        fmt.Printf("usage: stats filename limit\n")
        return
    }

    file := flag.Args()[0]
    f, err := os.Open(file)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer f.Close()

    limit, err := strconv.ParseFloat(flag.Args()[1], 64)
}

```

```

    if err != nil {
        fmt.Println(err)
        return
    }

    data := make([]float64, 0)
    r := bufio.NewReader(f)
    for {
        line, err := r.ReadString('\n')
        if err == io.EOF {
            break
        } else if err != nil {
            fmt.Printf("error reading file %s", err)
            break
        }
        line = strings.TrimRight(line, "\r\n")
        value, err := strconv.ParseFloat(line, 64)
        if err == nil {
            data = append(data, value)
        }
    }

    sort.Float64s(data)
    out := outliers(data, limit)
    fmt.Println(out)
}

```

The `main()` function deals with command-line arguments and the reading of the input file before calling the `outliers()` function.

Executing `outlier.go` will produce the following kind of output:

```

$ go run outlier.go data.txt 2
-94.21189713007178 95.36745268562734
[-100 100]
$ go run outlier.go data.txt 5
-236.3964094918461 237.55196504740167
[]
$ go run outlier.go data.txt 0.02
-0.3701189713007176 1.5256745268562737
[-100 -10 -2 2 3 10 100]

```

If you lower the value of the `limit` variable, you will find more outliers in your data. However, this depends on the problem you are trying to solve.

The contents of `data.txt` are the following:

```

$ cat data.txt
1.2
-2
1
2.0
not valid

```

3
10
100
-10
-100

Working with TensorFlow

TensorFlow is a rather famous open-source platform for machine learning. In order to use TensorFlow with Go, you will first need to download a Go package:

```
| $ go get github.com/tensorflow/tensorflow/tensorflow/go
```

However, for the aforementioned command to work, the C interface for TensorFlow should be already installed. On a macOS Mojave machine, this can be installed as follows:

```
| $ brew install tensorflow
```

If the C interface is not installed, and you try to install the Go package for TensorFlow, you will get the following error message:

```
| $ go get github.com/tensorflow/tensorflow/tensorflow/go
| # github.com/tensorflow/tensorflow/tensorflow/go
| ld: library not found for -ltensorflow
| clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

As TensorFlow is pretty complex, it would be good to execute the following command in order to validate your installation:

```
| $ go test github.com/tensorflow/tensorflow/tensorflow/go
| ok    github.com/tensorflow/tensorflow/tensorflow/go      0.109s
```

Apart from the Go tests, you can also execute the following Go program, which will print the version of the Go TensorFlow package that you are using:

```
package main

import (
    tf "github.com/tensorflow/tensorflow/tensorflow/go"
    "github.com/tensorflow/tensorflow/tensorflow/go/op"
    "fmt"
)

func main() {
    s := op.NewScope()
    c := op.Const(s, "Using TensorFlow version: "+tf.Version())
    graph, err := s.Finalize()

    if err != nil {
        fmt.Println(err)
        return
    }

    sess, err := tf.NewSession(graph, nil)
    if err != nil {
        fmt.Println(err)
        return
    }

    output, err := sess.Run(nil, []tf.Output{c}, nil)
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Println(output[0].Value())
}
```

If you save that program as `tfversion.go` and execute it, you will get the following output:

```
| $ go run tfversion.go
| 2019-06-10 22:30:12.880532: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that this TensorFlow version was not compiled to use: AVX2
| Using TensorFlow version: 1.13.1
```

The first message is a warning generated by the TensorFlow Go package and you can ignore it at the moment.

We are now ready to continue and I will show you a real TensorFlow program that will illustrate how TensorFlow works. The related code is saved in `tFlow.go` and it is going to be presented in four parts. What the program does is add and multiply two numbers that are given as command-line arguments to the program.

The first part of `tFlow.go` is the following:

```
package main

import (
    "fmt"
    "os"
    "strconv"

    "github.com/tensorflow/tensorflow/tensorflow/go"
    "github.com/tensorflow/tensorflow/tensorflow/go/op"
)
```

The second part of `tFlow.go` contains the following code:

```
func Add(sum_arg1, sum_arg2 int8) (interface{}, error) {
    sum_scope := op.NewScope()
    input1 := op.Placeholder(sum_scope.SubScope("a1"), tf.Int8)
    input2 := op.Placeholder(sum_scope.SubScope("a2"), tf.Int8)
    sum_result_node := op.Add(sum_scope, input1, input2)

    graph, err := sum_scope.Finalize()
    if err != nil {
        fmt.Println(err)
        return 0, err
    }

    a1, err := tf.NewTensor(sum_arg1)
    if err != nil {
        fmt.Println(err)
        return 0, err
    }

    a2, err := tf.NewTensor(sum_arg2)
    if err != nil {
        fmt.Println(err)
        return 0, err
    }

    session, err := tf.NewSession(graph, nil)
    if err != nil {
        fmt.Println(err)
        return 0, err
    }
    defer session.Close()

    sum, err := session.Run(
        map[tf.Output]*tf.Tensor{
            input1: a1,
            input2: a2,
        },
        []tf.Output{sum_result_node}, nil)

    if err != nil {
        fmt.Println(err)
        return 0, err
    }

    return sum[0].Value(), nil
}
```

The `Add()` function adds two `int8` numbers. The code is pretty big for just adding two numbers, which illustrates that TensorFlow has a certain way of working. This mainly happens because TensorFlow is an advanced environment with many capabilities.

The third part of `tFlow.go` is as follows:

```
func Multiply(sum_arg1, sum_arg2 int8) (interface{}, error) {
    sum_scope := op.NewScope()
    input1 := op.Placeholder(sum_scope.SubScope("x1"), tf.Int8)
    input2 := op.Placeholder(sum_scope.SubScope("x2"), tf.Int8)

    sum_result_node := op.Mul(sum_scope, input1, input2)
    graph, err := sum_scope.Finalize()
    if err != nil {
        fmt.Println(err)
    }
```

```

        return 0, err
    }

    x1, err := tf.NewTensor(sum_arg1)
    if err != nil {
        fmt.Println(err)
        return 0, err
    }

    x2, err := tf.NewTensor(sum_arg2)
    if err != nil {
        fmt.Println(err)
        return 0, err
    }

    session, err := tf.NewSession(graph, nil)
    if err != nil {
        fmt.Println(err)
        return 0, err
    }
    defer session.Close()

    sum, err := session.Run(
        map[tf.Output]*tf.Tensor{
            input1: x1,
            input2: x2,
        },
        []tf.Output{sum_result_node}, nil)

    if err != nil {
        fmt.Println(err)
        return 0, err
    }

    return sum[0].Value(), nil
}

```

The `Multiply()` function multiplies two `int8` values and returns the result.

The final part of `tFlow.go` is the following:

```

func main() {
    if len(os.Args) != 3 {
        fmt.Println("Need two integer parameters!")
        return
    }

    t1, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }
    n1 := int8(t1)

    t2, err := strconv.Atoi(os.Args[2])
    if err != nil {
        fmt.Println(err)
        return
    }
    n2 := int8(t2)

    res, err := Add(n1, n2)
    if err != nil {
        fmt.Println(err)
    } else {
        fmt.Println("Add:", res)
    }

    res, err = Multiply(n1, n2)
    if err != nil {
        fmt.Println(err)
    } else {
        fmt.Println("Multiply:", res)
    }
}

```

Executing `tFlow.go` will generate the following kind of output:

```

$ go run tFlow.go 1 20
2019-06-14 18:46:52.115676: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2
Add: 21
Multiply: 20
$ go run tFlow.go -2 20
2019-06-14 18:47:23.104918: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2
Add: 18
Multiply: -40

```

The warning message that appears on the output tells you that TensorFlow is not as fast as it can be on your machine. Basically, it informs you that you will need to compile TensorFlow from scratch in order to resolve that warning message.

Talking to Kafka

In this section, you will learn how to write and read JSON records using **Kafka**. Its name was inspired by the author Franz Kafka, because Kafka software is optimized for writing. Kafka is written in **Scala** and **Java**. It was originally developed by LinkedIn and donated to the Apache Software Foundation back in 2011. Kafka's design was influenced by transaction logs.

The main advantage of Kafka is that it can be used to store lots of data fast, which might interest you when you have to work with huge amounts of real-time data. However, its disadvantage is that in order to maintain that speed, the data is read-only and stored in a naive way.

The following program, which is named `writeKafka.go`, will illustrate how to write data to Kafka. In Kafka terminology, `writeKafka.go` is a **producer**. The utility will be presented in three parts.

The first part of `writeKafka.go` is as follows:

```
package main

import (
    "context"
    "encoding/json"
    "fmt"
    "github.com/segmentio/kafka-go"
    "math/rand"
    "os"
    "strconv"
    "time"
)
```

The Kafka Go driver requires the use of an external package that you will need to download on your own.

The second part of `writeKafka.go` contains the following Go code:

```
type Record struct {
    Name   string `json:"name"`
    Random int    `json:"random"`
}

func random(min, max int) int {
    return rand.Intn(max-min) + min
}

func main() {
    MIN := 0
    MAX := 0
    TOTAL := 0
    topic := ""
    if len(os.Args) > 4 {
        MIN, _ = strconv.Atoi(os.Args[1])
        MAX, _ = strconv.Atoi(os.Args[2])
        TOTAL, _ = strconv.Atoi(os.Args[3])
        topic = os.Args[4]
    } else {
        fmt.Println("Usage:", os.Args[0], "MIN MAX TOTAL TOPIC")
        return
    }
}
```

The `Record` structure is used to store the data that will be written to the desired Kafka topic.

The third part of `writeKafka.go` contains the following Go code:

```
partition := 0
conn, err := kafka.DialLeader(context.Background(), "tcp", "localhost:9092", topic, partition)
if err != nil {
    fmt.Printf("%s\n", err)
    return
}
rand.Seed(time.Now().Unix())
```

In this part of the program, you define the address (`localhost:9092`) of the Kafka server.

The final part of `writeKafka.go` is as follows:

```
for i := 0; i < TOTAL; i++ {
    myrand := random(MIN, MAX)
    temp := Record(strconv.Itoa(i), myrand)
    recordJSON, _ := json.Marshal(temp)

    conn.SetWriteDeadline(time.Now().Add(1 * time.Second))
    conn.WriteMessages(
        kafka.Message{Value: []byte(recordJSON)},
    )

    if i%50 == 0 {
        fmt.Println(".")
    }
    time.Sleep(10 * time.Millisecond)
}

fmt.Println()
conn.Close()
```

The following Go program, which is named `readKafka.go`, will illustrate how to write data to Kafka. Programs that read data from Kafka are called **consumers** in Kafka terminology. The Go code of `readKafka.go` will be presented in four parts.

The first part of `readKafka.go` is as follows:

```
package main

import (
    "context"
    "encoding/json"
    "fmt"
    "github.com/segmentio/kafka-go"
    "os"
)
```

The `readKafka.go` utility also requires an external Go package.

The second part of `readKafka.go` is as follows:

```
type Record struct {
    Name string `json:"name"`
    Random int `json:"random"`
}

func main() {
    if len(os.Args) < 2 {
        fmt.Println("Need a Kafka topic name.")
        return
    }

    partition := 0
    topic := os.Args[1]
    fmt.Println("Kafka topic:", topic)
```

This part of the program defines the Go structure that will hold the Kafka records and deals with the command-line arguments of the program.

The third part of `readKafka.go` contains the following Go code:

```
r := kafka.NewReader(kafka.ReaderConfig{
    Brokers:  []string{"localhost:9092"},
    Topic:    topic,
    Partition: partition,
    MinBytes: 10e3,
    MaxBytes: 10e6,
})
r.SetOffset(0)
```

The `kafka.NewReader()` structure holds the data required to connect to the Kafka server process and Kafka topic.

The final code segment from `readKafka.go` is the following:

```
for {
    m, err := r.ReadMessage(context.Background())
    if err != nil {
```

```

        break
    }
    fmt.Printf("message at offset %d: %s = %s\n", m.Offset, string(m.Key), string(m.Value))

    temp := Record{}
    err = json.Unmarshal(m.Value, &temp)
    if err != nil {
        fmt.Println(err)
    }
    fmt.Printf("%T\n", temp)
}

r.Close()
}

```

This `for` loop reads the data from the desired Kafka topic and prints it on the screen. Note that the program will keep waiting for data and will never end.

For the purposes of this section, we are going to use a Docker image with Kafka in order to execute both `readKafka.go` and `writeKafka.go`. So, first, we will need to execute the following command to download the desired Kafka image, if it is not already on our local machine:

```
$ docker pull landoop/fast-data-dev:latest
```

The output of the `docker images` command will be able to verify that the desired Kafka Docker image is there. Then, you should execute that image and run it as a **container**, as follows:

```
$ docker run --rm --name=kafka-box -it -p 2181:2181 -p 3030:3030 -p 8081:8081 -p 8082:8082 -p 8083:8083 -p 9092:9092 -p 9581:9!
Setting advertised host to 127.0.0.1.
Starting services.
This is Landoop's fast-data-dev. Kafka 2.0.1-L0 (Landoop's Kafka Distribution).
You may visit http://127.0.0.1:3030 in about a minute.
.
.
.
```

For the presented utilities to work, you will need to download the Go package that allows you to communicate with Kafka.

This can be done as follows:

```
$ go get -u github.com/segmentio/kafka-go
```

After that, you are ready to use the Docker image of Kafka and execute the two Go utilities. Executing `writeKafka.go` will generate the following kind of output:

```
$ go run writeKafka.go 1 1000 500 my_topic
.....
$ go run writeKafka.go 1 1000 500 my_topic
.....
```

Executing `readKafka.go` will produce the following output:

```
$ go run readKafka.go my_topic | head
Kafka topic: my_topic
message at offset 0:  = {"name":"0","random":134}
main.Record
message at offset 1:  = {"name":"1","random":27}
main.Record
message at offset 2:  = {"name":"2","random":168}
main.Record
message at offset 3:  = {"name":"3","random":317}
main.Record
message at offset 4:  = {"name":"4","random":455}
signal: broken pipe
```

Additional resources

Have a look at the following resources:

- You can learn more about Kafka by visiting <https://kafka.apache.org/>.
- You can learn more about TensorFlow at <https://www.tensorflow.org/>.
- **Lenses** is a great product for working with Kafka and Kafka records. You can learn more about Lenses at <https://lenses.io/>.
- You can find the documentation page of the Go TensorFlow package at <https://godoc.org/github.com/tensorflow/tensorflow/tensorflow/go>.
- You can learn more about the Iris dataset at <https://archive.ics.uci.edu/ml/datasets/iris>.
- Have a look at <https://machinebox.io/>, which is machine learning software for people without a science degree, is free for developers, and works using Docker images.
- The documentation page for the Anomalyzer Go package can be found at <https://github.com/lytics/anomalyzer>.

Exercises

- Develop your own Kafka producer that will write JSON records with three fields to a Kafka topic.
- A very interesting statistical property is **covariance**. Find its formula and implement it in Go.
- Change the code of `stats.go` in order to work with integer values only.
- Modify `cluster.go` in order to get the data from an external file that will be given as a command-line argument to the program.
- Change the code of `outlier.go` in order to divide the input into two slices and work with each one of these slices.
- Change the code of `outlier.go` in order to accept the upper and lower limits from the user without calculating them.
- If you find TensorFlow difficult to use, you can try `tfgo`, which can be found at <https://github.com/galeone/tfgo>.

Summary

This chapter talked about many interesting things related to machine learning, including regression, anomaly detection, classification, clustering, and outliers. Go is a great and robust programming language for use in machine learning areas, so feel free to include it in your machine learning projects.

Where to go next?

Philosophically speaking, no programming book can ever be perfect, and neither is this book! Did I leave some Go topics out? Absolutely, yes! Why? Because there are always more topics that could be covered in a book, so if I tried to cover everything, the book would never be ready for publication. This situation is somehow analogous to the specifications of a program: you can always add new and exciting features, but if you do not freeze its specifications, the program will always be under development and will never be ready.

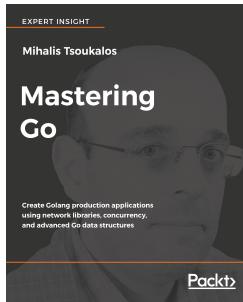
The good thing is that after reading this book, you will be ready to learn on your own, which is the biggest benefit you can get from any good computer book on programming. The main purpose of this book is to help you to learn how to program in Go and gain some experience. However, there is no substitute for trying things on your own and failing often because the only way to learn a programming language is to keep developing non-trivial things. You are now ready to start writing your own software in Go and learning new things.

I would like to congratulate you and thank you for choosing this book, with the hope that you found it useful and will continue to use it as a reference. Go is a great programming language that I believe you will not regret learning. This is the end of another Go book for me, but just the beginning of the journey for you!

Soli Deo gloria

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

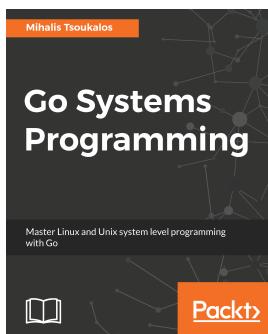


Mastering Go

Mihalis Tsoukalos

ISBN: 978-1-78862-654-5

- Understand the design choices of Golang syntax
- Know enough Go internals to be able to optimize Golang code
- Appreciate concurrency models available in Golang
- Understand the interplay of systems and networking code
- Write server-level code that plays well in all environments
- Understand the context and appropriate use of Go data types and data structures



Go Systems Programming

Mihalis Tsoukalos

ISBN: 978-1-78712-564-3

- Explore the Go language from the standpoint of a developer conversant with Unix, Linux, and so on
- Understand Goroutines, the lightweight threads used for systems and concurrent applications
- Learn how to translate Unix and Linux systems code in C to Golang code
- How to write fast and lightweight server code
- Dive into concurrency with Go
- Write low-level networking code

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!