Mark Summerfield

Programming in Go

Creating Applications for the 21st Century

Developer's Library

Programming in Go

Developer's Library Series

Visit **developers-library.com** for a complete list of available products

---

The **Developer's Library Series** from Addison-Wesley provides practicing programmers with unique, high-quality references and tutorials on the latest programming languages and technologies they use in their daily work. All books in the Developer's Library are written by expert technology practitioners who are exceptionally skilled at organizing and presenting information in a way that's useful for other programmers.

Developer's Library books cover a wide range of topics, from open-source programming languages and databases, Linux programming, Microsoft, and Java, to Web development, social networking platforms, Mac/iPhone programming, and Android programming.

PEARSON

Addison-Wesley | Cisco Press | EXAM/CRAM | IBM Press | que | PRENTICE HALL | SAMS | Safari Books Online

Programming in Go

Creating Applications for the 21st Century

Mark Summerfield

Addison-Wesley

Upper Saddle River, NJ · Boston · Indianapolis · San Francisco
New York · Toronto · Montreal · London · Munich · Paris · Madrid
Capetown · Sydney · Tokyo · Singapore · Mexico City

Library of Congress Cataloging-in-Publication Data

Summerfield, Mark.
Programming in Go : creating applications for the 21st century / Mark Summerfield.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-321-77463-7 (pbk. : alk. paper)
1. Go (Computer program language) 2. Computer programming 3. Application software—Development I. Title.

QA76.73.G63S86 2012
005.13′3—dc23

2012001914

ISBN-13: 978-0-321-77463-7
ISBN-10: 0-321-77463-9

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, April 2012

This book is dedicated to
Jasmin Blanchette and Trenton Schulz

This page intentionally left blank

Contents at a Glance

`www.qtrac.eu/gobook.html`

This page intentionally left blank

Contents

Tables

Introduction

The purpose of this book is to teach solid idiomatic Go programming using all the features the language provides, as well as the most commonly used Go packages from Go's standard library. The book is also designed to serve as a useful reference once the language is learned. To meet both of these goals the book is quite comprehensive and tries to cover every topic in just one place—and with forward and backward cross-references throughout.

Go is quite C-like in spirit, being a small and efficient language with convenient low-level facilities such as pointers. Yet Go also offers many features associated with high- or very high-level languages, such as Unicode strings, powerful built-in data structures, duck typing, garbage collection, and high-level concurrency support that uses communication rather than shared data and locks. Go also has a large and wide-ranging standard library.

The reader is assumed to have programming experience in a mainstream programming language such as C, C++, Java, Python, or similar, although all of Go's unique features and idioms are illustrated with complete runnable examples that are fully explained in the text.

To successfully learn any programming language it is necessary to write programs in that language. To this end the book's approach is wholly practical, and readers are encouraged to experiment with the examples, try the exercises, and write their own programs to get hands-on experience. As with all my previous books, the quoted code snippets are of "live code"; that is, the code was automatically extracted from `.go` source files and directly embedded in the PDF that went to the publisher—so there are no cut and paste errors, and the code works. Wherever possible, small but complete programs and packages are used as examples to provide realistic use cases. The examples, exercises, and solutions are available online at `www.qtrac.eu/gobook.html`.

The book's key aim is to teach the Go *language,* and although many of the standard Go packages are used, not all of them are. This is not a problem, since reading the book will provide enough Go knowledge for readers to be able to make use of any of the standard packages, or any third-party Go package, and of course, be able to create their own packages.

Why Go?

The Go programming language began as an internal Google project in 2007. The original design was by Robert Griesemer and Unix luminaries Rob Pike and Ken Thompson. On November 10, 2009, Go was publicly unveiled under a liberal

open source license. Go is being developed by a team at Google which includes the original designers plus Russ Cox, Andrew Gerrand, Ian Lance Taylor, and many others. Go has an open development model and many developers from around the world contribute to it, with some so trusted and respected that they have the same commit privileges as the Googlers. In addition, many third-party Go packages are available from the Go Dashboard (`godashboard.appspot.com/project`).

Go is the most exciting new mainstream language to appear in at least 15 years and is the first such language that is aimed squarely at 21st century computers—and their programmers.

Go is designed to scale efficiently so that it can be used to build very big applications—and to compile even a large program in mere seconds on a single computer. The lightning-fast compilation speed is made possible to a small extent because the language is easy to parse, but mostly because of its dependency management. If file *app.go* depends on file *pkg1.go*, which in turn depends on *pkg2.go*, in a conventional compiled language *app.go* would need both *pkg1.go*'s and *pkg2.go*'s object files. But in Go, everything that *pkg2.go* exports is cached in *pkg1.go*'s object file, so *pkg1.go*'s object file alone is sufficient to build *app.go*. For just three files this hardly matters, but it results in huge speedups for large applications with lots of dependencies.

Since Go programs are so fast to build, it is practical to use them in situations where scripting languages are normally used (see the sidebar "Go Shebang Scripts", ➤ 10). Furthermore, Go can be used to build web applications using Google's App Engine.

Go uses a very clean and easy-to-understand syntax that avoids the complexity and verbosity of older languages like C++ (first released in 1983) or Java (first released in 1995). And Go is a strongly statically typed language, something which many programmers regard as essential for writing large programs. Yet Go's typing is not burdensome due to Go's short "declare and initialize" variable declaration syntax (where the compiler deduces the type so it doesn't have to be written explicitly), and because Go supports a powerful and convenient version of duck typing.

Languages like C and C++ require programmers to do a vast amount of bookkeeping when it comes to memory management—bookkeeping that could be done by the computer itself, especially for concurrent programs where keeping track can be fiendishly complicated. In recent years C++ has greatly improved in this area with various "smart" pointers, but is only just catching up with Java with regard to its threading library. Java relieves the programmer from the burden of memory management by using a garbage collector. C has only third-party threading libraries, although C++ now has a standard threading library. However, writing concurrent programs in C, C++, or Java requires considerable

bookkeeping by programmers to make sure they lock and unlock resources at the right times.

The Go compiler and runtime system takes care of the tedious bookkeeping. For memory management Go has a garbage collector, so there's no need for smart pointers or for manually freeing memory. And for concurrency, Go provides a form of CSP (Communicating Sequential Processes) based on the ideas of computer scientist C. A. R. Hoare, that means that many concurrent Go programs don't need to do any locking at all. Furthermore, Go uses *goroutines*—very lightweight processes which can be created in vast numbers that are automatically load-balanced across the available processors and cores—to provide much more fine-grained concurrency than older languages' thread-based approaches. In fact, Go's concurrency support is so simple and natural to use that when porting single-threaded programs to Go it often happens that opportunities for using concurrency arise that lead to improved runtimes and better utilization of machine resources.

Go is a pragmatic language that favors efficiency and programmer convenience over purity. For example, Go's built-in types and user-defined types are not the same, since the former can be highly optimized in ways the latter can't be. Go also provides two fundamental built-in collection types: *slices* (for all practical purposes these are references to variable-length arrays) and *maps* (*key–value* dictionaries or hashes). These collection types are highly efficient and serve most purposes extremely well. However, Go supports pointers (it is a fully compiled language—there's no virtual machine getting in the way of performance), so it is possible to create sophisticated custom types, such as balanced binary trees, with ease.

While C supports only procedural programming and Java forces programmers to program everything in an object-oriented way, Go allows programmers to use the paradigm best suited to the problem. Go can be used as a purely procedural language, but also has excellent support for object-oriented programming. As we will see, though, Go's approach to object orientation is radically different from, say, C++, Java, or Python—and is easier to use and much more flexible than earlier forms.

Like C, Go lacks generics (templates in C++-speak); however, in practice the other facilities that Go provides in many cases obviate the need for generics. Go does not use a preprocessor or include files (which is another reason why it compiles so fast), so there is no need to duplicate function signatures as there is in C and C++. And with no preprocessor, a program's semantics cannot change behind a Go programmer's back as it can with careless #*define*s in C and C++.

Arguably, C++, Objective-C, and Java have all attempted to be better Cs (the latter indirectly as a better C++). Go can also be seen as an attempt to be a better C, even though Go's clean, light syntax is reminiscent of Python—and Go's slices and maps are very similar to Python's lists and dicts. However, Go is closer in

spirit to C than to any other language, and can be seen as an attempt to avoid C's drawbacks while providing all that's best in C, as well as adding many powerful and useful features that are unique to Go.

Originally Go was conceived as a systems programming language for developing large-scale programs with fast compilation that could take advantage of distributed systems and multicore networked computers. Go's reach has already gone far beyond the original conception and it is now being used as a highly productive general-purpose programming language that's a pleasure to use and maintain.

The Structure of the Book

Chapter 1 begins by explaining how to build and run Go programs. The chapter then provides a brief overview of Go's syntax and features, as well as introducing some of its standard library. This is done by presenting and explaining a series of five very short examples, each illustrating a variety of Go features. This chapter is designed to provide just a flavor of the language and to give readers a feel for the scope of what is required to learn Go. (How to obtain and install Go is also explained in this chapter.)

Chapters 2 to 7 cover the Go language in depth. Three chapters are devoted to built-in data types: Chapter 2 covers identifiers, Booleans, and numbers; Chapter 3 covers strings; and Chapter 4 covers Go's collection types.

Chapter 5 describes and illustrates Go's statements and control structures. It also explains how to create and use custom functions, and completes the chapters that show how to create procedural nonconcurrent programs in Go.

Chapter 6 shows how to do object-oriented programming in Go. This chapter includes coverage of Go `structs` used for aggregating and embedding (delegating) values, and Go `interfaces` for specifying abstract types, as well as how to produce an inheritance-like effect in some situations. The chapter presents several complete fully explained examples to help ensure understanding, since Go's approach to object orientation may well be different from most readers' experience.

Chapter 7 covers Go's concurrency features and has even more examples than the chapter on object orientation, again to ensure a thorough understanding of these novel aspects of the Go language.

Chapter 8 shows how to read and write custom binary, Go binary, text, JSON, and XML files. (Reading and writing text files is very briefly covered in Chapter 1 and several subsequent chapters since this makes it easier to have useful examples and exercises.)

The book's final chapter is Chapter 9. This chapter begins by showing how to import and use standard library packages, custom packages, and third-party

packages. It also shows how to document, unit test, and benchmark custom packages. The chapter's last sections provide brief overviews of the tools provided with the *gc* compiler, and of Go's standard library.

Although Go is quite a small language, it is a very rich and expressive language (as measured in syntactic constructs, concepts, and idioms), so there is a surprising amount to learn. This book shows examples in good idiomatic Go style right from the start.* This approach, of course, means that some things are shown before being fully explained. We ask the reader to take it on trust that everything will be explained over the course of the book (and, of course, cross-references are provided for everything that is not explained on the spot).

Go is a fascinating language, and one that is really nice to use. It isn't hard to learn Go's syntax and idioms, but it does introduce some novel concepts that may be unfamiliar to many readers. This book tries to give readers the conceptual breakthroughs—especially in object-oriented Go programming and in concurrent Go programming—that might take weeks or even months for those whose only guide is the good but rather terse documentation.

Acknowledgments

Every technical book I have ever written has benefited from the help and advice of others, and this one is no different in this regard.

I want to give particular thanks to two friends who are programmers with no prior Go experience: Jasmin Blanchette and Trenton Schulz. Both have contributed to my books for many years, and in this case their feedback has helped to ensure that this book will meet the needs of other programmers new to Go.

The book was also greatly enhanced by the feedback I received from core Go developer Nigel Tao. I didn't always take his advice, but his feedback was always illuminating and resulted in great improvements both to the code and to the text.

I had additional help from others, including David Boddie, a programmer new to Go, who gave some valuable feedback. And Go developers Ian Lance Taylor, and especially Russ Cox, between them solved many problems both of code and concepts, and provided clear and precise explanations that contributed greatly to the book's accuracy.

During the writing of the book I asked many questions on the `golang-nuts` mailing list and always received thoughtful and useful replies from many different

* The one exception is that in the early chapters we always declare channels to be bidirectional, even when they are used only unidirectionally. Channels are declared to have a particular direction wherever this makes sense, starting from Chapter 7.

posters. I also received feedback from readers of the Safari "rough cut" preview edition that led to some important clarifications.

The Italian software company `www.develer.com`, in the person of Giovanni Bajo, was kind enough to provide me with free Mercurial repository hosting to aid my peace of mind over the long process of writing this book. Thanks to Lorenzo Mancini for setting it all up and looking after it for me. I'm also very grateful to Anton Bowers and Ben Thompson who have been hosting my web site, `www.qtrac.eu`, on their web server since early 2011.

Thanks to Russel Winder for his coverage of software patents in his blog, `www.russel.org.uk`. Appendix B borrows a number of his ideas.

And as always, thanks to Jeff Kingston, creator of the *lout* typesetting system that I have used for all my books and many other writing projects over many years.

Particular thanks to my commissioning editor, Debra Williams Cauley, who so successfully made the case for this book with the publisher, and who provided support and practical help as the work progressed.

Thanks also to production manager Anna Popick, who once again managed the production process so well, and to the proofreader, Audrey Doyle, who did such excellent work.

As ever, I want to thank my wife, Andrea, for her love and support.

1 An Overview in Five Examples

This chapter provides a series of five explained examples. Although the examples are tiny, each of them (apart from "Hello Who?") does something useful, and between them they provide a rapid overview of Go's key features and some of its key packages. (What other languages often call "modules" or "libraries" are called *packages* in Go terminology, and all the packages supplied with Go as standard are collectively known as the *Go standard library*.) The chapter's purpose is to provide a flavor of Go and to give a feel for the scope of what needs to be learned to program successfully in Go. Don't worry if some of the syntax or idioms are not immediately understandable; everything shown in this chapter is covered thoroughly in subsequent chapters.

Learning to program Go the Go way will take a certain amount of time and practice. For those wanting to port substantial C, C++, Java, Python, and other programs to Go, taking the time to learn Go—and in particular how its object-orientation and concurrency features work—will save time and effort in the long run. And for those wanting to create Go applications from scratch it is best to do so making the most of all that Go offers, so again the upfront investment in learning time is important—and will pay back later.

1.1. Getting Going

Go programs are compiled rather than interpreted so as to have the best possible performance. Compilation is very fast—dramatically faster than can be the case with some other languages, most notably compared with C and C++.

The Go Documentation

Go's official web site is golang.org which hosts the most up-to-date Go documentation. The "Packages" link provides access to the documentation on all the Go standard library's packages—and to their source code, which can be very helpful when the documentation itself is sparse. The "Commands" link leads to the documentation for the programs distributed with Go (e.g., the compilers, build tools, etc.). The "Specification" link leads to an accessible, informal, and quite thorough Go language specification. And the "Effective Go" link leads to a document that explains many best practices.

The web site also features a sandbox in which small (somewhat limited) Go programs can be written, compiled, and run, all online. This is useful for beginners for checking odd bits of syntax and for learning the Go fmt package's sophisticated text formatting facilities or the regexp package's regular expression engine. The Go web site's search box searches only the Go documentation; to search for Go resources generally, visit go-lang.cat-v.org/go-search.

The Go documentation can also be viewed locally, for example, in a web browser. To do this, run Go's godoc tool with a command-line argument that tells it to operate as a web server. Here's how to do this in a Unix console (xterm, gnome-terminal, konsole, Terminal.app, or similar):

```
$ godoc -http=:8000
```

Or in a Windows console (i.e., a Command Prompt or MS-DOS Prompt window):

```
C:\>godoc -http=:8000
```

The port number used here is arbitrary—simply use a different one if it conflicts with an existing server. This assumes that godoc is in your PATH.

To view the served documentation, open a web browser and give it a location of http://localhost:8000. This will present a page that looks very similar to the golang.org web site's front page. The "Packages" link will show the documentation for Go's standard library, plus any third-party packages that have been installed under GOROOT. If GOPATH is defined (e.g., for local programs and packages), a link will appear beside the "Packages" link through which the relevant documentation can be accessed. (The GOROOT and GOPATH environment variables are discussed later in this chapter and in Chapter 9.)

It is also possible to view the documentation for a whole package or a single item in a package in the console using godoc on the command line. For example, executing **godoc image NewRGBA** will output the documentation for the image.NewRGBA() function, and executing **godoc image/png** will output the documentation for the entire image/png package.

The standard Go compiler is called *gc* and its toolchain includes programs such as `5g`, `6g`, and `8g` for compiling, `5l`, `6l`, and `8l` for linking, and `godoc` for viewing the Go documentation. (These are `5g.exe`, `6l.exe`, etc., on Windows.) The strange names follow the Plan 9 operating system's compiler naming conventions where the digit identifies the processor architecture (e.g., "5" for ARM, "6" for AMD-64—including Intel 64-bit processors—and "8" for Intel 386.) Fortunately, we don't need to concern ourselves with these tools, since Go provides the high-level `go` build tool that handles the compiling and linking for us.

All the examples in this book—available from `www.qtrac.eu/gobook.html`—have been tested using *gc* on Linux, Mac OS X, and Windows using Go 1. The Go developers intend to make all subsequent Go 1.*x* versions backward compatible with Go 1, so the book's text and examples should be valid for the entire 1.*x* series. (If incompatible changes occur, the book's examples will be updated to the latest Go release, so as time goes by, they may differ from the code shown in the book.)

To download and install Go, visit `golang.org/doc/install.html` which provides instructions and download links. At the time of this writing, Go 1 is available in source and binary form for FreeBSD 7+, Linux 2.6+, Mac OS X (Snow Leopard and Lion), and Windows 2000+, in all cases for Intel 32-bit and AMD 64-bit processor architectures. There is also support for Linux on ARM processors. Go prebuilt packages are available for the Ubuntu Linux distribution, and may be available for other Linuxes by the time you read this. For learning to program in Go it is easier to install a binary version than to build Go from scratch.

Programs built with *gc* use a particular calling convention. This means that programs compiled with *gc* can be linked only to external libraries that use the same calling convention—unless a suitable tool is used to bridge the difference. Go comes with support for using external C code from Go programs in the form of the `cgo` tool (`golang.org/cmd/cgo`), and at least on Linux and BSD systems, both C and C++ code can be used in Go programs using the SWIG tool (`www.swig.org`).

In addition to *gc* there is also the `gccgo` compiler. This is a Go-specific front end to `gcc` (the GNU Compiler Collection) available for `gcc` from version 4.6. Like *gc*, `gccgo` may be available prebuilt for some Linux distributions. Instructions for building and installing `gccgo` are given at `golang.org/doc/gccgo_install.html`.

1.2. Editing, Compiling, and Running

Go programs are written as plain text Unicode using the UTF-8 encoding.* Most modern text editors can handle this automatically, and some of the most popular may even have support for Go color syntax highlighting and automatic

* Some Windows editors (e.g., Notepad) go against the Unicode standard's recommendation and insert the bytes `0xEF`, `0xBB`, `0xBF`, at the start of UTF-8 files. This book's examples assume that UTF-8 files do not have these bytes.

Go Shebang Scripts

One side effect of Go's fast compilation is that it makes it realistic to write Go programs that can be treated as shebang `#!` scripts on Unix-like systems. This requires a one-off step of installing a suitable tool. At the time of this writing, two rival tools provide the necessary functionality: `gonow` (`github.com/kless/gonow`), and `gorun` (`wiki.ubuntu.com/gorun`).

Once `gonow` or `gorun` is available, we can make any Go program into a shebang script. This is done with two simple steps. First, add either `#!/usr/bin/env gonow` or `#!/usr/bin/env gorun`, as the very first line of the `.go` file that contains the `main()` function (in package `main`). Second, make the file executable (e.g., with `chmod +x`). Such files can only be compiled by `gonow` or `gorun` rather than in the normal way since the `#!` line is not legal in Go.

When `gonow` or `gorun` executes a `.go` file for the first time, it will compile the file (extremely fast, of course), and then run it. On subsequent uses, the program will only be recompiled if the `.go` source file has been modified since the previous compilation. This makes it possible to use Go to quickly and conveniently create various small utility programs, for example, for system administration tasks.

indentation. If your editor doesn't have Go support, try entering the editor's name in the Go search engine to see if there are suitable add-ons. For editing convenience, all of Go's keywords and operators use ASCII characters; however, Go identifiers can start with any Unicode letter followed by any Unicode letters or digits, so Go programmers can freely use their native language.

To get a feel for how we edit, compile, and run a Go program we'll start with the classic "Hello World" program—although we'll make it a tiny bit more sophisticated than usual. First we will discuss compiling and running, then in the next section we will go through the source code—in file `hello/hello.go`—in detail, since it incorporates some basic Go ideas and features.

All of the book's examples are available from `www.qtrac.eu/gobook.html` and unpack to directory `goeg`. So file `hello.go`'s full path (assuming the examples were unpacked in the home directory—although anywhere will do) is `$HOME/goeg/src/hello/hello.go`. When referring to files the book always assumes the first three components of the path, which is why in this case the path is given only as `hello/hello.go`. (Windows users must, of course, read "/"s as "\"s and use the directory they unpacked the examples into, such as `C:\goeg` or `%HOMEPATH%\goeg`.)

If you have installed Go from a binary package or built it from source and installed it as root or Administrator, you should have at least one environment variable, `GOROOT`, which contains the path to the Go installation, and your `PATH` should now include `$GOROOT/bin` or `%GOROOT%\bin`. To check that Go is installed

correctly, enter the following in a console (`xterm`, `gnome-terminal`, `konsole`, `Terminal.app`, or similar):

```
$ go version
```

Or on Windows in an MS-DOS Prompt or Command Prompt window:

```
C:\>go version
```

If you get a "command not found" or "'go' is not recognized…" error message then it means that Go isn't in the `PATH`. The easiest way to solve this on Unix-like systems (including Mac OS X) is to set the environment variables in `.bashrc` (or the equivalent file for other shells). For example, the author's `.bashrc` file contains these lines:

```
export GOROOT=$HOME/opt/go
export PATH=$PATH:$GOROOT/bin
```

Naturally, you must adjust the values to match your own system. (And, of course, this is only necessary if the `go version` command fails.)

On Windows, one solution is to create a batch file that sets up the environment for Go, and to execute this every time you start a console for Go programming. However, it is much more convenient to set the environment variables once and for all through the Control Panel. To do this, click Start (the Windows logo), then Control Panel, then System and Security, then System, then Advanced system settings, and in the System Properties dialog click the Environment Variables button, then the New… button, and add a variable with the name `GOROOT` and a suitable value, such as `C:\Go`. In the same dialog, edit the `PATH` environment variable by adding the text **`;C:\Go\bin`** at the end—the leading semicolon is vital! In both cases replace the `C:\Go` path component with the actual path where Go is installed if it isn't `C:\Go`. (Again, this is only necessary if the `go version` command failed.)

From now on we will assume that Go is installed and the Go `bin` directory containing all the Go tools is in the `PATH`. (It may be necessary—once only—to open a new console window for the new settings to take effect.)

Two steps are required to build Go programs: compiling and linking.* Both of these steps are handled by the `go` tool which can not only build local programs and packages, but can also fetch, build, and install third-party programs and packages.

* Since the book assumes the use of the *gc* compiler, readers using `gccgo` will need to follow the compile and link process described in `golang.org/doc/gccgo_install.html`. Similarly, readers using other compilers will need to compile and link as per their compiler's instructions.

For the go tool to be able to build local programs and packages, there are three requirements. First, the Go bin directory ($GOROOT/bin or %GOROOT%\bin) must be in the path. Second, there must be a directory tree that has an src directory and under which the source code for the local programs and packages resides. For example, the book's examples unpack to goeg/src/hello, goeg/src/bigdigits, and so on. Third, the directory *above* the src directory must be in the GOPATH environment variable. For example, to build the book's hello example using the go tool, we must do this:

```
$ export GOPATH=$HOME/goeg
$ cd $GOPATH/src/hello
$ go build
```

We can do almost exactly the same on Windows:

```
C:\>set GOPATH=C:\goeg
C:\>cd %gopath%\src\hello
C:\goeg\src\hello>go build
```

In both cases we assume that the PATH includes $GOROOT/bin or %GOROOT%\bin. Once the go tool has built the program we can run it. By default the executable is given the same name as the directory it is in (e.g., hello on Unix-like systems and hello.exe on Windows). Once built, we can run the program in the usual way.

```
$ ./hello
Hello World!
```

Or:

```
$ ./hello Go Programmers!
Hello Go Programmers!
```

On Windows it is very similar:

```
C:\goeg\src\hello>hello Windows Go Programmers!
Hello Windows Go Programmers!
```

We have shown what must be typed in **bold** and the console's text in roman. We have also assumed a $ prompt, but it doesn't matter what it is (e.g., C:\>).

Note that we do *not* need to compile—or even explicitly link—any other packages (even though as we will see, hello.go uses three standard library packages). This is another reason why Go programs build so quickly.

If we have several Go programs, it would be convenient if all their executables could be in a single directory that we could add to our PATH. Fortunately, the go tool supports this as follows:

```
$ export GOPATH=$HOME/goeg
$ cd $GOPATH/src/hello
$ go install
```

Again, we can do the same on Windows:

```
C:\>set GOPATH=C:\goeg
C:\>cd %gopath%\src\hello
C:\goeg\src\hello>go install
```

The go install command does the same as go build only it puts the executable in a standard location ($GOPATH/bin or %GOPATH%\bin). This means that by adding a single path ($GOPATH/bin or %GOPATH%\bin) to our PATH, *all* the Go programs that we install will conveniently be in the PATH.

In addition to the book's examples, we are likely to want to develop our own Go programs and packages in our own directory. This can easily be accommodated by setting the GOPATH environment variable to two (or more) colon-separated paths (semicolon-separated on Windows); for example, export GOPATH=$HOME/app/go:$HOME/goeg or SET GOPATH=C:\app\go;C:\goeg.* In this case we must put all our program and package's source code in $HOME/app/go/src or C:\app\go\src. So, if we develop a program called myapp, its .go source files would go in $HOME/app/go/src/myapp or C:\app\go\src\myapp. And if we use go install to build a program in a GOPATH directory where the GOPATH has two or more directories, the executable will be put in the corresponding directory's bin directory.

Naturally, it would be tedious to export or set the GOPATH every time we wanted to build a Go program, so it is best to set this environment variable permanently. This can be done by setting GOPATH in the .bashrc file (or similar) on Unix-like systems (see the book's example's gopath.sh file). On Windows it can be done either by writing a batch file (see the book's example's gopath.bat file), or by adding it to the system's environment variables: Click Start (the Windows logo), then Control Panel, then System and Security, then System, then Advanced system settings, and in the System Properties dialog click the Environment Variables button, then the New... button, and add a variable with the name GOPATH and a suitable value, such as C:\goeg or C:\app\go;C:\goeg.

Although Go uses the go tool as its standard build tool, it is perfectly possible to use make or some of the modern build tools, or to use alternative Go-specific build

* From now on we will almost always show Unix-style command lines only, and assume that Windows programmers can mentally translate.

tools, or add-ons for popular IDEs (Integrated Development Environments) such as Eclipse and Visual Studio.

1.3. Hello Who?

Now that we have seen how to build the `hello` program we will look at its source code. Don't worry about understanding all the details—everything shown in this chapter (and much more!) is covered thoroughly in the subsequent chapters. Here is the complete `hello` program (in file `hello/hello.go`):

```
// hello.go
package main

import (  ❶
    "fmt"
    "os"
    "strings"
)

func main() {
    who := "World!"  ❷
    if len(os.Args) > 1 { /* os.Args[0] is "hello" or "hello.exe" */  ❸
        who = strings.Join(os.Args[1:], " ")  ❹
    }
    fmt.Println("Hello", who)  ❺
}
```

Go uses C++-style comments: `//` for single-line comments that finish at the end of the line and `/* … */` for comments that can span multiple lines. It is conventional in Go to mostly use single-line comments, with spanning comments often used for commenting out chunks of code during development.*

Every piece of Go code exists inside a package, and every Go program must have a `main` package with a `main()` function which serves as the program's entry point, that is, the function that is executed first. In fact, Go packages may also have `init()` functions that are executed before `main()`, as we will see (§1.7, ➤ 40); full details are given later (§5.6.2, ➤ 224). Notice that there is no conflict between the name of the package and the name of the function.

Go operates in terms of packages rather than files. This means that we can split a package across as many files as we like, and from Go's point of view if they all have the same package declaration, they are all part of the same package and no different than if all their contents were in a single file. Naturally, we can also

* We use some simple syntax highlighting and sometimes highlight lines or annotate them with numbers (❶, ❷, …), for ease of reference in the text. None of this is part of the Go language.

break our applications' functionality into as many local packages as we like, to keep everything neatly modularized, something we will see in Chapter 9.

The `import` statement (14 ◀, ❶) imports three packages from the standard library. The `fmt` package provides functions for formatting text and for reading formatted text (§3.5, ➤ 93), the `os` package provides platform-independent operating-system variables and functions, and the `strings` package provides functions for manipulating strings (§3.6.1, ➤ 107).

Go's fundamental types support the usual operators (e.g., `+` for numeric addition and for string concatenation), and the Go standard library supplements these by providing packages of functions for working with the fundamental types, such as the `strings` package imported here. It is also possible to create our own custom types based on the fundamental types and to provide our own methods—that is, custom type-specific functions—for them. (We will get a taste of this in §1.5, ➤ 21, with full coverage in Chapter 6.)

The reader may have noticed that the program has no semicolons, that the imports are not comma-separated, and that the `if` statement's condition does not require parentheses. In Go, blocks, including function bodies and control structure bodies (e.g., for `if` statements and for `for` loops), are delimited using braces. Indentation is used purely to improve human readability. Technically, Go statements are separated by semicolons, but these are put in by the compiler, so we don't have to use them ourselves unless we want to put multiple statements on the same line. No semicolons and fewer commas and parentheses give Go programs a lighter look and require less typing.

Go functions and methods are defined using the `func` keyword. The `main` package's `main()` function always has the same signature—it takes no arguments and returns nothing. When `main.main()` finishes the program will terminate and return `0` to the operating system. Naturally, we can exit whenever we like and return our own choice of value, as we will see (§1.4, ➤ 16).

The first statement in the `main()` function (14 ◀, ❷; using the `:=` operator) is called a *short variable declaration* in Go terminology. Such a statement both declares and initializes a variable at the same time. Furthermore, we don't need to specify the variable's type because Go can deduce that from the initializing value. So in this case we have declared a variable called `who` of type `string`, and thanks to Go's strong typing we may only assign strings to `who`.

As with most languages the `if` statement tests a condition—in this case, how many strings were entered on the command-line—which if satisfied executes the corresponding brace-delimited block. We will see a more sophisticated `if` statement syntax later in this chapter (§1.6, ➤ 29), and further on (§5.2.1, ➤ 192).

The `os.Args` variable is a *slice* of strings (14 ◀, ❸). Arrays, slices, and other collection data types are covered in Chapter 4 (§4.2, ➤ 148). For now it is sufficient

to know that a slice's length can be determined using the built-in `len()` function and its elements can be accessed using the `[]` index operator using a subset of the Python syntax. In particular, `slice[n]` returns the slice's *n*th element (counting from zero), and `slice[n:]` returns another slice which has the elements from the *n*th element to the last element. In the collections chapter we will see the full generality of Go's syntax in this area. In the case of `os.Args`, the slice should always have at least one string (the program's name), at index position `0`. (All Go indexing is `0`-based.)

If the user has entered one or more command line arguments the `if` condition is satisfied and we set the `who` string to contain all the arguments joined up as a single string (14 ◀, ❹). In this case we use the assignment operator (`=`), since if we used the short variable declaration operator (`:=`) we would end up declaring and initializing a *new* `who` variable whose scope was limited to the `if` statement's block. The `strings.Join()` function takes a slice of strings and a separator (which could be empty, i.e., `""`), and returns a single string consisting of all the slice's strings with the separator between each one. Here we have joined them using a single space between each.

Finally, in the last statement (14 ◀, ❺), we print `Hello`, a space, the string held in the `who` variable, and a newline. The `fmt` package has many different print variants, some like `fmt.Println()` which will neatly print whatever they are given, and others like `fmt.Printf()` that use placeholders to provide very fine control over formatting. The print functions are covered in Chapter 3 (§3.5, ➤ 93).

The `hello` program presented here has shown far more of the language's features than such programs conventionally do. The subsequent examples continue in this vein, covering more advanced features while keeping the examples as short as possible. The idea here is to simply acquire some basic familiarity with the language and to get to grips with building, running, and experimenting with simple Go programs, while at the same time getting a flavor of Go's powerful and novel features. And, of course, everything presented in this chapter is explained in detail in the subsequent chapters.

1.4. Big Digits—Two-Dimensional Slices

The `bigdigits` program (in file `bigdigits/bigdigits.go`) reads a number entered on the command line (as a string), and outputs the same number onto the console using "big" digits. Back in the twentieth century, at sites where lots of users shared a high-speed line printer, it used to be common practice for each user's print job to be preceded by a cover page that showed some identifying details such as their username and the name of the file being printed, using this kind of technique.

We will review the code in three parts: first the imports, then the static data, and then the processing. But right now, let's look at a sample run to get a feel for how it works:

```
$ ./bigdigits 290175493
 222    9999    000     1   77777  55555     4     9999   333
2   2  9   9   0   0   11       7  5        44    9   9  3   3
   2   9   9  0     0   1      7   5       4 4    9   9      3
  2     9999  0     0   1     7     555   4  4     9999    33
 2         9  0     0   1    7         5  444444      9      3
2          9   0   0    1   7      5   5     4        9  3   3
22222      9    000    111  7       555      4        9 333
```

Each digit is represented by a slice of strings, with all the digits together represented by a slice of slices of strings. Before looking at the data, here is how we could declare and initialize single-dimensional slices of strings and numbers:

```
longWeekend := []string{"Friday", "Saturday", "Sunday", "Monday"}
var lowPrimes = []int{2, 3, 5, 7, 11, 13, 17, 19}
```

Slices have the form `[]`*Type*, and if we want to initialize them we can immediately follow with a brace-delimited comma-separated list of elements of the corresponding type. We could have used the same variable declaration syntax for both, but have used a longer form for the `lowPrimes` slice to show the syntactic difference and for a reason that will be explained in a moment. Since a slice's *Type* can itself be a slice type we can easily create multidimensional collections (slices of slices, etc.).

The `bigdigits` program needs to import only four packages.

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

The `fmt` package provides functions for formatting text and for reading formatted text (§3.5, ➤ 93). The `log` package provides logging functions. The `os` package provides platform-independent operating-system variables and functions including the `os.Args` variable of type `[]string` (slice of strings) that holds the command-line arguments. And the `path` package's `filepath` package provides functions for manipulating filenames and paths that work across platforms. Note that for packages that are logically inside other packages, we only specify the last component of their name (in this case `filepath`) when accessing them in our code.

For the `bigdigits` program we need two-dimensional data (a slice of slices of strings). Here is how we have created it, with the strings for digit 0 laid out to illustrate how a digit's strings correspond to rows in the output, and with the strings for digits 3 to 8 elided.

```
var bigDigits = [][]string{
    {"  000  ",
     " 0   0 ",
     "0     0",
     "0     0",
     "0     0",
     " 0   0 ",
     "  000  "},
    {" 1 ", "11 ", " 1 ", " 1 ", " 1 ", " 1 ", "111"},
    {" 222 ", "2   2", "   2 ", "  2  ", " 2   ", "2    ", "22222"},
    // ... 3 to 8 ...
    {" 9999", "9   9", "9   9", " 9999", "    9", "    9", "    9"},
}
```

Variables declared outside of any function or method may not use the `:=` operator, but we can get the same effect using the long declaration form (with keyword `var`) and the assignment operator (`=`) as we have done here for the `bigDigits` variable (and did earlier for the `lowPrimes` variable). We still don't need to specify `bigDigits`' type since Go can deduce that from the assignment.

We leave the bean counting to the Go compiler, so there is no need to specify the dimensions of the slice of slices. One of Go's many conveniences is its excellent support for composite literals using braces, so we don't have to declare a data variable in one place and populate it with data in another—unless we want to, of course.

The `main()` function that reads the command line and uses the data to produce the output is only 20 lines.

```
func main() {
    if len(os.Args) == 1 {  ❶
        fmt.Printf("usage: %s <whole-number>\n", filepath.Base(os.Args[0]))
        os.Exit(1)
    }

    stringOfDigits := os.Args[1]
    for row := range bigDigits[0] {  ❷
        line := ""
        for column := range stringOfDigits {  ❸
            digit := stringOfDigits[column] - '0'  ❹
            if 0 <= digit && digit <= 9 {  ❺
```

```
                line += bigDigits[digit][row] + "  "  ❻
            } else {
                log.Fatal("invalid whole number")
            }
        }
        fmt.Println(line)
    }
}
```

The program begins by checking to see if it was invoked with any command-line arguments. If it wasn't, `len(os.Args)` will be `1` (recall that `os.Args[0]` holds the program's name, so the slice's length is normally at least 1), and the first `if` statement (18 ◀, ❶) will be satisfied. In this case we output a suitable usage message using the `fmt.Printf()` function that accepts `%` placeholders similar to those supported by the C/C++ *printf()* function or by Python's % operator. (See §3.5, ➤ 93 for full details.)

The `path/filepath` package provides path manipulation functions—for example, the `filepath.Base()` function returns the basename (i.e., the filename) of the given path. After outputting the message the program terminates using the `os.Exit()` function and returns `1` to the operating system. On Unix-like systems a return value of `0` is used to indicate success, with nonzero values indicating a usage error or a failure.

The use of the `filepath.Base()` function illustrates a nice feature of Go: When a package is imported, no matter whether it is top-level or logically inside another package (e.g., `path/filepath`), we always refer to it using only the last component of its name (e.g., `filepath`). It is also possible to give packages local names to avoid name collisions; Chapter 9 provides the details.

If at least one command-line argument was given, we copy the first one into the `stringOfDigits` variable (of type `string`). To convert the number that the user entered into big digits we must iterate over each row in the `bigDigits` slice to produce each line of output, that is, the first (top) string for each digit, then the second, and so on. We assume that all the `bigDigits`' slices have the same number of rows and so take the row count from the first one. Go's `for` loop has various syntaxes for different purposes; here (18 ◀, ❷ and 18 ◀, ❸) we have used `for … range` loops that return the index positions of each item in the slices they are given.

The row and column loops part of the code could have been written like this:

```
for row := 0; row < len(bigDigits[0]); row++ {
    line := ""
    for column := 0; column < len(stringOfDigits); column++ {
        ...
```

This is a form familiar to C, C++, and Java programmers and is perfectly valid in Go.* However, the `for … range` syntax is shorter and more convenient. (Go's `for` loops are covered in §5.3, ➤ 203.)

At each row iteration we set that row's `line` to be an empty string. Then we iterate over the columns (i.e., the characters) in the `stringOfDigits` string we received from the user. Go `string`s hold UTF-8 bytes, so potentially a character might be represented by two or more bytes. This isn't an issue here because we are only concerned with the digits `0`, `1`, …, `9` each of which is represented by a single byte in UTF-8 and with the same byte value as in 7-bit ASCII. (We will see how to iterate over a string character by character—regardless of whether the characters are single- or multibyte—in Chapter 3.)

When we index a particular position in a string we get the *byte* value at that position. (In Go the `byte` type is a synonym for the `uint8` type.) So we retrieve the byte value of the command-line string at the given column and subtract the byte value of digit 0 from it to get the number it represents (18 ◀, ❹). In UTF-8 (and 7-bit ASCII) the character `'0'` is code point (character) 48 decimal, the character `'1'` is code point 49, and so on. So if, for example, we have the character `'3'` (code point 51), we can get its integer value by doing the subtraction `'3' - '0'` (i.e., 51 – 48) which results in an integer (of type `byte`) of value 3.

Go uses single quotes for character literals, and a character literal is an integer that's compatible with any of Go's integer types. Go's strong typing means we cannot add, say, an `int32` to an `int16` without explicit conversion, but Go's numeric constants and literals adapt to their context, so in this context `'0'` is considered to be a `byte`.

If the `digit` (of type `byte`) is in range (18 ◀, ❺) we can add the appropriate string to the `line`. (In the `if` statement the constants `0` and `9` are considered to be `byte`s because that's `digit`'s type, but if `digit` was of a different type, say, `int`, they would be treated as that type instead.) Although Go strings are immutable (i.e., they cannot be changed), the `+=` append operator is supported to provide a nice easy-to-use syntax. (It works by replacing the original string under the hood.) There is also support for the `+` concatenate operator which returns a new string that is the concatenation of its left and right string operands. (The `string` type is covered fully in Chapter 3.)

To retrieve the appropriate string (19 ◀, ❻) we access the `bigDigits`'s slice that corresponds to the digit, and then within that to the row (string) we need.

If the `digit` is out of range (e.g., due to the `stringOfDigits` containing a nondigit), we call the `log.Fatal()` function with an error message. This function logs the

* Unlike C, C++, and Java, in Go the ++ and -- operators may only be used as statements, not expressions. Furthermore, they may only be used as postfix operators, not prefix operators. This means that certain order of evaluation problems cannot occur in Go—so thankfully, expressions like *f(i++)* and *a[i] = b[++i]* cannot be written in Go.

date, time, and error message—to `os.Stderr` if no other log destination is explicitly specified—and calls `os.Exit(1)` to terminate the program. There is also a `log.Fatalf()` function that does the same thing and which accepts `%` placeholders. We didn't use `log.Fatal()` in the first `if` statement (18 ◀, ❶) because we want to print the program's usage message without the date and time that the `log.Fatal()` function normally outputs.

Once all the number's strings for the given row have been accumulated the complete line is printed. In this example, seven lines are printed because each digit in the `bigDigits` slice of strings is represented by seven strings.

One final point is that the order of declarations and definitions doesn't generally matter. So in the `bigdigits/bigdigits.go` file we could declare the `bigDigits` variable before or after the `main()` function. In this case we have put `main()` first since for the book's examples we usually prefer to order things top-down.

The first two examples have covered a fair amount of ground, but both of them show material that is familiar from other mainstream languages even though the syntax is slightly different. The following three examples take us beyond the comfort zone to illustrate Go-specific features such as custom Go types, Go file handling (including error handling) and functions as values, and concurrent programming using goroutines and communication channels.

1.5. Stack—Custom Types with Methods

Although Go supports object-oriented programming it provides neither classes nor inheritance (*is-a* relationships). Go does support the creation of custom types, and Go makes aggregation (*has-a* relationships) extremely easy. Go also allows for the complete separation of a type's data from its behavior, and supports *duck typing*. Duck typing is a powerful abstraction mechanism that means that values can be handled (e.g., passed to functions), based on the methods they provide, regardless of their actual types. The terminology is derived from the phrase, "If it walks like a duck, and quacks like a duck, it *is* a duck". All of this produces a more flexible and powerful alternative to the classes and inheritance approach—but does require those of us used to the more traditional approach to make some significant conceptual adjustments to really benefit from Go's object orientation.

Go represents data using the fundamental built-in types such as `keyword!!struct` `bool`, `int`, and `string`, or by aggregations of types using `structs`.* Go's custom types are based on the fundamental types, or on `structs`, or on other custom types. (We will see some simple examples later in this chapter; §1.7, ➤ 40.)

* Unlike C++, Go's `structs` are *not* classes in disguise. For example, Go's `structs` support aggregation and delegation, but not inheritance.

Go supports both named and unnamed custom types. Unnamed types with the same structure can be used interchangeably; however, they cannot have any methods. (We will discuss this more fully in §6.4, ➤ 275.) Any *named* custom type can have methods and these methods together constitute the type's interface. Named custom types—even with the same structure—are not interchangeable. (Throughout the book any reference to a "custom type" means a *named* custom type, unless stated otherwise.)

An interface is a type that can be formally defined by specifying a particular set of methods. Interfaces are abstract and cannot be instantiated. A concrete (i.e., noninterface) type that has the methods specified by an interface fulfills the interface, that is, values of such a concrete type can be used as values of the interface's type as well as of their own actual type. Yet no formal *connection* need be established between an interface and a concrete type that provides the methods specified by the interface. It is sufficient for a custom type to have the interface's methods for it to satisfy that interface. And, of course, a type can satisfy more than one interface simply by providing all the methods for all the interfaces we want it to satisfy.

The empty interface (i.e., the interface that has no methods) is specified as `interface{}`.* Since the empty interface makes no demands at all (because it doesn't require any methods), it can stand for any value (in effect like a pointer to any value), whether the value is of a built-in type or is of a custom type. (Go's pointers and references are explained later; §4.1, ➤ 140.) Incidentally, in Go terminology we talk about types and values rather than classes and objects or instances (since Go has no classes).

Function and method parameters can be of any built-in or custom type—or of any interface type. In the latter case this means that a function can have a parameter that says, for example, "pass a value that can read data", regardless of what that value's type actually is. (We will see this in practice shortly; §1.6, ➤ 29.)

Chapter 6 covers all of these matters in detail and presents many examples to ensure that the ideas are understood. For now, let's just look at a very simple custom type—a stack—starting with how values are created and used, and then looking at the implementation of the custom type itself.

We will start with the output produced by a simple test program:

```
$ ./stacker
81.52
[pin clip needle]
-15
hay
```

* Go's empty interface can serve the same role as a reference to a Java *Object* or as C/C++'s *void**.

Each item was popped from the custom stack and printed on its own line.

The simple test program that produced this output is `stacker/stacker.go`. Here are the imports it uses:

```
import (
    "fmt"
    "stacker/stack"
)
```

The `fmt` package is part of Go's standard library, but the `stack` package is a local package specific to the `stacker` application. A Go program or package's imports are first searched for under the `GOPATH` path or paths, and then under `GOROOT`. In this particular case the program's source code is in `$HOME/goeg/src/stacker/stacker.go` and the `stack` package is in `$HOME/goeg/src/stacker/stack/stack.go`. The `go` tool will build both of them so long as the `GOPATH` is (or includes) the path `$HOME/goeg/`.

Import paths are specified using Unix-style "/"s, even on Windows. Every local package should be stored in a directory with the same name as the package. Local packages can have their own packages (e.g., like `path/filepath`), in exactly the same way as the standard library. (Creating and using custom packages is covered in Chapter 9.)

Here's the simple test program's `main()` function that produced the output:

```
func main() {
    var haystack stack.Stack
    haystack.Push("hay")
    haystack.Push(-15)
    haystack.Push([]string{"pin", "clip", "needle"})
    haystack.Push(81.52)
    for {
        item, err := haystack.Pop()
        if err != nil {
            break
        }
        fmt.Println(item)
    }
}
```

The function begins by declaring the `haystack` variable of type `stack.Stack`. It is conventional in Go to always refer to types, functions, variables, and other items in packages using the syntax *`pkg.item`*, where *`pkg`* is the last (or only) component of the package's name. This helps prevent name collisions. We then push some items onto the stack and then pop them off and print each one until there are no more left.

One amazingly convenient aspect of our custom stack is that despite Go's strong typing, we are not limited to storing homogeneous items (items all of the same type), but can freely mix heterogeneous items (items of various types). This is because the `stack.Stack` type simply stores `interface{}` items (i.e., values of *any* type) and doesn't care what their types actually are. Of course, when those items are *used,* then their type does matter. Here, though, we only use the `fmt.Println()` function and this uses Go's introspection facilities (from the `reflect` package) to discover the types of the items it is asked to print. (Reflection is covered in a later chapter; §9.4.9, ➤ 427.)

Another nice Go feature illustrated by the code is the `for` loop with no conditions. This is an infinite loop, so in most situations we will need to provide a means of breaking out of the loop—for example, using a `break` statement as here, or a `return` statement. We will see an additional `for` syntax in the next example (§1.6, ➤ 29); the complete range of `for` syntaxes is covered in Chapter 5.

Go functions and methods can return a single value or multiple values. It is conventional in Go to report errors by returning an error value (of type `error`) as the last (or only) value returned by a function or method. The custom `stack.Stack` type respects this convention.

Now that we have seen the custom `stack.Stack` type in use we are ready to review its implementation (in file `stacker/stack/stack.go`).

```
package stack

import "errors"

type Stack []interface{}
```

The file starts conventionally by specifying its package name. Then it imports other packages that it needs—in this case just one, `errors`.

When we define a named custom type in Go what we are doing is binding an identifier (the type's name) to a new type that has the same underlying representation as an existing (built-in or custom) type—and which is treated by Go as different from the underlying representation. Here, the `Stack` type is a new name for a slice (i.e., a reference to a variable-length array) of `interface{}` values—and is considered to be different from a plain `[]interface{}`.

Because all Go types satisfy the empty interface, values of any type can be stored in a `Stack`.

The built-in collection types (maps and slices), communication channels (which can be buffered), and strings, can all return their length (or buffer size) using the built-in `len()` function. Similarly, slices and channels can also report their capacity (which may be greater than the length being used) using the built-in `cap()` function. (All of Go's built-in functions are listed in Table 5.1, ➤ 187, with cross-references to where they are covered; slices are covered in Chapter 4; §4.2,

➤ 148.) It is conventional for custom collection types—our own, and those in the Go standard library—to support corresponding `Len()` and `Cap()` methods when these make sense.

Since the `Stack` type uses a slice for its underlying representation it makes sense to provide `Stack.Len()` and `Stack.Cap()` methods for it.

```
func (stack Stack) Len() int {
    return len(stack)
}
```

Both functions and methods are defined using the `func` keyword. However, in the case of methods the type of value to which the method applies is written after the `func` keyword and before the method's name, enclosed in parentheses. After the function or method's name comes a—possibly empty—parenthesized list of comma-separated parameters (each written in the form *variableName type*). After the parameters comes the function or method's opening brace (if it has no return value), or a single return value (e.g., as a type name such as the `int` returned by the `Stack.Len()` method shown here), or a parenthesized list of return values, followed by an opening brace.

In most cases a variable name for the value on which the method is called is also given—as here where we have used the name `stack` (and with no conflict with the package's name). The value on which the method is called is known in Go terminology as the *receiver*.*

In this example the type of the receiver is `Stack`, so the receiver is passed by value. This means that any changes made to the receiver would be made on a copy of the original value and in effect lost. This is no problem for methods that don't modify the receiver, such as the `Stack.Len()` method shown here.

The `Stack.Cap()` method is almost identical to the `Stack.Len()` method (and so is not shown). The only difference is that the `Stack.Cap()` method returns the `cap()` rather than the `len()` of the receiver `stack`. The source code also includes a `Stack.IsEmpty()` method, but this is so similar to `Stack.Len()`—it just returns a `bool` indicating whether the stack's `len()` equals 0—that again it isn't shown.

```
func (stack *Stack) Push(x interface{}) {
    *stack = append(*stack, x)
}
```

The `Stack.Push()` method is called on a pointer to a `Stack` (explained in a moment), and is passed a value (`x`) of any type. The built-in `append()` function takes a slice and one or more values and returns a (possibly new) slice which has the

* In other languages the receiver is typically called *this* or *self*; using such names works fine in Go, but is not considered to be good Go style.

original slice's contents, plus the given value or values as its last element or elements. (See §4.2.3, ➤ 156.)

If the stack has previously had items popped from it (➤ 28), the underlying slice's capacity is likely to be greater than its length, so the push could be very cheap: simply a matter of putting the `x` item into the `len(stack)` position and increasing the stack's length by one.

The `Stack.Push()` method always works (unless the computer runs out of memory), so we don't need to return an `error` value to indicate success or failure.

If we want to modify a value we must make its receiver a pointer.* A *pointer* is a variable that holds the memory address of another value. One reason that pointers are used is for efficiency—for example, if we have a value of a large type it is much cheaper to pass a pointer to the value as a parameter than to pass the value itself. Another use is to make a value modifiable. For example, when a variable is passed into a function the function gets a *copy* of the value (e.g., the `stack` passed into the `stack.Len()` function; 25 ◀). This means that if we make any changes to the variable inside the function, they will have no effect on the original value. If we need to modify the original value—as here where we want to append to the stack—we must pass a pointer to the original value, and then inside the function we can modify the value that the pointer points to.

A pointer is declared by preceding the type name with a star (i.e., an asterisk, `*`). So here, in the `Stack.Push()` method, the `stack` variable is of type `*Stack`, that is, the `stack` variable holds a pointer to a `Stack` value and not an actual `Stack` value. We can access the actual `Stack` value that the pointer points to by *dereferencing* the pointer—this simply means that we access the value the pointer points to. Dereferencing is done by preceding the variable name with a star. So here, when we write `stack` we are referring to a pointer to a `Stack` (i.e., to a `*Stack`), and when we write `*stack` we are dereferencing the pointer, that is, referring to the actual `Stack` that the pointer points to.

So, in Go (and C and C++ for that matter), the star is overloaded to mean multiplication (when between a pair of numbers or variables, e.g., `x * y`), pointer declaration (when preceding a type name, e.g., `z *MyType`), and pointer dereference (when preceding a pointer variable's name, e.g., `*z`). Don't worry too much about these matters for now: Go's pointers are fully explained in Chapter 4.

Note that Go's channels, maps, and slices are all created using the `make()` function, and `make()` always returns a *reference* to the value it created. References behave very much like pointers in that when they are passed to functions any changes made to them inside the function affect the original channel, map, or slice. However, references don't need to be dereferenced, so in most cases there's no need to use stars with them. But if we want to modify a slice inside a func-

* Go pointers are essentially the same as in C and C++ except that pointer arithmetic isn't supported —or necessary; see §4.1, ➤ 140.

tion or method using `append()` (as opposed to simply changing one of its existing items), then we must either pass the slice by pointer, or return the slice (and set the original slice to the function or method's return value), since `append()` sometimes returns a different slice reference than the one it was passed.

The `Stack` type uses a slice for its representation and therefore `Stack` values can be used with functions that operate on a slice, such as `append()` and `len()`. Nonetheless, `Stack` values are values in their own right, distinct from their representation, so they must be passed by pointer if we want to modify them.

```
func (stack Stack) Top() (interface{}, error) {
    if len(stack) == 0 {
        return nil, errors.New("can't Top() an empty stack")
    }
    return stack[len(stack)-1], nil
}
```

The `Stack.Top()` method returns the item at the top of the stack (the item that was added last) and a `nil` error value; or a `nil` item and a non-`nil` error value, if the stack is empty. The `stack` receiver is passed by value since the stack isn't modified.

The `error` type is an `interface` type (§6.3, ➤ 265) which specifies a single method, `Error() string`. In general, Go's library functions return an `error` as their last (or only) return value to indicate success (where `error` is `nil`) or failure. Here, we have made our `Stack` type work like a standard library type by creating a new `error` value using the `errors` package's `errors.New()` function.

Go uses `nil` for zero pointers (and for zero references); that is, for pointers that point to nothing and for references that refer to nothing.* Such pointers should be used only in conditions or assignments; methods should not normally be called on them.

Constructors are never called implicitly in Go. Instead Go guarantees that when a value is created it is always initialized to its zero value. For example, numbers are initialized to 0, strings to the empty string, pointers to `nil`, and the fields inside `structs` are similarly initialized. So there is no uninitialized data in Go, thus eliminating a major source of errors that afflicts many other programming languages. If the zero value isn't suitable we can write a construction function—and call it explicitly—as we do here to create a new `error`. It is also possible to prevent values of a type being created without using a constructor function, as we will see in Chapter 6.

* Go's `nil` is in effect the same as *NULL* or *0* in C and C++, *null* in Java, and *nil* in Objective-C.

If the stack is nonempty we return its topmost value and a `nil` error value. Since Go uses 0-based indexing the first element in a slice or array is at position 0 and the last element is at position `len(`*`sliceOrArray`*`) - 1`.

There is no formality when returning more than one value from a function or method; we simply list the types we are returning after the function or method's name and ensure that we have at least one `return` statement that has a corresponding list of values.

```
func (stack *Stack) Pop() (interface{}, error) {
    theStack := *stack
    if len(theStack) == 0 {
        return nil, errors.New("can't Pop() an empty stack")
    }
    x := theStack[len(theStack)-1]  ❶
    *stack = theStack[:len(theStack)-1]  ❷
    return x, nil
}
```

The `Stack.Pop()` method is used to remove and return the top (last added) item from the stack. Like the `Stack.Top()` method it returns the item and a `nil` error, or if the stack is empty, a `nil` item and a non-`nil` error.

The method must have a receiver that is a pointer since it modifies the stack by removing the returned item. For syntactic convenience, rather than referring to `*stack` (the actual stack that the `stack` variable points to) throughout the method, we assign the actual stack to a local variable (`theStack`), and work with that variable instead. This is quite cheap, because `*stack` is pointing to a `Stack`, which uses a slice for its representation, so we are really assigning little more than a reference to a slice.

If the stack is empty we return a suitable error. Otherwise we retrieve the stack's top (last) item and store it in a local variable (`x`). Then we take a slice of the stack (which itself is a slice). The new slice has one less element than the original and is immediately set to be the value that the `stack` pointer points to. And at the end, we return the retrieved value and a `nil` error. We can reasonably expect any decent Go compiler to reuse the slice, simply reducing the slice's length by one, while leaving its capacity unchanged, rather than copying all the data to a new slice.

The item to return is retrieved using the `[]` index operator with a single index (❶); in this case the index of the slice's last element.

The new slice is obtained by using the `[]` slice operator with an index range (❷). An index range has the form *`first`*`:`*`end`*. If *`first`* is omitted—as here—0 is assumed, and if *`end`* is omitted, the `len()` of the slice is assumed. The slice thus obtained has elements with indexes from and including the *`first`* up to and

excluding the *end*. So in this case, by specifying the last index as one less than the length, we slice up to the last but one element, effectively removing the last element from the slice. (Slice indexing is covered in Chapter 4, §4.2.1, ➤ 153.)

In this example we used `Stack` receivers rather than pointers (i.e., of type `*Stack`) for those methods that don't modify the `Stack`. For custom types with lightweight representations (say, a few `int`s or `string`s), this is perfectly reasonable. But for heavyweight custom types it is usually best to always use pointer receivers since a pointer is much cheaper to pass (typically a simple 32- or 64-bit value), than a large value, even for methods where the value isn't modified.

A subtle point to note regarding pointers and methods is that if we call a method on a value, and the method requires a pointer to the value it is called on, Go is smart enough to pass the value's address rather than a copy of the value (providing the value is addressable; §6.2.1, ➤ 258). Correspondingly, if we call a method on a pointer to a value, and the method requires a value, Go is smart enough to dereference the pointer and give the method the pointed-to value.*

As this example illustrates, creating custom types in Go is generally straightforward, and doesn't involve the cumbersome formalities that many other languages demand. Go's object-oriented features are covered fully in Chapter 6.

1.6. Americanise—Files, Maps, and Closures

To have any practical use a programming language must provide some means of reading and writing external data. In previous sections we had a glimpse of Go's versatile and powerful print functions from its `fmt` package; in this section we will look at Go's basic file handling facilities. We will also look at some more advanced features such as Go's treatment of functions and methods as first-class values which makes it possible to pass them as parameters. And in addition we will make use of Go's `map` type (also known as a data dictionary or hash).

This section provides enough of the basics so that programs that read and write text files can be written—thus making the examples and exercises more interesting. Chapter 8 provides much more coverage of Go's file handling facilities.

By about the middle of the twentieth century, American English surpassed British English as the most widely used form of English. In this section's example we will review a program that reads a text file and writes out a copy of the file into a new file with any words using British spellings replaced with their U.S. counterparts. (This doesn't help with differences in semantics or idioms, of course.) The program is in the file `americanise/americanise.go`, and we will review it top-down, starting with its imports, then its `main()` function, then the functions that `main()` calls, and so on.

* This is why Go does not have or need the -> indirection operator used by C and C++.

```
import (
    "bufio"
    "fmt"
    "io"
    "io/ioutil"
    "log"
    "os"
    "path/filepath"
    "regexp"
    "strings"
)
```

All the `americanise` program's imports are from Go's standard library. Packages can be nested inside one another without formality, as the `io` package's `ioutil` package and the `path` package's `filepath` package illustrate.

The `bufio` package provides functions for buffered I/O, including ones for reading and writing strings from and to UTF-8 encoded text files. The `io` package provides low-level I/O functions—and the `io.Reader` and `io.Writer` interfaces we need for the `americanise()` program. The `io/ioutil` package provides high-level file handling functions. The `regexp` package provides powerful regular expression support. The other packages (`fmt`, `log`, `filepath`, and `strings`) have been mentioned in earlier sections.

```
func main() {
    inFilename, outFilename, err := filenamesFromCommandLine()  ❶
    if err != nil {
        fmt.Println(err)  ❷
        os.Exit(1)
    }
    inFile, outFile := os.Stdin, os.Stdout  ❸
    if inFilename != "" {
        if inFile, err = os.Open(inFilename); err != nil {
            log.Fatal(err)
        }
        defer inFile.Close()  ❹
    }
    if outFilename != "" {
        if outFile, err = os.Create(outFilename); err != nil {
            log.Fatal(err)
        }
        defer outFile.Close()  ❺
    }

    if err = americanise(inFile, outFile); err != nil {
        log.Fatal(err)
```

```
    }
}
```

The `main()` function gets the input and output filenames from the command line, creates corresponding file values, and then passes the files to the `americanise()` function to do the work.

The function begins by retrieving the names of the files to read and write and an `error` value. If there was a problem parsing the command line we print the error (which contains the program's usage message), and terminate the program. Some of Go's print functions use reflection (introspection) to print a value using the value's `Error() string` method if it has one, or its `String() string` method if it has one, or as best they can otherwise. If we provide our own custom types with one of these methods, Go's print functions will automatically be able to print values of our custom types, as we will see in Chapter 6.

If `err` is `nil`, we have `inFilename` and `outFilename` strings (which may be empty), and we can continue. Files in Go are represented by pointers to values of type `os.File`, and so we create two such variables initialized to the standard input and output streams (which are both of type `*os.File`). Since Go functions and methods can return multiple values it follows that Go supports multiple assignments such as the ones we have used here (30 ◀, ❶, ❸).

Each filename is handled in essentially the same way. If the filename is empty the file has already been correctly set to `os.Stdin` or `os.Stdout` (both of which are of type `*os.File`, i.e., a pointer to an `os.File` value representing the file); but if the filename is nonempty we create a new `*os.File` to read from or write to the file as appropriate.

The `os.Open()` function takes a filename and returns an `*os.File` value that can be used for reading the file. Correspondingly, the `os.Create()` function takes a filename and returns an `*os.File` value that can be used for reading or writing the file, creating the file if it doesn't exist and truncating it to zero length if it does exist. (Go also provides the `os.OpenFile()` function that can be used to exercise complete control over the mode and permissions used to open a file.)

In fact, the `os.Open()`, `os.Create()`, and `os.OpenFile()` functions return two values: an `*os.File` and `nil` if the file was opened successfully, or `nil` and an `error` if an error occurred.

If `err` is `nil` we know that the file was successfully opened so we immediately execute a `defer` statement to close the file. Any function that is the subject of a `defer` statement (§5.5, ➤ 212) must be called—hence the parentheses after the functions' names (30 ◀, ❹, ❺)—but the calls only actually occur when the function in which the `defer` statements are written returns. So the `defer` statement "captures" the function call and sets it aside for later. This means that the `defer` statement itself takes almost no time at all and control immediately passes to the following statement. Thus, the deferred `os.File.Close()` method won't

actually be called until the enclosing function—in this case, `main()`—returns (whether normally or due to a *panic,* discussed in a moment), so the file is open to be worked on and yet guaranteed to be closed when we are finished with it, or if a panic occurs.

If we fail to open the file we call `log.Fatal()` with the error. As we noted in a previous section, this function logs the date, time, and error (to `os.Stderr` unless another log destination is specified), and calls `os.Exit()` to terminate the program. When `os.Exit()` is called (directly, or by `log.Fatal()`), the program is terminated immediately—and any pending deferred statements are lost. This is not a problem, though, since Go's runtime system will close any open files, the garbage collector will release the program's memory, and any decent database or network that the application might have been talking to will detect the application's demise and respond gracefully. Just the same as with the `bigdigits` example, we don't use `log.Fatal()` in the first `if` statement (30 ◀, ❷), because the `err` contains the program's usage message and we want to print this without the date and time that the `log.Fatal()` function normally outputs.

In Go a *panic* is a runtime error (rather like an exception in other languages). We can cause panics ourselves using the built-in `panic()` function, and can stop a panic in its tracks using the `recover()` function (§5.5, ➤ 212). In theory, Go's panic/recover functionality can be used to provide a general-purpose exception handling mechanism—but doing so is considered to be poor Go practice. The Go way to handle errors is for functions and methods to return an `error` value as their sole or last return value—or `nil` if no error occurred—and for callers to always check the error they receive. The purpose of panic/recover is to deal with genuinely exceptional (i.e., unexpected) problems and *not* with normal errors.*

With both files successfully opened (the `os.Stdin`, `os.Stdout`, and `os.Stderr` files are automatically opened by the Go runtime sytem), we call the `americanise()` function to do the processing, passing it the files on which to work. If `americanise()` returns `nil` the `main()` function terminates normally and any deferred statements—in this case, ones that close the `inFile` and `outFile` if they are not `os.Stdin` and `os.Stdout`—are executed. And if `err` is not `nil`, the error is printed, the program is exited, and Go's runtime system closes any open files.

The `americanise()` function accepts an `io.Reader` and an `io.Writer`, not `*os.Files`, but this doesn't matter since the `os.File` type supports the `io.ReadWriter` interface (which simply aggregates the `io.Reader` and `io.Writer` interfaces) and can therefore be used wherever an `io.Reader` or an `io.Writer` is required. This is an example of duck typing in action—the `americanise()` function's parameters are interfaces, so the function will accept any values—no matter what their types—that satisfy the interfaces, that is, any values that have the methods the

* Go's approach is very different from C++, Java, and Python, where exception handling is often used for both errors and exceptions. The discussion and rationale for Go's panic/recover mechanism is at `https://groups.google.com/group/golang-nuts/browse_thread/thread/1ce5cd050bb973e4?pli=1`.

interfaces specify. The `americanise()` function returns `nil`, or an `error` if an error occurred.

```
func filenamesFromCommandLine() (inFilename, outFilename string,
    err error) {
    if len(os.Args) > 1 && (os.Args[1] == "-h" || os.Args[1] == "--help") {
        err = fmt.Errorf("usage: %s [<]infile.txt [>]outfile.txt",
            filepath.Base(os.Args[0]))
        return "", "", err
    }
    if len(os.Args) > 1 {
        inFilename = os.Args[1]
        if len(os.Args) > 2 {
            outFilename = os.Args[2]
        }
    }
    if inFilename != "" && inFilename == outFilename {
        log.Fatal("won't overwrite the infile")
    }
    return inFilename, outFilename, nil
}
```

The `filenamesFromCommandLine()` function returns two strings and an `error` value—and unlike the functions we have seen so far, here the return values are given variable names, not just types. Return variables are set to their zero values (empty strings and `nil` for `err` in this case) when the function is entered, and keep their zero values unless explicitly assigned to in the body of the function. (We will say a bit more on this topic when we discuss the `americanise()` function, next.)

The function begins by seeing if the user has asked for usage help.* If they have, we create a new `error` value using the `fmt.Errorf()` function with a suitable usage string, and return immediately. As usual with Go code, the caller is expected to check the returned `error` and behave accordingly (and this is exactly what `main()` does). The `fmt.Errorf()` function is like the `fmt.Printf()` function we saw earlier, except that it returns an `error` value containing a string using the given format string and arguments rather than writing a string to `os.Stdout`. (The `errors.New()` function is used to create an `error` given a literal string.)

If the user did not request usage information we check to see if they entered any command-line arguments, and if they did we set the `inFilename` return variable to their first command-line argument and the `outFilename` return variable

* The Go standard library includes a `flag` package for handling command-line arguments. Third-party packages for GNU-compatible command-line handling are available from `godashboard.appspot.com/project`. (Using third-party packages is covered in Chapter 9.)

to their second command-line argument. Of course, they may have given no command-line arguments, in which case both `inFilename` and `outFilename` remain empty strings; or they may have entered just one, in which case `inFilename` will have a filename and `outFilename` will be empty.

At the end we do a simple sanity check to make sure that the user doesn't overwrite the input file with the output file, exiting if necessary—but if all is well, we return.* Functions or methods that return one or more values *must* have at least one `return` statement. It can be useful for clarity, and for `godoc`-generated documentation, to give variable names for return types, as we have done in this function. If a function or method has variable names as well as types listed for its return values, then a bare `return` is legal (i.e., a `return` statement that does not specify any variables). In such cases, the listed variables' values are returned. We do not use bare `returns` in this book because they are considered to be poor Go style.

Go takes a consistent approach to reading and writing data that allows us to read and write to files, to buffers (e.g., to slices of bytes or to strings), and to the standard input, output, and error streams—or to our own custom types—so long as they provide the methods necessary to satisfy the reading and writing interfaces.

For a value to be readable it must satisfy the `io.Reader` interface. This interface specifies a single method with signature, `Read([]byte) (int, error)`. The `Read()` method reads data from the value it is called on and puts the data read into the given byte slice. It returns the number of bytes read and an `error` value which will be `nil` if no error occurred, or `io.EOF` ("end of file") if no error occurred and the end of the input was reached, or some other non-`nil` value if an error occurred. Similarly, for a value to be writable it must satisfy the `io.Writer` interface. This interface specifies a single method with signature, `Write([]byte) (int, error)`. The `Write()` method writes data from the given byte slice into the value the method was called on, and returns the number of bytes written and an `error` value (which will be `nil` if no error occurred).

The `io` package provides readers and writers but these are unbuffered and operate in terms of raw bytes. The `bufio` package provides buffered input/output where the input will work on any value that satisfies the `io.Reader` interface (i.e., provides a suitable `Read()` method), and the output will work on any value that satisfies the `io.Writer` interface (i.e., provides a suitable `Write()` method). The `bufio` package's readers and writers provide buffering and can work in terms of bytes or strings, and so are ideal for reading and writing UTF-8 encoded text files.

* In fact, the user could still overwrite the input file by using redirection—for example, `$ ./americanise infile > infile`—but at least we have prevented an obvious accident.

```
var britishAmerican = "british-american.txt"

func americanise(inFile io.Reader, outFile io.Writer) (err error) {
    reader := bufio.NewReader(inFile)
    writer := bufio.NewWriter(outFile)
    defer func() {
        if err == nil {
            err = writer.Flush()
        }
    }()

    var replacer func(string) string  ❶
    if replacer, err = makeReplacerFunction(britishAmerican); err != nil {
        return err
    }
    wordRx := regexp.MustCompile("[A-Za-z]+")
    eof := false
    for !eof {
        var line string  ❷
        line, err = reader.ReadString('\n')
        if err == io.EOF {
            err = nil   // io.EOF isn't really an error
            eof = true  // this will end the loop at the next iteration
        } else if err != nil {
            return err  // finish immediately for real errors
        }
        line = wordRx.ReplaceAllStringFunc(line, replacer)
        if _, err = writer.WriteString(line); err != nil {  ❸
            return err
        }
    }
    return nil
}
```

The `americanise()` function buffers the `inFile` reader and the `outFile` writer. Then it reads lines from the buffered reader and writes each line to the buffered writer, having replaced any British English words with their U.S. equivalents.

The function begins by creating a buffered reader and a buffered writer through which their contents can be accessed as bytes—or more conveniently in this case, as strings. The `bufio.NewReader()` construction function takes as argument any value that satisfies the `io.Reader` interface (i.e., any value that has a suitable `Read()` method) and returns a new buffered `io.Reader` that reads from the given reader. The `bufio.NewWriter()` function is synonymous. Notice that the `americanise()` function doesn't know or care what it is reading from or writing to—the reader and writer could be compressed files, network connections, byte slices

(`[]byte`), or anything else that supports the `io.Reader` and `io.Writer` interfaces. This way of working with interfaces is very flexible and makes it easy to compose functionality in Go.

Next we create an anonymous deferred function that will flush the writer's buffer before the `americanise()` function returns control to its caller. The anonymous function will be called when `americanise()` returns normally—or abnormally due to a panic. If no error has occurred and the writer's buffer contains unwritten bytes, the bytes will be written before `americanise()` returns. Since it is possible that the flush will fail we set the `err` return value to the result of the `writer.Flush()` call. A less defensive approach would be to have a much simpler `defer` statement of `defer writer.Flush()` to ensure that the writer is flushed before the function returns and ignoring any error that might have occurred before the flush—or that occurs during the flush.

Go allows the use of named return values, and we have taken advantage of this facility here (`err error`), just as we did previously in the `filenamesFromCommandLine()` function. Be aware, however, that there is a subtle scoping issue we must consider when using named return values. For example, if we have a named return value of *`value`*, we can assign to it anywhere in the function using the assignment operator (`=`) as we'd expect. However, if we have a statement such as `if` *`value`* `:= …`, because the `if` statement starts a new block, the *`value`* in the `if` statement will be a new variable, so the `if` statement's *`value`* variable will shadow the return *`value`* variable. In the `americanise()` function, `err` is a named return value, so we have made sure that we never assign to it using the short variable declaration operator (`:=`) to avoid the risk of accidentally creating a shadow variable. One consequence of this is that we must declare the other variables we want to assign to at the same time, such as the replacer function (35 ◀, ❶) and the line we read in (35 ◀, ❷). An alternative approach is to avoid named return values and return the required value or values explicitly, as we have done elsewhere.

One other small point to note is that we have used the *blank identifier*, `_` (35 ◀, ❸). The blank identifier serves as a placeholder for where a variable is expected in an assignment, and discards any value it is given. The blank identifier is not considered to be a new variable, so if used with `:=`, at least one other (new) variable must be assigned to.

The Go standard library contains a powerful regular expression package called `regexp` (§3.6.5, ➤ 120). This package can be used to create pointers to `regexp.Regexp` values (i.e., of type `*regexp.Regexp`). These values provide many methods for searching and replacing. Here we have chosen to use the `regexp.Regexp.ReplaceAllStringFunc()` method which given a string and a "replacer" function with signature `func(string) string`, calls the replacer function for every match, passing in the matched text, and replacing the matched text with the text the replacer function returns.

If we had a very small replacer function, say, one that simply uppercased the words it matched, we could have created it as an anonymous function when we called the replacement function. For example:

```
line = wordRx.ReplaceAllStringFunc(line,
    func(word string) string { return strings.ToUpper(word) })
```

However, the `americanise` program's replacer function, although only a few lines long, requires some preparation, so we have created another function, `makeReplacerFunction()`, that given the name of a file that contains lines of original and replacement words, returns a replacer function that will perform the appropriate replacements.

If the `makeReplacerFunction()` returns a non-`nil` error, we return and the caller is expected to check the returned `error` and respond appropriately (as it does).

Regular expressions can be compiled using the `regexp.Compile()` function which returns a `*regexp.Regexp` and `nil`, or `nil` and `error` if the regular expression is invalid. This is ideal for when the regular expression is read from an external source such as a file or received from the user. Here, though, we have used the `regexp.MustCompile()` function—this simply returns a `*regexp.Regexp`, or panics if the regular expression, or "regexp", is invalid. The regular expression used in the example matches the longest possible sequence of one or more English alphabetic characters.

With the replacer function and the regular expression in place we start an infinite loop that begins by reading a line from the reader. The `bufio.Reader.ReadString()` method reads (or, strictly speaking, *decodes*) the underlying reader's raw bytes as UTF-8 encoded text (which also works for 7-bit ASCII) up to and including the specified byte (or up to the end of the file). The function conveniently returns the text as a `string`, along with an `error` (or `nil`).

If the `error` returned by the call to the `bufio.Reader.ReadString()` method is not `nil`, either we have reached the end of the input or we have hit a problem. At the end of the input `err` will be `io.EOF` which is perfectly okay, so in this case we set `err` to `nil` (since there isn't really an error), and set `eof` to `true` to ensure that the loop finishes at the next iteration, so we won't attempt to read beyond the end of the file. We don't return immediately we get `io.EOF`, since it is possible that the file's last line doesn't end with a newline, in which case we will have received a line to be processed, in addition to the `io.EOF` error.

For each line we call the `regexp.Regexp.ReplaceAllStringFunc()` method, giving it the line and the replacer function. We then try to write the (possibly modified) line to the writer using the `bufio.Writer.WriteString()` method—this method accepts a `string` and writes it out as a sequence of UTF-8 encoded bytes, returning the number of bytes written and an `error` (which will be `nil` if no error occurred). We don't care how many bytes are written so we assign the number to the blank

identifier, _. If err is not nil we return immediately, and the caller will receive the error.

Using bufio's reader and writer as we have done here means that we can work with convenient high level string values, completely insulated from the raw bytes which represent the text on disk. And, of course, thanks to our deferred anonymous function, we know that any buffered bytes are written to the writer when the americanise() function returns, providing that no error has occurred.

```
func makeReplacerFunction(file string) (func(string) string, error) {
    rawBytes, err := ioutil.ReadFile(file)
    if err != nil {
        return nil, err
    }
    text := string(rawBytes)

    usForBritish := make(map[string]string)
    lines := strings.Split(text, "\n")
    for _, line := range lines {
        fields := strings.Fields(line)
        if len(fields) == 2 {
            usForBritish[fields[0]] = fields[1]
        }
    }

    return func(word string) string {
        if usWord, found := usForBritish[word]; found {
            return usWord
        }
        return word
    }, nil
}
```

The makeReplacerFunction() takes the name of a file containing original and replacement strings and returns a function that given an original string returns its replacement, along with an error value. It expects the file to be a UTF-8 encoded text file with one whitespace-separated original and replacement word per line.

In addition to the bufio package's readers and writers, Go's io/ioutil package provides some high level convenience functions including the ioutil.ReadFile() function used here. This function reads and returns the entire file's contents as raw bytes (in a []byte) and an error. As usual, if the error is not nil we immediately return it to the caller—along with a nil replacer function. If we read the bytes okay, we convert them to a string using a Go conversion of form *type*(*variable*). Converting UTF-8 bytes to a string is very cheap since Go's strings use the UTF-8 encoding internally. (Go's string conversions are covered in Chapter 3.)

The replacer function we want to create must accept a string and return a corresponding string, so what we need is a function that uses some kind of lookup table. Go's built-in `map` collection data type is ideal for this purpose (§4.3, ➤ 164). A `map` holds *key–value* pairs with very fast lookup by *key*. So here we will store British words as keys and their U.S. counterparts as values.

Go's map, slice, and channel types are created using the built-in `make()` function. This creates a value of the specified type and returns a reference to it. The reference can be passed around (e.g., to other functions) and any changes made to the referred-to value are visible to all the code that accesses it. Here we have created an empty map called `usForBritish`, with string keys and string values.

With the map in place we then split the file's text (which is in the form of a single long string) into lines, using the `strings.Split()` function. This function takes a string to split and a separator string to split on and does as many splits as possible. (If we want to limit the number of splits we can use the `strings.SplitN()` function.)

The iteration over the lines uses a `for` loop syntax that we haven't seen before, this time using a `range` clause. This form can be conveniently used to iterate over a map's keys and values, over a communication channel's elements, or—as here—over a slice's (or array's) elements. When used on a slice (or array), the slice index and the element at that index are returned on each iteration, starting at index 0 (if the slice is nonempty). In this example we use the loop to iterate over all the lines, but since we don't care about the index of each line we assign it to the blank identifier (`_`) which discards it.

We need to split each line into two: the original string and the replacement string. We could use the `strings.Split()` function but that would require us to specify an exact separator string, say, `" "`, which might fail on a hand-edited file where sometimes users accidentally put in more than one space, or sometimes use tabs. Fortunately, Go provides the `strings.Fields()` function which splits the string it is given on whitespace and is therefore much more forgiving of human-edited text.

If the `fields` variable (of type `[]string`) has exactly two elements we insert the corresponding *key–value* pair into the map. Once the map is populated we are ready to create the replacer function that we will return to the caller.

We create the replacer function as an anonymous function given as an argument to the `return` statement—along with a `nil error` value. (Of course, we could have been less succinct and assigned the anonymous function to a variable and returned the variable.) The function has the exact signature required by the `regexp.Regexp.ReplaceAllStringFunc()` method that it will be passed to.

Inside the anonymous replacer function all we do is look up the given word. If we access a map element with one variable on the left-hand side, that variable is set to the corresponding value—or to the value type's zero value if the given

key isn't in the map. If the map value type's zero value is a legitimate value, then how can we tell if a given key is in the map? Go provides a syntax for this case—and that is generally useful if we simply want to know whether a particular key is in the map—which is to put two variables on the left-hand side, the first to accept the value and the second to accept a `bool` indicating if the key was found. In this example we use this second form inside an `if` statement that has a simple statement (a short variable declaration), and a condition (the `found` Boolean). So we retrieve the `usWord` (which will be an empty string if the given word isn't a key in the map), and a `found` flag of type `bool`. If the British word was found we return the U.S. equivalent; otherwise we simply return the original word unchanged.

There is a subtlety in the `makeReplacerFunction()` function that may not be immediately apparent. In the anonymous function created inside it we access the `usForBritish` map, yet this map was created outside the anonymous function. This works because Go supports *closures* (§5.6.3, ➤ 225). A closure is a function that "captures" some external state—for example, the state of the function it is created inside, or at least any part of that state that the closure accesses. So here, the anonymous function that is created inside the `makeReplacerFunction()` is a closure that has captured the `usForBritish` map.

Another subtlety is that the `usForBritish` map is a local variable and yet we will be accessing it outside the function in which it is declared. It is perfectly fine to return local variables in Go. Even if they are references or pointers, Go won't delete them while they are in use and will garbage-collect them when they are finished with (i.e., when every variable that holds, refers, or points to them has gone out of scope).

This section has shown some basic low-level and high-level file handling functionality using `os.Open()`, `os.Create()`, and `ioutil.ReadFile()`. In Chapter 8 there is much more file handling coverage, including the writing and reading of text, binary, JSON, and XML files. Go's built-in collection types—slices and maps—largely obviate the need for custom collection types while providing extremely good performance and great convenience. Go's collection types are covered in Chapter 4. Go's treatment of functions as first-class values in their own right and its suppport for closures makes it possible to use some advanced and very useful programming idioms. And Go's `defer` statement makes it straightforward to avoid resource leakage.

1.7. Polar to Cartesian—Concurrency

One key aspect of the Go language is its ability to take advantage of modern computers with multiple processors and multiple cores, and to do so without burdening programmers with lots of bookkeeping. Many concurrent Go programs can be written without any explicit locking at all (although Go does have locking

primitives for when they're needed in lower-level code, as we will see in Chapter 7).

Two features make concurrent programming in Go a pleasure. First, *goroutines* (in effect very lightweight threads/coroutines) can easily be created at will without the need to subclass some "thread" class (which isn't possible in Go anyway). Second, *channels* provide type-safe one-way or two-way communication with goroutines and which can be used to synchronize goroutines.

The Go way to do concurrency is to *communicate* data, not to share data. This makes it much easier to write concurrent programs than using the traditional threads and locks approach, since with no shared data we can't get race conditions (such as deadlocks), and we don't have to remember to lock or unlock since there is no shared data to protect.

In this section we will look at the fifth and last of the chapter's "overview" examples. This section's example program uses two communication channels and does its processing in a separate Go routine. For such a small program this is complete overkill, but the point is to illustrate a basic use of these Go features in as clear and short a way as possible. More realistic concurrency examples that show many of the different techniques that can be used with Go's channels and goroutines are presented in Chapter 7.

The program we will review is called `polar2cartesian`; it is an interactive console program that prompts the user to enter two whitespace-separated numbers—a radius and an angle—which the program then uses to compute the equivalent cartesian coordinates. In addition to illustrating one particular approach to concurrency, it also shows some simple `structs` and how to determine if the program is running on a Unix-like system or on Windows for when the difference matters. Here is an example of the program running in a Linux console:

```
$ ./polar2cartesian
Enter a radius and an angle (in degrees), e.g., 12.5 90, or Ctrl+D to quit.
Radius and angle: 5 30.5
Polar radius=5.00 θ=30.50° → Cartesian x=4.31 y=2.54
Radius and angle: 5 -30.25
Polar radius=5.00 θ=-30.25° → Cartesian x=4.32 y=-2.52
Radius and angle: 1.0 90
Polar radius=1.00 θ=90.00° → Cartesian x=-0.00 y=1.00
Radius and angle: ^D
$
```

The program is in file `polar2cartesian/polar2cartesian.go`, and we will review it top-down, starting with the imports, then the `structs` it uses, then its `init()` function, then its `main()` function, and then the functions called by `main()`, and so on.

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

The `polar2cartesian` program imports several packages, some of which have been mentioned in earlier sections, so we will only mention the new ones here. The `math` package provides mathematical functions for operating on floating-point numbers (§2.3.2, ➤ 64) and the `runtime` package provides functions that access the program's runtime properties, such as which platform the program is running on.

```
type polar struct {
    radius float64
    θ      float64
}

type cartesian struct {
    x   float64
    y   float64
}
```

In Go a `struct` is a type that holds (aggregates or embeds) one or more data fields. These fields can be built-in types as here (`float64`), or `structs`, or `interfaces`, or any combination of these. (An interface data field is in effect a pointer to an item—of any kind—that satisfies the interface, i.e., that has the methods the interface specifies.)

It seems natural to use the Greek lowercase letter theta (θ) to represent the polar coordinate's angle, and thanks to Go's use of UTF-8 we are free to do so. This is because Go allows us to use any Unicode letters in our identifiers, not just English letters.

Although the two `structs` happen to have the same data field types they are distinct types and no automatic conversion between them is possible. This supports defensive programming; after all, it wouldn't make sense to simply substitute a cartesian's positional coordinates for polar coordinates. In some cases such conversions do make sense, in which case we can easily create a conversion method (i.e., a method of one type that returned a value of another type) that made use of Go's composite literal syntax to create a value of the target type populated by the fields from the source type. (Numeric data type conversions are covered in Chapter 2; string conversions are covered in Chapter 3.)

```
var prompt = "Enter a radius and an angle (in degrees), e.g., 12.5 90, " +
    "or %s to quit."

func init() {
    if runtime.GOOS == "windows" {
        prompt = fmt.Sprintf(prompt, "Ctrl+Z, Enter")
    } else { // Unix-like
        prompt = fmt.Sprintf(prompt, "Ctrl+D")
    }
}
```

If a package has one or more `init()` functions they are automatically executed *before* the `main` package's `main()` function is called. (In fact, `init()` functions must never be called explicitly.) So when our `polar2cartesian` program is invoked this `init()` function is the first function that is called. We use `init()` to set the prompt to account for platform differences in how end of file is signified—for example, on Windows end of file is given by pressing Ctrl+Z then Enter. Go's `runtime` package provides the `GOOS` (Go Operating System) constant which is a string identifying the operating system the program is running on. Typical values are `darwin` (Mac OS X), `freebsd`, `linux`, and `windows`.

Before diving into the `main()` function and the rest of the program we will briefly discuss channels and show some toy examples before seeing them in proper use.

Channels are modeled on Unix pipes and provide two-way (or at our option, one-way) communication of data items. Channels behave like FIFO (first in, first out) queues, hence they preserve the order of the items that are sent into them. Items cannot be dropped from a channel, but we are free to ignore any or all of the items we receive. Let's look at a very simple example. First we will make a channel:

```
messages := make(chan string, 10)
```

Channels are created with the `make()` function (Chapter 7) and are declared using the syntax, `chan` *Type*. Here we have created the `messages` channel to send and receive `strings`. The second argument to `make()` is the buffer size (which defaults to 0); here we have made it big enough to accept ten strings. If a channel's buffer is filled it blocks until at least one item is received from it. This means that any number of items can pass through a channel, providing the items are retrieved to make room for subsequent items. A channel with a buffer size of 0 can only send an item if the other end is waiting for an item. (It is also possible to get the effect of nonblocking channels using Go's `select` statement, as we will see in Chapter 7.)

Now we will send a couple of strings into the channel:

```
messages <- "Leader"
messages <- "Follower"
```

When the <- communication operator is used as a binary operator its left-hand operand must be a channel and its right-hand operand must be a value to send to the channel of the type the channel was declared with. Here, we first send the string `Leader` to the `messages` channel, and then we send the string `Follower`.

```
message1 := <-messages
message2 := <-messages
```

When the <- communication operator is used as a unary operator with just a right-hand operand (which must be a channel), it acts as a receiver, blocking until it has a value to return. Here, we retrieve two messages from the `messages` channel. The `message1` variable is assigned the string `Leader` and the `message2` variable is assigned the string `Follower`; both variables are of type `string`.

Normally channels are created to provide communication between goroutines. Channel sends and receives don't need locks, and the channel blocking behavior can be used to achieve synchronization.

Now that we have seen some channel basics, let's see channels—and goroutines—in practical use.

```
func main() {
    questions := make(chan polar)
    defer close(questions)
    answers := createSolver(questions)
    defer close(answers)
    interact(questions, answers)
}
```

Once any `init()` functions have returned, Go's runtime system then calls the `main` package's `main()` function.

Here, the `main()` function begins by creating a channel (of type `chan polar`) for passing `polar` `structs`, and assigns it to the `questions` variable. Once the channel has been created we use a `defer` statement to call the built-in `close()` function (➤ 187) to ensure that it is closed when it is no longer needed. Next we call the `createSolver()` function, passing it the `questions` channel and receiving from it an `answers` channel (of type `chan cartesian`). We use another `defer` statement to ensure that the `answers` channel is closed when it is finished with. And finally, we call the `interact()` function with the two channels, and in which the user interaction takes place.

```
func createSolver(questions chan polar) chan cartesian {
    answers := make(chan cartesian)
    go func() {
        for {
            polarCoord := <-questions  ❶
            θ := polarCoord.θ * math.Pi / 180.0 // degrees to radians
            x := polarCoord.radius * math.Cos(θ)
            y := polarCoord.radius * math.Sin(θ)
            answers <- cartesian{x, y}  ❷
        }
    }()
    return answers
}
```

The `createSolver()` function begins by creating an `answers` channel to which it will send the answers (i.e., cartesian coordinates) to the questions (i.e., polar coordinates) that it receives from the `questions` channel.

After creating the channel, the function then has a `go` statement. A `go` statement is given a function call (syntactically just like a `defer` statement), which is executed in a separate asynchronous goroutine. This means that the flow of control in the current function (i.e., in the main goroutine) continues immediately from the following statement. In this case the `go` statement is followed by a `return` statement that returns the `answers` channel to the caller. As we noted earlier, it is perfectly safe and good practice in Go to return local variables, since Go handles the chore of memory management for us.

In this case we have (created and) called an anonymous function in the `go` statement. The function has an infinite loop that waits (blocking its own goroutine, but not any other goroutines, and not the function in which the goroutine was started), until it receives a question—in this case a `polar` `struct` on the `questions` channel. When a polar coordinate arrives the anonymous function computes the corresponding cartesian coordinate using some simple math (and using the standard library's `math` package), and then sends the answer as a `cartesian` `struct` (created using Go's composite literal syntax), to the `answers` channel.

In ❶ the `<-` operator is used as a unary operator, retrieving a polar coordinate from the `questions` channel. And in ❷ the `<-` operator is used as a binary operator; its left-hand operand being the `answers` channel to send to, and its right-hand operand being the `cartesian` to send.

Once the call to `createSolver()` returns we have reached the point where we have two communication channels set up and where a separate goroutine is waiting for polar coordinates to be sent on the `questions` channel—and without any other goroutine, including the one executing `main()`, being blocked.

```
const result = "Polar radius=%.02f θ=%.02f° → Cartesian x=%.02f y=%.02f\n"

func interact(questions chan polar, answers chan cartesian) {
    reader := bufio.NewReader(os.Stdin)
    fmt.Println(prompt)
    for {
        fmt.Printf("Radius and angle: ")
        line, err := reader.ReadString('\n')
        if err != nil {
            break
        }
        var radius, θ float64
        if _, err := fmt.Sscanf(line, "%f %f", &radius, &θ); err != nil {
            fmt.Fprintln(os.Stderr, "invalid input")
            continue
        }
        questions <- polar{radius, θ}
        coord := <-answers
        fmt.Printf(result, radius, θ, coord.x, coord.y)
    }
    fmt.Println()
}
```

This function is called with both channels passed as parameters. It begins by creating a buffered reader for `os.Stdin` since we want to interact with the user in the console. It then prints the prompt that tells the user what to enter and how to quit. We could have made the program terminate if the user simply pressed Enter (i.e., didn't type in any numbers), rather than asking them to enter end of file. However, by requiring the use of end of file we have made `polar2cartesian` more flexible, since it is also able to read its input from an arbitrary external file using file redirection (providing only that the file has two whitespace-separated numbers per line).

The function then starts an infinite loop which begins by prompting the user to enter a polar coordinate (a radius and an angle). After asking for the user's input the function waits for the user to type some text and press Enter, or to press Ctrl+D (or Ctrl+Z, Enter on Windows) to signify that they have finished. We don't bother checking the error value; if it isn't `nil` we break out of the loop and return to the caller (`main()`), which in turn will return (and call its deferred statements to close the communication channels).

We create two `float64`s to hold the numbers the user has entered and then use Go's `fmt.Sscanf()` function to parse the line. This function takes a string to parse, a format—in this case two whitespace-separated floating-point numbers—and one or more pointers to variables to populate. (The `&` address of operator is used to get a pointer to a value; see §4.1, ➤ 140.) The function returns the number of

items it successfully parsed and an `error` (or `nil`). In the case of an error, we print an error message to `os.Stderr`—this is to make the error message visible on the console even if the program's `os.Stdout` is redirected to a file. Go's powerful and flexible scan functions are shown in use in Chapter 8 (§8.1.3.2, ➤ 380), and listed in Table 8.2 (➤ 383).

If valid numbers were input and sent to the `questions` channel (in a `polar struct`), we block the main goroutine waiting for a response on the `answers` channel. The additional goroutine created in the `createSolver()` function is itself blocked waiting for a `polar` on the `questions` channel, so when we send the `polar`, the additional goroutine performs the computation, sends the resultant `cartesian` to the `answers` channel, and then waits (blocking only itself) for another question to arrive. Once the `cartesian` answer is received in the `interact()` function on the `answers` channel, `interact()` is no longer blocked. At this point we print the `result` string using the `fmt.Printf()` function, and passing the polar and cartesian values as the arguments that the `result` string's `%` placeholders are expecting. The relationship between the goroutines and the channels is illustrated in Figure 1.1.

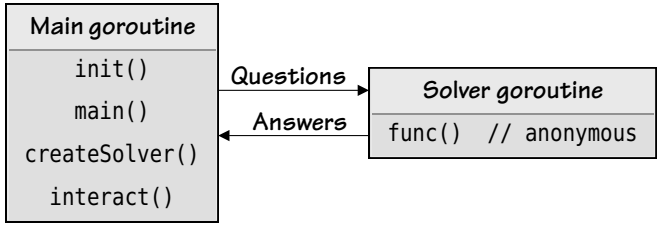

Figure 1.1 *Two communicating goroutines*

The `interact()` function's `for` loop is an infinite loop, so as soon as a result is printed the user is once again asked to enter a radius and angle, with the loop being broken out of only if the `reader` reads end of file—either interactively from the user or because the end of a redirected input file has been reached.

The calculations in `polar2cartesian` are very lightweight, so there was no real need to do them in a separate goroutine. However, a similar program that needed to do multiple independent heavyweight calculations as the result of each input might well benefit from using the approach shown here, for example, with one goroutine per calculation. We will see more realistic use cases for channels and goroutines in Chapter 7.

We have now completed our overview of the Go language as illustrated by the five example programs reviewed in this chapter. Naturally, Go has much more to offer than there has been space to show here, as we will see in the subsequent chapters, each of which focuses on a specific aspect of the language and any relevant packages from the standard library. This chapter concludes with a small exercise, which despite its size, requires some thought and care.

1.8. Exercise

Copy the `bigdigits` directory to, say, `my_bigdigits`, and modify `my_bigdigits/bigdigits.go` to produce a version of the `bigdigits` program (§1.4, 16 ◀) that can optionally output the number with an overbar and underbar of "*"s, and with improved command-line argument handling.

The original program output its usage message if no number was given; change this so that the usage message is also output if the user gives an argument of `-h` or `--help`. For example:

```
$ ./bigdigits --help
usage: bigdigits [-b|--bar] <whole-number>
-b --bar  draw an underbar and an overbar
```

If the `--bar` (or `-b`) option is *not* present the program should have the same behavior as before. Here is an example of the expected output if the option *is* present:

```
$ ./bigdigits --bar 8467243
************************************************
 888      4     666   77777   222      4     333
8   8    44    6          7  2   2    44    3   3
8   8   4 4    6         7       2    4 4        3
 888   4  4    6666     7       2    4  4      33
8   8  444444  6   6   7      2     444444      3
8   8     4    6   6  7      2         4    3   3
 888      4     666   7      22222     4     333
************************************************
```

The solution requires more elaborate command-line processing than the version shown in the text, although the code producing the output only needs a small change to output the overbar before the first row and the underbar after the last row. Overall, the solution needs about 20 extra lines of code—the solution's `main()` function is twice as long as the original (~40 vs. ~20 lines), mostly due to the code needed to handle the command line. A solution is provided in the file `bigdigits_ans/bigdigits.go`.

Hints: The solution also has a subtle difference in the way it builds up each row's line to prevent the bars extending too far. Also, the solution imports the `strings` package and uses the `strings.Repeat(string, int)` function. This function returns a string that contains the `string` it is given as its first argument repeated by the number of times of the `int` given as its second argument. Why not look this function up either locally (see the sidebar "The Go Documentation", 8 ◀), or at `golang.org/pkg/strings`, and start to become familiar with the Go standard library's documentation.

It would be much easier to handle command-line arguments using a package designed for the purpose. Go's standard library includes a rather basic command line parsing package, `flag`, that supports X11-style options (e.g., `-option`). In addition, several option parsers that support GNU-style short and long options (e.g., `-o` and `--option`) are available from `godashboard.appspot.com/project`.

This page intentionally left blank

Index

Symbols & Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Mark Summerfield

Mark is a computer science graduate with many years of experience working in the software industry, primarily as a programmer. He has also spent many years writing and editing technical documentation. Mark owns Qtrac Ltd. (`www.qtrac.eu`), where he works as an independent programmer, author, editor, and trainer, specializing in the C++, Go, and Python languages, and the Qt, PyQt, and PySide libraries.

Other books by Mark Summerfield:

- *Advanced Qt Programming* (2011, ISBN-13: 978-0-321-63590-7)
- *Programming in Python 3* (First Edition, 2009, ISBN-13: 978-0-13-712929-4; Second Edition, 2010, ISBN-13: 978-0-321-68056-3)
- *Rapid GUI Programming with Python and Qt* (2008, ISBN-13: 978-0-13-235418-9)

Other books by Jasmin Blanchette and Mark Summerfield:

- *C++ GUI Programming with Qt 4* (First Edition, 2006, ISBN-13: 978-0-13-187249-3; Second Edition, 2008, ISBN-13: 978-0-13-235416-5)
- *C++ GUI Programming with Qt 3* (2004, ISBN-13:!978-0-13-124072-8)

Production

The text was written using *gvim*. The typesetting—including all the diagrams—was done using the *lout* typesetting language. All of the code snippets were automatically extracted directly from the example programs and from test programs using a custom tool written in Go. The index was compiled by the author. The text and source code was version-controlled using *Mercurial*. The monospaced code font was derived from a condensed version of DejaVu Mono and modified using *FontForge*. The book was previewed using *evince* and *gv,* and converted to PDF by *Ghostscript*. The cover was provided by the publisher. Note that only *printed* editions are definitive: *eBook* versions are not under the author's control and are often retypeset, which can introduce errors.

All the editing and processing was done on Debian and Ubuntu systems. All the example programs have been tested using the official *gc* Go compiler on Linux, Mac OS X, and Windows using Go 1 and should work with all subsequent Go 1.*x* versions.