

The GO Programming Language

Alan A. A. Donovan
Brian W. Kernighan



FREE SAMPLE CHAPTER



SHARE WITH OTHERS

The Go Programming Language

This page intentionally left blank

The Go Programming Language

Alan A. A. Donovan
Google Inc.

Brian W. Kernighan
Princeton University

◆Addison-Wesley

New York • Boston • Indianapolis • San Francisco
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 authors 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.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the United States, please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2015950709

Copyright © 2016 Alan A. A. Donovan & Brian W. Kernighan

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, 200 Old Tappan Road, Old Tappan, New Jersey 07675, or you may fax your request to (201) 236-3290.

Front cover: Millau Viaduct, Tarn valley, southern France. A paragon of simplicity in modern engineering design, the viaduct replaced a convoluted path from capital to coast with a direct route over the clouds. © Jean-Pierre Lescourret/Corbis.

Back cover: the original Go gopher. © 2009 Renée French. Used under Creative Commons Attributions 3.0 license.

Typeset by the authors in Minion Pro, Lato, and Consolas, using Go, groff, ghostscript, and a host of other open-source Unix tools. Figures were created in Google Drawings.

ISBN-13: 978-0-13-419044-0

ISBN-10: 0-13-419044-0

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

First printing, October 2015

For Leila and Meg

This page intentionally left blank

Contents

Preface	xi
The Origins of Go	xii
The Go Project	xiii
Organization of the Book	xv
Where to Find More Information	xvi
Acknowledgments	xvii
1. Tutorial	1
1.1. Hello, World	1
1.2. Command-Line Arguments	4
1.3. Finding Duplicate Lines	8
1.4. Animated GIFs	13
1.5. Fetching a URL	15
1.6. Fetching URLs Concurrently	17
1.7. A Web Server	19
1.8. Loose Ends	23
2. Program Structure	27
2.1. Names	27
2.2. Declarations	28
2.3. Variables	30
2.4. Assignments	36
2.5. Type Declarations	39
2.6. Packages and Files	41
2.7. Scope	45

3. Basic Data Types	51
3.1. Integers	51
3.2. Floating-Point Numbers	56
3.3. Complex Numbers	61
3.4. Booleans	63
3.5. Strings	64
3.6. Constants	75
4. Composite Types	81
4.1. Arrays	81
4.2. Slices	84
4.3. Maps	93
4.4. Structs	99
4.5. JSON	107
4.6. Text and HTML Templates	113
5. Functions	119
5.1. Function Declarations	119
5.2. Recursion	121
5.3. Multiple Return Values	124
5.4. Errors	127
5.5. Function Values	132
5.6. Anonymous Functions	135
5.7. Variadic Functions	142
5.8. Deferred Function Calls	143
5.9. Panic	148
5.10. Recover	151
6. Methods	155
6.1. Method Declarations	155
6.2. Methods with a Pointer Receiver	158
6.3. Composing Types by Struct Embedding	161
6.4. Method Values and Expressions	164
6.5. Example: Bit Vector Type	165
6.6. Encapsulation	168
7. Interfaces	171
7.1. Interfaces as Contracts	171
7.2. Interface Types	174
7.3. Interface Satisfaction	175
7.4. Parsing Flags with <code>flag.Value</code>	179
7.5. Interface Values	181

7.6. Sorting with <code>sort.Interface</code>	186
7.7. The <code>http.Handler</code> Interface	191
7.8. The error Interface	196
7.9. Example: Expression Evaluator	197
7.10. Type Assertions	205
7.11. Discriminating Errors with Type Assertions	206
7.12. Querying Behaviors with Interface Type Assertions	208
7.13. Type Switches	210
7.14. Example: Token-Based XML Decoding	213
7.15. A Few Words of Advice	216
8. Goroutines and Channels	217
8.1. Goroutines	217
8.2. Example: Concurrent Clock Server	219
8.3. Example: Concurrent Echo Server	222
8.4. Channels	225
8.5. Looping in Parallel	234
8.6. Example: Concurrent Web Crawler	239
8.7. Multiplexing with <code>select</code>	244
8.8. Example: Concurrent Directory Traversal	247
8.9. Cancellation	251
8.10. Example: Chat Server	253
9. Concurrency with Shared Variables	257
9.1. Race Conditions	257
9.2. Mutual Exclusion: <code>sync.Mutex</code>	262
9.3. Read/Write Mutexes: <code>sync.RWMutex</code>	266
9.4. Memory Synchronization	267
9.5. Lazy Initialization: <code>sync.Once</code>	268
9.6. The Race Detector	271
9.7. Example: Concurrent Non-Blocking Cache	272
9.8. Goroutines and Threads	280
10. Packages and the Go Tool	283
10.1. Introduction	283
10.2. Import Paths	284
10.3. The Package Declaration	285
10.4. Import Declarations	285
10.5. Blank Imports	286
10.6. Packages and Naming	289
10.7. The Go Tool	290

11. Testing	301
11.1. The go test Tool	302
11.2. Test Functions	302
11.3. Coverage	318
11.4. Benchmark Functions	321
11.5. Profiling	323
11.6. Example Functions	326
12. Reflection	329
12.1. Why Reflection?	329
12.2. <code>reflect.Type</code> and <code>reflect.Value</code>	330
12.3. <code>Display</code> , a Recursive Value Printer	333
12.4. Example: Encoding S-Expressions	338
12.5. Setting Variables with <code>reflect.Value</code>	341
12.6. Example: Decoding S-Expressions	344
12.7. Accessing Struct Field Tags	348
12.8. Displaying the Methods of a Type	351
12.9. A Word of Caution	352
13. Low-Level Programming	353
13.1. <code>unsafe.Sizeof</code> , <code>Alignof</code> , and <code>Offsetof</code>	354
13.2. <code>unsafe.Pointer</code>	356
13.3. Example: Deep Equivalence	358
13.4. Calling C Code with <code>cgo</code>	361
13.5. Another Word of Caution	366
Index	367

Preface

“Go is an open source programming language that makes it easy to build simple, reliable, and efficient software.” (From the Go web site at golang.org)

Go was conceived in September 2007 by Robert Griesemer, Rob Pike, and Ken Thompson, all at Google, and was announced in November 2009. The goals of the language and its accompanying tools were to be expressive, efficient in both compilation and execution, and effective in writing reliable and robust programs.

Go bears a surface similarity to C and, like C, is a tool for professional programmers, achieving maximum effect with minimum means. But it is much more than an updated version of C. It borrows and adapts good ideas from many other languages, while avoiding features that have led to complexity and unreliable code. Its facilities for concurrency are new and efficient, and its approach to data abstraction and object-oriented programming is unusually flexible. It has automatic memory management or *garbage collection*.

Go is especially well suited for building infrastructure like networked servers, and tools and systems for programmers, but it is truly a general-purpose language and finds use in domains as diverse as graphics, mobile applications, and machine learning. It has become popular as a replacement for untyped scripting languages because it balances expressiveness with safety: Go programs typically run faster than programs written in dynamic languages and suffer far fewer crashes due to unexpected type errors.

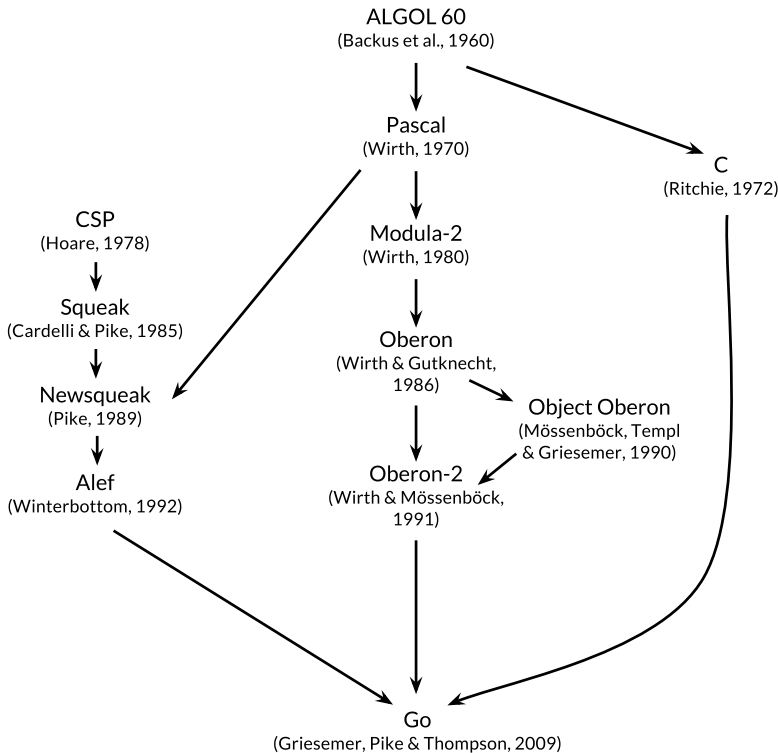
Go is an open-source project, so source code for its compiler, libraries, and tools is freely available to anyone. Contributions to the project come from an active worldwide community. Go runs on Unix-like systems—Linux, FreeBSD, OpenBSD, Mac OS X—and on Plan 9 and Microsoft Windows. Programs written in one of these environments generally work without modification on the others.

This book is meant to help you start using Go effectively right away and to use it well, taking full advantage of Go’s language features and standard libraries to write clear, idiomatic, and efficient programs.

The Origins of Go

Like biological species, successful languages beget offspring that incorporate the advantages of their ancestors; interbreeding sometimes leads to surprising strengths; and, very occasionally, a radical new feature arises without precedent. We can learn a lot about why a language is the way it is and what environment it has been adapted for by looking at these influences.

The figure below shows the most important influences of earlier programming languages on the design of Go.



Go is sometimes described as a “C-like language,” or as “C for the 21st century.” From C, Go inherited its expression syntax, control-flow statements, basic data types, call-by-value parameter passing, pointers, and above all, C’s emphasis on programs that compile to efficient machine code and cooperate naturally with the abstractions of current operating systems.

But there are other ancestors in Go's family tree. One major stream of influence comes from languages by Niklaus Wirth, beginning with Pascal. Modula-2 inspired the package concept. Oberon eliminated the distinction between module interface files and module implementation files. Oberon-2 influenced the syntax for packages, imports, and declarations, and Object Oberon provided the syntax for method declarations.

Another lineage among Go's ancestors, and one that makes Go distinctive among recent programming languages, is a sequence of little-known research languages developed at Bell Labs, all inspired by the concept of *communicating sequential processes* (CSP) from Tony Hoare's seminal 1978 paper on the foundations of concurrency. In CSP, a program is a parallel composition of processes that have no shared state; the processes communicate and synchronize using channels. But Hoare's CSP was a formal language for describing the fundamental concepts of concurrency, not a programming language for writing executable programs.

Rob Pike and others began to experiment with CSP implementations as actual languages. The first was called Squeak ("A language for communicating with mice"), which provided a language for handling mouse and keyboard events, with statically created channels. This was followed by Newsqueak, which offered C-like statement and expression syntax and Pascal-like type notation. It was a purely functional language with garbage collection, again aimed at managing keyboard, mouse, and window events. Channels became first-class values, dynamically created and storable in variables.

The Plan 9 operating system carried these ideas forward in a language called Alef. Alef tried to make Newsqueak a viable system programming language, but its omission of garbage collection made concurrency too painful.

Other constructions in Go show the influence of non-ancestral genes here and there; for example *iota* is loosely from APL, and lexical scope with nested functions is from Scheme (and most languages since). Here too we find novel mutations. Go's innovative slices provide dynamic arrays with efficient random access but also permit sophisticated sharing arrangements reminiscent of linked lists. And the *defer* statement is new with Go.

The Go Project

All programming languages reflect the programming philosophy of their creators, which often includes a significant component of reaction to the perceived shortcomings of earlier languages. The Go project was borne of frustration with several software systems at Google that were suffering from an explosion of complexity. (This problem is by no means unique to Google.)

As Rob Pike put it, "complexity is multiplicative": fixing a problem by making one part of the system more complex slowly but surely adds complexity to other parts. With constant pressure to add features and options and configurations, and to ship code quickly, it's easy to neglect simplicity, even though in the long run simplicity is the key to good software.

Simplicity requires more work at the beginning of a project to reduce an idea to its essence and more discipline over the lifetime of a project to distinguish good changes from bad or pernicious ones. With sufficient effort, a good change can be accommodated without compromising what Fred Brooks called the “conceptual integrity” of the design but a bad change cannot, and a pernicious change trades simplicity for its shallow cousin, convenience. Only through simplicity of design can a system remain stable, secure, and coherent as it grows.

The Go project includes the language itself, its tools and standard libraries, and last but not least, a cultural agenda of radical simplicity. As a recent high-level language, Go has the benefit of hindsight, and the basics are done well: it has garbage collection, a package system, first-class functions, lexical scope, a system call interface, and immutable strings in which text is generally encoded in UTF-8. But it has comparatively few features and is unlikely to add more. For instance, it has no implicit numeric conversions, no constructors or destructors, no operator overloading, no default parameter values, no inheritance, no generics, no exceptions, no macros, no function annotations, and no thread-local storage. The language is mature and stable, and guarantees backwards compatibility: older Go programs can be compiled and run with newer versions of compilers and standard libraries.

Go has enough of a type system to avoid most of the careless mistakes that plague programmers in dynamic languages, but it has a simpler type system than comparable typed languages. This approach can sometimes lead to isolated pockets of “untyped” programming within a broader framework of types, and Go programmers do not go to the lengths that C++ or Haskell programmers do to express safety properties as type-based proofs. But in practice Go gives programmers much of the safety and run-time performance benefits of a relatively strong type system without the burden of a complex one.

Go encourages an awareness of contemporary computer system design, particularly the importance of locality. Its built-in data types and most library data structures are crafted to work naturally without explicit initialization or implicit constructors, so relatively few memory allocations and memory writes are hidden in the code. Go’s aggregate types (structs and arrays) hold their elements directly, requiring less storage and fewer allocations and pointer indirections than languages that use indirect fields. And since the modern computer is a parallel machine, Go has concurrency features based on CSP, as mentioned earlier. The variable-size stacks of Go’s lightweight threads or *goroutines* are initially small enough that creating one goroutine is cheap and creating a million is practical.

Go’s standard library, often described as coming with “batteries included,” provides clean building blocks and APIs for I/O, text processing, graphics, cryptography, networking, and distributed applications, with support for many standard file formats and protocols. The libraries and tools make extensive use of convention to reduce the need for configuration and explanation, thus simplifying program logic and making diverse Go programs more similar to each other and thus easier to learn. Projects built using the `go` tool use only file and identifier names and an occasional special comment to determine all the libraries, executables, tests, benchmarks, examples, platform-specific variants, and documentation for a project; the Go source itself contains the build specification.

Organization of the Book

We assume that you have programmed in one or more other languages, whether compiled like C, C++, and Java, or interpreted like Python, Ruby, and JavaScript, so we won't spell out everything as if for a total beginner. Surface syntax will be familiar, as will variables and constants, expressions, control flow, and functions.

Chapter 1 is a tutorial on the basic constructs of Go, introduced through a dozen programs for everyday tasks like reading and writing files, formatting text, creating images, and communicating with Internet clients and servers.

Chapter 2 describes the structural elements of a Go program—declarations, variables, new types, packages and files, and scope. Chapter 3 discusses numbers, booleans, strings, and constants, and explains how to process Unicode. Chapter 4 describes composite types, that is, types built up from simpler ones using arrays, maps, structs, and *slices*, Go's approach to dynamic lists. Chapter 5 covers functions and discusses error handling, *panic* and *recover*, and the *defer* statement.

Chapters 1 through 5 are thus the basics, things that are part of any mainstream imperative language. Go's syntax and style sometimes differ from other languages, but most programmers will pick them up quickly. The remaining chapters focus on topics where Go's approach is less conventional: methods, interfaces, concurrency, packages, testing, and reflection.

Go has an unusual approach to object-oriented programming. There are no class hierarchies, or indeed any classes; complex object behaviors are created from simpler ones by composition, not inheritance. Methods may be associated with any user-defined type, not just structures, and the relationship between concrete types and abstract types (*interfaces*) is implicit, so a concrete type may satisfy an interface that the type's designer was unaware of. Methods are covered in Chapter 6 and interfaces in Chapter 7.

Chapter 8 presents Go's approach to concurrency, which is based on the idea of communicating sequential processes (CSP), embodied by goroutines and channels. Chapter 9 explains the more traditional aspects of concurrency based on shared variables.

Chapter 10 describes packages, the mechanism for organizing libraries. This chapter also shows how to make effective use of the *go* tool, which provides for compilation, testing, benchmarking, program formatting, documentation, and many other tasks, all within a single command.

Chapter 11 deals with testing, where Go takes a notably lightweight approach, avoiding abstraction-laden frameworks in favor of simple libraries and tools. The testing libraries provide a foundation atop which more complex abstractions can be built if necessary.

Chapter 12 discusses reflection, the ability of a program to examine its own representation during execution. Reflection is a powerful tool, though one to be used carefully; this chapter explains finding the right balance by showing how it is used to implement some important Go libraries. Chapter 13 explains the gory details of low-level programming that uses the *unsafe* package to step around Go's type system, and when that is appropriate.

Each chapter has a number of exercises that you can use to test your understanding of Go, and to explore extensions and alternatives to the examples from the book.

All but the most trivial code examples in the book are available for download from the public Git repository at gopl.io. Each example is identified by its package import path and may be conveniently fetched, built, and installed using the `go get` command. You'll need to choose a directory to be your Go workspace and set the `GOPATH` environment variable to point to it. The `go` tool will create the directory if necessary. For example:

```
$ export GOPATH=$HOME/gobook          # choose workspace directory
$ go get gopl.io/ch1/helloworld        # fetch, build, install
$ $GOPATH/bin/helloworld               # run
Hello, 世界
```

To run the examples, you will need at least version 1.5 of Go.

```
$ go version
go version go1.5 linux/amd64
```

Follow the instructions at <https://golang.org/doc/install> if the `go` tool on your computer is older or missing.

Where to Find More Information

The best source for more information about Go is the official web site, <https://golang.org>, which provides access to the documentation, including the *Go Programming Language Specification*, standard packages, and the like. There are also tutorials on how to write Go and how to write it well, and a wide variety of online text and video resources that will be valuable complements to this book. The Go Blog at blog.golang.org publishes some of the best writing on Go, with articles on the state of the language, plans for the future, reports on conferences, and in-depth explanations of a wide variety of Go-related topics.

One of the most useful aspects of online access to Go (and a regrettable limitation of a paper book) is the ability to run Go programs from the web pages that describe them. This functionality is provided by the Go Playground at play.golang.org, and may be embedded within other pages, such as the home page at golang.org or the documentation pages served by the `godoc` tool.

The Playground makes it convenient to perform simple experiments to check one's understanding of syntax, semantics, or library packages with short programs, and in many ways takes the place of a *read-eval-print loop* (REPL) in other languages. Its persistent URLs are great for sharing snippets of Go code with others, for reporting bugs or making suggestions.

Built atop the Playground, the Go Tour at tour.golang.org is a sequence of short interactive lessons on the basic ideas and constructions of Go, an orderly walk through the language.

The primary shortcoming of the Playground and the Tour is that they allow only standard libraries to be imported, and many library features—networking, for example—are restricted

for practical or security reasons. They also require access to the Internet to compile and run each program. So for more elaborate experiments, you will have to run Go programs on your own computer. Fortunately the download process is straightforward, so it should not take more than a few minutes to fetch the Go distribution from golang.org and start writing and running Go programs of your own.

Since Go is an open-source project, you can read the code for any type or function in the standard library online at <https://golang.org/pkg>; the same code is part of the downloaded distribution. Use this to figure out how something works, or to answer questions about details, or merely to see how experts write really good Go.

Acknowledgments

Rob Pike and Russ Cox, core members of the Go team, read the manuscript several times with great care; their comments on everything from word choice to overall structure and organization have been invaluable. While preparing the Japanese translation, Yoshiki Shibata went far beyond the call of duty; his meticulous eye spotted numerous inconsistencies in the English text and errors in the code. We greatly appreciate thorough reviews and critical comments on the entire manuscript from Brian Goetz, Corey Kosak, Arnold Robbins, Josh Bleecher Snyder, and Peter Weinberger.

We are indebted to Sameer Ajmani, Ittai Balaban, David Crawshaw, Billy Donohue, Jonathan Feinberg, Andrew Gerrand, Robert Griesemer, John Linderman, Minux Ma, Bryan Mills, Bala Natarajan, Cosmos Nicolaou, Paul Staniforth, Nigel Tao, and Howard Trickey for many helpful suggestions. We also thank David Brailsford and Raph Levien for typesetting advice.

Our editor Greg Doench at Addison-Wesley got the ball rolling originally and has been continuously helpful ever since. The AW production team—John Fuller, Dayna Isley, Julie Nahil, Chuti Prasertsith, and Barbara Wood—has been outstanding; authors could not hope for better support.

Alan Donovan wishes to thank: Sameer Ajmani, Chris Demetriou, Walt Drummond, and Reid Tatge at Google for allowing him time to write; Stephen Donovan, for his advice and timely encouragement; and above all, his wife Leila Kazemi, for her unhesitating enthusiasm and unwavering support for this project, despite the long hours of distraction and absenteeism from family life that it entailed.

Brian Kernighan is deeply grateful to friends and colleagues for their patience and forbearance as he moved slowly along the path to understanding, and especially to his wife Meg, who has been unfailingly supportive of book-writing and so much else.

New York
October 2015

This page intentionally left blank

1

Tutorial

This chapter is a tour of the basic components of Go. We hope to provide enough information and examples to get you off the ground and doing useful things as quickly as possible. The examples here, and indeed in the whole book, are aimed at tasks that you might have to do in the real world. In this chapter we'll try to give you a taste of the diversity of programs that one might write in Go, ranging from simple file processing and a bit of graphics to concurrent Internet clients and servers. We certainly won't explain everything in the first chapter, but studying such programs in a new language can be an effective way to get started.

When you're learning a new language, there's a natural tendency to write code as you would have written it in a language you already know. Be aware of this bias as you learn Go and try to avoid it. We've tried to illustrate and explain how to write good Go, so use the code here as a guide when you're writing your own.

1.1. Hello, World

We'll start with the now-traditional “hello, world” example, which appears at the beginning of *The C Programming Language*, published in 1978. C is one of the most direct influences on Go, and “hello, world” illustrates a number of central ideas.

gopl.io/ch1/helloworld

```
package main

import "fmt"

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

Go is a compiled language. The Go toolchain converts a source program and the things it depends on into instructions in the native machine language of a computer. These tools are accessed through a single command called `go` that has a number of subcommands. The simplest of these subcommands is `run`, which compiles the source code from one or more source files whose names end in `.go`, links it with libraries, then runs the resulting executable file. (We will use `$` as the command prompt throughout the book.)

```
$ go run helloworld.go
```

Not surprisingly, this prints

```
Hello, 世界
```

Go natively handles Unicode, so it can process text in all the world's languages.

If the program is more than a one-shot experiment, it's likely that you would want to compile it once and save the compiled result for later use. That is done with `go build`:

```
$ go build helloworld.go
```

This creates an executable binary file called `helloworld` that can be run any time without further processing:

```
$ ./helloworld
Hello, 世界
```

We have labeled each significant example as a reminder that you can obtain the code from the book's source code repository at `gopl.io`:

gopl.io/ch1/helloworld

If you run `go get gopl.io/ch1/helloworld`, it will fetch the source code and place it in the corresponding directory. There's more about this topic in Section 2.6 and Section 10.7.

Let's now talk about the program itself. Go code is organized into packages, which are similar to libraries or modules in other languages. A package consists of one or more `.go` source files in a single directory that define what the package does. Each source file begins with a package declaration, here `package main`, that states which package the file belongs to, followed by a list of other packages that it imports, and then the declarations of the program that are stored in that file.

The Go standard library has over 100 packages for common tasks like input and output, sorting, and text manipulation. For instance, the `fmt` package contains functions for printing formatted output and scanning input. `Println` is one of the basic output functions in `fmt`; it prints one or more values, separated by spaces, with a newline character at the end so that the values appear as a single line of output.

Package `main` is special. It defines a standalone executable program, not a library. Within package `main` the *function* `main` is also special—it's where execution of the program begins. Whatever `main` does is what the program does. Of course, `main` will normally call upon functions in other packages to do much of the work, such as the function `fmt.Println`.

We must tell the compiler what packages are needed by this source file; that's the role of the `import` declaration that follows the package declaration. The “hello, world” program uses only one function from one other package, but most programs will import more packages.

You must import exactly the packages you need. A program will not compile if there are missing imports or if there are unnecessary ones. This strict requirement prevents references to unused packages from accumulating as programs evolve.

The `import` declarations must follow the package declaration. After that, a program consists of the declarations of functions, variables, constants, and types (introduced by the keywords `func`, `var`, `const`, and `type`); for the most part, the order of declarations does not matter. This program is about as short as possible since it declares only one function, which in turn calls only one other function. To save space, we will sometimes not show the package and `import` declarations when presenting examples, but they are in the source file and must be there to compile the code.

A function declaration consists of the keyword `func`, the name of the function, a parameter list (empty for `main`), a result list (also empty here), and the body of the function—the statements that define what it does—enclosed in braces. We'll take a closer look at functions in Chapter 5.

Go does not require semicolons at the ends of statements or declarations, except where two or more appear on the same line. In effect, newlines following certain tokens are converted into semicolons, so where newlines are placed matters to proper parsing of Go code. For instance, the opening brace `{` of the function must be on the same line as the end of the `func` declaration, not on a line by itself, and in the expression `x + y`, a newline is permitted after but not before the `+` operator.

Go takes a strong stance on code formatting. The `gofmt` tool rewrites code into the standard format, and the `go` tool's `fmt` subcommand applies `gofmt` to all the files in the specified package, or the ones in the current directory by default. All Go source files in the book have been run through `gofmt`, and you should get into the habit of doing the same for your own code. Declaring a standard format by fiat eliminates a lot of pointless debate about trivia and, more importantly, enables a variety of automated source code transformations that would be infeasible if arbitrary formatting were allowed.

Many text editors can be configured to run `gofmt` each time you save a file, so that your source code is always properly formatted. A related tool, `goimports`, additionally manages the insertion and removal of import declarations as needed. It is not part of the standard distribution but you can obtain it with this command:

```
$ go get golang.org/x/tools/cmd/goimports
```

For most users, the usual way to download and build packages, run their tests, show their documentation, and so on, is with the `go` tool, which we'll look at in Section 10.7.

1.2. Command-Line Arguments

Most programs process some input to produce some output; that's pretty much the definition of computing. But how does a program get input data on which to operate? Some programs generate their own data, but more often, input comes from an external source: a file, a network connection, the output of another program, a user at a keyboard, command-line arguments, or the like. The next few examples will discuss some of these alternatives, starting with command-line arguments.

The `os` package provides functions and other values for dealing with the operating system in a platform-independent fashion. Command-line arguments are available to a program in a variable named `Args` that is part of the `os` package; thus its name anywhere outside the `os` package is `os.Args`.

The variable `os.Args` is a *slice* of strings. Slices are a fundamental notion in Go, and we'll talk a lot more about them soon. For now, think of a slice as a dynamically sized sequence `s` of array elements where individual elements can be accessed as `s[i]` and a contiguous subsequence as `s[m:n]`. The number of elements is given by `len(s)`. As in most other programming languages, all indexing in Go uses *half-open* intervals that include the first index but exclude the last, because it simplifies logic. For example, the slice `s[m:n]`, where $0 \leq m \leq n \leq \text{len}(s)$, contains $n-m$ elements.

The first element of `os.Args`, `os.Args[0]`, is the name of the command itself; the other elements are the arguments that were presented to the program when it started execution. A slice expression of the form `s[m:n]` yields a slice that refers to elements `m` through `n-1`, so the elements we need for our next example are those in the slice `os.Args[1:len(os.Args)]`. If `m` or `n` is omitted, it defaults to 0 or `len(s)` respectively, so we can abbreviate the desired slice as `os.Args[1:]`.

Here's an implementation of the Unix `echo` command, which prints its command-line arguments on a single line. It imports two packages, which are given as a parenthesized list rather than as individual `import` declarations. Either form is legal, but conventionally the list form is used. The order of imports doesn't matter; the `gofmt` tool sorts the package names into alphabetical order. (When there are several versions of an example, we will often number them so you can be sure of which one we're talking about.)

```
gopl.io/ch1/echo1
// Echo1 prints its command-line arguments.
package main

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

```
func main() {
    var s, sep string
    for i := 1; i < len(os.Args); i++ {
        s += sep + os.Args[i]
        sep = " "
    }
    fmt.Println(s)
}
```

Comments begin with `//`. All text from a `//` to the end of the line is commentary for programmers and is ignored by the compiler. By convention, we describe each package in a comment immediately preceding its package declaration; for a `main` package, this comment is one or more complete sentences that describe the program as a whole.

The `var` declaration declares two variables `s` and `sep`, of type `string`. A variable can be initialized as part of its declaration. If it is not explicitly initialized, it is implicitly initialized to the *zero value* for its type, which is `0` for numeric types and the empty string `""` for strings. Thus in this example, the declaration implicitly initializes `s` and `sep` to empty strings. We'll have more to say about variables and declarations in Chapter 2.

For numbers, Go provides the usual arithmetic and logical operators. When applied to strings, however, the `+` operator *concatenates* the values, so the expression

```
sep + os.Args[i]
```

represents the concatenation of the strings `sep` and `os.Args[i]`. The statement we used in the program,

```
s += sep + os.Args[i]
```

is an *assignment statement* that concatenates the old value of `s` with `sep` and `os.Args[i]` and assigns it back to `s`; it is equivalent to

```
s = s + sep + os.Args[i]
```

The operator `+=` is an *assignment operator*. Each arithmetic and logical operator like `+` or `*` has a corresponding assignment operator.

The `echo` program could have printed its output in a loop one piece at a time, but this version instead builds up a string by repeatedly appending new text to the end. The string `s` starts life empty, that is, with value `""`, and each trip through the loop adds some text to it; after the first iteration, a space is also inserted so that when the loop is finished, there is one space between each argument. This is a quadratic process that could be costly if the number of arguments is large, but for `echo`, that's unlikely. We'll show a number of improved versions of `echo` in this chapter and the next that will deal with any real inefficiency.

The loop index variable `i` is declared in the first part of the `for` loop. The `:=` symbol is part of a *short variable declaration*, a statement that declares one or more variables and gives them appropriate types based on the initializer values; there's more about this in the next chapter.

The increment statement `i++` adds 1 to `i`; it's equivalent to `i += 1` which is in turn equivalent to `i = i + 1`. There's a corresponding decrement statement `i--` that subtracts 1. These are

statements, not expressions as they are in most languages in the C family, so `j = i++` is illegal, and they are postfix only, so `--i` is not legal either.

The `for` loop is the only loop statement in Go. It has a number of forms, one of which is illustrated here:

```
for initialization; condition; post {
    // zero or more statements
}
```

Parentheses are never used around the three components of a `for` loop. The braces are mandatory, however, and the opening brace must be on the same line as the *post* statement.

The optional *initialization* statement is executed before the loop starts. If it is present, it must be a *simple statement*, that is, a short variable declaration, an increment or assignment statement, or a function call. The *condition* is a boolean expression that is evaluated at the beginning of each iteration of the loop; if it evaluates to true, the statements controlled by the loop are executed. The *post* statement is executed after the body of the loop, then the condition is evaluated again. The loop ends when the condition becomes false.

Any of these parts may be omitted. If there is no *initialization* and no *post*, the semicolons may also be omitted:

```
// a traditional "while" loop
for condition {
    // ...
}
```

If the condition is omitted entirely in any of these forms, for example in

```
// a traditional infinite loop
for {
    // ...
}
```

the loop is infinite, though loops of this form may be terminated in some other way, like a `break` or `return` statement.

Another form of the `for` loop iterates over a *range* of values from a data type like a string or a slice. To illustrate, here's a second version of `echo`:

```
gopl.io/ch1/echo2
// Echo2 prints its command-line arguments.
package main

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

```
func main() {
    s, sep := "", ""
    for _, arg := range os.Args[1:] {
        s += sep + arg
        sep = " "
    }
    fmt.Println(s)
}
```

In each iteration of the loop, `range` produces a pair of values: the index and the value of the element at that index. In this example, we don't need the index, but the syntax of a `range` loop requires that if we deal with the element, we must deal with the index too. One idea would be to assign the index to an obviously temporary variable like `temp` and ignore its value, but Go does not permit unused local variables, so this would result in a compilation error.

The solution is to use the *blank identifier*, whose name is `_` (that is, an underscore). The blank identifier may be used whenever syntax requires a variable name but program logic does not, for instance to discard an unwanted loop index when we require only the element value. Most Go programmers would likely use `range` and `_` to write the `echo` program as above, since the indexing over `os.Args` is implicit, not explicit, and thus easier to get right.

This version of the program uses a short variable declaration to declare and initialize `s` and `sep`, but we could equally well have declared the variables separately. There are several ways to declare a string variable; these are all equivalent:

```
s := ""
var s string
var s = ""
var s string = ""
```

Why should you prefer one form to another? The first form, a short variable declaration, is the most compact, but it may be used only within a function, not for package-level variables. The second form relies on default initialization to the zero value for strings, which is `""`. The third form is rarely used except when declaring multiple variables. The fourth form is explicit about the variable's type, which is redundant when it is the same as that of the initial value but necessary in other cases where they are not of the same type. In practice, you should generally use one of the first two forms, with explicit initialization to say that the initial value is important and implicit initialization to say that the initial value doesn't matter.

As noted above, each time around the loop, the string `s` gets completely new contents. The `+=` statement makes a new string by concatenating the old string, a space character, and the next argument, then assigns the new string to `s`. The old contents of `s` are no longer in use, so they will be garbage-collected in due course.

If the amount of data involved is large, this could be costly. A simpler and more efficient solution would be to use the `Join` function from the `strings` package:

gopl.io/ch1/echo3

```
func main() {
    fmt.Println(strings.Join(os.Args[1:], " "))
}
```

Finally, if we don't care about format but just want to see the values, perhaps for debugging, we can let `Println` format the results for us:

```
fmt.Println(os.Args[1:])
```

The output of this statement is like what we would get from `strings.Join`, but with surrounding brackets. Any slice may be printed this way.

Exercise 1.1: Modify the echo program to also print `os.Args[0]`, the name of the command that invoked it.

Exercise 1.2: Modify the echo program to print the index and value of each of its arguments, one per line.

Exercise 1.3: Experiment to measure the difference in running time between our potentially inefficient versions and the one that uses `strings.Join`. (Section 1.6 illustrates part of the `time` package, and Section 11.4 shows how to write benchmark tests for systematic performance evaluation.)

1.3. Finding Duplicate Lines

Programs for file copying, printing, searching, sorting, counting, and the like all have a similar structure: a loop over the input, some computation on each element, and generation of output on the fly or at the end. We'll show three variants of a program called `dup`; it is partly inspired by the Unix `uniq` command, which looks for adjacent duplicate lines. The structures and packages used are models that can be easily adapted.

The first version of `dup` prints each line that appears more than once in the standard input, preceded by its count. This program introduces the `if` statement, the `map` data type, and the `bufio` package.

gopl.io/ch1/dup1

```
// Dup1 prints the text of each line that appears more than
// once in the standard input, preceded by its count.
package main

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

```

func main() {
    counts := make(map[string]int)
    input := bufio.NewScanner(os.Stdin)
    for input.Scan() {
        counts[input.Text()]++
    }
    // NOTE: ignoring potential errors from input.Err()
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}

```

As with `for`, parentheses are never used around the condition in an `if` statement, but braces are required for the body. There can be an optional `else` part that is executed if the condition is false.

A *map* holds a set of key/value pairs and provides constant-time operations to store, retrieve, or test for an item in the set. The key may be of any type whose values can be compared with `==`, strings being the most common example; the value may be of any type at all. In this example, the keys are strings and the values are ints. The built-in function `make` creates a new empty map; it has other uses too. Maps are discussed at length in Section 4.3.

Each time `dup` reads a line of input, the line is used as a key into the map and the corresponding value is incremented. The statement `counts[input.Text()]++` is equivalent to these two statements:

```

line := input.Text()
counts[line] = counts[line] + 1

```

It's not a problem if the map doesn't yet contain that key. The first time a new line is seen, the expression `counts[line]` on the right-hand side evaluates to the zero value for its type, which is `0` for `int`.

To print the results, we use another range-based `for` loop, this time over the `counts` map. As before, each iteration produces two results, a key and the value of the map element for that key. The order of map iteration is not specified, but in practice it is random, varying from one run to another. This design is intentional, since it prevents programs from relying on any particular ordering where none is guaranteed.

Onward to the `bufio` package, which helps make input and output efficient and convenient. One of its most useful features is a type called `Scanner` that reads input and breaks it into lines or words; it's often the easiest way to process input that comes naturally in lines.

The program uses a short variable declaration to create a new variable `input` that refers to a `bufio.Scanner`:

```

input := bufio.NewScanner(os.Stdin)

```

The scanner reads from the program's standard input. Each call to `input.Scan()` reads the next line and removes the newline character from the end; the result can be retrieved by calling `input.Text()`. The `Scan` function returns `true` if there is a line and `false` when there is no more input.

The function `fmt.Printf`, like `printf` in C and other languages, produces formatted output from a list of expressions. Its first argument is a format string that specifies how subsequent arguments should be formatted. The format of each argument is determined by a conversion character, a letter following a percent sign. For example, `%d` formats an integer operand using decimal notation, and `%s` expands to the value of a string operand.

`Printf` has over a dozen such conversions, which Go programmers call *verbs*. This table is far from a complete specification but illustrates many of the features that are available:

<code>%d</code>	decimal integer
<code>%x</code> , <code>%O</code> , <code>%b</code>	integer in hexadecimal, octal, binary
<code>%f</code> , <code>%g</code> , <code>%e</code>	floating-point number: 3.141593 3.141592653589793 3.141593e+00
<code>%t</code>	boolean: <code>true</code> or <code>false</code>
<code>%c</code>	rune (Unicode code point)
<code>%s</code>	string
<code>%q</code>	quoted string <code>"abc"</code> or rune <code>'c'</code>
<code>%v</code>	any value in a natural format
<code>%T</code>	type of any value
<code>%%</code>	literal percent sign (no operand)

The format string in `dup1` also contains a tab `\t` and a newline `\n`. String literals may contain such *escape sequences* for representing otherwise invisible characters. `Printf` does not write a newline by default. By convention, formatting functions whose names end in `f`, such as `log.Printf` and `fmt.Errorf`, use the formatting rules of `fmt.Printf`, whereas those whose names end in `ln` follow `Println`, formatting their arguments as if by `%v`, followed by a newline.

Many programs read either from their standard input, as above, or from a sequence of named files. The next version of `dup` can read from the standard input or handle a list of file names, using `os.Open` to open each one:

gopl.io/ch1/dup2

```
// Dup2 prints the count and text of lines that appear more than once
// in the input. It reads from stdin or from a list of named files.
package main

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

```

func main() {
    counts := make(map[string]int)
    files := os.Args[1:]
    if len(files) == 0 {
        countLines(os.Stdin, counts)
    } else {
        for _, arg := range files {
            f, err := os.Open(arg)
            if err != nil {
                fmt.Fprintf(os.Stderr, "dup2: %v\n", err)
                continue
            }
            countLines(f, counts)
            f.Close()
        }
    }
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}

func countLines(f *os.File, counts map[string]int) {
    input := bufio.NewScanner(f)
    for input.Scan() {
        counts[input.Text()]++
    }
    // NOTE: ignoring potential errors from input.Err()
}

```

The function `os.Open` returns two values. The first is an open file (`*os.File`) that is used in subsequent reads by the `Scanner`.

The second result of `os.Open` is a value of the built-in error type. If `err` equals the special built-in value `nil`, the file was opened successfully. The file is read, and when the end of the input is reached, `Close` closes the file and releases any resources. On the other hand, if `err` is not `nil`, something went wrong. In that case, the error value describes the problem. Our simple-minded error handling prints a message on the standard error stream using `Fprintf` and the verb `%v`, which displays a value of any type in a default format, and `dup` then carries on with the next file; the `continue` statement goes to the next iteration of the enclosing `for` loop.

In the interests of keeping code samples to a reasonable size, our early examples are intentionally somewhat cavalier about error handling. Clearly we must check for an error from `os.Open`; however, we are ignoring the less likely possibility that an error could occur while reading the file with `input.Scan`. We will note places where we've skipped error checking, and we will go into the details of error handling in Section 5.4.

Notice that the call to `countLines` precedes its declaration. Functions and other package-level entities may be declared in any order.

A map is a *reference* to the data structure created by `make`. When a map is passed to a function, the function receives a copy of the reference, so any changes the called function makes to the underlying data structure will be visible through the caller's map reference too. In our example, the values inserted into the counts map by `countLines` are seen by `main`.

The versions of `dup` above operate in a “streaming” mode in which input is read and broken into lines as needed, so in principle these programs can handle an arbitrary amount of input. An alternative approach is to read the entire input into memory in one big gulp, split it into lines all at once, then process the lines. The following version, `dup3`, operates in that fashion. It introduces the function `ReadFile` (from the `io/ioutil` package), which reads the entire contents of a named file, and `strings.Split`, which splits a string into a slice of substrings. (`Split` is the opposite of `strings.Join`, which we saw earlier.)

We've simplified `dup3` somewhat. First, it only reads named files, not the standard input, since `ReadFile` requires a file name argument. Second, we moved the counting of the lines back into `main`, since it is now needed in only one place.

`gopl.io/ch1/dup3`

```
package main

import (
    "fmt"
    "io/ioutil"
    "os"
    "strings"
)

func main() {
    counts := make(map[string]int)
    for _, filename := range os.Args[1:] {
        data, err := ioutil.ReadFile(filename)
        if err != nil {
            fmt.Fprintf(os.Stderr, "dup3: %v\n", err)
            continue
        }
        for _, line := range strings.Split(string(data), "\n") {
            counts[line]++
        }
    }
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}
```

`ReadFile` returns a byte slice that must be converted into a string so it can be split by `strings.Split`. We will discuss strings and byte slices at length in Section 3.5.4.

Under the covers, `bufio.Scanner`, `ioutil.ReadFile`, and `ioutil.WriteFile` use the `Read` and `Write` methods of `*os.File`, but it's rare that most programmers need to access those lower-level routines directly. The higher-level functions like those from `bufio` and `io/ioutil` are easier to use.

Exercise 1.4: Modify `dup2` to print the names of all files in which each duplicated line occurs.

1.4. Animated GIFs

The next program demonstrates basic usage of Go's standard image packages, which we'll use to create a sequence of bit-mapped images and then encode the sequence as a GIF animation. The images, called *Lissajous figures*, were a staple visual effect in sci-fi films of the 1960s. They are the parametric curves produced by harmonic oscillation in two dimensions, such as two sine waves fed into the *x* and *y* inputs of an oscilloscope. Figure 1.1 shows some examples.

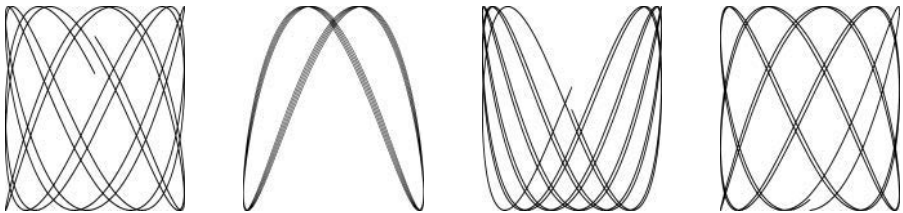


Figure 1.1. Four Lissajous figures.

There are several new constructs in this code, including `const` declarations, struct types, and composite literals. Unlike most of our examples, this one also involves floating-point computations. We'll discuss these topics only briefly here, pushing most details off to later chapters, since the primary goal right now is to give you an idea of what Go looks like and the kinds of things that can be done easily with the language and its libraries.

gopl.io/ch1/lissajous

```
// Lissajous generates GIF animations of random Lissajous figures.
package main

import (
    "image"
    "image/color"
    "image/gif"
    "io"
    "math"
    "math/rand"
    "os"
)
```



```

var palette = []color.Color{color.White, color.Black}
const (
    whiteIndex = 0 // first color in palette
    blackIndex = 1 // next color in palette
)
func main() {
    lissajous(os.Stdout)
}
func lissajous(out io.Writer) {
    const (
        cycles = 5      // number of complete x oscillator revolutions
        res     = 0.001  // angular resolution
        size    = 100    // image canvas covers [-size..+size]
        nframes = 64     // number of animation frames
        delay   = 8      // delay between frames in 10ms units
    )
    freq := rand.Float64() * 3.0 // relative frequency of y oscillator
    anim := gif.GIF{LoopCount: nframes}
    phase := 0.0 // phase difference
    for i := 0; i < nframes; i++ {
        rect := image.Rect(0, 0, 2*size+1, 2*size+1)
        img := image.NewPaletted(rect, palette)
        for t := 0.0; t < cycles*2*math.Pi; t += res {
            x := math.Sin(t)
            y := math.Sin(t*freq + phase)
            img.SetColorIndex(size+int(x*size+0.5), size+int(y*size+0.5),
                             blackIndex)
        }
        phase += 0.1
        anim.Delay = append(anim.Delay, delay)
        anim.Image = append(anim.Image, img)
    }
    gif.EncodeAll(out, &anim) // NOTE: ignoring encoding errors
}

```

After importing a package whose path has multiple components, like `image/color`, we refer to the package with a name that comes from the last component. Thus the variable `color.White` belongs to the `image/color` package and `gif.GIF` belongs to `image/gif`.

A `const` declaration (§3.6) gives names to constants, that is, values that are fixed at compile time, such as the numerical parameters for `cycles`, `frames`, and `delay`. Like `var` declarations, `const` declarations may appear at package level (so the names are visible throughout the package) or within a function (so the names are visible only within that function). The value of a constant must be a number, string, or boolean.

The expressions `[]color.Color{...}` and `gif.GIF{...}` are *composite literals* (§4.2, §4.4.1), a compact notation for instantiating any of Go's composite types from a sequence of element values. Here, the first one is a slice and the second one is a *struct*.

The type `gif.GIF` is a struct type (§4.4). A struct is a group of values called *fields*, often of different types, that are collected together in a single object that can be treated as a unit. The variable `anim` is a struct of type `gif.GIF`. The struct literal creates a struct value whose `LoopCount` field is set to `nframes`; all other fields have the zero value for their type. The individual fields of a struct can be accessed using dot notation, as in the final two assignments which explicitly update the `Delay` and `Image` fields of `anim`.

The `lissajous` function has two nested loops. The outer loop runs for 64 iterations, each producing a single frame of the animation. It creates a new 201×201 image with a palette of two colors, white and black. All pixels are initially set to the palette's zero value (the zeroth color in the palette), which we set to white. Each pass through the inner loop generates a new image by setting some pixels to black. The result is appended, using the built-in `append` function (§4.2.1), to a list of frames in `anim`, along with a specified delay of 80ms. Finally the sequence of frames and delays is encoded into GIF format and written to the output stream `out`. The type of `out` is `io.Writer`, which lets us write to a wide range of possible destinations, as we'll show soon.

The inner loop runs the two oscillators. The x oscillator is just the sine function. The y oscillator is also a sinusoid, but its frequency relative to the x oscillator is a random number between 0 and 3, and its phase relative to the x oscillator is initially zero but increases with each frame of the animation. The loop runs until the x oscillator has completed five full cycles. At each step, it calls `SetColorIndex` to color the pixel corresponding to (x, y) black, which is at position 1 in the palette.

The `main` function calls the `lissajous` function, directing it to write to the standard output, so this command produces an animated GIF with frames like those in Figure 1.1:

```
$ go build gopl.io/ch1/lissajous
$ ./lissajous >out.gif
```

Exercise 1.5: Change the `Lissajous` program's color palette to green on black, for added authenticity. To create the web color `#RRGGBB`, use `color.RGBA{0xRR, 0xGG, 0xBB, 0xff}`, where each pair of hexadecimal digits represents the intensity of the red, green, or blue component of the pixel.

Exercise 1.6: Modify the `Lissajous` program to produce images in multiple colors by adding more values to `palette` and then displaying them by changing the third argument of `SetColorIndex` in some interesting way.

1.5. Fetching a URL

For many applications, access to information from the Internet is as important as access to the local file system. Go provides a collection of packages, grouped under `net`, that make it easy to send and receive information through the Internet, make low-level network connections, and set up servers, for which Go's concurrency features (introduced in Chapter 8) are particularly useful.

To illustrate the minimum necessary to retrieve information over HTTP, here's a simple program called `fetch` that fetches the content of each specified URL and prints it as uninterpreted text; it's inspired by the invaluable utility `curl`. Obviously one would usually do more with such data, but this shows the basic idea. We will use this program frequently in the book.

gopl.io/ch1/fetch

```
// Fetch prints the content found at a URL.
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "os"
)

func main() {
    for _, url := range os.Args[1:] {
        resp, err := http.Get(url)
        if err != nil {
            fmt.Fprintf(os.Stderr, "fetch: %v\n", err)
            os.Exit(1)
        }
        b, err := ioutil.ReadAll(resp.Body)
        resp.Body.Close()
        if err != nil {
            fmt.Fprintf(os.Stderr, "fetch: reading %s: %v\n", url, err)
            os.Exit(1)
        }
        fmt.Printf("%s", b)
    }
}
```

This program introduces functions from two packages, `net/http` and `io/ioutil`. The `http.Get` function makes an HTTP request and, if there is no error, returns the result in the response struct `resp`. The `Body` field of `resp` contains the server response as a readable stream. Next, `ioutil.ReadAll` reads the entire response; the result is stored in `b`. The `Body` stream is closed to avoid leaking resources, and `Printf` writes the response to the standard output.

```
$ go build gopl.io/ch1/fetch
$ ./fetch http://gopl.io
<html>
<head>
<title>The Go Programming Language</title>
...
```

If the HTTP request fails, `fetch` reports the failure instead:

```
$ ./fetch http://bad.gopl.io
fetch: Get http://bad.gopl.io: dial tcp: lookup bad.gopl.io: no such host
```

In either error case, `os.Exit(1)` causes the process to exit with a status code of 1.

Exercise 1.7: The function call `io.Copy(dst, src)` reads from `src` and writes to `dst`. Use it instead of `ioutil.ReadAll` to copy the response body to `os.Stdout` without requiring a buffer large enough to hold the entire stream. Be sure to check the error result of `io.Copy`.

Exercise 1.8: Modify `fetch` to add the prefix `http://` to each argument URL if it is missing. You might want to use `strings.HasPrefix`.

Exercise 1.9: Modify `fetch` to also print the HTTP status code, found in `resp.Status`.

1.6. Fetching URLs Concurrently

One of the most interesting and novel aspects of Go is its support for concurrent programming. This is a large topic, to which Chapter 8 and Chapter 9 are devoted, so for now we'll give you just a taste of Go's main concurrency mechanisms, goroutines and channels.

The next program, `fetchall`, does the same fetch of a URL's contents as the previous example, but it fetches many URLs, all concurrently, so that the process will take no longer than the longest fetch rather than the sum of all the fetch times. This version of `fetchall` discards the responses but reports the size and elapsed time for each one:

```
gopl.io/ch1/fetchall
// Fetchall fetches URLs in parallel and reports their times and sizes.
package main

import (
    "fmt"
    "io"
    "io/ioutil"
    "net/http"
    "os"
    "time"
)

func main() {
    start := time.Now()
    ch := make(chan string)
    for _, url := range os.Args[1:] {
        go fetch(url, ch) // start a goroutine
    }
    for range os.Args[1:] {
        fmt.Println(<-ch) // receive from channel ch
    }
    fmt.Printf("%.2fs elapsed\n", time.Since(start).Seconds())
}
```

```

func fetch(url string, ch chan<- string) {
    start := time.Now()
    resp, err := http.Get(url)
    if err != nil {
        ch <- fmt.Sprintf(err) // send to channel ch
        return
    }

    nbytes, err := io.Copy(ioutil.Discard, resp.Body)
    resp.Body.Close() // don't leak resources
    if err != nil {
        ch <- fmt.Sprintf("while reading %s: %v", url, err)
        return
    }
    secs := time.Since(start).Seconds()
    ch <- fmt.Sprintf("%.2fs  %7d  %s", secs, nbytes, url)
}

```

Here's an example:

```

$ go build gopl.io/ch1/fetchall
$ ./fetchall https://golang.org http://gopl.io https://godoc.org
0.14s      6852  https://godoc.org
0.16s      7261  https://golang.org
0.48s      2475  http://gopl.io
0.48s elapsed

```

A *goroutine* is a concurrent function execution. A *channel* is a communication mechanism that allows one goroutine to pass values of a specified type to another goroutine. The function `main` runs in a goroutine and the `go` statement creates additional goroutines.

The `main` function creates a channel of strings using `make`. For each command-line argument, the `go` statement in the first range loop starts a new goroutine that calls `fetch` asynchronously to fetch the URL using `http.Get`. The `io.Copy` function reads the body of the response and discards it by writing to the `ioutil.Discard` output stream. `Copy` returns the byte count, along with any error that occurred. As each result arrives, `fetch` sends a summary line on the channel `ch`. The second range loop in `main` receives and prints those lines.

When one goroutine attempts a send or receive on a channel, it blocks until another goroutine attempts the corresponding receive or send operation, at which point the value is transferred and both goroutines proceed. In this example, each `fetch` sends a value (`ch <- expression`) on the channel `ch`, and `main` receives all of them (`<-ch`). Having `main` do all the printing ensures that output from each goroutine is processed as a unit, with no danger of interleaving if two goroutines finish at the same time.

Exercise 1.10: Find a web site that produces a large amount of data. Investigate caching by running `fetchall` twice in succession to see whether the reported time changes much. Do you get the same content each time? Modify `fetchall` to print its output to a file so it can be examined.

Exercise 1.11: Try `fetchall` with longer argument lists, such as samples from the top million web sites available at `alexa.com`. How does the program behave if a web site just doesn't respond? (Section 8.9 describes mechanisms for coping in such cases.)

1.7. A Web Server

Go's libraries makes it easy to write a web server that responds to client requests like those made by `fetch`. In this section, we'll show a minimal server that returns the path component of the URL used to access the server. That is, if the request is for `http://localhost:8000/hello`, the response will be `URL.Path = "/hello"`.

```
gopl.io/ch1/server1
// Server1 is a minimal "echo" server.
package main

import (
    "fmt"
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/", handler) // each request calls handler
    log.Fatal(http.ListenAndServe("localhost:8000", nil))
}

// handler echoes the Path component of the request URL r.
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "URL.Path = %q\n", r.URL.Path)
}
```

The program is only a handful of lines long because library functions do most of the work. The `main` function connects a handler function to incoming URLs that begin with `/`, which is all URLs, and starts a server listening for incoming requests on port 8000. A request is represented as a struct of type `http.Request`, which contains a number of related fields, one of which is the URL of the incoming request. When a request arrives, it is given to the handler function, which extracts the path component (`/hello`) from the request URL and sends it back as the response, using `fmt.Fprintf`. Web servers will be explained in detail in Section 7.7.

Let's start the server in the background. On Mac OS X or Linux, add an ampersand (`&`) to the command; on Microsoft Windows, you will need to run the command without the ampersand in a separate command window.

```
$ go run src/gopl.io/ch1/server1/main.go &
```

We can then make client requests from the command line:

```
$ go build gopl.io/ch1/fetch
$ ./fetch http://localhost:8000
URL.Path = "/"
$ ./fetch http://localhost:8000/help
URL.Path = "/help"
```

Alternatively, we can access the server from a web browser, as shown in Figure 1.2.



Figure 1.2. A response from the echo server.

It's easy to add features to the server. One useful addition is a specific URL that returns a status of some sort. For example, this version does the same echo but also counts the number of requests; a request to the URL `/count` returns the count so far, excluding `/count` requests themselves:

```
gopl.io/ch1/server2
// Server2 is a minimal "echo" and counter server.
package main

import (
    "fmt"
    "log"
    "net/http"
    "sync"
)

var mu sync.Mutex
var count int

func main() {
    http.HandleFunc("/", handler)
    http.HandleFunc("/count", counter)
    log.Fatal(http.ListenAndServe("localhost:8000", nil))
}

// handler echoes the Path component of the requested URL.
func handler(w http.ResponseWriter, r *http.Request) {
    mu.Lock()
    count++
    mu.Unlock()
    fmt.Fprintf(w, "URL.Path = %q\n", r.URL.Path)
}
```

```
// counter echoes the number of calls so far.
func counter(w http.ResponseWriter, r *http.Request) {
    mu.Lock()
    fmt.Fprintf(w, "Count %d\n", count)
    mu.Unlock()
}
```

The server has two handlers, and the request URL determines which one is called: a request for `/count` invokes `counter` and all others invoke `handler`. A handler pattern that ends with a slash matches any URL that has the pattern as a prefix. Behind the scenes, the server runs the handler for each incoming request in a separate goroutine so that it can serve multiple requests simultaneously. However, if two concurrent requests try to update `count` at the same time, it might not be incremented consistently; the program would have a serious bug called a *race condition* (§9.1). To avoid this problem, we must ensure that at most one goroutine accesses the variable at a time, which is the purpose of the `mu.Lock()` and `mu.Unlock()` calls that bracket each access of `count`. We'll look more closely at concurrency with shared variables in Chapter 9.

As a richer example, the handler function can report on the headers and form data that it receives, making the server useful for inspecting and debugging requests:

gopl.io/ch1/server3

```
// handler echoes the HTTP request.
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "%s %s %s\n", r.Method, r.URL, r.Proto)
    for k, v := range r.Header {
        fmt.Fprintf(w, "Header[%q] = %q\n", k, v)
    }
    fmt.Fprintf(w, "Host = %q\n", r.Host)
    fmt.Fprintf(w, "RemoteAddr = %q\n", r.RemoteAddr)
    if err := r.ParseForm(); err != nil {
        log.Print(err)
    }
    for k, v := range r.Form {
        fmt.Fprintf(w, "Form[%q] = %q\n", k, v)
    }
}
```

This uses the fields of the `http.Request` struct to produce output like this:

```
GET /?q=query HTTP/1.1
Header["Accept-Encoding"] = ["gzip, deflate, sdch"]
Header["Accept-Language"] = ["en-US,en;q=0.8"]
Header["Connection"] = ["keep-alive"]
Header["Accept"] = ["text/html,application/xhtml+xml,application/xml;..."]
Header["User-Agent"] = ["Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_5)..."]
Host = "localhost:8000"
RemoteAddr = "127.0.0.1:59911"
Form["q"] = ["query"]
```


Notice how the call to `ParseForm` is nested within an `if` statement. Go allows a simple statement such as a local variable declaration to precede the `if` condition, which is particularly useful for error handling as in this example. We could have written it as

```
err := r.ParseForm()
if err != nil {
    log.Print(err)
}
```

but combining the statements is shorter and reduces the scope of the variable `err`, which is good practice. We'll define scope in Section 2.7.

In these programs, we've seen three very different types used as output streams. The `fetch` program copied HTTP response data to `os.Stdout`, a file, as did the `lissajous` program. The `fetchall` program threw the response away (while counting its length) by copying it to the trivial sink `ioutil.Discard`. And the web server above used `fmt.Fprintf` to write to an `http.ResponseWriter` representing the web browser.

Although these three types differ in the details of what they do, they all satisfy a common *interface*, allowing any of them to be used wherever an output stream is needed. That interface, called `io.Writer`, is discussed in Section 7.1.

Go's interface mechanism is the topic of Chapter 7, but to give an idea of what it's capable of, let's see how easy it is to combine the web server with the `lissajous` function so that animated GIFs are written not to the standard output, but to the HTTP client. Just add these lines to the web server:

```
handler := func(w http.ResponseWriter, r *http.Request) {
    lissajous(w)
}
http.HandleFunc("/", handler)
```

or equivalently:

```
http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    lissajous(w)
})
```

The second argument to the `HandleFunc` function call immediately above is a *function literal*, that is, an anonymous function defined at its point of use. We will explain it further in Section 5.6.

Once you've made this change, visit `http://localhost:8000` in your browser. Each time you load the page, you'll see a new animation like the one in Figure 1.3.

Exercise 1.12: Modify the `Lissajous` server to read parameter values from the URL. For example, you might arrange it so that a URL like `http://localhost:8000/?cycles=20` sets the number of cycles to 20 instead of the default 5. Use the `strconv.Atoi` function to convert the string parameter into an integer. You can see its documentation with `go doc strconv.Atoi`.

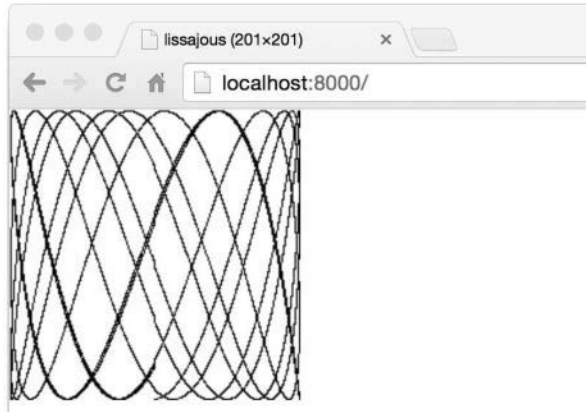


Figure 1.3. Animated Lissajous figures in a browser.

1.8. Loose Ends

There is a lot more to Go than we've covered in this quick introduction. Here are some topics we've barely touched upon or omitted entirely, with just enough discussion that they will be familiar when they make brief appearances before the full treatment.

Control flow: We covered the two fundamental control-flow statements, `if` and `for`, but not the `switch` statement, which is a multi-way branch. Here's a small example:

```
switch coinflip() {  
  case "heads":  
    heads++  
  case "tails":  
    tails++  
  default:  
    fmt.Println("landed on edge!")  
}
```

The result of calling `coinflip` is compared to the value of each case. Cases are evaluated from top to bottom, so the first matching one is executed. The optional default case matches if none of the other cases does; it may be placed anywhere. Cases do not fall through from one to the next as in C-like languages (though there is a rarely used `fallthrough` statement that overrides this behavior).

A `switch` does not need an operand; it can just list the cases, each of which is a boolean expression:

```
func Signum(x int) int {
    switch {
    case x > 0:
        return +1
    default:
        return 0
    case x < 0:
        return -1
    }
}
```

This form is called a *tagless switch*; it's equivalent to `switch true`.

Like the `for` and `if` statements, a `switch` may include an optional simple statement—a short variable declaration, an increment or assignment statement, or a function call—that can be used to set a value before it is tested.

The `break` and `continue` statements modify the flow of control. A `break` causes control to resume at the next statement after the innermost `for`, `switch`, or `select` statement (which we'll see later), and as we saw in Section 1.3, a `continue` causes the innermost `for` loop to start its next iteration. Statements may be labeled so that `break` and `continue` can refer to them, for instance to break out of several nested loops at once or to start the next iteration of the outermost loop. There is even a `goto` statement, though it's intended for machine-generated code, not regular use by programmers.

Named types: A type declaration makes it possible to give a name to an existing type. Since struct types are often long, they are nearly always named. A familiar example is the definition of a `Point` type for a 2-D graphics system:

```
type Point struct {
    X, Y int
}
var p Point
```

Type declarations and named types are covered in Chapter 2.

Pointers: Go provides pointers, that is, values that contain the address of a variable. In some languages, notably C, pointers are relatively unconstrained. In other languages, pointers are disguised as “references,” and there's not much that can be done with them except pass them around. Go takes a position somewhere in the middle. Pointers are explicitly visible. The `&` operator yields the address of a variable, and the `*` operator retrieves the variable that the pointer refers to, but there is no pointer arithmetic. We'll explain pointers in Section 2.3.2.

Methods and interfaces: A method is a function associated with a named type; Go is unusual in that methods may be attached to almost any named type. Methods are covered in Chapter 6. Interfaces are abstract types that let us treat different concrete types in the same way based on what methods they have, not how they are represented or implemented. Interfaces are the subject of Chapter 7.

Packages: Go comes with an extensive standard library of useful packages, and the Go community has created and shared many more. Programming is often more about using existing packages than about writing original code of one's own. Throughout the book, we will point out a couple of dozen of the most important standard packages, but there are many more we don't have space to mention, and we cannot provide anything remotely like a complete reference for any package.

Before you embark on any new program, it's a good idea to see if packages already exist that might help you get your job done more easily. You can find an index of the standard library packages at <https://golang.org/pkg> and the packages contributed by the community at <https://godoc.org>. The go doc tool makes these documents easily accessible from the command line:

```
$ go doc http.ListenAndServe
package http // import "net/http"

func ListenAndServe(addr string, handler Handler) error
    ListenAndServe listens on the TCP network address addr and then
    calls Serve with handler to handle requests on incoming connections.
...
```

Comments: We have already mentioned documentation comments at the beginning of a program or package. It's also good style to write a comment before the declaration of each function to specify its behavior. These conventions are important, because they are used by tools like go doc and godoc to locate and display documentation (§10.7.4).

For comments that span multiple lines or appear within an expression or statement, there is also the `/* ... */` notation familiar from other languages. Such comments are sometimes used at the beginning of a file for a large block of explanatory text to avoid a `//` on every line. Within a comment, `//` and `/*` have no special meaning, so comments do not nest.

This page intentionally left blank

Index

- !, negation operator 63
- %, remainder operator 52, 166
- &&, short-circuit AND operator 63
- &, address-of operator 24, 32, 94, 158, 167
- &, implicit 158, 167
- &^, AND-NOT operator 53
- &^, bit-clear operator 53
- ' quote character 56
- *, indirection operator 24, 32
- ++, increment statement 5, 37, 94
- +, string concatenation operator 5, 65
- +, unary operator 53
- +=, -=, etc., assignment operator 5
- , unary operator 53
- , decrement statement 5, 37
- ... argument 139, 142
- ... array length 82
- ... parameter 91, 142, 143, 172
- ... path 292, 299
- /*...*/ comment 5, 25
- // comment 5, 25
- := short variable declaration 5, 31, 49
- <<, left shift operator 54
- ==, comparison operator 40, 63
- >>, right shift operator 54
- ^, bitwise complement operator 53
- ^, exclusive OR operator 53
- _ , blank identifier 7, 38, 95, 120, 126, 287
- ` backquote character 66
- | in template 113
- |, bitwise OR operator 166, 167
- ||, short-circuit OR operator 63
- Abstract Syntax Notation One (ASN.1) 107
- abstract type 24, 171
- abstraction, premature 216, 316, 317
- ad hoc polymorphism 211
- address of local variable 32, 36
- address of struct literal 103
- addressable expression 159, 341
- addressable value 32
- address-of operator & 24, 32, 94, 158, 167
- aggregate type 81, 99
- Alef programming language xiii
- algorithm
 - breadth-first search 139, 239
 - depth-first search 136
 - Fibonacci 37, 218
 - GCD 37
 - insertion sort 101
 - Lissajous 15
 - slice rotation 86
 - topological sort 136
- aliasing, pointer 33
- alignment 354
- allocation
 - heap 36
 - memory 36, 71, 89, 169, 209, 322
 - stack 36
- anchor element, HTML 122
- AND operator &&, short-circuit 63
- AND-NOT operator &^ 53
- animation, GIF 13
- anonymous
 - function 22, 135, 236
 - function, defer 146
 - function, recursive 137
- struct field 104, 105, 106, 162
- API
 - encoding 213, 340
 - error 127, 152
 - package 284, 296, 311, 333, 352
 - runtime 324
 - SQL 211
 - system call 196
 - template 115
 - token-based decoder 213, 215, 347
- APL programming language xiii
- append built-in function 88, 90, 91
- appendInt example 88
- argument
 - ... 139, 142
 - command-line 4, 18, 33, 43, 179, 180, 290, 313
 - function 119
 - pointer 33, 83
 - slice 86
- arithmetic expression evaluator 197
- array
 - comparison 83
 - length, ... 82
 - literal 82, 84
 - type 81
 - underlying 84, 88, 91, 187
 - zero value 82
- ASCII 56, 64, 66, 67, 305
- ASN.1 (Abstract Syntax Notation One) 107
- assembly line, cake 234
- assertion
 - function 316
 - interface type 208, 210

- test 306
 - type 205, 211
- assignability 38, 175
- assignability, interface 175
- assignment
 - implicit 38
 - multiple-value 37
 - operator +=, -=, etc. 5
 - operators 36, 52
 - statement 5, 7, 36, 52, 94, 173
 - tuple 31, 37
- associativity, operator 52
- atomic operation 264
- attack, HTML injection 115
- attack, SQL injection 211
- autoescape example 117
- back-door, package 315
- back-off, exponential 130
- backquote character, ` 66
- bank example package 258, 261, 263
- bare return 126
- basename example 72
- behavior, undefined 260
- Benchmark function 302, 321
- bidirectional to unidirectional
 - channel conversion 231
- binary
 - operators, table of 52
 - semaphore 262
 - tree 102
- bit vector 165
- bit-clear operator &^ 53
- bit-set data type 77
- bitwise
 - complement operator ^ 53
 - operators, table of 53
 - OR operator | 166, 167
- black-box test 310
- blank identifier _ 7, 38, 95, 120, 126, 287
- blank import 287
- block
 - file 46
 - lexical 46, 120, 135, 141, 212
 - local 46
 - package 46
 - universe 46
- blocking profile 324
- Blog, Go xvi, 326
- boiling example 29
- bool type 63
- boolean
 - constant, false 63
 - constant, true 63
 - zero value 30
- breadthFirst function 139
- breadth-first search algorithm 139, 239
- break statement 24, 46
- break statement, labeled 249
- brittle test 317
- broadcast 251, 254, 276
- Brooks, Fred xiv
- btoi function 64
- buffered channel 226, 231
- bufio package 9
- bufio.NewReader function 98
- bufio.NewScanner function 9
- (*bufio.Reader).ReadRune
 - method 98
- bufio.Scanner type 9
- (*bufio.Scanner).Err method 97
- (*bufio.Scanner).Scan method 9
- (*bufio.Scanner).Split method 99
- bufio.ScanWords function 99
- +build comments 296
- build constraints 296
- build tags 296
- building packages 293
- built-in function
 - append 88, 90, 91
 - cap 84, 232
 - close 226, 228, 251
 - complex 61
 - copy 89
 - delete 94
 - imag 61
 - len 4, 54, 64, 65, 81, 84, 233
 - make 9, 18, 88, 94, 225
 - new 34
 - panic 148, 149
 - real 61
 - recover 152
- built-in interface, error 196
- built-in type, error 11, 128, 149, 196
- byte slice to string conversion 73
- byte type 52
- ByteCounter example 173
- bytes package 71, 73
- bytes.Buffer type 74, 169, 172, 185
- (*bytes.Buffer).Grow method 169
- (*bytes.Buffer).WriteByte
 - method 74
- (*bytes.Buffer).WriteRune
 - method 74
- (*bytes.Buffer).WriteString
 - method 74
- bytes.Equal function 86
- bzip C code 362
- bzip example package 363
- bzipper example 365
- C++ programming language xiv, xv, 361
- C programming language xii, xv, 1, 6, 52, 260, 361
- cache, concurrent non-blocking 272
- cache, non-blocking 275
- cake assembly line 234
- call
 - by reference 83
 - by value 83, 120, 158
 - interface method 182
 - ok value from function 128
- calling C from Go 361
- camel case 28
- cancellation 251, 252
- cancellation of HTTP request 253
- cap built-in function 84, 232
- capacity, channel 226, 232, 233
- capacity, slice 88, 89
- capturing iteration variable 140
- capturing loop variable 141, 236, 240
- case in type switch 212
- case, select 245
- Celsius type 39
- CelsiusFlag function 181
- cf example 43
- cgo tool 361, 362
- <-ch, channel receive 18, 225, 232
- ch<- , channel send 18, 225, 232
- chaining, method 114
- chan type 225
- channel
 - buffered 226, 231
 - capacity 226, 232, 233
 - close 228, 251
 - closing a 225
 - communication 225, 245
 - comparison 225
 - conversion, bidirectional to unidirectional 231
 - draining a 229, 252
 - make 18, 225
 - nil 246, 249
 - polling 246
 - range over 229
 - receive <-ch 18, 225, 232
 - receive, non-blocking 246
 - receive, ok value from 229
 - send ch<- 18, 225, 232
 - synchronous 226
 - type 18
 - type <-chan T, receive-only 230
 - type chan<- T, send-only 230
 - type, unidirectional 230, 231
 - unbuffered 226
 - zero value 225, 246
- character conversion 71
- character test 71
- charcount example 98
- chat example 254

- chat server 253
- CheckQuota function 312, 313
- client, email 312
- client, SMTP 312
- clock example 220, 222
- clock server, concurrent 219
- close built-in function 226, 228, 251
- close, channel 228, 251
- closer goroutine 238, 250
- closing a channel 225
- closure, lexical 136
- cmp1x.Sqrt function 61
- code
 - format 3, 6, 9, 48
 - point, Unicode 67
 - production 301
- ColoredPoint example 161
- comma example 73
- command, testing a 308
- command-line argument 4, 18, 33, 43, 179, 180, 290, 313
- comment
 - /*...*/ 5, 25
 - // 5, 25
 - doc 42, 296
 - // Output 326
- comments, +build 296
- communicating sequential processes (CSP) xiii, 217
- communication, channel 225, 245
- comparability 9, 38, 40, 53, 86, 93, 97, 104
- comparison
 - array 83
 - channel 225
 - function 133
 - interface 184
 - map 96
 - operator == 40, 63
 - operators 40, 93
 - operators, table of 53
 - slice 87
 - string 65
 - struct 104
- compilation, separate 284
- complement operator ^, bitwise 53
- complex built-in function 61
- complex type 61
- composite literal 14
- composite type xv, 14, 81
- composition, parallel 224
- composition, type xv, 107, 162, 189
- compress/bzip2 package 361
- compression 361
- conceptual integrity xiv
- concrete type 24, 171, 211, 214
- concurrency 17, 217, 257
 - excessive 241, 242
 - safe 275
 - safety 256, 257, 272, 365
 - with shared variables 257
- concurrent
 - clock server 219
 - directory traversal 247
 - echo server 222
 - non-blocking cache 272
 - web crawler 239
- confinement, serial 262
- confinement, variable 261
- consistency, sequential 268, 269
- const declaration 14, 75
- constant
 - false boolean 63
 - generator, iota xiii, 77
 - time.Minute 76
 - time.Second 164
 - true boolean 63
 - types, untyped 78
- constants, precision of 78
- constraints, build 296
- contention, lock 267, 272
- context switch 280
- continue statement 24, 46
- continue statement, labeled 249
- contracts, interfaces as 171
- control flow 46
- conversion
 - bidirectional to unidirectional
 - channel 231
 - byte slice to string 73
 - character 71
 - implicit 79
 - narrowing 40, 55
 - numeric 79
 - operation 40, 55, 64, 71, 78, 79, 173, 187, 194, 208, 231, 353, 358
 - rune slice to string 71
 - rune to string 71
 - string 71
 - string to byte slice 40, 73
 - string to rune slice 71, 88
 - unsafe.Pointer 356
- copy built-in function 89
- countdown example 244, 245, 246
- counting semaphore 241
- coverage, statement 318, 320
- coverage, test 318
- coverage_test example 319
- CPU profile 324
- crawl example 240, 242, 243
- crawler, concurrent web 239
- crawler, web 119
- critical section 263, 270, 275
- cross-compilation 295
- cryptography 55, 83, 121, 325
- crypto/sha256 package 83
- customSort example 190
- cyclic data structure 337
- cyclic test dependency 314
- data
 - race 259, 267, 275
 - structure, cyclic 337
 - structure, recursive 101, 102, 107
 - type, bit-set 77
- database driver, MySQL 284
- database/sql package 211, 288
- daysAgo function 114
- deadbeef 55, 80
- deadlock 233, 240, 265
- declaration
 - const 14, 75
 - func 3, 29, 119
 - import 3, 28, 42, 284, 285, 362
 - method 40, 155
 - package 2, 28, 41, 285
 - package-level 28
 - scope 45, 137
 - shadowing 46, 49, 206, 212
 - short variable 5, 7, 30, 31
 - statement, short variable 7
 - struct 99
 - type 39
 - var 5, 30
- declarations, order of 48
- decode example, S-expression 347
- decoder API, token-based 213, 215, 347
- decoding, S-expression 344
- decoding, XML 213
- decrement statement -- 5, 37
- dedup example 97
- deep equivalence 87, 317, 358
- default case in select 246
- default case in switch 23
- default case in type switch 212
- defer anonymous function 146
- defer example 150, 151
- defer statement 144, 150, 264
- deferred function call 144
- delete built-in function 94
- depth-first search algorithm 136
- dereference, implicit 159
- diagram
 - helloWorld substring 69
 - pipeline 228
 - slice capacity growth 90
 - slice of months 84
 - string sharing 65
 - struct hole 355
 - thumbnail sequence 238
- digital artifact example 178
- Dijkstra, Edsger 318
- Dilbert 100
- directed acyclic graph 136, 284
- directory traversal, concurrent 247

- discriminated union 211, 213, 214
- Display function 333
- display example 333
- display function 334
- displaying methods of a type 351
- Distance function 156
- doc comment 42, 296
- doc.go doc comment file 42, 296
- documentation, package 296
- domain name, import path 284
- dot . in template 113
- downloading packages 292
- Dr. Strangelove 336
- draining a channel 229, 252
- du example 247, 249, 250
- dup example 9, 11, 12
- duplicate suppression 276
- dynamic dispatch 183
- dynamic type, interface 181
- echo example 5, 7, 34, 309
- echo test 309
- echo server, concurrent 222
- echo_test.go 310
- effective tests, writing 316, 317
- email client 312
- embarrassingly parallel 235
- embedded struct field 161
- embedding, interface 174
- embedding, struct 104, 161
- Employee struct 100
- empty
 - interface type 176
 - select statement 245
 - string 5, 7, 30
 - struct 102
- encapsulation 168, 284
- encoding API 213, 340
- encoding, S-expression 338
- encoding/json package 107
- encoding/xml package 107, 213
- end of file (EOF) 131
- enum 77
- environment variable
 - GOARCH 292, 295
 - GOMAXPROCS 281, 321
 - GOOS 292, 295
 - GOPATH xvi, 291, 295
 - GOROOT 292
- equal function 87, 96
- equality, pointer 32
- equivalence, deep 87, 317, 358
- error built-in interface 196
- error built-in type 11, 128, 149, 196
- error API 127, 152
- error.Error method 196
- errorf function 143
- error-handling strategies 128, 152, 310, 316
- errors package 196
- errors.New function 196
- escape
 - hexadecimal 66
 - HTML 116
 - octal 66
 - sequence 10
 - sequences, table of 66
 - Unicode 68, 107
 - URL 111
- escaping variables 36
- eval example 198
- event multiplexing 244
- events 227, 244
- Example function 302, 326
- example
 - autoescape 117
 - basename 72
 - boiling 29
 - ByteCounter 173
 - zipper 365
 - cf 43
 - charcount 98
 - chat 254
 - clock 220, 222
 - ColoredPoint 161
 - comma 73
 - countdown 244, 245, 246
 - coverage_test 319
 - crawl 240, 242, 243
 - customSort 190
 - dedup 97
 - defer 150, 151
 - digital artifact 178
 - display 333
 - du 247, 249, 250
 - dup 9, 11, 12
 - echo 5, 7, 34, 309
 - eval 198
 - fetch 16, 148
 - fetchall 18
 - findlinks 122, 125, 139
 - ftoc 29
 - github 110, 111
 - graph 99
 - helloworld 1, 2
 - http 192, 194, 195
 - intset 166
 - issues 112
 - issueshtml 115
 - issuesreport 114
 - jpeg 287
 - lissajous 14, 22, 35
 - mandelbrot 62
 - memo 275, 276, 277, 278, 279
 - methods 351
 - movie 108, 110
 - netcat 221, 223, 227
 - netflag 78
 - nonempty 92
 - outline 123, 133
 - package, bank 258, 261, 263
 - package, bzip 363
 - package, format 332
 - package, geometry 156
 - package, http 192
 - package, links 138
 - package, memo 273
 - package, params 348
 - package, storage 312, 313
 - package, tempconv 42
 - package, thumbnail 235
 - palindrome 303, 305, 308
 - params 348
 - Parse 152
 - pipeline 228, 230, 231
 - playlist 187
 - rev 86
 - reverb 223, 224
 - server 19, 21
 - sexpr 340
 - S-expression decode 347
 - sha256 83
 - sleep 179
 - spinner 218
 - squares 135
 - sum 142
 - surface 59, 203
 - tempconv 39, 180, 289
 - temperature conversion 29
 - tempflag 181
 - test of word 303
 - thumbnail 236, 237, 238
 - title 153
 - topoSort 136
 - trace 146
 - treesort 102
 - urlvalues 160
 - wait 130
 - word 303, 305, 308
 - xmlselect 215
 - appendInt 88
- exception 128, 149
- excessive concurrency 241, 242
- exclusion, mutual 262, 267
- exclusive lock 263, 266, 270
- exclusive OR operator ^ 53
- exponential back-off 130
- export of struct field 101, 106, 109, 110, 168
- export_test.go file 315
- Expr.Check method 202
- expression
 - addressable 159, 341
 - evaluator 197
 - method 164
 - receive 225
- Expr.Eval method 199

- extending a slice 86
- Extensible Markup Language (XML) 107
- external test package 285, 314
- Fahrenheit type 39
- failure message, test 306
- fallthrough statement 23, 212
- false boolean constant 63
- fetch example 16, 148
- fetchall example 18
- fib function 37, 218
- Fibonacci algorithm 37, 218
- field
 - anonymous struct 104, 105, 106, 162
 - embedded struct 161
 - export of struct 101, 106, 109, 110, 168
 - order, struct 101, 355
 - selector 156
 - struct 15, 99
 - tag, omitempty 109
 - tag, struct 109, 348
- figure
 - Lissajous 13
 - Mandelbrot 63
 - 3-D surface 58, 203
- File Transfer Protocol (FTP) 222
- file
 - block 46
 - export_test.go 315
 - name, Microsoft Windows 72
 - name, POSIX 72
 - _test.go 285, 302, 303
- findlinks example 122, 125, 139
- fixed-size stack 124
- flag package 33, 179
- flag
 - go tool -bench 321
 - go tool -benchmem 322
 - go tool -covermode 319
 - go tool -coverprofile 319
 - go tool -cpuprofile 324
 - go tool -nodecount 325
 - go tool -text 325
 - go tool -web 326
 - godoc -analysis 176
 - go list -f 315
 - go -race 271
 - go test -race 274
 - go test -run 305
 - go test -v 304
- flag.Args function 34
- flag.Bool function 34
- flag.Duration function 179
- flag.Parse function 34
- flag.String function 34
- flag.Value interface 179, 180
- floating-point
 - number 56
 - precision 56, 57, 63, 78
 - truncation 40, 55
- fmt package 2
- fmt.Errorf function 129, 196
- fmt.Fprintf function 172
- fmt.Printf function 10
- fmt.Println function 2
- fmt.Scanf function 75
- fmt.Sscanf function 180
- fmt.Stringer interface 180, 210
- for scope 47
- for statement 6
- forEachNode function 133
- foreign-function interface (FFI) 361
- format, code 3, 6, 9, 48
- format example package 332
- formatAtom function 332
- framework, web 193
- ftoc example 29
- func declaration 3, 29, 119
- function
 - anonymous 22, 135, 236
 - append built-in 88, 90, 91
 - argument 119
 - assertion 316
 - Benchmark 302, 321
 - body, missing 121
 - breathFirst 139
 - btoi 64
 - bufio.NewReader 98
 - bufio.NewScanner 9
 - bufio.ScanWords 99
 - bytes.Equal 86
 - call, deferred 144
 - call, ok value from 128
 - cap built-in 84, 232
 - CelsiusFlag 181
 - CheckQuota 312, 313
 - close built-in 226, 228, 251
 - cmplx.Sqrt 61
 - comparison 133
 - complex built-in 61
 - copy built-in 89
 - daysAgo 114
 - delete built-in 94
 - Display 333
 - display 334
 - Distance 156
 - equal 87, 96
 - errorf 143
 - errors.New 196
 - Example 302, 326
 - fib 37, 218
 - flag.Args 34
 - flag.Bool 34
 - flag.Duration 179
 - flag.Parse 34
- flag.String 34
- fmt.Errorf 129, 196
- fmt.Fprintf 172
- fmt.Printf 10
- fmt.Println 2
- fmt.Scanf 75
- fmt.Sscanf 180
- forEachNode 133
- formatAtom 332
- gcd 37
- handler 19, 21, 152, 191, 194, 195, 348
- html.Parse 121, 125
- http.DefaultServeMux 195
- http.Error 193
- http.Get 16, 18
- http.Handle 195
- http.HandleFunc 19, 22, 195
- http.ListenAndServe 19, 191
- http.NewRequest 253
- http.ServeMux 193
- hypot 120
- imag built-in 61
- image.Decode 288
- image.RegisterFormat 288
- incr 33
- init 44, 49
- intsToString 74
- io.Copy 17, 18
- ioutil.ReadAll 16, 272
- ioutil.ReadDir 247
- ioutil.ReadFile 12, 145
- io.WriteString 209
- itob 64
- json.Marshal 108
- json.MarshalIndent 108
- json.NewDecoder 111
- json.NewEncoder 111
- json.Unmarshal 110, 114
- len built-in 4, 54, 64, 65, 81, 84, 233
- links.Extract 138
- literal 22, 135, 227
- log.Fatalf 49, 130
- main 2, 310
- make built-in 9, 18, 88, 94, 225
- math.Hypot 156
- math.Inf 57
- math.IsInf 57
- math.IsNaN 57
- math.NaN 57
- multi-valued 11, 30, 37, 96, 125, 126
- mustCopy 221
- net.Dial 220
- net.Listen 220
- new built-in 34
- nil 132
- os.Close 11

- os.Exit 16, 34, 48
- os.Getwd 48
- os.IsExist 207
- os.IsNotExist 207
- os.IsPermission 207
- os.Open 11
- os.Stat 247
- panic built-in 148, 149
- parameter 119
- params.Unpack 349
- png.Encode 62
- PopCount 45
- real built-in 61
- recover built-in 152
- recursive anonymous 137
- reflect.TypeOf 330
- reflect.ValueOf 331, 337
- reflect.Zero 345
- regexp.Compile 149
- regexp.MustCompile 149
- result list 119
- runtime.Stack 151
- SearchIssues 111
- sexpr.Marshal 340
- sexpr.readList 347
- sexpr.Unmarshal 347
- signature 120
- sort.Float64s 191
- sort.Ints 191
- sort.IntsAreSorted 191
- sort.Reverse 189
- sort.Strings 95, 137, 191
- Sprint 330
- sqlQuote 211, 212
- strconv.Atoi 22, 75
- strconv.FormatInt 75
- strconv.Itoa 75
- strconv.ParseInt 75
- strconv.ParseUint 75
- strings.Contains 69
- strings.HasPrefix 69
- strings.HasSuffix 69
- strings.Index 289
- strings.Join 7, 12
- strings.Map 133
- strings.NewReader 289
- strings.NewReplacer 289
- strings.Split 12
- strings.ToLower 72
- strings.ToUpper 72
- template.Must 114
- template.New 114
- Test 302
- time.After 245
- time.AfterFunc 164
- time.Now 220
- time.Parse 220
- time.Since 114
- time.Tick 244, 246
- title 144, 145
- type 119, 120
- unicode.IsDigit 71
- unicode.IsLetter 71
- unicode.IsLower 71
- unicode.IsSpace 93
- unicode.IsUpper 71
- unsafe.AlignOf 355
- unsafe.Offsetof 355
- unsafe.Sizeof 354
- url.QueryEscape 111
- utf8.DecodeRuneInString 69
- utf8.RuneCountInString 69
- value 132
- variadic 142, 172
- visit 122
- WaitForServer 130
- walkDir 247
- zero value 132
- garbage collection xi, xiii, 7, 35, 230, 353, 357
- garbage collector, moving 357
- GCD algorithm 37
- gcd function 37
- geometry example package 156
- geometry.Point.Distance method 156
- getter method 169
- GIF animation 13
- GitHub issue tracker 110
- github example 110, 111
- Go
 - Playground xvi, 326
 - Blog xvi, 326
 - issue 110, 112, 358
- go tool 2, 42, 44, 290
- go tool -bench flag 321
- go tool -benchmem flag 322
- go tool -covermode flag 319
- go tool -coverprofile flag 319
- go tool -cpuprofile flag 324
- go tool -nodecount flag 325
- go tool pprof 325
- go tool -text flag 325
- go tool -web flag 326
- go tool cover 318, 319
- go doc tool 25
- go statement 18, 218
- GOARCH environment variable 292, 295
- go build 2, 286, 293, 294
- go doc 296
- godoc -analysis flag 176
- godoc tool xvi, 25, 297, 326
- go env 292
- gofmt tool 3, 4, 44, 286
- go get xvi, 2, 292, 293
- go help 290
- goimports tool 3, 44, 286
- go install 295
- golang.org/x/net/html package 122
- golint tool 292
- go list 298, 315
- go list -f flag 315
- GOMAXPROCS environment variable 281, 321
- GOOS environment variable 292, 295
- GOPATH environment variable xvi, 291, 295
- gopl.io repository xvi
- go -race flag 271
- GOROOT environment variable 292
- goroutine 18, 217, 233, 235
 - closer 238, 250
 - identity 282
 - leak 233, 236, 246
 - monitor 261, 277
 - multiplexing 281
 - vs. OS thread 280
- go run 2, 294
- go test 301, 302, 304
- go test -race flag 274
- go test -run flag 305
- go test -v flag 304
- goto statement 24
- graph example 99
- GraphViz 326
- Griesemer, Robert xi
- growth, stack 124, 280, 358
- guarding mutex 263
- half-open interval 4
- handler function 19, 21, 152, 191, 194, 195, 348
- “happens before” relation 226, 257, 261, 277
- “has a” relationship 162
- hash table 9, 93
- Haskell programming language xiv
- heap
 - allocation 36
 - profile 324
 - variable 36
- helloworld example 1, 2
- helloworld substring diagram 69
- hexadecimal escape 66
- hexadecimal literal 55
- hidden pointer 357
- Hoare, Tony xiii
- hole, struct 354
- HTML
 - anchor element 122
 - escape 116
 - injection attack 115
 - metacharacter 116
 - parser 121

- html.Parse function 121, 125
- html/template package 113, 115
- HTTP
 - GET request 21, 127, 272, 348
 - POST request 348
 - request, cancellation of 253
 - request multiplexer 193
- http example 192, 194, 195
- http example package 192
- (*http.Client).Do method 253
- http.DefaultClient variable 253
- http.DefaultServeMux function 195
- http.Error function 193
- http.Get function 16, 18
- http.Handle function 195
- http.HandleFunc function 19, 22, 195
- http.Handler interface 191, 193
- http.HandlerFunc type 194, 203
- http.ListenAndServe function 19, 191
- http.NewRequest function 253
- http.Request type 21, 253
- (*http.Request).ParseForm method 22, 348
- http.ResponseWriter type 19, 22, 191, 193
- http.ServeMux function 193
- hypot function 120
- identifier `_`, blank 7, 38, 95, 120, 126, 287
- identifier, qualified 41, 43
- identity, goroutine 282
- IEEE 754 standard 56, 57
- if, initialization statement in 22, 206
- if-else scope 47
- if-else statement 9, 22, 47
- imag built-in function 61
- image manipulation 121
- image package 62, 287
- image/color package 14
- image.Decode function 288
- image/png package 288
- image.RegisterFormat function 288
- imaginary literal 61
- immutability 261
- immutability, string 65, 73
- implementation with slice, stack 92, 215
- implicit
 - & 158, 167
 - assignment 38
 - conversion 79
 - dereference 159
- import declaration 3, 28, 42, 284, 285, 362
- import
 - blank 287
 - path 284
 - path domain name 284
 - renaming 286
- incr function 33
- increment statement `++` 5, 37, 94
- index operation, string 64
- indirection operator `*` 24, 32
- infinite loop 6, 120, 228
- information hiding 168, 284
- init function 44, 49
- initialization
 - lazy 268
 - package 44
 - statement in if 22, 206
 - statement in switch 24
- initializer list 30
- injection attack, HTML 115
- injection attack, SQL 211
- in-place slice techniques 91
- insertion sort algorithm 101
- int type 52
- integer
 - literal 55
 - overflow 53, 113
 - signed 52, 54
 - unsigned 52, 54
- integration test 314
- interface
 - assignability 175
 - comparison 184
 - dynamic type 181
 - embedding 174
 - error built-in 196
 - flag.Value 179, 180
 - fmt.Stringer 180, 213
 - http.Handler 191, 193
 - io.Closer 174
 - io.Reader 174
 - io.Writer 15, 22, 172, 174, 186, 208, 209, 309
 - JSON 110
 - method call 182
 - nil 182
 - pitfall 184
 - ReadWriteCloser 174
 - ReadWriter 174
 - satisfaction 171, 175
 - sort.Interface 186
 - type 171, 174
- interface{} type 143, 176, 331
- interface
 - type assertion 208, 210
 - type, empty 176
 - value 181
 - with nil pointer 184
 - zero value 182
- interfaces as contracts 171
- internal package 298
- intset example 166
- intsToString function 74
- invariants 159, 169, 170, 265, 284, 311, 352
- io package 174
- io.Closer interface 174
- io.Copy function 17, 18
- io.Discard stream 22
- io.Discard variable 18
- io.EOF variable 132
- io/ioutil package 16, 145
- io.Reader interface 174
- iota constant generator xiii, 77
- ioutil.ReadAll function 16, 272
- ioutil.ReadDir function 247
- ioutil.ReadFile function 12, 145
- io.Writer interface 15, 22, 172, 174, 186, 208, 209, 309
- io.WriteString function 209
- “is a” relationship 162, 175
- issue, Go 110, 112, 358
- issue tracker, GitHub 110
- issues example 112
- issueshtml example 115
- issuesreport example 114
- iteration order, map 95
- iteration variable, capturing 140
- itob function 64
- Java programming language xv
- JavaScript Object Notation (JSON) 107, 338
- JavaScript programming language xv, 107
- jpeg example 287
- JSON
 - interface 110
 - interface, Open Movie Database 113
 - interface, xkcd 113
 - marshaling 108
 - unmarshaling 110
- json.Decoder type 111
- json.Encoder type 111
- json.Marshal function 108
- json.MarshalIndent function 108
- json.NewDecoder function 111
- json.NewEncoder function 111
- json.Unmarshal function 110, 114
- keyword, type 212
- keywords, table of 27
- Knuth, Donald 323
- label scope 46
- label, statement 46
- labeled

- break statement 249
- continue statement 249
- statement 46
- layout, memory 354, 355
- lazy initialization 268
- leak, goroutine 233, 236, 246
- left shift operator << 54
- len built-in function 4, 54, 64, 65, 81, 84, 233
- lexical block 46, 120, 135, 141, 212
- lexical closure 136
- lifetime, variable 35, 46, 135
- links example package 138
- links.Extract function 138
- Lisp programming language 338
- Lissajous algorithm 15
- Lissajous figure 13
- lissajous example 14, 22, 35
- list, initializer 30
- literal
 - array 82, 84
 - composite 14
 - function 22, 135, 227
 - hexadecimal 55
 - imaginary 61
 - integer 55
 - map 94
 - octal 55
 - raw string 66
 - rune 56
 - slice 38, 86
 - string 65
 - struct 15, 102, 106
- local
 - block 46
 - variable 29, 141
 - variable, address of 32, 36
 - variable scope 135
- locating packages 291
- lock
 - contention 267, 272
 - exclusive 263, 266, 270
 - mutex 102, 263, 264, 324
 - non-reentrant 265
 - readers 266
 - shared 266
 - writer 266
- log package 49, 130, 170
- log.Fatalf function 49, 130
- lookup m[key], map 94
- lookup, ok value from map 96
- loop
 - infinite 6, 120, 228
 - range 6, 9
 - variable, capturing 141, 236, 240
 - variable scope 141, 236
 - while 6
- main function 2, 310
- main, package 2, 285, 310
- make built-in function 9, 18, 88, 94, 225
- make channel 18, 225
- make map 9, 18, 94
- make slice 88, 322
- Mandelbrot figure 63
- Mandelbrot set 61
- mandelbrot example 62
- map
 - as set 96, 202
 - comparison 96
 - element, nonexistent 94, 95
 - iteration order 95
 - literal 94
 - lookup m[key] 94
 - lookup, ok value from 96
 - make 9, 18, 94
 - nil 95
 - range over 94
 - type 9, 93
 - with slice key 97
 - zero value 95
- marshaling JSON 108
- math package 14, 56
- math/big package 63
- math/cmplx package 61
- math.Hypot function 156
- math.Inf function 57
- math.IsInf function 57
- math.IsNaN function 57
- math.NaN function 57
- math/rand package 285, 308
- memo example 275, 276, 277, 278, 279
- memo example package 273
- memoization 272
- memory allocation 36, 71, 89, 169, 209, 322
- memory layout 354, 355
- metacharacter, HTML 116
- method
 - (*bufio.Reader).ReadRune 98
 - (*bufio.Scanner).Err 97
 - (*bufio.Scanner).Scan 9
 - (*bufio.Scanner).Split 99
 - (*bytes.Buffer).Grow 169
 - (*bytes.Buffer).WriteByte 74
 - (*bytes.Buffer).WriteRune 74
 - (*bytes.Buffer).WriteString 74
 - call, interface 182
 - chaining 114
 - declaration 40, 155
 - error.Error 196
 - Expr.Check 202
 - expression 164
 - Expr.Eval 199
 - geometry.Point.Distance 156
 - getter 169
 - (*http.Client).Do 253
 - (*http.Request).ParseForm 22, 348
 - name 156
 - net.Conn.Close 220
 - net.Listener.Accept 220
 - (*os.File).Write 183
 - path.Distance 157
 - promotion 161
 - receiver name 157
 - receiver parameter 156
 - receiver type 157
 - reflect.Type.Field 348
 - reflect.Value.Addr 342
 - reflect.Value.CanAddr 342
 - reflect.Value.Interface 331, 342
 - reflect.Value.Kind 332
 - selector 156
 - setter 169
 - String 40, 166, 329
 - (*sync.Mutex).Lock 21, 146, 263
 - (*sync.Mutex).Unlock 21, 146, 263
 - (*sync.Once).Do 270
 - (*sync.RWMutex).RLock 266
 - (*sync.RWMutex).RUnlock 266
 - (*sync.WaitGroup).Add 238
 - (*sync.WaitGroup).Done 238
 - template.Funcs 114
 - template.Parse 114
 - (*testing.T).Errorf 200, 304, 306
 - (*testing.T).Fatal 306
 - time.Time.Format 220
 - value 164
 - (*xml.Decoder).Token 213
- methods example 351
- methods of a type, displaying 351
- Microsoft Windows file name 72
- missing function body 121
- m[key], map lookup 94
- mobile platforms 121
- Modula-2 programming language xiii
- modularity 283
- monitor 264, 275
- monitor goroutine 261, 277
- movie example 108, 110
- moving garbage collector 357
- multimap 160, 193
- multiple-value assignment 37
- multiplexer, HTTP request 193
- multiplexing, event 244
- multiplexing, goroutine 281
- multithreading, shared-memory 217, 257
- multi-valued function 11, 30, 37, 96,

- 125, 126
- mustCopy function 221
- mutex 145, 163, 256, 269
 - guarding 263
 - lock 102, 263, 264, 324
 - read/write 266, 267
- mutual exclusion 262, 267
- MySQL database driver 284
- name
 - method 156
 - method receiver 157
 - package 28, 43
 - parameter 120
 - space 41, 156, 283
- named
 - result 120, 126
 - result zero value 120, 127
 - type 24, 39, 40, 105, 157
- naming convention 28, 169, 174, 289
- naming, package 289
- NaN (not a number) 57, 93
- narrowing conversion 40, 55
- negation operator ! 63
- net package 219
- netcat example 221, 223, 227
- net.Conn type 220
- net.Conn.Close method 220
- net.Dial function 220
- netflag example 78
- net/http package 16, 191
- net.Listen function 220
- net.Listener type 220
- net.Listener.Accept method 220
- net/smtp package 312
- net/url package 160
- networking 121, 219
- new built-in function 34
- new, redefining 35
- nil
 - channel 246, 249
 - function 132
 - interface 182
 - map 95
 - pointer 32
 - pointer, interface with 184
 - receiver 159, 185
 - slice 87
- non-blocking
 - cache 275
 - cache, concurrent 272
 - channel receive 246
 - select 246
- nonempty example 92
- nonexistent map element 94, 95
- non-reentrant lock 265
- non-standard package 121
- number, floating-point 56
- number zero value 5, 30
- numeric
 - conversion 79
 - precision 55, 78
 - type 51
- Oberon programming language xiii
- object 156
- object-oriented programming (OOP) 155, 168
- octal escape 66
- octal literal 55
- ok value 37
- ok value from channel receive 229
- ok value from function call 128
- ok value from map lookup 96
- ok value from type assertion 206
- omitempty field tag 109
- Open Movie Database JSON interface 113
- operation, atomic 264
- operation, conversion 40, 55, 64, 71, 78, 79, 173, 187, 194, 208, 231, 353, 358
- operator
 - +=, -=, etc., assignment 5
 - &, address-of 24, 32, 94, 158, 167
 - &^, AND-NOT 53
 - &^, bit-clear 53
 - ^, bitwise complement 53
 - ^, bitwise OR 166, 167
 - ==, comparison 40, 63
 - ^, exclusive OR 53
 - *, indirection 24, 32
 - <<, left shift 54
 - !, negation 63
 - %, remainder 52, 166
 - >>, right shift 54
 - &&, short-circuit AND 63
 - ||, short-circuit OR 63
 - +, string concatenation 5, 65
 - , unary 53
 - +, unary 53
 - associativity 52
 - precedence 52, 63
 - s[i:j], slice 84, 86
 - s[i:j], substring 65, 86
- operators
 - assignment 36, 52
 - comparison 40, 93
 - table of binary 52
 - table of bitwise 53
 - table of comparison 53
- optimization 264, 321, 323
- optimization, premature 324
- OR operator ||, short-circuit 63
- order of declarations 48
- order, struct field 101, 355
- organization, workspace 291
- OS thread vs. goroutine 280
- os package 4, 206
- os.Args variable 4
- os.Close function 11
- os.Exit function 16, 34, 48
- *os.File type 11, 13, 172, 175, 185, 336
- os.FileInfo type 247
- (*os.File).Write method 183
- os.Getwd function 48
- os.IsExist function 207
- os.IsNotExist function 207
- os.IsPermission function 207
- os.LinkError type 207
- os.Open function 11
- os.PathError type 207
- os.Stat function 247
- outline example 123, 133
- // Output comment 326
- overflow, integer 53, 113
- overflow, stack 124
- package declaration 2, 28, 41, 285
- package
 - API 284, 296, 311, 333, 352
 - back-door 315
 - bank example 258, 261, 263
 - block 46
 - bufio 9
 - bytes 71, 73
 - bzip example 363
 - compress/bzip2 361
 - crypto/sha256 83
 - database/sql 211, 288
 - documentation 296
 - encoding/json 107
 - encoding/xml 107, 213
 - errors 196
 - external test 285, 314
 - flag 33, 179
 - fmt 2
 - format example 332
 - geometry example 156
 - golang.org/x/net/html 122
 - html/template 113, 115
 - http example 192
 - image 62, 287
 - image/color 14
 - image/png 288
 - initialization 44
 - internal 298
 - io 174
 - io/ioutil 16, 145
 - links example 138
 - log 49, 130, 170
 - main 2, 285, 310
 - math 14, 56
 - math/big 63
 - math/cmplx 61

- math/rand 285, 308
- memo example 273
- name 28, 43
- naming 289
- net 219
- net/http 16, 191
- net/smtp 312
- net/url 160
- non-standard 121
- os 4, 206
- params example 348
- path 72
- path/filepath 72
- reflect 330
- regexp 149
- runtime 151
- sort 95, 186, 189
- storage example 312, 313
- strconv 22, 71, 75
- strings 7, 71, 72, 289
- sync 237, 263
- syscall 196, 208
- tempconv example 42
- testing 285, 302
- text/scanner 344
- text/tabwriter 188
- text/template 113, 300
- thumbnail example 235
- time 18, 77, 183
- unicode 71
- unicode/utf8 69
- unsafe 354
- package-level declaration 28
- packages
 - building 293
 - downloading 292
 - locating 291
 - querying 298
- palindrome 191
- palindrome example 303, 305, 308
- panic 64, 152, 253
- panic built-in function 148, 149
- paradoxical race 267
- parallel composition 224
- parallel, embarrassingly 235
- parallelism 217
- parameter
 - ... 91, 142, 143, 172
 - function 119
 - method receiver 156
 - name 120
 - passing 120
 - unused 120
- params example 348
- params example package 348
- params.Unpack function 349
- parentheses 4, 6, 9, 52, 63, 119, 146, 158, 285, 335, 345
- Parse example 152
- parser, HTML 121
- Pascal programming language xiii
- path, ... 292, 299
- path package 72
- path.Distance method 157
- path/filepath package 72
- Pike, Rob xi, xiii, 67, 107
- pipeline example 228, 230, 231
- pipeline 227
- pipeline diagram 228
- pitfall, interface 184
- pitfall, scope 140
- platforms, mobile 121
- Playground, Go xvi, 326
- playlist example 187
- png.Encode function 62
- pointer 24, 32, 34
 - aliasing 33
 - argument 33, 83
 - equality 32
 - hidden 357
 - nil 32
 - receiver 158, 167
 - to struct 100, 103
 - zero value 32
- polling channel 246
- polymorphism, ad hoc 211
- polymorphism, subtype 211
- PopCount function 45
- Portable Network Graphics (PNG) 62
- POSIX file name 72
- POSIX standard xi, 55, 72, 197
- precedence, operator 52, 63
- precision
 - floating-point 56, 57, 63, 78
 - numeric 55, 78
 - of constants 78
- predeclared names, table of 28
- premature abstraction 216, 316, 317
- premature optimization 324
- Printf %x 56
- Printf %x 10, 55, 83
- production code 301
- profile
 - blocking 324
 - CPU 324
 - heap 324
- profiling 324
- programming language
 - Alef xiii
 - APL xiii
 - C++ xiv, xv, 361
 - C xii, xv, 1, 6, 52, 260, 361
 - Haskell xiv
 - Java xv
 - JavaScript xv, 107
 - Lisp 338
 - Modula-2 xiii
 - Oberon xiii
 - Pascal xiii
 - Python xv, 193
 - Ruby xv, 193
 - Scheme xiii
 - Squeak, Newsqueak xiii
- promotion, method 161
- protocol buffers 107
- Python programming language xv, 193
- qualified identifier 41, 43
- querying packages 298
- quote character, ' 56
- race
 - condition 21, 257, 258, 259
 - detector 271, 274
 - paradoxical 267
- randomized testing 307
- range loop 6, 9
- range over channel 229
- range over map 94
- range over string 69, 88
- {{range}} template action 113
- raw string literal 66
- reachability 36
- read, stale 268
- readers lock 266
- read/write mutex 266, 267
- ReadWriteCloser interface 174
- ReadWriter interface 174
- real built-in function 61
- receive
 - <-ch, channel 18, 225, 232
 - expression 225
 - non-blocking channel 246
 - ok value from channel 229
- receive-only channel type <-chan T 230
- receiver
- Printf %x 56
- Printf %x 10, 55, 83
- production code 301
- profile
 - blocking 324
 - CPU 324
 - heap 324
- profiling 324
- programming language
 - Alef xiii
 - APL xiii
 - C++ xiv, xv, 361
 - C xii, xv, 1, 6, 52, 260, 361
 - Haskell xiv
 - Java xv
 - JavaScript xv, 107
 - Lisp 338
 - Modula-2 xiii
 - Oberon xiii
 - Pascal xiii
 - Python xv, 193
 - Ruby xv, 193
 - Scheme xiii
 - Squeak, Newsqueak xiii
- promotion, method 161
- protocol buffers 107
- Python programming language xv, 193
- qualified identifier 41, 43
- querying packages 298
- quote character, ' 56
- race
 - condition 21, 257, 258, 259
 - detector 271, 274
 - paradoxical 267
- randomized testing 307
- range loop 6, 9
- range over channel 229
- range over map 94
- range over string 69, 88
- {{range}} template action 113
- raw string literal 66
- reachability 36
- read, stale 268
- readers lock 266
- read/write mutex 266, 267
- ReadWriteCloser interface 174
- ReadWriter interface 174
- real built-in function 61
- receive
 - <-ch, channel 18, 225, 232
 - expression 225
 - non-blocking channel 246
 - ok value from channel 229
- receive-only channel type <-chan T 230
- receiver

- name, method 157
- nil 159, 185
- parameter, method 156
- pointer 158, 167
- type, method 157
- recover built-in function 152
- recursion 121, 124, 247, 333, 339, 345, 359
- recursive
 - anonymous function 137
 - data structure 101, 102, 107
 - type 48
- redefining new 35
- reference
 - call by 83
 - identity 87
 - type 9, 12, 93, 120
- reflect package 330
- reflection 329, 352, 359
- reflect.StructTag type 348
- reflect.Type type 330
- reflect.Type.Field method 348
- reflect.TypeOf function 330
- reflect.Value type 331, 342
- reflect.Value zero value 332
- reflect.Value.Addr method 342
- reflect.Value.CanAddr method 342
- reflect.Value.Interface method 331, 342
- reflect.Value.Kind method 332
- reflect.ValueOf function 331, 337
- reflect.Zero function 345
- regexp package 149
- regexp.Compile function 149
- regexp.MustCompile function 149
- regular expression 66, 149, 305, 321
- relation, “happens before” 226, 257, 261, 277
- relationship, “has a” 162
- relationship, “is a” 162, 175
- remainder operator % 52, 166
- renaming import 286
- rendezvous 234
- replacement character `\u`, Unicode 70, 98
- repository, `gop1.io` xvi
- request
 - HTTP GET 21, 127, 272, 348
 - HTTP POST 348
 - multiplexer, HTTP 193
- result list, function 119
- result, named 120, 126
- return, bare 126
- return statement 29, 120, 125
- rev example 86
- reverb example 223, 224
- right shift operator `>>` 54
- Ruby programming language xv, 193
- rune literal 56
- rune type 52, 67
- rune slice to string conversion 71
- rune to string conversion 71
- runtime package 151
- runtime API 324
- runtime scheduler 281
- runtime.Stack function 151
- satisfaction, interface 171, 175
- Scalable Vector Graphics (SVG) 58
- scheduler, runtime 281
- Scheme programming language xiii
- scope
 - declaration 45, 137
 - for 47
 - if-else 47
 - label 46
 - local variable 135
 - loop variable 141, 236
 - pitfall 140
 - short variable declaration 22, 48
 - switch 47
- search algorithm, breadth-first 139, 239
- search algorithm, depth-first 136
- SearchIssues function 111
- select case 245
- select, default case in 246
- select, non-blocking 246
- select statement 244, 245
- select{} statement 245
- selective recovery 152
- selector, field 156
- selector, method 156
- semaphore, binary 262
- semaphore, counting 241
- semicolon 3, 6
- send `ch<-`, channel 18, 225, 232
- send statement 225
- send-only channel type `chan<- T` 230
- separate compilation 284
- sequence diagram, thumbnail 238
- sequential consistency 268, 269
- serial confinement 262
- server example 19, 21
- server
 - chat 253
 - concurrent clock 219
 - concurrent echo 222
- set, map as 96, 202
- setter method 169
- sexpr example 340
- S-expression
 - decode example 347
 - decoding 344
 - encoding 338
- sexpr.Marshal function 340
- sexpr.readList function 347
- sexpr.Unmarshal function 347
- SHA256 message digest 83
- sha256 example 83
- shadowing declaration 46, 49, 206, 212
- shared
 - lock 266
 - variables 257
 - variables, concurrency with 257
- shared-memory multithreading 217, 257
- shift operator `<<`, left 54
- shift operator `>>`, right 54
- short
 - variable declaration 5, 7, 30, 31
 - variable declaration scope 22, 48
 - variable declaration statement 7
- short-circuit
 - AND operator `&&` 63
 - evaluation 63
 - OR operator `||` 63
- signature, function 120
- signed integer 52, 54
- `s[i:j]`, slice operator 84, 86
- `s[i:j]`, substring operator 65, 86
- simple statement 6, 22
- Sizeof table 354
- sleep example 179
- slice 4
 - argument 86
 - capacity 88, 89
 - capacity growth diagram 90
 - comparison 87
 - extending a 86
 - key, map with 97
 - literal 38, 86
 - make 88, 322
 - nil 87
 - of months diagram 84
 - operator `s[i:j]` 84, 86
 - rotation algorithm 86
 - techniques, in-place 91
 - type 84
 - used as stack 123
 - zero length 87
 - zero value 74, 87
- SMTP client 312
- socket
 - TCP 219
 - UDP 219
 - Unix domain 219
- sort algorithm, topological 136
- sort package 95, 186, 189
- sort.Float64s function 191
- sort.Interface interface 186
- sort.Ints function 191
- sort.IntsAreSorted function 191

- sort.IntSlice type 191
- sort.Reverse function 189
- sort.Strings function 95, 137, 191
- spinner example 218
- Sprint function 330
- SQL API 211
- SQL injection attack 211
- sqlQuote function 211, 212
- squares example 135
- Squeak, Newsqueak programming language xiii
- stack
 - allocation 36
 - fixed-size 124
 - growth 124, 280, 358
 - implementation with slice 92, 215
 - overflow 124
 - slice used as 123
 - trace 149, 253
 - variable 36
 - variable-size 124
- stale read 268
- standard
 - IEEE 754 56, 57
 - POSIX xi, 55, 72, 197
 - Unicode 2, 27, 52, 66, 67, 69, 97
- statement
 - , decrement 5, 37
 - ++, increment 5, 37, 94
 - assignment 5, 7, 36, 52, 94, 173
 - break 24, 46
 - continue 24, 46
 - coverage 318, 320
 - defer 144, 150, 264
 - fallthrough 23, 212
 - for 6
 - go 18, 218
 - goto 24
 - if-else 9, 22, 47
 - label 46
 - labeled 46
 - return 29, 120, 125
 - select{} 245
 - select 244, 245
 - send 225
 - short variable declaration 7
 - simple 6, 22
 - switch 23, 47
 - tagless switch 24
 - type switch 210, 212, 214, 329
 - unreachable 120
- storage example package 312, 313
- Strangelove, Dr. 336
- strategies, error-handling 128, 152, 310, 316
- strconv package 22, 71, 75
- strconv.Atoi function 22, 75
- strconv.FormatInt function 75
- strconv.Itoa function 75
- strconv.ParseInt function 75
- strconv.ParseUint function 75
- stream, io.Discard 22
- String method 40, 166, 329
- string
 - concatenation operator + 5, 65
 - conversion 71
 - immutability 65, 73
 - index operation 64
 - literal 65
 - literal, raw 66
 - range over 69, 88
 - sharing diagram 65
 - test 71
 - to byte slice conversion 40, 73
 - to rune slice conversion 71, 88
 - zero value 5, 7, 30
 - comparison 65
- strings package 7, 71, 72, 289
- strings.Contains function 69
- strings.HasPrefix function 69
- strings.HasSuffix function 69
- strings.Index function 289
- strings.Join function 7, 12
- strings.Map function 133
- strings.NewReader function 289
- strings.NewReplacer function 289
- strings.Reader type 289
- strings.Replacer type 289
- strings.Split function 12
- strings.ToLower function 72
- strings.ToUpper function 72
- struct declaration 99
- struct
 - comparison 104
 - embedding 104, 161
 - Employee 100
 - empty 102
 - field 15, 99
 - field, anonymous 104, 105, 106, 162
 - field, embedded 161
 - field, export of 101, 106, 109, 110, 168
 - field order 101, 355
 - field tag 109, 348
 - hole 354
 - hole diagram 355
 - literal 15, 102, 106
 - literal, address of 103
 - pointer to 100, 103
 - type 15, 24, 99
- struct{} type 227, 241, 250
- struct type, unnamed 163
- struct zero value 102
- substitutability 193
- substring operator s[i:j] 65, 86
- subtype polymorphism 211
- sum example 142
- surface example 59, 203
- surface figure, 3-D 58, 203
- SVG 58
- SWIG 361
- Swiss army knife 290
- switch, default case in 23
- switch, initialization statement in 24
- switch scope 47
- switch statement 23, 47
- switch statement, tagless 24
- switch statement, type 210, 212, 214, 329
- switch, context 280
- sync package 237, 263
- synchronous channel 226
- sync.Mutex type 263, 269
- (*sync.Mutex).Lock method 21, 146, 263
- (*sync.Mutex).Unlock method 21, 146, 263
- sync.Once type 270
- (*sync.Once).Do method 270
- sync.RWMutex type 266, 270
- (*sync.RWMutex).RLock method 266
- (*sync.RWMutex).RUnlock method 266
- sync.WaitGroup type 237, 250, 274
- (*sync.WaitGroup).Add method 238
- (*sync.WaitGroup).Done method 238
- syscall package 196, 208
- syscall.Errno type 196, 197
- system call API 196
- table of
 - binary operators 52
 - bitwise operators 53
 - comparison operators 53
 - escape sequences 66
 - keywords 27
 - predeclared names 28
 - Printf verbs 10
 - UTF-8 encodings 67
- table, Sizeof 354
- table-driven testing 200, 306, 319
- tag, struct field 109, 348
- tagless switch statement 24
- tags, build 296
- TCP socket 219
- techniques, in-place slice 91
- tempconv example 39, 180, 289
- tempconv example package 42
- temperature conversion example 29
- tempflag example 181
- template API 115
- template

- | in 113
- action, {{range}} 113
- dot . in 113
- template.Funcs method 114
- template.HTML type 116
- template.Must function 114
- template.New function 114
- template.Parse method 114
- Test function 302
- test
 - black-box 310
 - brittle 317
 - character 71
 - coverage 318
 - dependency, cyclic 314
 - echo 309
 - failure message 306
 - integration 314
 - of word example 303
 - package, external 285, 314
 - string 71
 - white-box 311
 - assertion 306
- _test.go file 285, 302, 303
- testing package 285, 302
- testing
 - a command 308
 - randomized 307
 - table-driven 200, 306, 319
- testing.B type 321
- testing.T type 302
- (*testing.T).Errorf method 200, 304, 306
- (*testing.T).Fatal method 306
- tests, writing effective 316, 317
- text/scanner package 344
- text/tabwriter package 188
- text/template package 113, 300
- Thompson, Ken xi, 67
- thread 218, 280
- thread-local storage 282
- 3-D surface figure 58, 203
- thumbnail example 236, 237, 238
- thumbnail example package 235
- thumbnail sequence diagram 238
- time package 18, 77, 183
- time.After function 245
- time.AfterFunc function 164
- time.Duration type 76, 179
- time.Minute constant 76
- time.Now function 220
- time.Parse function 220
- time.Second constant 164
- time.Since function 114
- time.Tick function 244, 246
- time.Time type 114
- time.Time.Format method 220
- title example 153
- title function 144, 145
- token-based decoder API 213, 215, 347
- token-based XML decoding 213
- tool
 - cgo 361, 362
 - go 2, 42, 44, 290
 - go doc 25
 - godoc xvi, 25, 297, 326
 - gofmt 3, 4, 44, 286
 - goimports 3, 44, 286
 - golint 292
- topological sort algorithm 136
- topoSort example 136
- trace example 146
- trace, stack 149, 253
- tree, binary 102
- treeSort example 102
- true boolean constant 63
- truncation, floating-point 40, 55
- tuple assignment 31, 37
- type declaration 39
- type keyword 212
- type
 - abstract 24, 171
 - aggregate 81, 99
 - array 81
 - assertion 205, 211
 - assertion, interface 208, 210
 - assertion, ok value from 206
 - bool 63
 - bufio.Scanner 9
 - byte 52
 - bytes.Buffer 74, 169, 172, 185
 - Celsius 39
 - chan 225
 - channel 18
 - <-chan T, receive-only channel 230
 - chan<- T, send-only channel 230
 - complex 61
 - composite xv, 14, 81
 - composition xv, 107, 162, 189
 - concrete 24, 171, 211, 214
 - displaying methods of a 351
 - empty interface 176
 - error built-in 11, 128, 149, 196
 - Fahrenheit 39
 - function 119, 120
 - http.HandlerFunc 194, 203
 - http.Request 21, 253
 - http.ResponseWriter 19, 22, 191, 193
 - int 52
 - interface{} 143, 176, 331
 - interface 171, 174
 - interface dynamic 181
 - json.Decoder 111
 - json.Encoder 111
 - map 9, 93
 - method receiver 157
 - mismatch 55
 - named 24, 39, 40, 105, 157
 - net.Conn 220
 - net.Listener 220
 - numeric 51
 - *os.File 11, 13, 172, 175, 185, 336
 - os.FileInfo 247
 - os.LinkError 207
 - os.PathError 207
 - recursive 48
 - reference 9, 12, 93, 120
 - reflect.StructTag 348
 - reflect.Type 330
 - reflect.Value 331, 342
 - rune 52, 67
 - slice 84
 - sort.IntSlice 191
 - strings.Reader 289
 - strings.Replacer 289
 - struct{} 227, 241, 250
 - struct 15, 24, 99
 - switch, case in 212
 - switch, default case in 212
 - switch statement 210, 212, 214, 329
 - sync.Mutex 263, 269
 - sync.Once 270
 - sync.RWMutex 266, 270
 - sync.WaitGroup 237, 250, 274
 - syscall.Errno 196, 197
 - template.HTML 116
 - testing.B 321
 - testing.T 302
 - time.Duration 76, 179
 - time.Time 114
 - uint 52
 - uintptr 52, 354, 357
 - underlying 39
 - unidirectional channel 230, 231
 - unnamed struct 163
 - unsafe.Pointer 356
 - url.URL 193
- types, untyped constant 78
- UDP socket 219
- uint type 52
- uintptr type 52, 354, 357
- unary operator + 53
- unary operator - 53
- unbuffered channel 226
- undefined behavior 260
- underlying array 84, 88, 91, 187
- underlying type 39
- Unicode
 - code point 67
 - escape 68, 107
 - replacement character 70, 98

- standard 2, 27, 52, 66, 67, 69, 97
- unicode package 71
- unicode.IsDigit function 71
- unicode.IsLetter function 71
- unicode.IsLower function 71
- unicode.IsSpace function 93
- unicode.IsUpper function 71
- unicode/utf8 package 69
- unidirectional channel type 230, 231
- union, discriminated 211, 213, 214
- universe block 46
- Unix domain socket 219
- unmarshaling JSON 110
- unnamed struct type 163
- unnamed variable 34, 88
- unreachable statement 120
- unsafe package 354
- unsafe.AlignOf function 355
- unsafe.Offsetof function 355
- unsafe.Pointer conversion 356
- unsafe.Pointer type 356
- unsafe.Pointer zero value 356
- unsafe.Sizeof function 354
- unsigned integer 52, 54
- untyped constant types 78
- unused parameter 120
- URL 123
- URL escape 111
- url.QueryEscape function 111
- url.URL type 193
- url.Values example 160
- UTF-8 66, 67, 98
- UTF-8 encodings, table of 67
- utf8.DecodeRuneInString function 69
- utf8.RuneCountInString function 69
- utf8.UTFMax value 98
- value
 - addressable 32
 - call by 83, 120, 158
 - function 132
 - interface 181
 - method 164
 - utf8.UTFMax 98
- var declaration 5, 30
- variable
 - confinement 261
 - heap 36
 - http.DefaultClient 253
 - io.Discard 18
 - io.EOF 132
 - lifetime 35, 46, 135
 - local 29, 141
 - os.Args 4
 - stack 36
 - unnamed 34, 88
- variables, escaping 36
- variables, shared 257
- variable-size stack 124
- variadic function 142, 172
- vector, bit 165
- vendoring 293
- visibility 28, 29, 41, 168, 297
- visit function 122
- wait example 130
- WaitForServer function 130
- walkDir function 247
- web
 - crawler 119
 - crawler, concurrent 239
 - framework 193
- while loop 6
- white-box test 311
- Wilkes, Maurice 301
- Wirth, Niklaus xiii
- word example 303, 305, 308
- word example, test of 303
- workspace organization 291
- writer lock 266
- writing effective tests 316, 317
- xkcd JSON interface 113
 - XML decoding 213
 - XML (Extensible Markup Language) 107
 - (*xml.Decoder).Token method 213
 - xmlselect example 215
- zero length slice 87
- zero value
 - array 82
 - boolean 30
 - channel 225, 246
 - function 132
 - interface 182
 - map 95
 - named result 120, 127
 - number 5, 30
 - pointer 32
 - reflect.Value 332
 - slice 74, 87
 - string 5, 7, 30
 - struct 102
 - unsafe.Pointer 356