O'REILLY®

# Head First GO

A Brain-Friendly Guide

Grayscale Edition

For Sale in the Indian Subcontinent & Select Countries Only*

*Refer Back Cover

A Learner's Guide to Go Programming

Learn to write simple, maintainable code

Avoid embarrassing type errors

Focus on the features that will make you most productive

Bend your mind around more than 40 Go exercises

Run functions concurrently with goroutines

SPD®

Jay McGavren

# Head First Go

Wouldn't it be dreamy if there were a book on Go that focused on the things you **need** to know? I guess it's just a fantasy...

Jay McGavren

Beijing • Boston • Farnham • Sebastopol • Tokyo

SHROFF PUBLISHERS & DISTRIBUTORS PVT. LTD.

Mumbai Bangalore Kolkata New Delhi

**Head First Go**
by Jay McGavren

Copyright © 2019 Jay McGavren. All rights reserved. ISBN: 978-1-491-96955-7
Originally printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles *(safari.oreilly.com)*. For more information, contact our corporate/institutional sales department: (800)998-9938 or *corporate@oreilly.com*.

| | |
|---|---|
| **Series Creators:** | Kathy Sierra, Bert Bates |
| **Editor:** | Jeff Bleiel |
| **Cover Designer:** | Randy Comer |
| **Production Editor:** | Kristen Brown |
| **Production Services:** | Rachel Monaghan |
| **Indexer:** | Lucie Haskins |
| **Brain image on spine:** | Eric Freeman |

**Printing History:**
April 2019: First Edition.

**First Indian Reprint:** April 2019

ISBN: 978-93-5213-825-8

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. The *Head First* series designations, *Head First Go*, and related trade dress are trademarks of O'Reilly Media, Inc.

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 O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Code for this book was developed using 100% recycled electrons.

**For sale in the Indian Subcontinent (India, Pakistan, Bangladesh, Sri Lanka, Nepal, Bhutan, Maldives) and African Continent (excluding Morocco, Algeria, Tunisia, Libya, Egypt, and the Republic of South Africa) only. Illegal for sale outside of these countries.**

Authorized reprint of the original work published by O'Reilly Media, Inc. All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, nor exported to any countries other than ones mentioned above without the written permission of the copyright owner.

Published by **Shroff Publishers & Distributors Pvt. Ltd.** B-103, Railway Commercial Complex, Sector 3, Sanpada (E), Navi Mumbai 400705 • TEL: (91 22) 4158 4158 • FAX: (91 22) 4158 4141 E-mail:spdorders@shroffpublishers.com•Web:www.shroff-publishers.com Printed at Jasmine Art Printers Pvt. Ltd., Navi Mumbai.

To my eternally patient Christine.

## Author of Head First Go

Jay McGavren

**Jay McGavren** is the author of *Head First Ruby* and *Head First Go*, both published by O'Reilly. He also teaches software development at Treehouse.

His home in the Phoenix suburbs houses himself, his lovely wife, and an alarmingly variable number of kids and dogs.

You can visit Jay's personal website at *http://jay.mcgavren.com*.

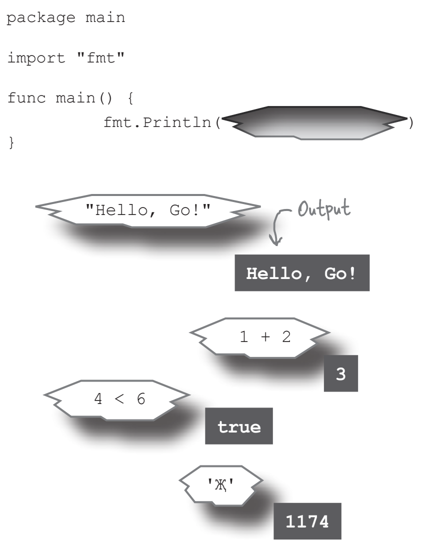
## Table of Contents (Summary)

## Table of Contents (the real thing)

# 1 let's get going
## Syntax Basics

**Are you ready to turbo-charge your software?** Do you want a **simple** programming language that **compiles fast**? That **runs fast**? That makes it **easy to distribute** your work to users? Then **you're ready for Go**!

Go is a programming language that focuses on **simplicity** and **speed**. It's simpler than other languages, so it's quicker to learn. And it lets you harness the power of today's multicore computer processors, so your programs run faster. This chapter will show you all the Go features that will make **your life as a developer easier**, and make your **users happier**.



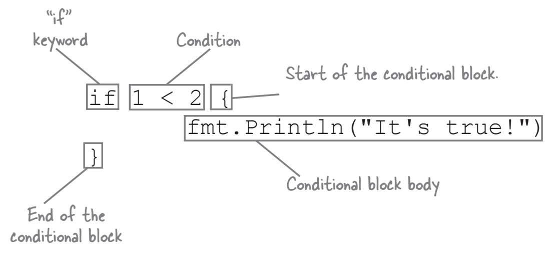
```
package main

import "fmt"

func main() {
        fmt.Println(                    )
}
```

# 2 which code runs next?

## Conditionals and Loops

**Every program has parts that apply only in certain situations.** “This code should run *if* there’s an error. Otherwise, that other code should run.” Almost every program contains code that should be run only when a certain *condition* is true. So almost every programming language provides **conditional statements** that let you determine whether to run segments of code. Go is no exception.

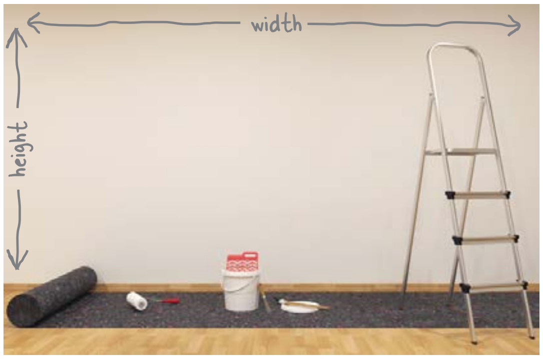
You may also need some parts of your code to run *repeatedly*. Like most languages, Go provides **loops** that run sections of code more than once. We’ll learn to use both conditionals and loops in this chapter!



```
if 1 < 2 {
    fmt.Println("It's true!")
}
```

# 3 call me Functions

**You've been missing out.** You've been calling functions like a pro. But the only functions you could call were the ones Go defined for you. Now, it's your turn. We're going to show you how to create your own functions. We'll learn how to declare functions with and without parameters. We'll declare functions that return a single value, and we'll learn how to return multiple values so that we can indicate when there's been an error. And we'll learn about **pointers**, which allow us to make more memory-efficient function calls.

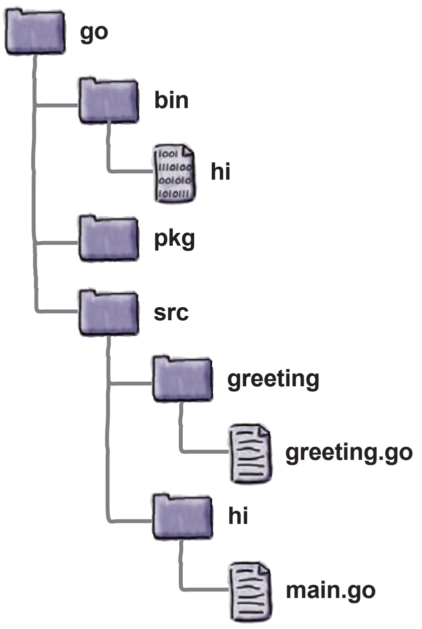


bundles of code

# 4 Packages

**It's time to get organized.** So far, we've been throwing all our code together in a single file. As our programs grow bigger and more complex, that's going to quickly become a mess.

In this chapter, we'll show you how to create your own **packages** to help keep related code together in one place. But packages are good for more than just organization. Packages are an easy way to *share code between your programs*. And they're an easy way to *share code with other developers*.

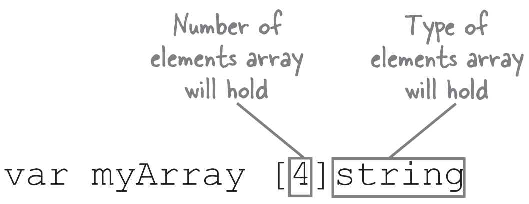


5

on the list

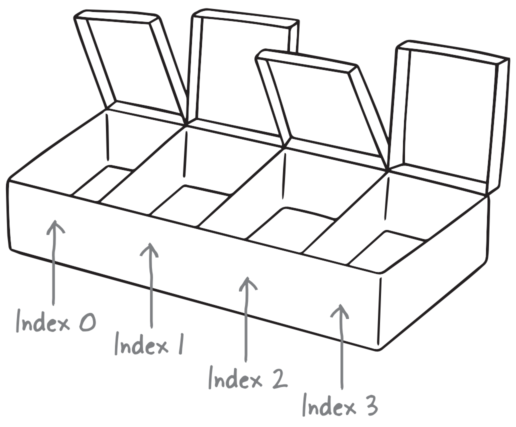
## Arrays

**A whole lot of programs deal with lists of things.** Lists of addresses. Lists of phone numbers. Lists of products. Go has *two* built-in ways of storing lists. This chapter will introduce the first: **arrays**. You'll learn about how to create arrays, how to fill them with data, and how to get that data back out again. Then you'll learn about processing all the elements in array, first the *hard* way with `for` loops, and then the *easy* way with `for...range` loops.



```
var myArray [4]string
```

Number of elements array will hold
Type of elements array will hold

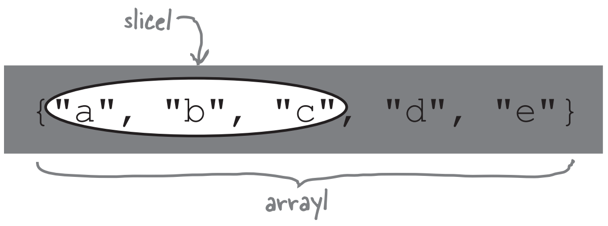


6

appending issue

# Slices

**We've learned we can't add more elements to an array.** That's a real problem for our program, because we don't know in advance how many pieces of data our file contains. But that's where Go **slices** come in. Slices are a collection type that can grow to hold additional items—just the thing to fix our current program! We'll also see how slices give users an easier way to provide data to *all* your programs, and how they can help you write functions that are more convenient to call.



labeling data

# 7 Maps

**Throwing things in piles is fine, until you need to find something again.** You've already seen how to create lists of values using *arrays* and *slices*. You've seen how to apply the same operation to *every value* in an array or slice. But what if you need to work with a *particular* value? To find it, you'll have to start at the beginning of the array or slice, and *look through Every. Single. Value.*

What if there were a kind of collection where every value had a label on it? You could quickly find just the value you needed! In this chapter, we'll look at **maps**, which do just that.

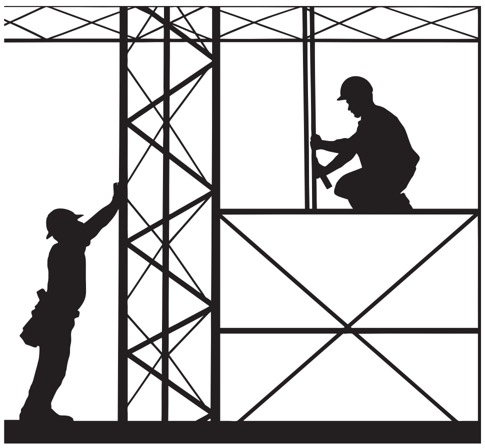
Keys let you quickly find data again!

building storage

# 8 Structs

**Sometimes you need to store more than one type of data.**

We learned about slices, which store a list of values. Then we learned about maps, which map a list of keys to a list of values. But both of these data structures can only hold values of *one* type. Sometimes, you need to group together values of *several* types. Think of mailing addresses, where you have to mix street names (strings) with postal codes (integers). Or student records, where you have to mix student names (strings) with grade point averages (floating-point numbers). You can't mix value types in slices or maps. But you *can* if you use another type called a **struct**. We'll learn all about structs in this chapter!



# 9 you're my type: Defined Types

**There's more to learn about defined types.** In the previous chapter, we showed you how to define a type with a struct underlying type. What we *didn't* show you was that you can use *any* type as an underlying type.

And do you remember methods—the special kind of function that's associated with values of a particular type? We've been calling methods on various values throughout the book, but we haven't shown you how to define your *own* methods. In this chapter, we're going to fix all of that. Let's get started!

How much Steve thought he bought
10 gallons
How much Steve actually bought!
10 liters

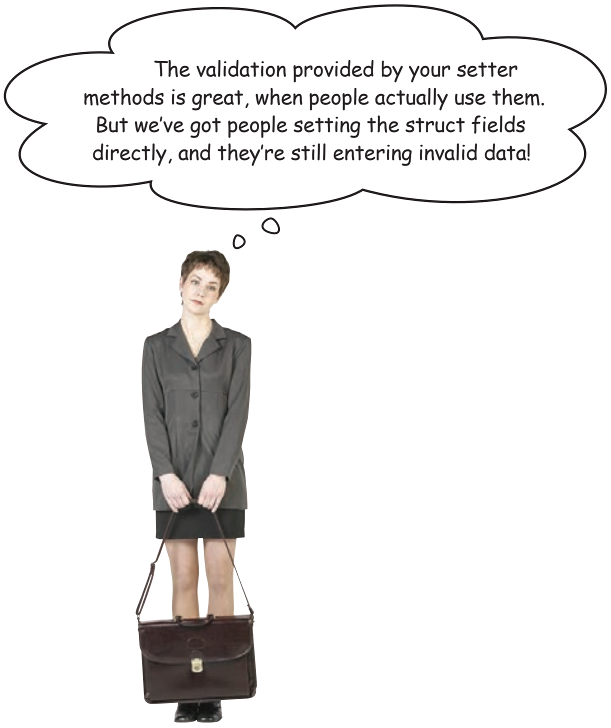
10

keep it to yourself

# Encapsulation and Embedding

**Mistakes happen.** Sometimes, your program will receive invalid data from user input, a file you're reading in, or elsewhere. In this chapter, you'll learn about **encapsulation**: a way to protect your struct type's fields from that invalid data. That way, you'll know your field data is safe to work with!

We'll also show you how to **embed** other types within your struct type. If your struct type needs methods that already exist on another type, you don't have to copy and paste the method code. You can embed the other type within your struct type, and then use the embedded type's methods just as if they were defined on your own type!


The validation provided by your setter methods is great, when people actually use them. But we've got people setting the struct fields directly, and they're still entering invalid data!

what can you do?

## 11 Interfaces

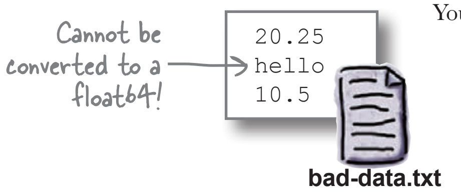
**Sometimes you don't care about the particular type of a value.** You don't care about what it *is*. You just need to know that it will be able to *do* certain things. That you'll be able to call *certain methods* on it. You don't care whether you have a `Pen` or a `Pencil`, you just need something with a `Draw` method. You don't care whether you have a `Car` or a `Boat`, you just need something with a `Steer` method.

That's what Go **interfaces** accomplish. They let you define variables and function parameters that will hold *any* type, as long as that type defines certain methods.

# 12 back on your feet
# Recovering from Failure

**Every program encounters errors. You should plan for them.**

Sometimes handling an error can be as simple as reporting it and exiting the program. But other errors may require additional action. You may need to close opened files or network connections, or otherwise clean up, so your program doesn't leave a mess behind. In this chapter, we'll show you how to **defer** cleanup actions so they happen even when there's an error. We'll also show you how to make your program **panic** in those (rare) situations where it's appropriate, and how to **recover** afterward.



13

# sharing work

## Goroutines and Channels

**Working on one thing at a time isn't always the fastest way to finish a task.** Some big problems can be broken into smaller tasks. **Goroutines** let your program work on several different tasks at once. Your goroutines can coordinate their work using **channels**, which let them send data to each other *and* synchronize so that one goroutine doesn't get ahead of another. Goroutines let you take full advantage of computers with multiple processors, so that your programs run as fast as possible!

A receiving goroutine waits until another goroutine sends a value.

14

code quality assurance

# Automated Testing

**Are you sure your software is working right now? Really sure?** Before you sent that new version to your users, you presumably tried out the new features to ensure they all worked. But did you try the *old* features to ensure you didn't break any of them? *All* the old features? If that question makes you worry, your program needs **automated testing**. Automated tests ensure your program's components work correctly, even after you change your code. Go's `testing` package and `go test` tool make it easy to write automated tests, using the skills that you've already learned!

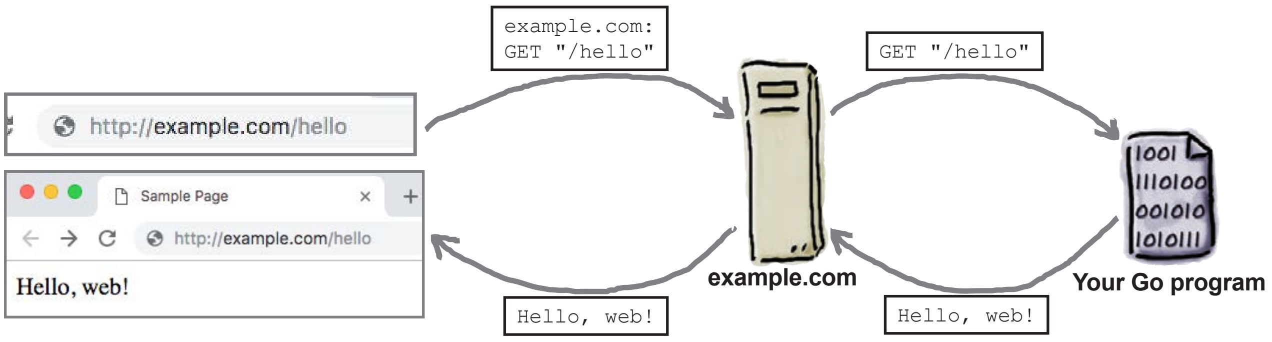
✓ For `[]slice{"apple", "orange", "pear"}`, `JoinWithCommas` should return `"apple, orange, and pear"`.

✗ For `[]slice{"apple", "orange"}`, `JoinWithCommas` should return `"apple and orange"`.

# 15 responding to requests
# Web Apps

**This is the 21st century. Users want web apps.** Go's got you covered there, too! The Go standard library includes packages to help you host your own web applications and make them accessible from any web browser. So we're going to spend the final two chapters of the book showing you how to build web apps.

The first thing your web app needs is the ability to respond when a browser sends it a request. In this chapter, we'll learn to use the `net/http` package to do just that.

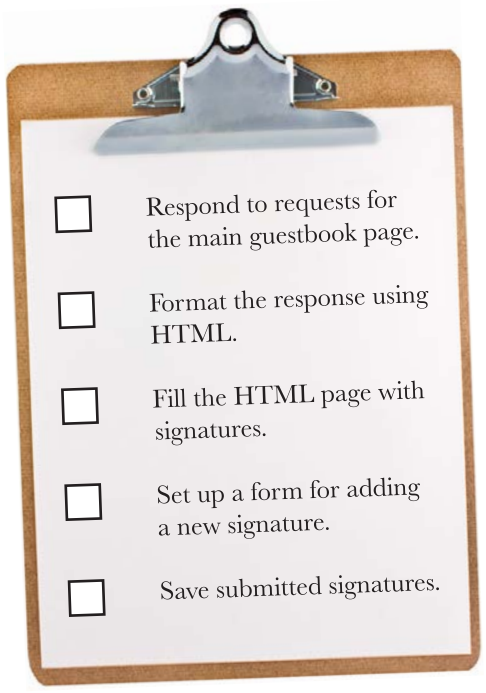


16

# a pattern to follow
# HTML Templates

**Your web app needs to respond with HTML, not plain text.** Plain text is fine for emails and social media posts. But your pages need to be formatted. They need headings and paragraphs. They need forms where your users can submit data to your app. To do any of that, you need HTML code.

And eventually, you'll need to insert data into that HTML code. That's why Go offers the `html/template` package, a powerful way to include data in your app's HTML responses. Templates are key to building bigger, better web apps, and in this final chapter, we'll show you how to use them!


☐ Respond to requests for the main guestbook page.
☐ Format the response using HTML.
☐ Fill the HTML page with signatures.
☐ Set up a form for adding a new signature.
☐ Save submitted signatures.

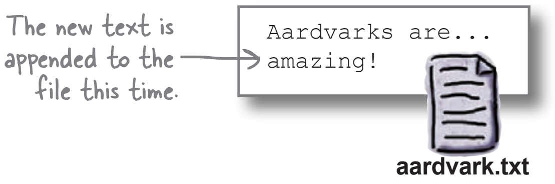
understanding os.openfile

# Appendix A: Opening Files

A

**Some programs need to write data to files, not just read data.** Throughout the book, when we've wanted to work with files, you had to create them in your text editor for your programs to read. But some programs *generate* data, and when they do, they need to be able to *write* data to a file.

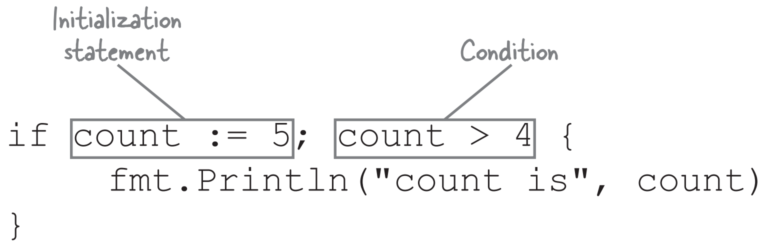
We used the `os.OpenFile` function to open a file for writing earlier in the book. But we didn't have space then to fully explore how it worked. In this appendix, we'll show you everything you need to know in order to use `os.OpenFile` effectively!


```
Aardvarks are...
amazing!
```

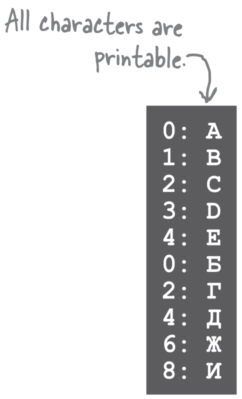
six things we didn't cover

# B Appendix B: Leftovers

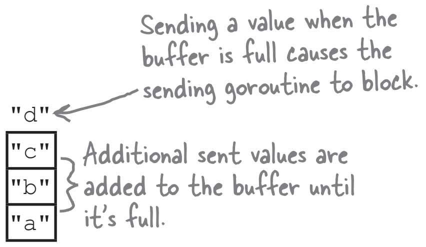
**We've covered a lot of ground, and you're almost finished with this book.** We'll miss you, but before we let you go, we wouldn't feel right about sending you out into the world without a *little* more preparation. We've saved six important topics for this appendix.



```
if count := 5; count > 4 {
        fmt.Println("count is", count)
}
```



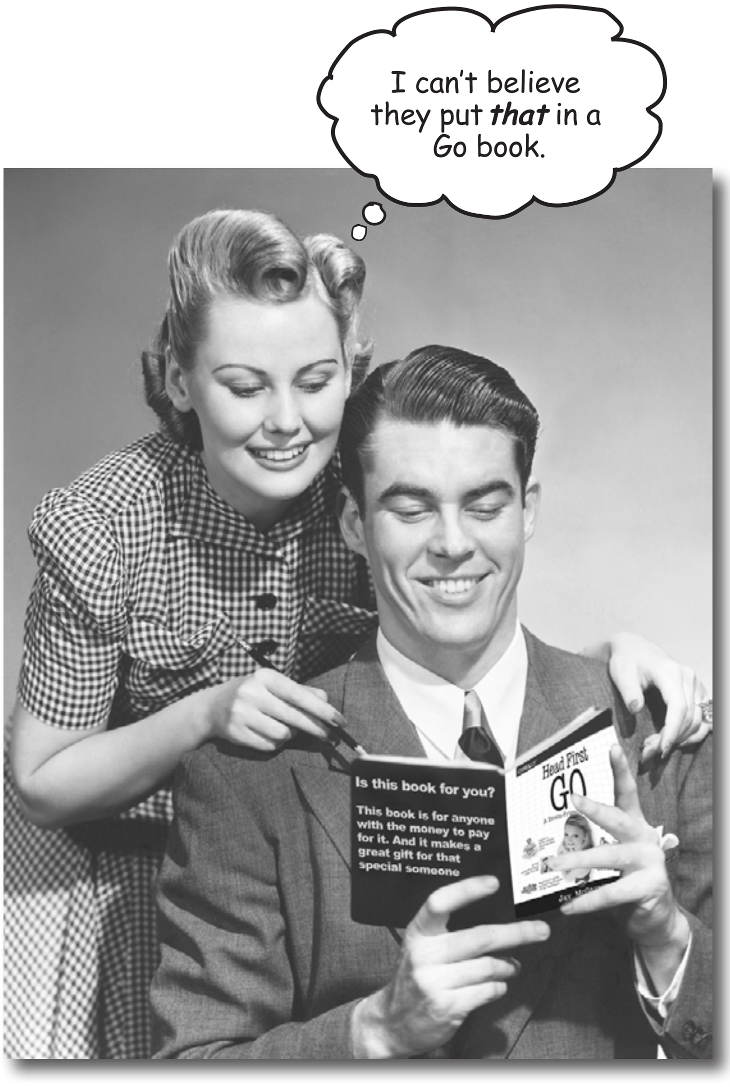
```
0: A
1: B
2: C
3: D
4: E
0: Б
2: Г
4: Д
6: Ж
8: И
```



Sending a value when the buffer is full causes the sending goroutine to block.
"d"
"c"
"b"
"a"
Additional sent values are added to the buffer until it's full.

how to use this book

# Intro


I can't believe they put *that* in a Go book.
Is this book for you?
This book is for anyone with the money to pay for it. And it makes a great gift for that special someone
Head First GO

In this section, we answer the burning question: "So why DID they put that in a book on Go?"

## Who is this book for?

If you can answer "yes" to ***all*** of these:

1. Do you have access to a computer with a text editor?
2. Do you want to learn a programming language that makes development **fast** and **productive**?
3. Do you prefer **stimulating dinner-party conversation** to **dry, dull, academic lectures**?

this book is for you.

## Who should probably back away from this book?

If you can answer "yes" to any ***one*** of these:

1. **Are you completely new to computers?**

   (You don't need to be advanced, but you should understand folders and files, how to open a terminal app, and how to use a simple text editor.)

2. Are you a ninja rockstar developer looking for a ***reference*** **book**?
3. Are you **afraid to try something new**? Would you rather have a root canal than mix stripes with plaid? Do you believe that a technical book can't be serious if it's full of bad puns?

this book is *not* for you.

[Note from Marketing: this book is for anyone with a valid credit card.]

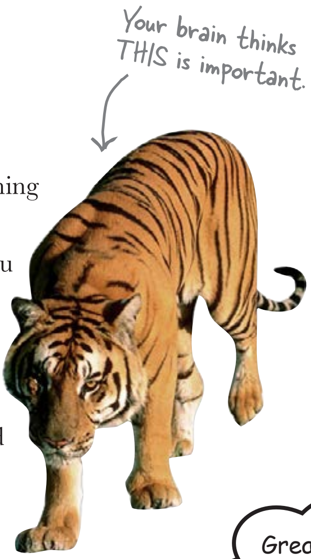
## We know what you're thinking

"How can *this* be a serious book on developing in Go?"

"What's with all the graphics?"

"Can I actually *learn* it this way?"

## We know what your *brain* is thinking



Your brain craves novelty. It's always searching, scanning, *waiting* for something unusual. It was built that way, and it helps you stay alive.

So what does your brain do with all the routine, ordinary, normal things you encounter? Everything it *can* to stop them from interfering with the brain's *real* job—recording things that *matter*. It doesn't bother saving the boring things; they never make it past the "this is obviously not important" filter.

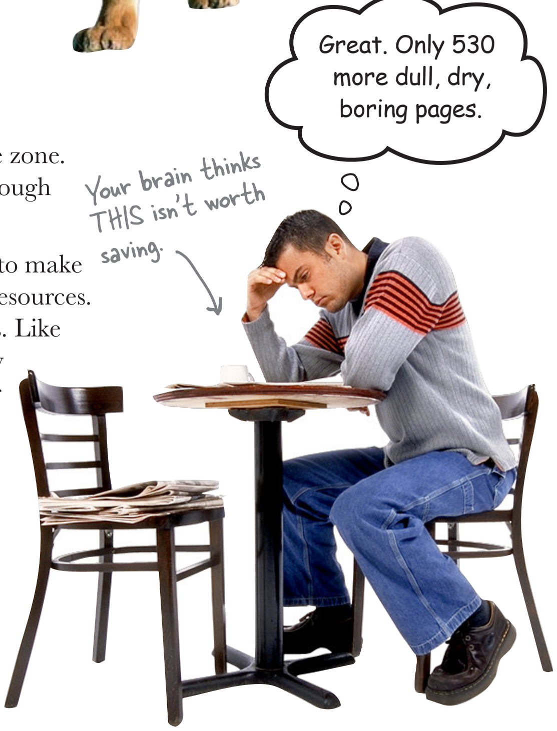
How does your brain *know* what's important? Suppose you're out for a day hike and a tiger jumps in front of you—what happens inside your head and body?

Neurons fire. Emotions crank up. *Chemicals surge*.

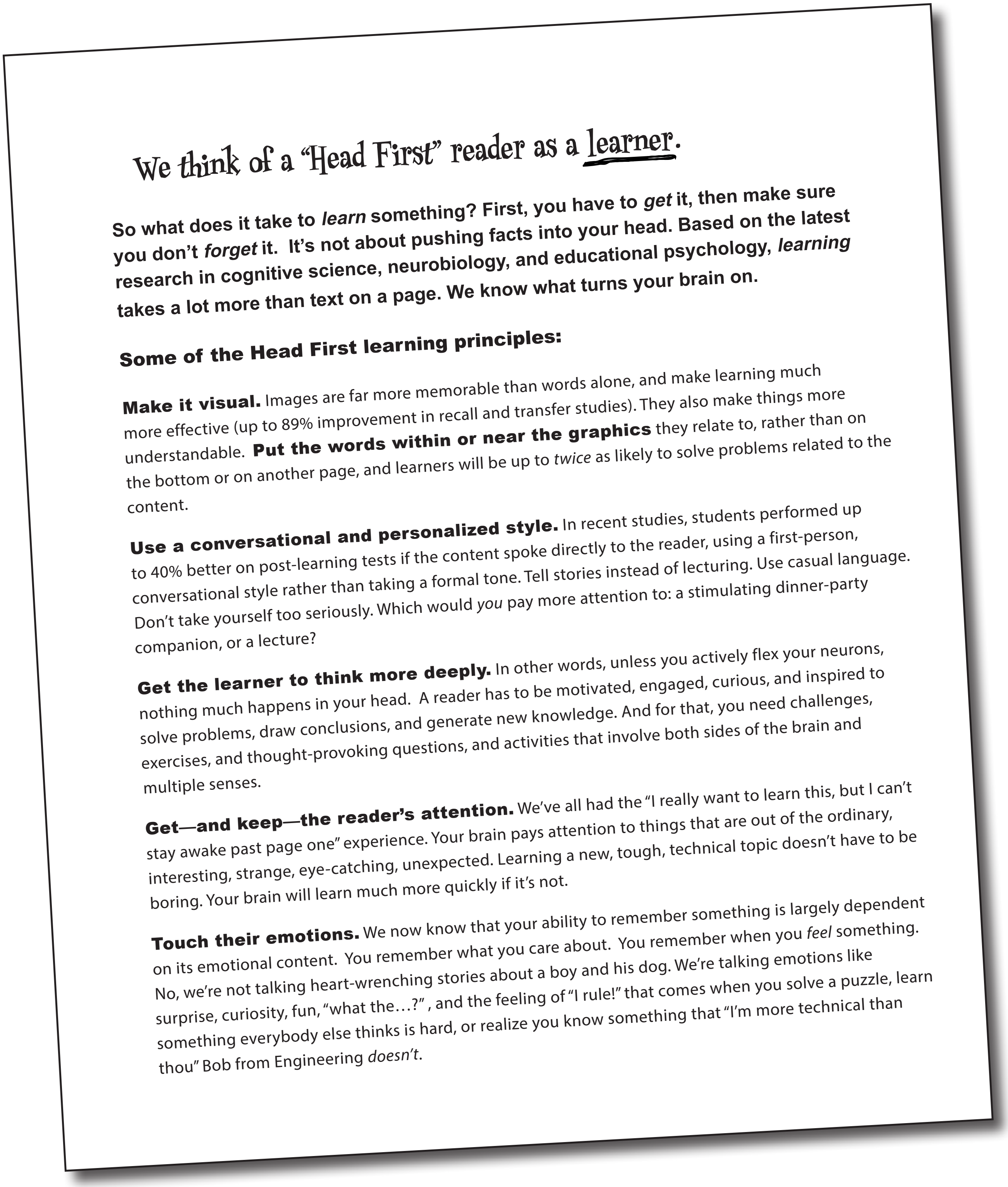
And that's how your brain knows…

**This must be important! Don't forget it!**

But imagine you're at home or in a library. It's a safe, warm, tiger-free zone. You're studying. Getting ready for an exam. Or trying to learn some tough technical topic your boss thinks will take a week, 10 days at the most.


Great. Only 530 more dull, dry, boring pages.
Your brain thinks THIS isn't worth saving.

Just one problem. Your brain's trying to do you a big favor. It's trying to make sure that this *obviously* unimportant content doesn't clutter up scarce resources. Resources that are better spent storing the really *big* things. Like tigers. Like the danger of fire. Like how you should never have posted those party photos on your Facebook page. And there's no simple way to tell your brain, "Hey, brain, thank you very much, but no matter how dull this book is, no matter how little I'm registering on the emotional Richter scale right now, I really *do* want you to keep this stuff around."



## We think of a "Head First" reader as a learner.

**So what does it take to *learn* something? First, you have to *get* it, then make sure you don't *forget* it. It's not about pushing facts into your head. Based on the latest research in cognitive science, neurobiology, and educational psychology, *learning* takes a lot more than text on a page. We know what turns your brain on.**

### Some of the Head First learning principles:

**Make it visual.** Images are far more memorable than words alone, and make learning much more effective (up to 89% improvement in recall and transfer studies). They also make things more understandable. **Put the words within or near the graphics** they relate to, rather than on the bottom or on another page, and learners will be up to *twice* as likely to solve problems related to the content.

**Use a conversational and personalized style.** In recent studies, students performed up to 40% better on post-learning tests if the content spoke directly to the reader, using a first-person, conversational style rather than taking a formal tone. Tell stories instead of lecturing. Use casual language. Don't take yourself too seriously. Which would *you* pay more attention to: a stimulating dinner-party companion, or a lecture?

**Get the learner to think more deeply.** In other words, unless you actively flex your neurons, nothing much happens in your head. A reader has to be motivated, engaged, curious, and inspired to solve problems, draw conclusions, and generate new knowledge. And for that, you need challenges, exercises, and thought-provoking questions, and activities that involve both sides of the brain and multiple senses.

**Get—and keep—the reader's attention.** We've all had the "I really want to learn this, but I can't stay awake past page one" experience. Your brain pays attention to things that are out of the ordinary, interesting, strange, eye-catching, unexpected. Learning a new, tough, technical topic doesn't have to be boring. Your brain will learn much more quickly if it's not.

**Touch their emotions.** We now know that your ability to remember something is largely dependent on its emotional content. You remember what you care about. You remember when you *feel* something. No, we're not talking heart-wrenching stories about a boy and his dog. We're talking emotions like surprise, curiosity, fun, "what the…?" , and the feeling of "I rule!" that comes when you solve a puzzle, learn something everybody else thinks is hard, or realize you know something that "I'm more technical than thou" Bob from Engineering *doesn't*.

## Metacognition: thinking about thinking

I wonder how I can trick my brain into remembering this stuff…

If you really want to learn, and you want to learn more quickly and more deeply, pay attention to how you pay attention. Think about how you think. Learn how you learn.

Most of us did not take courses on metacognition or learning theory when we were growing up. We were *expected* to learn, but rarely *taught* to learn.

But we assume that if you're holding this book, you really want to learn how to write Go programs. And you probably don't want to spend a lot of time. If you want to use what you read in this book, you need to *remember* what you read. And for that, you've got to *understand* it. To get the most from this book, or *any* book or learning experience, take responsibility for your brain. Your brain on *this* content.

The trick is to get your brain to see the new material you're learning as Really Important. Crucial to your well-being. As important as a tiger. Otherwise, you're in for a constant battle, with your brain doing its best to keep the new content from sticking.

### So just how *DO* you get your brain to treat programming like it's a hungry tiger?

There's the slow, tedious way, or the faster, more effective way. The slow way is about sheer repetition. You obviously know that you *are* able to learn and remember even the dullest of topics if you keep pounding the same thing into your brain. With enough repetition, your brain says, "This doesn't *feel* important to him, but he keeps looking at the same thing *over* and *over* and *over*, so I suppose it must be."

The faster way is to do ***anything that increases brain activity,*** especially different *types* of brain activity. The things on the previous page are a big part of the solution, and they're all things that have been proven to help your brain work in your favor. For example, studies show that putting words *within* the pictures they describe (as opposed to somewhere else in the page, like a caption or in the body text) causes your brain to try to make sense of how the words and picture relate, and this causes more neurons to fire. More neurons firing = more chances for your brain to *get* that this is something worth paying attention to, and possibly recording.

A conversational style helps because people tend to pay more attention when they perceive that they're in a conversation, since they're expected to follow along and hold up their end. The amazing thing is, your brain doesn't necessarily *care* that the "conversation" is between you and a book! On the other hand, if the writing style is formal and dry, your brain perceives it the same way you experience being lectured to while sitting in a roomful of passive attendees. No need to stay awake.

But pictures and conversational style are just the beginning…

## Here's what WE did

We used ***pictures***, because your brain is tuned for visuals, not text. As far as your brain's concerned, a picture really *is* worth a thousand words. And when text and pictures work together, we embedded the text *in* the pictures because your brain works more effectively when the text is *within* the thing it refers to, as opposed to in a caption or buried in the body text somewhere.

We used ***redundancy***, saying the same thing in *different* ways and with different media types, and *multiple senses*, to increase the chance that the content gets coded into more than one area of your brain.

We used concepts and pictures in ***unexpected*** ways because your brain is tuned for novelty, and we used pictures and ideas with at least *some* ***emotional*** *content*, because your brain is tuned to pay attention to the biochemistry of emotions. That which causes you to *feel* something is more likely to be remembered, even if that feeling is nothing more than a little ***humor***, ***surprise***, or ***interest.***

We used a personalized, ***conversational style***, because your brain is tuned to pay more attention when it believes you're in a conversation than if it thinks you're passively listening to a presentation. Your brain does this even when you're *reading*.

We included ***activities***, because your brain is tuned to learn and remember more when you ***do*** things than when you *read* about things. And we made the exercises challenging-yet-doable, because that's what most people prefer.

We used ***multiple learning styles***, because *you* might prefer step-by-step procedures, while someone else wants to understand the big picture first, and someone else just wants to see an example. But regardless of your own learning preference, *everyone* benefits from seeing the same content represented in multiple ways.

We include content for ***both sides of your brain***, because the more of your brain you engage, the more likely you are to learn and remember, and the longer you can stay focused. Since working one side of the brain often means giving the other side a chance to rest, you can be more productive at learning for a longer period of time.

And we included ***stories*** and exercises that present ***more than one point of view,*** because your brain is tuned to learn more deeply when it's forced to make evaluations and judgments.

We included ***challenges***, with exercises, and by asking ***questions*** that don't always have a straight answer, because your brain is tuned to learn and remember when it has to *work* at something. Think about it—you can't get your *body* in shape just by *watching* people at the gym. But we did our best to make sure that when you're working hard, it's on the *right* things. That ***you're not spending one extra dendrite*** processing a hard-to-understand example, or parsing difficult, jargon-laden, or overly terse text.

We used ***people***. In stories, examples, pictures, etc., because, well, *you're* a person. And your brain pays more attention to *people* than it does to *things*.

## Here's what YOU can do to bend your brain into submission

So, we did our part. The rest is up to you. These tips are a starting point; listen to your brain and figure out what works for you and what doesn't. Try new things.

Cut this out and stick it on your refrigerator.

1. **Slow down. The more you understand, the less you have to memorize.**
   Don't just *read*. Stop and think. When the book asks you a question, don't just skip to the answer. Imagine that someone really *is* asking the question. The more deeply you force your brain to think, the better chance you have of learning and remembering.

2. **Do the exercises. Write your own notes.**
   We put them in, but if we did them for you, that would be like having someone else do your workouts for you. And don't just *look* at the exercises. **Use a pencil.** There's plenty of evidence that physical activity *while* learning can increase the learning.

3. **Read "There Are No Dumb Questions."**
   That means all of them. They're not optional sidebars, ***they're part of the core content!*** Don't skip them.

4. **Make this the last thing you read before bed. Or at least the last challenging thing.**
   Part of the learning (especially the transfer to long-term memory) happens *after* you put the book down. Your brain needs time on its own, to do more processing. If you put in something new during that processing time, some of what you just learned will be lost.

5. **Talk about it. Out loud.**
   Speaking activates a different part of the brain. If you're trying to understand something, or increase your chance of remembering it later, say it out loud. Better still, try to explain it out loud to someone else. You'll learn more quickly, and you might uncover ideas you hadn't known were there when you were reading about it.

6. **Drink water. Lots of it.**
   Your brain works best in a nice bath of fluid. Dehydration (which can happen before you ever feel thirsty) decreases cognitive function.

7. **Listen to your brain.**
   Pay attention to whether your brain is getting overloaded. If you find yourself starting to skim the surface or forget what you just read, it's time for a break. Once you go past a certain point, you won't learn faster by trying to shove more in, and you might even hurt the process.

8. **Feel something.**
   Your brain needs to know that this *matters*. Get involved with the stories. Make up your own captions for the photos. Groaning over a bad joke is *still* better than feeling nothing at all.

9. **Write a lot of code!**
   There's only one way to learn to develop Go programs: **write a lot of code**. And that's what you're going to do throughout this book. Coding is a skill, and the only way to get good at it is to practice. We're going to give you a lot of practice: every chapter has exercises that pose a problem for you to solve. Don't just skip over them—a lot of the learning happens when you solve the exercises. We included a solution to each exercise—don't be afraid to **peek at the solution** if you get stuck! (It's easy to get snagged on something small.) But try to solve the problem before you look at the solution. And definitely get it working before you move on to the next part of the book.

# Read me

This is a learning experience, not a reference book. We deliberately stripped out everything that might get in the way of learning whatever it is we're working on at that point in the book. And the first time through, you need to begin at the beginning, because the book makes assumptions about what you've already seen and learned.

## It helps if you've done a *little* programming in some other language.

Most developers discover Go *after* they've learned some other programming language. (They often come seeking refuge from that other language.) We touch on the basics enough that a complete beginner can get by, but we don't go into great detail on what a variable is, or how an `if` statement works. You'll have an easier time if you've done at least a *little* of this before.

## We don't cover every type, function, and package ever created.

Go comes with a *lot* of software packages built in. Sure, they're all interesting, but we couldn't cover them all even if this book was *twice* as long. Our focus is on the core types and functions that *matter* to you, the beginner. We make sure you have a deep understanding of them, and confidence that you know how and when to use them. In any case, once you're done with *Head First Go*, you'll be able to pick up any reference book and get up to speed quickly on the packages we left out.

## The activities are NOT optional.

The exercises and activities are not add-ons; they're part of the core content of the book. Some of them are to help with memory, some are for understanding, and some will help you apply what you've learned. ***Don't skip the exercises.***

## The redundancy is intentional and important.

One distinct difference in a Head First book is that we want you to *really* get it. And we want you to finish the book remembering what you've learned. Most reference books don't have retention and recall as a goal, but this book is about *learning*, so you'll see some of the same concepts come up more than once.

## The code examples are as lean as possible.

It's frustrating to wade through 200 lines of code looking for the two lines you need to understand. Most examples in this book are shown in the smallest possible context, so that the part you're trying to learn is clear and simple. So don't expect the code to be robust, or even complete. That's *your* assignment after you finish the book. The book examples are written specifically for *learning*, and aren't always fully functional.

We've placed all the example files on the web so you can download them. You'll find them at *http://headfirstgo.com/*.

## Acknowledgments

***Series founders:***

Huge thanks to the Head First founders, **Kathy Sierra** and **Bert Bates**. I loved the series when I encountered it more than a decade ago, but never imagined I might be writing for it. Thank you for creating this amazing style of teaching!

***At O'Reilly:***

Thanks to everyone at O'Reilly who made this happen, particularly editor **Jeff Bleiel**, and to **Kristen Brown**, **Rachel Monaghan**, and the rest of the production team.

***Technical reviewers:***

Everyone makes mistakes, but luckily I have tech reviewers **Tim Heckman**, **Edward Yue Shung Wong**, and **Stefan Pochmann** to catch all of mine. You will never know how many problems they found, because I swiftly destroyed all the evidence. But their help and feedback were definitely necessary and are forever appreciated!

***And more thanks:***

Thanks to **Leo Richardson** for additional proofreading.

Perhaps most importantly, thanks to **Christine**, **Courtney**, **Bryan**, **Lenny**, and **Jeremy** for their patience and support (for two books now)!

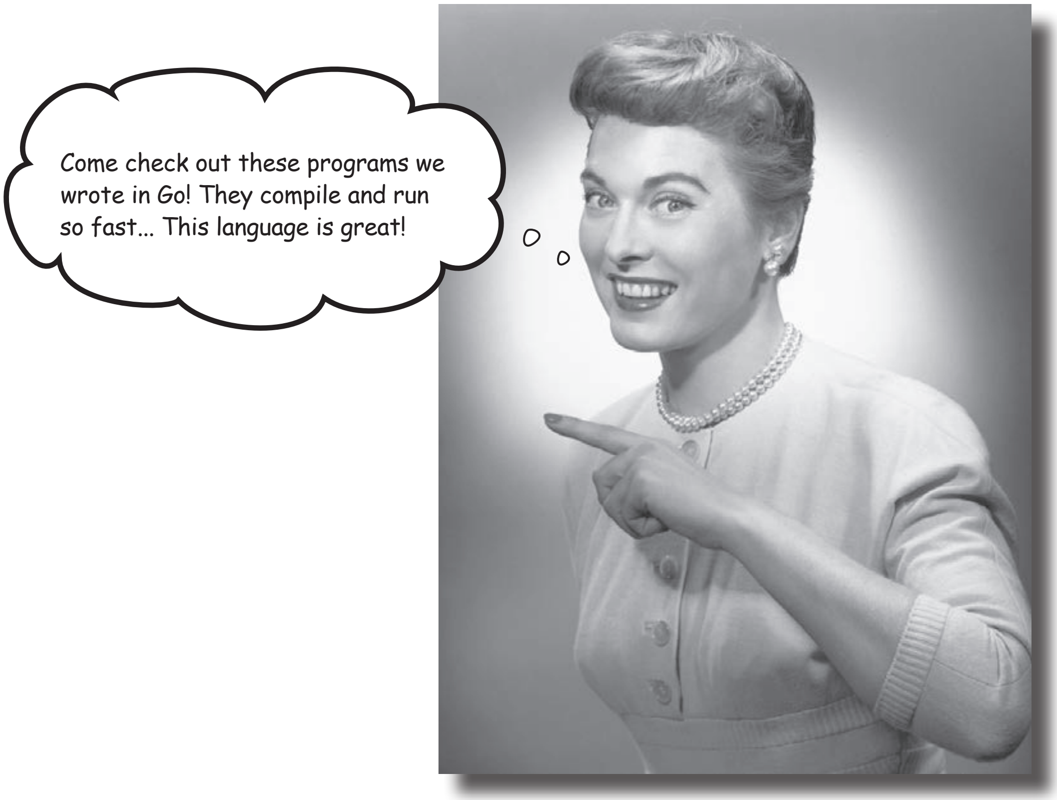
## O'Reilly Online Learning

For almost 40 years, O'Reilly Media has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

# 1 let's get going

# Syntax Basics


Come check out these programs we wrote in Go! They compile and run so fast... This language is great!

**Are you ready to turbo-charge your software?** Do you want a **simple** programming language that **compiles fast**? That **runs fast**? That makes it **easy to distribute** your work to users? Then **you're ready for Go**!

Go is a programming language that focuses on **simplicity** and **speed**. It's simpler than other languages, so it's quicker to learn. And it lets you harness the power of today's multicore computer processors, so your programs run faster. This chapter will show you all the Go features that will make **your life as a developer easier**, and make your **users happier**.

## Ready, set, Go!

Back in 2007, the search engine Google had a problem. They had to maintain programs with millions of lines of code. Before they could test new changes, they had to compile the code into a runnable form, a process which at the time took the better part of an hour. Needless to say, this was bad for developer productivity.

So Google engineers Robert Griesemer, Rob Pike, and Ken Thompson sketched out some goals for a new language:

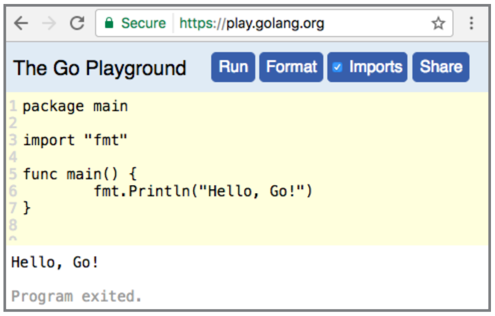
- Fast compilation
- Less cumbersome code
- Unused memory freed automatically (garbage collection)
- Easy-to-write software that does several operations simultaneously (concurrency)
- Good support for processors with multiple cores

After a couple years of work, Google had created Go: a language that was fast to write code for and produced programs that were fast to compile and run. The project switched to an open source license in 2009. It's now free for anyone to use. And you should use it! Go is rapidly gaining popularity thanks to its simplicity and power.

If you're writing a command-line tool, Go can produce executable files for Windows, macOS, and Linux, all from the same source code. If you're writing a web server, it can help you handle many users connecting at once. And no matter *what* you're writing, it will help you ensure that your code is easier to maintain and add to.

Ready to learn more? Let's Go!

## The Go Playground



```
package main

import "fmt"

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

```
Hello, Go!

Program exited.
```

The easiest way to try Go is to visit *https://play.golang.org* in your web browser. There, the Go team has set up a simple editor where you can enter Go code and run it on their servers. The result is displayed right there in your browser.

(Of course, this only works if you have a stable internet connection. If you don't, see page 25 to learn how to download and run the Go compiler directly on your computer. Then run the following examples using the compiler instead.)

Let's try it out now!

Do this!

1. Open *https://play.golang.org* in your browser. (Don't worry if what you see doesn't quite match the screenshot; it just means they've improved the site since this book was printed!)

2. Delete any code that's in the editing area, and type this instead:

```
package main

import "fmt"

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

Don't worry, we'll explain what all this means on the next page!

3. Click the Format button, which will automatically reformat your code according to Go conventions.

4. Click the Run button.

You should see "Hello, Go!" displayed at the bottom of the screen. Congratulations, you've just run your first Go program!

Turn the page, and we'll explain what we just did...

Output →

```
Hello, Go!
```

## What does it all mean?

You've just run your first Go program! Now let's look at the code and figure out what it actually means...

Every Go file starts with a `package` clause. A **package** is a collection of code that all does similar things, like formatting strings or drawing images. The `package` clause gives the name of the package that this file's code will become a part of. In this case, we use the special package `main`, which is required if this code is going to be run directly (usually from the terminal).

Next, Go files almost always have one or more `import` statements. Each file needs to **import** other packages before its code can use the code those other packages contain. Loading all the Go code on your computer at once would result in a big, slow program, so instead you specify only the packages you need by importing them.

This line says all the rest of the code in this file belongs to the "main" package.

This says we'll be using text-formatting code from the "fmt" package.

The "main" function is special; it gets run first when your program runs.

```
package main

import "fmt"

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

This line displays ("prints") "Hello, Go!" in your terminal (or web browser, if you're using the Go Playground).

It does this by calling the "Println" function from the "fmt" package.

The last part of every Go file is the actual code, which is often split up into one or more functions. A **function** is a group of one or more lines of code that you can **call** (run) from other places in your program. When a Go program is run, it looks for a function named `main` and runs that first, which is why we named this function `main`.

**Don't worry if you don't understand all this right now!**

We'll look at everything in more detail in the next few pages.

### The typical Go file layout

You'll quickly get used to seeing these three sections, in this order, in almost every Go file you work with:

1. The package clause
2. Any `import` statements
3. The actual code

```
The package clause  {package main

The imports section {import "fmt"

                    {func main() {
The actual code     {        fmt.Println("Hello, Go!")
                    {}
```

The saying goes, "a place for everything, and everything in its place." Go is a very *consistent* language. This is a good thing: you'll often find you just *know* where to look in your project for a given piece of code, without having to think about it!

## there are no Dumb Questions

**Q: My other programming language requires that each statement end with a semicolon. Doesn't Go?**

A: You *can* use semicolons to separate statements in Go, but it's not required (in fact, it's generally frowned upon).

**Q: What's this Format button? Why did we click that before running our code?**

A: The Go compiler comes with a standard formatting tool, called `go fmt`. The Format button is the web version of `go fmt`.

Whenever you share your code, other Go developers will expect it to be in the standard Go format. That means that things like indentation and spacing will be formatted in a standard way, making it easier for everyone to read. Where other languages achieve this by relying on people manually reformatting their code to conform to a style guide, with Go all you have to do is run `go fmt`, and it will automatically fix everything for you.

We ran the formatter on every example we created for this book, and you should run it on all your code, too!

## What if something goes wrong?

Go programs have to follow certain rules to avoid confusing the compiler. If we break one of these rules, we'll get an error message.

Suppose we forgot to add parentheses on our call to the `Println` function on line 6.

If we try to run this version of the program, we get an error:

```
Line 1 package main
     2
     3 import "fmt"
     4
     5 func main() {
     6         fmt.Println "Hello, Go!"
     7 }
```

Suppose we forgot the parentheses that used to be here...

Name of file used by Go Playground

Line number where the error occurred

Description of the error

```
prog.go:6:14: syntax error: unexpected literal "Hello, Go!" at end of statement
```

Character number within the line where the error occurred

Go tells us which source code file and line number we need to go to so we can fix the problem. (The Go Playground saves your code to a temporary file before running it, which is where the *prog.go* filename comes from.) Then it gives a description of the error. In this case, because we deleted the parentheses, Go can't tell we're trying to call the `Println` function, so it can't understand why we're putting `"Hello, Go"` at the end of line 6.

## Breaking Stuff is Educational!

We can get a feel for the rules Go programs have to follow by intentionally breaking our program in various ways. Take this code sample, try making one of the changes below, and run it. Then undo your change and try the next one. See what happens!

```
package main

import "fmt"

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

Try breaking our code sample and see what happens!

| If you do this... | | ...it will fail because... |
|---|---|---|
| Delete the package clause... | `~~package main~~` | Every Go file has to begin with a package clause. |
| Delete the `import` statement... | `~~import "fmt"~~` | Every Go file has to import every package it references. |
| Import a second (unused) package... | `import "fmt"` `import "strings"` | Go files must import *only* the packages they reference. (This helps keep your code compiling fast!) |
| Rename the `main` function... | `func ~~main~~hello` | Go looks for a function named `main` to run first. |
| Change the `Println` call to lowercase... | `fmt.~~P~~println("Hello, Go!")` | Everything in Go is case-sensitive, so although `fmt.Println` is valid, there's no such thing as `fmt.println`. |
| Delete the package name before `Println`... | `~~fmt.~~Println("Hello, Go!")` | The `Println` function isn't part of the `main` package, so Go needs the package name before the function call. |

Let's try the first one as an example...

Delete the package clause...

```

import "fmt"

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

You'll get an error!

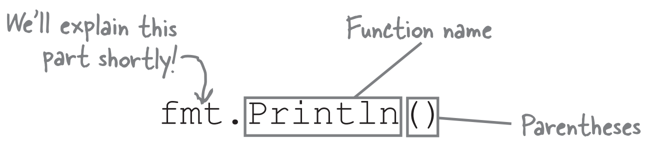
```
can't load package: package main:
prog.go:1:1: expected 'package', found 'import'
```

## Calling functions

Our example includes a call to the `fmt` package's `Println` function. To call a function, type the function name (`Println` in this case), and a pair of parentheses.

```
package main

import "fmt"

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

A call to the Println function



Like many functions, `Println` can take one or more **arguments**: values you want the function to work with. The arguments appear in parentheses after the function name.

Inside the parentheses are one or more arguments, separated by commas.

```
fmt.Println("First argument", "Second argument")
```

Output

```
First argument Second argument
```

`Println` can be called with no arguments, or you can provide several arguments. When we look at other functions later, however, you'll find that most require a specific number of arguments. If you provide too few or too many, you'll get an error message saying how many arguments were expected, and you'll need to fix your code.

## The Println function

Use the `Println` function when you need to see what your program is doing. Any arguments you pass to it will be printed (displayed) in your terminal, with each argument separated by a space.

After printing all its arguments, `Println` will skip to a new terminal line. (That's why "ln" is at the end of its name.)

```
fmt.Println("First argument", "Second argument")
fmt.Println("Another line")
```

Output

```
First argument Second argument
Another line
```

## Using functions from other packages

The code in our first program is all part of the `main` package, but the `Println` function is in the `fmt` package. (The `fmt` stands for "format.") To be able to call `Println`, we first have to import the package containing it.

```
package main

import "fmt"

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

We have to import the "fmt" package before we can access its Println function.

This specifies that we're calling a function that's part of the "fmt" package.

Once we've imported the package, we can access any functions it offers by typing the package name, a dot, and the name of the function we want.

Package name

Name of the function

```
fmt.Println()
```

Here's a code sample that calls functions from a couple other packages. Because we need to import multiple packages, we switch to an alternate format for the `import` statement that lets you list multiple packages within parentheses, one package name per line.

```
package main

import (
	"math"
	"strings"
)

func main() {
	math.Floor(2.75)
	strings.Title("head first go")
}
```

This alternate format for the "import" statement lets you import multiple packages at once.

Import the "math" package so we can use math.Floor.

Import the "strings" package so we can use strings.Title.

Call the Floor function from the "math" package.

Call the Title function from the "strings" package.

This program has no output. (We'll explain why in a moment!)

Once we've imported the `math` and `strings` packages, we can access the `math` package's `Floor` function with `math.Floor`, and the `strings` package's `Title` function with `strings.Title`.

You may have noticed that in spite of including those two function calls in our code, the above sample doesn't display any output. We'll look at how to fix that next.

## Function return values

In our previous code sample, we tried calling the math.Floor and strings.Title functions, but they didn't produce any output:

```
package main

import (
	"math"
	"strings"
)

func main() {
	math.Floor(2.75)
	strings.Title("head first go")
}
```

This program produces no output!

When we call the fmt.Println function, we don't need to communicate with it any further after that. We pass one or more values for Println to print, and we trust that it printed them. But sometimes a program needs to be able to call a function and get data back from it. For this reason, functions in most programming languages can have **return values**: a value that the function computes and returns to its caller.

The math.Floor and strings.Title functions are both examples of functions that use return values. The math.Floor function takes a floating-point number, rounds it down to the nearest whole number, and returns that whole number. And the strings.Title function takes a string, capitalizes the first letter of each word it contains (converting it to "title case"), and returns the capitalized string.

To actually see the results of these function calls, we need to take their return values and pass those to fmt.Println:

```
package main

import (
	"fmt"    <—— Import the "fmt" package as well.
	"math"
	"strings"
)

func main() {
	fmt.Println(math.Floor(2.75))
	fmt.Println(strings.Title("head first go"))
}
```

Call fmt.Println with the return value from math.Floor.

Takes a number, rounds it down, and returns that value

Call fmt.Println with the return value from strings.Title.

Takes a string, and returns a new string with each word capitalized

Output

```
2
Head First Go
```

Once this change is made, the return values get printed, and we can see the results.

## Pool Puzzle

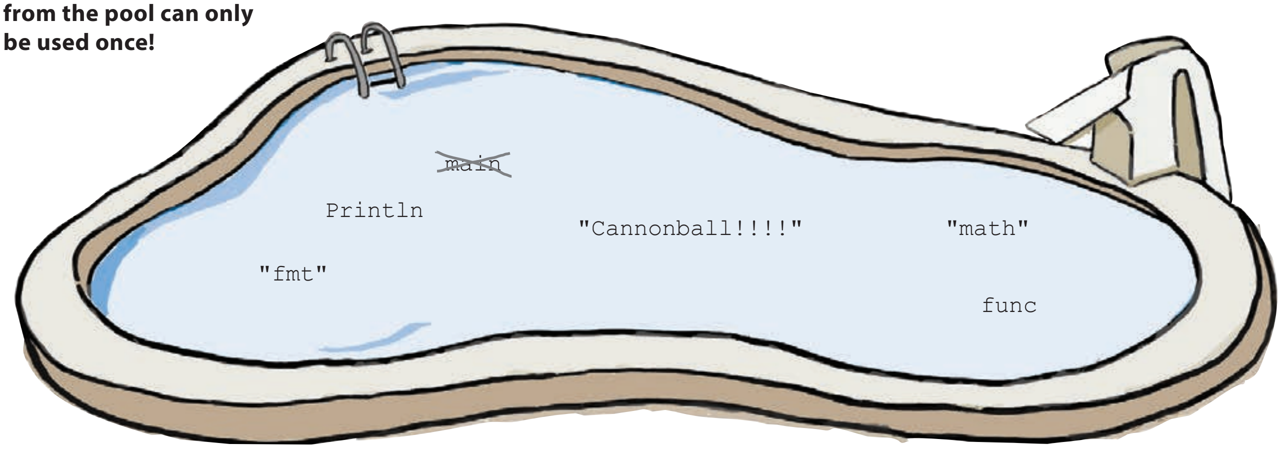
Your **job** is to take code snippets from the pool and place them into the blank lines in the code. **Don't** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make code that will run and produce the output shown.

```
package  main  <- We've done the first one for you!

import (
        ______
)

____ main() {
    fmt.Println(________________)
}
```

Output

```
Cannonball!!!!
```

**Note: each snippet from the pool can only be used once!**



Answers on page 29.

## A Go program template

For the code snippets that follow, just imagine inserting them into this full Go program:

Better yet, try typing this program into the Go Playground, and then insert the snippets one at a time to see for yourself what they do!

```
package main

import "fmt"

func main() {
        fmt.Println(                  )
}
```

Insert your code here!

## Strings

We've been passing **strings** as arguments to `Println`. A string is a series of bytes that usually represent text characters. You can define strings directly within your code using **string literals**: text between double quotation marks that Go will treat as a string.

Opening double quote → `"Hello, Go!"` ← Closing double quote

Output

```
Hello, Go!
```

Within strings, characters like newlines, tabs, and other characters that would be hard to include in program code can be represented with **escape sequences**: a backslash followed by characters that represent another character.

A newline within a string

```
"Hello,\nGo!"
```

Output

```
Hello,
Go!
```

```
"Hello,\tGo!"
```

```
Hello,	Go!
```

```
"Quotes: \"\""
```

```
Quotes: ""
```

```
"Backslash: \\"
```

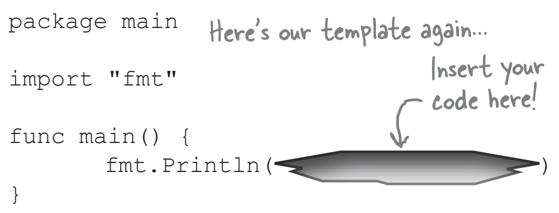
```
Backslash: \
```

| Escape sequence | Value |
|---|---|
| \n | A newline character. |
| \t | A tab character. |
| \\" | Double quotation marks. |
| \\\\ | A backslash. |

## Runes

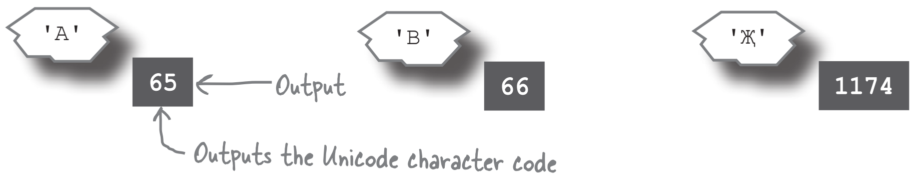
Whereas strings are usually used to represent a whole series of text characters, Go's **runes** are used to represent single characters.

String literals are written surrounded by double quotation marks ("), but **rune literals** are written with single quotation marks (').



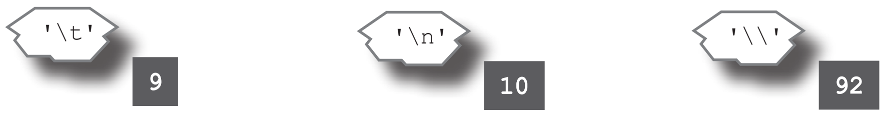
```
package main

import "fmt"

func main() {
        fmt.Println(                    )
}
```

Go programs can use almost any character from almost any language on earth, because Go uses the Unicode standard for storing runes. Runes are kept as numeric codes, not the characters themselves, and if you pass a rune to `fmt.Println`, you'll see that numeric code in the output, not the original character.



```
'A'    65
'B'    66
'Ж'    1174
```

Just as with string literals, escape sequences can be used in a rune literal to represent characters that would be hard to include in program code:



```
'\t'    9
'\n'    10
'\\'    92
```

## Booleans

**Boolean** values can be one of only two values: `true` or `false`. They're especially useful with conditional statements, which cause sections of code to run only if a condition is true or false. (We'll look at conditionals in the next chapter.)

```
true     true
false    false
```

## Numbers

You can also define numbers directly within your code, and it's even simpler than string literals: just type the number.

```
package main

import "fmt"

func main() {
        fmt.Println(                        )
}
```

Here's our template again...

Insert your code here!

```
42
```

An integer

```
42
```

Output

```
3.1415
```

A floating-point number

```
3.1415
```

As we'll see shortly, Go treats integer and floating-point numbers as different types, so remember that a decimal point can be used to distinguish an integer from a floating-point number.

## Math operations and comparisons

Go's basic math operators work just like they do in most other languages. The + symbol is for addition, - for subtraction, * for multiplication, and / for division.

```
1 + 2
```

```
3
```

```
5.4 - 2.2
```

```
3.2
```

```
3 * 4
```

```
12
```

```
7.5 / 5
```

```
1.5
```

You can use < and > to compare two values and see if one is less than or greater than another. You can use == (that's *two* equals signs) to see if two values are equal, and != (that's an exclamation point and an equals sign, read aloud as "not equal") to see if two values are not equal. <= tests whether the second value is less than *or* equal to the first, and >= tests whether the second value is greater than or equal to the first.

The result of a comparison is a Boolean value, either `true` or `false`.

```
4 < 6
```

```
true
```

```
4 > 6
```

```
false
```

```
2+2 == 5
```

```
false
```

```
2+2 != 5
```

```
true
```

```
4 <= 6
```

```
true
```

```
4 >= 4
```

```
true
```

## Types

In a previous code sample, we saw the `math.Floor` function, which rounds a floating-point number down to the nearest whole number, and the `strings.Title` function, which converts a string to title case. It makes sense that you would pass a number as an argument to the `Floor` function, and a string as an argument to the `Title` function. But what would happen if you passed a string to `Floor` and a number to `Title`?

```
package main

import (
	"fmt"
	"math"
	"strings"
)

func main() {
	fmt.Println(math.Floor("head first go"))
	fmt.Println(strings.Title(2.75))
}
```

Normally takes a floating-point number!

Normally takes a string!

Errors

```
cannot use "head first go" (type string) as type float64 in argument to math.Floor
cannot use 2.75 (type float64) as type string in argument to strings.Title
```

Go prints two error messages, one for each function call, and the program doesn't even run!

Things in the world around you can often be classified into different types based on what they can be used for. You don't eat a car or truck for breakfast (because they're vehicles), and you don't drive an omelet or bowl of cereal to work (because they're breakfast foods).

Likewise, values in Go are all classified into different **types**, which specify what the values can be used for. Integers can be used in math operations, but strings can't. Strings can be capitalized, but numbers can't. And so on.

Go is **statically typed**, which means that it knows what the types of your values are even before your program runs. Functions expect their arguments to be of particular types, and their return values have types as well (which may or may not be the same as the argument types). If you accidentally use the wrong type of value in the wrong place, Go will give you an error message. This is a good thing: it lets you find out there's a problem before your users do!

**Go is statically typed. If you use the wrong type of value in the wrong place, Go will let you know.**

## Types (continued)

You can view the type of any value by passing it to the `reflect` package's `TypeOf` function. Let's find out what the types are for some of the values we've already seen:

```
package main

import (
        "fmt"
        "reflect"
)

func main() {
        fmt.Println(reflect.TypeOf(42))
        fmt.Println(reflect.TypeOf(3.1415))
        fmt.Println(reflect.TypeOf(true))
        fmt.Println(reflect.TypeOf("Hello, Go!"))
}
```

Import the "reflect" package so we can use its TypeOf function.

Returns the type of its argument

Output

```
int
float64
bool
string
```

Here's what those types are used for:

| Type | Description |
|---|---|
| `int` | An integer. Holds whole numbers. |
| `float64` | A floating-point number. Holds numbers with a fractional part. (The `64` in the type name is because 64 bits of data are used to hold the number. This means that `float64` values can be fairly, but not infinitely, precise before being rounded off.) |
| `bool` | A Boolean value. Can only be `true` or `false`. |
| `string` | A string. A series of data that usually represents text characters. |

**Exercise**

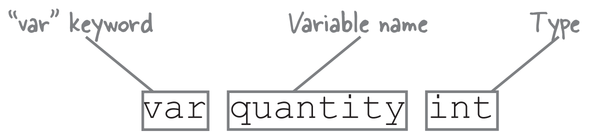
Draw lines to match each code snippet below to a type. Some types will have more than one snippet that matches with them.

```
reflect.TypeOf(25)                    int
reflect.TypeOf(true)
reflect.TypeOf(5.2)                   float64
reflect.TypeOf(1)
reflect.TypeOf(false)                 bool
reflect.TypeOf(1.0)
reflect.TypeOf("hello")               string
```

Answers on page 29.

## Declaring variables

In Go, a **variable** is a piece of storage containing a value. You can give a variable a name by using a **variable declaration**. Just use the `var` keyword followed by the desired name and the type of values the variable will hold.



Variable name → Type of value the variable will hold

```
var quantity int
var length, width float64   ← You can declare multiple variables of the same type at once.
var customerName string
```

Once you declare a variable, you can assign any value of that type to it with = (that's a *single* equals sign):

```
quantity = 2
customerName = "Damon Cole"
```

You can assign values to multiple variables in the same statement. Just place multiple variable names on the left side of the =, and the same number of values on the right side, separated with commas.

```
length, width = 1.2, 2.4   ← Assigning multiple variables at once.
```

Once you've assigned values to variables, you can use them in any context where you would use the original values:

```
package main

import "fmt"

func main() {
	var quantity int
	var length, width float64
	var customerName string

	quantity = 4
	length, width = 1.2, 2.4
	customerName = "Damon Cole"

	fmt.Println(customerName)
	fmt.Println("has ordered", quantity, "sheets")
	fmt.Println("each with an area of")
	fmt.Println(length*width, "square meters")
}
```

Declaring the variables (the three `var` lines)

Assigning values to the variables (the three assignment lines)

Using the variables (the four `fmt.Println` lines)

```
Damon Cole
has ordered 4 sheets
each with an area of
2.88 square meters
```

## Declaring variables (continued)

If you know beforehand what a variable's value will be, you can declare variables and assign them values on the same line:

Just add an assignment onto the end.

Declaring variables AND assigning values

```
var quantity int = 4
var length, width float64 = 1.2, 2.4
var customerName string = "Damon Cole"
```

If you're declaring multiple variables, provide multiple values.

You can assign new values to existing variables, but they need to be values of the same type. Go's static typing ensures you don't accidentally assign the wrong kind of value to a variable.

Assigned types don't match the declared types!

```
quantity = "Damon Cole"
customerName = 4
```

Errors

```
cannot use "Damon Cole" (type string) as type int in assignment
cannot use 4 (type int) as type string in assignment
```

If you assign a value to a variable at the same time as you declare it, you can usually omit the variable type from the declaration. The type of the value assigned to the variable will be used as the type of that variable.

Omit variable types.

```
var quantity = 4
var length, width = 1.2, 2.4
var customerName = "Damon Cole"
fmt.Println(reflect.TypeOf(quantity))
fmt.Println(reflect.TypeOf(length))
fmt.Println(reflect.TypeOf(width))
fmt.Println(reflect.TypeOf(customerName))
```

```
int
float64
float64
string
```

## Zero values

If you declare a variable without assigning it a value, that variable will contain the **zero value** for its type. For numeric types, the zero value is actually 0:

```
var myInt int
var myFloat float64
fmt.Println(myInt, myFloat)
```

The zero value for "int" variables is 0.

```
0 0
```

The zero value for "float64" variables is 0.

But for other types, a value of 0 would be invalid, so the zero value for that type may be something else. The zero value for `string` variables is an empty string, for example, and the zero value for `bool` variables is `false`.

```
var myString string
var myBool bool
fmt.Println(myString, myBool)
```

The zero value for "string" variables is an empty string.

```
 false
```

The zero value for "bool" variables is false.

## Code Magnets

A Go program is all scrambled up on the fridge. Can you reconstruct the code snippets to make a working program that will produce the given output?

Output

```
I started with 10 apples.
Some jerk ate 4 apples.
There are 6 apples left.
```

```
, "apples.")
, "apples.")
, "apples left.")
var
var
int
originalCount
func main() {
}
int
originalCount
fmt.Println("I started with",
eatenCount
fmt.Println("Some jerk ate",
=
=
fmt.Println("There are",
10
4
eatenCount
package main
originalCount-eatenCount
import (
        "fmt"
)
```

Answers on page 30.

## Short variable declarations

We mentioned that you can declare variables and assign them values on the same line:

Just add an assignment onto the end.

Declaring variables AND assigning values

```
var quantity int = 4
var length, width float64 = 1.2, 2.4
var customerName string = "Damon Cole"
```

If you're declaring multiple variables, provide multiple values.

But if you know what the initial value of a variable is going to be as soon as you declare it, it's more typical to use a **short variable declaration**. Instead of explicitly declaring the type of the variable and later assigning to it with =, you do both at once using :=.

Let's update the previous example to use short variable declarations:

```
package main

import "fmt"

func main() {
	quantity := 4
	length, width := 1.2, 2.4
	customerName := "Damon Cole"

	fmt.Println(customerName)
	fmt.Println("has ordered", quantity, "sheets")
	fmt.Println("each with an area of")
	fmt.Println(length*width, "square meters")
}
```

Declaring variables AND assigning values

```
Damon Cole
has ordered 4 sheets
each with an area of
2.88 square meters
```

There's no need to explicitly declare the variable's type; the type of the value assigned to the variable becomes the type of that variable.

Because short variable declarations are so convenient and concise, they're used more often than regular declarations. You'll still see both forms occasionally, though, so it's important to be familiar with both.

## Breaking Stuff is Educational!

Take our program that uses variables, try making one of the changes below, and run it. Then undo your change and try the next one. See what happens!

```
package main

import "fmt"

func main() {
        quantity := 4
        length, width := 1.2, 2.4
        customerName := "Damon Cole"

        fmt.Println(customerName)
        fmt.Println("has ordered", quantity, "sheets")
        fmt.Println("each with an area of")
        fmt.Println(length*width, "square meters")
}
```

```
Damon Cole
has ordered 4 sheets
each with an area of
2.88 square meters
```

| If you do this... | | ...it will fail because... |
|---|---|---|
| Add a second declaration for the same variable | `quantity := 4`<br>`quantity := 4` | You can only declare a variable once. (Although you can assign new values to it as often as you want. You can also declare other variables with the same name, as long as they're in a different scope. We'll learn about scopes in the next chapter.) |
| Delete the `:` from a short variable declaration | `quantity = 4` | If you forget the `:`, it's treated as an assignment, not a declaration, and you can't assign to a variable that hasn't been declared. |
| Assign a `string` to an `int` variable | `quantity := 4`<br>`quantity = "a"` | Variables can only be assigned values of the same type. |
| Mismatch number of variables and values | `length, width := 1.2` | You're required to provide a value for every variable you're assigning, and a variable for every value. |
| Remove code that uses a variable | ~~`fmt.Println(customerName)`~~ | All declared variables must be used in your program. If you remove the code that uses a variable, you must also remove the declaration. |

## Naming rules

Go has one simple set of rules that apply to the names of variables, functions, and types:

- A name must begin with a letter, and can have any number of additional letters and numbers.
- If the name of a variable, function, or type begins with a capital letter, it is considered **exported** and can be accessed from packages outside the current one. (This is why the `P` in `fmt.Println` is capitalized: so it can be used from the `main` package or any other.) If a variable/function/type name begins with a lowercase letter, it is considered **unexported** and can only be accessed within the current package.

OK
```
length
stack2
sales.Total
```

Illegal
```
2stack
sales.total
```

Can't start with a number!

Can't access anything in another package unless its name is capitalized!

Those are the only rules enforced by the language. But the Go community follows some additional conventions as well:

- If a name consists of multiple words, each word after the first should be capitalized, and they should be attached together without spaces between them, like this: `topPrice`, `RetryConnection`, and so on. (The first letter of the name should only be capitalized if you want to export it from the package.) This style is often called *camel case* because the capitalized letters look like the humps on a camel.
- When the meaning of a name is obvious from the context, the Go community's convention is to abbreviate it: to use `i` instead of `index`, `max` instead of `maximum`, and so on. (However, we at Head First believe that nothing is obvious when you're learning a new language, so we will *not* be following that convention in this book.)

OK
```
sheetLength
TotalUnits
i
```

Breaks conventions
```
sheetlength
Total_Units
index
```

Subsequent words should be capitalized!

This is legal, but words should be joined directly!

Consider replacing with an abbreviation!

**Only variables, functions, or types whose names begin with a capital letter are considered exported: accessible from packages outside the current package.**

## Conversions

Math and comparison operations in Go require that the included values be of the same type. If they're not, you'll get an error when trying to run your code.

Set up a float64 variable.

Set up an int variable.

If we use both the float64 and the int in a math operation...

Or a comparison...

```
var length float64 = 1.2
var width int = 2
fmt.Println("Area is", length*width)
fmt.Println("length > width?", length > width)
```

...we'll get errors!

Errors

```
invalid operation: length * width (mismatched types float64 and int)
invalid operation: length > width (mismatched types float64 and int)
```

The same is true of assigning new values to variables. If the type of value being assigned doesn't match the declared type of the variable, you'll get an error.

Set up a float64 variable.

Set up an int variable.

If we assign the int value to the float64 variable...

```
var length float64 = 1.2
var width int = 2
length = width
fmt.Println(length)
```

...we'll get an error!

Error

```
cannot use width (type int) as type float64 in assignment
```

The solution is to use **conversions**, which let you convert a value from one type to another type. You just provide the type you want to convert a value to, immediately followed by the value you want to convert in parentheses.

```
var myInt int = 2
float64(myInt)
```

Type to convert to

Value to convert

The result is a new value of the desired type. Here's what we get when we call `TypeOf` on the value in an integer variable, and again on that same value after conversion to a `float64`:

Without a conversion...

```
var myInt int = 2
fmt.Println(reflect.TypeOf(myInt))
fmt.Println(reflect.TypeOf(float64(myInt)))
```

With a conversion...

```
int
float64
```

Type is changed.

## Conversions (continued)

Let's update our failing code example to convert the `int` value to a `float64` before using it in any math operations or comparisons with other `float64` values.

```
var length float64 = 1.2
var width int = 2
fmt.Println("Area is", length*float64(width))
fmt.Println("length > width?", length > float64(width))
```

Convert the int to a float64 before multiplying it with another float64.

Convert the int to a float64 before comparing it with another float64.

```
Area is 2.4
length > width? false
```

The math operation and comparison both work correctly now!

Now let's try converting an `int` to a `float64` before assigning it to a `float64` variable:

```
var length float64 = 1.2
var width int = 2
length = float64(width)
fmt.Println(length)
```

Convert the int to a float64 before assigning it to the float64 variable.

```
2
```

Again, with the conversion in place, the assignment is successful.

When making conversions, be aware of how they might change the resulting values. For example, `float64` variables can store fractional values, but `int` variables can't. When you convert a `float64` to an `int`, the fractional portion is simply dropped! This can throw off any operations you do with the resulting value.

```
var length float64 = 3.75
var width int = 5
width = int(length)
fmt.Println(width)
```

This conversion causes the fractional portion to be dropped!

```
3
```

The resulting value is 0.75 lower!

As long as you're cautious, though, you'll find conversions essential to working with Go. They allow otherwise-incompatible types to work together.

We've written the Go code below to calculate a total price with tax and determine if we have enough funds to make a purchase. But we're getting errors when we try to include it in a full program!

```
var price int = 100
fmt.Println("Price is", price, "dollars.")

var taxRate float64 = 0.08
var tax float64 = price * taxRate
fmt.Println("Tax is", tax, "dollars.")

var total float64 = price + tax
fmt.Println("Total cost is", total, "dollars.")

var availableFunds int = 120
fmt.Println(availableFunds, "dollars available.")
fmt.Println("Within budget?", total <= availableFunds)
```

Errors

```
invalid operation: price * taxRate (mismatched types int and float64)
invalid operation: price + tax (mismatched types int and float64)
invalid operation: total <= availableFunds (mismatched types float64 and int)
```

Fill in the blanks below to update this code. Fix the errors so that it produces the expected output. (Hint: Before doing math operations or comparisons, you'll need to use conversions to make the types compatible.)

```
var price int = 100
fmt.Println("Price is", price, "dollars.")

var taxRate float64 = 0.08
var tax float64 = ________________________
fmt.Println("Tax is", tax, "dollars.")

var total float64 = ____________________
fmt.Println("Total cost is", total, "dollars.")

var availableFunds int = 120
fmt.Println(availableFunds, "dollars available.")
fmt.Println("Within budget?", ______________________________)
```

Expected output

```
Price is 100 dollars.
Tax is 8 dollars.
Total cost is 108 dollars.
120 dollars available.
Within budget? true
```

Answers on page 30.

## Installing Go on your computer

The Go Playground is a great way to try out the language. But its practical uses are limited. You can't use it to work with files, for example. And it doesn't have a way to take user input from the terminal, which we're going to need for an upcoming program.

So, to wrap up this chapter, let's download and install Go on your computer. Don't worry, the Go team has made it really easy! On most operating systems, you just have to run an installer program, and you'll be done.

Do this!

1. Visit *https://golang.org* in your web browser.
2. Click the download link.
3. Select the installation package for your operating system (OS). The download should begin automatically.
4. Visit the installation instructions page for your OS (you may be taken there automatically after the download starts), and follow the directions there.
5. Open a new terminal or command prompt window.
6. Confirm Go was installed by typing `go version` at the prompt and hitting the Return or Enter key. You should see a message with the version of Go that's installed.

Watch it!

**Websites are always changing.**

*It's possible that* golang.org *or the Go installer will be updated after this book is published, and these directions will no longer be completely accurate. In that case, visit:*

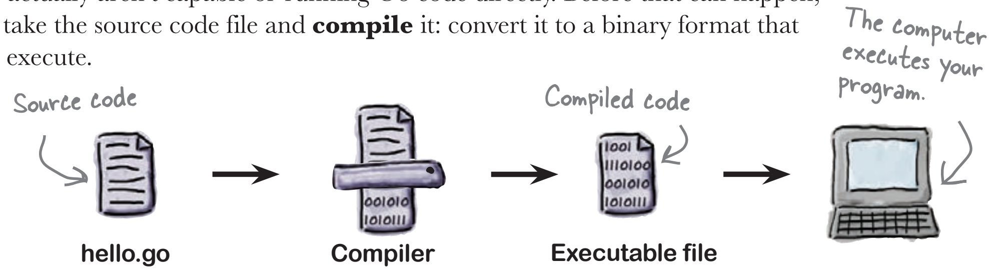
http://headfirstgo.com

*for help and troubleshooting tips!*

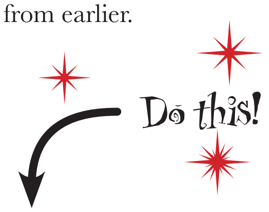
## Compiling Go code

Our interaction with the Go Playground has consisted of typing in code and having it mysteriously run. Now that we've actually installed Go on your computer, it's time to take a closer look at how this works.

Computers actually aren't capable of running Go code directly. Before that can happen, we need to take the source code file and **compile** it: convert it to a binary format that a CPU can execute.



Let's try using our new Go installation to compile and run our "Hello, Go!" example from earlier.

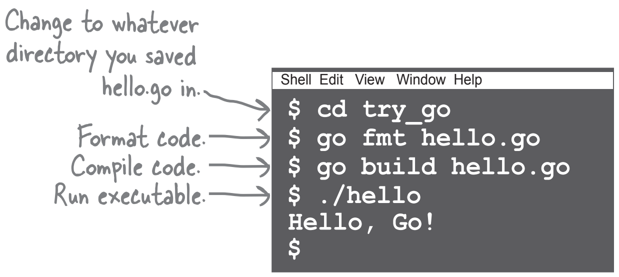


Save this to a file.

hello.go

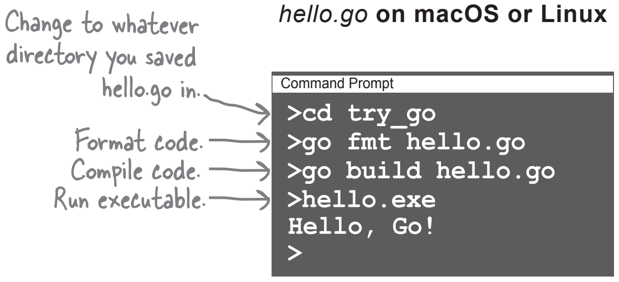
```
package main

import "fmt"

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

1. Using your favorite text editor, save our "Hello, Go!" code from earlier in a plain-text file named *hello.go*.
2. Open a new terminal or command prompt window.
3. In the terminal, change to the directory where you saved *hello.go*.
4. Run **`go fmt hello.go`** to clean up the code formatting. (This step isn't required, but it's a good idea anyway.)
5. Run **`go build hello.go`** to compile the source code. This will add an executable file to the current directory. On macOS or Linux, the executable will be named just *hello*. On Windows, the executable will be named *hello.exe*.
6. Run the executable file. On macOS or Linux, do this by typing **`./hello`** (which means "run a program named `hello` in the current directory"). On Windows, just type **`hello.exe`**.


Shell Edit View Window Help
Change to whatever directory you saved hello.go in.
Format code.
Compile code.
Run executable.

```
$ cd try_go
$ go fmt hello.go
$ go build hello.go
$ ./hello
Hello, Go!
$
```

**Compiling and running *hello.go* on macOS or Linux**



```
>cd try_go
>go fmt hello.go
>go build hello.go
>hello.exe
Hello, Go!
>
```

**Compiling and running *hello.go* on Windows**

## Go tools

When you install Go, it adds an executable named *go* to your command prompt. The *go* executable gives you access to various commands, including:

| Command | Description |
|---|---|
| `go build` | Compiles source code files into binary files. |
| `go run` | Compiles and runs a program, without saving an executable file. |
| `go fmt` | Reformats source files using Go standard formatting. |
| `go version` | Displays the current Go version. |

We just tried the `go fmt` command, which reformats your code in the standard Go format. It's equivalent to the Format button on the Go Playground site. We recommend running `go fmt` on every source file you create.

Most editors can be set up to automatically run go fmt every time you save a file! See https://blog.golang.org/go-fmt-your-code.

We also used the `go build` command to compile code into an executable file. Executable files like this can be distributed to users, and they'll be able to run them even if they don't have Go installed.

But we haven't tried the `go run` command yet. Let's do that now.

## Try out code quickly with "go run"

The `go run` command compiles and runs a source file, without saving an executable file to the current directory. It's great for quickly trying out simple programs. Let's use it to run our *hello.go* sample.

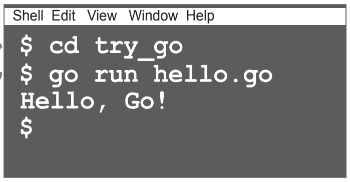
**hello.go**

```
package main

import "fmt"

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

1. Open a new terminal or command prompt window.
2. In the terminal, change to the directory where you saved *hello.go*.
3. Type **`go run hello.go`** and hit Enter/Return. (The command is the same on all operating systems.)

Change to whatever directory you saved hello.go in.

Run source file.



```
$ cd try_go
$ go run hello.go
Hello, Go!
$
```

**Running *hello.go* with `go run` (works on any OS)**

You'll immediately see the program output. If you make changes to the source code, you don't have to do a separate compilation step; just run your code with `go run` and you'll be able to see the results right away. When you're working on small programs, `go run` is a handy tool to have!

## Your Go Toolbox

**That's it for Chapter 1! You've added function calls and types to your toolbox.**

### Function calls

A function is a chunk of code that you can call from other places in your program.

When calling a function, you can use arguments to provide the function with data.

### Types

Values in Go are classified into different types, which specify what the values can be used for.

Math operations and comparisons between different types are not allowed, but you can convert a value to a new type if needed.

Go variables can only store values of their declared type.

### BULLET POINTS

- A **package** is a group of related functions and other code.
- Before you can use a package's functions within a Go file, you need to **import** that package.
- A `string` is a series of bytes that usually represent text characters.
- A `rune` represents a single text character.
- Go's two most common numeric types are `int`, which holds integers, and `float64`, which holds floating-point numbers.
- The `bool` type holds Boolean values, which are either `true` or `false`.
- A **variable** is a piece of storage that can contain values of a specified type.
- If no value has been assigned to a variable, it will contain the **zero value** for its type. Examples of zero values include `0` for `int` or `float64` variables, or `""` for `string` variables.
- You can declare a variable and assign it a value at the same time using a `:=` **short variable declaration**.
- A variable, function, or type can only be accessed from code in other packages if its name begins with a capital letter.
- The `go fmt` command automatically reformats source files to use Go standard formatting. You should run `go fmt` on any code that you plan to share with others.
- The `go build` command **compiles** Go source code into a binary format that computers can execute.
- The `go run` command compiles and runs a program without saving an executable file in the current directory.

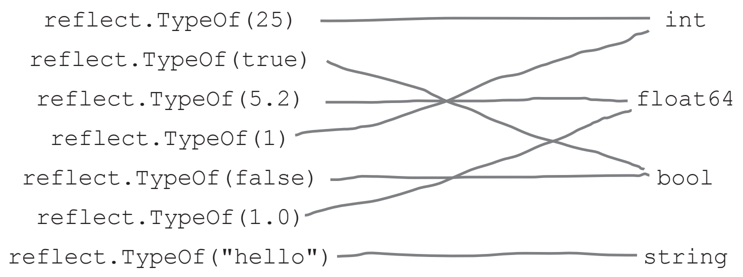
## Pool Puzzle Solution

```
package main

import (
	"fmt"
)

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

Output

```
Cannonball!!!!
```

Draw lines to match each code snippet below to a type.
Some types will have more than one snippet that matches with them.



## Code Magnets Solution

```
package main

import (
        "fmt"
)

func main() {

        var originalCount int = 10
        fmt.Println("I started with", originalCount , "apples.")
        var eatenCount int = 4
        fmt.Println("Some jerk ate", eatenCount , "apples.")
        fmt.Println("There are", originalcount-eatenCount , "apples left.")
}
```

Output

```
I started with 10 apples.
Some jerk ate 4 apples.
There are 6 apples left.
```

### Exercise Solution

Fill in the blanks below to update this code. Fix the errors so that it produces the expected output. (Hint: Before doing math operations or comparisons, you'll need to use conversions to make the types compatible.)

```
var price int = 100
fmt.Println("Price is", price, "dollars.")

var taxRate float64 = 0.08
var tax float64 = float64(price) * taxRate
fmt.Println("Tax is", tax, "dollars.")

var total float64 = float64(price) + tax
fmt.Println("Total cost is", total, "dollars.")

var availableFunds int = 120
fmt.Println(availableFunds, "dollars available.")
fmt.Println("Within budget?", total <= float64(availableFunds))
```

Expected output

```
Price is 100 dollars.
Tax is 8 dollars.
Total cost is 108 dollars.
120 dollars available.
Within budget? true
```

O'REILLY®

# Go

## What will you learn from this book?

Go makes it easy to build software that's simple, reliable, and efficient. And this book makes it easy for programmers like you to get started. Go is designed for high-performance networking and multiprocessing, but—like Python and JavaScript—the language is easy to read and use. With this practical hands-on guide, you'll learn how to write Go code using clear examples that demonstrate the language in action. Best of all, you'll understand the conventions and techniques that employers want entry-level Go developers to know.

Understand data structures like arrays and slices.

```
{"a", "b", "c", "d", "e"}
```

Describe common features using interfaces.

Serve your web app to the world.

http://localhost:8080/guestbook

**Guestbook**

5 total signatures - Add Your Signature

First signature

Second signature

Third signature

Can I sign now?

Hooray, it works!

Send data between goroutines using channels.

"*Head First Go* strikes the right tone for those of us who just want to get on and do things rather than wring our hands over endless syntax and technical decisions. Given just a little of your time, you'll learn something new and useful here, even if your work predominantly focuses on other languages."

—*Peter Cooper*
*Editor,* Golang Weekly

## Why does this book look so different?

Based on the latest research in cognitive science and learning theory, *Head First Go* uses a visually rich format to engage your mind rather than a text-heavy approach that puts you to sleep. Why waste your time struggling with new concepts? This multisensory learning experience is designed for the way your brain really works.

**For sale in the Indian Subcontinent (India, Pakistan, Bangladesh, Sri Lanka, Nepal, Bhutan, Maldives) and African Continent (excluding Morocco, Algeria, Tunisia, Libya, Egypt, and the Republic of South Africa) only. Illegal for sale outside of these countries**

oreilly.com
twitter.com/headfirstlabs
facebook.com/HeadFirst

**MRP: ₹ 1,500.00**

SPD®
**SHROFF PUBLISHERS & DISTRIBUTORS PVT. LTD.**

ISBN: 978-93-5213-825-8

9 789352 138258
First Edition/2019/Paperback/English