Mark Summerfield

# Programming in **GO**

#### Creating Applications for the 21st Century

#### **Developer's Library**







in

SHARE WITH OTHERS

# Programming in Go



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 opensource programming languages and databases, Linux programming, Microsoft, and Java, to Web development, social networking platforms, Mac/iPhone programming, and Android programming.

# 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 Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales (800) 382-3419 corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales international@pearsoned.com

Visit us on the Web: informit.com/aw

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

Copyright © 2012-14 Qtrac Ltd.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

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 Second printing, March 2014 This book is dedicated to Jasmin Blanchette and Trenton Schulz This page intentionally left blank

## **Contents at a Glance**

Tables	xv
Introduction	1
Chapter 1. An Overview in Five Examples	7
Chapter 2. Booleans and Numbers	51
Chapter 3. Strings	81
Chapter 4. Collection Types	139
Chapter 5. Procedural Programming	185
Chapter 6. Object-Oriented Programming	253
Chapter 7. Concurrent Programming	315
Chapter 8. File Handling	361
Chapter 9. Packages	407
Appendix A. Epilogue	435
Appendix B. The Dangers of Software Patents	437
Appendix C. Selected Bibliography	441
Index	443

www.qtrac.eu/gobook.html

This page intentionally left blank

## Contents

Tables	xv
Introduction	1
Why Go?	1
The Structure of the Book	4
Acknowledgments	5
Chapter 1. An Overview in Five Examples	7
1.1. Getting Going	7
1.2. Editing, Compiling, and Running	9
1.3. Hello Who?	14
1.4. Big Digits—Two-Dimensional Slices	16
1.5. Stack—Custom Types with Methods	21
1.6. Americanise—Files, Maps, and Closures	29
1.7. Polar to Cartesian—Concurrency	40
1.8. Exercise	48
Chapter 2. Booleans and Numbers	51
2.1. Preliminaries	51
2.1.1. Constants and Variables	53
2.1.1.1. Enumerations	54
2.2. Boolean Values and Expressions	56
2.3. Numeric Types	57
2.3.1. Integer Types	59
2.3.1.1. Big Integers	61
2.3.2. Floating-Point Types	64
2.3.2.1. Complex Types	70
2.4. Example: Statistics	72
2.4.1. Implementing Simple Statistics Functions	73
2.4.2. Implementing a Basic HTTP Server	75
2.5. Exercises	78

Chapter 3. Strings	81
3.1. Literals, Operators, and Escapes	83
3.2. Comparing Strings	86
3.3. Characters and Strings	87
3.4. Indexing and Slicing Strings	90
3.5. String Formatting with the Fmt Package	93
3.5.1. Formatting Booleans	97
3.5.2. Formatting Integers	98
3.5.3. Formatting Characters	99
3.5.4. Formatting Floating-Point Numbers	100
3.5.5. Formatting Strings and Slices	101
3.5.6. Formatting for Debugging	103
3.6. Other String-Related Packages	106
3.6.1. The Strings Package	107
3.6.2. The Strconv Package	113
3.6.3. The Utf8 Package	117
3.6.4. The Unicode Package	118
3.6.5. The Regexp Package	120
3.7. Example: M3u2pls	130
3.8. Exercises	135
Chapter 4. Collection Types	139
4.1. Values, Pointers, and Reference Types	140
4.2. Arrays and Slices	148
4.2.1. Indexing and Slicing Slices	153
4.2.2. Iterating Slices	154
4.2.3. Modifying Slices	156
4.2.4. Sorting and Searching Slices	160
4.3. Maps	164
4.3.1. Creating and Populating Maps	166
4.3.2. Map Lookups	168
4.3.3. Modifying Maps	169
4.3.4. Key-Ordered Map Iteration	170
4.3.5. Map Inversion	170
•	170
4.4. Examples	170
4.4. Examples      4.4.1. Example: Guess Separator	170 171 171
4.4. Examples         4.4.1. Example: Guess Separator         4.4.2. Example: Word Frequencies	170 171 171 174

Chapter 5. Procedural Programming	185
5.1. Statement Basics	186
5.1.1. Type Conversions	190
5.1.2. Type Assertions	191
5.2. Branching	192
5.2.1. If Statements	192
5.2.2. Switch Statements	195
5.2.2.1. Expression Switches	195
5.2.2.2. Type Switches	197
5.3. Looping with For Statements	203
5.4. Communication and Concurrency Statements	205
5.4.1. Select Statements	209
5.5. Defer, Panic, and Recover	212
5.5.1. Panic and Recover	213
5.6. Custom Functions	219
5.6.1. Function Arguments	220
5.6.1.1. Function Calls as Function Arguments	220
5.6.1.2. Variadic Functions	221
5.6.1.3. Functions with Multiple Optional Arguments	222
5.6.2. The init() and main() Functions	224
5.6.3. Closures	225
5.6.4. Recursive Functions	227
5.6.5. Choosing Functions at Runtime	230
5.6.5.1. Branching Using Maps and Function References	230
5.6.5.2. Dynamic Function Creation	231
5.6.6. Generic Functions	232
5.6.7. Higher Order Functions	238
5.6.7.1. Memoizing Pure Functions	241
5.7. Example: Indent Sort	244
5.8. Exercises	250
Chapter 6 Object-Oriented Programming	253
6.1 Kay Concepts	254
6.2 Custom Types	256
6.2.1 Adding Methods	258
6.2.1.1 Overriding Methods	261
6.2.1.2. Method Expressions	263
6.2.2. Validated Types	263
one of the second state of	200

6.3. Interfaces	265
6.3.1. Interface Embedding	270
6.4. Structs	275
6.4.1. Struct Aggregation and Embedding	275
6.4.1.1. Embedding Values	276
6.4.1.2. Embedding Anonymous Values That Have Methods	277
6.4.1.3. Embedding Interfaces	279
6.5. Examples	282
6.5.1. Example: FuzzyBool—A Single-Valued Custom Type	282
6.5.2. Example: Shapes—A Family of Custom Types	289
6.5.2.1. Package-Level Convenience Functions	289
6.5.2.2. A Hierarchy of Embedded Interfaces	294
6.5.2.3. Freely Composable Independent Interfaces	294
6.5.2.4. Concrete Types and Methods	295
6.5.3. Example: Ordered Map—A Generic Collection Type	302
6.6. Exercises	311
Chapter 7. Concurrent Programming	315
7.1. Key Concepts	317
7.2. Examples	322
7.2.1. Example: Filter	322
7.2.2. Example: Concurrent Grep	326
7.2.3. Example: Thread-Safe Map	334
7.2.4. Example: Apache Report	341
7.2.4.1. Synchronizing with a Shared Thread-Safe Map	341
7.2.4.2. Synchronizing with a Mutex-Protected Map	345
7.2.4.3. Synchronizing by Merging Local Maps via Channels	347
7.2.5. Example: Find Duplicates	349
7.3. Exercises	357
Chapter 8. File Handling	361
8.1. Custom Data Files	362
8.1.1. Handling JSON Files	365
8.1.1.1. Writing JSON Files	366
8.1.1.2. Reading JSON Files	368
8.1.2. Handling XML Files	371
8.1.2.1. Writing XML Files	371
8.1.2.2. Reading XML Files	375
8.1.3. Handling Plain Text Files	377

8.1.3.1. Writing Plain Text Files	378
8.1.3.2. Reading Plain Text Files	380
8.1.4. Handling Go Binary Files	385
8.1.4.1. Writing Go Binary Files	385
8.1.4.2. Reading Go Binary Files	386
8.1.5. Handling Custom Binary Files	387
8.1.5.1. Writing Custom Binary Files	388
8.1.5.2. Reading Custom Binary Files	392
8.2. Archive Files	397
8.2.1. Creating Zip Archives	397
8.2.2. Creating Optionally Compressed Tarballs	399
8.2.3. Unpacking Zip Archives	401
8.2.4. Unpacking Optionally Compressed Tarballs	403
8.3. Exercises	405
Chapton 9 Backagos	407
0.1 Custom Pockages	407
9.1. Custom rackages	400
9.1.1. Distform Specific Code	400
9.1.1.2 Documenting Packages	410
9.1.1.2. Documenting rackages	41A
9.1.9 Importing Packages	414
9.9 Third-Party Packages	410
9.3 A Brief Survey of Go's Commands	418
9.4 A Brief Survey of the Go Standard Library	419
9.4.1 Archive and Compression Packages	419
9.4.2 Bytes and String-Related Packages	419
943 Collection Packages	421
9.4.4 File Operating System and Related Packages	423
9 4 4 1 File Format-Related Packages	424
9.4.5. Graphics-Related Packages	425
9.4.6. Mathematics Packages	425
9.4.7. Miscellaneous Packages	425
9.4.8. Networking Packages	427
9.4.9. The Reflect Package	427
9.5. Exercises	431

Appendix A. Epilogue	435
Appendix B. The Dangers of Software Patents	437
Appendix C. Selected Bibliography	441
Index	443

## **Tables**

2.1.	Go's Keywords	52
2.2.	Go's Predefined Identifiers	52
2.3.	Boolean and Comparison Operators	57
2.4.	Arithmetic Operators Applicable to All Built-In Numbers	59
2.5.	Go's Integer Types and Ranges	60
2.6.	Arithmetic Operators Applicable Only to Built-In Integer Types	60
2.7.	Go's Floating-Point Types	64
2.8.	The Math Package's Constants and Functions #1	65
2.9.	The Math Package's Constants and Functions #2	66
2.10.	The Math Package's Constants and Functions #3	67
2.11.	The Complex Math Package's Functions	71
3.1.	Go's String and Character Escapes	84
3.2.	String Operations	85
3.3.	The Fmt Package's Print Functions	94
3.4.	The Fmt Package's Verbs	95
3.5.	The Fmt Package's Verb Modifiers	96
3.6.	The Strings Package's Functions #1	108
3.7.	The Strings Package's Functions #2	109
3.8.	The Strconv Package's Functions #1	114
3.9.	The Strconv Package's Functions #2	115
3.10.	The Utf8 Package's Functions	118
3.11.	The Unicode Package's Functions	119
3.12.	The Regexp Package's Functions	121
3.13.	The Regexp Package's Escape Sequences	121
3.14.	The Regexp Package's Character Classes	122
3.15.	The Regexp Package's Zero-Width Assertions	122
3.16.	The Regexp Package's Quantifiers	123
3.17.	The Regexp Package's Flags and Groups	123
3.18.	The *regexp.Regexp Type's Methods #1	124
3.19.	The *regexp.Regexp Type's Methods #2	125
4.1.	Slice Operations	151

4.2.	The Sort Package's Functions	161
4.3.	Map Operations	165
5.1.	Built-In Functions	187
8.1.	Format Speed and Size Comparisons	363
8.2.	The Fmt Package's Scan Functions	383

# 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 builtin 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.gtrac.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 thirdparty 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 *#defines* 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

<sup>\*</sup> 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.

# An Overview in Five Examples

§1.1. Getting Going ➤ 7
§1.2. Editing, Compiling, and Running ➤ 9
§1.3. Hello Who? ➤ 14
§1.4. Big Digits—Two-Dimensional Slices ➤ 16
§1.5. Stack—Custom Types with Methods ➤ 21
§1.6. Americanise—Files, Maps, and Closures ➤ 29
§1.7. Polar to Cartesian—Concurrency ➤ 40

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.xseries. (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

<sup>\*</sup> 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/kisom/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 %HOME-PATH%\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 GOR00T=\$HOME/opt/go
export PATH=\$PATH:\$GOR00T/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 GOR00T 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.

<sup>\*</sup> 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) colonseparated 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

<sup>\*</sup> 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 ( ①
    "fmt"
    "os"
    "strings"
)
func main() {
    who := "World!" ②
    if len(os.Args) > 1 { /* os.Args[0] is "hello" or "hello.exe" */ ③
    who = strings.Join(os.Args[1:], " ") ④
    }
    fmt.Println("Hello", who) ⑤
}
```

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

<sup>\*</sup> We use some simple syntax highlighting and sometimes highlight lines or annotate them with numbers  $(\mathbf{0}, \mathbf{Q}, ...)$ , 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  $(14 \prec, \mathbf{0})$  imports three packages from the standard library. The fmt package provides functions for formatting text and for reading formatted text (§3.5,  $\succ$  93), the os package provides platform-independent operating-system variables and functions, and the strings package provides functions for manipulating strings (§3.6.1,  $\succ$  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  $\triangleleft$ , @; 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,  $\geq$  29), and further on (§5.2.1,  $\geq$  192).

The os.Args variable is a *slice* of strings (14  $\triangleleft$ , S). Arrays, slices, and other collection data types are covered in Chapter 4 (§4.2,  $\succ$  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  $(14 \blacktriangleleft, \bigcirc)$ . 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  $(14 \prec, \bigcirc)$ , 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:

\$	./bi	gdig	gits	290	17549	3					
2	222	99	999	00	90	1	77777	55555	4	9999	333
2	2	9	9	0	0	11	7	5	44	99	33
	2	9	9	0	0	1	7	5	4 4	99	3
	2	99	999	0	0	1	7	555	4 4	9999	33
2	2		9	0	0	1	7	5	444444	9	3
2			9	0	0	1	7	5 5	4	9	3 3
22	2222		9	00	90	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",
    "0 0",
    "0 0",
    "0 0",
    "0 00 "},
    {" 1 ", "11 ", " 1 ", " 1 ", " 1 ", " 11", "111"},
    {" 222 ", "2 2", " 2 ", " 2 ", "2 ", "2 ", "22222"},
    // ... 3 to 8 ...
    {" 9999", "9 9", "9 9", " 9999", " 9", " 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.

```
line += bigDigits[digit][row] + " " 
} 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  $\triangleleft$ ,  $\bullet$ ) 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,  $\succ$  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  $\theta$  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  $(18 \blacktriangleleft, 2 \text{ and } 18 \blacktriangleleft, 3)$  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++ {
        ...</pre>
```
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 strings 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  $\triangleleft$ ,  $\bigcirc$ ). 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 ( $18 \lt , \textcircled{o}$ ) we can add the appropriate string to the line. (In the if statement the constants 0 and 9 are considered to be bytes 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  $(19 \lt, \bigcirc)$  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

<sup>\*</sup> 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  $\lt$ , ①) 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 bool, int, and string, or by aggregations of types using structs.<sup>\*</sup> 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.)

<sup>\*</sup> 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,  $\geq$  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
```

<sup>\*</sup> 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 GOR00T. 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

<sup>\*</sup> 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 ( $\geq$  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.<sup>\*</sup> 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;  $25 \blacktriangleleft$ ). 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-

<sup>\*</sup> 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 ( $\S6.3$ , > 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 Goguarantees 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.

<sup>\*</sup> 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] ①
    *stack = theStack[:len(theStack)-1] ②
    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 (**0**); 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 (O). 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,  $\succ 153$ .)

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 ints or strings), 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.

<sup>\*</sup> 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() 
   if err != nil {
        fmt.Println(err) ②
        os.Exit(1)
   }
   inFile, outFile := os.Stdin, os.Stdout 3
   if inFilename != "" {
        if inFile, err = os.Open(inFilename); err != nil {
            log.Fatal(err)
        }
       defer inFile.Close() 4
    }
   if outFilename != "" {
        if outFile, err = os.Create(outFilename); err != nil {
            log.Fatal(err)
        }
       defer outFile.Close() 
   }
   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 ( $30 < , 0, \odot$ ).

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 <,  $\bigcirc$ ,  $\bigcirc$ )—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 < , @), 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

<sup>\*</sup> 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/lce5cd050bb973e4?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

<sup>\*</sup> 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.

<sup>\*</sup> In fact, the user could still overwrite the input file by using redirection—for example,

<sup>\$ ./</sup>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 1
    if replacer, err = makeReplacerFunction(britishAmerican); err != nil {
        return err
    }
    wordRx := regexp.MustCompile("[A-Za-z]+")
    eof := false
    for !eof {
        var line string 2
        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 { 3
            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 filenamesFromCommand-Line() 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 <, **0**) and the line we read in (35 <, **0**). 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*,  $(35 \blacktriangleleft, \bigcirc)$ . 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,  $\succ$  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 reg-exp.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.Read-String() 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(vari-able)*. 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,  $\succ$  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 support 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 \theta=30.50° \rightarrow Cartesian x=4.31 y=2.54
Radius and angle: 5 -30.25
Polar radius=5.00 \theta=-30.25° \rightarrow Cartesian x=4.32 y=-2.52
Radius and angle: 1.0 90
Polar radius=1.00 \theta=90.00° \rightarrow 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 ( $\theta$ ) 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"</pre>
```

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</pre>
```

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 ●
            e = polarCoord.0 * math.Pi / 180.0 // degrees to radians
            x := polarCoord.radius * math.Cos(0)
            y := polarCoord.radius * math.Sin(0)
            answers <- cartesian{x, y} ●
        }
    }()
    return answers
}</pre>
```

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 **0** the <- operator is used as a unary operator, retrieving a polar coordinate from the questions channel. And in **2** 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 0=%.02f° → Cartesian x=%.02f y=%.02f\n"
func interact(guestions 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, \theta}
        coord := <-answers</pre>
        fmt.Printf(result, radius, \theta, 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 float64s 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  $\blacktriangleleft$ ) 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:

\$.	/bi	gdigits	ba	ar	8467243				
***************************************									
88	88	4 666		66	77777	222	4	3	33
8	8	44	6		7	22	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	3	33
******************									

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",  $8 \lt$ ), 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

- ! logical NOT operator, 57
- != inequality operator, 56–57, 68–69, 70, 164
- " " double quotes, 83
- #! shebang scripts, 10

\$ replacements in regular expressions, 120, 126, 129

% modulus operator and formatting placeholder, 47, 60, 69; *see also* format specifier

%= augmented modulus operator, 60

- && logical AND operator, 56, 57
- &= augmented bitwise AND operator, 60
- &^ bitwise clear operator, 60
- &^= augmented bitwise clear operator, 60
- \* multiplication, dereference, pointer declaration operator and formatting placeholder, 26, 59, 69, 96, 100, 142, 143, 144, 178, 247, 248, 249, 259, 284, 305, 370, 382, 394
- \*= augmented multiplication operator, 59, 147
- + addition, concatenation, and unary plus operator, 20, 59, 84, 85, 226
- ++ increment operator, 20, 59, 186, 188
- += augmented addition and string append operator, 20, 59, 84, 85, 88, 140; *see also* append()

- subtraction and unary minus operator, 59
- -- decrement operator, 20, 59, 186, 188
- -= augmented subtraction operator, 59
- . selector operator, 148, 275
- ... ellipsis operator, 149, 156, 158, 160, 176, 219, 221, 222, 233, 242, 268, 287, 378
- / division operator, 59
- /\* \*/ multiline comments, 14, 51
- // single-line comments, 14, 51
- /= augmented division operator, 59
- := short variable declaration operator, 15, 18, 36, 53, 188, 189, 198, 203
- ; semicolon, 15, 186
- < less than comparison operator, 56–57
- <- send/receive communication operator, 44, 45, 207, 210, 318–357
- << bitwise shift left operator, 55, 60
- <<= augmented bitwise shift left operator, 60
- <= less than or equal comparison operator, 56–57
- = assignment operator, 16, 18, 36, 188, 212
- == equality operator, 56–57, 68–69, 70, 164
- > greater than comparison operator, 56–57
- >= greater than or equal comparison operator, 56–57
- >> bitwise right shift operator, 60
- >>= augmented bitwise right shift operator, 60
- [] index and slice operator, 16, 28, 85, 91, 203, 242, 339, 355, 357, 393

 $\land$  (backslash), 84 \a (alert or bell), 84 b (backspace), 84 f (form feed). 84 \n (newline), 51, 84 r (carriage return), 84 \t (tab), 84 \Uhhhhhhhh (rune literal), 84 \uhhhh (rune literal), 84 v (vertical tab), 84 xhh (rune literal), 84 ^ bitwise XOR and complement operator, 60 ^= augmented bitwise XOR operator, 60 blank identifier, 36, 52–53, 154, 170, 188, 291, 358, 417; see also identifiers `` backticks, 75, 78, 96 { } braces, 15, 186 | bitwise OR operator, 55, 60 = augmented bitwise OR operator, 60 || logical OR operator, 56, 57, 178 5q, 6q, 8q (tool), 9 51, 61, 81 (tool), 9 7-bit ASCII encoding, 82

## A

Abs() cmplx package, 71 math package, 65, 68 abstract vs. concrete types, 22 abstraction and abstract types; *see* interfaces access operator; *see* [] index operator access, serialized, 318–319, 335, 341 accessing maps, 39, 168–169, 231 accuracy, floating-point, 64 Acos() cmplx package, 71 math package, 65 Acosh() cmplx package, 71 math package, 65 Add() Int type, 63 WaitGroup type, 350, 351, 352, 354 address; see pointers After() (time package), 332, 333, 426 aggregation, 254–256, 275–282; see also embedding alert or bell (\a), 84 aliasing, package names, 409, 418 americanise (example), 29-40 and, logical; see && operator anonymous fields, struct keyword anonymous functions, 36, 37, 110, 112, 206, 208, 212, 216, 218, 225, 226, 239, 240, 243, 290 anonymous struct, 275 apachereport (example), 341-349 **API** (Application Programming Interface); see interfaces App Engine, Google, 2, 435 append() (built-in), 25, 27, 55, 77, 129, 132, 150, 151, 156-157, 158, 159, 160, 170, 176, 178, 179, 187, 232,240, 247, 249, 272, 355, 374, 382, 392, 410; see also += AppendBool() (strconv package), 114 AppendFloat() (strconv package), 114 AppendInt() (strconv package), 114 AppendQuote() (strconv package), 114 AppendQuoteRune() (strconv package), 114 AppendQuoteRuneToASCII() (strconv package), 114 AppendUInt() (strconv package), 114 archive files; see .tar files and .zip files archive (package) tar (package); see top-level entry zip (package); see top-level entry archive file list (exercise), 250

Args slice (os package), 14, 15, 17, 19, 131-132,232 arguments, command line, 16, 17, 19, 232; see also flag package and commandLineFiles() arguments, function and method; see parameters arrays, 140, 148–150; see also slices iterating, 203 multidimensional, 148 mutability, 149 ASCII encoding, 82 Asin() cmplx package, 71 math package, 65 Asinh() cmplx package, 71 math package, 65 assertions, type; see type assertions assertions, zero-width in regular expressions, 122 assignment operators; see = and := operators assignments, multiple, 31, 188 associative array; see map type asynchronous, channel, 207; see also channels Atan() cmplx package, 71 math package, 65 Atan2() (math package), 65 Atanh() cmplx package, 71 math package, 65 Atoi() (strconv package), 116, 134, 390 atomic updates, 338 audio format, Vorbis, 130

### В

backslash (\), 84 backreferences, in regular expressions, 126

backspace (\b), 84 backticks (``), 75, 96 backtracking, in regular expressions, 120balanced binary tree; see omap example bare returns, 34, 189, 219 Base() (filepath package), 19. 131-132, 194, 327 benchmarking, 415–416 big (package; math package) Int (type); see top-level entry NewInt().63 ProbablyPrime(), 425Rat (type); see top-level entry big-O notation, 89 bigdigits (example), 16–21, 48 bigdigits (exercise), 48 BigEndian (variable; binary package), 389 binary files, 387-397; see also .gob files binary number, formatting as, 98 binary (package; encoding package), 388, 391 BigEndian (variable), 389 LittleEndian (variable), 388, 389 Read(), 391, 393, 395 Write(), 388 binary tree; see omap example binary vs. linear search, 162–163 bisectLeft() (example), 314 blank identifier (), 36, 52–53, 154, 170, 188, 291, 358, 417; see also identifiers blocking, channel, 207-208, 209; see also channels bool (type; built-in), 53, 56–57, 195, 204,318 formatting, 97 Bool() (Value type), 428Boolean expressions, 193, 204 braces ({ }), 15, 186

- branching, 192-202
- break (statement), 24, 177, 186, 204, 205, 331
- Buffer (type; bytes package), 111, 200, 201, 243 ReadRune(), 113
  - String(), 88, 200, 243
  - WriteRune(),111
  - WriteString(), 88, 111, 200, 243
- buffers; *see* bufio package and File type
- bufio (package), 30, 34, 38
   NewReader(), 35, 176, 333, 380
   NewWriter(), 35, 378
   Reader (type); see top-level entry
  - Writer (type); *see* top-level entry
- building Go programs, 11–13
- built-in functions
  - $\begin{array}{r} {\rm append\,(),\,25,\,27,\,55,\,77,\,129,\,132,}\\ {\rm 150,\,151,\,156{-}157,\,158,\,159,}\\ {\rm 160,\,170,\,176,\,178,\,179,\,187,}\\ {\rm 232,\,240,\,247,\,249,\,272,\,355,}\\ {\rm 374,\,382,\,392,\,410;\,see\,also}\\ {\rm +=} \end{array}$
  - cap(), 24, 149, 151, 152, 157, 187, 324
  - $\begin{array}{c} \texttt{close(), 44, 187, 211, 320, 321,} \\ 324, 325, 329, 330, 340, 343 \end{array}$
  - complex(), 58, 187; see also cmplx
     package
  - copy(), 157-158, 159, 187, 268
  - delete(), 165, 169, 187, 339
  - imag(), 70, 101, 187
  - len(), 15, 20, 24, 27, 69, 85, 90, 148, 149, 151, 152, 157, 159, 165, 169, 187, 340
  - make(), 26, 38, 39, 43, 44, 127, 129, 150, 151, 152, 157, 159, 165, 172, 176, 178, 179, 187, 207, 208, 209, 211, 240, 242, 246, 247, 323, 324, 328, 337, 339, 341, 346, 348, 355, 374, 392, 393, 395, 410
  - new(), 145, 146, 152, 187, 346

- panic(), 32, 69, 70, 113, 187, 191, 196, 213–218, 219, 220, 243 real(), 70, 101, 187 recover(), 32, 187, 213–218
- see also functions and special functions
- built-in types; see bool, byte, error, int, map, rune, string, uint; see also standard library's types
- byte ordering, 83, 389
- byte (type; built-in), 20, 59, 60, 82, 104, 132, 190
  - conversion from string, 89–90, 164, 373, 391
- conversion of []byte to string, 38, 85, 164, 334, 395
- formatting, 102
- *see also* rune and string types
- bytes (package), 419 Buffer (type); *see* top-level entry TrimRight(), 333, 334

### С

C code, external, 9 Call() (Value type), 429, 430 Caller() (runtime package), 291 calling functions, 220–221; see also functions CanBackguote() (strconv package), 114CanSet() (Value type), 428 cap() (built-in), 24, 149, 151, 152, 157, 187, 324 carriage return (\r), 84 case (keyword); see select and switch statements Cbrt() (math package), 65 Ceil() (math package), 65 cgo (tool), 9 cgrep (example), 326-334 chan (keyword), 43, 44, 208, 209, 210, 318–357; see also channels

channels, 41, 43-44, 206-212, 318 - 357infinite, 208 iterating, 203 character; see rune type character classes in regular expressions, 122 character encoding, fixed vs. variable width, 83 character literal, 20; see also rune type checked type assertion; see type assertions class, 254; see also type keyword close() (built-in), 44, 187, 211, 320, 321, 324, 325, 329, 330, 340, 343 Close() (File type), 31, 176, 213, 293, 333, 343, 353, 398, 400 closures, 40, 163, 225–227, 239, 240, 243, 244, 352 cmplx (package; math package), 70 Abs(), 71 Acos(), 71 Acosh(), 71Asin(),71 Asinh(), 71Atan(), 71 Atanh(), 71Conj(),71 Cos(), 71 Cosh(),71 Cot(),71 Exp(), 71 Inf(),71 IsInf(), 71IsNaN(), 71Log(), 71 Log10(),71 NaN(), 71Phase(), 71Polar(), 71 Pow(), 71 Rect(),71

Sin(),71 Sinh(), 71 Sqrt(),71 Tan(), 71 Tanh(), 71 code point, Unicode; *see* rune type collection packages, 421–423 collections, slices, map type, and omap example color (package) RGBA (type); see top-level entry command-line arguments, 16, 17, 19, 232; see also flag package commandLineFiles() (example), 176, 410 commas() (example), 357 comments, Go, 14, 51 CommonPathPrefix() (exercise), 250 CommonPrefix() (exercise), 250 **Communicating Sequential Process**es (CSP), 3, 315 communication, between goroutines; see channels, goroutines, and the chan and go keywords comparisons, 56-57, 70, 84, 86-87; *see also* <, <=, ==, !=, >=, and > operators compilation speed, 2 Compile() (regexp package), 37, 121, 214, 327CompilePOSIX() (regexp package), 121 complex() (built-in), 58, 70, 187; see also cmplx package Complex() (Value type), 428complex64 (type; built-in), 64, 70 complex128 (type; built-in), 64, 70, 101, 187 comparisons, 70 conversion to complex64, 70formatting, 101 literals, 53, 70 see also cmplx package and imag() and real()
composing functionality, 35 composite literals, 18, 45, 150, 152, 153, 166, 167compositing, image, 290 composition; see aggregation and embedding compress (package) gzip (package), see top-level entry concatenation, fast string, 88 concatenation, string; see + and += operators concrete vs. abstract types, 22 concurrency; see channels, goroutines, and the chan and go keywords conditional branching, 192-202 Conj() (cmplx package), 9 console; see Stderr, Stdin, and Stdout streams const (keyword), 45, 53, 54, 58, 133, 336, 364, 379, 390 constant expressions, 58 constants, numeric; see under literals and specific types construction functions, 27, 263, 264, 306 constructors; see construction functions container (package), 421–423 heap (package), 421-422 list (package), 422–423 ring (package), 423 containers; see slices, map type, and omap example Contains() (strings package), 108 contents of; see pointers and \* dereference operator continue (statement), 132, 133, 186, 204, 205, 324conversions, 42, 57, 61, 162, 190–191, 288[]byte to string, 38, 85, 164, 334,395

[]rune to string, 91, 272 complex128 to complex64, 70downsizing, 58, 61 float64 to int, 69 int to float64, 61, 73 int to int64.63 rune to string, 87-88, 246 string to []byte, 85, 164, 373, 391 string to []rune, 85 string to float64,77 see also the strconv package copy() (built-in), 157–158, 159, 187, 268Copy() (io package), 353, 354, 399, 401, 402, 405 copy on write, of strings, 140 Copysign() (math package), 65 coroutines; see channels, goroutines, and the chan and go keywords Cos() cmplx package, 71 math package, 45, 65 Cosh() cmplx package, 71 math package, 65 Cot() (cmplx package), 9 Count() (strings package), 108, 172, 173Cox, Russ, 2, 120, 436 Create() (File type), 31, 293, 397, 400CreateHeader() (Writer type), 398, 399 cross-platform code, 410–411 crypto (package), 425 rand (package), see top-level entrv sha1 (package), see top-level entry .csv files, 424 csv (package; encoding package), 424 currying; see closures

custom packages, 24–29, 408–417; see also packages custom types, 55, 103, 255–282 named vs. unnamed, 22 see also type

### D

Dashboard, Go, 2 data structures, slices, map type, and struct database (package) sql (package), 423 deadlock, 317-318, 340 debugging, 55, 103-106 decimal number, formatting as, 99; see also int type declarations, order of, 21 Decode() gob package Decoder type, 387 ison package Decoder type, 369, 370 xml package Decoder type, 375, 377 DecodeConfig() (image package), 358 DecodeLastRune() (utf8 package), 118 DecodeLastRuneInString() (utf8 package), 91, 118, 229, 230 Decoder (type) qob package Decode(), 387json package), 369 Decode(), 369, 370 xml package Decode(), 375, 377 DecodeRune() (utf8 package), 118 DecodeRuneInString() (utf8 package), 91, 92, 93, 118, 203, 229, 230 DeepEqual() (reflect package), 57, 236, 427 default (keyword); see select and switch statements

defer (statement), 31-32, 35, 36, 44, 97, 176, 211–213, 216, 218, 333, 343, 353, 378, 398, 400 defining methods, 25 definitions, order of, 21 delegation; see embedding delete() (built-in), 165, 169, 187, 339 dereferencing pointers; see pointers dictionary; see map type Dim() (math package), 65 Div() (Int type), 63 division by zero, 68 documentation, Go, 8 documentation, package, 411–413 domain name resolution; see net package Done() (WaitGroup type), 350, 352, 354double quotes (" "), 83 doubly linked list; see list package downloading Go, 9 downsizing; see conversions Draw() (draw package), 290 draw (package; image package) Draw(), 290 Image (interface), 290, 293, 319 duck typing, 21, 32, 254–255, 268 Duration (type; time package), 332, 333

#### Ε

E (constant; math package), 65, 104, 105 Elem() (Value type), 429 else (keyword); see if statement embedding, 254–256, 261, 270–274, 275–282, 294, 300; see also aggregation empty interface; see interface{} Encode() gob package Encoder type, 385, 386 json package Encoder type, 367, 370 xml package Encoder type, 373, 375 Encoder (type) gob package Encode(), 385, 386 json package, 367 Encode(), 367, 370 xml package, 373 Encode(), 373, 375 EncodeRune() (utf8 package), 118 encoding ASCII (7-bit, US-), 82 of characters, fixed vs. variable width, 83 UTF-8; see Unicode encoding (package) binary (package), see top-level entry csv (package); see top-level entry qob (package); see top-level entry json (package), see top-level entry xml (package); see top-level entry end of file; see EOF endianness, 83, 389 endsoftpatents.org (web site), 439 entry point, 14, 224-225 enumerations, 54-56; see also const and iota environment variables GOPATH, 8, 13, 23, 408, 409, 410, 411, 417, 418 GOROOT, 8, 10, 11, 23, 408, 410, 418, 424**PATH**, 10 see also variables EOF (io package), 34, 35, 37, 113, 177, 268, 333, 343, 381, 404

EqualFloat() (example), 68–69 EqualFloatPrec() (example), 69 EqualFold() (strings package), 108, 163 equality comparisons (==, !=), 56-57,68-69, 70, 164 Erf() (math package), 65 Erfc() (math package), 65 error handling, 24, 32, 145, 213 Error() (method), 31 error (type; built-in), 24, 27, 34, 58, 93, 134, 145, 213, 214, 216, 284, 285Errorf() (fmt package), 33, 58, 94, 97, 216, 285, 293, 365, 382, 384 errors (package), 24 New(), 27, 33, 194, 384 escapes, 84, 102, 375, 377; see also regexp package's escapes EscapeString() (html package), 78 examples, 10 americanise, 29-40 apachereport, 341–349 bigdigits, 16-21, 48 bisectLeft(), 314 cgrep, 326-334 commandLineFiles(), 176, 410 commas(), 357 EqualFloat(), 68-69EqualFloatPrec(), 69 filter, 322-326 Filter(), 240-241 findduplicates, 349-357 FuzzyBool, 282-288 guess separator, 171-174 hello, 14-16 Humanize(), 100 indent sort, 244-249 InsertStringSlice(), 158 InsertStringSliceCopy(), 157 - 158invoicedata, 362-397 logPanics(), 218

m3u2pls, 130-135Memoize(), 242-244 omap, 302-310, 409, 412-413, 414 - 416pack, 397-405 Pad(), 99 palindrome, 232 pi\_by\_digits, 62-64 polar2cartesian, 40-47 RemoveStringSlice(), 160 RemoveStringSliceCopy(), 159 - 160RomanForDecimal(), 243–244 safemap, 334-340shaper, 289 - 301SimplifyWhitespace(), 111 SliceIndex(), 238-239stacker, 21-29, 408-409 statistics, 72–78 statistics nonstop, 216–218 unpack, 397-405 wordfrequencies, 174-180 exceptions; see panic() and recover() exec (package; os package), 426 exercises archive file list, 250 bigdigits, 48 CommonPathPrefix(), 250CommonPrefix(), 250Flatten(), 181 font, 311 imagetag, 358 .ini file to map, 181 invoicedata, 406 IsPalindrome(), 250 linkcheck, 432linkutil, 431-432 Make2D(), 181 map to .ini file, 182 oslice, 313-314 playlists, 135-136 quadratic, 79 safeslice, 357-358

shaper, 311-313 sizeimages, 359 soundex, 136-137 statistics, 79 UniqueInts(), 180 unpack, 405 utf16-to-utf8, 405 Exit() (os package), 19, 20, 32, 131 - 132, 327exiting; see termination and Exit() Exp() cmplx package, 71 Int type, 63 math package, 65 Exp2() (math package), 65 Expand() (Regexp type), 124ExpandString() (Regexp type), 124 Expm1() (math package), 65 exponential notation, 64, 101 exported identifiers, 52, 202, 264 expression switches, 195–197 Ext() (filepath package), 293, 324, 325extension, file, 194 external C code, 9

#### F

file globbing, 176, 410-411 file suffix, 194 File (type; os package), 31, 32, 176 Close(), 31, 176, 213, 293, 333, 343, 353, 398, 400 Create(), 31, 293, 397, 400 Open(), 31, 176, 212, 333, 342, 353, 398,400OpenFile(), 31, 397 ReadAt(), 397Readdir(), 361Readdirnames(), 361 Seek(), 397 Stat(), 397, 398, 399, 400 WriteAt(), 397 file types .csv, 424 .go, 84, 408, 410 .gob, 385-387 .ini, 131, 181-182 .jpg and .jpeg, 293 .m3u, 130-135 .pls, 130-135 .png, 293 .tar, 399-401, 403-405 .txt, 377–384 .zip, 397-399, 401-403 FileInfo (interface; os package), 351, 361, 397, 399 Mode(), 351, 401 ModTime(), 401Size(), 351, 353, 401 FileInfoHeader() (zip package), 398, 399 filepath (package; path package), 17, 19,424Base(), 19, 131–132, 194, 327 Ext(), 293, 324, 325 FromSlash(), 135 Glob(), 176, 410 Separator (constant), 134, 135 ToSlash(), 399Walk(), 349, 352

filter (example), 322–326 Filter() (example), 240–241 Find() (Regexp type), 124 FindAll() (Regexp type), 124 FindAllIndex() (Regexp type), 124FindAllString() (Regexp type), 124, 127FindAllStringIndex() (Regexp type), 124FindAllStringSubmatch() (Regexp type), 124, 127 FindAllStringSubmatchIndex() (Regexp type), 124, 128 FindAllSubmatch() (Regexp type), 124FindAllSubmatchIndex() (Regexp type), 124findduplicates (example), 349–357 FindIndex() (Regexp type), 124 FindReaderIndex() (Regexp type), 124FindReaderSubmatchIndex() (Regexp type), 124 FindString() (Regexp type), 124 FindStringIndex() (Regexp type), 124 FindStringSubmatch() (Regexp type), 124, 127, 343, 344, 348 FindStringSubmatchIndex() (Regexp type), 124 FindSubmatch() (Regexp type), 125FindSubmatchIndex() (Regexp type), 125fixed vs. variable-width character encoding, 83 flag (package), 426 flags, regular expression, 123 Flatten() (exercise), 181 Float() (Value type), 428 float32 (type; built-in), 61, 64, 70, 283, 285, 427Float32bits() (math package), 65 Float32frombits() (math package), 65 float64 (type; built-in), 61, 62, 64-70, 73, 100, 187, 221, 304, 318 accuracy, 64

comparisons, 57, 68-69 conversion from int, 61, 69, 73 conversion from string, see Parse-Float() formatting, 100–101 literals, 53, 58 see also math package Float64bits() (math package), 65 Float64frombits() (math package), 65 Float64s() (sort package), 73, 161 Float64sAreSorted() (sort package), 161 Floor() (math package), 66 Flush() (Writer), 35, 36, 378 fmt (package), 55, 93-106, 192 Errorf(), 33, 58, 94, 97, 216, 285, 293, 365, 382, 384format specifier, 96, 97; see also % symbol Fprint(), 76, 94, 96 Fprintf(), 76, 94, 97, 200, 201, 378Fprintln(), 45, 94, 96 Fscan(), 383 Fscanf(), 383, 384 Fscanln(), 383Print(), 94, 96 Printf(), 19, 47, 94–106, 113, 178, 192Println(), 19, 24, 45, 53, 72, 94 - 106Scan(), 383 Scanf(), 383 Scanln(), 383 Sprint(), 94, 99, 178, 242, 357 Sprintf(), 43, 55, 69, 78, 85, 94, 97, 100, 101, 242, 286, 355 Sprintln(),94 Sscan(), 383 Sscanf(), 45, 46, 382, 383 Sscanln(), 383Stringer (interface), 265, 266-267,286 font (exercise), 311

for loop, 19, 23, 24, 38, 39, 74, 89, 110, 132, 147, 154, 155, 168, 170,172, 177, 186, 200, 203-205, 320,321, 324, 325, 330, 331, 339, 343, 355, 390 form feed (f), 84 Form (field; Request type), 76 format specifier, fmt package, 96, 97; see also % symbol Format() (Time type), 368, 379, 390 FormatBool() (strconv package), 114, 116 FormatFloat() (strconv package), 114 FormatInt() (strconv package), 114, 117formatting bools, 97 complex numbers, 101 floating-point numbers, 100–101 for debugging, 103–106 integers, 98–99 maps, 106 pointers, 96, 104 runes, 99 slices, 101–103 strings, 101-103 FormatUInt() (strconv package), 114 Fprint() (fmt package), 76, 94, 96 Fprintf() (fmt package), 76, 94, 97, 200, 201, 378Fprintln() (fmt package), 45, 94, 96Frexp() (math package), 66 FromSlash() (filepath package), 135 Fscan() (fmt package), 383 Fscanf() (fmt package), 383, 384 Fscanln() (fmt package), 383 FullRune() (utf8 package), 118 FullRuneInString() (utf8 package), 118

func (keyword), 14, 15, 25, 35, 45, 55, 208, 216, 218, 219, 223, 226, 232,238, 240, 241, 243, 246, 291, 303,305, 323, 324, 343, 378, 379, 388, 389,413FuncForPC() (runtime package), 291, 292functionality, composing, 35 functions, 219-244 anonymous, 36, 37, 110, 112, 206, 208, 212, 216, 218, 225, 226, 239, 240, 243, 290 calling, 220–221 closures, 40, 163, 225–227, 239, 240, 243, 244 construction, 27, 306 factory, 226, 291, 298 generic, 232-238; see also higher order functions higher order, 37, 38, 238–244, 257literal: see closures optional parameters, 222–223 parameters, 220-223, 254-255 pure, 241; see also memoizing recursive, 227-230, 247, 307 references to, 92, 110, 112, 140, 148, 223, 226, 230-231, 242, 310variadic, 198, 219, 221-222 wrapper, 218, 226 see also built-in functions and special functions FuzzyBool (example), 282–288

### G

Gamma() (math package), 66
garbage collector, 3, 32, 40, 139, 141
gc (tool), 9
gccgo (tool), 9
generic functions, 232-238; see also
higher order functions
Gerrand, Andrew, 2, 207

Getgid() (os package), 401 getters, 264–265 Getuid() (os package), 401 GID; see Getgid() Glob() (filepath package), 176, 410 globbing, file, 176, 410–411 Go building programs, 11–13 comments, 14, 51 Dashboard, 2 documentation, 8 downloading, 9 history of, 1 identifiers, 9, 42, 52-53, 58; see also blank identifier installing, 9, 10–11 shebang (#!) scripts, 10 source code encoding, 9 specification, 69 go build (tool), 11-12, 23, 409, 411 .go files, 84, 408, 410 go fix (tool), 418 go get (tool), 417-418 go install (tool), 1, 13, 409 go (statement), 45, 206, 208, 209, 211, 224, 320–357; see also goroutines go test (tool), 415-416 go version (tool), 11 go vet (tool), 418 GOARCH (constant; runtime package), 410, 424.gob files, 385-387 gob (package; encoding package), 385 GobDecoder (interface), 386 GobEncoder (interface), 386 NewDecoder(), 386NewEncoder(), 385godashboard.appspot.com (web site), 407, 417, 423, 426godoc (tool), 8, 411-413, 419 gofmt (tool), 186, 188, 419 golang.org (web site), 8, 436

GOMAXPROCS() (runtime package), 327 gonow (third-party tool), 10 Google, 1–2 App Engine, 2, 435 GOOS (constant; runtime package), 43, 176, 399, 410, 424 GOPATH (environment variable), 8, 13, 23, 408, 409, 410, 411, 417, 418GOROOT (environment variable), 8, 10, 11, 23, 408, 410, 418, 424 GOROOT() (runtime package), 424 goroutines, 3, 41, 45, 206–212, 318 - 357gorun (third-party tool), 10 goto (statement), 205 greedy matching in regular expressions, 123, 127 Griesemer, Robert, 1 grouping constants, imports, and variables, 54 grouping in regular expressions, 123guard, type switch, 198, 199 quess separator (example), 171–174 gzip (package; compress package), 400 NewReader(), 403NewWriter(), 400 Reader (type), 403 Writer (type), 400

# Η

HandleFunc() (http package), 75, 218
handling errors, 24, 32, 213
hash table; see map type
HasPrefix() (strings package), 108,
 132, 194, 246, 260, 382
HasSuffix() (strings package), 108,
 131–132, 226, 400, 403
Header (constant; xml package), 373
Header (type; tar package), 401, 404
heap (package; container package),
 421–422

hello (example), 14-16 hexadecimal number, formatting as, 98,102higher order functions, 37, 38, 238 - 244, 257history, of Go, 1 Hoare, C. A. R. 3 html (package) EscapeString(), 78 template (package), see top-level entry HTMLEscape() (template package), 78 http (package; net package) HandleFunc(), 75, 218ListenAndServe(), 75 Request (type); see top-level entry ResponseWriter (interface), 76 Humanize() (example), 100 Hyphen (constant; unicode package), 272Hypot() (math package), 66, 304

# Ι

identifiers, Go, 9, 42, 52-53, 58; see also blank identifier IEEE-754 floating-point format, 64 if (statement), 15, 189, 192–194, 220; see also switch statement Ilogb() (math package), 66 imag() (built-in), 70, 101, 187 Image (interface) draw package, 290, 293, 319 image package, 289, 293, 319 image (package), 289, 425 DecodeConfig(), 358draw (package), 290 Image (interface), 289, 293, 319 jpeg (package), 293 NewRGBA(), 290, 319 png (package), 293 RGBA (type), 290 Uniform(), 290

image (package) (continued) ZP (zero Point), 290 imagetag (exercise), 358 imaginary numbers; see complex128 type and imag() immutability, of strings, 84 import paths, 23 import (statement), 14, 15, 358, 409, 416 - 417, 418indent sort (example), 244-249 Index() reflect package, 235, 236, 428 strings package, 92, 103, 108, 133, 134, 383index operator; see [] index and slice operator IndexAny() (strings package), 108, 133, 134IndexFunc() (strings package), 92, 93, 108 indexing slices, 153–154 indexing strings, 20, 90–93 IndexRune() (strings package), 108 indirection; see pointers and \* dereference operator Inf() cmplx package, 71 math package, 66 inferred type, 53, 70 infinite channel, 208 infinite loop, 24, 203, 208; see also for loop inheritance, 240, 294, 300, 436 .ini file to map (exercise), 181 .ini files, 131, 181–182 init() (special function), 43, 215, 224-225, 231-232, 241, 242, 243, 290, 417 initializing, 27 slices, 17 variables, 15, 74 input/output (I/O); see File type and fmt package

InsertStringSlice() (example), 158 InsertStringSliceCopy() (example), 157 - 158installing Go, 9, 10–11 instances; see values Int (type; big package), 57, 61–64 Add(), 63Div(),63 Exp(), 63Mul(),63 Sub(), 63 int (type; built-in), 55, 57, 58, 59-61, 69, 116, 117, 188, 208, 215, 237,318, 390, 394comparisons, 57 conversion from int64,63 conversion to float64, 61, 69 conversion to string, 85 formatting, 98–99 literals, 53, 58 see also strconv package Int() (Value type), 428, 430int8 (type; built-in), 60, 391, 395 int16 (type; built-in), 60, 392 int32 (type; built-in), 59, 60, 388, 390, 395; see also rune type int64 (type; built-in), 60, 61, 116, 117, 215, 241, 356, 391, 401, 430 conversion from int, 63 integer literals, 53, 58 integers; see int and similar types and Int and Rat types Interface (interface; sort package), 161, 162, 246, 249, 421 interface (keyword), 220, 237, 265-274, 294, 295, 335, 364interface{}, 24, 27, 150, 165, 191, 192, 197, 198, 199, 220, 234, 235, 237, 241, 242, 243, 255,265, 284, 303, 337, 378, 388, 389,421 - 423Interface() (reflect package), 235

interfaces, 22, 202, 255, 265-274, 301, 319 see also Image, Interface, Reader, ReaderWriter, ResponseWriter, Stringer, and Writer interfaces interpreted string literals, 83 Intn() (rand package), 209, 426 introspection; see reflect package Ints() (sort package), 161, 180 IntsAreSorted() (sort package), 161 inversion, map, 170–171, 179 invoicedata (example), 362–397 invoicedata (exercise), 406 io (package), 30, 34, 424 Copy(), 353, 354, 399, 401, 402, 405EOF, 34, 35, 37, 113, 177, 268, 333, 343, 381, 404 Pipe(), 322 ReadCloser (interface), 403 Reader (interface), 32, 34, 35, 255, 268, 269–270, 364, 365, 369, 375, 380, 384, 386, 392, 393 ReaderWriter (interface), 32 WriteCloser (interface), 400 Writer (interface), 32, 34, 35, 93, 255, 354, 364, 366, 373, 378,385, 388, 399 iota (keyword), 54, 336 ioutil (package; io package), 30, 424 ReadAll(), 424ReadFile(), 38, 131-132, 424 TempFile(), 424WriteFile(), 424 Is() (unicode package), 118, 119, 258, 272IsControl() (unicode package), 119 IsDigit() (unicode package), 119 IsGraphic() (unicode package), 119 IsInf() cmplx package, 71 math package, 66

IsLetter() (unicode package), 119, 178IsLower() (unicode package), 119 IsMark() (unicode package), 119 IsNaN() cmplx package, 71 math package, 66 IsOneOf() (unicode package), 119 IsPalindrome() (exercise), 250 IsPrint() (strconv package), 114 IsPrint() (unicode package), 119 IsPunct() (unicode package), 119 IsSorted() (sort package), 161 IsSpace() (unicode package), 92, 111, 119,272IsSymbol() (unicode package), 119 IsTitle() (unicode package), 119 IsUpper() (unicode package), 119 IsValid() (Value type), 430 iterating; see for loop and range iterating arrays, 203 iterating channels, 203 iterating maps, 170, 203 iterating slices, 154–156, 203 iterating strings, 88–90, 203 Itoa() (strconv package), 85, 114, 117

# J

J0() (math package), 66 J1() (math package), 66 JavaScript Object Notation; see JSON Jn() (math package), 66 Join() (strings package), 14, 16, 55, 108, 111, 180, 414 . jpeg and . jpg files, 293 jpeg (package; image package), 293 JSON (JavaScript Object Notation), 199–202, 363, 365–371 json (package; encoding package), 202, 366 

### Κ

keywords, 52 case; see select and switch statements chan, 43, 44, 208, 209, 210, 318–357; see also channels const, 45, 53, 58, 133, 336, 364, 379, 390 default; see select and switch statements else; see if statement func, 14, 15, 25, 35, 45, 55, 208, 216, 218, 219, 223, 226, 232, 238, 240, 241, 243, 246, 291, 303, 305, 323, 324, 343, 378, 379, 388, 389, 413 interface, 220, 237, 265–274, 294, 295, 335, 364 iota, 54, 336 nil, 27, 216, 257, 305 range, 19, 38, 39, 74, 89, 110, 147, 154, 155, 168, 170, 172, 200, 203-205, 324, 325, 330, 331, 339, 343, 355, 390 struct, 42, 73, 96, 104, 132, 167, 199, 202, 222, 223, 233, 237, 241, 245, 255, 259-260, 261-262, 275-282, 284, 285, 305, 308, 326, 328, 330, 335, 337, 343, 350, 354, 362, 366, 372, 387

type, 24, 42, 55, 73, 132, 198–199, 202, 223, 245, 246, 254, 255, 256–257, 265, 284, 294, 295, 305, 335, 366, 379, 389, 412 var, 23, 53, 188, 192, 201, 241, 257, 272, 378, 382, 388 see also statements Kind (type; reflect package), 430 Kind() (Value type), 235

### L

label, 205, 331 LastIndex() (strings package), 92, 108, 194LastIndexAny() (strings package), 108 LastIndexFunc() (strings package), 92,108Ldexp() (math package), 66 left-justification, of output, 96, 98 left-leaning red-black tree; see omap example len() (built-in), 15, 20, 24, 27, 69, 85, 90, 148, 149, 151, 152, 157, 159, 165, 169, 187, 340 Len() (reflect package), 235, 430Lgamma() (math package), 66 library types; *see* standard library's types linear vs. binary search, 162–163 linefeed; see newline linkcheck (exercise), 432 linkutil (exercise), 431–432 list (package; container package), 422 - 423ListenAndServe() (http package), 75 LiteralPrefix() (Regexp type), 125literals, 58 character, 20 complex, 53, 70 composite, 18, 45, 150, 152, 153,

166, 167

floating point, 53; see also float64 type function; see closures integer, 53; see also int and similar types string, 75, 83 LittleEndian (variable; binary package), 388 Ln2 (constant; math package), 66 Ln10 (constant; math package), 66 local variables, 40, 45, 141 Lock() (RWMutex type), 346 Log() cmplx package, 71 math package, 66 log (package), 426 Fatal(), 19, 20, 32, 131–132, 342 Fatalf(), 20, 327 Printf(), 134, 217, 291 Println(), 176, 177, 353 SetFlags(), 426SetOutput(), 426Log2E (constant; math package), 66 Log10() cmplx package, 71 math package, 66 Log10E (constant; math package), 66 Log1p() (math package), 66 Log2() (math package), 66 Logb() (math package), 66 logic, short circuit, 56 logical operators; see !, &&, ^, and || operators logPanics() (example), 218 lookups, map, 39, 168–169, 231 looping; see for loop

#### Μ

.m3u files, 130-135 m3u2pls (example), 130-135 main (package), 14, 15, 206, 224 main() (special function), 14, 15, 206, 214, 215, 224-225, 327 make() (built-in), 26, 38, 39, 43, 44, 127, 129, 150, 151, 152, 157, 159,165, 172, 176, 178, 179, 187, 207, 208, 209, 211, 240, 242, 246, 247,323, 324, 328, 337, 339, 341, 346, 348, 355, 374, 392, 393, 395, 410Make2D() (exercise), 181 Map() (strings package), 108, 111-112, 132, 133, 258 map to .ini file (exercise), 182 map (type; built-in), 38, 39, 77, 127, 146, 164-171, 175-180, 199,242, 243, 298, 318, 335, 339, 345, 346, 348, 355, 356 accessing, 39, 168–169, 231 formatting, 106 inversion, 170-171, 179 iterating, 170, 203 modifying, 169 multivalued. 175 operations, 165 see also omap example Marshal() (json package), 368, 370 Match() regexp package, 121 Regexp type, 125, 333, 334 MatchReader() regexp package, 121 Regexp type, 125 MatchString() regexp package, 121 Regexp type, 125 math (package), 69 Abs(), 65, 68 Acos(), 65Acosh(), 65Asin(), 65Asinh(), 65Atan(), 65Atan2(),65 Atanh(), 65

math (package) (continued) big (package); see top-level entry Cbrt(),65 Ceil(),65 cmplx (package), see top-level entry Copysign(), 65Cos(), 45, 65 Cosh(),65 Dim(), 65E (constant), 65, 104, 105 Erf(),65 Erfc(),65 Exp(), 65Exp2(),65 Expm1(), 65 Float32bits(),65 Float32frombits(), 65 Float64bits(),65 Float64frombits(), 65 Floor(), 66Frexp(), 66Gamma(), 66 Hypot(), 66, 304Ilogb(), 66 Inf(),66 IsInf(), 66IsNaN(), 66J0(),66 J1(),66 Jn(),66 Ldexp(), 66Lgamma(), 66Ln2 (constant), 66 Ln10 (constant), 66 Log(), 66 Log2E (constant), 66 Log10(),66 Log10E (constant), 66 Log1p(),66 Log2(),66 Logb(), 66Max(),66

MaxInt32 (constant), 69, 215, 239 MaxUint8 (constant), 58 Min(), 66, 68 MinInt32 (constant), 69, 215 Mod(), 66, 68Modf(), 67, 68, 69, 70, 100 NaN(), 67, 68Nextafter(), 67, 68Phi (constant), 67, 105 Pi (constant), 45, 67, 105 Pow(), 67 Pow10(), 67 rand (package), see top-level entrv Remainder(), 67 Signbit(),67 Sin(), 45, 67 SinCos(), 67Sinh(), 67SmallestNonzeroFloat64 (constant), 68 Sqrt(), 67, 221 Sgrt2 (constant), 67 SgrtE (constant), 67 SgrtPhi (constant), 67 SgrtPi (constant), 67 Tan(), 67 Tanh(), 67Trunc(), 67Y0(),67 Y1(),67 Yn(), 67Max() (math package), 66 maximum characters to output, 96, 103MaxInt32 (constant; math package), 69, 215, 239MaxRune (constant; unicode package), 82 MaxUint8 (constant; math package), 58Memoize() (example), 242-244memoizing, 241-244

memory management; see garbage collection method expressions, 263 method sets, 22, 191, 260 MethodByName() (Value type), 430 methods, 29, 255, 258-265, 277-278 defining, 25 Error(), 31overriding, 261-262 String(), 31, 55, 96, 103, 155, 166, 260, 265, 266-267, 286 Min() (math package), 66, 68 minimum field width, 96, 103 MinInt32 (constant; math package), 69,215MkdirAll() (os package), 401, 402, 404Mod() (math package), 66 Mode() (FileInfo interface), 351, 401 ModeType (constant; os package), 351, 352Modf() (math package), 67, 69, 70, 100 modifying maps, 169 modifying slices, 147, 156-160 ModTime() (FileInfo interface), 401 Mul() (Int type), 63 multidimensional arrays, 148 multidimensional slices, 17-18, 150, 204 - 205multiple assignments, 31, 188 multivalued maps, 175 MustCompile() (regexp package), 35, 37, 121, 126, 214, 343, 348 MustCompilePOSIX() (regexp package), 121mutability, of arrays, 149 mutability, of slices, 140 mutual recursion, 227, 228-229

### Ν

Name (type; xml package), 372, 374 named fields; *see* struct keyword

named replacements in regular expressions, 126 named return values, 36, 189, 212, 219, 221, 309 named vs. unnamed custom types, 22NaN() cmplx package, 71 math package, 67 net (package), 427 http (package), see top-level entry rpc (package), 427 smtp (package), 427 url (package), 427 New() errors package, 27, 33, 194, 384 sha1 package, 353, 354 new() (built-in), 145, 146, 152, 187, 346NewDecoder() gob package, 386 json package, 369 xml package, 375 NewEncoder() gob package, 385 json package, 367 xml package, 373 NewInt() (big package), 63 newline (\n), 51, 84 NewReader() bufio package, 35, 45, 176, 333, 380 qzip package, 403 strings package, 108 tar package, 403 NewReplacer() (strings package), 108 NewRGBA() (image package), 290, 319 NewTicker() (time package), 426 NewWriter() bufio package, 35, 378 gzip package, 400 tar package, 400

NewWriter() (continued) zip package, 398 Nextafter() (math package), 67 nil (keyword), 27, 216, 257, 305 nonblocking, channel, 207, 209; see also channels nongreedy matching in regular expressions, 123, 127 normalization, of whitespace, 111 normalization, Unicode, 86 not, logical; see ! operator null; see nil number formatting, 98-101 numbers; see float64, int, and other specific numeric types NumCPU() (runtime package), 327, 328 NumGoroutine() (runtime package), 351, 353 NumSubexp() (Regexp type), 125

# 0

0 RDWR (constant; os package), 397 objects; see values octal number, formatting as, 98 Ogg container, 130 omap (example), 302-310, 409, 412-413, 414-416 Open() (File type), 31, 176, 212, 333, 342, 353, 398, 400 OpenFile() (File type), 31, 397 OpenReader() (zip package), 401, 402 operations on maps, 165 operations on slices, 151 operators ! logical NOT, 57 != inequality, 56–57, 68–69, 70, 164% modulus and formatting placeholder, 47, 60, 69; see also format specifier %= augmented modulus, 60

& address of and bitwise AND, 45, 46, 55, 60, 142, 143, 144, 167, 246, 247, 248, 267, 269, 284, 382, 383, 384, 387, 393, 394, 395&& logical AND, 56, 57 &= augmented bitwise AND, 60 &^ bitwise clear, 60 & = augmented bitwise clear, 60 \* multiplication, dereference, pointer declaration and formatting placeholder, 26, 59, 69, 96, 100, 142, 143, 144, 178, 247, 248, 249, 259, 284, 305, 370, 382, 394 \*= augmented multiplication, 59, 147+ addition, concatenation, and unary plus, 20, 59, 85, 226 ++ increment, 20, 59, 186, 188 += augmented addition and string append, 20, 59, 84, 85, 88,140 - subtraction and unary minus, 59-- decrement, 20, 59, 186, 188 -= augmented subtraction, 59 . selector, 148, 275 ... ellipsis, 149, 156, 158, 160, 176, 219, 221, 222, 233, 242, 268, 287, 378 / division, 59 /= augmented division, 59 := short variable declaration, 15, 18, 36, 53, 140, 188, 189, 198, 203< less than comparison, 56–57 <- send/receive communication, 44, 45, 207, 210, 318-357 << bitwise shift left, 55, 60 <= augmented bitwise shift left, 60 <= less than or equal comparison, 56 - 57

= assignment, 16, 18, 36, 188, 212== equality, 56–57, 68–69, 70, 164> greater than comparison, 56 - 57>= greater than or equal comparison, 56–57 >> bitwise right shift, 60 >>= augmented bitwise right shift.60 [] index and slice, 15, 28, 85, 91, 203, 242, 339, 355, 357, 393 ^ bitwise xor and complement, 60 ^= augmented bitwise xor, 60 | bitwise or, 55, 60 |= augmented bitwise OR, 60 || logical OR, 56, 57, 178 overloading, 61 optional parameters, 222–223 optional statement, 193, 195, 198, 203or, logical; see || operator order of declarations and definitions, 21ordered comparisons (<, <=, >=, >), 56 - 57ordered map; see omap example os (package), 423 Args (slice), 14, 15, 17, 19, 131 - 132, 232exec (package), see top-level entrv Exit(), 19, 20, 32, 131–132, 327 File (type); see top-level entry FileInfo (interface), see top-level entry Getgid(), 401Getuid(), 401MkdirAll(), 401, 402, 404 ModeType (constant), 351, 352 0 RDWR (constant), 397 Stderr (stream), 20, 32, 46

Stdin (stream), 31, 32 Stdout (stream), 31, 32, 46, 94, 131 oslice (exercise), 313–314 overloading, 258 overloading, operator, 61 overriding methods, 261–262

#### Ρ

pack (example), 397-405 package, 14, 215–216, 407–431 aliasing names, 409, 418 collection, 421-423; see also container package custom, 24-29, 408-417 documenting, 411-413 main, 14, 15 third-party, 417-418 variables. 18 see also bufio, bytes, cmplx, container, crypto, draw, encoding, errors, filepath, fmt, http, image, io, ioutil, json, math, net, os, rand, regexp, reflect, runtime, sha1, sort, strings, sync, and time packages package (statement), 408, 412 Pad() (example), 99 padding, of output, 96, 98 palindrome (example), 232 panic() (built-in), 32, 69, 70, 113, 187, 191, 196, 213–218, 219, 220, 243parameters, 22, 141–142, 220–223, 254 - 255Parse() (time package), 370, 376, 377, 383, 395 ParseBool() (strconv package), 98, 115, 116ParseFloat() (strconv package), 77, 115, 116ParseForm() (Request type), 76

ParseInt() (strconv package), 115, 116ParseUInt() (strconv package), 115, 116patents, software, 437-439 PATH (environment variable), 10 path, import, 23 path (package), 424 path/filepath package; see filepath package Phase() (cmplx package), 9 Phi (constant; math package), 67, 105 Pi (constant; math package), 45, 67, 105pi by digits (example), 62-64 Pike, Rob, 1, 385, 431 Pipe() (io package), 322 placeholder (%, \*); see %, Printf(), and Sprintf(); see also blank identifier plain text files; see .txt files platform-specific code, 410-411 playlists (excercise), 135–136 .pls files, 130-135 .png files, 293 png (package; image package), 293 pointers, 26, 27, 28, 29, 139, 141-148, 152, 167, 247 - 248, 260, 267, 285,318, 362, 369 formatting, 96, 104 Polar() (cmplx package), 9 polar2cartesian (example), 40-47 polymorphism, 198 Porter-Duff image compositing, 290 Pow() cmplx package, 71 math package, 67 Pow10() (math package), 67 predefined identifiers, 52 Print() (fmt package), 94, 96 Printf() fmt package, 19, 47, 94–106, 113, 178, 192

log package, 134, 217, 291 Println() fmt package, 19, 24, 45, 53, 72, 94–106 log package, 176, 177, 353 private; *see* unexported identifiers ProbablyPrime() (big package), 425 public; *see* exported identifiers pure functions, 241; *see also* memoizing

# Q

# R

rand (package) crypto package, 426 math package, 426 Intn(), 209, 426 range (keyword), 19, 38, 39, 74, 89, 110, 147, 154, 155, 168, 170, 172, 200, 203-205, 324, 325, 330, 331,339, 343, 355, 390 Rat (type; big package), 57, 61 rationals; see Rat type raw string (``), 75, 96 RE2 regular expression engine; see regexp package Read() (binary package), 391, 393, 395 ReadAll() (ioutil package), 424 ReadAt() (File type), 397

ReadBytes() (Reader type), 333, 334 ReadCloser (interface; io package), 403ReadCloser (type; zip package), 401 Readdir() (File type), 361 Readdirnames() (File type), 361 Reader (interface; io package), 32, 34, 35, 255, 268, 269-270, 364, 365, 369, 375, 380, 384, 386, 392, 393 Reader (type) bufio package, 35, 38, 177 ReadBytes(), 333, 334 ReadString(), 35, 37, 45, 177, 343, 381 gzip package, 403 strings package, 113 tar package, 403, 405 ReaderWriter (interface; io package), 32ReadFile() (ioutil package), 38, 131 - 132reading files; see File type and ioutil package ReadRune() (Buffer type), 113 ReadString() (Reader type), 35, 37, 45, 177, 343, 381real() (built-in), 70, 101, 187 real numbers; see float64 and complex128 types and real() receive, channel; see channels, <- operator, and chan keyword receiver, 25, 28, 258, 261, 266, 267, 269, 277, 367recover() (built-in), 32, 187, 213 - 218Rect() (cmplx package), 9 recursion, mutual, 227, 228-229 recursive functions, 227–230, 247, 307red-black tree; *see* omap example references, 26, 27, 39, 92, 110, 112, 140, 141, 146, 148, 150, 153, 223,226, 230-231, 242, 310, 318

reflect (package), 235–236, 427 - 431DeepEqual(), 57, 236, 427Kind (type), 430 Slice (constant), 235 TypeOf(), 427, 428, 430 Value (type); see top-level entry ValueOf(), 235, 427, 428, 429, 430 regexp (package), 36, 120–129, 214 assertions, zero-width, 122 character classes, 122 Compile(), 37, 121, 214, 327 CompilePOSIX(), 121 escapes, 121 flags, 123 greedy and nongreedy matching, 123, 127grouping, 123 Match(), 121 MatchReader(), 121MatchString(), 121 MustCompile(), 35, 37, 121, 126, 214, 343, 348MustCompilePOSIX(), 121 quantifiers, 123 QuoteMeta(), 121, 128 Regexp (type); see top-level entry zero-width assertions, 122 Regexp (type; regexp package), 35, 37, 318, 328, 344 Expand(), 124ExpandString(), 124 Find(), 124 FindAll(), 124FindAllIndex(), 124 FindAllString(), 124, 127 FindAllStringIndex(), 124 FindAllStringSubmatch(), 124, 127 FindAllStringSubmatchIndex(), 124, 128FindAllSubmatch(), 124 FindAllSubmatchIndex(), 124FindIndex(), 124

Regexp (type; regexp package) (continued) FindReaderIndex(), 124 FindReaderSubmatchIndex(), 124 FindString(), 124 FindStringIndex(), 124 FindStringSubmatch(), 124, 127, 343, 344, 348 FindStringSubmatchIndex(), 124 FindSubmatch(), 125 FindSubmatchIndex(), 125 LiteralPrefix(), 125Match(), 125, 333, 334MatchReader(), 125MatchString(), 125 NumSubexp(), 125ReplaceAll(), 120, 125 ReplaceAllFunc(), 125ReplaceAllLiteral(), 125 ReplaceAllLiteralString(), 125, 128 - 129ReplaceAllString(), 120, 125, 126, 129ReplaceAllStringFunc(), 35, 36-37, 125, 129, 359String(), 125SubexpNames(), 125Remainder() (math package), 67 remote procedure call (RPC), 427 RemoteAddr (field; Request type), 217 RemoveStringSlice() (example), 160 RemoveStringSliceCopy() (example), 159 - 160Repeat() (strings package), 48, 99, 108, 243, 246 Replace() (strings package), 76, 77, 109, 110, 399 ReplaceAll() (Regexp type), 120, 125ReplaceAllFunc() (Regexp type), 125ReplaceAllLiteral() (Regexp type), 125ReplaceAllLiteralString() (Regexp type), 125, 128-129

ReplaceAllString() (Regexp type), 120, 125, 126, 129 ReplaceAllStringFunc() (Regexp type), 35, 36-37, 125, 129, 359 replacement character, Unicode (U+FFFD), 85, 118 replacements, \$ in regular expressions, 120, 126, 129 reporting errors; see error handling Request (type; http package), 76 Form (field), 76 ParseForm(), 76 RemoteAddr (field), 217 ResponseWriter (interface; http package), 76 return (statement), 28, 34, 70, 186, 189, 194, 219, 220, 240, 309 return values, 24, 28, 31, 33, 40, 45, 53, 145, 189, 219 bare, 34, 189, 219 named, 36, 212, 221, 309 unnamed, 190 RGBA (color type), 147 RGBA (image type), 290 right-justification, of output, 96, 98 ring (package; container package), 423 RLock() (RWMutex type), 346 RomanForDecimal() (example), 243 - 244rpc (package; net package), 427 rune (type; built-in), 59, 60, 82, 83, 87–90, 104, 190, 230, 246, 420 conversion to string, 87–88, 89-90, 91, 246, 272 formatting, 96, 99 literal, 84 see also int32 and string types RuneCount() (utf8 package), 118 RuneCountInString() (utf8 package), 85, 99, 177, 178, 229 RuneLen() (utf8 package), 118

RuneStart() (utf8 package), 118 RUnlock() (RWMutex type), 346 runtime (package), 424 Caller(), 291 FuncForPC(), 291, 292 GOARCH (constant), 410, 424 GOMAXPROCS(), 327G00S (constant), 43, 176, 399, 410, 424 GOROOT(), 424NumCPU(), 327, 328 NumGoroutine(), 351, 353Version(), 424runtime system, Go's, 32 RWMutex (type; sync package), 345, 346Lock(), 346 RLock(), 346 RUnlock(), 346Unlock(), 346

### $\mathbf{S}$

safemap (example), 334-340 safeslice (exercise), 357–358 Scan() (fmt package), 383 Scanf() (fmt package), 383 Scanln() (fmt package), 383 scientific notation, 64, 101 scope, 141, 225, 239, 240 scoping problem; see shadow variables Search() (sort package), 161, 163 SearchFloat64s() (sort package), 161 searching, slices, 162–164 searching, strings, 87 SearchInts() (sort package), 161 SearchStrings() (sort package), 161 Seek() (File type), 397 select (statement), 209–212, 321, 331, 333 self; see receiver semicolon (;), 15, 186

send, channel; see channels, <- operator, and chan keyword Separator (constant; filepath package), 134, 135 serialized access, 318-319, 335, 341 SetFlags() (log package), 426 SetInt() (Value type), 429 SetOutput() (log package), 426 SetString() (Value type), 428 setters, 264-265 SHA-1 (Secure Hash Algorithm), 349, 354-53 sha1 (package; crypto package) New(), 353, 354 Size (constant), 355 shadow variables, 36, 189, 192, 200, 201, 281, 282, 300, 301 shaper (example), 289-301 shaper (exercise), 311–313 shebang (#!) scripts, 10 short circuit logic, 56 short variable declaration, 15, 53, 140, 188 Signbit() (math package), 67 Simple Mail Transport Protocol (SMTP), 427 simple statement, 193, 195, 203 SimpleFold() (unicode package), 119 simplification, of whitespace, 111, 128 - 129SimplifyWhitespace() (example), 111 Sin() cmplx package, 71 math package, 45, 67 SinCos() (math package), 67 Sinh() cmplx package, 71 math package, 67 Size (constant; shal package), 355 Size() (FileInfo interface), 351, 353, 401sizeimages (exercise), 359

Slice (constant; reflect package), 235slice operator; see [] index and slice operator SliceIndex() (example), 238–239 slices, 17, 140, 146–147, 149–164, 232, 234, 318 formatting, 101–103 indexing, 153-154 initializing, 17 iterating, 154–156, 203 modifying, 147, 156-160 multidimensional, 17-18, 150, 204 - 205mutability, 140 operations on, 151 searching, 162–164 slicing, 153–154 sorting, 160–164 see also under types, e.g., byte for []byte and string for []string slicing, slices, 153–154 slicing, strings, 90-93 SmallestNonzeroFloat64 (constant; math package), 68 smtp (package; net package), 427 sockets; see net package software patents, 437-439 sort (package), 160–164, 246 Float64s(), 73, 161 Float64sAreSorted(), 161 Interface (interface), 161, 162, 246, 249, 421 Ints(), 161, 180 IntsAreSorted(), 161 IsSorted(), 161 Search(), 161, 163 SearchFloat64s(), 161 SearchInts(), 161 SearchStrings(), 161 Sort(), 161, 162, 248, 249

Strings(), 160, 161, 163, 170, 178, 180, 356 StringsAreSorted(), 161 Sort() (sort package), 161, 162, 248, 249sorted map; see omap example sorting, slices, 160–164 sorting, strings, 87 soundex (exercise), 136-137 source code encoding, Go, 9 special functions init(), 43, 215, 224-225, 231-232, 241, 242, 243, 290,417 main(), 14, 15, 206, 214, 215, 224-225, 327 see also built-in functions and functions specification, Go, 69 Split() (strings package), 38, 39, 107, 109, 132, 133, 422 SplitAfter() (strings package), 107, 109SplitAfterN() (strings package), 107, 109SplitN() (strings package), 39, 107, 109Sprint() (fmt package), 94, 99, 178, 242,357Sprintf() (fmt package), 43, 55, 69, 78, 85, 94, 97, 100, 101, 242, 286, 355 Sprintln() (fmt package), 94 sql (package; database package), 423 Sqrt() cmplx package, 71 math package, 67 Sqrt2 (constant; math package), 67 SqrtE (constant; math package), 67 SqrtPhi (constant; math package), 67 SgrtPi (constant; math package), 67 Sscan() (fmt package), 383 Sscanf() (fmt package), 45, 46, 382, 383

Sscanln() (fmt package), 383 stack trace, 214 stacker (example), 21–29, 408–409 standard library, 419–431 types; *see* File, Int, Rat, Reader, Regexp, and Writer startup, application; see entry point stat call; see FileInfo interface Stat() (File type), 397, 398, 399, 400 statements break, 24, 177, 186, 204, 205, 331continue, 132, 133, 186, 204, 205, 324defer, 31–32, 35, 36, 44, 176, 211-213, 216, 218, 333, 343, 353, 378, 398, 400 fallthrough, 186, 195, 196-197 for loop, 19, 23, 24, 38, 39, 74, 89, 110, 132, 147, 154, 155, 168, 170, 172, 177, 186, 200, 203-205, 320, 321, 324, 325, 330, 331, 339, 343, 355, 390 go, 45, 206, 208, 209, 211, 224, 320–357; see also goroutines goto, 205 if, 15, 189, 192–194, 220; see also switch statement import, 14, 15, 358, 409, 416-417, 418optional, 193, 195, 198, 203 package, 408, 412 return, 28, 34, 70, 186, 189, 194, 219, 220, 240, 309 select, 209-212, 321, 331, 333 simple, 193, 195, 203 switch, 110, 129, 174, 195-202, 220, 233, 235, 282, 285, 365; see also if statement terminator, (;), 186 see also keywords statistics (example), 72–78 statistics (exercise), 79 statistics nonstop (example), 216 - 218

Stderr stream (os package), 20, 32, 46 Stdin stream (os package), 31, 32, 46 Stdout stream (os package), 31, 32, 94,131strconv (package), 113–117 AppendBool(), 114 AppendFloat(), 114 AppendInt(), 114 AppendQuote(), 114 AppendQuoteRune(), 114 AppendQuoteRuneToASCII(), 114 AppendUInt(), 114 Atoi(), 116, 134, 390 CanBackguote(), 114 FormatBool(), 114, 116 FormatFloat(), 114 FormatInt(), 114, 117 FormatUInt(), 114 IsPrint(), 114 Itoa(), 85, 114, 117 ParseBool(), 98, 115, 116 ParseFloat(), 77, 115, 116 ParseInt(), 115, 116 ParseUInt(), 115, 116 Quote(), 115, 117 QuoteRune(), 115 QuoteRuneToASCII(), 115 Unquote(), 115, 117 UnquoteChar(), 115see also conversions String() Buffer type, 88, 200, 243 Regexp type, 125 Value type, 428, 429 String() (method), 31, 55, 96, 103. 155, 166, 260, 265, 266-267,286string (type; built-in), 20, 81–129, 140, 190, 238, 318 comparisons, 57, 84, 86-87 concatenation; see + and += operators concatenation, fast, 88

string (type; built-in) (continued) conversion from []byte, 38, 85, 164, 334, 373, 395conversion from []rune, 85, 91 conversion from float64; see ParseFloat() conversion from rune, 89-90, 246 conversion to []byte, 85, 89–90, 164,391conversion to []rune, 85, 87-88, 246, 272conversion to int, 85 formatting, 101–103 immutability, 84, 140 indexing, 20, 90-93 interpreted literals, 83 iterating, 88–90, 203 literals, 75, 83 raw (``), 75, 78, 96 searching, 87 slicing, 90-93 sorting, 87 see also byte and rune types Stringer (interface; fmt package), 265, 266-267, 286 strings package, 91, 107-113, 419 Contains(), 108 Count(), 108, 172, 173 EqualFold(), 108, 163 Fields(), 38, 39, 76, 77, 107–110, 108, 111FieldsFunc(), 108, 178 HasPrefix(), 108, 132, 194, 246, 260, 382HasSuffix(), 108, 131–132, 226, 400, 403Index(), 92, 103, 108, 133, 134, 383 IndexAny(), 108, 133, 134IndexFunc(), 92, 93, 108 IndexRune(), 108 Join(), 14, 16, 55, 108, 111, 180, 414

LastIndex(), 92, 108, 194 LastIndexAny(), 108 LastIndexFunc(), 92, 108Map(), 108, 111-112, 132, 133,258NewReader(), 108, 113 NewReplacer(), 108Reader (type); see top-level entry Repeat(), 48, 99, 108, 243, 246 Replace(), 76, 77, 109, 110, 399 Split(), 38, 39, 107, 109, 132, 133, 422SplitAfter(), 107, 109 SplitAfterN(), 107, 109 SplitN(), 39, 107, 109 Title(), 109 ToLower(), 109, 162, 163, 177, 194, 246, 259, 293, 306, 324 ToTitle(), 109 ToUpper(), 37, 109, 259, 302 Trim(), 109 TrimFunc(), 109 TrimLeft(), 109, 399 TrimLeftFunc(), 109 TrimRight(), 109, 127 TrimRightFunc(), 109 TrimSpace(), 111, 132, 177, 246, 376, 383 Strings() (sort package), 160, 161, 163, 170, 178, 180, 356StringsAreSorted() (sort package), 161 strong typing, 15, 20, 24 struct (keyword), 42, 73, 96, 104, 132, 167, 199, 202, 222, 223, 245, 254, 255, 259-260, 261-262, 275-282, 284, 285, 305, 308, 326, 328, 330, 335, 337, 343, 350, 354,362, 366, 372, 387 Sub() (Int type), 63 SubexpNames() (Regexp type), 125substrings; see string type's slicing suffix, file, 194 swapping values, 188

switch (statement), 110, 174, 195-202, 220, 233, 235, 282, 285, 365; see also if statement sync (package) RWMutex (type); see top-level entry WaitGroup (type), see top-level en- try synchronization, 44, 315, 318, 321 synchronous, channel, 207; see also channels

#### Т

tab (\t), 84 tags, struct, 279, 371, 372, 428 Tan() cmplx package, 71 math package, 67 Tanh() cmplx package, 71 math package, 67 .tar files, 399-401, 403-405 tar (package; archive package), 399-401, 403-405 Header (type), 401, 404 NewReader(), 403NewWriter(), 400 Reader (type), 403, 405 Writer (type); see top-level entry Taylor, Ian Lance, 2 TCP/IP; see net package TempFile() (ioutil package), 424 template (package; html package), 420 - 421template (package; text package), 420 - 421HTMLEscape(), 78terminal; see Stderr, Stdin, and Stdout streams Terminal Punctuation (constant; unicode package), 258 termination, 15, 317, 321 terminator, statement (;), 186 testing (package), 414-416

text; see string type text files; see .txt files text (package) template (package), see top-level entry third-party packages, 417–418 this: see receiver Thompson, Ken, 1 threads; see channels, goroutines, and the chan and go keywords Tick() (time package), 426 time (package), 426 After(), 332, 333, 426 Duration (type), 332, 333 NewTicker(), 426Parse(), 370, 376, 377, 383, 395 Tick(), 426 Time (type); *see* top-level entry Unix(),391 Time (type; time package), 362, 363, 368, 377, 383, 390, 394, 395, 426 Format(), 368, 379, 390Unix(), 391 Title() (strings package), 109 To() (unicode package), 119 ToLower() strings package, 109, 162, 163, 177, 194, 246, 259, 293, 306, 324unicode package, 119, 272 tools 5g, 6g, 8g, 9 51,61,81,9 cgo, 9 gc, 9gccgo, 9 go build, 11-12, 23, 409, 411 go fix, 418 go get, 417-418 go install, 1, 13, 409 go test, 415-416 go version, 11 go vet, 418

tools (continued) godoc, 8, 411-413, 419 gofmt, 186, 188, 419 tools (third-party) gonow, 10 gorun, 10 ToSlash() (filepath package), 399 ToTitle() strings package, 109 unicode package, 119 ToUpper() strings package, 37, 109, 259, 302unicode package, 119, 272 trace, stack, 214 Trim() (strings package), 109 TrimFunc() (strings package), 109 TrimLeft() (strings package), 109, 399TrimLeftFunc() (strings package), 109TrimRight() bytes package, 333, 334 strings package, 109, 127 TrimRightFunc() (strings package), 109 TrimSpace() (strings package), 111, 132, 177, 246, 376, 383true (built-in); see bool type Trunc() (math package), 67 .txt files, 377–384 type abstract vs. concrete, 22 deduction, 18 method sets; see 22, 191, 260; see also methods see also built-in types, custom types, and standard library's types type assertions, 191–192, 200, 233, 234, 237, 242, 300, 319 type conversions, 190–191; see also conversions and the strconv package

type, inference, 53, 70 type (keyword), 24, 42, 55, 73, 132, 161, 190, 198–199, 202, 223, 233, 237, 241, 245, 246, 254, 255,256-257, 265, 284, 294, 295, 305, 335, 366, 379, 389, 412 type modifier; see pointers and \* pointer declaration operator type safety; see duck typing and interfaces type switch guard, 198, 199 type switches, 197–202, 233, 235, 282, 285; see also switch statement TypeOf() (reflect package), 427, 428, 430typing, duck, 21, 32, 268 typing, strong, 15, 20, 24

### U

UDP; see net package UID; see Getuid() uint (type; built-in), 60, 69 uint8 (type; built-in); see byte type uint16 (type; built-in), 60, 388, 393 uint32 (type; built-in), 60, 388, 393 uint64 (type; built-in), 60 uintptr (type; built-in), 60 unbound methods (method expressions). 263 unchecked type assertion; see type assertions unexported identifiers, 52, 264 Unicode, 52, 82-84, 86-87 normalization, 86 U+FFFD replacement character, 85, 118 whitespace, 92 Unicode code point; *see* rune type unicode (package), 118, 420 Hyphen (constant), 272 Is(), 118, 119, 258, 272 IsControl(), 119

IsDigit(), 119 IsGraphic(), 119 IsLetter(), 119, 178 IsLower(), 119IsMark(), 119 IsOneOf(), 119 IsPrint(), 119 IsPunct(), 119 IsSpace(), 92, 111, 119, 272 IsSymbol(), 119 IsTitle(), 119 IsUpper(), 119 MaxRune (constant), 82 SimpleFold(), 119 Terminal Punctuation (constant), 258To(), 119 ToLower(), 119, 272ToTitle(), 119 ToUpper(), 119, 272utf8 (package), see top-level entrv utf16 (package), 420 Uniform() (image package), 290 UniqueInts() (exercise), 180 unit testing, 414–415 Unix() time package, 391 Time type, 391 Unlock() (RWMutex type), 346 Unmarshal() (json package), 199, 201, 202, 369, 370 unnamed return values, 190, 219 unnamed struct, 275 unnamed vs. named custom types, 22unpack (example), 397-405 unpack (exercise), 405 Unquote() (strconv package), 115, 117UnquoteChar() (strconv package), 115 untyped constants; see under literals and specific types

url (package; net package), 427 US-ASCII encoding, 82 UTF-8; see string type and Unicode utf8 (package; unicode package), 117, 420DecodeLastRune(), 118DecodeLastRuneInString(), 91, 118, 229, 230 DecodeRune(), 118 DecodeRuneInString(), 91, 92, 93, 203, 229, 230 EncodeRune(), 118 FullRune(), 118 FullRuneInString(), 118 RuneCount(), 118 RuneCountInString(), 85, 99, 177, 178,229RuneLen(), 118RuneStart(), 118 UTFMax (constant), 177 Valid(),118 ValidString(), 118 utf16 (package; unicode package), 420utf16-to-utf8 (exercise), 405 UTFMax (constant; utf8 package), 177

#### V

Value (type; reflect package) (continued) IsValid(), 430Kind(), 235 Len(), 235, 430 MethodByName(), 430SetInt(), 429SetString(), 428 String(), 428, 429 ValueOf() (reflect package), 235, 427, 428, 429, 430 values, 140-148, 255; see also variables values, swapping, 188 var (keyword), 23, 53, 188, 192, 201, 241, 257, 272, 378, 382, 388 variable vs. fixed-width character encoding, 83 variables, 140-148, 242, 265, 290 declaration, short, 15, 53, 140, 188initializing, 15, 74 local, 40, 45, 141 package, 18 shadow, 36, 189, 192, 200, 201, 281, 282, 300, 301 see also environment variables variadic function, 198, 219, 221–222 variant; see interface{} Version() (runtime package), 424vertial tab (\v), 84 virtual functions, 254 Vorbis audio format, 130

#### W

web applications, 2, 72, 79-80 web sites endsoftpatents.org, 439 godashboard.appspot.com, 407, 417, 423, 426 golang.org, 8, 436www.nosoftwarepatents.com, 439www.gtrac.eu,1 while loop; see for loop whitespace, 92, 111, 128-129 wordfrequencies (example), 174–180 wrapper function, 218, 226 Write() (binary package), 388 WriteAt() (File type), 397 WriteCloser (interface; io package), 400 WriteFile() (ioutil package), 424 WriteHeader() (Writer type), 401 Writer (interface; io package), 32, 34, 35, 93, 255, 354, 364, 366, 373, 378, 385, 388, 399 Writer (type) bufio package, 35, 38 Flush(), 35, 36, 378 WriteString(), 35, 37gzip package, 400 tar package, 400 WriteHeader(), 401zip package, 397, 398 CreateHeader(), 398, 399WriteRune() (Buffer type), 111 WriteString() Buffer type, 88, 111, 200, 243 Writer type, 35, 37 writing files; see File type and ioutil package

### Х

XML format, 363, 371-377 xml (package; encoding package), 371, 372 Decoder (type); *see* top-level entry Encoder (type); *see* top-level entry Header (constant), 373 Name (type), 372, 374 NewDecoder(), 375 NewEncoder(), 373 xor, logical; *see* ^ operator

# Y

Y0() (math package), 67 Y1() (math package), 67 Yn() (math package), 67

# Ζ

zero, division by, 68 zero value, 27, 33, 39, 53, 54, 149, 150, 152, 168, 173, 189, 191, 216, 222, 257, 263, 275, 308, 346, 383 zero-width assertions in regular expressions, 122 .zip files, 397-399, 401-403 zip (package; archive package), 397-399, 401-403 FileHeader (type), 399 FileInfoHeader(), 398, 399 NewWriter(), 398 OpenReader(), 401, 402 ReadCloser (type), 401 Writer (type); see top-level entry ZP (zero Point; image package), 290