

Up to date for iOS 9,
Xcode 7, and Swift 2!

Swift Apprentice

Beginning programming
with Swift 2

By the raywenderlich.com Tutorial Team

Janie Clayton, Alexis Gallagher, Matt Galloway,
Eli Ganem, Erik Kerber, and Ben Morrow

Swift Apprentice

By the raywenderlich.com Tutorial Team

Janie Clayton, Alexis Gallagher, Matt Galloway,
Eli Ganem, Erik Kerber, and Ben Morrow

Copyright © 2015 Razeware LLC.

All rights reserved. No part of this book or corresponding materials (such as text, images, or source code) may be reproduced or distributed by any means without prior written permission of the copyright owner.

This book and all corresponding materials (such as source code) are provided on an "as is" basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose, and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this book are the property of their respective owners.

Dedications

"To the iOS programming community. This year has been a difficult one for me and so many people helped me get through it in both large and small ways.
Stay awesome and think different."

— *Janie Clayton*

"To my dear wife, Ringae, my beautiful children, Odysseus & Kallisto, my beloved parents, Ralph & Sophia, and to the human beings and other sentient life of planet Earth."

— *Alexis Gallagher*

"To my amazing family who keep putting up with me spending my spare hours writing books like this."

— *Matt Galloway*

"To the two loves of my life, Moriah and Lia."

— *Eli Ganem*

"To the Swift community, both old and new—and to Brigitte, who puts up with my long hours at the keyboard to help make this book possible."

— *Erik Kerber*

"To my grandmother, Maw Maw.
You're still superwoman in my book!"

— *Ben Morrow*

About the authors



Janie Clayton is an iOS/Mac developer at Black Pixel. Janie does extensive conference speaking, cohosts the NSBrief podcast, and blogs about her existential crises at redqueencoder.com.

She lives in Madison, WI with her grumble of pugs.



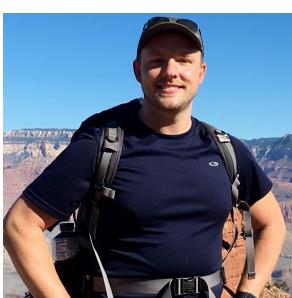
Alexis Gallagher is a San Francisco-based freelance iOS engineer, who has worked for surgeons and for oyster-fanciers, for startups and for bond traders, as a software developer, an analyst, a designer, a research scientist, an actor, and an ornament. He is currently learning how to draw, and to manage unruly dreams.



Matt Galloway is a software engineer with a passion for excellence. He stumbled into iOS programming when it first was a thing, and has never looked back. When not coding he likes to brew his own beer.



Eli Ganem is an iOS developer at Facebook, in charge of training new iOS engineers who join the company. He is passionate about teaching, writing, and sharing his knowledge with others.



Erik Kerber is a software developer in Minneapolis, MN, and a lead iOS developer at Target. He does his best to balance a life behind the keyboard with cycling, hiking, scuba diving and traveling.

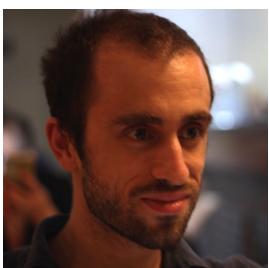


Ben Morrow is a developer, author, and hackathon organizer. With the Apple Watch community, he's building apps made for the new device platform. You can find him at <http://happy.watch>

About the editors



Jawwad Ahmad was a technical editor for this book. He is a freelance iOS developer that dove into Swift head first and has not looked back. He enjoys mentoring and teaching and was the original founder of the NYC iOS Study Group. He's worked for companies as large as The New York Times and as small as GateGuru, a six-person startup.



Rui Peres was a technical editor for this book, and a Lead Software Engineer at MailOnline. He has a particular interest in the Open Source world, where he spends some of his time exploring, contributing and using. When not blogging at codeplease.io or preparing the next iOS Goodies issue, you can find him on Twitter, or doing one more set in the gym.



Bradley C. Phillips was the editor for this book, and was the first editor to come aboard at raywenderlich.com.

He has worked as a journalist and previously directed the intelligence department of an investigative firm in New York City. Right now, Bradley works freelance and pursues his own projects. Contact him if you need a skilled and experienced editor for your blog, books or anything else.



Tim Mitra was a developmental editor for this book. He is a mobile app developer and artist. He also teaches iOS dev, Swift and Objective-C. Tim runs iT Guy Technologies, a software development company in Toronto, Canada.

He studied Fine Arts before there were Macs and he's worked for many years in software development, IT, graphic design, publishing and printing. He is also the founder and host of the MTJC Podcast on app development & business.



Audrey Tam was a developmental editor for this book. She writes tutorials and book chapters for raywenderlich.com, and teaches iOS and watchOS app dev to beginners. She knits a lot, which her Watch interprets as walking, 1 step per stitch!

Find her on twitter and ravelry as mataharimau.



Greg Heo was a final pass editor for this book, and has been part of the editorial team at raywenderlich.com since 2012.

He has been nerding out with computers since the Commodore 64 era in the 80s and continues to this day on the web and on iOS. He likes caffeine, codes with two-space tabs and writes with semicolons.

Table of Contents: Overview

Introduction.....	15
Section I: Swift Basics	20
Chapter 1: Coding Essentials & Playground Basics ..	22
Chapter 2: Variables & Constants	34
Chapter 3: Numeric Types & Operations	44
Chapter 4: Strings	56
Chapter 5: Making Decisions	66
Chapter 6: Repeating Steps.....	78
Chapter 7: Functions	90
Chapter 8: Closures.....	102
Chapter 9: Optionals	109
Section II: Collection Types	119
Chapter 10: Arrays	121
Chapter 11: Dictionaries	136
Chapter 12: Sets.....	145
Section III: Building Your Own Types.....	153
Chapter 13: Structures.....	155
Chapter 14: Classes	168
Chapter 15: Advanced Classes.....	183
Chapter 16: Enumerations.....	201
Chapter 17: Properties.....	213

Chapter 18: Methods.....	226
Chapter 19: Protocols	238
Chapter 20: Protocol-Oriented Programming	258
Section IV: Advanced Topics	273
Chapter 21: Error Handling	274
Chapter 22: Generics	287
Chapter 23: Functional Programming	299
Conclusion	310

Table of Contents: Extended

Introduction.....	15
Who this book is for	15
What you need.....	16
How to use this book	16
What's in store	17
Book source code and forums	17
Book updates.....	18
License.....	18
Acknowledgments	19
Section I: Swift Basics	20
Chapter 1: Coding Essentials & Playground Basics	22
How a computer works	22
Playgrounds	29
Key points.....	32
Where to go from here?	33
Chapter 2: Variables & Constants.....	34
Naming data	34
Tuples	39
Type inference.....	41
Key points.....	42
Where to go from here?	43
Challenges.....	43
Chapter 3: Numeric Types & Operations	44
Arithmetic operations	44
Comparison operators.....	51
Key points.....	54
Where to go from here?	54
Challenges.....	55
Chapter 4: Strings	56
How computers represent strings	56
Strings in Swift.....	59
Key points.....	64

Where to go from here?	64
Challenges.....	65
Chapter 5: Making Decisions.....	66
The if statement.....	66
Switch statements.....	71
Key Points	75
Where to go from here?	76
Challenges.....	76
Chapter 6: Repeating Steps.....	78
Ranges.....	78
Loops	79
Key points.....	88
Where to go from here?	88
Challenges.....	89
Chapter 7: Functions.....	90
Function basics	90
Functions as variables	97
Key points.....	99
Where to go from here?	99
Challenges.....	99
Chapter 8: Closures	102
Closure basics	102
Key points	107
Where to go from here?.....	107
Challenges	108
Chapter 9: Optionals	109
Introducing nil.....	109
Introducing optionals.....	111
Unwrapping optionals.....	112
Nil coalescing	115
Key points	116
Where to go from here?.....	116
Challenges	117

Section II: Collection Types 119**Chapter 10: Arrays 121**

Creating arrays	122
Accessing elements.....	123
Manipulating elements	127
Iterating through an array.....	129
Sequence operations	131
Running time for array operations.....	132
Key points	133
Where to go from here?.....	133
Challenges	133

Chapter 11: Dictionaries..... 136

Creating dictionaries	137
Accessing values	138
Modifying dictionaries	139
Iterating through dictionaries.....	140
Sequence operations	141
Running time for dictionary operations.....	141
Key points	142
Where to go from here?.....	142
Challenges	143

Chapter 12: Sets 145

Creating sets	146
Accessing elements.....	147
Adding and removing elements	147
Iterating through a set.....	148
Set operations.....	148
Running time for set operations	150
Key points	150
Where to go from here?.....	150
Challenges	151

Section III: Building Your Own Types..... 153**Chapter 13: Structures 155**

Introducing structures	155
------------------------------	-----

Introducing methods	162
Structures as values.....	164
Structs everywhere.....	165
Key points	166
Where to go from here?.....	166
Challenges	166
Chapter 14: Classes	168
Creating classes.....	168
Reference types.....	169
Introducing access control	174
Understanding state and side effects	177
When to use a class versus a struct	179
Key points	180
Where to go from here?.....	181
Challenges	181
Chapter 15: Advanced Classes.....	183
Introducing inheritance	183
Inheritance and class initialization	188
Two-phase initialization	190
When and why to subclass.....	194
Understanding the class lifecycle	196
Key points	199
Where to go from here?.....	200
Challenge.....	200
Chapter 16: Enumerations	201
Your first enumeration.....	202
Raw values.....	205
Associated values	207
Enumeration as state machine.....	209
Optionals	210
Key points	211
Where to go from here?.....	211
Challenges	211
Chapter 17: Properties	213
Stored properties.....	213

Computed properties	215
Type properties	218
Property observers	221
Lazy properties	223
Key points	224
Where to go from here?.....	224
Challenges	225
Chapter 18: Methods	226
Method refresher	226
Introducing self.....	228
Introducing initializers.....	229
Introducing mutating methods	232
Type methods.....	233
Key points	235
Where to go from here?.....	235
Challenges	235
Chapter 19: Protocols	238
Introducing protocols.....	238
Implementing protocols	242
Protocols in action.....	248
Protocols in the standard library.....	250
Key points	255
Where to go from here?.....	256
Challenges	256
Chapter 20: Protocol-Oriented Programming.....	258
Introducing protocol extensions.....	259
Default implementations	260
Type constraints	264
Protocol-oriented benefits	265
Why Swift is a protocol-oriented language	269
Key points	271
Where to go from here?.....	271
Challenges	272

Section IV: Advanced Topics	273
Chapter 21: Error Handling	274
What is error handling?	274
The <code>ErrorType</code> protocol.....	275
Throwing errors.....	276
Advanced error handling.....	279
Key points	285
Where to go from here?.....	286
Chapter 22: Generics.....	287
Introducing generics	287
Anatomy of generic types.....	290
Dictionaries	293
Optionals	294
Generic functions	295
Key points	296
Where to go from here?.....	296
Challenges	297
Chapter 23: Functional Programming	299
Higher-order functions: <code>map</code>	299
Introducing functional programming?	303
Map as philosophy in practice	306
Two more classics: <code>filter</code> and <code>reduce</code>	307
Key points	309
Where to go from here?.....	309
Challenges	309
Conclusion	310

Introduction

By Greg Heo

Welcome to the world of Swift! Swift is a programming language introduced by Apple in the summer of 2014. Since then, Swift has gone through one major version bump and has become the easiest way to get started with developing on Apple's platforms: iOS, OS X, watchOS and tvOS.

Swift is a great language for learning general programming concepts. You'll study things like data structures, classes and functional programming in this book, which are common topics that come up in many other programming languages.

Swift is also a lot of fun! It's easy to try out small bits of code and play around with different values as you test new ideas. Programming is a hands-on experience, and Swift makes it fast and easy to both follow along with this book, as well as explore on your own.

Who this book is for

This book is for people who know a little about programming and want to learn Swift. Maybe you've written a bit of JavaScript for your website or some short programs in Python. This is the book for you! You'll learn the fundamental concepts of programming while also mastering the Swift language.

If you're a complete beginner to programming, this is also the book for you! There are short exercises and challenges throughout the book to give you some programming practice and test your knowledge along the way.

If you want to get right into iOS app development while learning bits of the Swift language as you go, we recommend you read through *The iOS Apprentice, 4th Edition*. *The iOS Apprentice* and this book make very good companions—you can read them in parallel, or use this book as a reference to expand on topics you read about in *The iOS Apprentice*.



What you need

To follow along with this book, you need the following:

- **A Mac running OS X Yosemite (10.10.4) or El Capitan (10.11)**, with the latest point release and security patches installed. This is so you can install the latest version of the required development tool: Xcode.
- **Xcode 7 or later.** Xcode is the main development tool for writing code in Swift. You need Xcode 7 at a minimum, since that version includes Swift 2. You can download the latest version of Xcode for free from the Mac App Store, here: apple.co/1FLn51R.

If you haven't installed the latest version of Xcode, be sure to do that before continuing with the book.

How to use this book

Each chapter of this book presents some theory on the topic at hand, along with plenty of Swift code to demonstrate the practical applications of what you're learning.

Since this is a book for beginners, we suggest reading it in order the first time. After that, the book makes a great reference for you to return to and refresh your memory on particular topics.

All the code in this book is platform-neutral; that means it isn't specific to iOS, OS X or any other platform. The code runs in **playgrounds**, which you'll learn about in the very first chapter.

As you read through the book, you can follow along and type the code into your own playground. That means you'll be able to "play" with the code by making changes and immediately seeing the results.

You'll find **mini-exercises** throughout the book, which are short exercises about the topic at hand. There are also **challenges** at the end of each chapter, which are either programming questions or longer coding exercises to test your knowledge. You'll get the most out of this book if you follow along with these exercises and challenges.

What's in store

This book is divided into four sections. Each section has a short introduction that describes its chapters, their topics and the overarching themes of the section. Here's a brief overview of the book's sections:

Section I: Swift Basics

The first section of the book starts at the very beginning of the computing environment: first, how computers work, and then, how Swift's playgrounds feature works. With those logistics out of the way, you'll take a tour of the fundamentals of the Swift language and learn the basics of managing data, structuring your code and performing simple operations and calculations.

Section II: Collection Types

Stored data is a core component of any app, whether it's a list of friends in your social networking app or a set of unlockable characters in your hit game. In this section, you'll learn how to store collections of data in Swift.

Section III: Building Your Own Types

Swift comes with basic building blocks, but its real power is in the custom things you can build to model parts of your app. Swift has no idea about playable characters and monsters and power-ups, for example—these are things you need to build yourself! You'll learn how to do that in this section.

Section IV: Advanced Topics

The final section of the book covers a few advanced topics in Swift. You'll learn about specific things, such as how to handle problems that come up as your code runs, as well as about more general things, like generics, which help you understand some of Swift's behind-the-scenes mechanisms.

Book source code and forums

This book comes with complete source code for each of the chapters—it's shipped with the PDF. You'll find playgrounds with the code from the chapters, as well as solutions to the challenges for your reference.

We've set up an official forum for the book at www.raywenderlich.com/forums. This is a great place to ask any questions you have about the book and the Swift language in general, or to submit any errors you might find.

Book updates

Since you've purchased the PDF version of this book, you get free access to any updates we make to the book!

The best way to get update notifications is to sign up for our monthly newsletter. This includes a list of the tutorials that came out on raywenderlich.com that month, any important news like book updates or new books, and a list of our favorite development links for that month. You can sign up here:

- www.raywenderlich.com/newsletter

License

By purchasing *Swift Apprentice*, you have the following license:

- You are allowed to use and/or modify the source code in *Swift Apprentice* in as many apps as you want, with no attribution required.
- You are allowed to use and/or modify all art, images and designs that are included in *Swift Apprentice* in as many apps as you want, but must include this attribution line somewhere inside your app: "Artwork/images/designs: from *Swift Apprentice* book, available at www.raywenderlich.com".
- The source code included in *Swift Apprentice* is for your personal use only. You are NOT allowed to distribute or sell the source code in *Swift Apprentice* without prior authorization.
- This book is for your personal use only. You are NOT allowed to sell this book without prior authorization, or distribute it to friends, coworkers or students; they would need to purchase their own copies.

All materials provided with this book are provided on an "as is" basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and non-infringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this guide are the properties of their respective owners.

Acknowledgments

We would like to thank many people for their assistance in making this book possible:

- **Our families:** For bearing with us in this crazy time as we worked all hours of the night to get this book ready for publication!
- **Everyone at Apple:** For producing the amazing hardware and software we know and love, and for creating an exciting new programming language that we can use to make apps for that hardware!
- And most importantly, **the readers of raywenderlich.com—especially you!** Thank you so much for reading our site and purchasing this book. Your continued readership and support is what makes all of this possible!

Section I: Swift Basics

The chapters in this section will introduce you to the very basics of programming in Swift. From the fundamentals of how computers work all the way up to language structures, you'll cover enough of the language to be able to work with data and organize your code's behavior.

The section begins with some groundwork to get you started:

- **Chapter 1, Coding Essentials & Playground Basics** – This is it, your whirlwind introduction to the world of programming! You'll begin with an overview of computers and programming, and then say hello to Swift playgrounds, which are where you'll spend your coding time for the rest of this book.

Next, you'll learn the basics of data in Swift:

- **Chapter 2, Variables & Constants** – You'll learn about variables and constants, which are the "places" to store data. You'll also learn about data types and how Swift keeps track of the *kinds* of data that pass through your code.
- **Chapter 3, Numeric Types & Operations** – You'll begin working with basic numeric types such as integers and floating-point numbers, as well as the Boolean type. You'll also see some operations in action, from comparisons to arithmetic operations such as addition and subtraction.
- **Chapter 4, Strings** – You use strings to store text—anything from the text in a button to a caption for an image to the text of this entire book! You'll learn about the string and character types, as well as some common operations on those types.

Once you have the basic data types in your head, it'll be time to *do* things with that data:

- **Chapter 5, Making Decisions** – You don't always want your code to run straight through from start to finish. You'll learn how to make decisions in code and set up conditions for certain blocks of code to run.
- **Chapter 6, Repeating Steps** – Continuing the theme of code not running in a straight line, you'll learn how to use loops to repeat certain steps.
- **Chapter 7, Functions** – Functions are the basic building blocks you use to

structure your code in Swift. You'll learn how to define functions to group your code into reusable units.

- **Chapter 8, Closures** – Closures are similar to functions. You'll learn how to use them to easily pass around blocks of code.

The final chapter of the section loops back to data:

- **Chapter 9, Optionals** – This chapter covers optionals, a special type in Swift that represents either a real value or the absence of a value. By the end of this chapter, you'll know why you need optionals and how to use them safely.

These fundamentals will get you Swiftly on your way, and before you know it, you'll be ready for the more advanced topics that follow. Let's get started!

Chapter 1: Coding Essentials & Playground Basics

By Matt Galloway

Welcome to our book! In this first chapter, you're going to learn a few basics. You'll learn how code works first. Then you'll learn about the tools you'll be using to write Swift code.

How a computer works

You may not believe me when I say it, but a computer is not very smart on its own. The power of computers is all derived from how they're programmed by people like you and me. If you want to successfully harness the power of a computer—and I assume you do if you're reading this book—it's important to understand how computers work.

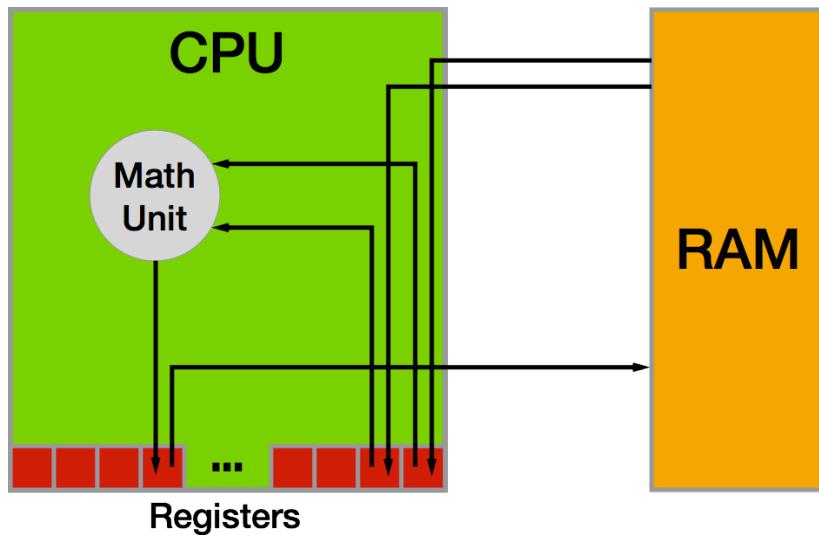
It may also surprise you to learn that computers themselves are rather simple machines. At the heart of a computer is a **Central Processing Unit (CPU)**. This is essentially a math machine. It performs addition, subtraction, and other arithmetical operations on numbers. Everything you see when you operate your computer is all built upon a CPU crunching numbers many millions of times per second. Isn't it amazing what can come from just numbers?



The CPU stores the numbers it acts upon in small memory units called **registers**. The CPU is able to read numbers into registers from the computer's main memory,

known as **Random Access Memory (RAM)**. It's also able to write the number stored in a register back into RAM. This allows the CPU to work with large amounts of data that wouldn't all fit in the bank of registers.

Here is a diagram of how this works:



As the CPU pulls values from RAM into its registers, it uses those values in its math unit and stores the results back in another register.

Each time the CPU makes an addition, a subtraction, a read from RAM or a write to RAM, it's executing a single **instruction**. Each computer program is usually made up of thousands to millions of instructions. A complex computer program such as your operating system, Mac OS X (yes, that's a computer program too!), may have many millions of instructions in total.

It's entirely possible to write individual instructions to tell a computer what to do, but for all but the simplest programs, it would be immensely time-consuming and tedious. This is because most computer programs aim to do much more than simple math—computer programs manipulate images, they allow you to surf the Internet and they allow you to chat with your friends.

Instead of writing individual instructions, you write **code** in a specific **programming language**, which in your case will be Swift. This code is put through a computer program called a **compiler**, which converts the code into instructions the CPU knows how to execute. Each line of code you write will turn into many instructions—some lines could end up being tens of instructions!

Representing numbers

As you know by now, numbers are a computer's bread and butter, the fundamental basis of everything it does. Whatever information you send to the compiler will eventually become a number. For example, each character within a block of text is represented by a number. You'll learn more about this in Chapter 4, which delves

into **strings**, the computer term for a block of text.

Images are no exception: In a computer, each image is also represented by a series of numbers. An image is split into many thousands or even millions of blocks called pixels, where each pixel is a solid color. If you look closely at your computer screen, you may be able to make out these blocks. That is, unless you have a particularly high-resolution display where the pixels are incredibly small! Each of these solid color pixels is usually represented by three numbers: one for the amount of red, one for the amount of green and one for the amount of blue. For example, an entirely red pixel would be 100% red, 0% green and 0% blue.

The numbers the CPU works with are notably different from those you are used to. When you deal with numbers in day-to-day life, you use them in **base 10**, otherwise known as the **decimal** system. Having used this numerical system for so long, you intuitively understand how it works, but let's take a deeper look so that later, you can appreciate the CPU's point of view.

The decimal or base 10 number **423** contains **three units**, **two tens** and **four hundreds**:

1000	100	10	1
0	4	2	3

In the base-10 system, each digit of a number can have a value of 0, 1, 2, 3, 4, 5, 6, 7, 8 or 9, giving a total of 10 possible values for each digit. Yep, that's why it's called base 10!

But the true value of each digit depends on its position within the number. Moving from right to left, each digit gets multiplied by an increasing power of 10. So the multiplier for the far-right position is 10 to the power of 0, which is 1. Moving to the left, the next multiplier is 10 to the power of 1, which is 10. Moving again to the left, the next multiplier is 10 to the power of 2, which is 100. And so on.

This means each digit has a value ten times that of the digit to its right. The number **432** is therefore equal to:

$$(0 * 1000) + (4 * 100) + (3 * 10) + (2 * 1) = 432$$

Binary numbers

Because you're trained to operate in base 10, you don't have to think about how to read most numbers—it feels quite natural. But to a computer, base 10 is way too complicated! Computers are simple-minded, remember? They like to work with base 2.

Base 2 is often called **binary**, which you've likely heard of before. It follows that base 2 has only two options for each digit: 0 or 1.

Almost all modern computers use binary because at the physical level, it's easiest to handle only two options for each digit. In digital electronic circuitry, which is mostly what comprises a computer, the presence of an electrical current is 1 and the absence is 0—that's base 2!

Note: There have been computers both real and imagined that use the ternary numeral system, which has three possible values instead of two. Computer scientists, engineers and dedicated hackers continue to explore the possibilities of a base-3 computer. See https://en.wikipedia.org/wiki/Ternary_computer and <http://hackaday.com/tag/ternary-computer/>.

Here's a representation of the base-2 number 1101:

8	4	2	1
1	1	0	1

In the base-10 number system, the place values increased by a factor of 10: 1, 10, 100, 1000, etc. Here, you can see they increase by a factor of 2: 1, 2, 4, 8, 16, etc. The general rule is to multiply each digit by an increasing power of the base number—in this case, powers of 2—moving from right to left.

So the far-right digit represents $(1 * 2^0)$, which is $(1 * 1)$, which is 1. The next digit to the left represents $(0 * 2^1)$, which is $(0 * 2)$, which is 0. In the illustration above, you can see the powers of 2 on top of the blocks.

Put another way, every power of 2 either is (1) or isn't (0) present as a component of a binary number. The decimal version of a binary number is the sum of all the powers of 2 that make up that number. So the binary number 1101 is equal to:

$$(1 * 8) + (1 * 4) + (0 * 2) + (1 * 1) = 13$$

And if you wanted to convert the base-10 number 423 into binary, you would simply need to break down 423 into its component powers of 2. You would wind up with:

$$(1 * 256) + (1 * 128) + (0 * 64) + (1 * 32) + (0 * 16) + (0 * 8) + (1 * 4) + (1 * 2) + (1 * 1) = 423$$

As you can see by scanning the binary digits in the above equation, the resulting binary number is 110100111. You can prove to yourself that this is equivalent to 423 by doing the math!

The computer term given to each digit of a binary number is a **bit**. Eight bits make up a **byte**. Four bits is called a **nibble**, a play on words that shows even old school computer scientists had a sense of humor.



A computer's limited memory means it can normally deal with numbers up to a certain length. Each register, for example, is usually 32 or 64 bits in length, which is why we speak of 32-bit and 64-bit CPUs.

Therefore, a 32-bit CPU can handle a maximum base-10 number of 4,294,967,295, which is the base-2 number 111111111111111111111111111111. That is 32 ones—count them!

It's possible for a computer to handle numbers that are larger than the CPU maximum, but the calculations have to be split up and managed in a special and longer way, much like the long multiplication you performed in school.

Hexadecimal numbers

As you can imagine, working with binary numbers can become quite tedious, because it can take a long time to write or type them. For this reason, in computer programming we often use another number format known as **hexadecimal**, or **hex** for short. This is **base 16**.

Of course, there are not 16 distinct numbers to use for digits; there are only 10. To supplement these, we use the first six letters, **a** through **f**. They are equivalent to decimal numbers like so:

- a = 10
- b = 11
- c = 12
- d = 13
- e = 14
- f = 15

Here's a base-16 example using the same format as before:

4096	256	16	1
c	0	d	e

Notice first that you can make hexadecimal numbers look like words. That means you can have a little bit of fun. :]

Now the values of each digit refer to powers of 16. In the same way as before, you can convert this number to decimal like so:

$$(12 * 4096) + (0 * 256) + (13 * 16) + (14 * 1) = 49374$$

You translate the letters to their decimal equivalents and then perform the usual calculations.

But why bother with this?

Hexadecimal is important because each hexadecimal digit can represent precisely four binary digits. The binary number 1111 is equivalent to hexadecimal f. It follows that you can simply concatenate the binary digits representing each hexadecimal digit, creating a hexidecimal number that is shorter than its binary or decimal equivalents.

For example, consider the number c0de from above:

```
c = 1100  
0 = 0000  
d = 1101  
e = 1110  
  
c0de = 1100 0000 1101 1110
```

This turns out to be rather helpful, given how computers use long 32-bit or 64-bit binary numbers. Recall that the longest 32-bit number in decimal is 4,294,967,295. In hexadecimal, it is ffffff. That's much more compact and clear.

How code works

Computers have a lot of constraints, and by themselves, they can only do a small number of things. The power that the computer programmer adds, through coding, is putting these small things together, in the right order, to produce something much bigger.

Coding is much like writing a recipe. You assemble ingredients (the data) and give the computer a step-by-step recipe for how to use them.

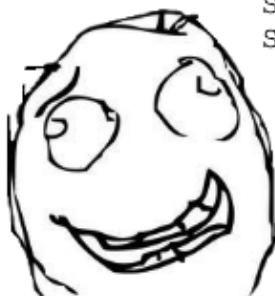
Here's an example:

```
Step 1. Load photo from hard drive.  
Step 2. Resize photo to 400 pixels wide by 300 pixels high.  
Step 3. Apply sepia filter to photo.  
Step 4. Print photo.
```

This is what's known as **pseudo-code**. It isn't written in a valid computer programming language, but it represents the **algorithm** that you want to use. In this case, the algorithm takes a photo, resizes it, applies a filter and then prints it. It's a relatively straightforward algorithm, but it's an algorithm nonetheless!

Here's my plan:

```
Step 1. Think up amazing app!  
Step 2. Write some code!  
Step 3. Rock the app store!  
Step 4. Make a million bucks!
```



I'll write a program in
Swift to execute my plan!

You do know it's not
that simple, right?



Swift code is just like this: a step-by-step list of instructions for the computer. These instructions will get more and more complex as you read through this book, but the principle is the same: You are simply telling the computer what to do, one step at a time.

Each programming language is a high-level, pre-defined way of expressing these steps. The compiler knows how to interpret the code you write and convert it into instructions that the CPU can execute.

There are many different programming languages, each with its own advantages and disadvantages. Swift is an extremely modern language and it's just in the early stages of development. It incorporates the strengths of many other languages while ironing out some of their weaknesses. In years to come, we'll look back on Swift as being old and crusty, too. But for now, it's an extremely exciting language because it is quickly evolving.

This has been a brief tour of computer hardware, number representation and code, and how they all work together to create a modern program. That was a lot to cover in one section! Now it's time to learn about the tools you'll use to write in Swift as you follow along with this book.

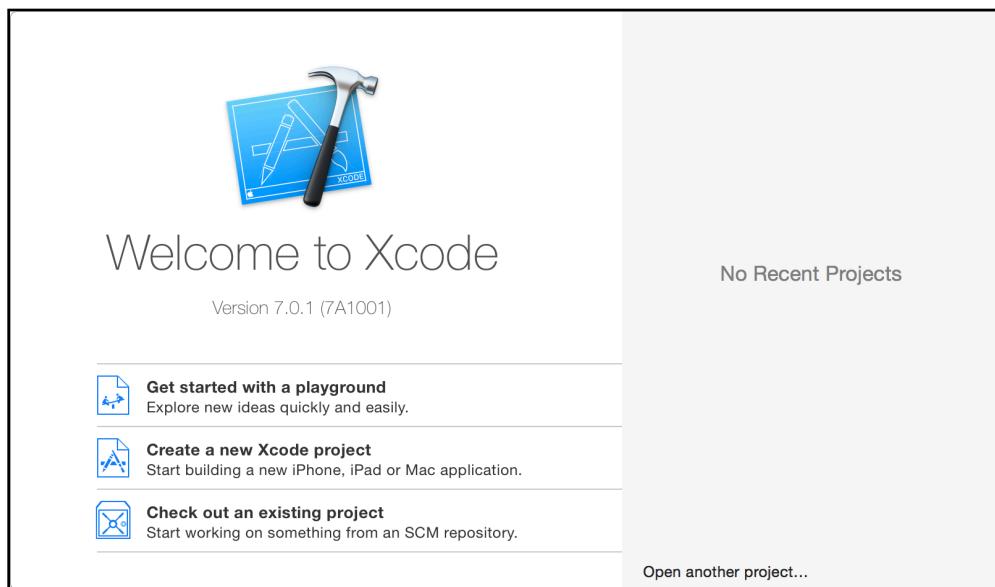
Playgrounds

The set of tools you use to write software is often referred to as the **toolchain**. The part of the toolchain into which you write your code is known as the **Integrated Development Environment (IDE)**. The most commonly used IDE for Swift is called Xcode, and that's what you'll be using.

Xcode includes a handy feature called a **Playground**, which allows you to quickly write and test code without needing to build a complete app. You'll use playgrounds throughout the book to practice coding, so it's important to understand how they work. That's what you'll learn during the rest of this chapter.

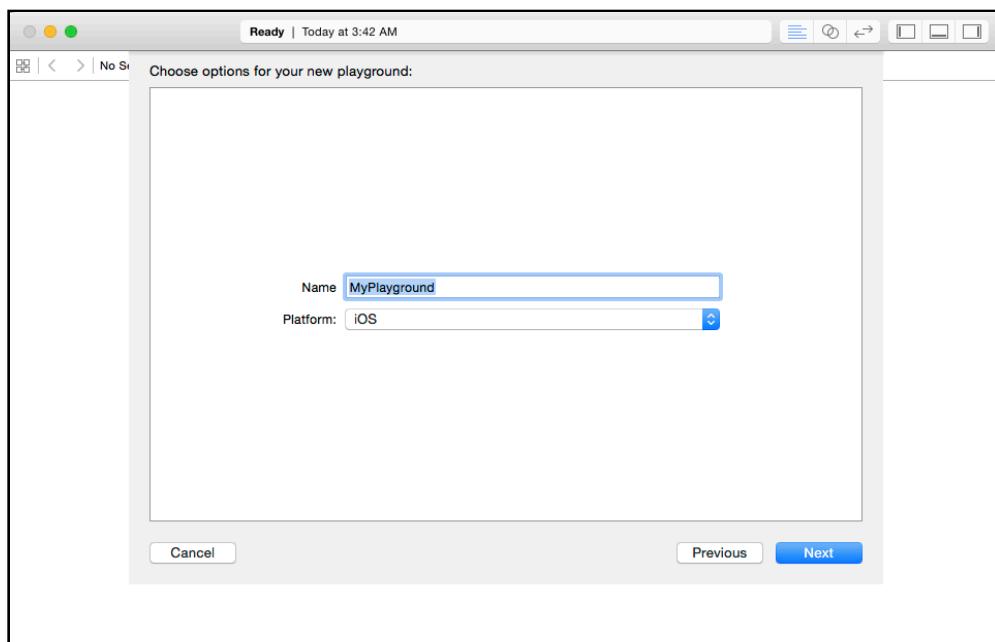
Creating a playground

When you open Xcode, it will greet you with the following welcome screen:



If you don't see this screen, it's most likely because the "Show this window when Xcode launches" option was unchecked. You can also open the screen by pressing **Command-Shift-1** or clicking **Window\Welcome to Xcode** from the menu bar.

From the welcome screen, you can jump quickly into a playground by clicking on **Get started with a playground**. Click on that now and Xcode will take you to a new screen:



From here, you can name the playground and select your desired platform. The name is merely cosmetic and for your own use; when you create your playgrounds, feel free to choose names that will help you remember what they're about. For example, while you're working through Chapter 2, you may want to name your playground **Chapter2**.

The second option you can see here is the platform. Currently, this can be either **iOS** or **Mac**.

The one you choose simply defines which template Xcode will use to create the playground. Each platform comes with its own environment set up and ready for you to begin playing around with code for that platform. For the purposes of this book, choose whichever you wish, as you're not going to be writing any platform-specific code anyway. You're going to be learning the core principles of the Swift language!

Once you've chosen a name and a platform, click on **Next** and then save the playground. Xcode then presents you with the playground, like so:



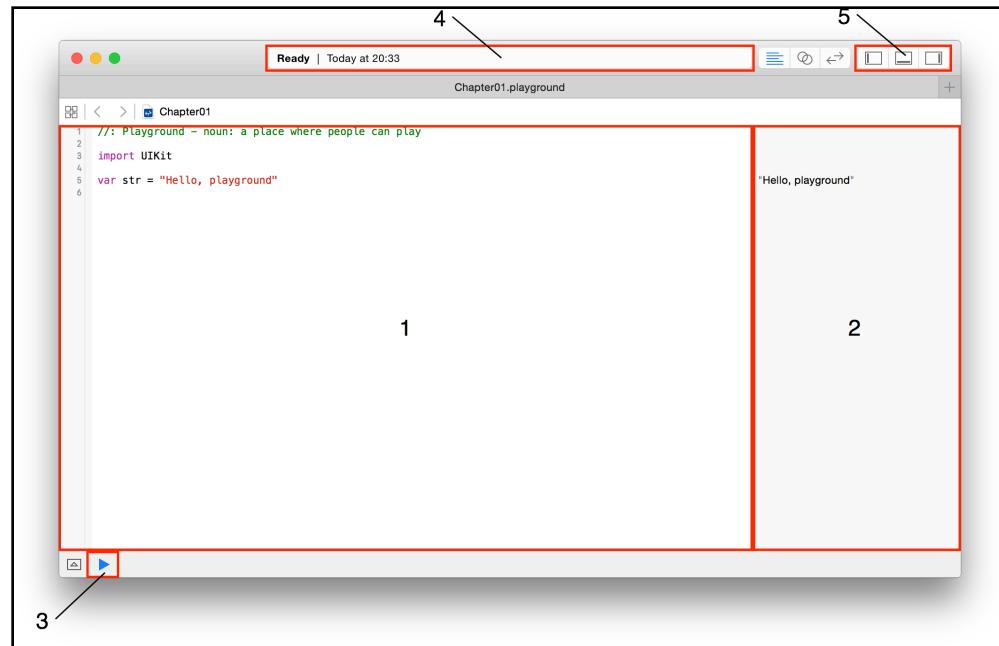
```
//: Playground - noun: a place where people can play
import UIKit
var str = "Hello, playground"
```

"Hello, playground"

New playgrounds don't start entirely empty, but have some basic starter code to get you going. Don't worry—you'll soon learn what this code means.

Playgrounds overview

At first glance, a playground may look like a rather fancy text editor. Well, here's some news for you: It is essentially just that!



The above screenshot highlights the first and most important things to know about:

1. **Source editor.** This is the area in which you'll write your Swift code. It's much like a text editor such as Notepad orTextEdit. You'll notice the use of what we call a monospace font, meaning all characters are the same width. This makes the code much easier to read.

2. **Results sidebar.** This area shows the results of your code. You'll learn more about how code is executed as you read through the book. The results sidebar will be the main place you'll look to confirm your code is working as expected.
3. **Execution control.** Playgrounds execute automatically by default, meaning you can write code and immediately see the output. This control allows you to execute the playground again. Holding down the button allows you to switch between automatic execution and manual execution modes.
4. **Activity viewer.** This shows the status of the playground. In the screenshot, it shows that the playground has finished executing and is ready to handle more code in the source editor. When the playground is executing, this viewer will indicate that with a spinner.
5. **Panel controls.** These toggle switches show and hide three panels, one that appears on the left, one on the bottom and one on the right. The panels each display extra information that you may need to access from time to time. You'll usually keep them hidden, as they are in the screenshot. You'll learn more about each of these panels as you move through the book.

Playgrounds execute the code in the source editor from top to bottom. Every time you change the code, the playground will re-execute everything. You can also force a re-execution by clicking **Editor\Execute Playground**. Alternatively you can use the execution control.

You can turn on line numbers on the left side of the source editor by clicking **Xcode\Preferences...\Text Editing\Line Numbers**. Line numbers can be very useful when you want to refer to parts of your code.

Once the playground execution is finished, Xcode updates the results sidebar to show the results of the corresponding line in the source editor. You'll see how to interpret the results of your code as you work through the examples in this book.

Key points

- Computers at their most fundamental level perform simple mathematics.
- A programming language allows you to write code, which the compiler converts into instructions that the CPU can execute.
- Computers operate on numbers in base-2 form, otherwise known as binary.
- The editor you use to write Swift code is called Xcode.
- By providing immediate feedback about how code is executing, playgrounds allow you to write and test Swift code quickly and efficiently.

Where to go from here?

If you haven't done so already, open Xcode for yourself and create your very first playground. Give it the name **Chapter02** and use the **iOS** platform. Save it somewhere on your hard drive to end up with an open playground.

Now you're all set to follow along with the next chapter!

Chapter 2: Variables & Constants

By Matt Galloway

Now that you know how computers interpret the code you write and what tools you'll be using to write it, it's time to begin learning about Swift itself.

In this chapter, you'll discover constants, variables, types and tuples, and learn how to declare them, name them and change them. You'll also learn about type inference, one of Swift's most important features and one that will make your coding life a lot easier.

Naming data

At its simplest, computer programming is all about manipulating data. Remember, everything you see on your screen can be reduced to numbers that you send to the CPU. Sometimes you yourself represent and work with this data as various types of numbers, but other times the data comes in more complex forms such as text, images and collections.

In your Swift code, you can give each piece of data a name you can use to refer to it later. The name carries with it an associated **type** that denotes what sort of data the name refers to: text, numbers, a date, etc.

You'll learn about some of the basic types in this chapter, and many others throughout the rest of this book.

Constants

Take a look at your first bit of Swift for this book:

```
let number: Int = 10
```

This declares a constant called `number` which is of type `Int`. Then it sets the value of the constant to the number `10`.

The type `Int` can store integers—that is, whole numbers. It would be rather limiting to be able to store only whole numbers, so there's also a way to store decimals. For example:

```
let pi: Double = 3.14159
```

This is similar to the `Int` constant, except the name and the type are different. This time the constant is a `Double`, a type that can store decimals with high precision.

There's also a type called `Float`, short for floating point, that stores decimals with lower precision than `Double`. A `Float` takes up less memory than a `Double` but generally, memory use for numbers isn't a huge issue and you'll see `Double` used in most places.

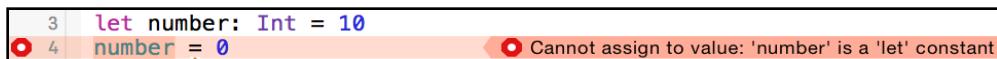
Once you've declared a constant, you can't change its data. For example, consider the following code:

```
let number: Int = 10  
number = 0
```

This code produces an error:

```
Cannot assign to value: 'number' is a 'let' constant
```

In Xcode, you would see the error represented this way:



```
3 let number: Int = 10  
4 number = 0
```

Cannot assign to value: 'number' is a 'let' constant

Constants are useful for values that aren't going to change. For example, if you were modeling an airplane and needed to keep track of the total number of seats available, you could use a constant.

You might even use a constant for something like a person's age. Even though their age will change as their birthday comes, you might only be concerned with their age at this particular instant.

Variables

Often you want to change the data behind a name. For example, if you were keeping track of your bank account balance with deposits and withdrawals, you might use a variable rather than a constant.

If your program's data never changed, then it would be a rather boring program! But as you've seen, it's not possible to change the data behind a constant.

When you know you'll need to change some data, you should use a variable to represent that data instead of a constant. You declare a variable in a similar way, like so:

```
var variableNumber: Int = 42
```

Only the first part of the statement is different: You declare constants using `let`, whereas you declare variables using `var`.

Once you've declared a variable, you're free to change it to whatever you wish, as long as the type remains the same. For example, to change the variable declared above, you could do this:

```
var variableNumber: Int = 42
variableNumber = 0
variableNumber = 1_000_000
```

To change a variable, you simply set it equal to a new value.

Note: In Swift, you can optionally use underscores to make larger numbers more human-readable. The quantity and placement of the underscores is up to you.

This is a good time to introduce the results sidebar of the playground. When you type the code above into a playground, you'll see that the results sidebar on the right shows the current value of `variableNumber` at each line:

8	<code>var variableNumber: Int = 42</code>	42
9	<code>variableNumber = 0</code>	0
10	<code>variableNumber = 1_000_000</code>	1,000,000

The results sidebar will show a relevant result for each line, if one exists. In the case of a variable or constant, the result will be the new value, whether you've just declared a constant, or declared or reassigned a variable.

Naming

Always try to choose meaningful names for your variables and constants.

A good name *specifically* describes what the variable or constant represents. Here are some examples of good names:

- personAge
- numberOfRows
- gradePointAverage

Often a bad name is simply not descriptive enough. Here are some examples of bad names:

- a
- temp
- average

The key is to ensure that you'll understand what the variable or constant refers to when you read it again later. Don't make the mistake of thinking you have an infallible memory! It's common in computer programming to look back at code as early as a day or two later and have forgotten what it does. Make it easier for yourself by giving your variables and constants intuitive, precise names.

In Swift, you can even use the full range of Unicode characters. This means you could, for example, declare a variable like so:

```
var 🐶💩: Int = -1
```

That might make you laugh, but use caution with special characters like these. They are harder to type and therefore may end up causing you more pain than amusement.



Special characters like these probably make more sense in *data* that you store rather than in Swift code; you'll learn more about Unicode in Chapter 4, "Strings."

Type conversion

Sometimes you'll have data in one format and need to convert it to another. The naïve way to attempt this would be like so:

```
var integer: Int = 100
var decimal: Double = 12.5
integer = decimal
```

Swift will complain if you try to do this and spit out an error on the third line:

```
Cannot assign a value of type 'Double' to a value of type 'Int'
```

Some programming languages aren't as strict and will perform simple numeric conversions like this automatically. Swift, however, is very strict about the types it uses and won't let you assign a value of one type to a variable of another type.

Remember, computers rely on us programmers to tell them what to do. In Swift, that includes being explicit about type conversions. If you want the conversion to happen, you have to say so!

Instead of simply assigning, you need to explicitly say that you want to convert the type. You do it like so:

```
var integer: Int = 100
var decimal: Double = 12.5
integer = Int(decimal)
```

The assignment on the third line now tells Swift unequivocally that you want to convert from the original type, Double, to the new type, Int.

Note: In this case, assigning the decimal value to the integer results in a loss of precision: The integer variable ends up with the value 12 instead of 12.5. This is why it's important to be explicit. Swift wants to make sure you know what you're doing and that you may end up losing data by performing the type conversion.

Mini-exercises

If you haven't been following along with the code in Xcode, now's the time to create a new playground and try some exercises to test yourself!

1. Declare a constant of type Int called `myAge` and set it to your age.
2. Declare a variable of type Double called `averageAge`. Initially, set it to `myAge`; you'll need to use an explicit type conversion to do this. Then, set it to the average of your age and my own age of 30. Just do the calculation for that in

your head—you'll get on to doing the calculation in Swift in the following chapter!

Tuples

Sometimes data comes in pairs or triplets. An example of this is a pair of (x, y) coordinates on a 2D grid. Similarly, a set of coordinates on a 3D grid is comprised of an x-value, a y-value and a z-value.

In Swift, you can represent such related data in a very simple way through the use of a *tuple*.

A tuple is a type that represents data composed of more than one value of any type. You can have as many values in your tuple as you like. For example, you can define a pair of 2D coordinates where each axis value is an integer, like so:

```
let coordinates: (Int, Int) = (2, 3)
```

The type of coordinates is a tuple containing two Int values. The types of the values within the tuple, in this case Int, are separated by commas surrounded by parentheses. The code for creating the tuple is much the same, with each value separated by commas and surrounded by parentheses.

You could similarly create a tuple of Double values, like so:

```
let coordinates: (Double, Double) = (2.1, 3.5)
```

Or you could mix and match the types comprising the tuple, like so:

```
let coordinates: (Double, Int) = (2.1, 3)
```

And here's how to access the data inside a tuple:

```
let coordinates: (Int, Int) = (2, 3)
let x: Int = coordinates.0
let y: Int = coordinates.1
```

You can reference each item in the tuple by its position in the tuple, starting with zero. So in this example, x will equal 2 and y will equal 3.

Note: Starting with zero is a common practice in computer programming and is called **zero indexing**. You'll see this again in Chapter 10, "Arrays".

In the previous example, it may not be immediately obvious that the first value, at index 0, is the x-coordinate and the second value, at index 1, is the y-coordinate. This is another demonstration of why it's important to *always* name your variables in a way that avoids confusion.

Fortunately, Swift allows you to name the individual parts of a tuple, so you to be explicit about what each part represents. For example:

```
let coordinatesNamed: (x: Int, y: Int) = (2, 3)
```

Here, the code annotates the type of `coordinatesNamed` to contain a label for each part of the tuple.

Then, when you need to access each part of the tuple, you can access it by its name:

```
let x: Int = coordinatesNamed.x
let y: Int = coordinatesNamed.y
```

This is much clearer and easier to understand. More often than not, it's helpful to name the components of your tuples.

If you want to access multiple parts of the tuple at the same time, as in the examples above, you can also use a shorthand syntax to make it easier:

```
let coordinates3D: (x: Int, y: Int, z: Int) = (2, 3, 1)
let (x, y, z) = coordinates3D
```

This declares three new constants, `x`, `y` and `z`, and assigns each part of the tuple to them in turn. The code is equivalent to the following:

```
let coordinates3D: (x: Int, y: Int, z: Int) = (2, 3, 1)
let x = coordinates3D.x
let y = coordinates3D.y
let z = coordinates3D.z
```

If you want to ignore a certain element of the tuple, you can replace the corresponding part of the declaration with an underscore. For example, if you were performing a 2D calculation and wanted to ignore the `z`-coordinate of `coordinates3D`, then you'd write the following:

```
let (x, y, _) = coordinates3D
```

This line of code only declares `x` and `y`. The `_` is special and simply means you're ignoring this part for now.

Note: You'll find that you can use the underscore throughout Swift to ignore a value.

Mini-exercises

1. Declare a constant tuple that contains three `Int` values followed by a `Double`. Use this to represent a date (month, day, year) followed by an average temperature for that date.

2. Change the tuple to name the constituent components. Give them names related to the data that they contain: month, day, year and averageTemperature.
3. In one line, read the day and average temperature values into two constants. You'll need to employ the underscore to ignore the month and year.
4. Up until now, you've only seen constant tuples. But you can create variable tuples, too. Change the tuple you created in the exercises above to a variable by using var instead of let. Now change the average temperature to a new value.

Type inference

Every time you've seen a variable or constant declared, there's been an associated type. That is, until you saw the syntax for reading multiple values from a tuple at the same time. Take a look at it again:

```
let coordinates3D: (x: Int, y: Int, z: Int) = (2, 3, 1)
let (x, y, z) = coordinates3D
```

There's nothing on the second line to indicate that x, y and z are of type Int. So how does Swift know that they are of type Int?

Reading the code yourself, it's clear that the constants created on the second line are of type Int. You know that because the code declares a tuple comprising three Int values.

It turns out the Swift compiler can do this process of deduction, as well. It doesn't need you to tell it the type—it can figure it out on its own. Neat! :]

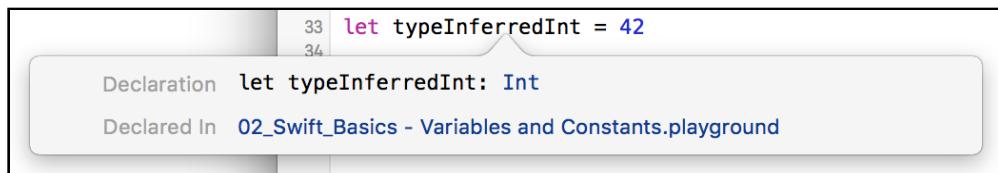
As it happens, you can drop the type in other places besides tuples. In fact, in most places, you can omit the type because Swift already knows it.

For example, consider the following constant declaration:

```
let typeInferredInt = 42
```

The value on the right-hand side is an integer type. Swift knows this, too, and therefore it *infers* that the type of the constant should be an Int. This process is known as **type inference**, and it's a key component of Swift's power as a language.

Sometimes it's useful to check the inferred type of a variable or constant. You can do this in a playground by holding down the **Option** key and clicking on the variable or constant's name. Xcode will display a popover like this:

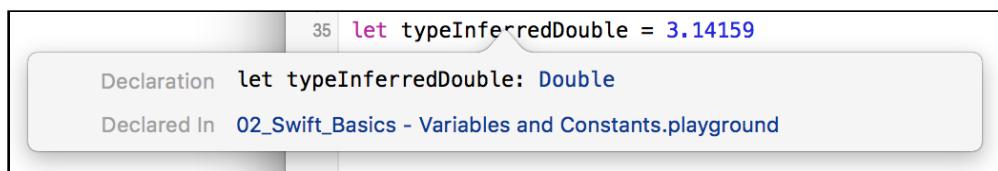


Xcode tells you the inferred type by giving you the declaration you would have had to use if there were no type inference. In this case, the type is `Int`.

It works for other types, too:

```
let typeInferredDouble = 3.14159
```

Option-clicking on this reveals the following:



You can see from this that type inference isn't magic. Swift is simply doing what your human brain does very easily. Programming languages that don't use type inference can often feel verbose, because you need to specify the often obvious type each time you declare a variable or constant.

Note: In later chapters, you'll learn about more complex types where sometimes Swift can't infer the type. That's a pretty rare case though and you'll see type inference used for most of the code examples in this book except in cases where we want to highlight the type for you.

Mini-exercises

1. Create a constant tuple containing two integers but omit the type. Use Option-click to check that the type is indeed `(Int, Int)`.
2. Read the first value of the tuple you created above into another constant. Again, omit the type from the constant. Use Option-click again to check that the type of this constant is `Int`.

Key points

- Constants and variables give names to data.
- Once you've declared a constant, you can't change its data, but you can change a variable's data at any time.
- Always give variables and constants meaningful names to save yourself and your

colleagues headaches later.

- You can use tuples to group data into a single data type.
- Type inference allows you to omit the type when Swift already knows it.

Where to go from here?

Now that you know about variables and constants, it's time to focus on the data that you're storing there! In the next two chapters, you'll learn about some of the most common data types—strings and numbers—and the operations you can do with them.

Challenges

Before moving on, here are some challenges to test your knowledge of variables and constants. You can try the code in a playground to check your answers.

Challenge A: You be the compiler

Which of the following two sections of code is valid?

```
// 1
var age = 25
age = 30

// 2
let age = 25
age = 30
```

Is this valid code?

```
let tuple: (day: Int, month: Int, year: Int) = (15, 8, 2015)
let day = tuple.Day
```

Challenge B: Predict the outcome

What is the type of the constant called value?

```
let tuple = (100, 1.5, 10)
let value = tuple.1
```

What is the value of the constant called month?

```
let tuple: (day: Int, month: Int, year: Int) = (15, 8, 2015)
let month = tuple.month
```

Chapter 3: Numeric Types & Operations

By Matt Galloway

Now that you know how to declare variables and constants, it's time to learn how to use operations to manipulate your data. What use is data if you can't do anything with it?

Manipulating numbers with addition, subtraction and other simple operations is extremely important for all computer programming. After all, as you learned in the first chapter, everything a computer does stems from arithmetic.

In this chapter, you'll learn how to use Swift to perform basic arithmetic on numbers. You'll also learn about **Booleans**, which represent true and false values, and how you can use these to compare data.

Arithmetic operations

When you take one or more pieces of data and turn them into another piece of data, this is known as an **operation**.

The simplest way to understand operations is to think about arithmetic. The addition operation takes two numbers and converts them into the sum of the two numbers. The subtraction operation takes two numbers and converts them into the difference of the two numbers.

You'll find simple arithmetic all over your apps; from tallying the number of "likes" on a post, to calculating the correct size and position of a button or a window, numbers are indeed everywhere!

In this section, you'll learn about the various arithmetic operations that Swift has to offer by considering how they apply to numbers. In later chapters, you see operations for types other than numbers.

Simple operations

All operations in Swift use a symbol known as the **operator** to denote the type of operation they perform. In fact, you've already seen one operator: the equals sign, `=`, known as the **assignment operator**, which you've used to assign a value to a variable or constant.

Consider the four arithmetic operations you learned in your early school days: addition, subtraction, multiplication and division. For these simple operations, Swift uses the following operators:

- Add: `+`
- Subtract: `-`
- Multiple: `*`
- Divide: `/`

These operators are used like so:

```
let add = 2 + 6  
let subtract = 10 - 2  
let multiply = 2 * 4  
let divide = 24 / 3
```

Each of these constants ends up with the same value: 8. You write the code to perform these operations much as you would write it if you were using pen and paper.



If you want, you can remove the white space surrounding the operator, like so:

```
let add = 2+6
```

But it's often easier to read if you have white space on either side of the operator.

You can also perform operations on variables and constants themselves. For example:

```
let two = 2
let six = 6
let add = two + six
```

Or you can mix and match, like so:

```
var add = 2
add = add + 6
```

These four operations are easy to understand because you've been doing them for most of your life. Swift also has more complex operations you can use, all of them standard mathematical operations, just less common ones. Let's turn to them now.

The modulo operation

The first of these is the **modulo** operation. In division, the denominator goes into the numerator a whole number of times, plus a remainder. That's what the modulo operation gives: the remainder. For example, 10 modulo 3 equals 1, because 3 goes into 10 three times, with a remainder of 1.

In Swift, the modulo operator is the % symbol, and you use it like so:

```
let modulo = 28 % 10
```

In this case, modulo equals 8, because 10 goes into 28 twice with a remainder of 8.

In Swift, you can even use the modulo operator with fractional numbers, like so:

```
let moduloDecimal = 11.6 % 1.2
```

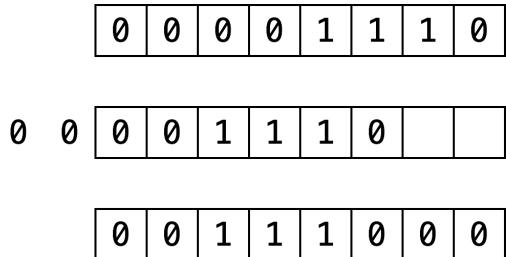
In this case, moduloDecimal equals 0.8, because 1.2 goes into 11.6 nine times with a remainder of 0.8. It can be a little tricky to wrap your head around modulo with fractional numbers, so take some time to think about it if it doesn't make immediate sense to you.

Shift operations

The **shift left** and **shift right** operations take the binary form of a decimal number and shift the digits left or right, respectively. Then they return the decimal form of the new binary number.

For example, the decimal number 14 in binary, padded to 8 digits, is `00001110`. Shifting this left by two places results in `00111000`, which is 56 in decimal.

Here's an illustration of what happens during this shift operation:



The digits that come in to fill the empty spots on the right become 0. The digits that fall off the end on the left are lost.

Shifting right is the same, but the digits move to the right.

The operators for these two operations are as follows:

- Shift left: <<
- Shift right: >>

These are the first operators you've seen that contain more than one character. Operators can contain any number of characters, in fact.

Here's an example that uses both of these operators:

```
let shiftLeft = 1 << 3
let shiftRight = 32 >> 2
```

Both of these values equal the number 8.

One reason for using shifts is to make multiplying or dividing by powers of two easy. Notice that shifting left by one is the same as multiplying by two, shifting left by two is the same as multiplying by four, and so on. Likewise, shifting right by one is the same as dividing by two, shifting right by two is the same as dividing by four, and so on.

In the old days, code often made use of this trick because shifting bits is much simpler for a CPU to do than complex multiplication and division arithmetic. Therefore the code was quicker if it used shifting. However these days, CPUs are much faster and compilers can even convert multiplication and division by powers of two into shifts for you. So you'll see shifting only for binary twiddling, which you probably won't see unless you become an embedded systems programmer!

Increment and decrement

Swift has two operators that respectively let you increment and decrement a value by one:

- Increment: `++`
- Decrement: `--`

You use them like so:

```
var counter = 0

counter++
// counter = 1

counter--
// counter = 0
```

The counter variable begins as 0. The increment operator sets its value to 1, and then the decrement operator sets its value back to 0.

Both of these operators have another form. Above you see what's known as the **postfix form**, because the `++` or `--` goes after the subject variable. The other form, known as the **prefix form**, puts the `++` or `--` before the subject variable.

To understand the difference between prefix and postfix, consider this example:

```
var start = 8

let prefix = ++start
// prefix = 9
// start = 9

let postfix = start++
// postfix = 9
// start = 10
```

Notice that the code assigns the result of the `++` operator to a constant. Both forms of the operator return a value, as well as increment the variable. The difference between the prefix and postfix forms lies in *what* the operation returns.

The prefix form returns the value of the variable *after* incrementing it, whereas the postfix form returns the value of the variable *before* incrementing it. The comments in the code above make this clear:

- In the prefix case, both the `start` variable and the `prefix` constant equal 9, which is the value of `start` before the operation, plus one.
- In the postfix case, on the other hand, `start` is incremented by 1 and so equals 10, but the `postfix` variable equals 9, which was the value of `start` before the operation.

The difference between prefix and postfix is subtle but important. Mix them up, and you'll find yourself with code that doesn't behave as you expect, until you realize your mistake!

There's also a way to add or subtract by more than one:

```
var counter = 0  
  
counter += 5  
// counter = 5  
  
counter -= 2  
// counter = 3
```

These operators are similar to the assignment operator (=), except they also perform an addition or subtraction. They take the current value of the variable, add or subtract the given value and assign the result to the variable.

In other words, the code above is shorthand for the following:

```
var counter = 0  
counter = counter + 5  
counter = counter - 2
```

Similarly, the *= and /= operators do the equivalent for multiplication and division, respectively:

```
var counter = 10  
counter *= 3 // same as counter = counter * 3  
// counter = 30  
  
counter /= 2 // same as counter = counter / 2  
// counter = 15
```

Order of operations in Swift

Of course, it's likely that when you calculate a value, you'll want to use multiple operators. Here's an example of how to do this in Swift:

```
let answer = ((8_000 / (5 * 10)) - 32) >> (29 % 5)
```

Notice the use of parentheses, which in Swift serve two purposes: to make it clear to anyone reading the code—including yourself—what you meant, and to disambiguate. For example, consider the following:

```
let answer = 350 / 5 + 2
```

Does this equal 72 (350 divided by 5, plus 2) or 50 (350 divided by 7)? Those of you who paid attention in school will be screaming "72!" And you would be right!

Swift uses the same reasoning and achieves this through what's known as **operator precedence**. The division operator (/) has a higher precedence than the addition operator (+), so in this example, the code executes the division operation first.

If you wanted Swift to do the addition first—that is, to return 50—then you could

use parentheses like so:

```
let answer = 350 / (5 + 2)
```

The precedence rules follow the same that you learned in math at school. Multiply and divide have the same precedence, higher than add and subtract which also have the same precedence.

Operators with mixed types

So far, you've only seen these operators acting on integers or doubles independently. But what if you have an integer that you want to multiply by a double?

You might naïvely attempt to do it like this:

```
let hourlyRate = 19.5
let hoursWorked = 10
let totalCost = hourlyRate * hoursWorked
```

But if you try that, you'll get an error on the final line:

```
Binary operator '*' cannot be applied to operands of type 'Double' and
'Int'
```

This is because in Swift, you can't apply the `*` operator to mixed types. This rule also applies to the other arithmetic operators. It may seem surprising at first, but Swift is being rather helpful.

Swift forces you to be explicit about what you mean when you want an `Int` multiplied by a `Double`, because the result can be only one type. Do you want the result to be an `Int`, converting the `Double` to an `Int` before performing the multiplication? Or do you want the result to be a `Double`, converting the `Int` to a `Double` before performing the multiplication?

In this example, you want the result to be a `Double`. You don't want an `Int`, because in that case, Swift would convert the `hourlyRate` constant into an `Int` to perform the multiplication, rounding it down to 19 and losing the precision of the `Double`.

You need to tell Swift you want it to consider the `hoursWorked` constant to be a `Double`, like so:

```
let hourlyRate = 19.5
let hoursWorked = 10
let totalCost = hourlyRate * Double(hoursWorked)
```

Now, each of the operands will be a `Double` when Swift multiplies them, so `totalCost` will be a `Double`, as well.

Mini-exercises

1. Create a constant called `testNumber` and initialize it with whatever integer you'd like. Next, create another constant called `evenOdd` and set it equal to `testNumber` modulo 2. Now change `testNumber` to various numbers. What do you notice about `evenOdd`?
2. Create a variable called `answer` and initialize it with the value 0. Increment it by 1. Add 10 to it. Multiply it by 10. Then, shift it to the right by 3. After all of these operations, what's the answer?

Comparison operators

Until now, you've seen only the `Int` and `Double` data types. These are useful, but it's time to branch out and learn about another type, one that will let you compare values through the **comparison operators**.

When you perform a comparison, such as looking for the greater of two numbers, the answer is either `true` or `false`. Swift has a data type just for this! It's called a `Bool`, which is short for Boolean, after a rather clever man named George Boole who invented an entire field of mathematics around the concept of true and false.

This is how you use a Boolean in Swift:

```
let yes: Bool = true
let no: Bool = false
```

And because of Swift's type inference, you can leave off the type:

```
let yes = true
let no = false
```

A Boolean can only be either `true` or `false`, denoted by the keywords `true` and `false`. In the code above, you use the keywords to set the initial state of each constant.

Boolean operators

Booleans are commonly used to compare values. For example, you may have two values and you want to know if they're equal: either they are (`true`) or they aren't (`false`).

In Swift, you do this using the **equality operator**, which is denoted by `==`:

```
let doesOneEqualTwo = (1 == 2)
```

Swift infers that `doesOneEqualTwo` is a `Bool`. Clearly 1 does not equal 2, and therefore `doesOneEqualTwo` will be `false`.

Similarly, you can find out if two values are *not* equal using the `!=` operator:

```
let doesOneNotEqualTwo = (1 != 2)
```

This time, the comparison is true because 1 does not equal 2, so `doesOneNotEqualTwo` will be true.

Two more operators let you determine if a value is greater than (`>`) or less than (`<`) another value. You'll likely know these from mathematics:

```
let isOneGreaterThanOrEqualToTwo = (1 > 2)
let isOneLessThanTwo = (1 < 2)
```

And it's not rocket science to work out that `isOneGreaterThanOrEqualToTwo` will equal false and `isOneLessThanTwo` will equal true.

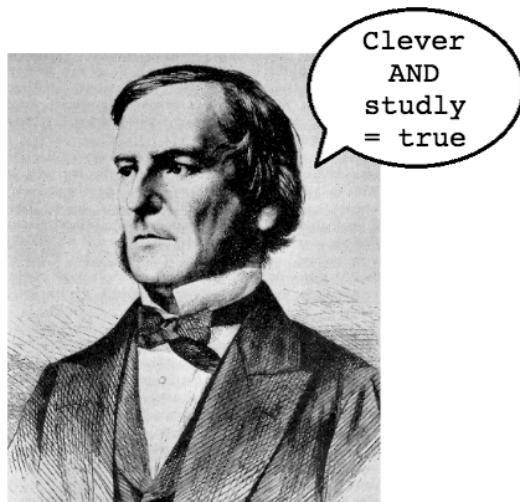
There's also an operator that lets you test if a value is less than *or* equal to another value: `<=`. It's a combination of `<` and `==`, and will therefore return true if the first value is either less than the second value or equal to it.

Similarly, there's an operator that lets you test if a value is greater than or equal to another, and you may have guessed that it's `>=`.

Boolean logic

Each of the examples above tests just one condition. When George Boole invented the Boolean, he had much more planned for it than these humble beginnings. He invented Boolean logic, which lets you combine multiple conditions to form a result.

One way to combine conditions is by using **AND**. When you AND together two Booleans, the result is another Boolean. If both input Booleans are true, then the result is true. Otherwise, the result is false.



George Boole

In Swift, the operator for Boolean AND is `&&`, used like so:

```
let and = true && true
```

In this case, `and` will be true. If either of the values on the right were false, then `and` would be false.

Another way to combine conditions is by using **OR**. When you OR together two Booleans, the result is true if either of the input Booleans is true. Only if both input Booleans are false will the result be false.

In Swift, the operator for Boolean OR is `||`, used like so:

```
let or = true || false
```

In this case, `or` will be true. If both values on the right were false, then `or` would be false. If both were true, then `or` would still be true.

In Swift, Boolean logic is usually applied to multiple conditions. Maybe you want to determine if two conditions are true; in that case, you'd use AND. If you only care about whether one of two conditions is true, then you'd use OR.

For example, consider the following code:

```
let andTrue = 1 < 2 && 4 > 3
let andFalse = 1 < 2 && 3 > 4

let orTrue = 1 < 2 || 3 > 4
let orFalse = 1 == 2 || 3 == 4
```

Each of these tests two separate conditions, combining them with either AND or OR.

It's also possible to use Boolean logic to combine more than two comparisons. For example, you can form a complex comparison like so:

```
let andOr = (1 < 2 && 3 > 4) || 1 < 4
```

The parentheses disambiguates the expression. First Swift evaluates the sub-expression inside the parentheses, and then it evaluates the full expression, following these steps:

1. `(1 < 2 && 3 > 4)` `|| 1 < 4`
2. `(true && false)` `|| true`
3. `false` `|| true`
4. `true`

Mini-exercises

1. Create a constant called `myAge` and set it to your age. Then, create a constant called `isTeenager` that uses Boolean logic to determine if the age denotes a

- teenager, or someone in the age range of 13 to 19.
2. Create another constant called `theirAge` and set it to my age, which is 30. Then, create a constant called `bothTeenagers` that uses Boolean logic to determine if both you and I are teenagers.

Key points

1. The arithmetic operators are:

```
Add: +
Subtract: -
Multiply: *
Divide: /
Modulo: %
```

2. You can combine arithmetic operators with the assignment operator, `=`, to perform arithmetic on a variable and then store the result back in the variable.
3. Operators to increment or decrement a value are:

```
Increment: ++
Decrement: --
```

4. You use the Boolean data type, `Bool`, to represent true and false.
5. The comparison operators, all of which return a Boolean, are:

```
Equal: ==
Not equal: !=
Less than: <
Greater than: >
Less than or equal: <=
Greater than or equal: >=
```

6. You can use Boolean logic to combine comparison conditions.

Where to go from here?

You'll find numbers and especially Boolean logic all throughout the rest of this book, and as you work on code for your apps. In the next chapter, you'll add another data type to your Swift arsenal: strings, to store text. You'll find that working with text has many similarities with working with numbers. It's just another type with operations, after all!

Give the following challenges a try to solidify your knowledge of numeric types, and you'll be ready to move on!

Challenges

Challenge A: Predict the outcome

Consider the following code:

```
let a = 46
var b = 10

let answer = ...
```

Work out what answer equals when you replace the final line of code above with each of these options:

```
// 1
let answer = (a * 100) + b

// 2
let answer = (a * 100) + (b++)

// 3
let answer = (a * 100) + (++b)
```

Challenge B: Boolean logic

In each of the following statements, what is the value of the Boolean answer constant?

```
let answer = true && true
let answer = false || false
let answer = (true && 1 != 2) || (4 > 3 && 100 < 1)
let answer = ((10 / 2) > 3) && ((10 % 2) == 0)
```

Chapter 4: Strings

By Matt Galloway

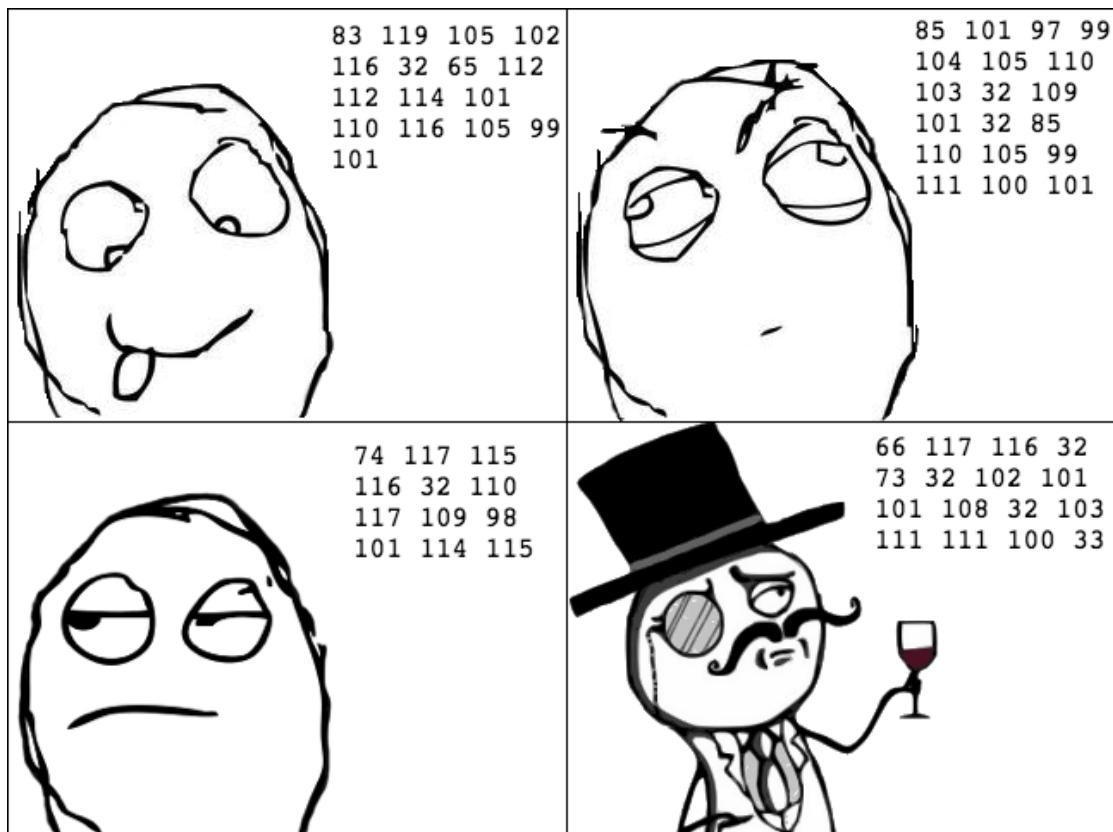
Numbers are essential in programming, but they aren't the only type of data you need to work with in your apps. Text is also an extremely common data type—people's names; their addresses; the words of a book. All of these are examples of text that an app might need to handle.

Most computer programming languages store text in a data type called a **string**. This chapter introduces you to strings, first by giving you background on the concept of strings and then by showing you how to use them in Swift.

How computers represent strings

Computers think of strings as a collection of individual **characters**. In Chapter 1 of this book, you learned that numbers are the language of CPUs, and all code, in whatever programming language, can be reduced to raw numbers. Strings are no different!

That may sound very strange. How can characters be numbers? At its base, a computer needs to be able to translate a character into the computer's own language, and it does so by assigning each character a different number. This forms a two-way mapping from character to number that is called a **character set**.



When you press a character key on your keyboard, you are actually communicating the number of the character to the computer. Your word processor application converts that number into a picture of the character and finally, presents that picture to you.

Unicode

In isolation, a computer is free to choose whatever character set mapping it likes. If the computer wants the letter **a** to equal the number 10, then so be it. But when computers start talking to each other, they need to use a common character set. If two computers used different character sets, then when one computer transferred a string to the other, they would end up thinking the strings contained different characters.

For this reason, a standard was born: **Unicode**. It defines the character set mapping that almost all computers use today.

Note: You can read more about Unicode at its official website, <http://unicode.org/>.

As an example, consider the word **cafe**. The Unicode standard tells us that the letters of this word should be mapped to numbers like so:

c	a	f	e
99	97	102	101

The number associated with each character is called a **code point**. So in the example above, **c** uses code point 99, **a** uses code point 97, and so on.

Of course, Unicode is not just for the simple Latin characters used in English, such as **c**, **a**, **f** and **e**. It also lets you map characters from languages around the world. The word **cafe**, as you're probably aware, is derived from French, in which it's written as **caf **. Unicode maps these characters like so:

c	a	f	�
99	97	102	233

And here's an example using Chinese characters (this, according to Google translate, means 'Computer Programming'):

电	脑	编	程
30005	33041	32534	31243

You've probably heard of emoji, which is made up of small pictures you can use in your text. These pictures are, in fact, just normal characters and are also mapped by Unicode. For example:

💩	😊
128169	128512

This is only two characters. The code points for these are very large numbers, but each is still only a single code point. The computer considers these as no different than any other two characters.

Note: The word "emoji" comes from Japanese, where "e" means picture and "moji" means character.

Strings in Swift

Swift, like any good programming language, can work directly with characters and strings. It does so through the data types `Character` and `String`, respectively. In this section, you'll learn about these data types and how to work with them.

Characters and strings

The `Character` data type can store a single character. For example:

```
let characterA: Character = "a"
```

This stores the character `a`. It can hold any character—even an emoji:

```
let characterDog: Character = "🐶"
```

But this data type is designed to hold only single characters. The `String` data type, on the other hand, stores multiple characters. For example:

```
let stringDog: String = "Dog"
```

It's as simple as that! The right-hand side of this expression is what's known as a **string literal**, and it's the Swift syntax for representing a string.

Of course, type inference applies here, as well. If you remove the type in the above declaration, then Swift does the right thing and makes the `stringDog` a `String` constant:

```
let stringDog = "Dog" // it's a String
```

Concatenation

You can do much more than create simple strings. Sometimes you need to manipulate a string, and one common way to do so is to combine it with another string.

In Swift, you do this in a rather simple way, using the addition operator. Just as you can add numbers, you can add strings:

```
var message = "Hello" + " my name is "
let name = "Matt"
message += name // "Hello my name is Matt"
```

You need to declare `message` as a variable rather than a constant, because you want to modify it. You can add string literals together, as in the first line, and you can add string variables or constants together, as in the last line.

It's also possible to add characters to a string. However, Swift's strictness with types means you have to be explicit when doing so, just as you have to be when you work with numbers if one is an `Int` and the other is a `Double`.

To add a character to a string, you do this:

```
let exclamationMark: Character = "!"  
message += String(exclamationMark) // "Hello my name is Matt!"
```

With this code, you explicitly convert the `Character` to a `String` before you add it to `message`.

Interpolation

You can also build up a string by using **interpolation**, which is a special Swift syntax that lets you build a string in a way that's easy to read:

```
let name = "Matt"  
let message = "Hello my name is \(name)!" // "Hello my name is Matt!"
```

As I'm sure you'll agree, this is much more readable than the example from the previous section. It's an extension of the string literal syntax, whereby you replace certain parts of the string with other values. You enclose the value you want to give the string with parentheses preceded by a backslash.

This syntax works in just the same way to build a string from other data types, such as numbers:

```
let oneThird = 1.0 / 3.0  
let oneThirdLongString = "One third is \(oneThird) as a decimal."
```

Here, you use a `Double` in the interpolation. At the end of this code, your `oneThirdLongString` constant will contain the following:

```
One third is 0.3333333333333333 as a decimal.
```

Of course, it would actually take infinite characters to represent one third as a decimal, because it's a repeating decimal. String interpolation with a `Double` gives you no way to control the precision of the resulting string.

This is an unfortunate consequence of using string interpolation; it's simple to use, but offers no ability to customize the output.

Equality and other methods

Sometimes you want to determine if two strings are equal. For example, a children's game of naming an animal in a photo would need to determine if the player answered correctly.

In Swift, you can compare strings using the standard equality operator, `==`, in exactly the same way as you compare numbers. For example:

```
let guess = "dog"  
let dogEqualsCat = guess == "cat"
```

Here, `dogEqualsCat` is a Boolean that in this case equals `false`, because "dog" does not equal "cat". Simple!

Just as with numbers, you can compare not just for equality, but also to determine if one value is greater than or less than another value. For example:

```
let order = "cat" < "dog"
```

This syntax checks if one string comes before another alphabetically. In this case, `order` equals `true` because "cat" comes before "dog".

The subject of equality brings up an interesting feature of Unicode: There are two ways to represent some characters. One example is the é in café, which is an e with an acute accent. You can represent this character with either a single character or with two.

You saw the single character, code point 233, earlier in this chapter. The two-character case is an e on its own followed by an acute accent **combining character**, which is a special character that modifies the previous character.

So you can represent the e with an acute accent by either of these means:

é	e	'
233	101	769

The combination of these two characters in the second diagram forms what is known as a **grapheme cluster**.

Another example of combining characters are the special characters used to change the skin color of certain emojis. For example:

128077	127997



Here, the thumbs up emoji is followed by a skin tone combining character. On platforms that support it, including iOS and Mac OS X, the rendered emoji is a single thumbs up character with the skin tone applied.

Combining characters make equality of strings a little trickier. For example, consider the word **café** written once using the single é character, and once using the combining character, like so:

c	a	f	é
99	97	102	233

c	a	f	e	'
99	97	102	101	769

These two strings are of course logically equal. When they are printed onscreen, they look exactly the same. But they are represented inside the computer in different ways. Many programming languages would consider these strings to be unequal, because those languages work by comparing the characters one by one. Swift, however, considers these strings to be equal by default. Let's see that in action:

```
let stringA = "café"
let stringB = "cafe\u{0301}"

let equal = stringA == stringB
```

Note: In the code above, the acute accent combining character is written using the Unicode shorthand, which is \u followed by the code point in hexadecimal, in braces. You can use this shorthand to write any Unicode character. I had to use it here for the combining character because there's no way to type this character on my keyboard!

In this case, `equal` is true, because the two strings are logically the same.

String comparison in Swift uses a technique known as **canonicalization**. Say that three times fast! Before checking equality, Swift canonicalizes both strings, which means they're converted to use the same special character representation.

It doesn't matter which way it does the canonicalization—using the single character or using the combining character—as long as both strings get converted to the same style. Once the canonicalization is complete, Swift can compare individual characters to check for equality.

The same canonicalization comes into play when considering how many characters are in a certain string. Let's take a look at that:

```
let stringA = "café"
let stringB = "cafe\u{0301}"

stringA.characters.count
stringB.characters.count
```

The last two lines of this code obtain the character counts of `stringA` and `stringB`. This is the syntax for accessing a property on an object, and this is the first time you've seen it in this book. For now, you don't need to understand anything except that these lines obtain the character counts.

Here, both character counts are 4, even though the second string comprises five code points. This is because the code is counting the characters as they would be displayed onscreen—that is, it's counting the number of grapheme clusters.

Finally, Swift lets you easily create uppercase and lowercase strings, which can come in handy. You achieve this like so:

```
let string = "Swift Apprentice is a great book!"

string.uppercaseString
string.lowercaseString
```

This results in the following:

```
SWIFT APPRENTICE IS A GREAT BOOK!
swift apprentice is a great book!
```

Once again, the code uses syntax that accesses a property on the string. You'll learn more about properties in upcoming chapters. For now, just be aware that there are many different ways to manipulate strings in Swift.

Mini-exercises

1. Create a string constant called `firstName` and initialize it to your first name. Also

- create a string constant called `lastName` and initialize it to your last name.
2. Create a string constant called `fullName` by adding the `firstName` and `lastName` constants together, separated by a space.
 3. Using interpolation, create a string constant called `myDetails` that uses the `fullName` constant to create a string introducing yourself. For example, my string would read: "Hello, my name is Matt Galloway.".

Key points

- **Unicode** is the standard for mapping characters to numbers.
- A single mapping in Unicode is called a **code point**.
- The `Character` data type stores single characters.
- The `String` data type stores collections of characters, or strings.
- You can combine strings by using the addition operator.
- You can use **string interpolation** to build a string in-place.
- Swift's use of **canonicalization** ensures that the comparison of strings accounts for combining characters.

Where to go from here?

Numbers, Booleans and strings are the fundamental data types in Swift. In the last few chapters you've been reading all about data types and storing data; now it's time to circle back to the code side of things!

In the next two chapters, you'll learn how to *use* all this data you now know how to store. With control flow and repeating steps, you'll be able to have your code decide when and how often to do something based on the data.

Challenges

Challenge A: You be the compiler

Which of the following are valid statements?

```
let character: Character = "Dog"
let character: Character = "\u26c8"
let string: String = "Dog"
let string: String = "\u26c8 "
```

What is wrong with the following code?

```
let name = "Matt"
name += " Galloway"
```

Challenge B: Predict the outcome

What is the value of the constant called sum?

```
let value = 10
let multiplier = 5
let sum = "\(value) multiplied by \(multiplier) equals \(value *
multiplier)"
```

Chapter 5: Making Decisions

By Matt Galloway

When writing a computer program, you need to be able to tell the computer what to do in different scenarios. For example, a calculator app would need to do one thing if the user tapped the addition button and another thing if the user tapped the subtraction button.

In computer-programming terms, this concept is known as **control flow**. It is so named because the flow of the program is controlled by various methods. In this chapter, you'll learn how to make decisions in your programs by using syntax to control the flow.

The if statement

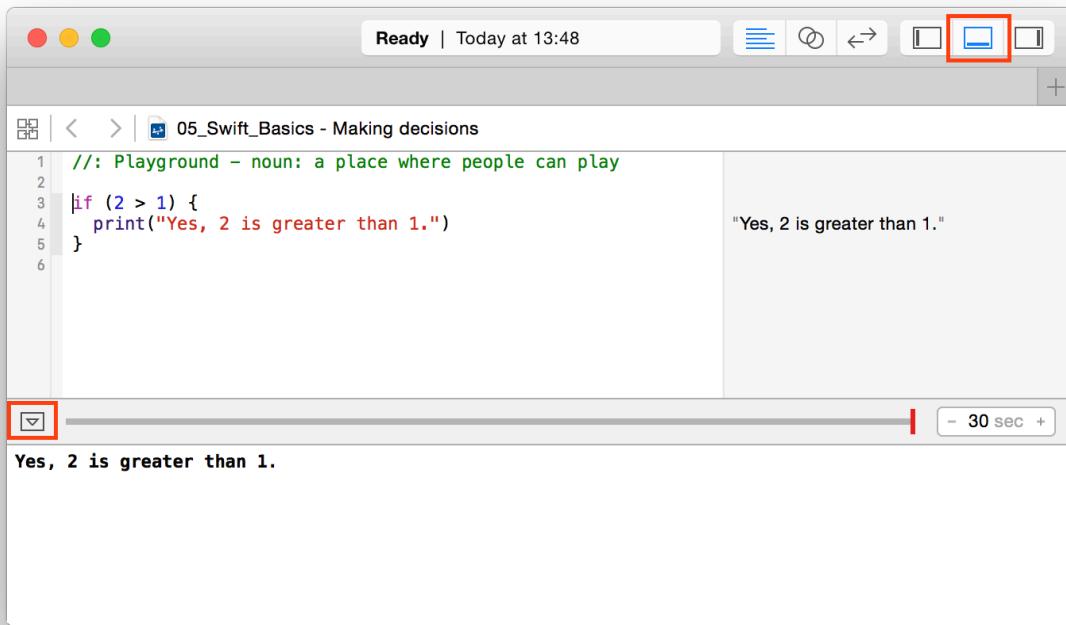
The first and most common way of controlling the flow of a program is through the use of an **if statement**, which allows the program to do something only *if* a certain condition is true. For example, consider the following:

```
if (2 > 1) {  
    print("Yes, 2 is greater than 1.")  
}
```

This is a simple if statement. If the condition inside the parentheses is true, then the statement will execute the code between the braces. If the condition inside the parentheses is false, then the statement won't execute the code between the braces. It's as simple as that!

The code above also introduces you to new syntax: the `print()` function, which simply prints something to the screen. Within your playground, you can see the results of the `print()` in the code above by clicking **View\Debug Area>Show Debug Area**.

You'll see something like this:



It's also possible to show the debug area by clicking on either of the two buttons highlighted in the screenshot above. Xcode has many ways to achieve the same thing!

You can extend an `if` statement to provide code to run in case the condition turns out to be false. This is known as the **else clause**. Here's an example:

```
let animal = "Fox"

if (animal == "Cat" || animal == "Dog") {
    print("Animal is a house pet.")
} else {
    print("Animal is not a house pet.")
}
```

Here, if `animal` equals either "Cat" or "Dog", then the statement will run the first block of code. If `animal` does not equal either "Cat" or "Dog", then the statement will run the block inside the `else` part of the `if` statement, printing the following to the debug area:

```
Animal is not a house pet.
```

But you can go even further than that with `if` statements. Sometimes you want to check one condition, then another. This is where the **else-if** statement comes into play.

You can use it like so:

```
let hourOfDay = 12
let timeOfDay: String

if (hourOfDay < 6) {
    timeOfDay = "Early morning"
} else if (hourOfDay < 12) {
    timeOfDay = "Morning"
} else if (hourOfDay < 17) {
    timeOfDay = "Afternoon"
} else if (hourOfDay < 20) {
    timeOfDay = "Evening"
} else if (hourOfDay < 24) {
    timeOfDay = "Late evening"
} else {
    timeOfDay = "INVALID HOUR!"
}
print(timeOfDay)
```

An else-if statement tests multiple conditions one by one until it finds a true condition. It will only execute the code associated with that first true condition, regardless of whether subsequent else-if conditions are true. In other words, the order of your else- if conditions matters!

You can add else at the end to handle the case where none of the conditions are true. The else is optional if you don't need it; in this example you *do* need it, to ensure that timeOfDay has a valid value by the time you print it out.

In this example, the else-if statement takes a number representing an hour of the day and converts it to a string representing the part of the day to which the hour belongs. Working with a 24-hour clock, the statements are checked one-by-one in order:

- The first check is to see if the hour is less than 6. If so, that means it's early morning.
- If the hour is not less than 6, the statement continues executing to the first else-if, where it checks the hour to see if it's less than 12.
- Then in turn, as conditions prove false, the statement checks the hour to see if it's less than 17, then less than 20, then less than 24.
- Finally, if the hour is out of range, the statement detects an invalid condition and prints that information to the console.

In the code above, the hourOfDay constant is 12. Therefore, the code will print the following:

```
Afternoon
```

Notice that even though both the `hourOfDay < 20` and `hourOfDay < 24` conditions are also true, the statement only executes the first `else-if` that the input satisfies—that is, the `hourOfDay < 17` condition.

Encapsulating variables

If statements introduce a new concept, that of **scope**, which is a way to encapsulate variables through the use of braces.

Let's take an example. Imagine you want to calculate the fee to charge your client. Here's the deal you've made:

You earn \$25 for every hour up to 40 hours, and \$50 for every hour over 50 hours.

Using Swift, you can calculate your fee in this way:

```
var hoursWorked = 45

var price = 0
if (hoursWorked > 40) {
    let hoursOver40 = hoursWorked - 40
    price += hoursOver40 * 50
    hoursWorked -= hoursOver40
}
price += hoursWorked * 25

print(price)
```

This code takes the number of hours and checks if it's over 40. If so, the code calculates the number of hours over 40 and multiplies that by \$50, then adds the result to the price. The code then subtracts the number of hours over 40 from the hours worked. It multiplies the remaining hours worked by \$25 and adds that to the total price.

In the example above, the result is as follows:

1250

The interesting thing here is the code inside the `if` statement. There is a declaration of a new constant, `hoursOver40`, to store the number of hours over 40. Clearly you can use it inside the `if` statement. But what happens if you try to use it at the end of the above code?

```
...
print(price)
print(hoursOver40)
```

This would result in the following error:

```
error: use of unresolved identifier 'hoursOver40'
```

This error informs you that you're only allowed to use the `hoursOver40` constant within the scope in which it was created. In this case, the `if` statement introduced a new scope, so when that scope is finished, you can no longer use the constant.

However, each scope can use variables and constants from its parent scope. In the example above, the scope inside the `if` statement uses the `price` and `hoursWorked` variables, which you created in the parent scope.

The ternary conditional operator

Now I want to introduce a new operator, one you didn't see in Chapter 3. It's called the **ternary conditional operator** and it's related to `if` statements.

If you wanted to determine the minimum and maximum of two variables, you could use `if` statements, like so:

```
let a = 5
let b = 10

let min: Int
if (a < b) {
    min = a
} else {
    min = b
}

let max: Int
if (a > b) {
    max = a
} else {
    max = b
}
```

By now you know how this works, but it's a lot of code. Wouldn't it be nice if you could shrink this to just a couple of lines? Well, you can, thanks to the ternary conditional operator!

The ternary conditional operator takes a condition and returns one of two values, depending on whether the condition was true or false. The syntax is as follows:

```
(<CONDITION>) ? <TRUE VALUE> : <FALSE VALUE>
```

You can use this operator to rewrite your long code block above, like so:

```
let a = 5
let b = 10

let min = a < b ? a : b
let max = a > b ? a : b
```

In the first example, the condition is `a < b`. If this is true, the result assigned back to `min` will be of `a`; if it's false, the result will be the value of `b`.

I'm sure you agree that's much simpler! This is a useful operator that you'll find yourself using regularly.

Note: Because finding the greater or smaller of two numbers is such a common operation, the Swift standard library provides two functions for this purpose: `max` and `min`.

Mini-exercise

Create a constant called `myAge` and initialize it with your age. Write an `if` statement to print out `Teenager` if your age is between 13 and 19, and `Not a teenager` if your age is not between 13 and 19.

Switch statements

Another way to control flow is through the use of a `switch` statement, which lets you execute different bits of code depending on the value of a variable or constant.

Here's a very simple `switch` statement that acts on an integer:

```
let number = 10

switch (number) {
    case 0:
        print("Zero")
    default:
        print("Non-zero")
}
```

In this example, the code will print the following:

```
Non-zero
```

The purpose of this `switch` statement is to determine whether or not a number is zero. It will get more complex—I promise!

To handle a specific case, you use `case` followed by the value you want to check for, which in this case is 0. Then, you use `default` to signify what should happen for all other values.

Here's another example:

```
let number = 10

switch (number) {
    case 10:
        print("It's ten!")
    default:
        break
}
```

This time you check for 10, in which case, you print a message. Nothing should happen for other values. When you want nothing to happen for a case, or you want the `default`, you use the `break` keyword. This tells Swift that you *meant* not to write any code here and that nothing should happen.

Of course, switch statements also work with data types other than integers. They work with any data type! Here's an example of switching on a string:

```
let string = "Dog"

switch (string) {
    case "Cat", "Dog":
        print("Animal is a house pet.")
    default:
        print("Animal is not a house pet.")
}
```

This will print the following:

```
Animal is a house pet.
```

In this example, you provide two values for the `case`, meaning that if the value is equal to either "Cat" or "Dog", then the statement will execute the case.

You can also give your switch statements more than one case. Earlier in this chapter, you saw an `if` statement with `else-if` clauses that could convert an hour of the day to a string describing that part of the day. You could rewrite that more succinctly with a switch statement, like this:

```
let hourOfDay = 12
let timeOfDay: String

switch (hourOfDay) {
    case 0, 1, 2, 3, 4, 5:
        timeOfDay = "Early morning"
    case 6, 7, 8, 9, 10, 11:
```

```
timeOfDay = "Morning"
case 12, 13, 14, 15, 16:
    timeOfDay = "Afternoon"
case 17, 18, 19:
    timeOfDay = "Evening"
case 20, 21, 22, 23:
    timeOfDay = "Late evening"
default:
    timeOfDay = "INVALID HOUR!"
}

print(timeOfDay)
```

This code will print the following:

```
Afternoon
```

When there are multiple cases, the statement will execute the first one that matches. You'll probably agree that this is more succinct and clear than using an `if` statement for this example. It's slightly more precise as well, because the `if` statement method didn't address negative numbers, which here are correctly deemed to be invalid.

It's also possible to match a case to a condition based on a property of the value. As you learned in Chapter 3, you can use the modulo operator to determine if an integer is even or odd. Consider this code:

```
let number = 10

switch (number) {
case let x where x % 2 == 0:
    print("Even")
default:
    print("Odd")
}
```

This will print the following:

```
Even
```

This switch statement uses the **let-where** syntax, meaning the case will match only when a certain condition is true. In this example, you've designed the case to match if the value is even—that is, if the value modulo 2 equals 0.

The method by which you can match values based on conditions is known as **pattern matching**.

Another way you can use switch statements to great effect is as follows:

```
let coordinates: (x: Int, y: Int, z: Int) = (3, 2, 5)

switch (coordinates) {
    case (0, 0, 0): // 1
        print("Origin")
    case (_, 0, 0): // 2
        print("On the x-axis.")
    case (0, _, 0): // 3
        print("On the y-axis.")
    case (0, 0, _): // 4
        print("On the z-axis.")
    default:          // 5
        print("Somewhere in space")
}
```

This switch statement makes use of **partial matching**. Here's what each case does, in order:

1. Matches precisely the case where the value is $(0, 0, 0)$. This is the origin of 3D space.
2. Matches $y=0, z=0$ and any value of x . This means the coordinate is on the x -axis.
3. Matches $x=0, z=0$ and any value of y . This means the coordinate is on the y -axis.
4. Matches $x=0, y=0$ and any value of z . This means the coordinate is on the z -axis.
5. Matches the remainder of coordinates.

You're using the underscore to mean that you don't care about the value. You saw this earlier in chapter 2 and will see it throughout the book in various places. It's pretty handy!

If you don't want to ignore the value, then you can use it in your switch statement, like this:

```
let coordinates: (x: Int, y: Int, z: Int) = (3, 2, 5)

switch (coordinates) {
    case (0, 0, 0):
        print("Origin")
    case (let x, 0, 0):
        print("On the x-axis at x = \(x)")
    case (0, let y, 0):
        print("On the y-axis at y = \(y)")
    case (0, 0, let z):
        print("On the z-axis at z = \(z)")
    case (let x, let y, let z):
        print("Somewhere in space at x = \(x), y = \(y), z = \(z)")
}
```

Here, the axis cases use the `let` syntax to pull out the pertinent values. The code then prints the values using string interpolation to build the string.

Notice how you don't need a default in this `switch` statement. This is because the final case is essentially the default—it matches anything, because there are no constraints on any part of the tuple. If the `switch` statement exhausts all possible values with its cases, then no default is necessary.

Finally, you can use the same `let`-`where` syntax you saw earlier to match more complex cases. For example:

```
let coordinates: (x: Int, y: Int, z: Int) = (3, 2, 5)

switch (coordinates) {
    case (let x, let y, _) where y == x:
        print("Along the y = x line.")
    case (let x, let y, _) where y == x * x:
        print("Along the y = x^2 line.")
    default:
        break
}
```

Here, you match the "y equals x" and "y equals x squared" lines.

And those are the basics of `switch` statements!

Mini-exercises

1. Write a `switch` statement that takes an age as an integer and prints out the life stage related to that age. You can make up the lifestages, or use my categorisation as follows. 0-2 years: Infant, 3-12 years: Child, 13-19 years: Teenager, 20-39: Adult, 40-60: Middle aged, 61+: Elderly.
2. Write a `switch` statement that takes a tuple containing a string and an integer. The string is a name, and the integer is an age. Use the same cases that you used in the previous exercise and `let` syntax to print out the name followed by the lifestage. For example, for myself it would print out "Matt is an adult.".

Key Points

- You use `if` statements to make simple decisions based on a condition.
- You use `else` and `else-if` within an `if` statement to extend the decision-making beyond a single condition.
- You can use the ternary operator in place of simple `if` statements.
- Variables and constants belong to a certain scope, beyond which you cannot use them. A scope inherits visible variables and constants from its parent.

- You use switch statements to decide which code to run depending on the value of a variable or constant.
- The power of a switch statement comes from its ability to use pattern matching, with which values can be compared using complex rules.

Where to go from here?

Apps very rarely run all the way through in the same way every time; depending on data coming in from the Internet or from user input, your code will need to make decisions on which path to take. With if, else and switch statements, you can have your code make decisions on what to do based on some condition.

In a similar way to deciding whether to do something or not, you can also make decisions on *how many times* to do something. In the next chapter, you'll see how to use loops to repeat steps multiple times.

Challenges

Challenge A: You be the compiler

What's wrong with the following code?

```
let firstName = "Matt"

if (firstName == "Matt") {
    let lastName = "Galloway"
} else if (firstName == "Ray") {
    let lastName = "Wenderlich"
}

let fullName = firstName + " " + lastName
```

Challenge B: Predict the outcome

Consider the following switch statement:

```
switch (coordinates) {  
    case (let x, let y, let z) where x == y && y == z:  
        print("x = y = z")  
    case (_, _, 0):  
        print("On the x/y plane")  
    case (_, 0, _):  
        print("On the x/z plane")  
    case (0, _, _):  
        print("On the y/z plane")  
    default:  
        print("Nothing special")  
}
```

What will this code print when coordinates is each of the following?

```
let coordinates = (1, 5, 0)  
let coordinates = (2, 2, 2)  
let coordinates = (3, 0, 1)  
let coordinates = (3, 2, 5)  
let coordinates = (0, 2, 4)
```

6 Chapter 6: Repeating Steps

By Matt Galloway

In the previous chapter, you learned how to control the flow of execution using the decision-making powers of `if` and `switch` statements. In this chapter, you'll continue to learn how to control the flow of execution, this time using loop statements.

Loop statements allow you to perform the same operation multiple times. That may not sound very interesting or important, but loops are very common in computer programs.

For example, you might have code to download an image from the cloud; with a loop, you could run that multiple times to download your entire photo library. Or if you have a game with multiple computer-controlled characters, you might need a loop to go through each one and make sure it knows what to do next.

Let's get started!

Ranges

Before you dive into the loop statements themselves, you need to know about one more data type called a **Range**, which lets you represent a sequence of numbers. Let's look at two types of Range.

First, there's **closed range**, which you represent like so:

```
let closedRange = 0...5
```

The three dots (...) indicate that this range is closed, which means the range goes from 0 to 5, inclusive of both 0 and 5. That's the numbers (0, 1, 2, 3, 4, 5).

Second, there's half-open range, which you represent like so:

```
let halfOpenRange = 0..<5
```

Here, you replace the three dots with two dots and a less-than sign (`..<<`). Half-open means the range goes from 0 to 5, inclusive of 0 but not of 5. That's the numbers `(0, 1, 2, 3, 4)`.

You can use ranges to simplify many operations. For example, consider the `switch` statement from the previous chapter that you used to convert an hour of the day into a string representing that portion of the day. You can rewrite it using ranges like so:

```
let hourOfDay = 12
let timeOfDay: String

switch (hourOfDay) {
case 0...5:
    timeOfDay = "Early morning"
case 6...11:
    timeOfDay = "Morning"
case 12...16:
    timeOfDay = "Afternoon"
case 17...19:
    timeOfDay = "Evening"
case 20..<24:
    timeOfDay = "Late evening"
default:
    timeOfDay = "INVALID HOUR!"
}
```

This is much more succinct than writing out each value individually for all cases.

You need to know about ranges because they're commonly used with loops, the main focus of this chapter—which means that throughout the rest of the chapter, you'll use ranges, too!

Loops

Loops are Swift's way of executing code multiple times. In the following sections, you'll learn about two variants of loops available to you in Swift: `for` loops and `while` loops. If you know another programming language, you'll find the concepts and maybe even the syntax familiar.

For loops

First, let's turn to the **for loop**. This is probably the most common loop you'll see, and you'll use them to run code a certain number of times, incrementing a counter at each stage.

You construct a `for` loop like this:

```
for <INITIAL CONDITION>; <LOOP CONDITION>; <ITERATION CODE> {
    <LOOP CODE>
}
```

The loop begins with the `for` keyword, followed by three expressions:

1. You set up the loop with an initial condition, such as the value of a variable.
2. The loop runs as long as the loop condition is true.
3. At the end of each loop, the loop runs the iteration code.

Here's an example:

```
let count = 10

var sum = 0
for var i = 1; i <= count; i++ {
    sum += i
}
```

In the code above, you set up the loop with a variable called `i` that you initially assign the value of 1; the loop runs until `i` is no longer less than or equal to `count` (that is, until `i` is greater than `count`).

Inside the loop, you add `i` to the `sum` variable, and at the end of each iteration of the loop, you increment `i` by 1.

In terms of scope, the `i` variable is only visible inside the scope of the `for` loop, which means it's not available outside of the loop.

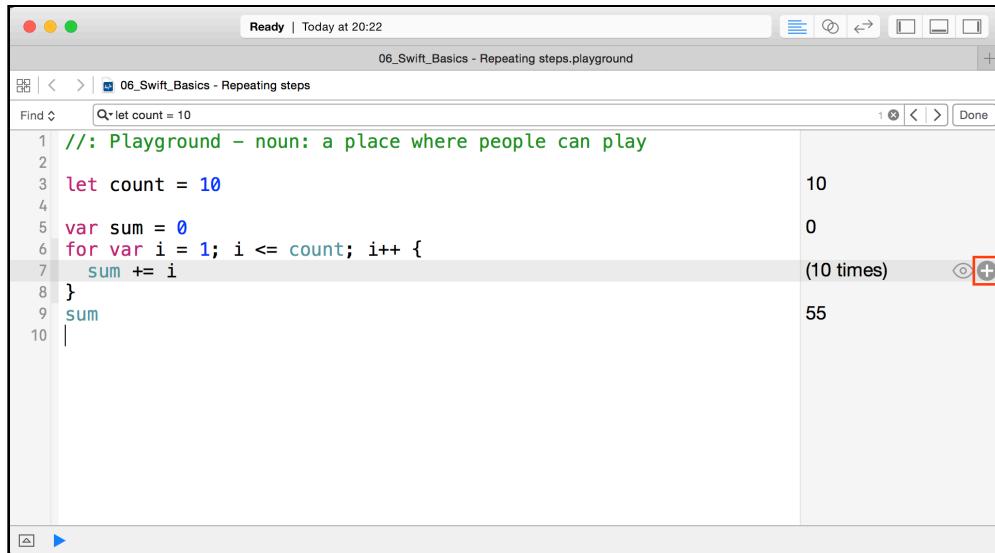
This loop runs 10 times to calculate the sequence $1 + 2 + 3 + 4 + 5 + \dots$ all the way up to 10.

Here are the values of the variables for each iteration:

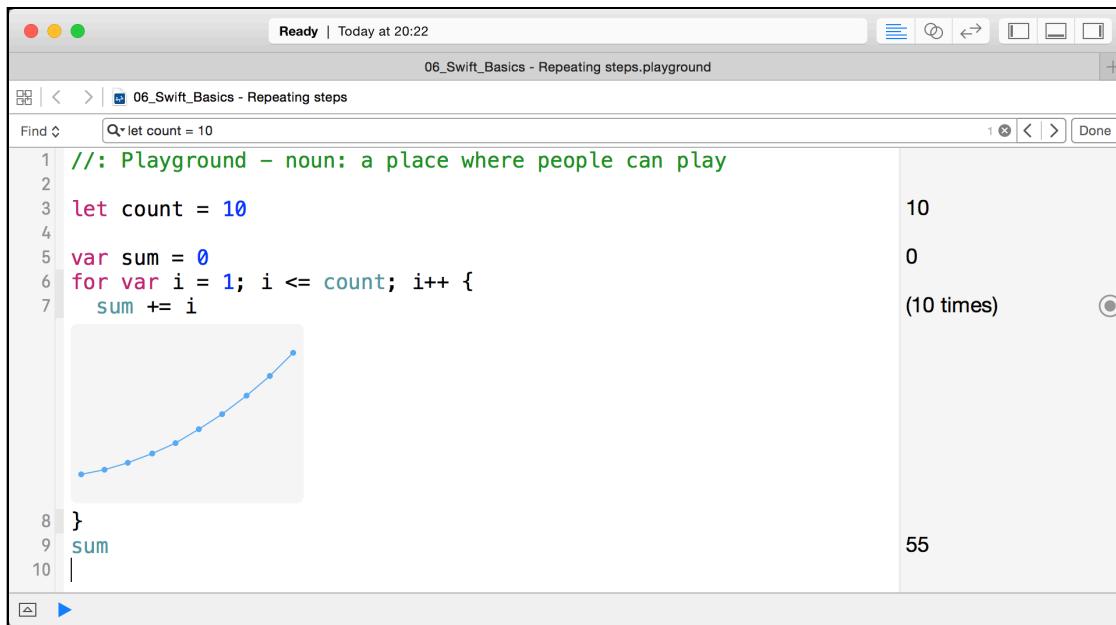
- **Before iteration 1:** `i = 1, sum = 0`
- **After iteration 1:** `i = 2, sum = 1`
- **After iteration 2:** `i = 3, sum = 3`
- **After iteration 3:** `i = 4, sum = 6`
- **After iteration 4:** `i = 5, sum = 10`
- **After iteration 5:** `i = 6, sum = 15`
- **After iteration 6:** `i = 7, sum = 21`
- **After iteration 7:** `i = 8, sum = 28`
- **After iteration 8:** `i = 9, sum = 36`
- **After iteration 9:** `i = 10, sum = 45`
- **After iteration 10:** `i = 11, sum = 55`

Note: If you're mathematically astute, you might notice that this example computes **triangle numbers**. Here's a quick explanation: <http://bbc.in/1O89TGP>

Xcode's playground gives you a handy way to visualize such an iteration. Hover over the `sum += i` line in the results pane, and you'll see a white dot on the right. Hover over that dot to reveal a plus (+) button:



Click this plus (+) button and Xcode will display a graph underneath the line within the playground code editor:



This graph lets you visualise the `sum` variable as the loop iterates.

There's another way to implement the same for loop, and it involves using a special for loop called a **for-in loop**. Instead of having to create a variable to hold the loop counter and increment it yourself, you can iterate through a **range**, like so:

```
let count = 10
var sum = 0

for i in 1...count {
    sum += i
}
```

This code executes in exactly the same way as the previous loop and computes the same number; it's simply a more succinct way of doing so. Because of this, for-in loops are desirable over standard for loops wherever possible.

Finally, sometimes you only want to loop a certain number of times, and so you don't need to use the loop variable at all. In that case, you can employ the underscore once again, like so:

```
let count = 10
var sum = 1
var lastSum = 0

for _ in 0..
```

This code doesn't require the loop variable; the loop simply needs to run a certain number of times. In this case, the range is 0 through count and is half-open. This is the usual way of writing loops that run a certain number of times.

While loops

The next type of loop continues to iterate only while a certain condition is true. Because of this, it's called the **while loop**.

You create a while loop this way:

```
while <CONDITION> {
    <LOOP CODE>
}
```

Every iteration, the loop checks the condition. If the condition is true, then the loop executes and moves on to another iteration. If the condition is false, then the loop stops. Just like for loops and if statements, while loops introduce a scope.

The simplest while loop takes this form:

```
while true {
}
```

This is a while loop that never ends, because the condition is always true. Of course, you would never write such a while loop, because your program would spin forever! This situation is known as an **infinite loop**, and while it might not cause your program to crash, it will very likely cause your computer to freeze.



Here's a more useful example of a while loop:

```
var sum = 1

while sum < 1000 {
    sum = sum + (sum + 1)
}
```

This code calculates a mathematical sequence, up to the point where the value is greater than 1000.

The loop executes as follows:

- **Before iteration 1:** sum = 1, loop condition = true
- **After iteration 1:** sum = 3, loop condition = true
- **After iteration 2:** sum = 7, loop condition = true
- **After iteration 3:** sum = 15, loop condition = true
- **After iteration 4:** sum = 31, loop condition = true
- **After iteration 5:** sum = 63, loop condition = true
- **After iteration 6:** sum = 127, loop condition = true
- **After iteration 7:** sum = 255, loop condition = true
- **After iteration 8:** sum = 511, loop condition = true
- **After iteration 9:** sum = 1023, loop condition = false

After the ninth iteration, the `sum` variable is 1023, and therefore the loop condition of `sum < 1000` becomes false. At this point, the loop stops.

Repeat-while loops

Another variant of the while loop is called the **repeat-while loop**. It differs from the while loop in that the condition is evaluated *at the end* of the loop rather than at the beginning.

You construct a repeat-while loop like this:

```
repeat {  
    <LOOP CODE>  
} while <CONDITION>
```

Here's the example from the last section, but using a repeat-while loop:

```
var sum = 1  
  
repeat {  
    sum = sum + (sum + 1)  
} while sum < 1000
```

In this example, the outcome is the same as before. However, that isn't always the case—you might get a different result with a different condition.

Consider the following while loop:

```
var sum = 1  
  
while sum < 1 {  
    sum = sum + (sum + 1)  
}
```

And now consider the corresponding repeat-while loop, which uses the same condition:

```
var sum = 1  
  
repeat {  
    sum = sum + (sum + 1)  
} while sum < 1
```

In the case of the regular while loop, the condition `sum < 1` is false right from the start. That means the body of the loop won't be reached! The value of `sum` will equal 1, because the loop won't execute any iterations.

In the case of the repeat-while loop, however, `sum` will equal 3 because the loop will execute once.

Breaking out of a loop

Sometimes you want to break out of a loop early. You can do this using the `break` keyword, which immediately stops the execution of the loop and continues on to the code after the loop.

For example, consider the following code:

```
var sum = 1

while true {
    sum = sum + (sum + 1)
    if (sum >= 1000) {
        break
    }
}
```

Here, the loop condition is true, so the loop would normally iterate forever. However, the break means the while loop will exit once the sum is greater than or equal to 1000. Neat!

You've seen how to write the same loop in different ways, demonstrating that in computer programming, there are often many ways to achieve the same result. You should choose the method that's easiest to read and conveys your intent in the best way possible, an approach you'll internalize with enough time and practice.

Note: The break keyword also works in for loops, in exactly the same way it does in while loops.

The continue keyword and labeled statements

Sometimes you want to be able to skip a loop iteration. For example, if you were going through a range but wanted to skip over all odd numbers, you don't want to break out of the loop entirely—you just want to skip the current iteration but let the loop continue.

You can do this by using the continue keyword, which immediately finishes the current iteration of the loop and begins the next iteration.

To demonstrate this, I'll use an example of a chess board that's an 8 by 8 grid, where each cell holds a value of the row multiplied by the column, like a multiplication table:

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	8	10	12	14
3	0	3	6	9	12	15	18	21
4	0	4	8	12	16	20	24	28
5	0	5	10	15	20	25	30	35
6	0	6	12	18	24	30	36	42
7	0	7	14	21	28	35	42	49

The first code example will calculate the sum of all cells, excluding all even rows. To illustrate, it will sum the following cells:

	0	1	2	3	4	5	6	7
0								
1	0	1	2	3	4	5	6	7
2								
3	0	3	6	9	12	15	18	21
4								
5	0	5	10	15	20	25	30	35
6								
7	0	7	14	21	28	35	42	49

Using a `for` loop, you can achieve this as follows:

```
var sum = 0

for row in 0..<8 {
    if row % 2 == 0 {
        continue
    }

    for column in 0..<8 {
        sum += row * column
    }
}
```

When the row modulo 2 equals 0, the row is even. In this case, `continue` makes the `for` loop skip to the next row.

Just like `break`, `continue` works with both `for` loops and `while` loops.

The second code example will calculate the sum of all cells, excluding those where the column is greater than or equal to the row. To illustrate, it will sum the following cells:

	0	1	2	3	4	5	6	7
0								
1	0							
2	0	2						
3	0	3	6					
4	0	4	8	12				
5	0	5	10	15	20			
6	0	6	12	18	24	30		
7	0	7	14	21	28	35	42	

Using a `for` loop, you can achieve this as follows:

```
var sum = 0

rowLoop: for row in 0..<8 {
    columnLoop: for column in 0..<8 {
        if row == column {
            continue rowLoop
        }
        sum += row * column
    }
}
```

This last code block makes use of **labeled statements**, labeling the two loops as the `rowLoop` and the `columnLoop`, respectively. When the row equals the column inside the inner `columnLoop`, the outer `rowLoop` will continue.

You can use labeled statements like these with `break` to break out of a certain loop, if you like. Normally, `break` and `continue` work on the inner-most loop, so you need to use labeled statements if you want to manipulate an outer loop.

Key points

- You can use **ranges** to create a sequence of numbers, incrementing to move from one value to another.
- **Closed ranges** include both the start and end values.
- **Half-open ranges** include the start value and stop one before the end value.
- **Loops** let you execute the same code a number of times.
- The **break** command lets you break out of a loop.
- The **continue** command lets you finish the current iteration of a loop and begin the next iteration.
- **Labeled statements** let you use `break` and `continue` on an outer loop.

Where to go from here?

You've learned about the core language features for dealing with data over these past few chapters—from data types to variables, then on to decision making with Booleans and loops with ranges.

In the next two chapters you'll learn one of the key ways to make your code more reusable and easy to read through the use of functions and closures.

Challenges

Challenge A: Predict the outcome

What will be the value of `sum` and how many iterations will happen?

```
var sum = 0
for i in 0...5 {
    sum += i
}
```

How many instances of the character "a" will there be in `aLotOfAs`?

```
var aLotOfAs = ""
while message.characters.count < 10 {
    aLotOfAs += "a"
}
```

Chapter 7: Functions

By Matt Galloway

Functions are at the core of many programming languages. Simply put, a function lets you define a block of code that performs a given task. Then, whenever your app needs to execute that task, you can run the function instead of having to copy and paste the same code everywhere.

In this chapter, you'll learn how to write your own functions, and see firsthand how Swift makes them easy to use.

Function basics

Say you have an app that often needs to print your name. You can write a function to do this:

```
func printMyName() {  
    print("My name is Matt Galloway.")  
}
```

The code above is known as a **function declaration**. You define a function using the `func` keyword. After that comes the name of the function, followed by parentheses. You'll learn more about the need for these parentheses in the next section.

After the parentheses comes an opening brace, followed by the code you want to run in the function, followed by a closing brace. With your function defined, you can use it like so:

```
printMyName()
```

This prints out the following:

```
My name is Matt Galloway.
```

If you suspect that you've already used a function in previous chapters, you're correct! `print`, which prints the text you give it to the console, is indeed a function.

This leads nicely into the next section, in which you'll learn how to pass data to a function and get data back in return.



Function parameters

In the previous example, the function simply prints out a message. That's great, but sometimes you want to **parameterize** your function, which lets the function perform differently depending on the data passed into it via its **parameters**.

As an example, consider the following function:

```
func printMultipleOfFive(multiplier: Int) {  
    print("\(multiplier) * 5 = \(multiplier * 5)")  
}  
printMultipleOfFive(10)
```

Here, you can see the definition of one parameter inside the parentheses after the function name. It's called `multiplier` and is of type `Int`. In any function, the parentheses contain what's known as the **parameter list**.

This function will print out any given multiple of five. In the example, you call the function with a value of 10, so the function prints the following:

```
10 * 5 = 50
```

You can take this one step further and make the function more general. With two parameters, the function can print out a multiple of any two values. You could achieve this like so:

```
func printMultipleOf(multiplier: Int, andValue: Int) {  
    print("\(multiplier) * \(andValue) = \(multiplier * andValue)")  
}  
printMultipleOf(4, andValue: 2)
```

There are now two parameters inside the parentheses after the function name: one called `multiplier` and the other called `andValue`. Both are of type `Int`.

This time, you pass in two values when you call the function. Notice that the second parameter has a label before it. This syntax is Swift's way of letting you write code that reads like a sentence. In example above, you would read the last line of code like this:

Print multiple of 4 and value 2

You can make this even clearer by giving a parameter a different external name. For example, you can change the external name of the `andValue` parameter:

```
func printMultipleOf(multiplier: Int, and andValue: Int) {  
    print("\(multiplier) * \(andValue) = \(multiplier * andValue)")  
}  
printMultipleOf(4, and: 2)
```

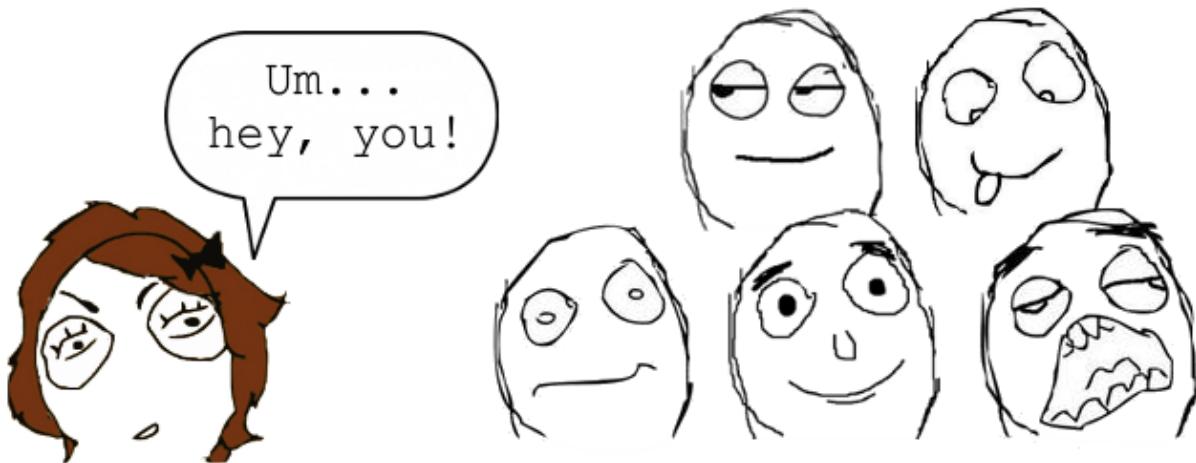
You assign a different external name by writing it in front of the parameter name. In this example, `andValue` is still the name of the parameter, but the label in the function call is now simply `and`. You can read the new call as:

Print multiple of 4 and 2

If you want to have no external name at all, then you can employ the underscore `_`, as you've seen in previous chapters:

```
func printMultipleOf(multiplier: Int, _ andValue: Int) {  
    print("\(multiplier) * \(andValue) = \(multiplier * andValue)")  
}  
printMultipleOf(4, 2)
```

In this example, the second parameter has no external name, just like the first parameter. But use the underscore wisely. Here, your expression is still understandable, but more complex functions that take many parameters can become confusing and unwieldy with no external parameter names. Imagine if a function took five parameters!



You can also give parameters default values:

```
func printMultipleOf(multiplier: Int, andValue: Int = 1) {  
    print("\(multiplier) * \(andValue) = \(multiplier * andValue)")  
}  
printMultipleOf(4)
```

The difference is the `= 1` after the second parameter, which means that if no value is given for the second parameter, it defaults to 1.

Therefore, this code prints the following:

```
4 * 1 = 4
```

It can be useful to have a default value when you expect a parameter to be one value the majority of the time, and it will simplify your code when you call the function.

Return values

All of the functions you've seen so far have performed a simple task, namely, printing out something. Functions can also return a value. The caller of the function can assign the return value to a variable or constant.

This means you can use a function to manipulate data. You simply take in data through parameters, manipulate it and then return it. Here's how you define a function that returns a value:

```
func multiply(number: Int, by byValue: Int) -> Int {  
    return number * byValue  
}  
let result = multiply(4, by: 2)
```

To declare that a function returns a value, between the set of parentheses and before the opening brace, you add a `->` followed by the type of the return value. In

this example, the function returns an Int.

Inside the function, you use the return keyword to return the value. In this example, you return the product of the two parameters.

It's also possible to return multiple values through the use of tuples:

```
func multiplyAndDivide(number: Int, by byValue: Int) -> (multiply: Int,  
divide: Int) {  
    return (number * byValue, number / byValue)  
}  
let result = multiplyAndDivide(4, by: 2)  
let multiply = result.multiply  
let divide = result.divide
```

This function returns both the product and quotient of the two parameters. It does so by defining the function so it returns a tuple containing two Int values with appropriate member value names.

The ability to return multiple values through tuples is one thing that makes it such a pleasure to work with Swift. And it turns out to be a very useful feature, as you'll soon see.

Advanced parameter handling

Parameters passed to functions are constants by default, which means they can't be modified. To illustrate this point, consider the following code:

```
func incrementAndPrint(value: Int) {  
    value++  
    print(value)  
}
```

This results in an error:

```
Cannot pass immutable value to mutating operator: 'value' is a 'let'  
constant
```

The parameter value is the equivalent of a constant declared with let. Therefore, when the function attempts to increment it, the compiler throws the error.

You can change this behavior by making the parameter a variable:

```
func incrementAndPrint(var value: Int) {  
    value++  
    print(value)  
}
```

Now that value is a variable, the function can increment it without a problem.

This function illustrates another interesting point about function parameters. Watch

what happens when you call the above function:

```
var value = 5
incrementAndPrint(value)
print(value)
```

Have a guess at what this code prints out.

It prints the following:

```
6
5
```

This may be unintuitive. You initialize the `value` variable as 5 and pass it to the function, which increments the variable to 6. But when the final line prints `value`, it still equals 5. Why is this?

It's because Swift copies the value before passing it to the function, a behavior known as **pass-by-value**.

Note: Pass-by-value and making copies is the standard behavior for all of the types you've seen so far in this book. You'll see another way for things to be passed into functions in Chapter 14, "Classes".

Usually you want this behavior. Ideally, a function doesn't alter its parameters. If it did, then you couldn't be sure of the parameters' value and you might make incorrect assumptions in your code, leading to the wrong data.

Sometimes you *do* want to let a function change a parameter directly, a behavior called **pass-by-reference**. You do it like so:

```
func incrementAndPrintInOut(inout value: Int) {
    value++
    print(value)
}
```

The `inout` keyword before the parameter name indicates that this parameter should use pass-by-reference.

Now you need to make a slight tweak to the function call:

```
var value = 5
incrementAndPrintInOut(&value)
print(value)
```

You add an ampersand (&) before the parameter, which helps you remember that the parameter is using pass-by-reference. Now the function can change the value however it wishes.

This example will print the following:

6
6

This time, the function successfully increments `value`, which retains its modified data after the function finishes. It's as if the parameter goes *in* to the function and then comes back *out* again, thus the keyword `inout`.

Examples from the standard library

The Swift standard library contains a lot of functions ready for you to use.

You've already seen one used throughout this book: `print`. It takes a single string parameter and prints it to the console.

There are a few math functions worth remembering because they come in handy quite often.

The first is `max`, which takes two values and returns the maximum:

```
let result = max(10, 20)
```

In this example, `result` equals 20.

Similarly, `min` takes two values and returns the minimum:

```
let result = min(10, 20)
```

Here, `result` equals 10.

Finally, `abs` takes a single value and returns the absolute value, which is the non-negative version of the original value. For example, the absolute value of -10 is 10, and the absolute value of 10 is 10. Here's the Swift function:

```
let result = abs(-10)
```

This time, `result` equals 10.

As you continue using Swift, you'll come across many other functions from the standard library. You won't remember all of them, but the important thing to know is that many pre-defined functions exist. If you think there might be one for what you're trying to do, then search for it. There's no point laboring over a function when Apple has written and fully tested it already!



Is there a
function to
design my
art?

Mini-exercises

1. Write a function called `printFullName` that takes two strings called `firstName` and `lastName`. The function should print out the full name defined as `firstName + " " + lastName`. Use it to print out your own full name.
2. Change the declaration of `printFullName` to have no external name for the second parameter, `lastName`.
3. Write a function called `calculateFullName` that returns the full name as a string. Use it to store your own full name in a constant.
4. Change `calculateFullName` to return a tuple containing both the full name and the length of the name. You can find a string's length by using the following syntax: `string.characters.count`. Use this function to determine the length of your own full name.

Functions as variables

This may come as a surprise, but functions in Swift are simply another data type. You can assign them to variables and constants just as you can any other type of value, like an `Int` or a `String`.

To see how this works, consider the following function:

```
func add(a: Int, _ b: Int) -> Int {  
    return a + b  
}
```

This takes two parameters and returns their sum.

You can assign this function to a variable, like so:

```
var function: (Int, Int) -> Int = add
```

Here, the name of the variable is `function` and its type is `(Int, Int) -> Int`. You

assign the add function to this variable.

The type definition is important, though. You define the variable in the same way you write the parameter list and return type in the function declaration. Here, the function variable is a function type that takes two Int parameters and returns an Int.

Now you can use the function variable in just the same way you'd use add, like so:

```
let result = function(4, 2)
```

In this example, result equals 6.

Now consider the following code:

```
func subtract(a: Int, _ b: Int) -> Int {
    return a - b
}
```

Here, you declare another function that takes two Int parameters and returns an Int. You can set the function variable from before to your new subtract function, because the parameter list and return type of subtract are compatible with the type of the function variable.

```
function = subtract
let result = function(4, 2)
```

This time, result equals 2.

The fact that you can assign functions to variables comes in handy because it means you can pass functions to other functions. Here's an example of this in action:

```
func printResult(function: (Int, Int) -> Int, _ a: Int, _ b: Int) {
    let result = function(a, b)
    print(result)
}
printResult(add, 4, 2)
```

printResult takes three parameters:

1. function is a function type that takes two Int parameters and returns an Int, declared like so: (Int, Int) -> Int.
2. a has no external name and is of type Int.
3. b has no external name and is of type Int.

printResult calls the passed-in function, passing into it the two Int parameters. Then it prints the result to the console:

It's extremely useful to be able to pass functions to other functions, and it can help you write reusable code. Not only can you pass data around to manipulate, but passing functions as parameters lets you be flexible about what code gets executed too!

Key points

- You use a **function** to define a task, which you can execute as many times as you like without having to write the code multiple times.
- Functions can take zero or more **parameters** and optionally return a value.
- The first parameter to a function has no label in the function call. All subsequent parameters are labeled with their names.
- You can add an external name to a function parameter to change the label you use in a function call, or you can use an underscore to denote no label.
- You can assign functions to variables and pass them to other functions.

Where to go from here?

Functions are the first step in grouping together small pieces of code into a larger unit. You'll continue with this theme as you learn about collection types and structures over the next chapters and section of this book.

In the next chapter you'll learn about **closures**, which are related to functions but are more lightweight and easier to declare "on the fly" as needed. Before you move on, check out the challenges ahead as you'll need to understand functions before understanding closures!

Challenges

Challenge A: It's prime time

When I'm acquainting myself with a programming language, one of the first things I do is write a function to determine whether or not a number is prime. That is your first challenge.

First, write the following function:

```
func isNumberDivisible(number: Int, by byNumber: Int) -> Bool
```

You'll use this to determine if one number is divisible by another. It should return true when number is divisible by byNumber.

Hint: You can use the modulo (%) operator to determine if a number is divisible by another number: $x \% y = 0$ when x is divisible by y.

Next, write the main function:

```
func isPrime(number: Int) -> Bool
```

This should return true if number is prime, and false otherwise. A number is prime if it's only divisible by 1 and itself. You should loop through the numbers from 1 to the number and find the number's divisors. If it has any divisors other than 1 and itself, then the number isn't prime. You'll need to use the `isNumberDivisible(_:_by:)` function you wrote earlier.

Use this function to check the following cases:

```
isPrime(6) // false  
isPrime(13) // true  
isPrime(8893) // true
```

Hint 1: Numbers less than 0 should be considered not prime. Check for this case at the start of the function and return early if the number is less than 0.

Hint 2: Use a for loop to find divisors. If you start at 2 and end before the number itself, then as soon as you find a divisor, you can return false.

Hint 3: If you want to get *really* clever, you can simply loop from 2 until you reach the square root of number rather than going all the way up to number itself. I'll leave it as an exercise for you to figure out why. It may help to think of the number 16, whose square root is 4. The divisors of 16 are 1, 2, 4, 8 and 16.

Challenge B: Recursive functions

In this challenge, you're going to see what happens when a function calls *itself*, a behavior called **recursion**. This may sound unusual, but it can be quite useful.

You're going to write a function that computes a value from the **Fibonacci sequence**. Any value in the sequence is the sum of the previous two values. The sequence is defined such that the first two values equal 1. That is, `fibonacci(1) = 1` and `fibonacci(2) = 1`.

Write your function using the following declaration:

```
func fibonacci(number: Int) -> Int
```

Then, verify you've written the function correctly by executing it with the following

numbers:

```
fibonacci(1) // = 1
fibonacci(2) // = 1
fibonacci(3) // = 2
fibonacci(4) // = 3
fibonacci(5) // = 5
fibonacci(10) // = 55
```

Hint 1: For values of number less than 0, you should return 0.

Hint 2: To start the sequence, hard-code a return value of 1 when number equals 1 or 2.

Hint 3: For any other value, you'll need to return the sum of calling fibonacci with number - 1 and number - 2.

8 Chapter 8: Closures

By Matt Galloway

The previous chapter was all about functions. But Swift has another object you can use to break up code into reusable chunks, and it's called a **closure**.

Remember how you began with plain values such as 10 and "Hello" and then started assigning them names with variables and constants? Closures and functions are similar—except you began by learning about functions, which are the ones with names.

A closure is simply a function with no name. You can assign them to variables and pass them around like any other value. Being able to declare them without needing to create a fully-formed function makes them more convenient to use and pass around, as you'll see in this chapter.

Closure basics

Closures are so named because they have the ability to "close over" the variables and constants within the closure's own scope. This simply means that if a closure wants to access, store and manipulate the value of any variable or constant from the surrounding context, it can. Variables and constants within the body of a closure are said to have been **captured** by the closure.

You may ask, "If closures are functions without names, then how do you use them?" To use a closure, you must first assign it to a variable or constant. In this way, a closure is a type like any other type.

Here's a declaration for a variable that can hold a closure:

```
var multiplyClosure: (Int, Int) -> Int
```

`multiplyClosure` takes two `Int` values and returns an `Int`. Notice that this is exactly the same as a variable declaration for a function. Like I said, a closure is a function without a name!

You assign a closure to a variable like so:

```
multiplyClosure = { (a: Int, b: Int) -> Int in
    return a * b
}
```

This looks similar to a function declaration, but there's a subtle difference. There's the same parameter list, `->` symbol and return type. But for closures, these appear inside braces, and there is an `in` keyword after the return type.

With your closure variable defined, you can use it just as if it were a function, like so:

```
let result = multiplyClosure(4, 2)
```

As you'd expect, `result` equals 8.

Shorthand syntax

Compared to functions, closures are designed to be lightweight. There are many ways to shorten their syntax. First, you can get rid of the return within a closure, like so:

```
multiplyClosure = { (a: Int, b: Int) ->
    a * b
}
```

When the last line of the closure contains an expression like this, Swift automatically regards it as a return.

You can use Swift's type inference to shorten the syntax even more by removing the return type:

```
multiplyClosure = { (a: Int, b: Int) in
    a * b
}
```

The only difference here is you've removed the `-> Int`. Remember, you already declared `multiplyClosure` as a closure returning an `Int`, so you can let Swift infer this part of the type for you.

Type inference allows us to get even shorter. The types can be removed for the parameters when declaring the closure, like so:

```
multiplyClosure = { (a, b) in
    a * b
}
```

And finally, if you want you can even omit the parameter list. Swift lets you refer to each parameter by number, starting at zero, like so:

```
multiplyClosure = {  
    $0 * $1  
}
```

The parameter list, return type and `in` keyword are all gone, and your new closure declaration is much shorter than the original. Numbered parameters like this should really only be used when the closure is short and sweet, like this one is. If it is much longer then it can be confusing to remember what each numbered parameter refers to, and you should use the named syntax.

Now let's see how useful this can be.



Consider the following code:

```
func operateOnNumbers(a: Int, _ b: Int,  
operation: (Int, Int) -> Int) -> Int {  
    let result = operation(a, b)  
    print(result)  
    return result  
}
```

This declares a function called `operateOnNumbers`, which takes `Int` values as its first two parameters. The third parameter is called `operation`, and it appears to be a function type. `operateOnNumbers` itself returns an `Int`.

You can then use `operateOnNumbers` with a closure, like so:

```
let addClosure = { (a: Int, b: Int) in  
    a + b  
}  
operateOnNumbers(4, 2, operation: addClosure)
```

Remember, closures are simply functions without names. So you shouldn't be surprised to learn that you can also pass in a function as the third parameter of `operateOnNumbers`, like so:

```
func addFunction(a: Int, b: Int) -> Int {
```

```
    return a + b
}
operateOnNumbers(4, 2, operation: addFunction)
```

operateOnNumbers is called the same way, whether the operation is a function or a closure.

However, here the power of the closure syntax comes in handy again. You can define the closure inline with the operateOnNumbers function call, like this:

```
operateOnNumbers(4, 2, operation: { (a: Int, b: Int) -> Int in
    return a + b
})
```

There's no need to define the closure and assign it to a local variable or constant; you can simply declare the closure right where you pass it into the function as a parameter!

But recall that you can simplify the closure syntax to remove a lot of the boilerplate code. You can therefore reduce the above to the following:

```
operateOnNumbers(4, 2, operation: {
    $0 + $1
})
```

There's one more way you can simplify the syntax, but it can only be done when the closure is the final parameter passed to a function. In this case, you can move the closure outside of the function call:

```
operateOnNumbers(4, 2) {
    $0 + $1
}
```

This may look strange, but it's just the same as the previous code snippet, except you've removed the operation label and pulled the braces outside of the function call parameter list. This is called **trailing closure syntax**.

Closures with no return value

Until now, all the closures you've seen have taken one or more parameters and have returned values. But just like functions, closures aren't required to do these things. Here's how you declare a closure that takes no parameters and returns nothing:

```
let voidClosure: () -> Void = {
    print("Swift Apprentice is awesome!")
}
voidClosure()
```

The closure's type is `() -> Void`. The empty parentheses denote there are no parameters. You must declare a return type, so Swift knows you're declaring a

closure. This is where `Void` comes in handy, and it means exactly what its name suggests: the closure returns nothing.

Capturing from the enclosing scope

Finally, let's return to the defining characteristic of a closure: it can access the variables and constants from within its own scope.

Note: Recall that scope defines the range in which an entity (variable, constant, etc) is accessible. You saw a new scope introduced with if-statements. Closures also introduce a new scope and inherit all entities visible to the scope in which it is defined.

For example, take the following closure:

```
var counter = 0
let incrementCounter = {
    counter++
}
```

`incrementCounter` is rather simple: It increments the `counter` variable. The `counter` variable is defined outside of the closure. The closure is able to access the variable because the closure is defined in the same scope as the variable. The closure is said to **capture** the `counter` variable. Any changes it makes to the variable are visible both inside and outside of the closure.

Let's say you call the closure five times, like so:

```
incrementCounter()
incrementCounter()
incrementCounter()
incrementCounter()
incrementCounter()
```

After these five calls, `counter` will equal 5.

The fact that closures can be used to capture variables from the enclosing scope can be extremely useful. For example, you could write this function:

```
func countingClosure() -> (() -> Int) {
    var counter = 0
    let incrementCounter: () -> Int = {
        return counter++
    }
    return incrementCounter
}
```

This function takes no parameters and returns a closure. The closure it returns itself takes no parameters and returns an `Int`.

The closure returned from this function will increment its internal counter each time it is called. Each time you call this function you get a different counter.

For example, this could be used like so:

```
let counter1 = countingClosure()
let counter2 = countingClosure()

counter1() // 0
counter2() // 0
counter1() // 1
counter1() // 2
counter2() // 1
```

The two counters created by the function are mutually exclusive and count independently. Neat!

And that's closures!

Key points

- **Closures** are functions without names. They can be assigned to variables and passed as parameters to functions.
- Closures have **shorthand syntax** that makes them a lot easier to use than other functions.
- A closure can **capture** the variables and constants from its surrounding context.

Where to go from here?

Closures and functions are the fundamental types for storing your code into reusable pieces. Aside from declaring them and calling them, you've also seen how useful they are when passing them around as parameters to *other* functions and closures.

You'll return to dealing with data in the next few chapters, as you learn about optionals and collection types. Still, keep closures and functions in mind and you'll see how to combine everything—variables and constants and functions—together into your own custom types starting in Chapter 13, "Structures".

In the meantime, check out the challenges below to test your knowledge before moving on.

Challenges

Challenge A: Repeating yourself

Your first challenge is to write a function that will run a given closure a given number of times.

Declare the function like so:

```
func repeatTask(times: Int, task: () -> Void)
```

The function should run the task closure, `times` number of times.

Use this function to print "Swift Apprentice is a great book!" 10 times.

Challenge B: Closure sums

In this challenge, you're going to write a function that you can reuse to create different mathematical sums.

Declare the function like so:

```
func mathSum(times: Int, operation: (Int) -> Int) -> Int
```

The first parameter, `times`, defines the number of iterations. The second parameter, `operation`, is a closure that takes an `Int` and returns an `Int`.

`operation` should have a parameter that is the number of the current iteration, and a return value of the number to add to the result for this iteration.

`mathSum` should iterate `times` number of times, starting at 1. It should keep a sum of the `operation` result for each iteration.

Use the function to find the sum of the first 10 square numbers, which equals 386. Then use the function to find the sum of the first 10 Fibonacci numbers, which equals 144. For the Fibonacci numbers, you can use the function you wrote in the previous chapter's challenge—or grab it from the solutions if you didn't do it!

9 Chapter 9: Optionals

By Matt Galloway

All the variables and constants you've dealt with so far have had concrete values. When you had a string variable, like `var name`, it had a string value associated with it, like `"Matt Galloway"`. It could have been an empty string, like `""`, but nevertheless, there was a value to which you could refer.

That's one of the built-in safety features of Swift: if the type says `Int` or `String` then there's an actual integer or string there, guaranteed.

This chapter will introduce you to the concept of **optionals**, a special Swift type that can represent not just a value, but also the absence of a value. By the end of this chapter, you'll know why you need optionals and how to use them safely.

Introducing `nil`

Sometimes, it's useful to have a value with no value. Imagine a scenario where you need to refer to a person's identifying information. You want to store the person's name, age and occupation. Name and age are both things that must have a value—everyone has them. But not everyone is employed, so the absence of a value for occupation is something you need to be able to handle.

Let's take a look at this in practice.

Without knowing about optionals, this is how you might represent the person's name, age and occupation:

```
var name: String = "Matt Galloway"
var age: Int = 30
var occupation: String = "Software Developer & Author"
```

But what if I become unemployed? Maybe I've won the lottery and want to give up work altogether (I wish!). This is when it would be useful to be able to refer to the absence of a value.

Why couldn't you just use an empty string? Well, you could! Let's discuss how that would work, and why optionals are a better solution.

Sentinel values

A valid value that's used to represent an absence is known as a **sentinel value**. That's what your empty string would be in the previous example.



Let's look at another example. Say you write code that requests something from a server, and you use a variable to store any error code that the server returns:

```
var errorCode: Int = 0
```

In the success case, you represent the lack of an error with a zero. That means `0` is the sentinel value.

Just like the empty string for occupation, this works, but it's potentially confusing for the programmer. Maybe `0` is actually a valid error code—or will be in the future. Either way, you can't be completely confident that there's no error.



In these two examples, it would be much better if there were a special type that

could represent the absence of a value. It would then be explicit when a value exists and when one doesn't.

Nil is the name given to the absence of a value, and you're about to see how Swift incorporates this concept directly into the language in a rather elegant way.

Some other programming languages just use sentinel values. Some, like Objective-C, have the concept of `nil`, but it is merely a synonym for zero. It is just another sentinel value.

Swift introduces a whole new type, called an **optional** to handle the possibility that a value is `nil` or not. This means that if you are handling a non-optional type then you are guaranteed to have a value and don't need to worry about if there is a value or not. There always is one. Similarly, if you are using an optional type then you know you must handle the `nil` case. It takes away the ambiguity introduced by using sentinel values.

Introducing optionals

Optionals are Swift's solution to the problem of representing both a value and the absence of a value. An optional type is allowed to reference either a value *or* `nil`.

Think of an optional as a box: it either contains a value, or it doesn't. When it doesn't contain a value, it's said to contain `nil`. The box itself always exists; it's always there for you to open and look inside.



Optional box
containing a
value



Optional box
containing no
value

A string or an integer, on the other hand, doesn't have this box around it. Instead there's always a value, such as "hello" or 42. Remember, non-optional types are guaranteed to have an actual value.

Note: Those of you who've studied physics may be thinking about Schrödinger's cat right now. Optionals are a little bit like that, except it's not a matter of life and death!

You declare an optional type by using the following syntax:

```
var errorCode: Int?
```

The only difference between this and a standard declaration is the question mark at the end of the type. In this case, `errorCode` is an "optional Int". This means the variable itself is like a box containing either an `Int` or `nil`.

Setting the value is simple. You can either set it to an `Int`, like so:

```
errorCode = 100
```

Or you can set it to `nil`, like so:

```
errorCode = nil
```

This diagram may help you visualize what's happening:

errorCode =

errorCode =

The optional box always exists. When you assign `100` to the variable, you're filling the box with the value. When you assign `nil` to the variable, you're emptying the box.

Take a few minutes to think about this concept. The box analogy will be a big help as you go through the rest of the chapter and begin to use optionals.

Mini-exercise

Make an optional `String` called `myFavoriteSong`. If you have a favorite song, set it to a string representing that song. If you have more than one favorite song or no favorite, set the optional to `nil`.

Unwrapping optionals

It's all well and good that optionals exist, but you may be wondering how you can look inside the box and manipulate the value it contains.

Take a look at what happens when you print out the value of an optional:

```
let ageInteger: Int? = 30
print(ageInteger)
```

This prints the following:

```
Optional(30)
```

That isn't really what you want—although if you think about it, it makes sense. Your code has printed the box. The result says, "ageInteger is an optional that contains the value 30".

To see how this value type is different from a non-optional type, let's see what happens if you try to use ageInteger as if it were a normal integer:

```
print(ageInteger + 1)
```

This code triggers an error:

```
error: value of optional type 'Int?' not unwrapped; did you mean to use
'!' or '?'?
```

It doesn't work because you're trying to add an integer to a box—not to the value inside the box, but to the box itself. That doesn't make sense!

Force unwrapping

The error message gives an indication of the solution: It tells you that the optional is "not unwrapped". You need to unwrap the value from its box. It's like Christmas!

Let's see how that works. Consider the following declaration:

```
var authorName: String? = "Matt Galloway"
```

There are two different methods you can use to unwrap this optional. The first is known as **force unwrapping**, and you perform it like so:

```
var unwrappedAuthorName = authorName!
print("Author is \(unwrappedAuthorName)")
```

The exclamation mark after the variable name tells the compiler that you want to look inside the box and take out the value. Here's the result:

```
Author is Matt Galloway
```

Great! That's what you'd expect.

The use of the word "force" and the exclamation mark ! should convey a sense of danger to you, and that's worth noting. You should use force unwrapping sparingly. To see why, consider what happens when the optional doesn't contain a value:

```
authorName = nil
var unwrappedAuthorName = authorName!
print("Author is \(unwrappedAuthorName)")
```

This code produces the following runtime error:

```
fatal error: unexpectedly found nil while unwrapping an Optional value
```

The error happens because the variable contains no value when you try to unwrap it. What's worse is that you get this error at runtime rather than compile time—which means you'd only notice the error if you happened to execute this part of the code with some invalid input. Worse yet, if this code were inside an app, the runtime error would cause the app to crash!

How can you play it safe?

To stop the runtime error here, you could wrap the code that unwraps the optional in a check, like so:

```
if authorName != nil {
    var unwrappedAuthorName = authorName!
    print("Author is \(unwrappedAuthorName)")
} else {
    print("No author.")
}
```

The `if` statement checks if the optional contains `nil`. If it doesn't, that means it contains a value you can unwrap.

The code is now safe, but it's still not perfect. If you rely on this technique, you'll have to remember to check for `nil` every time you want to unwrap an optional. That will start to become tedious, and one day you'll forget and once again end up with the possibility of a runtime error.

Back to the drawing board, then!

If let binding

Fortunately, Swift includes a feature known as **if let binding**, which lets you safely access the value inside an optional. You use it like so:

```
if let unwrappedAuthorName: String = authorName {
    print("Author is \(unwrappedAuthorName)")
} else {
    print("No author.")
}
```

You'll immediately notice that there are no exclamation marks and the type of `unwrappedAuthorName` is a plain `String`, not an optional `String`.

Note: Normally you wouldn't specify the type of the unwrapped variable in an `if let` statement, but it's shown here for clarity.

This special syntax rids the type of the optional. If the optional contains a value, then the code executes the `if` side of the `if` statement, within which you automatically unwrap the `unwrappedAuthorName` variable because you know it's safe to do so.

If the optional doesn't contain a value, then the code executes the `else` side of the `if` statement. In that case, the `unwrappedAuthorName` variable doesn't even exist.

You can see how `if let` binding is much safer than force unwrapping, and you should opt for it whenever possible.

You can even unwrap multiple values at the same time, like so:

```
let authorName: String? = "Matt Galloway"
let authorAge: Int? = 30

if let name: String = authorName,
    age: Int = authorAge {
    print("The author is \(name) who is \(age) years old.")
} else {
    print("No author or no age.")
}
```

This code unwraps two values. It will only execute the `if` part of the statement when both optionals contain a value.

Now you know how to safely look inside an optional and extract its value, if there is one.

Mini-exercises

1. Using your `myFavoriteSong` variable from earlier, use `if let` binding to check if it contains a value. If it does, print out the value. If it doesn't, print "I don't have a favorite song."
2. Change `myFavoriteSong` to the opposite of what it is now; that is, if it's `nil`, set it to a string, and if it's a string, set it to `nil`. Observe how your printed result changes.

Nil coalescing

There's one final and rather handy way to unwrap an optional. You use it when you want to get a value out of the optional no matter what—and in the case of `nil`, you'll use a default value. This is called **nil coalescing**.

Here's how it works:

```
var optionalInt: Int? = 10
var result: Int = optionalInt ?? 0
```

The nil coalescing happens on the second line, with the double question mark (??). This line means result will equal either the value inside optionalInt, or 0 if optionalInt contains nil.

So in this example, result contains the concrete Int value of 10.

The code above is equivalent to the following:

```
var optionalInt: Int? = 10
var result: Int
if let unwrapped = optionalInt {
    result = unwrapped
} else {
    result = 0
}
```

Set the optionalInt to nil, like so:

```
optionalInt = nil
var result: Int = optionalInt ?? 0
```

Now your result equals 0.

Key points

- **Nil** represents the absence of a value.
- Non-optional variables and constants must always have a non-nil value.
- **Optional** variables and constants are like boxes that can contain a value *or* be empty (nil).
- To work with the value inside an optional, you must first unwrap it from the optional.
- The safest ways to unwrap an optional's value is by using **if let binding** or **nil coalescing**. Avoid **forced unwrapping**, as it could produce a runtime error.

Where to go from here?

And that's optionals, a core feature of Swift that helps make the language safe and easy to use. You'll find yourself using them throughout your code. They help to make your code safe by ensuring that the absence of values is handled explicitly. This is something that you'll hopefully come to admire over the course of your Swift

experience!

In particular, look forward to using them in Chapters 10–12, where you'll learn about collections.

Challenges

You've learned the theory behind optionals and seen them in practice. Now it's your turn to play!

Challenge A: You be the compiler

Which of the following are valid statements?

```
var name: String? = "Ray"
var age: Int = nil
let distance: Float = 26.7
var middleName: String? = nil
```

Challenge B: Divide and conquer

First, create a function that returns the number of times an integer can be divided by another integer without a remainder. The function should return `nil` if the division doesn't produce a whole number. Name the function `divideIfWhole`.

Then, write code that tries to unwrap the optional result of the function. There should be two cases: upon success, print "Yep, it divides \answer times", and upon failure, print "Not divisible :[".

Finally, test your function:

1. Divide 10 by 2. This should print "Yep, it divides 5 times."
2. Divide 10 by 3. This should print "Not divisible :[."

Hint 1: Use the following as the start of the function signature:

```
func divideIfWhole(value: Int, by divisor: Int)
```

You'll need to add the return type, which will be an optional!

Hint 2: You can use the modulo operator (%) to determine if a value is divisible by another; recall that this operation returns the remainder from the division of two numbers. For example, $10 \% 2 = 0$ means that 10 is divisible by 2 with no remainder, whereas $10 \% 3 = 1$ means that 10 is divisible by 3 three times with a remainder of 1.

Challenge C: Refactor and reduce

The code you wrote in the last challenge used `if` statements. In this challenge, refactor that code to use `nil` coalescing instead. This time, make it print "It divides X times" in all cases, but if the division doesn't result in a whole number, then X should be 0.

Section II: Collection Types

So far, you've mostly seen data in the form of single elements. Although tuples can have multiple pieces of data, you have to specify the size up front; a tuple with three strings is a completely different type from a tuple with two strings, and converting between them isn't trivial.

In this section, you'll learn about **collection types** in Swift. Collections are flexible "containers" that let you store any number of values together.

There are three collection types in Swift: arrays, dictionaries and sets. You'll learn about them in that order over the next three chapters:

- **Chapter 10, Arrays**
- **Chapter 11, Dictionaries**
- **Chapter 12, Sets**

The collection types have similar interfaces but very different use cases. As you read through these chapters, keep the differences in mind, and you'll begin to develop a feel for which type you should use when.

As part of exploring the differences between the collection types, we'll also consider performance: how quickly the collections can perform certain operations, such as adding to the collection or searching through it.

The usual way to talk about performance is with **big-O notation**. If you're not familiar with it already, read on for a brief introduction.

Introducing big-O notation

Big-O notation is a way to describe **running time**, or how long an operation takes to complete. The idea is that the exact time an operation takes isn't important; it's the relative difference in scale that matters.

Imagine you have a list of names in some random order, and you have to look up the first name on the list. It doesn't matter whether the list has a single name or a million names—glancing at the very first name always takes the same amount of time. That's an example of a **constant time** operation, or **O(1)** in big-O notation.

Now say you have to find a particular name on the list. You need to scan through

the list and look at every single name until you either find a match or reach the end. Again, we're not concerned with the exact amount of time this takes, just the relative time compared to other operations.

To figure out the running time, think in terms of units of work. You need to look at every name, so consider there to be one "unit" of work per name. If you had 100 names, that's 100 units of work. What if you double the number of names to 200; how does that change the amount of work? It *also* doubles the amount of work. Similarly, if you quadruple the number of names, that quadruples the amount of work.

This is an example of a **linear time** operation, or **O(N)** in big-O notation. The size of the input is the variable N, which means the amount of time the operation takes is also N. There's a direct, linear relationship between the input size (the number of names in the list) and the time it will take to search for one name.

You can see why constant time operations have the number 1 in O(1). They're just a single unit of work, no matter what!

You can read more about big-O notation by searching the Web. You'll only need constant time and linear time in this book, but there are other such **time complexities** out there.

Big-O notation is particularly important when dealing with collection types, because collections can store very large amounts of data, and you need to be aware of running times when you add, delete or edit values.

For example, if collection type A has constant-time searching and collection type B has linear-time searching, which you choose to use will depend on how much searching you're planning to do.

Chapter 10: Arrays

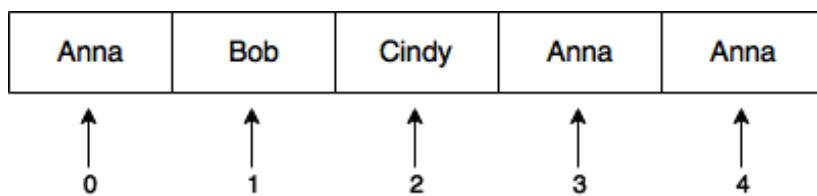
By Eli Ganem

Arrays are the most common collection type you'll run into in Swift. Arrays are typed, as are regular variables and constants, and store multiple values like a simple list.

Before you create your first array, let's take some time to consider in detail what an array is and why you might want to use one.

What is an array?

An array is an ordered collection of values of the same type. The elements in the array are **zero-indexed**, which means the index of the first element is 0, the index of the second element is 1 and so on. Knowing this, you can work out that the last element's index is the number of values in the array minus 1.



There are five elements in this array, at index values 0–4.

Also, all values are of type `String`. You can't add non-string types to an array that holds strings. Notice that the same value can appear multiple times.

When are arrays useful?

Arrays are useful when you want to store your items in a particular order. You may want ordering so you can sort the elements, or because you need to be able to fetch elements by index without iterating over the entire array.

For example, if you were storing high score data then order matters. You would want the highest score to come first in the list (i.e. at index 0) with the next-

highest score after that, and so on.

Mutable versus immutable arrays

Just like the previous types you've read about, like `String` or `Int`, when you create an array, you must declare it as either a constant or a variable.

If the array doesn't need to change after you've created it, you should make it immutable by declaring it as a constant with `let`. If you need to add, remove or update values in the array, then you should create a mutable array by declaring it as a variable.

Creating arrays

In this section, you'll practice declaring arrays and assigning them initial values.

Explicit declaration

You can declare an array either explicitly or by taking advantage of Swift's type inference. Here's an example of an explicit declaration:

```
let numbers: Array<Int>
```

The type inside the angle brackets defines the type of values the array can store, which the compiler will enforce when you add elements to the array. If you try to add a string, for example, the compiler will return an error and your code won't compile. This powerful enforcement mechanism is called **generics**, and you'll read a lot about it in Chapter 22.

In this example, you declare `numbers` as an array that can only store `Int` values. You've defined this array as a constant, so its values cannot change.

Inferred declaration

Swift can also infer the type of the array from the type of initializer:

```
let inferredNumbers = Array<Int>()
```

Here `inferredNumbers` is also an array of integers. If you're following along by typing this code into a playground, you'll notice `[]` in the results area. This is Swift's representation of an empty array.

Another, shorter, way to define an array is by enclosing the type inside square brackets, like so:

```
let alsoInferredNumbers = [Int]()
```

This is the most common declaration form for an array, and you'll be using it

throughout this book.

Array literals

When you declare an array, you might want to give it initial values. You can do that using **array literals**, which is a concise way to provide array values. An array literal is a list of values separated by commas and surrounded by square brackets:

```
let evenNumbers = [2, 4, 6, 8]
```

Since the declaration only contains integers, Swift infers the type of evenNumbers to be an array of Int values, or [Int].

It's also possible to create an array with all of its values set to a default value:

```
let allZeros = [Int](count: 5, repeatedValue: 0)
// > [0, 0, 0, 0, 0]
```

Since [Int]—including the square brackets—is just a type, you're calling its initializer in this example. The two parameters specify the count and the value that will be repeated.

All of the arrays you've created so far were immutable, because you assigned them to constants. That's a good practice for arrays that aren't going to change. For example, consider this array:

```
let vowels = ["A", "E", "I", "O", "U"]
```

vowels is an array of strings and its values can't be changed. But that's fine, since the list of vowels doesn't tend to change very often!

Accessing elements

Being able to create arrays is useless unless you know how to fetch values from an array. In this section, you're learn several different ways to access elements in an array.

Using properties and methods

Let's say you're creating a game of cards and you want to store the players' names in an array. The list will need to change as players join or leave the game, so you need to declare a mutable array:

```
var players = ["Alice", "Bob", "Cindy", "Dan"]
```

In this example, players is a mutable array because you assigned it to a variable.

Before the game starts, you need to make sure there are enough players. You can

use the `isEmpty` property to check if there's at least one player:

```
print(players.isEmpty)
// > false
```

The array isn't empty, but you need at least two players to start a game. You can get the number of players using the `count` property:

```
if players.count < 2 {
    print("We need at least two players!")
} else {
    print("Let's start!")
}
// > Let's start!
```

It's time to start the game! You decide that the order of play is by the order of names in the array. How would you get the first player's name?

Arrays provide the `first` property to fetch the first object of an array:

```
var currentPlayer = players.first
```

Printing the value of `currentPlayer` reveals something interesting:

```
print(currentPlayer)
// > Optional("Alice")
```

The property `first` actually returns an *optional*, because if the array were empty, `first` would return `nil`.

Similarly, arrays have a `last` property that returns the last value in an array, or `nil` if the array is empty:

```
print(players.last)
// > Optional("Dan")
```

Another way to get values from an array is by calling `minElement()`. This method returns the element with the lowest *value* in the array—not the lowest index! If the array contains strings, then it returns the string that's the lowest in alphabetical order, which in this case is "Alice":

```
currentPlayer = players.minElement()
print(currentPlayer)
// > Optional("Alice")
```

Obviously, `first` and `minElement()` will not always return the same value. For example:

```
print([2, 3, 1].first)
// > Optional(2)
print([2, 3, 1].minElement())
// > Optional(1)
```

As you might have guessed, arrays also have a `maxElement()` method.

Note: The `first` and `last` properties and the `minElement()` and `maxElement()` methods aren't unique to arrays. Every collection type has these properties and methods, in addition to a plethora of others. You'll learn more about this behavior when you read about protocols in Chapter 19.

Now that you've figured out how to get the first player, you'll announce who that player is:

```
if let currentPlayer = currentPlayer {  
    print("\(currentPlayer) will start")  
}  
// > Alice will start
```

You use `if let` to unwrap the optional you got back from `first`; otherwise, the statement would print `Optional("Alice") will start` and that's probably not what you want.

These properties and methods are helpful if you want to get the first, last, minimum or maximum elements. What if the element you want can't be obtained with one of these properties or methods?

Using subscripting

The most convenient way to access elements in an array is by using the subscript syntax. This syntax lets you access any value directly by using its index inside square brackets:

```
var firstPlayer = players[0]  
print("First player is \(firstPlayer)")  
// > First player is "Alice"
```

Because arrays are zero-indexed, you use index 0 to fetch the first object. You can use a bigger index to get the next elements in the array, but if you try to access an index that's beyond the size of the array, you'll get a runtime error.

```
var player = players[4]  
// > fatal error: Array index out of range
```

Why did you get this error? It's because `players` contains only four strings. Index 4 represents the fifth element, but there is no fifth element in this array.

When you use subscripts, you don't have to worry about optionals, since trying to access a non-existing index doesn't return `nil`; it simply causes a runtime error.

Using ranges

You can use the subscript syntax with ranges to fetch more than a single value from an array. For example, if you'd like to get the next two players to have their turns, you could do this:

```
let upcomingPlayers = players[1...2]
print(upcomingPlayers)
// > ["Bob", "Cindy"]
```

As you can see, the constant `upcomingPlayers` is actually an array of the same type as the original array.

The range you used is `1...2`, which represent the second and third items in each array. You can use any index here as long as the start value is smaller than the end value and they're both within the bounds of the array.

Checking for an element

You can check if there's at least one occurrence of a specific element in an array by using `contains(_:_)`, which returns `true` if it finds the element in the array, and `false` if it doesn't.

You can use this method to write a function that checks if a given player is in the game:

```
func isPlayerEliminated(playerName: String) -> Bool {
    if players.contains(playerName) {
        return false
    } else {
        return true
    }
}
```

Now you can use this function any time you need to check if a player has been eliminated:

```
print(isPlayerEliminated("Bob"))
// > false
```

You could even test for the existence of an element in a specific range:

```
players[1...3].contains("Bob")
// > true
```

Now that you can get data *out* of your arrays, it's time to look at mutable arrays and how to change their values.

Manipulating elements

You can make all kinds of changes to mutable arrays: adding and removing elements, updating existing values, and moving elements around into a different order. In this section, you'll see how to work with the array to match up what's going on with your game in progress.

Appending elements

If new players want to join the game, they need to sign up and add their names to the array. Eli is the first player to join the existing four players. You can add Eli to the end of the array using `append(_:)`:

```
players.append("Eli")
```

If you try to append anything other than a string, the compiler will show an error. Remember, arrays can only store values of the same type. Also, `append(_:)` only works with mutable arrays.

The next player to join the game is Gina. You can append her to the game another way, by using the `+ =` operator:

```
players += ["Gina"]
```

The right-hand side of this expression is an array with a single element, the string "Gina". By using `+ =`, you're appending the elements of that array to `players`. Now the array looks like this:

```
print(players)
// > ["Alice", "Bob", "Cindy", "Dan", "Eli", "Gina"]
```

Here, you added a single element to the array but you see how easy it would be to append *multiple* items by adding more names after Gina's.

Inserting elements

An unwritten rule of this card game is that the players' names have to be in alphabetical order. As you can see, the list is missing a player that starts with the letter F. Luckily, Frank has just arrived. You want to add him to the list between Eli and Gina. To do that, you can use `insert(_:atIndex:)`:

```
players.insert("Frank", atIndex: 5)
```

The `atIndex` argument defines where you want to add the element. Remember that the array is zero-indexed, so index 5 is exactly between Eli and Gina.

Removing elements

During the game, the other players caught Cindy and Gina cheating. They should be removed from the game! You know that Gina is last in the players list, so you can remove her easily with `removeLast()`:

```
var removedPlayer = players.removeLast()
print("\(removedPlayer) was removed")
// > Gina was removed
```

This method does two things: It removes the last element and then it returns it, in case you need to print it or store it somewhere else—like in an array of cheaters!

To remove Cindy from the game, you need to know the exact index where her name is stored. Looking at the list of players, you see that she's third in the list, so her index is 2.

```
removedPlayer = players.removeAtIndex(2)
print("\(removedPlayer) was removed")
// > Cindy was removed
```

How would you get the index of an element? There's a method for that! `indexOf(_:_)` returns the *first index* of the element, because the array might contain multiple copies of the same value. If the method doesn't find the element, it returns `nil`.

Mini-exercise

Use `indexOf(_:_)` to determine the position of the element "Dan" in `players`.

Updating elements

Frank has decided that everyone should call him Franklin from now on. You could remove the value "Frank" from the array and then add "Franklin", but that's just too much work for a simple task. Instead, you should use the subscript syntax to update the name:

```
print(players)
// > ["Alice", "Bob", "Eli", "Frank"]
players[3] = "Franklin"
print(players)
// > ["Alice", "Bob", "Eli", "Franklin"]
```

You have to be careful not to use an index beyond the bounds of the array, or your app will crash.

As the game continues, some players are eliminated and new ones come to replace them. Luckily, you can also use subscripting with ranges to update multiple values in a single line of code:

```
players[0...1] = ["Donna", "Craig", "Brian", "Anna"]
print(players)
// > ["Donna", "Craig", "Brian", "Anna", "Eli", "Franklin"]
```

This code replaces the first two players, Alice and Bob, with the four players in the new players array. As you can see, the size of the range doesn't have to be equal to the size of the array that holds the values you're adding.

Moving elements

Take a look at this mess! The players array contains names that start with A to F, but they aren't in the correct order, and that's violating the rules of the game.

You can try to fix this situation by manually moving values one by one to their correct positions, like so:

```
let playerAnna = players.removeAtIndex(3)
players.insert(playerAnna, atIndex: 0)
print(players)
// > ["Anna", "Donna", "Craig", "Brian", "Eli", "Franklin"]
```

This works if you want to move a single element, but if you want to sort the entire array, you should use `sort()`:

```
players = players.sort()
print(players)
// > ["Anna", "Brian", "Craig", "Donna", "Eli", "Franklin"]
```

`sort()` does exactly what you expect it to do: It returns a sorted *copy* of the array. If you'd like to sort the array in place instead of returning a sorted copy, you should use `sortInPlace()`.

Iterating through an array

It's getting late and the players decide to stop for the night and continue tomorrow; in the meantime, you'll keep their scores in a separate array. You'll see a better approach for this in the chapter about dictionaries, but for now you'll continue using arrays:

```
let scores = [2, 2, 8, 6, 1, 2]
```

Before the players leave, you want to print the names of those still in the game. You can do this using the `for-in` loop you read about in Chapter 6:

```
for playerName in players {
    print(playerName)
}
// > Anna
// > Brian
// > Craig
// > Donna
// > Eli
// > Franklin
```

This code goes over all the elements of `players`, from index 0 up to `players.count - 1`, and prints their values. In the first iteration, `playerName` is equal to the first element of the array; in the second iteration, it's equal to the second element of the array; and so on until the loop has printed all the items in the array.

If you also need the index of each element, you can iterate over the return value of the array's `enumerate()` method, which returns a tuple with the index and value of each element in the array.

```
for (index, playerName) in players.enumerate() {
    print("\(index + 1). \(playerName)")
}
// > 1. Anna
// > 2. Brian
// > 3. Craig
// > 4. Donna
// > 5. Eli
// > 6. Franklin
```

Now you can use the technique you've just learned to write a function that takes an array of integers as its input and returns the sum of its elements:

```
func sumOfAllItems(intArray: [Int]) -> Int {
    var sum = 0
    for number in intArray {
        sum += number
    }
    return sum
}
```

You could use this function to calculate the sum of the players' scores:

```
print(sumOfAllItems(scores))
// > 21
```

Mini-exercise

Write a `for-in` loop that prints the players' names and scores.

Sequence operations

The `for-in` loops you've just seen all have something in common: They iterate over all the elements in the array and take a certain action on each item.

Swift provides a few powerful methods on collections that can significantly reduce the amount of code you need to write for these repetitive tasks.

Reduce

`reduce(_:_:combine:)` takes an initial value as the first argument and accumulates every value in the array in turn, using the `combine` operation. An example is worth a thousand words:

```
let sum = scores.reduce(0, combine: +)
print(sum)
// > 21
```

This is the same as writing `let sum = 0 + scores[0] + scores[1] + ... + scores[5]`. It does exactly what `sumOfAllValues(_:_:)` does, but with a single line of code.

The `combine` argument is very sophisticated and it supports more than just a single arithmetic operation. It is a **closure**, and you'll read more about that in Chapter 23, "Functional Programming."

Filter

`filter(_:_:)` returns a new array by filtering out elements from the array on which it's called. Its only argument is a closure that returns a Boolean, and it will execute this closure once for each element in the array. If the closure returns true, `filter(_:_:)` will add the element to the returned array; otherwise it will ignore the element.

For example, in your card game, you'd like to print the highest scores of the day, which you define as scores higher than 5 points:

```
print(scores.filter({ $0 > 5 }))
// > [8, 6]
```

You can refer to the first parameter of a closure using the shorthand argument of `$0`, and this is how you can reference the element on which `filter(_:_:)` is currently operating.

Map

`map(_:_:)` also takes a single closure argument. As its name suggests, it maps each value in an array to a new value, using the closure argument.

Again, an example will come in handy. Let's say you're feeling generous and you'd like to double the amount of points each player has:

```
print(scores)
// > [2, 2, 8, 6, 1, 2]
let newScores = scores.map({ $0 * 2 })
print(newScores)
// > [4, 4, 16, 12, 2, 4]
```

With a single line of code, you've created a new array with all the scores multiplied by 2.

Note: You'll learn more about these sequence operations in Chapter 23, "Functional Programming."

Running time for array operations

Arrays are stored as a continuous block in memory. That means if you have ten elements in an array, the ten values are all stored one next to the other. With that in mind, here's what the performance of various array operations costs:

Accessing elements: The cost of fetching an element is $O(1)$. Since all the values are sequential, it's easy to do *random access* and fetch a value at a particular index; all the compiler needs to know is where the array starts and what index you want to fetch.

Inserting elements: The complexity of adding an element depends on the position in which you add the new element:

- If you add to the beginning of the array, Swift can do it in $O(1)$.
- If you add to the middle of the array, all values from that index on need to be shifted over. Doing so will require $N/2$ operations, therefore the running time is $O(N)$.
- If you add to the end of the array and there's room, it will take $O(1)$. If there isn't room, Swift will need to make space somewhere else and copy the entire array over before adding the new element, which will take $O(N)$. The average case is $O(1)$ though, because arrays are not full most of the time.

Deleting elements: Deleting an element leaves a gap where the removed element was. As was mentioned earlier, all elements in the array have to be sequential so this gap needs to be closed by shifting elements forward.

The complexity is similar to inserting elements: If you're removing an element from the beginning or the end, it's an $O(1)$ operation. If you're removing from the middle, the complexity is $O(N)$.

Searching for an Element: If the element you're searching for is the first element in the array, then the search will end after a single operation. If the element doesn't exist, you need to perform N operations until you realize that the element is not found. On average, searching for an element will take $N/2$ operations, therefore searching has a complexity of $O(N)$.

As you read the following chapters on dictionaries and sets, you'll see how their performance characteristics differ from arrays. That could give you a hint on which collection type to use for your particular case.

Key points

- **Arrays** are ordered collections of values of the same type.
- Use **subscripting** to access and update elements.
- Arrays are a value type, so they are copied when assigned to a new variable or passed as an argument to a function.
- Be wary of accessing an index that's out of bounds.

Where to go from here?

Arrays are very common in programming, and you'll see them in heavy use when you have many elements of a particular type to work with.

In the next two chapters, you'll learn about dictionaries and sets, the other two built-in Swift collection types. As you continue, keep arrays in the back of your mind so you can compare and contrast; that will let you see the differences and help you decide when to use each collection type.

Challenges

Challenge A: You be the compiler

Which of the following are valid statements?

1. `let array1 = [Int]()`
2. `let array2 = []`
3. `let array3: [String] = []`

For the next five statements, array4 has been declared as:

```
let array4 = [1, 2, 3]
```

```
4. print(array4[0])
5. print(array4[5])
6. array4[1...2]
7. array4[0] = 4
8. array4.append(4)
```

For the final five statements, array5 has been declared as:

```
var array5 = [1, 2, 3]
```

```
9. array5[0] = array5[1]
10. array5[0..1] = [4, 5]
11. array5[0] = "Six"
12. array5 += 6
13. for item in array5 { print(item) }
```

Challenge B: Removing an element

Write a function that removes the first occurrence of a given integer from an array of integers. This is the signature of the function:

```
func removeOnce(itemToRemove: Int, fromArray: [Int]) -> [Int]
```

Challenge C: Pick and choose

Write a function that removes all occurrences of a given integer from an array of integers. This is the signature of the function:

```
func remove(itemToRemove: Int, fromArray: [Int]) -> [Int]
```

Challenge D: Reversing an array

Arrays have a reverse() method that reverses the order of an array's items. Write a function that reverses an array without using reverse(). This is the signature of the function:

```
func reverse(array: [Int]) -> [Int]
```

Challenge E: Randomizing an array

The function below returns a random number between 0 and the given argument:

```
import Foundation
func randomFromZeroTo(number: Int) -> Int {
    return Int(arc4random_uniform(UInt32(number)))
}
```

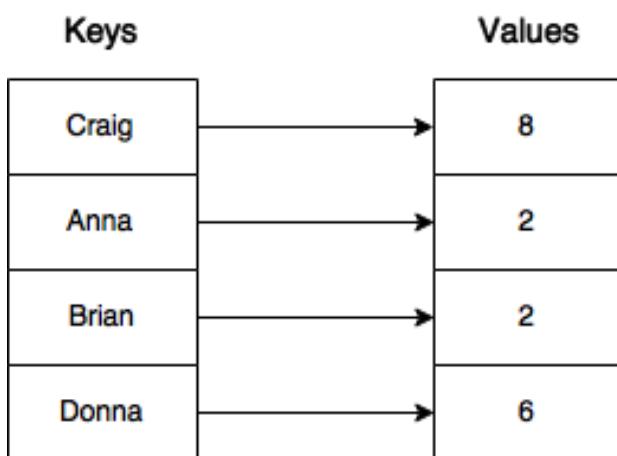
Use it to write a function that shuffles the elements of an array in random order.
This is the signature of the function:

```
func randomArray(array: [Int]) -> [Int]
```

Chapter 11: Dictionaries

By Eli Ganem

A dictionary is an unordered collection of pairs, where each pair is comprised of a **key** and a **value**.



As you can see in the diagram above, keys are unique. The same key can't appear twice in a dictionary, but different keys may point to the same value. All keys have to be of the same type, and all values have to be of the same type.

Dictionaries are useful when you want to look up values by means of an identifier. For example, the table of contents of this book maps chapter names to their page numbers, making it easy to skip to the chapter you want to read.

How is this different from an array? With an array, you can only fetch a value by its index, which has to be an integer, and all indexes have to be sequential. In a dictionary, the keys can be of any type and in no particular order.

Creating dictionaries

When you create a dictionary, you can explicitly declare the type of the keys and the type of the values. Here's an example of an explicit declaration of a dictionary:

```
let pairs: Dictionary<String, Int>
```

This is similar to an explicit declaration of an array, but here you have two types inside the angle brackets, separated by a comma: the type of the keys and the type of the values. In this example, `pairs` is an *immutable* dictionary with strings as keys and integers as values.

Inferred declaration

Swift can also infer the type of a dictionary from the type of the initializer:

```
let inferredPairs = Dictionary<String, Int>()
```

Or written with the preferred shorthand:

```
let alsoInferredPairs = [String: Int]()
```

Within the square brackets, the type of the keys is followed by a colon and the type of the values. This is the most common way to declare a dictionary, and you'll be using it throughout this book.

Dictionary literals

If you want to declare a dictionary with initial values, you can use a **dictionary literal**. This is a list of key-value pairs separated by commas, enclosed in square brackets.

For your card game from the last chapter, instead of using the two arrays to map players to their scores, you can use a dictionary literal:

```
let namesAndScores = ["Anna": 2, "Brian": 2, "Craig": 8, "Donna": 6]
print(namesAndScores)
// > ["Brian": 2, "Anna": 2, "Craig": 8, "Donna": 6]
```

In this example, the dictionary has pairs of `[String: Int]`. When you print the dictionary, you see there's no particular order to the pairs.

An empty dictionary literal looks like this: `[:]`. You can use that to empty an existing dictionary like so:

```
var emptyDictionary: [Int: Int]
emptyDictionary = [:]
```

Accessing values

As with arrays, there are several ways to access dictionary values.

Using subscripting

Dictionaries support subscripting to access values. Unlike arrays, you don't access a value by its index but rather by its key. For example, if you want to get Anna's score, you would type:

```
print(namesAndScores["Anna"])
// > Optional(2)
```

Notice that the return type is an optional. The dictionary will check if there's a pair with the key "Anna", and if there is, return its value. If the dictionary doesn't find the key, it will return nil.

```
print(namesAndScores["Greg"])
// > nil
```

Remember with arrays, out-of-bounds subscript access caused a runtime error but dictionaries are different since their results are wrapped in an optional.

You might not be aware of this yet, but subscript access with optionals is really powerful. It lets you find out if a specific player is in the game without needing to iterate over all the keys, as you must do when you use an array.

Using properties and methods

Dictionaries have the same isEmpty and count properties as arrays:

```
print(namesAndScores.isEmpty)
// > false
print(namesAndScores.count)
// > 4
```

If you'd like to look only at the keys or the values of a dictionary, you can create an array from the dictionary's keys or values properties, respectively:

```
print(Array(namesAndScores.keys))
// > ["Brian", "Anna", "Craig", "Donna"]
print(Array(namesAndScores.values))
// > [2, 2, 8, 6]
```

These are regular arrays as you've learned about in the previous chapter.

Modifying dictionaries

Adding pairs

Bob wants to join the game.



Take a look at his info before you let him join:

```
var bobData = ["name": "Bob",
    "profession": "Card Player",
    "country": "USA"]
```

This dictionary is of type [String: String], and it's mutable because it's assigned to a variable. Let's say you got more information about Bob and you wanted to add it to the dictionary. This is how you'd do it:

```
bobData.updateValue("CA", forKey: "state")
```

There's even a shorter way, using subscripting:

```
bobData["city"] = "San Francisco"
```

Bob's a professional card player. So far, he sounds like a good addition to your roster.

Mini-exercise

Write a function that prints a given player's city and state.

Updating values

It appears that in the past, Bob was caught cheating when playing cards. He's not just a professional, he's a card shark! He asks you to change his name and profession so no one will recognize him.

Because Bob seems eager to change his ways, you agree. First, you change his

name from Bob to Bobby:

```
bobData.updateValue("Bobby", forKey: "name")
// > Bob
```

You saw this method above when you read about adding pairs. Why does it return the string "Bob"? `updateValue(_:_:forKey:)` replaces the value of the given key with the new value and returns the old value. If the key doesn't exist, this method will add a new pair and return `nil`.

As with adding, you can do this with less code by using subscripting:

```
bobData["profession"] = "Mailman"
```

Like the first method, the code updates the value for this key or, if the key doesn't exist, creates a new pair.

Removing pairs

Bob—er, sorry—*Bobby*, still doesn't feel safe, and he wants you to remove all information about his whereabouts:

```
bobData.removeValue(forKey: "state")
```

This method will remove the key "state" and its associated value from the dictionary. As you might expect, there's a shorter way to do this using subscripting:

```
bobData["city"] = nil
```

Assigning `nil` as a key's associated value removes the pair from the dictionary.

Iterating through dictionaries

The `for-in` loop also works when you want to iterate over a dictionary. But since the items in a dictionary are pairs, you need to use a tuple:

```
for (key, value) in namesAndScores {
    print("\(key) - \(value)")
}
// > Brian - 2
// > Anna - 2
// > Craig - 8
// > Donna - 6
```

It's also possible to iterate over just the keys:

```
for key in namesAndScores.keys {
    print("\(key), ", terminator: "") // no newline
}
print("") // print one final newline
```

```
// > Brian, Anna, Craig, Donna,
```

You can iterate over just the values in the same manner with the `values` property on the dictionary.

Sequence operations

You can also perform sequence operations on dictionaries. For example, you could use `reduce(_:_:combine:)` to replace the previous code snippet with a single line of code:

```
let namesString = namesAndScores.reduce("",  
    combine: { $0 + "\($1.0), " })  
print(namesString)
```

In a `reduce` statement, `$0` refers to the partially-combined result, and `$1` refers to the current element. Since the elements in a dictionary are tuples, you need to use `$1.0` to get the *key* of the pair.

Let's see how you could use `filter(_:)` to find all the players with a score of less than 5:

```
print(namesAndScores.filter({ $0.1 < 5 }))  
// > [("Brian", 2), ("Anna", 2)]
```

Here you use `$0.1` to access the *value* of the pair.

Running time for dictionary operations

In order to be able to examine how dictionaries work, you need to understand what **hashing** is and how it works. Hashing is the process of transforming a value - String, Int, Double, Bool, etc - to a numeric value, known as the *hash value*.

In Swift, all basic types are *hashable* and have a `hashValue` property. Here's an example:

```
print("some string".hashValue)  
// > 4799450059642629719  
print(1.hashValue)  
// > 1  
print(false.hashValue)  
// > 0
```

The hash value has to be deterministic - meaning that a given value must always return the same hash value. No matter how many times you calculate the hash value for "some string", it will always give the same value.

Dictionaries can only store keys that are hashable. This is an implementation detail beyond the scope of this book, but dealing with hashes makes checking for uniqueness and searching much easier.

Here's what the performance of various dictionary operations costs:

Accessing Elements: Getting the value of a given key is a constant time operation, or O(1).

Inserting Elements: To insert an element, the dictionary needs to calculate the hash value of the key and then store data based on that hash. These are all O(1) operations.

Deleting Elements: Again, the dictionary needs to calculate the hash value to know exactly where to find the element, and then remove it. This is also an O(1) operation.

Searching for an Element: As mentioned above, accessing an element has constant running time, so the complexity for searching is also O(1).

Key points

- A **dictionary** is an unordered collection of key-value pairs.
- The **keys** of a dictionary are all of the same type, and the values are all of the same type.
- Use **subscripting** to get values and to add, update or remove pairs.
- Iterating through a dictionary returns a tuple containing both the key and the value.

Where to go from here?

Dictionaries are quite different from arrays: they're unordered, have keys and values, and have excellent searching performance.

However, the interface is similar with subscripts and for-in iteration. As you'll learn about later in Chapter 19, "Protocols," this similarity is thanks to a common interface across the collection types.

In the next chapter, you'll learn about **sets**, which combine concepts you've learned about from both dictionaries and arrays.

Challenges

Challenge A: You be the compiler

Which of the following are valid statements?

1. `let dict1 = Int,Int()`
2. `let dict2 = []`
3. `let dict3 = [Int: Int]()`

For the next four statements, use the following dictionary:

```
let dict4 = ["One": 1, "Two": 2, "Three": 3]
```

4. `dict4[1]`
5. `dict4["One"]`
6. `dict4["Zero"] = 0`
7. `dict4[0] = "Zero"`

For the next three statements, use the following dictionary:

```
var dict5 = ["NY": "New York", "CA": "California"]
```

8. `dict5["NY"]`
9. `dict5["WA"] = "Washington"`
10. `dict5["CA"] = nil`

Challenge B: Replacing dictionary values

Write a function that replaces the values of two keys in a dictionary. This is the function's signature:

```
func replaceValueForKey(key1: String, withValueForKey key2: String,  
inDictionary: [String: Int]) -> [String: Int]`
```

Challenge C: Searching through a dictionary

Given a dictionary with two-letter state codes as keys, and the full state names as values, write a function that prints all the states with a name longer than eight characters. For example, for the dictionary `["NY": "New York", "CA": "California"]`, the output would be "California".

Challenge D: Combining dictionaries

Write a function that combines two dictionaries into one. If a certain key appears in both dictionaries, ignore the pair from the first dictionary. This is the function's signature:

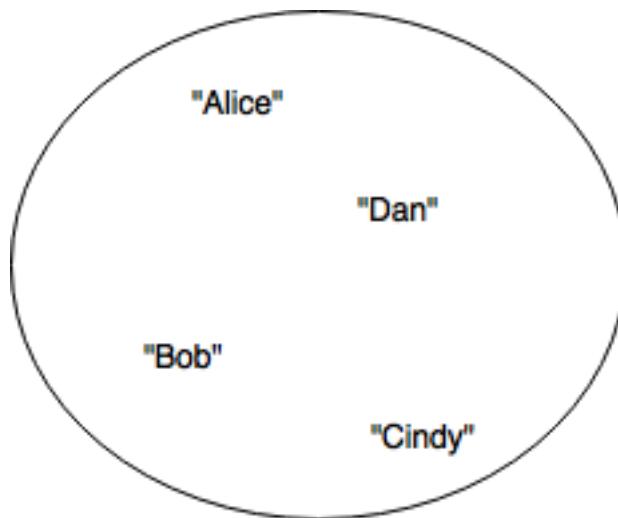
```
func combine(dict1: [String: String], with dict2: [String: String]) ->
    [String: String]
```

Chapter 12: Sets

By Eli Ganem

A set is an unordered collection of unique values of the same type. It can be extremely useful when you want to ensure that an item doesn't appear more than once in your collection, and the order of your items isn't important.

For example, if you wanted to keep track of which players were still active in the game, you could use a set.



You're not so interested in ordering the players here; the thing you want the set to do is store people and not allow them to be in the set twice. Then, you would just ask the set, "is Alice still in the game?" and expect a Boolean true / false.

As you can see in the diagram above, there are no indices or keys by which you can retrieve values. As you'll read about in this chapter, you can still iterate through the set and query whether a particular value is in the set.

Creating sets

You can declare a set explicitly by writing `Set` followed by the type inside angle brackets:

```
let setOne: Set<Int>
```

Swift can also infer the type of the set from the type of the initializer:

```
let setTwo = Set<Int>()
```

Print `setTwo`, and you'll discover something interesting:

```
print(setTwo)
// > []
```

It appears that the empty set literal is the same as that of an empty array.

Set literals

Arrays and sets share more than just the empty literal. In fact, sets don't have their own literals, and you use **array literals** to create a set with initial values. Consider this example:

```
let someArray = [1, 2, 3, 1]
```

This is an array. So how would you use array literals to create a set? Like this:

```
let someSet: Set<Int> = [1, 2, 3, 1]
```

You have to explicitly declare the constant as a `Set`. However, thanks to Swift's type inference, you can omit the type of values the set contains:

```
let anotherSet: Set = [1, 2, 3, 1]
```

This is obviously a set of integers, so there's no need to define the set's type.

To see the most important features of a set in action, let's print one of the sets you just created:

```
print(someSet)
// > [2, 3, 1]
```

First, you can see there's no specific ordering. Second, although you created the set with two instances of the value 1, that value only appears once—remember, a set's values must be unique.

Accessing elements

Elements in sets don't have indices or keys, so how can you know if the set even has elements, and how can you access any elements it does have? Like arrays and dictionaries, sets support the `count` and `isEmpty` properties:

```
print(someSet.isEmpty)
// > false
print(someSet.count)
// > 3
```

You can use `contains(_:_)` to check for the existence of a specific element:

```
print(someSet.contains(1))
// > true
print(someSet.contains(4))
// > false
```

You can also use the `first` and `last` properties, which return one of the elements in the set. However, because sets are unordered, you can't know exactly which item you'll get. For sets of size 0 or 1 though, these can be a quick way to access the single set element.

```
print(someSet.first)
// > Optional(2)
```

As expected, `first` returns an optional. It has to because the set might be empty, in which case the property would return `nil`.

Adding and removing elements

Let's return to your card game. You want to schedule a game for tomorrow and you want to choose a time that's convenient for everyone. You begin by suggesting your preferred times:

```
var myTimes: Set = ["8am", "9am", "10am"]
```

You declare this set as a variable because your preferences will likely change. For example, let's say you decide that 11am is also a good time to start the game, so you add another element to your set:

```
myTimes.insert("11am")
print(myTimes)
// > ["9am", "8am", "11am", "10am"]
```

`insert(_:_)` adds the element to the set; if the element already exists, the method does nothing.

You realize that 8am is too early. You can remove the element from the set like

this:

```
let removedElement = myTimes.remove("8am")
print(removedElement)
// > Optional("8am")
```

`remove(_:)` returns the removed element if it's in the set, or `nil` otherwise.

Mini-exercise

Write a function that removes all duplicates from an array of integers. Arrays? Isn't this a chapter about sets, you ask?

Here's a hint: Arrays have an initializer with a Set type, and sets have an initializer with an Array type.

Iterating through a set

Use a `for-in` loop when you want to iterate over the elements of a set:

```
for element in myTimes {
    print(element)
}
// > 9am
// > 11am
// > 10am
```

As you'd expect, the results are in no defined order. However, it's the same familiar `for-in` syntax you've already seen in the other collection types.

Set operations

One of the most powerful features of sets is their support of **set operations**, which let you combine two sets into one, create a set with only the common values of two sets, and more. Let's examine these operations with your set of preferred times and another set of times.

As a reminder, you suggested a game time of 9am, 10am or 11am. Adam sent you his preferred times:

```
let adamTimes: Set = ["9am", "11am", "1pm"]
```

Let's consider each of the four types of set operations in turn.

Union

`union(_:)` creates a new set with all the values of the two sets:

```
let unionSet = myTimes.union(adamTimes)
print(unionSet)
// > ["9am", "11am", "10am", "1pm"]
```

The main thing to notice is that even though you and Adam both selected 9am and 11am as good times to meet, these values only appear once in the new set.

Here, you're left with every time suggested by both you and Adam.

Intersect

`intersect(_:_)` creates a new set with only the values common to both sets. You can use it to create a set with the times upon which you and Adam agree:

```
let intersectSet = myTimes.intersect(adamTimes)
print(intersectSet)
// > ["9am", "11am"]
```

Here, the values in the resulting set are the ones that were present in *both* sets `myTimes` and `adamTimes`.

Subtract

`subtract(_:_)` creates a new set by removing values that appear in the second set.

```
let subtractSet = myTimes.subtract(adamTimes)
print(subtractSet)
// > ["10am"]
```

You begin with all the times from `myTimes`. `subtract(_:_)` then removes all the times that appear in `adamTimes`, which leaves only the start times that you suggested, but Adam didn't.

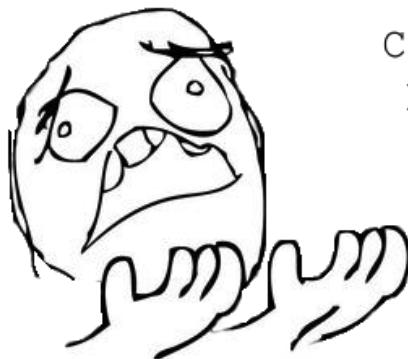
In this case, 10am is the only time suggested by you that wasn't also suggested by Adam.

Exclusive Or

`exclusiveOr(_:_)` creates a new set with the values that appear in one of the sets, but not both.

```
let exclusiveOrSet = myTimes.exclusiveOr(adamTimes)
print(exclusiveOrSet)
// > ["10am", "1pm"]
```

These are the start times that you suggested and Adam didn't, as well as the start times that Adam suggested and you didn't.



Can't we just
pick a time
already?

Mini-exercise

Quick question: which set operation would you use to find a good time to meet?

Running time for set operations

Sets have a very similar implementations to those of dictionaries, and they also require the elements to be hashable. The running time of all the operations is identical to those of dictionaries.

Key points

- **Sets** are unordered collections of unique values of the same type.
- Sets are most useful when you need to know whether something is included in the collection or not.
- You can initialize sets from arrays and vice versa.
- **Set operations** are very powerful and let you work with potentially large amounts of data to combine and filter elements.

Where to go from here?

Now that you've learned about the three collection types in Swift, you should have a good idea of what they can do and when you should use them. You'll see them come up as you continue on in the book.

Now that you know how to combine data together into arrays, dictionaries and sets, the next section of the book covers **named types**. These let you combine both data and code together into your own custom types to act as building blocks for even bigger and better things!

Before you move on, check out the following challenges to test your set knowledge.

Challenges

Challenge A: You be the compiler

Which of the following are valid statements?

1. `let set1: Set<Int> = [1]`
2. `let set2 = [1]`
3. `let set3 = Set<Int>()`

For the next three statements, use this set:

```
let set4: Set = [1, 2, 3]
```

4. `set4[1]`
5. `set4.insert(1)`
6. `set4.contains(1)`

For the next two statements, use this set:

```
var set5: Set = [1, 2, 3]
```

7. `set5.insert("1")`
8. `set5.remove(1)`

Challenge B: Checking for set inclusion

Assume you have a set with the email addresses of 10,000 people: `let emails: Set<String>`. Write a function that returns `true` if a given email address is in the set, or `false` otherwise. You're not allowed to use `contains(_:_)` or to iterate over the set.

Challenge C: Tracking attendance

You're a professor at Swift University and you're teaching a three-day course. Each day, you track attendance by adding the names of every student in attendance to a new set.

For example:

```
let day1: Set = ["Anna", "Benny", "Charlie"]
let day2: Set = ["Anna", "Benny", "Danny"]
let day3: Set = ["Anna", "Danny", "Eric"]
```

1. Write a function that returns a set with the students who attended all three days.

2. Write a function that determines who dropped out of the course after the first day.

The signature for these two functions is:

```
func funcName(day1: Set<String>, day2: Set<String>, day3: Set<String>) ->  
Set<String>
```

Section III: Building Your Own Types

Now that you know the basics of Swift, it's time to put everything you've learned together to create your own types.

You can create your own type by combining variables and functions into a new type definition. For example, integers and doubles might not be enough for your purposes, so you might need to create a type to store complex numbers. Or maybe storing first, middle and last names in three independent variables is getting difficult to manage, so you decide to create a `FullName` type.

When you create a new type, you get to give it a name; thus, these custom types are known as **named types**. Swift includes four kinds of named types: structures, classes, enumerations and protocols.

The first four chapters of this section cover structures, classes and enumerations. You'll learn about what each of these named types can do, the differences between them and when to use them:

- **Chapter 13, Structures**
- **Chapter 14, Classes**
- **Chapter 15, Advanced Classes**
- **Chapter 16, Enumerations**

In the following two chapters, you'll explore common features that apply to all named types. **Properties** are the data you store in named types, and **methods** are the bundles of code that perform tasks:

- **Chapter 17, Properties**
- **Chapter 18, Methods**

In the final chapters of the section, you'll learn about **protocols**, a special kind of named type that acts more like a blueprint for a type rather than a type you use directly. Protocols also have advanced features that lend themselves to a *protocol-oriented* way to organize your code:

- **Chapter 19, Protocols**
- **Chapter 20, Protocol-Oriented Programming**

Custom types make it possible to build large and complex things with the basic building blocks you've learned so far. It's time to take your Swift apprenticeship to the next level!



Chapter 13: Structures

By Erik Kerber

You've covered many of the fundamental building blocks of programming in Swift. With variables, conditionals, strings, functions and collections, you're ready to conquer the world! Almost. ;]

Most programs that perform complex tasks would probably benefit from higher levels of abstraction. In other words, in addition to an Int, String or Array, they'll need new types that are specific to the domain of the task at hand. Keeping track of photos or contacts, for example, demands more than just the simple types you've seen thus far.

This chapter will introduce **structures**, which are a "named type". Like a String, Int or Array, you can define your own structures to create named types to later use in your code. By the end of this chapter, you'll know how to define and use your own structures.

Introducing structures

Imagine you're writing a program that calculates if a potential customer is within range of a pizza delivery restaurant. You might write code like this:

```
let latitude: Double = 44.9871
let longitude: Double = -93.2758
let range: Double = 200.0

func isInRange(lat: Double, long: Double) -> Bool {
    // And you thought in Math class
    // you would never use the Pythagorean theorem!
    let difference = sqrt(pow((latitude - lat), 2) +
        pow((longitude - long), 2))
    let distance = difference * 0.002
    return distance < range
}
```

Simple enough, right? A successful pizza delivery business may eventually expand to include multiple locations, which adds a minor twist to the deliverable calculator:

```
let latitude_1: Double = 44.9871
let longitude_1: Double = -93.2758

let latitude_2: Double = 44.9513
let longitude_2: Double = -93.0942
```

Now what? Do you update your function to check against both sets of coordinates? Eventually, the rising number of customers will force the business to expand, and soon it might grow to a total of 10 stores! Not only that, but some new stores might have different delivery ranges, depending on whether they're in a big city or a sprawling suburb.

You might briefly consider creating an array of latitudes and longitudes, but it would be difficult to both read and maintain. Fortunately, Swift has additional tools to help you simplify the problem.

Your first structure

Structures, or **structs**, are one of the **named types** in Swift that allow you to encapsulate related properties and behaviors. You can define it, give it a name and then use it in your code.

In the example of the pizza business, it's clear that latitude and longitude are closely related—close enough that you could think of them as a single value:

```
struct Location {
    let latitude: Double
    let longitude: Double
}
```

This block of code demonstrates the basic syntax for defining a struct. In this case, it's a type named `Location` that combines both latitude and longitude.

The basic syntax begins with the `struct` keyword followed by the name and a pair of curly braces. Everything between the curly braces is a “member” of the struct.

Now that you've defined your first struct, you can instantiate one and store it in a constant or variable just like any other type you've worked with:

```
let pizzaLocation = Location(latitude: 44.9871,
                             longitude: -93.2758)
```

Instead of storing the coordinates in two separate variables, you store them together!

To create the `Location` value, you use the name of the type along with parentheses containing both the `latitude` and the `longitude` parameters. Swift defines this **initializer** automatically, and it's a subject this chapter will cover in more detail

later.

You may remember that there's also a range involved, and now that the pizza business is expanding, there may be different ranges associated with different restaurants. You can create another struct that represents the location and the range, like so:

```
struct DeliveryRange {  
    var range: Double  
    let center: Location  
}  
  
let storeLocation = Location(latitude: 44.9871,  
                             longitude: -93.2758)  
  
var pizzaRange = DeliveryRange(range: 200,  
                               center: storeLocation)
```

Now there's a new struct named `DeliveryRange` that contains a variable range of the pizza restaurant, along with the constant center of the location. As you can see, you can have a struct value inside a struct; here, you use the `Location` type as a member of the `DeliveryRange` struct.

Mini-exercise

Write a struct that represents a pizza order. Include toppings, size and any other option you'd want for a pizza.

Accessing members

With your `DeliveryRange` defined and an instantiated value in hand, you may be wondering how you can *use* these values. In Swift, structures and the other types you'll learn about use a simple "dot syntax" to access their members:

```
print(pizzaRange.range) // 200
```

You can even access *members of members* using dot syntax:

```
print(pizzaRange.center.latitude) // 44.9871
```

Similar to how you can read values with dot syntax, you can also *assign* them. If the delivery range of one pizza location becomes larger, you could assign the new value to the existing variable:

```
pizzaRange.range = 250
```

The semantics of constants and variables play a significant role in determining if a value can be assigned. In this case, you can assign `range` because you declared it as a `var`. On the other hand, you declared the `center` with `let`, so you can't modify it. Your `DeliveryRange` struct allows a pizza restaurant's delivery range to be

changed, but not its location!



In addition to properties, you must declare the struct itself as a variable to be able to modify it:

```
let constPizzaRange =  
    DeliveryRange(range: 200, center: storeLocation)  
  
// Error: change 'let' to 'var' above to make it mutable.  
constPizzaRange.range = 250
```

Mini-exercise

Rewrite `isInRange` to use `Location` and `DeliveryRange`.

Initializing a struct

When you defined the `Location` and `DeliveryRange` structs, Swift automatically generated a way to create these values:

```
Location(latitude: 44.9871, longitude: -93.2758)
```

When you create an instance of a type such as `Location`, you need to use a special kind of function called an **initializer**. Swift will automatically create a default initializer, such as the one shown above, that takes each member as a named parameter.

You can also create your own initializer, which allows you to customize how a structure's members are assigned:

```
import Darwin  
  
struct Location {  
    let latitude: Double  
    let longitude: Double  
  
    // String in GPS format "44.9871,-93.2758"  
    init(coordinateString: String) {  
        let crdSplit = coordinateString.characters.split(",")  
        latitude = atof(String(crdSplit.first!))  
        longitude = atof(String(crdSplit.last!))  
    }  
}
```

```
}
```

Here, the `init` keyword marks this function as an initializer for the `Location` struct.

In this case, the initializer takes a latitude and longitude together as a comma-separated string. This could be useful if you're reading locations from a text file.

```
let coords = Location(coordinateString: "44.9871,-93.2758")
print(coords.latitude) // 44.9871
print(coords.longitude) // -93.2758
```

Keep in mind that as soon as you define a custom initializer, Swift won't add the automatically-generated one. If you still need it, you'll have to define it yourself!

You can create as many initializers on a struct as you like, depending on your needs:

```
struct Order {
    var toppings: [String]
    var size: String
    var crust: String

    init(toppings: [String], size: String, crust: String) {
        self.toppings = toppings
        self.size = size
        self.crust = crust
    }

    init(size: String, crust: String) {
        self.toppings = ["Cheese"]
        self.size = size
        self.crust = crust
    }

    init(special: String) {
        self.size = "Large"
        self.crust = "Regular"

        if special == "Veggie" {
            self.toppings = ["Tomatoes", "Green Pepper", "Mushrooms"]
        } else if special == "Meat" {
            self.toppings = ["Sausage", "Pepperoni", "Ham", "Bacon"]
        } else {
            self.toppings = ["Cheese"]
        }
    }
}
```

As you can see, you can create a variety of initializers to suit a number of use cases. The first initializer is like the auto-generated one and takes parameters to match the properties exactly. The next two examples let you provide alternate values, and the initializer has some extra logic to fill in the properties.

Mini-exercises

1. Write an initializer for `DeliveryRange` that takes a `Location` and defaults the range to 150.
2. Create an initializer that takes a `String` for "City" or "Suburb" instead of an `Int` for range. Cities should have a range of 100 and suburbs a range of 150.

Introducing self

In the last example, you may have noticed the keyword `self` written in the initializers. When you use `self` in code, you're explicitly accessing the current value of the named type.

In other words, using dot syntax on `self` is just like using dot syntax on a variable storing that value.

```
self.crust = "Thin"  
print(self.crust) // Thin
```

You're not required to use `self` when writing code within a named type; Swift infers it automatically, which is why you've been able to write code without it so far.

Why use it then? It's especially useful in initializers: When two variables of the same name exist in the same scope, `self` can prevent what's known as **shadowing**.

```
init(toppings: [String], size: String, crust: String) {  
    toppings = toppings  
    size = size  
    crust = crust  
}
```

If you were to omit `self`, the compiler would complain and "helpfully" suggest you mark the function's parameters with `var` rather than the constants they are by default. But that would just assign the value to itself, which would be pointless!

Instead, you can simply add `self.` before the variable you're assigning, as follows:

```
init(toppings: [String], size: String, crust: String) {  
    self.toppings = toppings  
    self.size = size  
    self.crust = crust  
}
```

This makes it clear that you're assigning the parameter values to the properties.

Of course, you could name the input parameters differently from the stored properties, making it clear which are which. If you don't like this approach, you can always go with the explicit `self`.

Initializer rules

Initializers in structs have a few rules that guard against unset values. By the end of the initializer, the struct *must* have initial values set in all of its stored properties.

If you were to forget to assign the value of `longitude`, for instance, you would see an error at compile time:

```
struct Location {
    var latitude: Double
    var longitude: Double

    init(coordinateString: String) {
        }
    } ! Return from initializer without initializing all stored properties
```

Its purpose is simple: Stored properties need to be initialized with a value.

Other languages have types similar to structs and may assign default values, such as a value of 0 for an integer. Swift, focusing on safety, ensures that every value has been explicitly set.

There's one exception to the rule that stored properties must have values: optionals!

```
struct ClimateControl {
    var temperature: Double
    var humidity: Double?

    init(temp: Double) {
        temperature = temp
    }
}
```

In this simplified example, `humidity` is an optional—perhaps an “optional” setting on a thermostat! Here, the initializer doesn’t specify a value for `humidity`. Because `humidity` is an optional, the compiler will happily oblige.

For convenience, you can specify another initializer that would include both values:

```
struct ClimateControl {
    var temperature: Double
    var humidity: Double?

    init(temp: Double) {
        temperature = temp
    }

    init(temp: Double, hum: Double) {
        temperature = temp
        humidity = hum
    }
}
```

Now you can create `ClimateControl` values with or without a humidity value:

```
let ecoMode = ClimateControl(temp: 75.0)
let dryAndComfortable = ClimateControl(temp: 71.0, hum: 30.0)
```

Optional variables *do* get a default value of `nil`, which means the first initializer with only a temperature parameter is valid.

Note: If you declare the optional as a constant, you must still provide an initial value, whether it's `nil` or an actual value. Once the struct is defined, it can no longer be changed!

In some cases, you might have a sensible default in mind and want to save some typing later on. Swift also allows you to initialize a struct with default values:

```
struct ClimateControl {
    var temperature: Double = 68.0
    var humidity: Double?
}
```

Since `temperature` now has a default value of `68.0`, and because `humidity` doesn't require a value as an optional, you can create a `ClimateControl` value without specifying *any* parameters:

```
let defaultClimate = ClimateControl()
print(defaultClimate.temperature) // 68.0
```

Note: Just like when declaring constants and variables outside of structs, if you include an initial value along with the declaration, you don't need to specify a type—Swift will use type inference to determine the type for you!

Introducing methods

Using some of the capabilities of structs, you could now make a pizza delivery range calculator that looks something like this:

```
let pizzaJoints = [
    DeliveryRange(location:
        Location(coordinateString: "44.9871,-93.2758")),
    DeliveryRange(location:
        Location(coordinateString: "44.9513,-93.0942")),
]

func isInRange(location: customer) -> Bool {
    for pizzaRange in pizzaJoints {
        let difference = sqrt(pow((latitude - lat), 2) +
            pow((longitude - long), 2))
```

```

        if (difference < joint.range) {
            return true
        }
    }
    return false
}

let customer = Location("44.9850,-93.2750")

print(isInRange(location: customer)) // Pizza time!

```

In this example, there's an array `pizzaJoints` and a function that uses that array to determine if a customer's location is within range of any of them.

The idea of being "in range" is very tightly coupled to the characteristics of a single pizza restaurant. In fact, all of the calculations in `isInRange` occur on one location at a time. Wouldn't it be great if `DeliveryRange` itself could tell you if the restaurant can deliver to a certain customer?

Much like a struct can have constants and variables, it can also define its *own* functions:

```

struct DeliveryRange {
    var range: Double
    let center: Location

    func isInRange(customer: Location) -> Bool {
        let difference = sqrt(pow((latitude - center.latitude), 2) +
            pow((longitude - center.longitude), 2))
        return difference < range
    }

    // initializers here
}

```

This code defines the **method** `isInRange`, which is now a member of `DeliveryRange`. In Swift, methods are simply functions that are associated with a type. Just like other members of structs, you can use dot syntax to access a method through a value of its associated type:

```

let range = DeliveryRange(range: 150,
    center: Location("44.9871,-93.2758"))

let customer = Location(coordinateString: "44.9850,-93.2750")

range.isInRange(customer) // true!

```

Extensions

In addition to defining member methods and properties in the body of the struct, you can also define them in something called an **extension**. An extension can be declared using the following syntax:

```

extension Location {
    func isNorthernHemisphere() -> Bool {
        return latitude > 0.0
    }
}

let location = Location(coordinateString: "44.9850,-93.2750")
location.isNorthernHemisphere() // true

```

This defines an extension on `Location` that declares `isNorthernHemisphere()`. Much like any method defined in the original body of `Location`, `isNorthernHemisphere()` will be available for all instances of `Location`.

Extensions can be used to extend types you do not own yourself:

```

extension String {
    func evenOrOdd() -> String {
        return characters.count % 2 == 0 ? "Even!" : "Odd!"
    }
}

"I'm odd".evenOrOdd() // "Odd!"

```

This code adds an extension onto `String` that will return "Even!" if the string length is even or "Odd!" if the length is odd. More importantly, the extension allowed you to add your own method to a built-in Swift type!

Extensions can be applied to any type you will learn about including structs, classes, and enums. You will find them particularly useful to you when you reach Chapter 19, "Protocols."

Mini-exercise

Add a method on `DeliveryRange` that can tell you if the restaurant is within range of *another* pizza delivery shop.

Structures as values

The term **value** has an important meaning when it comes to structs in Swift, and that's because structs create what are known as **value types**.

A value type is an object or a piece of data that is *copied* on assignment, which means the assignment gets an *exact copy of the data* rather than a reference to the very same data.

```

// Assign the literal '5' into a, an Int.
var a: Int = 5

// Assign the value in a to b.
var b: Int = a

```

```
print(a) // 5
print(b) // 5

// Assign the value '10' to a
a = 10

// a now has '10', b still has '5'
print(a) // 10
print(b) // 5
```

Simple, right! Obvious, even? How about the same principle, except with the `Location` struct:

```
// Build a DeliveryRange value
var range1: DeliveryRange = DeliveryRange(range: 200,
    center: Location("44.9871,-93.2758"))

// Assign the value in range1 to range2
var range2: DeliveryRange = range1

print(range1.range) // 200
print(range2.range) // 200

// Modify the range of range1 to '100'
range1.range = 100

// range1 now has '100', b still has '200'
print(range1.range) // 100
print(range2.range) // 200
```

As with the `Int` example, `range2` didn't pick up the new value set in `range1`. The important thing is that this demonstrates the **value semantics** of working with structs. When you assign `range2` the value from `range1`, it gets an exact copy of the value. That means you can modify `range1` without also modifying the range in `range2`.

Structs everywhere

You saw how the `Location` struct and a simple `Int` have much the same copy-on-assignment behavior. This is because they are both value types, and both have value semantics.

You know structs represent values, so what exactly is an `Int`? If you were to look at the definition of `Int`, you might be a bit surprised:

```
public struct Int {
    // ...
}
```

That's right! The `Int` type is *also* a struct. In fact, many of the standard Swift types are structs: `Array`, `Float`, `Double`, `Bool`, `Dictionary`, `Set` and `String` are all defined

as structs.

In contrast to many other languages such as C or Objective-C, the use of structs to define and implement core types is a unique part of Swift's language design. Generally speaking, types such as integers, Booleans and strings are what are known as **primitive types**. This means they point to raw data in memory, such as the 64 bits representing an integer. Because Swift wraps those bits in a higher-level type, an Int can have methods, properties and even custom initializers!

Key points

- Structures, or **structs**, are a named type you can define and use in your code.
- Structs are **value types**.
- You use dot syntax to access the members of named types such as structs.
- Named types can have their own methods and properties, which are owned by values of those types.
- Struct initializers must set initial values to all stored properties.
- Value semantics means that values are copied on assignment.

Where to go from here?

Thanks to value semantics and copying, structs are *safe* and you'll never need to worry about values being shared and possibly being changed behind your back.

Structs are also very *fast* compared to their reference alternatives, which you'll learn about in the next chapter.

As you read the following chapters about the other named types in Swift, keep these key features of structs in mind as you compare and contrast the benefits of the different types.

Challenges

Challenge A: Clothing your structs

Create a T-shirt struct that has size, color and material options. Provide methods to calculate the cost of a shirt based on its attributes.

Challenge B: Battling ships

Write the engine for a Battleship-like game. If you aren't familiar with Battleship, see here: <http://abt.cm/1YBeWms>

- Use an (x,y) coordinate system for your locations.
- Make a struct for each ship. Record an origin, direction and length.
- Each ship should be able to report if a “shot” has resulted in a “hit” or is off by 1.

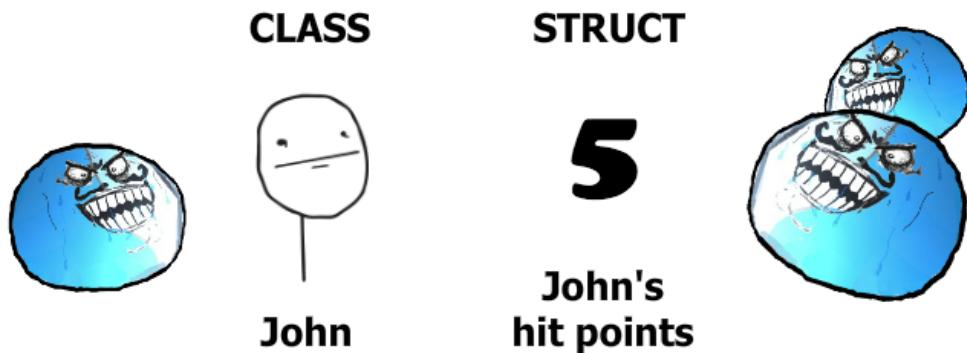
Chapter 14: Classes

By Erik Kerber

Structures introduced you to your first “named type”, allowing you to define your own types. In this chapter, you’ll get acquainted with **classes**, which are much like structures—they are named types, have stored properties and can define their own methods.

But classes, which you’ll learn are *reference types* instead of *value types*, have substantially different capabilities and benefits than their structure counterparts. While you’ll often use structures in your apps to represent values, you’ll generally use classes to represent *objects*.

What does this mean, values vs objects? Read on!



Creating classes

Consider this class definition in Swift:

```
class Person {  
    var firstName: String  
    var lastName: String  
  
    init(firstName: String, lastName: String) {
```

```
    self.firstName = firstName
    self.lastName = lastName
}

func fullName() -> String {
    return "\(firstName) \(lastName)"
}

let john = Person(firstName: "Johnny", lastName: "Appleseed")
```

That's simple enough! It may surprise you that the definition is almost identical to its struct counterpart. The keyword `class` is followed by the name of the class, and everything in the curly braces is a member of that class.

But you can also see some differences between a class and a struct: The code above explicitly sets both `firstName` and `lastName` to initial values. Unlike a struct, a class doesn't provide initializers automatically—and that means you must provide them.

If you forget to provide an initializer, Swift will flag that as an error.

```
2
3 class Person {    ● Class 'Person' has no initializers
4     var firstName: String
5     var lastName: String
6
7     func fullName() -> String {
8         return "\(firstName) \(lastName)"
9     }
10}
11
```

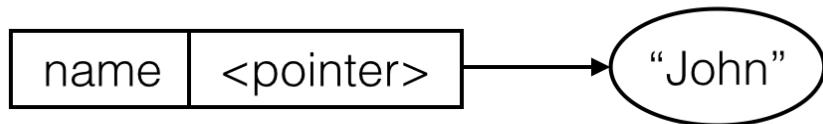
That difference aside, the initialization rules for classes and structs are very similar. Class initializers are functions marked `init`, and all stored properties must have initial values before the end of `init`, unless they are optionals.

There's much more to class initialization, but you'll have to wait until the next chapter, "Advanced Classes", which will introduce you to the concept of **inheritance**. For now, you'll get comfortable with classes in Swift by working with basic class initializers.

Reference types

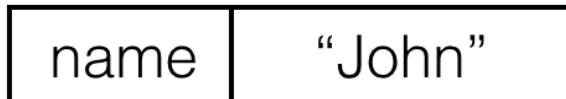
In Swift, a structure is generally an immutable value. A class, on the other hand, is a mutable reference.

What do I mean by this? As a reference type, a class stores a **pointer** to a location in memory that stores the value.



This has its advantages, as you'll soon see.

As a value type, a structure stores the actual value, providing direct access to it. This, too, has its advantages.



Later in the chapter, you'll consider the question of which type to use in a given situation. For now, let's examine how classes and structs work at the hardware level.

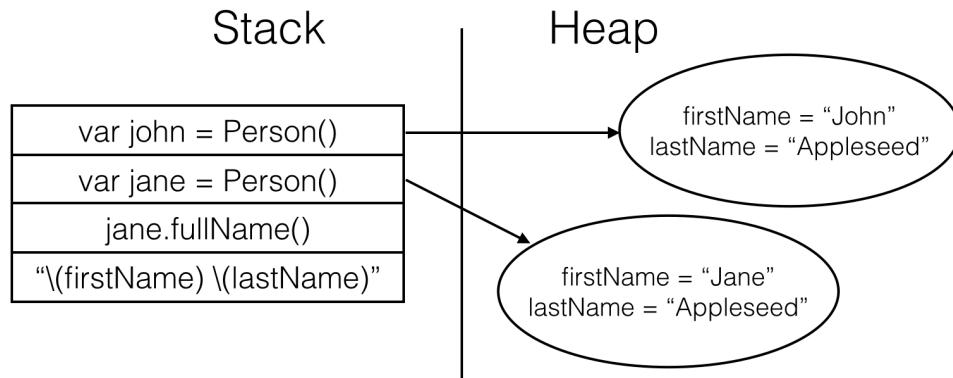
The heap vs. the stack

When you create a reference type such as class, the system stores the actual value in a region of memory known as the **heap**. A value type such as a struct resides in a region of memory called the **stack**.

Both the heap and the stack have essential roles in the execution of any program. A general understanding of what they are and how they work will help you visualize the functional differences between a class and a structure:

- The system uses the **stack** to store anything on the immediate thread of execution; it is tightly managed and optimized by the CPU. When a function creates a variable, the stack stores that variable and then destroys it when the function exits. Since the stack is so well organized, it's very efficient, and thus quite fast.
- The system uses the **heap** to store data referenced by other objects. The heap is generally a large pool of memory from which the system can request and dynamically allocate blocks of memory. The heap doesn't automatically destroy its objects like the stack does, so you're responsible for both allocating and deallocating. This makes creating and removing data on the heap a slower process, compared to on the stack.

Maybe you've already figured out how this relates to structs and classes. Take a look at the diagram below:



- When you create an instance of a class, your code requests a block of memory on the heap to store the object itself; that's the first name and last name inside the instance on the right side of the diagram. It stores the *address* of that memory in your named variable on the stack; that's the *reference* stored on the left side of the diagram.
- When you create a struct, the value itself is stored on the stack, and the heap is never involved.

This has only been a brief introduction to the dynamics of heaps and stacks, but you know enough at this point to understand the mutable nature of classes in Swift and the reference semantics you'll use to work with them.

Working with references

In the previous chapter, you saw the copy semantics involved when working with structures and other value types. Here's a little review, using the `ClimateControl` struct from that chapter:

```

var homeSetting = ClimateControl(temperature: 71.0, humidity: 30.0)
var awaySetting = homeSetting

awaySetting.temperature = 63.0

print(homeSetting.temperature) // 71.0
print(awaySetting.temperature) // 63.0

```

When you assign the value of `homeSetting` into `awaySetting`, `awaySetting` receives a *copy* of the `homeSetting` value. That way when `awaySetting.temperature` receives a new value of `63.0`, the number is only reflected in `awaySetting` while `homeSetting` still has the original value of `71.0`.

Since a class is a reference, when you assign it to a new variable or modify its data, the system does *not* copy it. Instead, the system modifies the data behind the pointer. Contrast the previous code with the following code:

```
var john = Person(firstName: "Johnny", lastName: "Appleseed")
var homeOwner = john

john.firstName = "John" // John wants to use his short name!

print(john.firstName) // "John"
print(homeOwner.firstName) // "John"
```

As you can see, `john` and `homeOwner` truly have the same value!

This implied sharing among class instances results in a new way of thinking when passing things around. For instance, if the `john` object changes, then anything *pointing* to `john` will automatically see the update. If you were using a structure, you would have to update each copy individually, or it would still have the old value of "Johnny".

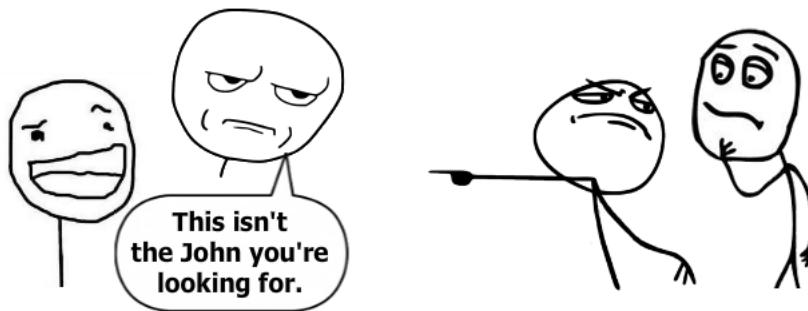
Mini-exercise

Change the value of `lastName` on `homeOwner`, then try calling `fullName()` on both `john` and `homeOwner`. What do you observe?

Object identity

In the previous code sample, it's easy to see that `john` and `homeOwner` are pointing to the same object. The code is short and both references are named variables. What if you want to see if the value behind a variable *is* John?

You might think to check the value of `firstName`, but how would you know it's the John you're looking for and not an imposter? Or worse, what if John changed his name again?



In Swift, the `==` operator lets you check if the *identity* of one object is equal to the identity of another:

```
print( john == homeOwner ) // true
```

Just as the `==` operator checks if two *values* are equal, the `==` identity operator compares the memory address of two *references*. In other words, it tells you whether the value of the pointers on the stack are the same, meaning they point to the same block of data on the heap.

That means this `==` operator can tell the difference between the John you're looking for and an imposter-John.

```
var john = Person(firstName: "John", lastname: "Appleseed")
var imposter = Person(firstName: "John", lastname: "Appleseed")

homeOwner = john

print( john === homeOwner ) // true
print( john === imposter ) // false
print( imposter === homeOwner ) // false

homeOwner = imposter

print( john === homeOwner ) // false

homeOwner = john
print( john === homeOwner ) // true
```

You'll learn later in Chapter 19, "Protocols" that you can use `==` with classes and reference types, as well.

Mini-exercise

Write a function `memberOf(person: Person, group: [Group]) -> Bool` that will return true if person can be found inside group, and false if it is not.

Test it by creating two arrays of five Person objects for group and using john as the person. Put john in one of the arrays, but not in the other.

Methods and mutability

You've seen how a class's mutability is one its most important attributes. Up to this point, you've been mutating classes similarly to how you've modified structs—meaning you've manipulated a class object's stored properties from *outside* of the object.

Because classes are mutable, it's also possible for them to mutate *themselves*.

```
struct Grade {
    let letter: String
    let points: Double
    let credits: Double
}

class Student {
    var firstName: String
    var lastName: String
    var grades: [Grade] = []

    init(firstName: String, lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
    }
}
```

```
func recordGrade(grade: Grade) {
    grades.append(grade)
}
```

Somewhat like the Person class, the Student class has a `firstName` and a `lastName` property. It also has an array of Grade values that represent that student's recorded grades.

Note that `recordGrade(_:)` can mutate the array `grades` by adding more values to the end. That means you can use that method rather than accessing the `grades` array yourself:

```
let jane = Student(firstName: "Jane", lastName: "Appleseed")
let history = Grade(letter: "B", points: 9.0, credits: 3.0)
let math = Grade(letter: "A", points: 16.0, credits: 4.0)

jane.recordGrade(history)
jane.recordGrade(math)
```

When you call `recordGrade(_:)` and pass in a grade, it's the *class itself* that modifies its own stored property.

If you'd tried this with a struct, you'd have wound up with a build failure, because structures are usually immutable. Remember, when you change the value of a struct, instead of modifying the value, you're making a *new* value.

Note: Generally speaking, immutability semantics are a very useful part of how structures work. Sometimes you might need to mutate struct value though, and you'll learn how to allow your structs to mutate their values in Chapter 18, "Methods."

Mini-exercise

Write a method on `Student` that returns the student's GPA.

Introducing access control

Swift includes a language feature known as **access control** that allows you to specify if other code can see and use entities such as functions, properties, or even types. The three levels of access control available to you are **public**, **internal**, and **private**, and can prevent code from being visible to other areas of the codebase depending on which level is specified.

In brief, the 3 levels provide access control provide the following scope:

- **public:** Entities are available to code inside the module it is defined in, as well as

any module importing the module it is defined in.

- **internal:** Entities are available to code inside the module it is defined in, but not to code in a module that imports the module it is defined in.
- **private:** Entities are available only to the same file they are defined in.

Not specifying an access modifier results in an entity defaulting to **internal**.

You can specify access control at declaration using the keyword `public`, `private`, or `internal` before the variable, function, or type name:

```
public publicString: String = "Everyone can see me!"

internal InternalClass {
    private func sayHi() {
        print("Hi!")
    }

    func speak() {
        sayHi()
    }
}
```

In the code above, `publicString` will be available to all modules, `InternalClass` will only be visible to the module you define the class in, and `sayHi()` will only be available within the same file.

Note: "Nested" declarations such as properties or methods inside of a class or struct cannot have "greater" level of access than their nested type. In other words, you could not declare `speak()` "public" because `InternalClass` has a more restrictive access control of "internal". This is because if you were never able to "see" the `InternalClass` type from another module, you certainly couldn't access any of its members even if they are public!

Attempting to use any of these entities when they are not in the scope of the access modifier will result in a build failure (and likely no autocomplete help from your IDE!):

```
// Assuming a different file from the previous snippet

print(publicString) // "Everyone can see me!"

let myClass = InternalClass()
myClass.sayHi() // Build error!
myClass.speak() // "Hi!"
```

In this example you will get a build error trying to invoke `sayHi()` because it is declared `private` and you are writing this code from another file. However, you are able to call `speak()` because it is an `internal` method and can still be called from another file. `speak()` in turn is still allowed to call the `private` `sayHi()` because they

are both defined in the same file.

Calling the private sayHi() through a more accessible speak() give you a preview into the idea of "encapsulation", which you will learn about next!

Access control and encapsulation

When you create a class or struct with methods and properties, by default anything in your code can both see and use those methods and properties. If you recall, even though there is a recordGrade(grade:) in Student, one could still see and modify the grades array directly:

```
class Student {
    var firstName: String
    var lastName: String
    var grades: [Grade] = []

    init(firstName: String, lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
    }

    func recordGrade(grade: Grade) {
        grades.append(grade)
    }
}

let jane = Student(firstName: "Jane", lastName: "Appleseed")
let history = Grade(letter: "B", points: 9.0, credits: 3.0)

jane.recordGrade(history)
jane.grades.append(history) // Hey! I didn't want you to do that!
```

This represents the traditional case for something in programming called **encapsulation**. The idea of encapsulation is that you want to control how the actual data (grades) is operated on by providing methods (such as recordGrade(grade:)) that operate on that data. When you wrote the grades array, it was only meant as a storage mechanism for grades entered through recordGrade(grade:).

If code is able to access grades directly, it could bypass other important code. Suppose you were to modify recordGrade(grade:) to perform additional logic to recording the grade itself:

```
func recordGrade(grade: Grade) {
    if grade.letter == "F" &&
        (grades.contains { $0.letter == "F" }) {
        // Second F! Double-secret probation!
    }
    grades.append(grade)
}
```

If code were to access grades directly, code could add a grade to the student's

record and bypass this new code you wrote!

```
let history = Grade(letter: "F", points: 0.0, credits: 3.0)  
jane.recordGrade(history) // Double secret probation!  
jane.grades.append(history) // Sneaky!
```

To prevent this from happening, you can add an access modifier to grades to only allow it to be used by the class:

```
class Student {  
    var firstName: String  
    var lastName: String  
    private var grades: [Grade] = []  
  
    //...  
}
```

Now, attempting to record the grades using anything other than recordGrade(grade:) will result in a build error:

```
let history = Grade(letter: "F", points: 0.0, credits: 3.0)  
jane.recordGrade(history) // Double secret probation!  
jane.grades.append(history) // Build error!
```

Note: Because private in Swift will be applied at the file level and not at the class level, you can ensure it is only used within the class by giving the Student class its own file.

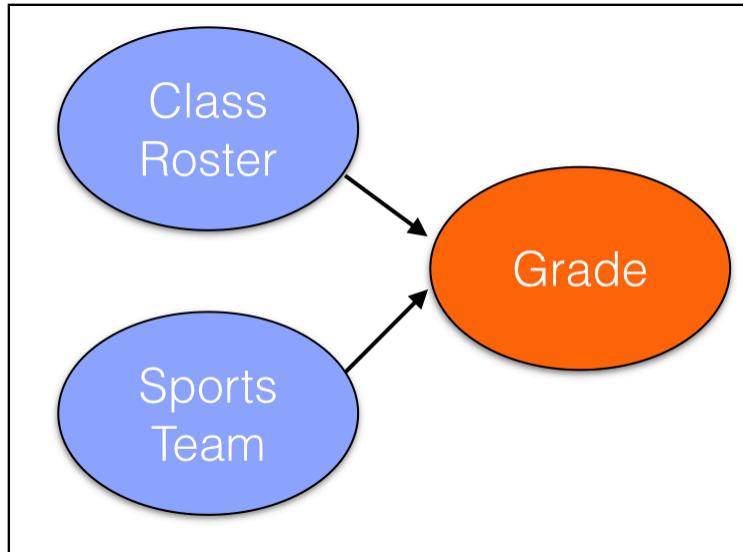
Mini-challenge

Make both firstName and lastName private as well. Create a way to update the student's name that doesn't involve accessing the name properties directly.

Understanding state and side effects

The referenced and mutable nature of classes leads to numerous programming possibilities as well as many concerns. If you update an object with a new value, then every reference to that object will also see the new value.

You can use this to your advantage. Perhaps you pass a Student object to a sports team, a report card and a class roster. Imagine all of these entities need to know the student's grades, and because they all point to the same object, they'll all see new grades as the object automatically records them.



The result of this sharing is that class objects have **state**. Changes in state can sometimes be obvious, but often they're not.

To illustrate this, consider the following code which introduces a side effect:

```
class Student {  
    // ...  
    var credits: Double = 0.0  
  
    func recordGrade(grade: Grade) {  
        grades.append(grade)  
        totalCredits += grade.credits  
    }  
}
```

In this slightly modified example of `Student`, `recordGrade(_:_)` now adds the number of credits to the `credits` property. Calling `recordGrade(_:_)` has the side effect of updating `credits`.

Now, observe how side effects can result in non-obvious behavior:

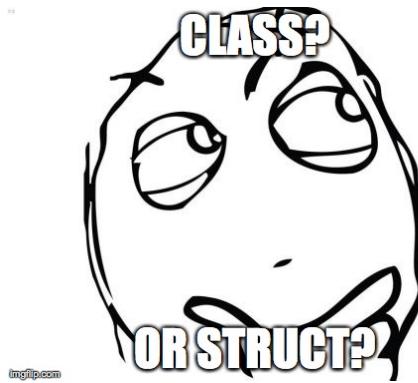
```
let jane = Student(firstName: "Jane", lastName: "Appleseed")  
var history = Grade(letter: "B", points: 9.0, credits: 3.0)  
var math = Grade(letter: "C", points: 16.0, credits: 2.0)  
  
jane.recordGrade(history)  
jane.recordGrade(math)  
  
print(jane.credits) // 5.0  
  
// The teacher made a mistake; the class has 4 credits  
math.credits = 4.0  
jane.recordGrade(math)  
  
print(jane.credits) // 9.0!
```

Whoever wrote the modified Student class did so somewhat naïvely by assuming that the same grade won't get recorded twice! Because a class can modify itself, you need to be careful about unexpected behavior around shared references.

While confusing in a small example such as this, mutability and state could be extremely jarring as classes grow in size and complexity. Situations like this would be much more common with a Student class that scales to 20 stored properties and has 10 methods.

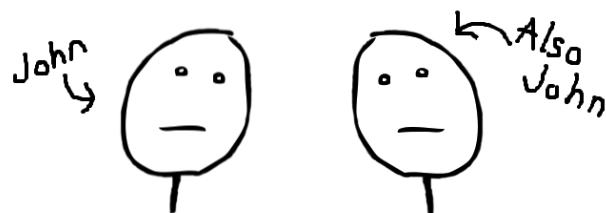
When to use a class versus a struct

Now that you know the differences and similarities between a class and a struct, you may be wondering "How do I know which to use?"



Copy vs. reference semantics

While there are no hard and fast rules, one strategy is to think about copy versus reference semantics and use structures as *values*, and classes as *things*. A temperature or a delivery time range is a value, while a person or a student is a thing. If you changed the name of a person, for instance, you wouldn't expect there to be two *copies* of that person!



Speed

Speed considerations may also come into play, as structs are created on the (faster) stack while classes are created on the (slower) heap. If you will have *many* instances of an object created (hundreds to many thousands), or if the object will only exist in memory for a very short time, then you should generally lean towards using a struct. If your object will have a longer lifecycle in memory, or if you will

create relatively few instances, then creating class instances on the heap generally won't create much overhead.

For instance, you may want to use a struct to calculate the total distance of a running route using many GPS-based waypoints, like the `Location` struct you used earlier. Not only will you create many waypoints, but they will also be created and destroyed quickly as you modify the route.

Conversely, you could use a class for an object to store route history as you'll only have one object for each user, and you would likely use the same history object for the lifetime of the user.

Minimalist approach

Another approach is to use only what you need. If your data will never change or you need a simple data store, then use structures. If you need to update your data and you need it to contain logic to update its own state, then use classes. Often, it's best to begin with a struct. If later you need the added capabilities of a class, then convert the struct to a class.

Structures vs. classes recap

Let's review the attributes of structs and classes:

Structures

- Implicit copying of values
- Data is immutable
- Useful for representing values
- Fast memory allocation (stack)

Classes

- Implicit sharing of objects
- Data is mutable
- Useful for representing things
- Slower memory allocation (heap)

Key points

- Like structures, classes are a “named type” that can have stored properties and methods.
- Classes are references that are shared on assignment.

- Classes are mutable.
- Mutability introduces state, which adds another level of complexity when managing your class objects.
- Use classes when you want reference semantics, and structures when you want value semantics.

Where to go from here?

Classes should feel familiar to you after having learned about structs. However, there are enough subtle differences around copying, sharing and mutability that it's important to keep them distinct in your head!

When deciding between a struct and a class, you've seen some of the factors to take into consideration—and there will be more things to consider in the very next chapter!

Challenges

Challenge A: Movie lists - benefits of reference types

Imagine you're writing a movie-viewing application in Swift. Users can create "lists" of movies and share those lists with other users.

Create a User and a List class that uses reference semantics to help maintain lists between users.

- User - Has a method `addList(list:)` which adds the given list to a dictionary of List objects (using the name as a key), and a `getList(name:)` → List which will return the List for the provided name.
- List - Contains a name and a Set of movie titles. A `printList()` method will print all the movies in the list.
- Create jane and john users and have them create and share lists. Have both jane and john modify the same list and call `printList()` from both users. Are all the changes reflected?
- What happens when you implement the same with structs. What problems do you run into?

Challenge B: T-Shirt store - class or struct?

Your challenge here is to build a set of objects to support a t-shirt store. Decide if each object should be a class or a struct, and why.

- TShirt - Represents a shirt style you can buy. Each TShirt has a size, color, price, and an optional image on the front.
- User - A registered user of the t-shirt store app. A user has a name, email, and a ShoppingCart (below).
- Address - Represents a shipping address, containing the name, street, city, and zip code.
- ShoppingCart - Holds a current order, which is composed of an array of TShirt that the User wants to buy, as well as a method to calculate the total cost. Additionally, there is an Address that represents where the order will be shipped.

Bonus: After you've decided on class or struct ofr each object, go ahead and implement them all!

Chapter 15: Advanced Classes

By Erik Kerber

The previous chapter introduced you to the basics of defining and using classes in Swift. Classes are, in general, much more sophisticated types than their struct counterparts.

Classes introduce inheritance, overriding, and polymorphism which give them added abilities compared to structs. This in turn requires considerations for initialization, class hierarchies, and understanding the class lifecycle in memory.

This chapter will introduce you to the finer points of classes in Swift, and help you understand how you can create, use and manage complex classes.

Introducing inheritance

In the previous chapter, you saw a pair of class examples: Person and Student.

```
class Person {
    var firstName: String
    var lastName: String

    init(firstName: String, lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
    }
}

class Student {
    var firstName: String
    var lastName: String
    var grades: [Grade] = []

    init(firstName: String, lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
    }
}
```

It's not difficult to see that there's an incredible amount of redundancy between Person and Student. Maybe you've also noticed that a Student *is* a Person!

This simple case demonstrates the idea behind class inheritance. Much like in the real world, where you can think of a student as a person, you can represent the same relationship in code:

```
class Person {
    var firstName: String
    var lastName: String

    init(firstName: String, lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
    }
}

class Student: Person {
    var grades: [Grade] = []
}
```

In this modified example, the Student class now **inherits** from Person, indicated by a colon after the naming of Student, followed by the class from which Student inherits, which in this case is Person.

Through inheritance, Student automatically gets the stored properties declared in the Person class, as well as the initializer and any methods defined within. In code, it would be accurate to say that a Student *is* a Person.

With drastically less code duplication, you can now create Student objects that have all the properties of a Person:

```
let john = Person(firstName: "Johnny", lastName: "Appleseed")
let jane = Student(firstName: "Jane", lastName: "Appleseed")
```

A class that inherits from another class is known as a **subclass**, and the class from which it inherits as is known as a **superclass** or a **base class**. The rules for subclassing are fairly simple:

- A Swift class can inherit from only one other class, a concept known as **single inheritance**.
- There's no limit to the depth of subclassing, meaning you can subclass from a class that is *also* a subclass, like below:

```
class Student: Person {
    var grades: [Grade] = []
}

class StudentAthlete: Student {
    // ...
}
```

```
class StarStudentAthlete: StudentAthlete {  
    // ...  
}
```

A chain of subclasses is called a **class hierarchy**. In this example, the hierarchy would be StarStudentAthlete -> StudentAthlete -> Student -> Person

Polymorphism

The Student/Person relationship demonstrates a computer science concept known as **polymorphism**. In brief, polymorphism is a programming languages ability to treat an object differently based on context.

A StarStudentAthlete is of course a StarStudentAthlete, but it is also a Person. Because it derives from Person, you can use a StarStudentAthlete object anywhere a Person object is expected.

This example demonstrates how you can treat a StarStudentAthlete as a Person:

```
func phonebookName(person: Person) -> String {  
    return "\u{0028}person.lastName\u{0029), \u{0028}person.firstName\u{0029"  
}  
  
let person = Person(firstName: "Johnny", lastName: "Appleseed")  
let star = StarStudentAthlete(firstName: "Jane", lastName: "Appleseed")  
  
print( phonebookName(person) ) // Appleseed, John  
print( phonebookName(star) ) // Appleseed, Jane
```

Because StarStudentAthlete derives from Person, it is a valid input into the function phonebookName(_:_). More importantly, the function has no idea that the object passed in is anything *other* than a regular Person. It can only observe the elements of StarStudentAthlete that are defined in the Person base class.

With the polymorphism characteristics provided by class inheritance, Swift is treating the object pointed to by star differently based on the context. This can be particularly useful to you when you have diverging class hierarchies, but want to have code that operates on a common type or base class.

Inheritance, methods and overrides

Subclasses receive all stored properties and methods defined in their superclass, plus any additional properties and methods the subclass defines for itself. In that sense, subclasses are *additive*; for example, Student could add an additional property and method:

```
class Student: Person {  
    var grades: [Grade] = []  
  
    func recordGrade(grade: Grade) {  
        grades.append(grade)
```

```
    }
```

Here, `Student` objects have a property and method defined to handle the student's grade. This property and method wouldn't be available to `Person`, but it *would* be available to `Student` subclasses such as `StudentAthlete`.

Besides creating their own methods, subclasses can *override* methods defined in their superclass. Say for student athletes, they become ineligible for the athletics program if they're failing three or more classes. That means you need to keep track of failing grades somehow.

```
27 class StudentAthlete: Student {
28     var failedClasses: [Grade] = []
29
30     override func recordGrade(grade: Grade) {
31         super.recordGrade(grade)
32
33         if grade.letter == "F" {
34             failedClasses.append(grade)
35         }
36     }
37
38     func athleteIsEligible() -> Bool {
39         return failedClasses.count < 3
40     }
41 }
42 }
```

In this example, the `StudentAthlete` class has overridden `recordGrade(_:_)`, so it can keep track of any courses the student has failed. The `StudentAthlete` class then has its own method, `athleteIsEligible()`, that uses this information to determine the athlete's eligibility.

When overriding a method, use the `override` keyword before the method declaration. If your subclass were to have an identical method declaration as its superclass, but you omitted the `override` keyword, Swift would indicate a build error:

```
27
28 class StudentAthlete: Student {
29     var failedClasses: [Grade] = []
30
31     func recordGrade(grade: Grade) { // Overriding declaration requires an 'override' keyword
32         super.recordGrade(grade)
33
34         if grade.letter == "F" {
35             failedClasses.append(grade)
36         }
37     }
38
39     func athleteIsEligible() -> Bool {
40         return failedClasses.count < 3
41     }
42 }
```

This make it very clear whether a method is an override of an existing one or not.

Introducing super

You may have also noticed the line `super.recordGrade(grade)` in the overridden method. The `super` keyword is similar to `self`, except it will invoke the method in the nearest implementing superclass. In the example of `recordGrade(grade:)`, calling `super.recordGrade(grade)` will execute the method as defined in the `Student` class.

Remember how inheritance let you define `Person` with first name and last name properties and avoid repeating those properties in subclasses? Similarly, being able to call the superclass methods means you can write the code to actually record the grade once in `Student` and then call "up" to it as needed in subclasses.

Although the compiler doesn't enforce it, it's important to call `super` when overriding a method in Swift. The `super` call is what will record the grade itself in the `grades` array, because that behavior isn't duplicated in `StudentAthlete`. Calling `super` is also a way of avoiding the need for duplicate code in `StudentAthlete` and `Student`.

When to call super

As you may notice, exactly *when* you call `super` can have an important effect on your overridden method.

Suppose you have an alternate implementation of `recordGrade(_:)` that recalculates the `failedClasses` each time a grade is recorded:

```
override func recordGrade(grade: Grade) {
    var newFailedClasses: [Grade] = []
    for grade in grades {
        if grade.letter == "F" {
            newFailedClasses.append(grade)
        }
    }
    failedClasses = newFailedClasses
    super.recordGrade(grade)
}
```

This version of `recordGrade(grade:)` uses the `grades` array to find the current list of failed classes. If you've spotted a bug in the code above, good job! Because you call `super` last, if the new `grade.letter` is an F, the code won't update the `failedClasses` array.

While it's not a hard rule, it's generally best practice to call the `super` version of a method first when overriding. That way, the superclass won't experience any side effects introduced by its subclass, and the subclass won't need to know the superclass's implementation details.

Preventing inheritance

Sometimes you'll want to disallow subclasses of a particular class. Swift provides the `final` keyword for you to guarantees a class will never get a subclass:

```
final class Student: Person {  
    //...  
}  
  
// Build error!  
class StudentAthlete: Student {  
    //...  
}
```

By marking the `Student` class `final`, you tell the compiler to prevent any classes from inheriting from `Student`. This can remind you—or others on your team!—that a class wasn't designed to have subclasses.

Additionally, you can mark individual *methods* as `final`, if you want to allow a class to have subclasses, but protect individual methods from being overridden:

```
class Student: Person {  
    final func recordGrade(grade: Grade) {  
        //...  
    }  
}  
  
class StudentAthlete: Student {  
  
    // Build error!  
    override func recordGrade(grade: Grade) {  
        //...  
    }  
}
```

There are benefits to initially marking any new class you write as `final`. Not only does it tell the compiler it doesn't need to look for any more subclasses, which can shorten compile time, it also requires you to be very explicit when deciding to subclass a class previously marked `final`.

Inheritance and class initialization

The previous chapter briefly introduced you to class initializers, which are similar to their struct counterparts. With subclasses, there are a few more considerations with regard to how you set up initial values.

Let's introduce a modification to the `StudentAthlete` class that adds a list of sports an athlete plays:

```
class StudentAthlete: Student {  
    var sports: [String]  
  
    //...  
}
```

Because `sports` doesn't have an initial value, `StudentAthlete` must provide one in its own initializer:

```
class StudentAthlete: Student {  
    var sports: [String]  
  
    init(sports: [String]) {  
        self.sports = sports  
        // Build error - super.init isn't called before  
        // returning from initializer  
    }  
}
```

Uh-oh! The compiler complains that you didn't call `super.init` by the end of the initializer:

```
27  
28 class StudentAthlete: Student {  
29     var sports: [String]  
30  
31     init(sports: [String]) {  
32         self.sports = sports  
33         // Build error - super.init isn't called before  
34         // returning from initializer  
35     }  
36 }
```

The line at index 35 is highlighted with a red background and a red exclamation mark icon, indicating a build error: "Super.init isn't called before returning from initializer".

Initializers *require* calling `super.init` because without it, the superclass won't be able to provide initial states for all its stored properties—in this case, `firstName` and `lastName`. Let's make the compiler happy:

```
class StudentAthlete: Student {  
    var sports: [String]  
  
    init(firstName: String, lastName: String, sports: [String]) {  
        self.sports = sports  
        super.init(firstName: firstName, lastName: lastName)  
    }  
}
```

The initializer now calls the initializer of its superclass, and the build error is gone. Notice that the initializer now takes in a `firstName` and a `lastName` to satisfy the requirements for calling the `Person` initializer.

You also call `super.init` after you initialize the `sports` property, which is an enforced rule. In Swift, this is referred to as **two-phase initialization**.

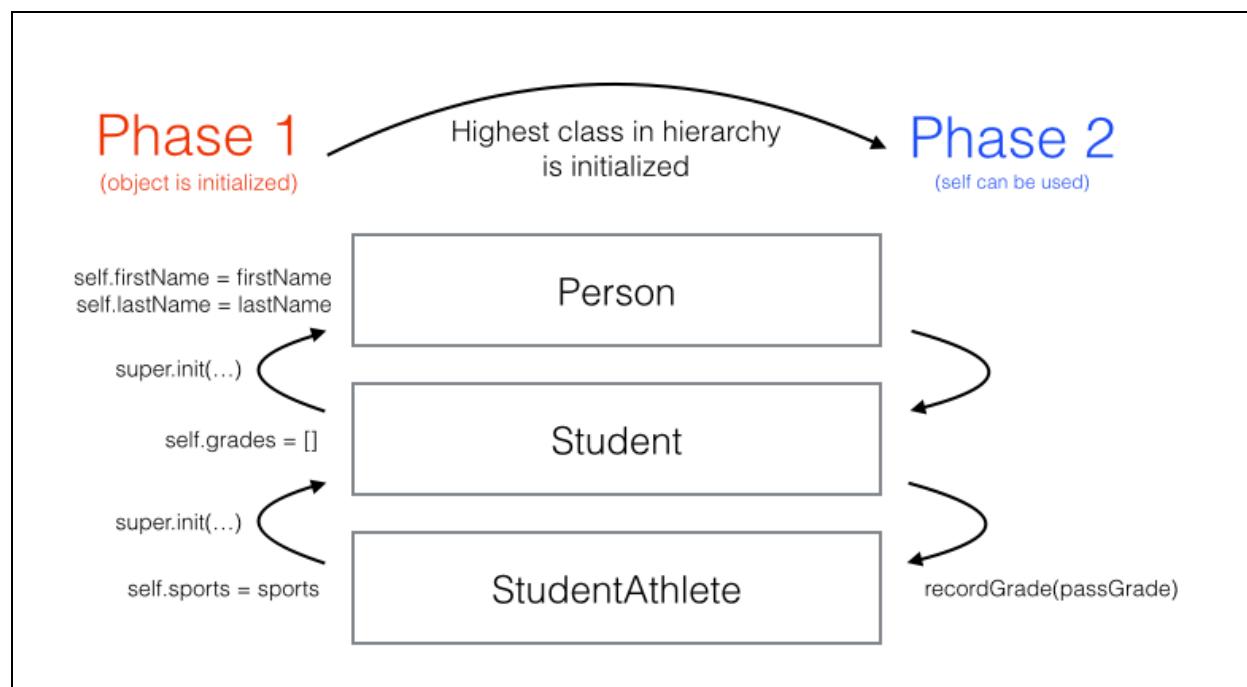
Two-phase initialization

Because of Swift's requirement that all stored properties have initial values, initializers in subclasses must adhere to Swift's convention of two-phase initialization.

- **Phase one:** Initialize all of the stored properties in the class instance, from the bottom to the top of the class hierarchy. You can't use properties and methods until phase 1 is complete.
- **Phase two:** You can now use properties and methods, as well as initializations that require the use of `self`.

Without two-phase initialization, methods and operations on the class might interact with properties before they've been initialized.

The transition from phase one to phase two happens after you've initialized all stored properties in the base class of a class hierarchy. In the scope of a subclass initializer, you can think of this as coming after the call to `super.init`.



Here's the `StudentAthlete` class again, with athletes automatically getting a starter grade:

```
class StudentAthlete: Student {
    var sports: [String]

    init(firstName: String, lastName: String, sports: [String]) {
        // 1
        self.sports = sports

        // 2
        let passGrade = Grade(letter: "P", points: 0.0, credits: 0.0)

        // 3
        super.init(firstName: firstName, lastName: lastName)

        // 4
        recordGrade(passGrade)
    }
}
```

The above *complete* initializer shows two-phase initialization in action:

1. First, you initialize the `sports` property of `StudentAthlete`. This is part of the first phase of initialization and has to be done early, before you call the superclass initializer.
2. Although you can create local variables for things like grades, you can't call `recordGrade(_:)` yet because the object is still in the first phase.
3. You call `super.init`. When this returns, you know that you've also initialized every class in the hierarchy, because the same rules are applied at every level.
4. After `super.init` returns, the initializer is in phase 2, so you call `recordGrade(_:)`.

Mini-exercise

Add code that references `self` at every level of the `Person/Student/StudentAthlete` class hierarchy. What's different in the two-phase initialization in the base class `Person`, as compared to the others?

Required and convenience initializers

You already know it's possible to have multiple initializers in a class, which means you could potentially call *any* of those initializers from a subclass.

Often, you'll find that your classes have various initializers that simply provide a "convenient" way to initialize an object:

```
class Student: Person {
    var grades: [Grade]
```

```
    init(firstName: String, lastName: String) {
        grades = []
    }

    init(transfer: Student) {
        self.firstName = transfer.firstName
        self.lastName = transfer.lastName
    }
}
```

In this example, the `Student` class can be built with another `Student` object. Perhaps the student switched majors? Both initializers fully set the first and last names.

Both of these initializers are now available to subclasses, as well:

```
class StudentAthlete: Student {
    var sports: [String]

    init(firstName: String, lastName: String, sports: [String]) {
        self.sports = sports
        super.init(firstName: firstName, lastName: lastName)
    }

    init(transfer: Student, sports: [String]) {
        self.sports = sports
        super.init(transfer: transfer)
    }
}
```

Subclasses of `Student` could potentially rely on the `Student`-based initializer when they make their call to `super.init`. Additionally, the subclasses might not even provide a method to initialize with first and last names. You might decide that the first and last name-based initializer is important enough that you want it to be available to *all* subclasses.

Swift supports this through the language feature known as **designated initializers**:

```
class Student: Person {
    var grades: [Grade]

    override required init(firstName: String, lastName: String) {
        grades = []
        super.init(firstName: firstName, lastName: lastName)
    }

    //...
}
```

In the modified version of `Student` above, the first and last name-based initializer has been marked with the keyword **required**. This keyword will force all subclasses of `Student` to not only implement this initializer, but also use this initializer when they make their calls to `super.init`.

Now that there's a required initializer on `Student`, `StudentAthlete` *must* override and implement it too.

```
class StudentAthlete: Student {
    var sports: [String]

    // Now required by the compiler!
    override required init(firstName: String, lastName: String) {
        self.sports = []
        super.init(firstName: firstName, lastName: lastName)
    }

    init(firstName: String, lastName: String, sports: [String]) {
        self.sports = sports
        super.init(firstName: firstName, lastName: lastName)
    }

    init(transfer: Student, sports: [String]) {
        self.sports = sports
        super.init(transfer: transfer)
    }
}
```

With the code above, you require the `StudentAthlete` class to implement the initializer marked `required` in the superclass.

You can also mark an initializer as a **convenience** initializer:

```
class Student: Person {
    var grades: [Grade]

    override required init(firstName: String, lastName: String) {
        grades = []
        super.init(firstName: firstName, lastName: lastName)
    }

    convenience init(transfer: Student) {
        self.init(firstName: transfer.firstName, lastName: transfer.lastName)
    }
}
```

The compiler forces a convenience initializer to call a required initializer, instead of handling the initialization of stored properties itself.

You might want to mark an initializer as convenience if you only use that initializer as an easy way to initialize an object, but you still want it to leverage one of your required initializers.

Here's a summary of the compiler rules for using designated and convenience initializers:

1. A designated initializer must call a designated initializer from its immediate superclass.
2. A convenience initializer must call another initializer from the same class.

3. A convenience initializer must ultimately call a designated initializer.

Mini-exercise

Create two more convenience initializers on Student. What other initializers are you able to call?

When and why to subclass

This chapter has introduced you to class inheritance, along with the numerous programming techniques that subclassing enables.

But you might be asking, "When should I subclass?" That's an important question.

Rarely is there a right or wrong answer to that question, but understanding the trade-offs can help you make the best decision for any particular case.

Using the Student and StudentAthlete classes as an example, you might decide you can simply put all of the characteristics of StudentAthlete into Student:

```
class Student: Person {  
    var grades: [Grade]  
    var sports: [Sport]  
  
    //...  
}
```

In reality, this *could* solve all of the use cases for your needs. A Student that doesn't play sports would simply have an empty sports array, and you would avoid some of the added complexities of subclassing.

Single responsibility

In software development, there is a guideline known as the **single responsibility principle**. This principle states that any class should have a single concern only, and it should own all the functionality it uses. In Student/StudentAthlete, you might argue that it shouldn't be the Student class's job to encapsulate responsibilities that only make sense to student athletes.

Type constraints

By subclassing, you are creating an additional type. Thanks to Swift's type system, this means you can declare properties or behavior based on objects you know are student athletes and not regular students:

```
class Team {  
    var players: [StudentAthlete]  
  
    func teamEligable() -> Bool {
```

```
    for player in players {
        if !player.isPlayerEligible() {
            return false
        }
    }
    return true
}
```

A team has players who are student athletes. If you tried to add a regular Student object to the array of players, the type system wouldn't allow it. This can be a useful feature to have the compiler help you enforce the logic and requirement of your system.

Shared base classes

It's very common to have a shared base class that is subclassed many times by classes that have mutually exclusive behavior:

```
/// A button that can be pressed.
class Button {
    func press() {}
}

/// A button that is composed entirely of an image.
class ImageButton: Button {
    var image: Image
}

/// A button that renders as text.
class TextButton: Button {
    var text: String
}
```

In this example, you can imagine numerous Button subclasses that share only the fact that they can be pressed. The ImageButton and TextButton classes likely have entirely different mechanisms to render the appearance of a button, so they might have to implement their own behavior when the button is pressed.

You can see here how storing `image` and `text`—not to mention any other kind of button there might be—in the `Button` class would quickly become impractical. It makes sense for `Button` to be concerned with the `press` behavior, and the subclasses to handle the actual look and feel of the button.

Extensibility

Sometimes you simply must subclass if you're extending the behavior of code you don't own. In the example above, it's possible `Button` is part of a framework you're using, and there's no way you can modify the source code to fit your needs.

In that case, subclass `Button` so you can add your custom subclass to something that's expecting an object of type `Button`.

Understanding the class lifecycle

In the previous chapter, you learned how classes are created in memory and that they're stored in the heap, as compared with value types like structs, which are created on the stack.

Recall that in contrast to objects on the heap, objects on the stack are automatically destroyed when they go out of scope. Objects on the heap, such as those defined by classes, are *not* automatically destroyed, because the heap is simply a giant pool of memory. Without the utility of the call stack, there's no automatic way for a process to know that a piece of memory will no longer be in use.

In Swift, the mechanism for deciding when to clean up unused objects on the heap is known as **reference counting**. In short, each object has a reference count that's incremented for each object with a reference to that object, and decremented each time a reference is removed.

Note: You might see the reference count called a "retain count" in other books and online resources. They refer to the same thing!

When a reference count reaches zero, that means the object is now abandoned since nothing in the system is holding a reference to it. When that happens, Swift will clean up the object.

Here's a demonstration of how the reference count changes for an object. Note that there's only one actual object created in this example; the one object just has many references to it.

```
// Person object has a reference count of 1 (john variable)
var john = Person(firstName: "Johnny", lastName: "Appleseed")

// Reference count 2 (john, anotherJohn)
var anotherJohn: Person? = john

// Reference count 6 (john, anotherJohn, 4 references)
// The same reference is inside both john and anotherJohn
var lotsaJohns = [john, john, anotherJohn, john]

// Reference count 5 (john, 4 references in lotsaJohns)
anotherJohn = nil

// Reference count 1 (john)
lotsaJohns = []

// Reference count 0!
john = Person(firstName: "Johnny", lastName: "Appleseed")
```

In this example, you don't have to do any work yourself to increase or decrease the

object's reference count. That's because Swift has a feature known as **automatic reference counting** or **ARC**.

While some languages require references to increment and decrement reference counts in *your* code, the Swift compiler adds these calls automatically at compile time.

Note: If you use a low-level language like C, you're required to manually free memory you're no longer using yourself. Higher-level languages like Java and C# use something called **garbage collection**. In that case, the runtime of the language will search your process for references to objects, before cleaning up those that are no longer in use.

Deinitialization

When an object's reference count reaches 0, Swift removes the object from memory and marks that memory as free.

There's a special method on classes in Swift that runs when an object's reference count reaches 0, but before Swift removes the object from memory.

Take a look:

```
class Person {  
    //...  
  
    deinit {  
        print("\(firstName) \(lastName) is being removed from memory!")  
    }  
}
```

Much like `init` is a special method in class initialization, `deinit` is a special method that handles deinitialization. Unlike `init`, `deinit` isn't required and is automatically invoked by Swift. You also aren't required to override it or call `super` within it. Swift will make sure to call each class deinitializer.

```
// Reference count == 1  
var john = Person(firstName: "Johnny", lastName: "Appleseed")  
  
// Reference count == 2  
var anotherJohn: Person? = john  
  
// Reference count = 1  
john = Person(firstName: "Johnny", lastName: "Appleseed")  
  
// Reference count = 0  
anotherJohn = nil
```

If you try this in a playground, you'll see the message "Johnny Appleseed is being removed from memory!" in the debug area.

What you do in an deinitializer is up to you. Often you'll use it to clean up other resources, save state to a disk, or execute any other logic you might want when an object goes out of scope.

Mini-exercises

Modify the Student class to have the ability to record the student's name to a list of graduates. Add the name of the student to the list when the object is deallocated.

Retain cycles and weak references

Because classes in Swift rely on reference counting to remove them from memory, it's important to understand the concept of a **retain cycle**.

Imagine a modified version of Student that has a field representing a classmate—for example, a lab partner:

```
class Student: Person {
    var partner: Student?

    deinit {
        print("\(firstName) being deallocated!")
    }
}

var john: Student? = Student(firstName: "Johnny", lastName: "Appleseed")
var jane: Student? = Student(firstName: "Jane", lastName: "Appleseed")

john?.partner = jane
jane?.partner = john
```

Now suppose both jane and john drop out of school:

```
john = nil
jane = nil
```

If you run this in your playground, you'll notice that you don't see the message "Johnny/Jane being deallocated!", and Swift doesn't call deinit. Why is that?

It's because John and Jane each have a reference to each other, so the reference count never reaches 0! To make things worse, by assigning `nil` to `john` and `jane`, there are no more references to the initial objects. This is a classic case of a retain cycle, which leads to a software bug known as a **memory leak**.



With a memory leak, memory isn't freed up even though its practical lifecycle has ended. Retain cycles are the most common cause of memory leaks.

Fortunately, there's a way that the `Student` object can reference another `Student` without being prone to retain cycles, and that's by making the reference **weak**:

```
class Student: Person {  
    weak var partner: Student?  
  
    // ...
```

This simple modification marks the `partner` variable as weak, which will prevent the reference count of `Student` from increasing when the reference is assigned to it.

When a property isn't marked weak, it's called a **strong reference**, which is the default for all properties in Swift. As you might imagine, a strong reference means that assigning a value to the property will increase its reference count.

Key points

- **Class inheritance** is one of the most important features of classes and enables **polymorphism**.
- Swift classes use **two-phase initialization** as a safety measure to ensure all stored properties are initialized before they are used.
- Subclassing is a powerful tool, but it's good to know when to subclass. Subclass when you want to extend an object and could benefit from an "is-a" relationship between subclass and superclass, but be mindful of the inherited state and deep class hierarchies.
- Classes have their own lifecycles which are controlled by their **reference counts**.
- **Automatic reference counting**, or **ARC**, handles reference counting for you automatically, but it's important to watch out for **retain cycles**.

Where to go from here?

Classes and structs are the most common types you'll use to model things in your apps, from students to grades to people to teams. As you've seen, classes have some more complexity to them, with reference semantics and inheritance.

In the next chapter, you'll learn about the third type available to you: enumerations. After that, you'll loop back around to learn more about properties and methods—things that apply to all of classes, structs and enumerations.

Challenge

Challenge A: Visualizing the initialization chain

Create 3 simple classes called A, B, and C where C inherits from B and B inherits from A. In each class initializer, call `print("I'm A!")` in each respective classes initializer both before and after `super.init()`.

What order do you see each `print()` called in?

Challenge B: Deepen the class hierarchy

Create a subclass of `StudentAthlete` called `StudentBaseballPlayer` and include properties for position, number, and battingAverage.

- What are the benefits and drawbacks of subclassing `StudentAthlete` in this scenario?
- Can you think of an alternative to subclassing? Assume you could modify any class in the hierarchy.

Chapter 16: Enumerations

By Ben Morrow

One day in your life as a developer, you realize you're being held captive by your laptop. Determined to break from convention, you decide to set off on a long trek by foot. Of course, you need a map of the terrain you'll encounter. Since it's the 21st century, and you're fluent in Swift, you apply yourself to one final project: a custom map app.

As you code away, you think it would be nice if you could represent the cardinal directions as variables: north, south, east, west. But what's the best way to do this in code?

You could represent each value as an integer, like so:

- North: 1
- South: 2
- East: 3
- West: 4

You can see how this could quickly get confusing if you or your users happen to think of the directions in a different order. "What does 3 mean again?" To alleviate that, you might represent the values as strings, like so:

- North: "north"
- South: "south"
- East: "east"
- West: "west"

The trouble with strings, though, is that the value can be any string. What would your app do if it received "up" instead of "north"? Furthermore, when you're composing code, it's easy to miss a typo like "nrth".

Wouldn't it be great if there were a way to create a group of related values? If you find yourself headed in this... *direction*, you'll want to use an **enumeration**.

An enumeration is a list of related values that define a common type, letting you work with values in a type-safe way. The compiler will catch your mistake if your code expects a `Direction` and you try to pass in a float like 10.7 or a misspelled direction like "Souuth".

Besides cardinal directions, other good examples of related values are colors (black, red, blue), card suits (hearts, spades, clubs, diamonds) and roles (administrator, editor, reader).



Enumerations in Swift are much more powerful than they are in other languages such as C or Objective-C. They share features with the structure and class types you learned about in the previous chapters. An enumeration can have methods, computed properties and protocol declarations, all while acting as a nice state machine.

In this chapter, you'll learn how enumerations work and when they're useful. As a bonus, you'll finally discover what an optional is under the hood. Hint: They are implemented with enumerations!

Your first enumeration

Your challenge: construct a function that will determine the school semester based on the month. One way to solve this would be to use an array of strings and match them to the semesters with a `switch` statement:

```
// List out all the months in a year
let months = ["January", "February", "March", "April", "May",
    "June", "July", "August", "September", "October", "November",
    "December"]

// Figure out which semester the month belongs
func schoolSemester(month: String) -> String {
    switch month {
        case "August", "September", "October", "November", "December":
            return "Autumn"
        case "January", "February", "March", "April", "May":
            return "Spring"
        default:
            return "Not in the school year"
    }
}
```

```
// Print out the result in the playground sidebar
schoolSemester("April") // Spring
```

Running this code in a playground, you can see that the function correctly returns "Spring". But as I mentioned in the introduction, you could easily mistype a string. A better way to tackle this would be with an enumeration.

Declaring an enumeration

To declare an enumeration, you list out all the possible member values as case statements:

```
enum Month {
    case January
    case February
    case March
    case April
    case May
    case June
    case July
    case August
    case September
    case October
    case November
    case December
}
```

This code creates a new enumeration called `Month` with 12 possible member values. You can simplify the code a bit by collapsing the case statements down to one line, with each value separated by a comma:

```
enum Month {
    case January, February, March, April, May, June, July, August,
          September, October, November, December
}
```

That looks snazzy and simple. So far, so good.

Deciphering an enumeration in a function

You can rewrite the function that determines the semester so that it uses enumeration values instead of string-matching:

```
func schoolSemester(month: Month) -> String {
    switch month {
        case Month.August, Month.September, Month.October,
              Month.November, Month.December:
            return "Autumn"
        case Month.January, Month.February, Month.March, Month.April,
              Month.May:
            return "Spring"
        default:
            return "Not in the school year"
```

```
}
```

Since Swift is strongly-typed and knows the types of your variables and constants, you can simplify `schoolSemester()` by removing the enumeration name in places where the compiler already knows the type. Keep the dot prefix, but lose the enumeration name, like this:

```
func schoolSemester(month: Month) -> String {
    switch month {
        case .August, .September, .October, .November, .December:
            return "Autumn"
        case .January, .February, .March, .April, .May:
            return "Spring"
        default:
            return "Not in the school year"
    }
}
```

That's much more readable. The compiler knows the `month` is of type `Month`. Since the cases in the `switch` statement will be matched to `month`, you can leave off the type. You would test this function in a playground like so:

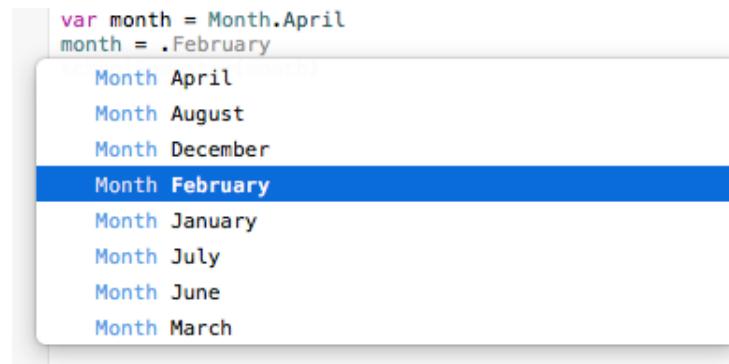
```
var month = Month.April
month = .September
schoolSemester(month) // "Autumn"
```

The variable declaration for `month` uses the full enumeration type and value. In the assignment on the next line, you can use the shorthand `.September`, since the compiler knows the type. Finally, you pass the `month` to `schoolSemester()`, where a `switch` statement matches it to the string "Autumn".

Note: Wouldn't it be nice to request the semester as `month.schoolSemester` instead? This is called a computed property and is covered in the next chapter, "Properties".

Code completion prevents typos

As an added bonus, you'll never have a typo in your enumeration member values because Xcode provides code completion:



And if you do misspell an enumeration value, the compiler will complain with an error, so you won't get too far down the line without recognizing your mistake:

```
! month = .Janury   ! Type of expression is ambiguous without more context
```

Raw values

Unlike enumeration values in C, Swift enum values, by default, are *not* backed by integers—that means January is itself the value. But you can associate a raw value with each enumeration case simply by declaring the raw value type on the enumeration name:

```
enum Month: Int {
```

As in C, if you use integers and don't specify values as you've done here, Swift will assign the values 0, 1, 2 and so on. Swift enumerations are flexible; so you can specify other raw value types like String, Float or Character.

To specify your own raw values, use the = assignment operator:

```
enum Month: Int {
    case January = 1, February = 2, March = 3, April = 4, May = 5,
        June = 6, July = 7, August = 8, September = 9,
        October = 10, November = 11, December = 12
}
```

This assigns an integer value to each enumeration case.

There's another handy shortcut here: the compiler will automatically increment the values if you provide the first one and leave off the rest:

```
enum Month: Int {
    case January = 1, February, March, April, May, June, July,
        August, September, October, November, December
}
```

You can use enumerations and never refer to the raw values if you don't need

them. But the raw values will be there behind the scenes if you ever do need them!

Accessing the raw value

Enumeration instances with raw values have a handy `rawValue` property. With the raw values in place, your enumeration has a sense of order and you can calculate the number of months left until winter break:

```
func monthsUntilWinterBreak(month: Month) -> Int {  
    return Month.December.rawValue - month.rawValue  
}  
monthsUntilWinterBreak(.April) // 8
```



Initializing with the raw value

You can use the raw value to instantiate an enumeration value with an initializer. You can use `init(rawValue:)` to do this, but if you try to use the value afterward, you'll get an error:

```
let fifthMonth = Month(rawValue: 5)  
monthsUntilWinterBreak(fifthMonth) // Error: value not unwrapped
```

There's no guarantee that the raw value you submitted exists in the enumeration, so the initializer returns an optional. Enumeration initializers with the `rawValue:` parameter are **failable initializers**, meaning if things go wrong, the initializer will return `nil`.

If you're using these raw value initializers in your own projects, remember that they return optionals. You'll need to either check them for `nil` or use optional binding. As you've learned, one way to unwrap an optional is to use `if let` syntax:

```
if let fifthMonth = Month(rawValue: 5) {  
    monthsUntilWinterBreak(fifthMonth) // 7  
}
```

That's better! Now there's no error, and `monthsUntilWinterBreak(_:)` returns "7", as expected.

Note: Wouldn't it be nice to instead run `fifthMonth.monthsUntilWinterBreak()`? To do that, you'd add a method to the enumeration. This will be covered in Chapter 18, "Methods".

Unordered raw values

Raw values don't have to be in an incremental order. Coins are a good use case:

```
enum Coin: Int {  
    case Penny = 1  
    case Nickel = 5  
    case Dime = 10  
    case Quarter = 25  
}
```



You can instantiate values of this type and access their raw values as usual:

```
let coin = Coin.Quarter  
coin.rawValue // 25
```

Mini-exercise

Create an array called `coinPurse` that contains coins. Add an assortment of pennies, nickels, dimes and quarters to it.

Associated values

You saw how with raw values, you can assign enumerations a value of a certain type, such as `Int`. That means each member value defined with the `case` keyword *must* have a unique integer to go along with it.

Associated values are similar to raw values, with a few differences:

1. Each enumeration case has zero or many associated values.
2. The associated values for each enumeration case have their own data type.
3. You define associated values with names like you would for named function parameters.

An enumeration can have raw values or associated values, but not both.

In the last mini-exercise, you defined a coin purse. Let's say you took your money to the bank and deposited it. You could then go to an ATM and withdraw your money:

```
var balance = 100

func withdraw(amount: Int) {
    balance -= amount
}
```

The ATM will never let you withdraw more than you put in, so it needs a way to let you know whether the transaction was successful. You can implement this as an enumeration with associated values:

```
enum WithdrawalResult {
    case Success(Int)
    case Error(String)
}
```

Each case has a required value to go along with it. For the success case, the associated `Int` will hold the new balance; for the error case, the associated `String` will have some kind of error message.

Then you can rewrite the `withdraw` function to utilize the enumeration cases:

```
func withdraw(amount: Int) -> WithdrawalResult {
    if (amount <= balance) {
        balance -= amount
        return .Success(balance)
    } else {
        return .Error("Not enough money!")
    }
}
```

Now you can perform a withdraw and handle the result:

```
let result = withdraw(99)

switch result {
case let .Success(newBalance):
    print("Your new balance is: \(newBalance)")
case let .Error(message):
    print(message)
}
```

You'll see the "Your new balance is:" message printed out in the debug console.

Each enumeration case can have a different set of associated values. You access associated values with pattern matching. In this example, you're binding them inside a `switch` statement. That way, each `switch` case can handle its own potentially unique set of associated values.

Many real-world contexts function by accessing associated values in an

enumeration. For example, servers often use enumerations to differentiate between types of requests:

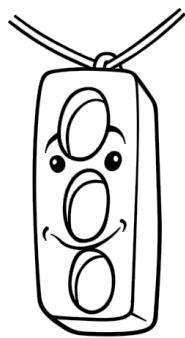
```
enum HTTPMethod {
    case GET
    case POST(String)
}
```

You'll also see enumerations used in error-handling. In the bank account example, there was just one generic error case with an associated string. Later in Chapter 21, "Error Handling," you'll see how to set up an enumeration with multiple cases to cover individual error conditions.

Enumeration as state machine

An enumeration is an example of a state machine, meaning it can only ever be one of the values it contains. The friendly traffic light illustrates this concept well:

```
enum TrafficLight {
    case Red, Yellow, Green
}
let trafficLight = TrafficLight.Red
```



A working traffic light will never be red and green simultaneously. You can observe this state machine behavior in many other modern devices that perform a predetermined sequence of actions that depend on a sequence of events. Good examples include:

- Vending machines that dispense soda when the customer deposits the proper amount of money;
- Elevators that drop riders off at upper floors before going down;
- Combination locks that require combination numbers in the proper order.

To operate as expected, these devices depend on an enumeration's guarantee that it will only ever be in one state at a time.

Mini-exercise

A household light switch is another example of a state machine. Create an enumeration for a light that can switch .on and .off.

Optionals

Since you've made it through the lesson on enumerations, the time has come to let you in on a little secret. There's a Swift language feature that has been using enumerations right under your nose all along: optionals! In this section, you'll explore their underlying mechanism.

As you know, optionals act like containers that have either something or nothing inside:

```
var age: Int?  
age = 17  
age = nil
```

Imagine your favorite social networking site profile. age as a form field that might be left blank. The variable is an optional that can hold an Int value.

You can declare an optional without the ? shorthand:

```
let email: Optional<String> = .None  
let website: Optional<String> = .Some("raywenderlich.com")
```

You saw the angle bracket syntax already when dealing with sets. In this case, the syntax describes the constant as an optional that can hold a String value.

You can see that optionals are really enumerations with two cases:

1. .None means there's no value.
2. .Some means there is a value, which is attached to the enumeration case as an associated value.

You extract the associated value out with a switch statement, as you've already seen:

```
switch website {  
case .None:  
    print("No value")  
case let .Some(value):  
    print("Got a value: \(value)")  
}
```

You'll see the "Got a value:" message printed out in the debug console.

Although optionals are really enumerations under the hood, Swift hides the

implementation details with things like `if-let` binding, the `?` and `!` operators, and keywords such as `nil`.

```
let optionalNil: Optional<Int> = .None  
  
optionalNil == nil    // true  
optionalNil == .None // true
```

If you try this in a playground, you'll see that both `nil` and `.None` are the same value.

Now that you know how optionals work, the next time you need a value container, you'll know just the right tool for the job.

In this chapter, you learned how to make enumerations dance. As you develop apps in Swift, you'll have many opportunities to use enumerations.

Key points

- An enumeration is a list of related values that define a common type.
- Enumerations provide an alternative to old-fashioned integer values.
- You can use enumerations to handle responses, store state and encapsulate values.

Where to go from here?

In this chapter, you learned how to make enumerations dance. Using enumerations makes your code much more readable than the alternative of integer values. Xcode makes this even easier by providing code completion and the Swift compiler guards against typos.

As you develop apps in Swift, you'll have many opportunities to use enumerations. Their inherent safety characteristics, such as strong typing and exhaustive case handler checking, will make your code more robust and less prone to errors.

Challenges

Challenge A: Adding raw values

Taking the coin example from earlier in the chapter, begin with an array of coins:

```
enum Coin: Int {  
    case Penny = 1
```

```
case Nickel = 5
case Dime = 10
case Quarter = 25
}

let coinPurse: [Coin] =
[.Dime, .Quarter, .Penny, .Penny, .Nickel, .Nickel]
```

Write a function where you can pass in the array of coins, add up the value and return the number of cents.

Challenge B: Computing with raw values

Taking the example from earlier in the chapter, begin with the Month enumeration:

```
enum Month: Int {
    case January = 1, February, March, April, May, June, July,
          August, September, October, November, December
}
```

Write a function to calculate the number of months until summer. Use the following signature:

```
monthsUntilSummer(month: Month) -> Int
```

Hint: You'll need to account for a negative value if summer has already passed in the current year. To do that, imagine looping back around for the next full year.

Challenge C: Pattern matching enumeration values

Taking the example from earlier in the chapter, begin with the Direction enumeration:

```
enum Direction {
    case North
    case South
    case East
    case West
}
```

Imagine starting a new level in a video game. The character makes a series of movements in the game. Calculate the position of the character on a top-down level map:

```
var movements: [Direction] = [.North, .North, .West, .South,
                           .West, .South, .South, .East, .East, .South, .East]
```

Hint: Use a tuple for the location:

```
var location = (x: 0, y: 0)
```

Chapter 17: Properties

By Ben Morrow

So far, this book has covered three kinds of "named types": structures, classes and enumerations. These make you a more efficient programmer by organizing attributes and behaviors that are common and reusable.

In the example below, the `Car` structure has two attributes, both constants that store `String` values:

```
struct Car {  
    let make: String  
    let color: String  
}
```

Attributes like these, and those of other named types, are called **properties**. The two attributes of `Car` are both **stored properties**, which means they store actual string values for each instance of `Car`. Only classes and structures can have stored properties.

Some properties calculate values rather than store them; that means there's no actual memory allocated for them, but they get calculated on-the-fly each time you access them. Naturally, these are called **computed properties**. Classes and structures can have computed properties, and so can enumerations.

In this chapter, you'll learn about both kinds of properties. You'll also learn some other neat tricks in dealing with properties, such as how to monitor changes in a property's value delay initialization of a stored property.

Stored properties

As you may have guessed from the example in the introduction, you're already familiar with many of the features of stored properties. To review, imagine you're building an address book. The common unit you'll need is a `Contact`:

```
struct Contact {  
    var fullName: String  
    var emailAddress: String  
}
```

You can use this structure over and over again, letting you build an array of contacts, each with a different value. The attributes you want to store are an individual's full name and email address.



These are the properties of the `Contact` structure. You provide a data type for each but opt not to assign a default value, because you plan to assign the value upon initialization. After all, the values will be different for each instance of `Contact`.

Remember that Swift automatically creates an initializer for you based on the properties you defined in your structure:

```
var person = Contact(fullName: "Grace Murray",  
                     emailAddress: "grace@navy.mil")
```

You can access the individual properties using dot notation:

```
let name = person.fullName // Grace Murray  
let email = person.emailAddress // grace@navy.mil
```

You can assign values to properties as long as they're defined as variables. When Grace married, she changed her last name:

```
person.fullName = "Grace Hopper"  
let grace = person.fullName // Grace Hopper
```

If you'd prefer to make it so that a value can't be changed, you can define a property as a constant instead:

```
struct Contact {  
    var fullName: String  
    let emailAddress: String  
}  
  
// Error: cannot assign a constant
```

```
person.emailAddress = "grace@gmail.com"
```

Once you've initialized an instance of this structure, you can't change `emailAddress`.

Default values

If you can make a reasonable assumption about what a value should be when the type is initialized, you can give the property a default value.

It doesn't make sense to create a default name or email address for a contact, but imagine there's a new property type to say what kind of contact it is:

```
enum Type {
    case Work, Family, Friend
}

struct Contact {
    var fullName: String
    let emailAddress: String
    var type: Type = .Friend
}
```

By including the assignment in the definition of `Contact`, you give it a default value. Any contact created from here on will automatically be a friend, unless you change the value to `.Work` or `.Family`.

The downside is that the automatic initializer doesn't notice default values, so you'll still need to provide a value for each property, unless you create your own custom initializer. You'll learn more about creating initializers in the next chapter.

Computed properties

Stored properties are certainly are the most common, but there are also properties that are computed, which simply means they perform a calculation before returning a value.

While a stored property can be a constant or a variable, a computed property must be defined as a variable. Computed properties must also include a type, because the compiler needs to know what to expect as a return value.

The measurement for a TV is the perfect use case for a computed property. The industry definition of the screen size of a TV isn't the screen's height or width, but its diagonal measurement:

```
struct TV {
    var height: Double
    var width: Double

    // 1
    var diagonal: Int {
```

```
// 2
let aSquared = pow(height, 2)
let bSquared = pow(width, 2)
let cSquared = aSquared + bSquared
// 3
let c = sqrt(cSquared)
// 4
let rounded = round(c)
// 5
return Int(rounded)
}
```

Let's go through this code one step at a time:

1. You use an `Int` type for your `diagonal` property. Although `height` and `width` are each a `Double`, TV sizes are usually advertised as nice, round numbers such as 50" rather than 49.52". Instead of the usual assignment operator `=` to assign a value as you would for a stored property, you use curly braces to enclose your computed property's calculation.
2. As you've seen before in this book, geometry can be handy; once you have the `width` and `height`, you can use the Pythagorean theorem to calculate the width of the diagonal: $a^2 + b^2 = c^2$. The Swift standard library thankfully has a math function to help you here: You can use `pow(_::_:)` to calculate the power of a number.
3. To solve for `c`, the diagonal value, you need the square root, which you can get from another standard library function, `sqrt(_:_:)`.
4. If you convert a `Double` directly to `Int`, it truncates the decimal so 109.99 will become just 109. That just won't do! Instead, you use `round(_:_:)` to round the value with the standard rules: If it the decimal is 0.5 or above, it rounds up; otherwise it rounds down.
5. Now that you've got a properly rounded number, you return it as an `Int`.

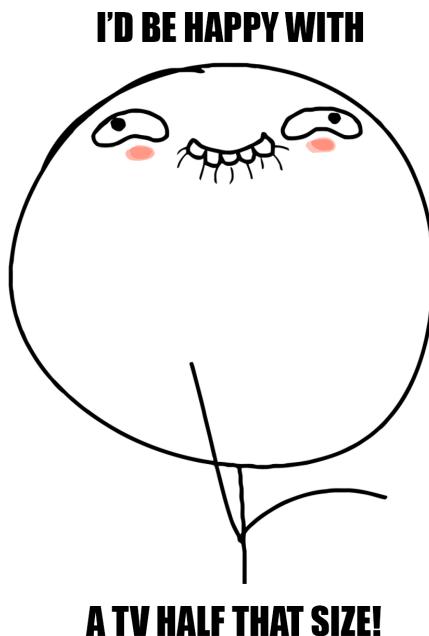
Computed properties don't store any values; they simply return values based on calculations. From outside of the structure, a computed property lets you retrieve a value just like you would from a stored property. Test this with the TV size calculation:

```
var tv = TV(height: 53.93, width: 95.87)
let size = tv.diagonal // 110
```

You have a 110-inch TV. Let's say you decide you don't like the standard movie aspect ratio and would instead prefer a square screen. You cut off some of the screen width to make it equivalent to the height:

```
tv.width = 53.93
let diagonal = tv.diagonal // 76
```

Now you've *only* got a 76-inch square screen. The computed property automatically provides the new value based on the new width.



Mini-exercise

Do you have a television or a computer monitor? Measure the height and width, plug it into a TV struct, and see if the diagonal measurement matches what you think it is.

Getter and setter

So far, you've been using the getter method of a computed property. You can also use a property's setter method, but it works differently than you might expect. Because the computed property has no place to store a value, the setter method sets the related *stored* properties indirectly:

```
var diagonal: Int {
    // 1
    get {
        // 2
        return Int(round(sqrt(height * height + width * width)))
    }
    set {
        // 3
        let ratioWidth: Double = 16
        let ratioHeight: Double = 9
        // 4
        height = Double(newValue) * ratioHeight /
            sqrt(ratioWidth * ratioWidth + ratioHeight * ratioHeight)
        width = height * ratioWidth / ratioHeight
    }
}
```

Here's what's happening in this code:

1. Because you want to include the setter, you now have to be explicit about which calculations comprise the getter and which the setter, so you surround each with curly braces.
2. You use the same Pythagorean theorem formula as before, simplified to one line.
3. For a setter, you usually have to make some kind of assumption. In this case, you provide a reasonable default value for the screen ratio.
4. The formulas to calculate a height and width, given a diagonal and a ratio, are a bit deep. You could work them out with a bit of time, but I've done the dirty work for you and provided them here. The important parts to focus on are:
 - The `newValue` keyword lets you use whatever was passed in during the assignment.
 - Remember, the diagonal is an `Int`, so to use it in a calculation with a `Double`, you must first transform it into a `Double`.
 - Once you've done the calculations, you assign the height and width properties of the `TV` structure.

Now, in addition to setting the height and width directly you can set them *indirectly* by changing the `diagonal` computed property. When you set this value, your setter will calculate and store the height and width.

Notice that there's no return statement in a setter method—it only modifies the other stored properties. With the setter in place, you have a nice little screen size calculator:

```
tv.diagonal = 70
let height = tv.height // 34.32...
let width = tv.width // 61.01...
```

Now you can discover the biggest TV that will fit in your cabinet or on your shelf. :]

Type properties

In the previous section, you learned how to associate stored and computed properties with instances of a particular type. The properties on your instance of `TV` are separate from the properties on my instance of `TV`.

However, the type *itself* may also need properties that are common across all instances. These properties are called **type properties**.

Imagine you're building a game with many levels. Each level has a few attributes,

or stored properties:

```
struct Level {  
    let id: Int  
    var boss: String  
    var unlocked: Bool  
}  
  
let level1 = Level(id: 1, boss: "Chameleon", unlocked: true)  
let level2 = Level(id: 2, boss: "Squid", unlocked: false)  
let level3 = Level(id: 3, boss: "Chupacabra", unlocked: false)  
let level4 = Level(id: 4, boss: "Yeti", unlocked: false)
```

You declare a type property using `static` for value-based types like structures. You can use a type property to store the player's progress as she unlocks each level:

```
struct Level {  
    static var highestLevel = 1  
    let id: Int  
    var boss: String  
    var unlocked: Bool  
}
```

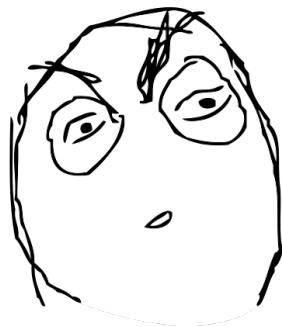
Here, `highestLevel` is a property on `Level` itself rather than on the instances. That means you don't access a type property on an instance:

```
// Error: you can't access a type property on an instance  
let highestLevel = level3.highestLevel
```

Instead, you access it on the type itself:

```
let highestLevel = Level.highestLevel // 1
```

WHEN WOULD I USE THIS?



This means you can retrieve the same stored property value from anywhere in the code for your app or algorithm. The player's progress is accessible from any level or any other place in the game, like the main menu.

Singleton pattern

You'll see type properties out in the world implementing the **singleton** pattern. A

singleton is a special case object where you have a defined class, but you only want a single instance of that class to exist at one time.

Note: In app development, you'll see singletons in use to represent things like the app itself or the current screen. The screen, for example, is modeled as a class. There's only one main screen at a time. The singleton pattern is a good fit to ensure you're always referring to the same instance.

Imagine you have a game, and you need to store some shared information such as the current score and the current state of the game. It makes sense to model this as a class, but you only need one instance of this class.

One way to produce a singleton in Swift is by using a constant type property, static let:

```
class GameManager {
    // 1
    static let defaultManager = GameManager()
    var gameScore = 0
    var saveState = 0
    // 2
    private init() {}
}
```

Here's what's happening:

1. A constant type property has the default value of a new instance of GameManager.
2. The private initializer must be a true singleton. A private initializer ensures that the class gets initialized only once, in the default value of the defaultManager. There's no way to initialize this class from outside of itself.

You must declare a singleton as a reference-type class rather than a value-type structure. If you were to implement one as a structure, then you wouldn't be able to change the values of gameScore and saveState later because the defaultManager property is a constant.

When you declare GameManager as a class, only its address in memory is a constant rather than the values themselves. You can change the values of gameScore and saveState since GameManager is a reference-type class.

The same principles you learned about type properties also apply to singletons:

- No matter where you are in the game, you can always access the GameManager.defaultManager to get the score and the save state. For example, maybe in the code for the enemy's AI, you check the player's score and adjust the difficulty accordingly.
- You can also set the score and save state from anywhere, and they'll be available everywhere they're used.

Singletons are akin to global variables. Using one is a powerful way to share data between different parts of your code without having to pass the data around manually.

Mini-exercise

Update the gameScore to 1024 and the saveState to 12 using the singleton instance. Then write code to access the gameScore and saveState values again. Remember that when accessing a type property, you use the type itself and not an instance.

Property observers

For your Level implementation, it would be useful to automatically set the highestLevel when the player unlocks a new one. For that, you'll need a way to listen to property changes. Thankfully, there are a couple of property observers that get called before and after property changes. You can use willSet {} and didSet {} similar to the way you used set {} and get {}:

```
struct Level {
    static var highestLevel = 1
    let id: Int
    var boss: String
    var unlocked: Bool {
        didSet {
            if unlocked && id > Level.highestLevel {
                Level.highestLevel = id
            }
        }
    }
}
```

Now, when the player unlocks a new level, it will update the highestLevel type property if the level is a new high. There are a couple of notable syntax quirks here:

- You *can* access the value of unlocked from inside the didSet {} implementation!
- Even though you're "inside" the type, you still have to access the type properties with their full names like Level.highestLevel rather than just highestLevel alone.
- unlocked is a stored property, not a computed property, because you're not providing a get {}.

Also, keep in mind that the willSet {} and didSet {} observers are *not* called when a property is set during initialization; they only get called when you assign a new value to a fully-initialized instance. That means property observers are only useful for variable properties, since constant properties are only set during initialization.

Limits a variable

You can also use property observers to limit the value of a variable. Say you had a light bulb that can't handle more than a maximum current flowing through its filament.

```
class LightBulb {  
    static let maxCurrent = 40  
    var currentCurrent = 0 {  
        didSet {  
            if currentCurrent > LightBulb.maxCurrent {  
                print("Current too high, falling back to previous setting.")  
                currentCurrent = oldValue  
            }  
        }  
    }  
}
```

In this example, if the current flowing into the bulb exceeds the maximum value, it will revert to its last successful value. Notice there's a helpful `oldValue` constant available in `didSet {}` so you can access the previous value.

Give it a try:

```
var light = LightBulb()  
light.currentCurrent = 50  
var current = light.currentCurrent // 0  
light.currentCurrent = 40  
current = light.currentCurrent // 40
```

You try to set the lightbulb to 50 amps, but the bulb rejected that input. Pretty cool!



Mini-exercise

In the lightbulb example, the bulb goes back to a successful setting if the current gets too high. In real life, that wouldn't work. The bulb would burn out! Rewrite the structure so that the bulb turns off before the current burns it out.

Hint: You can use newValue in the willSet observer.

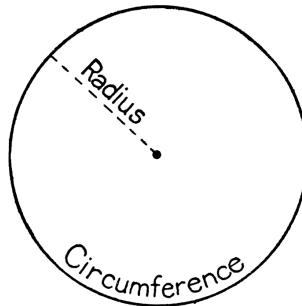
Lazy properties

If you have a property that might take some time to calculate, you don't want to slow things down until you actually need the property. Say hello to the **lazy stored property**! It could be useful for such things as downloading a user's profile picture or making a serious calculation.

Look at this example of a Circle class that uses pi in its circumference calculation:

```
class Circle {  
    lazy var pi = {  
        return ((4.0 * atan(1.0 / 5.0)) - atan(1.0 / 239.0)) * 4.0  
    }()  
    var radius: Double = 0  
    var circumference: Double {  
        return pi * radius * radius  
    }  
    init (radius: Double) {  
        self.radius = radius  
    }  
}
```

Here, you're not trusting the value of pi available to you from the standard library; you want to calculate it yourself!



You can create a new Circle with its initializer, and the pi calculation won't run yet:

```
let circle = Circle(radius: 5) // got a circle, pi has not been run
```

The value of pi waits patiently until you need it. Only when you ask for the circumference property is pi calculated and assigned a value:

```
let circumference = circle.circumference // 78.53  
// also, pi now has a value
```

Since you've got eagle eyes, you've noticed that pi uses a `{ }()` pattern to calculate its value, even though it's a stored property. The trailing parentheses execute the code inside the closure curly braces immediately. But since pi is

marked as `lazy`, the immediately executed closure won't *compute* immediately.

A lazy property is different from a computed property. In a lazy property, the closure gets calculated only the first time the property is accessed. On the other hand, `circumference` is a computed property and therefore gets calculated every time it's accessed. You expect the circumference's value to change if the radius changes.

`pi`, as a lazy stored property, only gets calculated the first time. That's great, because who wants to calculate the same thing over and over again?

The lazy property must be a variable, defined with `var`, instead of a constant defined with `let`. When you first initialize the class, the property effectively has no value. Then when some part of your code requests the property, the runtime changes the property's value. So even though the value only changes once, you still use `var`.

Note: Of course, you should absolutely trust the value of `pi` from the standard library. You can access it with the constant `M_PI`.

Key points

- **Properties** are variables and constants that comprise the attributes of a named type.
- **Stored properties** are available in structures and classes.
- **Computed properties** are available in structures, classes and enumerations. They are calculated each time your code requests them.
- The **static** keyword marks a **type property** that's universal to all instances of a particular type.
- The **lazy** keyword prevents a value of a stored property from being calculated until your code uses it for the first time. You'll want to use **lazy initialization** when a property's initial value is computationally intensive or when you won't know the initial value of a property until after you've initialized the object.
- `{ }()` is the pattern for a closure that is evaluated immediately, and you use it for lazy properties or to add computation to a stored property.

Where to go from here?

You saw the basics of properties while learning about structures, classes and enumerations, and now you've seen the more advanced features they have to offer.

As you continue on with app development, you'll find singletons and lazy properties in use throughout the various Apple frameworks.

Continuing with the advanced theme: you've already learned about methods, but in the next chapter, you learn even more about methods including initializers and advanced things to do with parameters.

Challenges

Challenge A: Computed properties

Given the Month enumeration you saw earlier in the chapter:

```
enum Month {
    case January, February, March, April, May, June, July, August,
    September, October, November, December

    func schoolSemester() -> String {
        switch self {
            case .August, .September, .October, .November, .December:
                return "Autumn"
            case .January, .February, .March, .April, .May:
                return "Spring"
            default:
                return "Not in the school year"
        }
    }
}
```

Rewrite schoolSemester() as a computed property instead of a method.

Challenge B: We all scream for ice cream

Rewrite the IceCream class to use default values and lazy initialization:

```
class IceCream {
    let name: String
    let ingredients: [String]

    init() {
        name = "Plain"
        ingredients = ["sugar", "milk"]
    }
}
```

1. Change the values in the initializer to default values for the properties.
2. Lazily initialize the ingredients array.

Chapter 18: Methods

By Ben Morrow

In the previous chapter, you learned about properties, which are common across structs, classes and enumerations. **Methods**, as you've already seen, are merely functions that reside inside a named type.

In this chapter, you'll take a closer look at methods and initializers. As with properties, the things you learn in this chapter will apply to methods across all types.

Method refresher

Remember `Array().removeLast()`? It pops the last item off an instance of an array:

```
var numbers = [1, 2, 3]
numbers.removeLast()
let newArray = numbers // [1, 2]
```

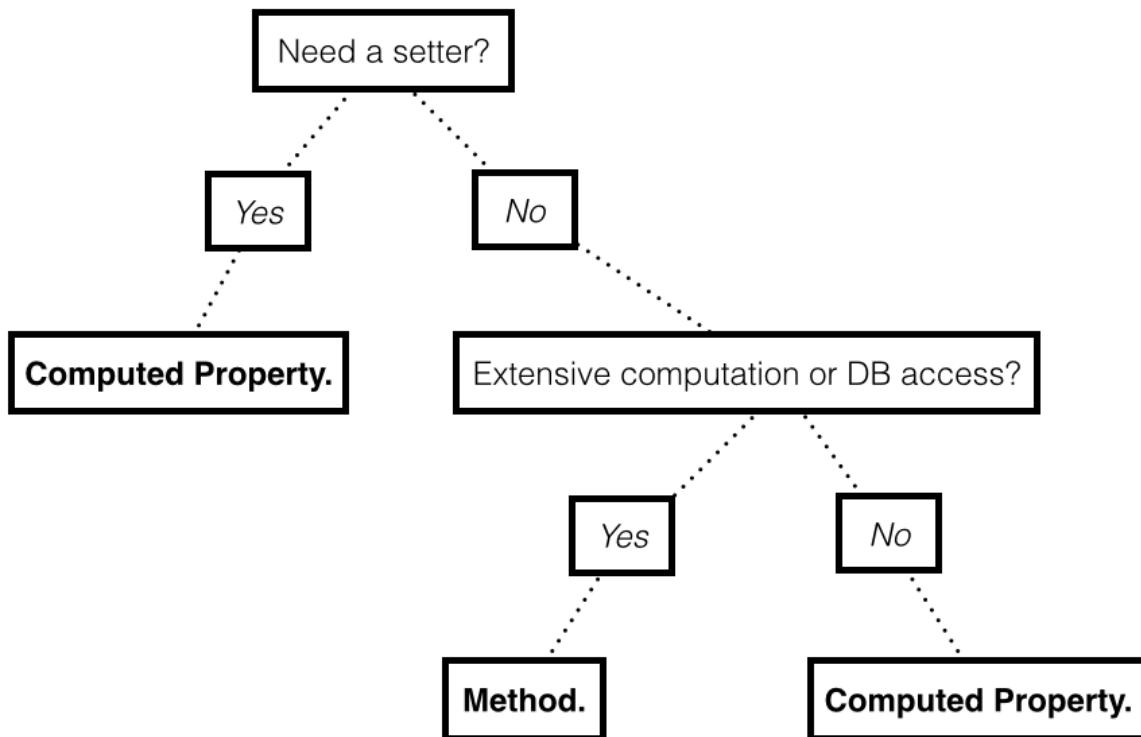


Methods like `removeLast()` help you control the data in the named type.

Comparing methods to computed properties

With a computed properties, you saw in the last chapter that you could run code from inside a named type. That sounds a lot like a method, so what is the difference? It is a matter of style, but there are a few helpful thoughts to help you decide. Properties hold values that you can get and set. Methods perform work. Sometimes this distinction gets fuzzy when a method's sole purpose is to return a single value.

Should I implement this value getter
as a method or as a computed property?



Ask yourself whether you want to be able to set a value as well as get the value. A computed property can have a setter component inside to write values. Another question to consider is whether the calculation requires extensive computation or reads from a database. Even for a simple value, a method helps you indicate to future developers that the call is expensive in time and computational resources. If the call is cheap, stick with a computed property.

Turning a function into a method

How could you convert this function into a method?

```
enum Month: Int {
    case January = 1, February, March, April, May, June,
        July, August, September, October, November, December
}

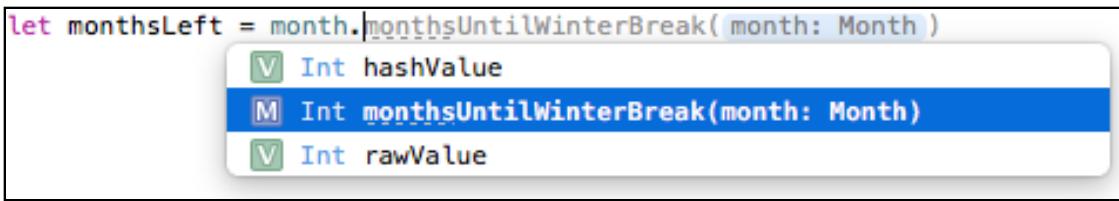
func monthsUntilWinterBreak(month: Month) -> Int {
    return Month.December.rawValue - month.rawValue
}
```

It's as easy as moving the function inside the named type definition:

```
enum Month: Int {
    case January = 1, February, March, April, May, June,
        July, August, September, October, November, December

    func monthsUntilWinterBreak(month: Month) -> Int {
        return Month.December.rawValue - month.rawValue
    }
}
```

There's no identifying keyword for a method; it really is just a function inside a named type. You call methods on an instance using dot syntax just as you do for properties. And just like properties, as soon as you start typing a method name, Xcode will provide suggestions, and you can autocomplete the call by pressing Tab.



```
let monthsLeft = month.monthsUntilWinterBreak(month: Month)
```



```
let month = Month.October
let monthsLeft = month.monthsUntilWinterBreak(month) // 2
```

If you think about this code for a minute, you'll realize that the method's definition is awkward. There must be a way to access the content stored by the instance instead of passing the instance itself as a parameter to the method. It would be so much nicer to call this:

```
let monthsLeft = month.monthsUntilWinterBreak() // Error!
```

Introducing self

A named type definition is like a blueprint, whereas an instance is a real object. To access the value of an instance, you use the keyword **self** inside the named type. The method definition transforms into this:

```
// 1
func monthsUntilWinterBreak() -> Int {
    // 2
    return Month.December.rawValue - self.rawValue
}
```

Here's what changed:

1. Now there's no parameter in the method definition.
2. In the implementation, `self` replaces the old parameter name.

You can now call the method without passing a parameter:

```
let monthsLeft = month.monthsUntilWinterBreak() // 2
```

Mini-exercise

Since `monthsUntilWinterBreak()` returns a single value and there's not much calculation involved, transform the method into a computed property with a getter component.

Introducing initializers

You learned about initializers in the previous chapters, but let's look at them again with your newfound knowledge of methods.

Initializers are special methods you can call to create a new instance. They omit the `func` keyword and even a name, instead using `init`. An initializer can have parameters, but it doesn't have to.

Right now, when you create a new instance of the `Month` enumeration, you have to specify its value:

```
let month = Month.January
```

You would like to create a new `Month` instance using the no-parameter initializer with a sane default value:

```
let month = Month() // Error!
```

You can provide the default by defining the initializer. By implementing `init`, you can create the simplest path to initialization:

```
enum Month: Int {
    case January = 1, February, March, April, May, June,
        July, August, September, October, November, December

    // 1
    init() {
        // 2
        self = .January
    }
}
```

Here's what's happening in this code:

1. The `init()` definition requires neither the `func` keyword nor a name.
2. In the initializer, you assign values for all the stored properties of a structure or class, or the enumeration value for an enum. To assign the value of an enumeration, you use the assignment operator on `self`.

Now you can use your simple initializer to create an instance:

```
let month = Month() // January  
let monthsLeft = month.monthsUntilWinterBreak() // 11
```

You can test a change to the value in the initializer:

```
init() {  
    self = .March  
}
```

The value of `monthsUntilWinterBreak()` will change accordingly:

```
let month = Month() // March  
let monthsLeft = month.monthsUntilWinterBreak() // 9
```



As you think about the implementation here, you might decide that the initializer should use a default value based on today's date. Implementing the initializer to set the value to today's date would provide a good user experience.

In the future, you'll be capable of retrieving the current date. Eventually you'll use the `NSDate` class to work with dates. Before you get carried away with all the power that `NSDate` provides, let's think more about how you would implement your own Date type.

Initializers in structures and classes

Structures and classes (but not enumerations) have stored properties. As you know, these stored properties can have default values:

```
struct Date {  
    var month = Month.January  
    var day = 1  
}
```

You can follow a similar pattern as the implementation of `Month` and instead assign

the default values inside the initializer:

```
struct Date {  
    var month: Month  
    var day: Int  
  
    init() {  
        month = .January  
        day = 1  
    }  
}
```

Structure types automatically offer a **memberwise initializer** if they don't define any of their own custom initializers. Recall that the auto-generated memberwise initializer accepts all the properties in order as parameters, like: `init(month:day:)`, for the `Date` structure. When you write even a single initializer, as you've done here, the compiler scraps the automatically created one. So this code no longer works:

```
let date = Date(month: .February, day: 14) // Error!
```

You'll have to define your own initializer with parameters:

```
init(month: Month, day: Int) {  
    self.month = month  
    self.day = day  
}
```

In this code, you assign the parameters to the properties of the structure. Notice that you use the `self.` syntax to tell the compiler that you're referring to the property rather than the local parameter. The compiler assumes you mean the most local one.

In the simple initializer, you didn't have to use `self.`:

```
init() {  
    month = .January  
    day = 1  
}
```

In this code, there aren't parameters with the same names as the properties. Since the properties are the most locally defined names, you don't need to prefix them with `self.` for the compiler to understand you're referring to properties.

With the complex initializer in place, you can call the new initializer the same way you used to call the automatically generated initializer:

```
let date = Date(month: .February, day: 14)  
let dateMonth = date.month // February  
let dateDay = date.day // 14
```

Introducing mutating methods

Structures have a limitation: methods cannot change the values of the instance without being marked as mutating. You can imagine a method in the Date structure that advances to the next day:

```
mutating func advance() {  
    day++  
}
```

The `mutating` keyword marks a method that changes a structure's value. Since a structure is a value type, each time it's passed around an app, the system copies it. If a method changes the value of one of the properties, then the original instance and the copied instance are no longer equivalent.

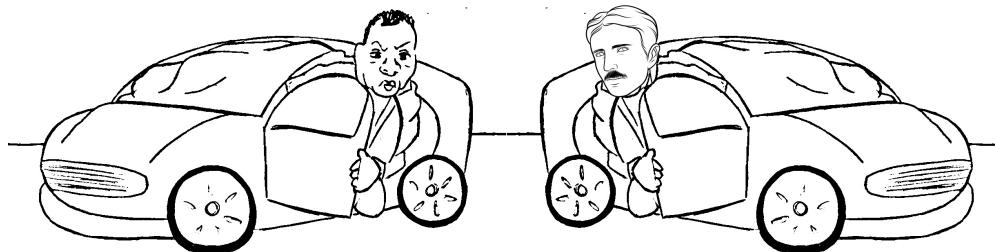
By marking a method as `mutating`, you're saying, "I know this changes the value of the property so that the instance is different from its other copies, and I'm OK with that."

The implementation above is a naive way of writing `advance()` because it doesn't account for what happens at the end of a month. In a challenge at the end of this chapter, you'll tackle the robust logic. For now, let's investigate the difference in mutating methods on structures and classes.

Mutating methods in a class

Structures require you to mark methods as `mutating` when you change the instance. What about classes? It makes sense for a date to be a value type, because two dates are always the same as long as they have the same value. "March 20" in one place is the same value as "March 20" in another.

Cars, on the other hand, are not the same even if they have the same value. A "red Tesla" in one place is not the same as a "red Tesla" in another. They are two different automobiles with two different histories, even if that difference isn't reflected in the values of their attributes. They are distinct objects.



A reference-type class is a better fit for a car than a value-type structure. Methods that alter the properties of a class instance are *not* marked as mutating:

```
class Car {  
    // 1  
    let make: String  
    // 2  
    private(set) var color: String  
    init() {  
        make = "Ford"  
        color = "Black"  
    }  
    required init(make: String, color: String) {  
        self.make = make  
        self.color = color  
    }  
    // 3  
    func paint(color: String) {  
        self.color = color  
    }  
}
```

Let's go through this code step by step:

1. You declare `make` as a constant with `let` because the manufacturer of a car cannot change after production. Notice that even though `make` is a constant, you later set the value in initializers. You can't change the value after initializing the instance, though. If you try to change a constant property in a method, you'll get an error.
2. You can change `color`, but in reality, but you might want to check the new color or perform some other logic when it's repainted. You declare `color` as `private(set)` so that it can't be altered from outside of the class.
3. Instead of direct property access, you define `paint(_:_)` to assign a new value to the `color`. You don't mark this method as `mutating` even though a property changes after initialization. `Car` is a reference-type class, not a value-type structure.

Now you can paint the car a new color:

```
let car = Car(make: "Tesla", color: "Red")  
car.paint("Blue")
```

Type methods

Like type properties, you can use **type methods** to access data across all instances. You call it on the type itself, instead of on an instance. To define a type method, you prefix it with the `static` modifier.

Type methods are useful for things that are *about* a type in general rather than

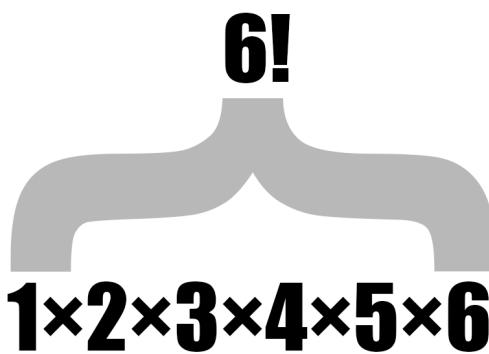
something about specific instances; for example, you could use type methods to group similar methods into a class or struct:

```
struct Utils {  
    // 1  
    static func factorial(number: Int) -> Int {  
        // 2  
        return (1...number).reduce(1, combine: *)  
    }  
    // 3  
    let factorial = Utils.factorial(6) // 720
```

You might have custom calculations for things such as factorial. In this example, you would group all of them together into this `Utils` class.

Here's what's happening:

1. You use `static` to declare the type method, which accepts an integer and returns an integer.
2. The implementation uses a higher-order function called `reduce(_:_:combine:)`. It effectively follows the formula for calculating a factorial: "The product of all the whole numbers from 1 to n". You could write this using a `for` loop, but the higher-order function expresses your intent in a cleaner way.
3. You call the type method on `Utils`, rather than on an instance of the type.



Utility type methods are helpful for code completion. In this example, you can see all the utility methods available to you by typing "`Utils.`"

```
let factorial = Utils.factorial(number: Int)  
    M  Int factorial(number: Int)  
    M  Utils init()
```

Mini-exercise

Add a type method to the `Utils` structure that calculates the nth triangle number. It will be very similar to the factorial formula, except instead of multiplying the numbers, you add them.

Key points

- Properties are the attributes of a named type and **methods** are behaviors that extend its functionality.
- A method is a function defined inside of a named type.
- A method can access the value of an instance by using the keyword `self`.
- Initializers** are methods that aid in the creation of a new instance.
- A **type method** can access data across all instances of a type. To define a type method, you prefix it with the `static` modifier.

Where to go from here?

Methods and properties are the things that make up your types. Learning about them as you have these two chapters is important since you'll find them in all the named types—structs, classes and enumerations.

But wait! There's one more type that's different from the others although it also has properties and methods, in a sense. In the next chapter, you'll learn about **protocols**, which you'll see are useful types even though you don't instantiate them directly!

Challenges

Challenge A: Growing a circle

Given the `Circle` structure below:

```
class Circle {  
    var radius: Double = 0  
    var area: Double {  
        return M_PI * radius * radius  
    }  
  
    init (radius: Double) {  
        self.radius = radius  
    }  
}
```

```
}
```

Write a method that can change an instance's area by a growth factor. For example if you call `circle.growByAFactor(3)`, the area of the instance will triple.

Hint: Make use of the setter for `area`.

Challenge B: Date calculations

Below is a naive way of writing `advance()` for the `Date` structure you saw earlier in the chapter:

```
enum Month: Int {
    case January = 1, February, March, April, May, June,
    July, August, September, October, November, December
}

struct Date {
    var month: Month
    var day: Int
    init(month: Month, day: Int) {
        self.month = month
        self.day = day
    }
    mutating func advance() {
        day++
    }
}

var current = Date(month: .December, day: 31)
current.advance()
let currentMonth = current.month // December; should be January!
let currentDay = current.day // 32; should be 1!
```

What happens when the function should go from the end of one month to the start of the next? Rewrite `advance()` to account for advancing from December 31st to January 1st.

Challenge C: Modeling a car

The `Car` class you saw earlier in the chapter needs an attribute to describe movement:

```
class Car {
    let make: String
    private(set) var color: String
    init() {
        make = "Ford"
        color = "Black"
    }
    required init(make: String, color: String) {
        self.make = make
        self.color = color
    }
}
```

```
func paint(color c: String) {  
    self.color = c  
}
```

1. Add a property called speed.
2. Ensure that speed can only be assigned internally from the class. W
3. Write two methods: accelerate(), which sets the speed to 20, and applyBrakes(), which sets the speed to 0.

Chapter 19: Protocols

By Erik Kerber

In this book, you've learned about the core types the Swift language provides, including the user-defined named types. There is one final type that can bridge common behaviors between structs, classes and enums. In fact, it is itself a user-defined named type: the **protocol**.

However, protocols don't define a type that you can instantiate directly. Instead, they define an interface or a template for an actual concrete type such as a struct or class or enumeration. With a protocol, you can define a common set of behaviors and then define the actual types that implement them.

In this chapter, you'll learn about protocols and see why they're useful when programming in Swift.

Introducing protocols

You define a protocol much as you do any other named type:

```
protocol Vehicle {  
    func accelerate()  
    func stop()  
}
```

The keyword `protocol` is followed by the name of the protocol, followed by the curly braces with the members of the protocol inside. The big difference you'll notice is that the protocol *does not contain any implementation*.

That means you can't instantiate a `Vehicle` directly:

```
8  
9 let vehicle = Vehicle()  
10 // ! 'Vehicle' cannot be constructed because it has no accessible initializers  
11
```

Instead, you use protocols to enforce methods and properties on *other* types. What you've defined here is like the *idea* of a vehicle—it's something that can accelerate and stop.

Protocol syntax

A protocol can be **adopted** by a class, struct, or enum—and when another type adopts a protocol, it's required to implement the methods and properties defined in the protocol. Once a type implements all members of a protocol, the type is said to **conform** to the protocol.

Here's how you declare protocol conformance for your type:

```
class Unicycle: Vehicle {  
}
```

You follow the name of the named type with a semicolon and the name of the protocol you want to conform to. This syntax might look familiar, since it's the same syntax that declares that a class inherits from another class. In this example, `Unicycle` is conforming to the `Vehicle` protocol.

Note that it *looks* like inheritance but it isn't—structs and enumerations can also conform to protocols with this syntax.

Swift will display an error for this class though, since you haven't implemented the two protocol methods. Remember, your declaration claims that `Unicycle` conforms to the `Vehicle` protocol, but it doesn't—the required `accelerate()` and `stop()` methods are missing.

We'll come back to implementing protocols in a little bit, after continuing the tour of what you can do inside the protocol definition itself.

Methods in protocols

In the `Vehicle` protocol above, you define a pair of methods, `accelerate()` and `stop()`, that all types conforming to `Vehicle` need to implement.

You define methods on protocols much like you would on any class, struct or enum:

```
enum Direction {  
    case Left  
    case Right  
}
```

```
protocol Vehicle {  
    func accelerate()  
    func stop()  
    func turn(direction: Direction)  
    func description() -> String  
}
```

There are a few differences to note. You don't, and in fact cannot, define any *implementation* for the methods. This is to help you enforce a strict separation of interface and code, as the protocol by itself makes no assumption about the implementation details of any type you mark as conforming to the protocol.

Also, methods defined in protocols can't contain default parameters:

```
protocol Vehicle {  
    // Build error!  
    func turn(direction: Direction = .Left)  
}
```

If you want to provide direction as an optional argument, the approach you would take is to define both versions of the method explicitly:

```
protocol Vehicle {  
    func turn(direction: Direction)  
    func turn()  
}
```

You would then define both versions of turn() in any conforming type.

Also, you also can't set any access control—such as private, internal or public—in a protocol's methods. Because protocols define the interface, Swift assumes that all members of the protocol have the same access scope as the protocol itself. In other words, if code can "see" the protocol, it can see all members of the protocol.

Properties in protocols

You can also define properties in a protocol:

```
protocol Vehicle {  
    var weight: Int { get }  
    var name: String { get set }  
}
```

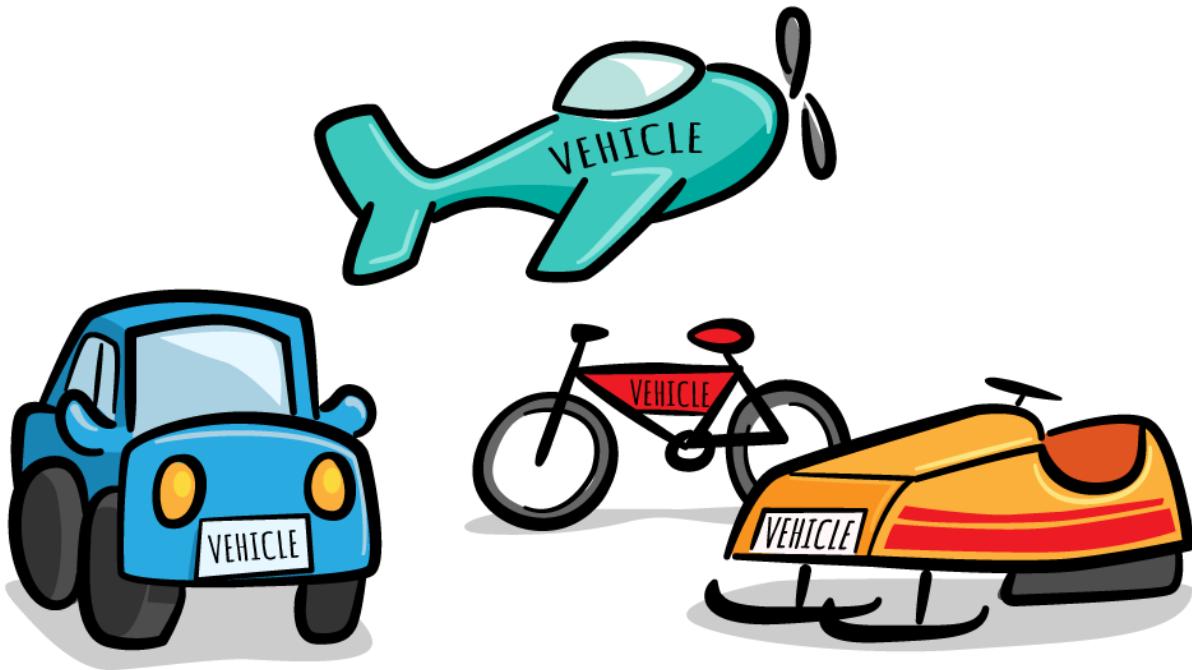
When defining properties in a protocol, you must explicitly mark them as get and/or set, somewhat similar to the way you declare computed properties. However, much like methods, you don't include any *implementation* for properties. Are you beginning to see a pattern?

The fact that you must mark get and set on properties shows that a protocol doesn't know about a property's implementation, which means it makes no assumption about the property's *storage*. You can then implement these properties

as computed properties *or* as regular variables. All the protocol requires is that the property has a setter and/or getter as declared.

Protocol inheritance

The Vehicle protocol contains a set of methods that could apply to any type of vehicle—a bike, car, snowmobile or even an airplane!



You may wish to define a protocol that contains all the qualities of a Vehicle, but that is also specific to those vehicles that have wheels. For this, you can have protocols that inherit from other protocols, much like you can have classes that inherit from other classes:

```
protocol WheeledVehicle: Vehicle {  
    var numberofWheels: Int { get }  
    var wheelSize: Double { get set }  
}
```

Now any type you mark as conforming to the WheeledVehicle protocol will have all the members defined within the braces, in addition to all of the characteristics of Vehicle. As with subclassing, any type you mark as a WheeledVehicle will have an *is-a* relationship with the protocol Vehicle.

```
let bike: WheeledVehicle // any WheeledVehicle
```

In this example, bike **is** a wheeled vehicle and it **is** as vehicle.

Implementing protocols

Now that you know how to define a protocol, let's see more details on how you can implement them.

As you've already seen, when you declare your type as conforming to a protocol, you also have to define *all* the methods declared in the protocol:

```
protocol Vehicle {
    func accelerate()
    func stop()
}

class Bike: Vehicle {
    var peddling: Bool = false
    var brakesApplied: Bool = false

    func accelerate() {
        peddling = true
        brakesApplied = false
    }

    func stop() {
        peddling = false
        brakesApplied = true
    }
}
```

Now the class `Bike` implements all the methods defined in `Vehicle`, and the class definition no longer results in a build error.

As you can see, by defining a protocol, you guarantee that any type that conforms to the protocol will have *all* the members you've defined in the protocol.

Implementing properties

Recall that you declare properties in protocols with a `get` and/or a `set`:

```
protocol WheeledVehicle: Vehicle {
    var numberOfWorks: Int { get }
    var wheelSize: Double { get set }
}
```

When implementing properties defined in protocols, you can use the same computed properties to fulfill the protocol's requirement:

```
class Bike: WheeledVehicle {
    var numberOfWorks: Int {
        return 2
    }

    private var myWheelSize: Double = 16.0
    var wheelSize: Double {

```

```
    get {
        return myWheelSize
    }
    set {
        myWheelSize = newValue
    }
}
```

In the code above, you define the `numberOfWheels` property with an implicit `get`, and the `wheelSize` property using both a `get` and a `set` with an internal backing property called `myWheelSize`.

Fortunately, Swift doesn't require you to use computed properties to implement protocols. As long as you define a property that can be read if it is defined with a `get`, and assigned if it's defined with a `set`, then that property will satisfy the protocol:

```
class Bike: WheeledVehicle {
    let numberOfWheels = 2
    var wheelSize = 16.0
}
```

That's amazingly more concise, isn't it? In this example, you've satisfied `numberOfWheels` with a constant `let` because the value can be read, and you've satisfied `wheelSize` with a `var` because it can be both read and assigned. Likewise, if you were to define `wheelSize` with a `let`, the protocol wouldn't be satisfied because the constant cannot be assigned!

It's important to note that just because a protocol defines a property as read-only, it doesn't mean you necessarily need to implement it with a read-only property:

```
class Bike: WheeledVehicle {
    var numberOfWheels = 2
    // ...
}
```

Now you define `numberOfWheels` as a `var`, effectively making it read/write. If the type system sees the type as `Bike`, you can edit the `numberOfWheels` to whatever `Int` you like. From the perspective of the `WheeledVehicle` protocol, however, the property is still read-only:

```
var bike: Bike = Bike()
bike.numberOfWheels = 16 // 16-wheeler!

var wheeledBike: WheeledVehicle = bike
// Cannot assign to numberOfWheels!
//wheeledBike.numberOfWheels = 4
```

Typealias in protocols

One additional member you can add to a protocol is a `typealias`. When using

typealias in a protocol, you are simply declaring that there *is* a typealias and are deferring to the protocol adopter the exact type it is referring to.

This allows you to use arbitrarily named types without specifying exactly which type it will eventually be:

```
protocol WeightCalculatable {
    typealias WeightType

    func calculateWeight() -> WeightType
}
```

This delegates the eventual type that calculateWeight() will return until the adopter declares, or "implements" you might say, the typealias in the protocol.

You can see how this works using two examples:

```
class HeavyThing: WeightCalculatable {
    // This heavy thing only needs integer accuracy
    typealias WeightType = Int

    func calculateWeight() -> Int {
        return 100
    }
}

class LightThing: WeightCalculatable {
    // This light thing needs decimal places
    typealias WeightType = Double

    func calculateWeight() -> Double {
        return 0.0025
    }
}
```

Because the typealias for HeavyThing declares an Int, the compiler enforces that calculateWeight() should also return an Int. The same is then enforced on LightThing which declares a Double.

You may have noticed that this changes the contract of WeightCalculatable depending on how the typealias is declared in the adopting type. It is important to note that this prevents you from using the protocol as a simple variable, because the compiler doesn't know ahead of time what WeightType will be:

```
// Build error!
// protocol 'WeightCalculatable' can only be used as a generic
// constraint because it has Self or associated type requirements.
let b: WeightCalculatable = LightThing()
```

Fear not, however! You will find associated types such as when you dive into **generic constraints** in both Chapter 20, "Protocol-Oriented Programming" and Chapter 22, "Generics."

Implementing multiple protocols

Recall that a class can only inherit from a single class—a concept known as "single inheritance". In contrast, a class can inherit from as many protocols as you'd like!

Suppose instead of creating a `WheeledVehicle` protocol that inherits from `Vehicle`, you instead made `Wheeled` its own protocol:

```
protocol Wheeled {
    var numberofWheels: Int { get }
}

class Bike: Vehicle, Wheeled {
    // Implement both Vehicle and Wheeled
}
```

Protocols support "multiple conformance", so you can apply any number of protocols to types you define. In the example above, the `Bike` class is now required to implement all members defined in both `Vehicle` and `Wheeled`.

Maybe you've noticed that while you've successfully decoupled `Wheeled` types and `Vehicle` types, you no longer have one single type that represents a "wheeled vehicle". Fortunately, you can "have your cake and eat it too!" using multiple inheritance with protocols:

```
protocol WheeledVehicle: Vehicle, Wheeled {
    // WheeledVehicle is a Wheeled, and a Vehicle!
}

class Bike: WheeledVehicle {
    // Implement both Vehicle and Wheeled
}

let bike5: Bike = Bike()
// is-a Wheeled
let wheeledBike: Wheeled = bike
// is-a Vehicle
let vehicleBike: Vehicle = bike
// is-a WheeledVehicle
let wheeledVehicleBike: WheeledVehicle5 = bike
```

Now the `Bike` class "is a" `Wheeled`, a `Vehicle` and a `WheeledVehicle`!

Extensions and protocol conformance

You can adopt protocols using extensions as well as on the original type declaration. This allows you to add protocols to types you don't necessarily own. Consider the simple example below which adds a custom protocol to `String`:

```
protocol WhatType {
    var typeName: String { get }
}
```

```
extension String: WhatType {
    var typeName: String {
        return "I'm a String"
    }
}

let myType: TypeName = "Swift by Tutorials!"
myType.typeName // I'm a String
```

Even though `String` is part of the standard library and not accessible to your code, you are able make `String` both adopt and conform to the `TypeName` protocol!

Adding many protocols to your type definition also clutters up the definition. By using extensions, you can nicely group together the protocol adoption with the methods and properties needed for conformance.

The following code breaks out the adoption of `Wheeled` and `Vehicle` into extensions on `Bike`:

```
class Bike { }

extension Bike: Wheeled {
    var numberOfWorkers: Int {
        return 2
    }
}

extension Bike: Vehicle {
    func accelerate() {
        // Accelerate
    }

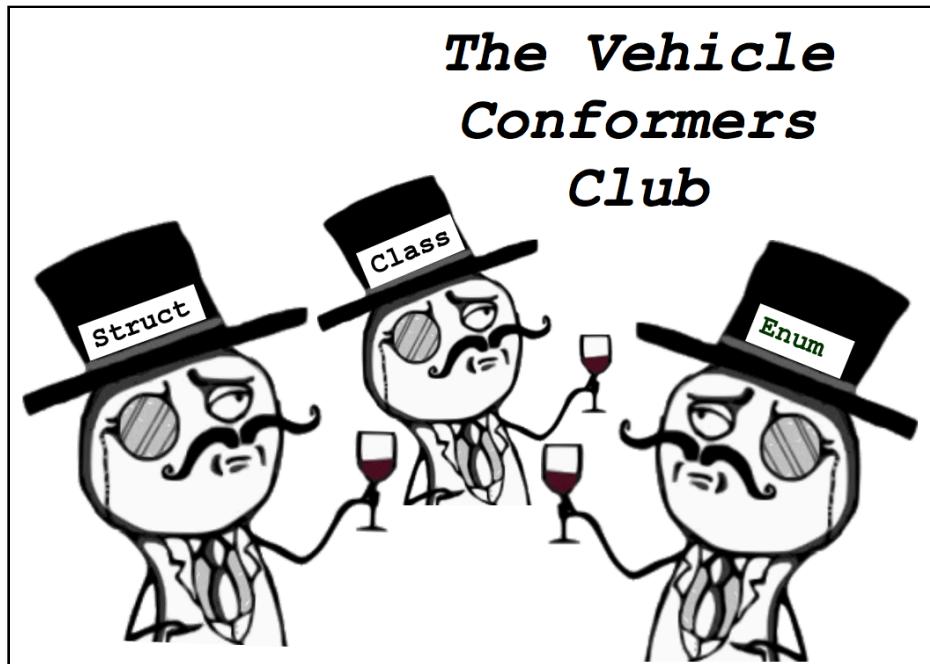
    func stop() {
        // Stop
    }
}
```

The extensions now pair `numberOfWorkers` with `Wheeled`, and also pairs `accelerate` and `stop` with `Vehicle`. If you were to remove a protocol from `Bike`, you could simply delete the extension that adopts that protocol entirely.

An important caveat of this is that you cannot declare *stored* properties in extensions, only computed properties such as `numberOfWorkers` above. Fortunately, you can still declare stored properties in the original type declaration and satisfy protocol conformance to any protocol adopted in an extension.

Classes, structs and enums

One of the more powerful features protocols provide your code is their agnosticism to what type implements them. You can apply the same protocol to a class, a struct, or even an enum!



```
protocol Wheeled {
    var numberOfWorkers: Int { get }
}

class ClassyBike: Wheeled {
    let numberOfWorkers = 2
}

struct StructyBike: Wheeled {
    let numberOfWorkers = 2
}

enum EnummmyBike: Wheeled {
    case Mountain
    case Road
    case Tricycle

    var numberOfWorkers: Int {
        switch self {
            case Mountain, Road:
                return 2
            case Tricycle:
                return 3
        }
    }
}
```

Because the class, struct and enum above all implement Wheeled, you can assign all three to a variable of type Wheeled:

```
// Class
var wheels: Wheeled = ClassyBike()
wheels.numberOfWorkers // 2
```

```
// Struct  
wheels = StructyBike()  
wheels.numberOfWheels // 2  
  
// Enum  
wheels = EnummyBike.Tricycle  
wheels.numberOfWheels // 3!
```

You declare the `wheels` variable with the type `Wheeled`, so the compiler doesn't care about what kind of type implements it, as long as it conforms to the `Wheeled` protocol.

Protocols in action

As you've seen thus far, a protocol is a way for you to define required interfaces on types without specifying implementation. What may or may not be immediately apparent to you is when to use a protocol or what your motivations might be.

Recall the `Person`, `Student`, `StudentAthlete` example from previous chapters:

```
class Person {  
    // First and last name  
}  
  
class Student: Person {  
    // Grades, student ID  
}  
  
class StudentAthlete: Student {  
    // Eligible, sports  
}
```

With three classes, this is a reasonably deep class hierarchy. `StudentAthlete` is very coupled to the `Student` and `Person` classes.

Suppose you want to keep a roster of all members of a team. You could define an array of `StudentAthlete` objects:

```
var roster: [StudentAthlete]
```

Simple enough, except what would happen if you had another class, `Coach`, and wanted the roster to include both the players *and* the coaches?

```
class Teacher: Person { }  
  
class Coach: Teacher {  
    let role: String = "Coach"  
  
    func play() {  
        print("Coach the game!")  
    }  
}
```

```
}
```

The Coach class has a similarly deep class hierarchy, of which the only common type is Person. You could make the roster an array of Person types, but that would only give you access to their first and last names:

```
var roster: [Person] = [
    Coach(firstName: "Steve", lastName: "Cook"),
    StudentAthlete(firstName: "Jane", lastName: "Appleseed"),
    StudentAthlete(firstName: "Johnny", lastName: "Appleseed")
]

func printRoster(roster: [Person]) {
    for member in roster {
        print("\(member.firstName): \(member.role)")
        // Build error, Person does not have role!
    }
}
```

Because the only way both Coach and StudentAthlete can go into a collection is to use the "lowest common denominator" type, you wouldn't be able to use any properties or methods except those found in the Person type.

In many cases, there may not be *any* common base class between two types. Especially if those types are structs or enums!

Thankfully, a protocol is the perfect solution to provide type parity between Coach and StudentAthlete:

```
protocol TeamMember {
    var role: String { get }
    var firstName: String { get }
    func play()
}
```

Now, you can apply the TeamMember protocol to both Coach and StudentAthlete:

```
class Coach: Teacher, TeamMember {
    let role: String = "Coach"

    func play() {
        print("Coach the game!")
    }
}

class StudentAthlete: Student, TeamMember {
    let role: String = "Player"

    func play() {
        print("Play the game")
    }
}
```

Since both types now share a common (sports related!) protocol, you can add them

to your collection of team members:

```
let roster: [TeamMember] = [
    Coach(firstName: "Steve", lastName: "Cook"),
    StudentAthlete(firstName: "Jane", lastName: "Appleseed"),
    StudentAthlete(firstName: "Johnny", lastName: "Appleseed")
]

func printRoster(roster: [TeamMember]) {
    for member in roster {
        print("\(member.firstName): \(member.role)")
    }
}

printRoster(roster)
// "Steve: Coach"
// "Jane: Player"
// "Johnny: Player"
```

Mini-exercise

Completely remove the class hierarchy from the Coach and StudentAthlete example. In other words, use Person, Student and TeamMember protocols. What are the advantages of this approach? Disadvantages?

Protocols in the standard library

The Swift standard library uses protocols extensively, in ways that may surprise you. Understanding the roles protocols play in Swift will not only help you learn how to do things the "Swifty" way—it might also teach you new ways to write clean, decoupled code.

Equatable and comparable

Some of the simplest Swift code you can write involves comparing two integers with the `==` operator:

```
let a = 5
let b = 5

a == b // true
```

You can, of course, do the same thing with strings:

```
let swiftA = "Swift"
let swiftB = "Swift"

swiftA == swiftB // true
```

You cannot, however, use the `==` operator on just *any* type. Suppose you wrote a struct that represents a team's record and wanted to determine if two records were

equal:

```
struct Record {  
    var wins: Int  
    var losses: Int  
}  
  
let recordA = Record(wins: 10, losses: 5)  
let recordB = Record(wins: 10, losses: 5)  
  
recordA == recordB // Build error!
```

You can't apply the `==` operator to the struct you just defined, but why?

Recall that `Int` and `String` are structs just like `Record`, and their use of the equality operator isn't simply "magic" reserved for standard Swift types—you can, in fact, extend it to your own code!

Both `Int` and `String` conform to the `Equatable` protocol, which is a protocol in the standard library that defines a single function:

```
protocol Equatable {  
    func ==(lhs: Self, rhs: Self) -> Bool  
}
```

You can apply this protocol to the `TeamMember` struct:

```
extension Record: Equatable {}  
  
func ==(lhs: Record, rhs: Record) -> Bool {  
    return lhs.wins == rhs.wins &&  
        lhs.losses == rhs.losses  
}
```

Here, you've *extended* the `Record` type. With extensions, you can add additional methods or properties to an existing type. In this case, you're adding protocol conformance to `Record`.

Note: You'll learn more about extensions very soon in Chapter 20, "Protocol-Oriented Programming."

Here, you're defining the `==` operator for comparing two `Record` instances. In this case, two records are equal if they have the same number of wins and losses.

Now, you're able to use the `==` operator to compare two `Record` types, just like you can with `String` or `Int`:

```
let recordA = Record(wins: 10, losses: 5)  
let recordB = Record(wins: 10, losses: 5)  
  
recordA == recordB // True
```

Similar protocols exist for other operators, such as `<`, `<=`, `>=` and `>`. These operators are defined in the `Comparable` protocol, which inherits from `Equatable`:

```
protocol Comparable : Equatable {
    func <(lhs: Self, rhs: Self) -> Bool
    func <=(lhs: Self, rhs: Self) -> Bool
    func >=(lhs: Self, rhs: Self) -> Bool
    func >(lhs: Self, rhs: Self) -> Bool
}
```

By adding the `Comparable` protocol to `Record`, you can not only check equality between records, you can determine if one record has more wins than the other:

```
extension Record: Comparable {}

func <(lhs: Record, rhs: Record) -> Bool {
    let lhsPercent = Double(lhs.wins) / (Double(lhs.wins) +
    Double(lhs.losses))
    let rhsPercent = Double(rhs.wins) / (Double(rhs.wins) +
    Double(rhs.losses))

    return lhsPercent < rhsPercent
}

let team1 = Record(wins: 23, losses: 8)
let team2 = Record(wins: 23, losses: 8)
let team3 = Record(wins: 14, losses: 11)

team1 < team2 // false
team1 > team3 // true
```

Your implementation of `<` compares the overall win percentages of the two records. If record A has a lower win percentage than record B, then that's considered "less than."

Note: You only defined `<` but what about `>`? Similarly, you defined `==` but what about `!=` for inequality? Swift gives you a helping hand here and automatically generates those other comparison operators. If you read the documentation for `Equatable` and `Comparable`, you'll see it tells you that only `==` and `<` are needed.

"Free" functions

While `==` and `<` are useful in their own right, the Swift library provides you with many "free" functions and methods for types that conform to `Equatable` and `Comparable`.



For any collection you define, such as an `Array`, that contains a `Comparable` type, you have access to methods such as `sort()` that are part of the standard library:

```
let leagueRecords = [team1, team2, team3]

leagueRecords.sort()
// {wins 14, losses 11}
// {wins 23, losses 8}
// {wins 23, losses 8}
```

Since you've given `Record` the ability to compare two elements, the standard library has all the information it needs to sort each element in an array!

As you can see, implementing `Comparable` and `Equatable` gives you quite an arsenal of tools:

```
leagueRecords.maxElement() // {wins 23, losses 8}
leagueRecords.minElement() // {wins 23, losses 8}
leagueRecords.startsWith([team1, team2]) // true
leagueRecords.contains(team1) // true
```

Other useful protocols

While learning *all* of the Swift standard library isn't vital to your success as a Swift developer, there are a few other important protocols that you'll find useful in almost any project.

Hashable

The `Hashable` protocol is a requirement for any type you want to use as a key to a `Dictionary` or a member of a `Set`:

```
protocol Hashable : Equatable {
    var hashValue: Int { get }
}
```

It provides a single unique value you can use to represent an object. A `Dictionary`

requires keys to be unique, and you can use `hashValue` to determine if a key exists in the dictionary, needs to be added, or if the value exists and needs to be replaced. In a Set, you would use `hashValue` to ensure that the same element will only appear once, if added multiple times.

To implement `Hashable`, you should return an integer for `hashValue` that's guaranteed to be unique to a particular value. A good example of this is the `studentId` from the `Student` example:

```
extension Student: Hashable {  
    var hashValue: Int {  
        return studentId  
    }  
}
```

You can now use the `Student` type in a Set or Dictionary:

```
let john = Student(firstName: "Johnny", lastName: "Appleseed")  
  
// Dictionary  
let lockerMap: [Student: String] = [john: "14B"]  
  
// Set  
let classRoster: Set<Student> = [john, john, john, john]  
classRoster.count // 1
```

BooleanType

When you use an `if` statement, you typically use a `Bool` as an argument:

```
let isSwiftCool = true  
  
if isSwiftCool {  
    print("I agree!")  
}
```

One of the interesting "tricks" Swift uses in its protocol-heavy language design is that `if` statements don't actually require a `Bool` value, but instead a type that conforms to the `BooleanType` protocol:

```
protocol BooleanType {  
    var boolValue: Bool { get }  
}
```

This is possible because Swift uses structs for almost all of its core types, including `Bool`. In the standard library, the `Bool` struct conforms to `BooleanType` and simply returns itself for `boolValue`!

The flexibility afforded by using a protocol instead of the concrete `Bool` type allows you to create your own type that you can use in place of a `Bool`:

```
extension Record: BooleanType {  
    var boolValue: Bool {
```

```
        return wins > losses
    }
}
```

You can now use the Record struct directly within an `if` statement that will evaluate to true if the record is a winning record:

```
if Record(wins: 10, losses: 5) {
    print("winning!")
} else {
    print("losing :(")
}
```

CustomStringConvertible

While not as important as the previous protocol, the `CustomStringConvertible` protocol can help log and debug your objects.

When you call `print()` with an object such as `Record`, Swift prints a generic text representation of the object:

```
let record = Record(wins: 23, losses: 8)

print(record)
// {wins 23, losses 8}
```

The `CustomStringConvertible` has only a `description` property, which customizes how the object is represented both within `print()` statements and in the debugger:

```
protocol CustomStringConvertible {
    var description: String { get }
}
```

By adopting `CustomStringConvertible` on the `Record` type, you can provide a more readable representation:

```
extension Record: CustomStringConvertible {
    var description: String {
        return "\(wins) - \(losses)"
    }
}

print(record)
// 23 - 8
```

Key points

- Protocols define a contract that classes, structs and enums can **adopt**.
- By adopting a protocol, a type is required to **conform** to the protocol by implementing all methods and properties of the protocol.

- A type can adopt any number of protocols, which allows for a quasi-multiple inheritance not allowed with subclassing.
- You can use extensions for protocol adoption and conformance.
- The Swift standard library uses protocols extensively. You can use many of them, such as Comparable and Hashable, on your own named types.

Where to go from here?

Protocols help you decouple interface from implementation. Since protocols are types themselves, you can still declare an array of Vehicle instances. The array could then contain bicycles, trucks, cars, etc. In addition, bicycles could be an enumerations and trucks could be classes! But in the end, every Vehicle has a particular set of properties and methods that you know it will implement.

In the next chapter, you'll learn about protocol-oriented programming and see how you can use protocols as alternative to object-oriented techniques and inheritance when building your types.

Challenges

Challenge A: Bike protocols

Implement Comparable and Hashable on the Bike class. Create a Set of bikes of various wheel numbers and sizes, then sort them by their wheel size.

Note: You may simply use wheelSize to calculate hashCode. If you'd like, add another property of your choice to Bike that can make it even more unique.

Challenge B: Pet shop tasks

Create a collection of protocols for tasks that need doing at a pet shop. The pet shop has dogs, cats, fish and birds.

The pet shop duties can be broken down into these tasks:

- All pets need to be fed.
- Pets that can fly need to be caged.
- Pets that can swim need a tank.
- Pets that walk need exercise.

- Tanks and cages need to occasionally be cleaned.
1. Create classes or structs for each animal and adopt the appropriate protocols. Feel free to simply use a `print()` statement for the method implementations.
 2. Create homogenous arrays for animals that need to be fed, caged, cleaned, walked, and tanked. Add the appropriate animals to these arrays. The arrays should be declared using the protocol as the element type, for example `var caged: [Cageable]`
 3. Write a loop that will perform the proper tasks (such as feed, cage, walk) on each element of each array.

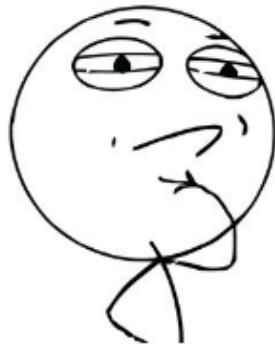
Chapter 20: Protocol-Oriented Programming

By Erik Kerber

When Apple announced Swift 2 at the World Wide Developers Conference in 2015, they declared Swift to be a "protocol-oriented programming language". This declaration was made possible by the introduction of **protocol extensions**.

Although protocols have been in Swift since the very beginning, this announcement along with the changes Apple made to the standard library to make heavy use of protocols changed the way we can think about our types.

In the previous chapter about protocols, you saw extensions to classes. However, extending *protocols* is also possible and is the key to an entirely new style of programming!



Tell me more...

In brief, **protocol-oriented programming** emphasizes coding to protocols, instead of to specific classes, structs or enums. It does this by breaking the old rules of protocols and allowing you to write *implementations* for protocols on the protocols themselves.

This chapter will change the way you think about and write code by introducing you to the power of protocol extensions and protocol-oriented programming. Along the

way, you'll learn how to use default implementations, type constraints, mixins and traits to vastly simplify your code.

Introducing protocol extensions

You briefly saw extensions in the previous chapter; they let you add additional computed properties or methods to a type:

```
extension String {
    func shout() {
        print(self.uppercaseString)
    }
}
```

Here, you're extending the `String` type itself to add a new method. You can extend any type in the system, including ones that you didn't write yourself. You can have any number of extensions on a type.

You define a protocol extension using the following syntax:

```
protocol TeamRecord {
    var wins: Int { get }
    var losses: Int { get }
    func winningPercentage() -> Double
}

extension TeamRecord {
    var gamesPlayed: Int {
        return wins + losses
    }
}
```

Similar to the way you extend a class, struct or enum, you use the keyword `extension` followed by the name of the protocol you are extending. Within the extension's braces, you define all the additional members of the protocol.

The biggest difference in the definition of a protocol extension, compared to the protocol itself, is that the extension includes the actual *implementation* of the member. In the example above, you define a new computed property named `gamesPlayed` that combines `wins` and `losses` to return the total number of games played.

Although you haven't written code for a concrete type that's adopting the protocol, you are able to use the members of the protocol within its extension. That's because the compiler knows that any type conforming to `TeamRecord` will have all the members of `TeamRecord`.

Now you can write a simple type that adopts `TeamRecord`, and use `gamesPlayed` without the need to re-implement it!

```
struct BaseballRecord: TeamRecord {
    var wins: Int
    var losses: Int

    func winningPercentage() -> Double {
        return Double(wins) / Double(wins) + Double(losses)
    }
}

let sanFranciscoSwifts = BaseballRecord(wins: 10, losses: 5)
sanFranciscoSwifts.gamesPlayed // 15
```

Since `BaseballRecord` conforms to `TeamRecord` you have access to `gamesPlayed`, which was defined in the protocol extension.

You can see how useful protocol extensions can be to define "free" behavior on a protocol, but this is only the beginning. Next, you'll learn how protocol extensions can provide implementations for members of the protocol itself!

Default implementations

As you learned in the previous chapter, a protocol is a way to define a contract for any type that adopts it. If a protocol defines a method or a property, any type that adopts the protocol *must* implement that method or property.

Consider the example of two `TeamRecord` types:

```
protocol TeamRecord {
    var wins: Int { get }
    var losses: Int { get }
    func winningPercentage() -> Double
}

struct BaseballRecord: TeamRecord {
    var wins: Int
    var losses: Int
    let seasonLength = 162

    func winningPercentage() -> Double {
        return Double(wins) / (Double(wins) + Double(losses))
    }
}

struct BasketballRecord: TeamRecord {
    var wins: Int
    var losses: Int
    let seasonLength = 82

    func winningPercentage() -> Double {
        return Double(wins) / (Double(wins) + Double(losses))
    }
}
```

The two TeamRecord types have identical implementations of winningPercentage(). You can imagine that most of the TeamRecord types will implement the function the same way. That could be a lot of repetitive code.

Fortunately, Swift has a shortcut:

```
protocol TeamRecord {
    var wins: Int { get }
    var losses: Int { get }
    func winningPercentage() -> Double
}

extension TeamRecord {
    func winningPercentage() -> Double {
        return Double(wins) / (Double(wins) + Double(losses))
    }
}
```

While this is much like the protocol extension you defined in the previous example, it differs in that winningPercentage() is a member of the TeamRecord protocol itself. By implementing a member of a protocol in an extension, you create what's known as a **default implementation** for that member.

You've already seen default arguments to functions and this is a similar idea: if you don't implement winningPercentage() in your type, then it'll use the default provided by the protocol extension.

In other words, you no longer need to explicitly implement winningPercentage() on types that adopt TeamRecord!

```
struct BaseballRecord: TeamRecord {
    var wins: Int
    var losses: Int
    let seasonLength = 162
}

struct BasketballRecord: TeamRecord {
    var wins: Int
    var losses: Int
    let seasonLength = 82
}
```

Default implementations let you add a capability to a protocol while greatly cutting down what would otherwise be repeated or "boilerplate" code.

As you may guess from the name, a default implementation doesn't prevent a type from implementing a protocol member on its own. Some team records may require a slightly different formula for the winning percentage, such as a sport that includes ties as a possible outcome:

```
struct HockeyRecord: TeamRecord {
    var wins: Int
    var losses: Int
```

```
var ties: Int

// Hockey record introduces ties, and has
// its own implementation of winningPercentage
func winningPercentage() -> Double {
    return Double(wins) / (Double(wins) + Double(losses) + Double(ties))
}
```

Now, if you call `winningPercentage()` on a `TeamRecord` that's a `HockeyRecord` value type, it will calculate the winning percentage as a function of wins, losses and ties. If you call `winningPercentage()` on another type that doesn't have its own implementation, it will fall back to the default implementation:

```
let baseballRecord: TeamRecord = BaseballRecord(wins: 10, losses: 6)
let hockeyRecord: TeamRecord = HockeyRecord(wins: 8, losses: 7, ties: 1)

baseballRecord.winningPercentage() // 10/(10+6) == .625
hockeyRecord.winningPercentage() // 8/(8+7+1) == .500
```

Mini-exercise

Write a default implementation on `CustomStringConvertible` that will simply remind you to implement `description` by returning `Remember to implement CustomStringConvertible!`.

In other words, once you have your default implementation, you should be able to write code like this:

```
struct MyStruct: CustomStringConvertible {}

print(MyStruct())
// should print "Remember to implement CustomStringConvertible!"
```

Understanding protocol extension dispatching

There's an important "gotcha" to keep in mind when defining protocol extensions such as the `gamesPlayed` property: If a type *redeclares* a method or property defined in an extension, the property or method that gets invoked is dependent on the context of the type declaration.

Suppose you were to define `BaseballRecord` with a `gamesPlayed` property that instead represented the total number of games played in one baseball season:

```
struct BaseballRecord: TeamRecord {
    var wins: Int
    var losses: Int
    let gamesPlayed: Int = 162

    func winningPercentage() -> Double {
        return Double(wins) / Double(wins) + Double(losses)
    }
}
```

```
}
```

The implementation used would then depend on the context of the variable:

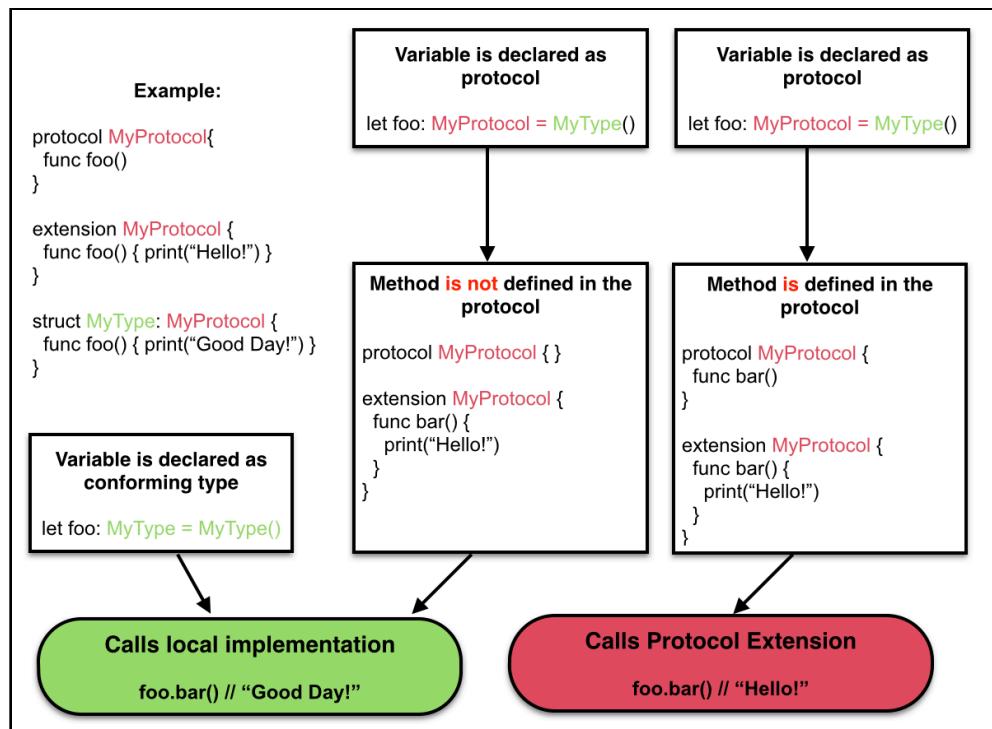
```
// Defined as a TeamRecord protocol
let team1: TeamRecord = BaseballRecord(wins: 10, losses: 5)
// Defined specifically as BaseballRecord
let team2: BaseballRecord = BaseballRecord(wins: 10, losses: 5)

// Uses gamesPlayed defined in the TeamRecord extension
team1.gamesPlayed // 15
// Uses gamesPlayed defined in BaseballRecord
team2.gamesPlayed // 162!!
```

You get this behavior because the `BaseballRecord` type is neither *conforming* to nor *overriding* the `gamesPlayed` property, and the compiler makes no connection between them. As an extension itself, `gamesPlayed` is simply a convenience and not part of the protocol.

If you're operating on a `TeamRecord`, it will use the `TeamRecord` extension version. Likewise, if you declare the type as a `BaseballRecord`, it will use the property defined in `BaseballRecord`.

If it all seems a bit confusing, fear not! The table below will help you understand the various dispatch rules Swift follows when you call a property or method on a protocol-conforming type:



Type constraints

For the protocol extensions on TeamRecord, you were able to use members of the TeamRecord protocol, such as wins and losses, within the implementations of winningPercentage() and gamesPlayed. Much like in an extension on a struct, class or enum, you write code as if you were writing instance methods on the type you're extending.

When you write extensions on protocols, there's an additional dimension to consider: The adopting type could also be any number of *other* types. In other words, when a type adopts TeamRecord, it could very well also adopt Comparable, CustomStringConvertible, or even another protocol you wrote yourself!

Swift lets you write extensions that are used only when the type adopting a protocol is also another type you specify. By using a **type constraint** on a protocol extension, you're able to use methods and properties from another type inside the implementation of your extension.

If that sounds like a bit much, an example of a type constraint will help clarify:

```
protocol PlayoffEligible {
    var minimumWinsForPlayoffs: Int { get }
}

extension TeamRecord where Self: PlayoffEligible {
    func isPlayoffEligible() -> Bool {
        return self.wins > minimumWinsForPlayoffs
    }
}
```

You have a new protocol, PlayoffEligible, that defines a minimumWinsForPlayoffs field. The "magic" happens in the extension of TeamRecord, which has a type constraint on Self: PlayoffEligible that will apply the extension to all adopters of TeamRecord that *also* adopt PlayoffEligible.

Applying the type constraint to the TeamRecord extension means that within the extension, self is known to be both a TeamRecord and PlayoffEligible. That means you can use properties and methods defined on both of those types.

You can also use type constraints to create default implementations on specific type combinations. Consider the case of HockeyRecord, which introduced ties in its record along with another implementation of winningPercentage():

```
struct HockeyRecord: TeamRecord {
    var wins: Int
    var losses: Int
    var ties: Int

    // Hockey record introduces ties, and has
    // its own implementation of winningPercentage
    func winningPercentage() -> Double {
```

```
        return Double(wins) / (Double(wins) + Double(losses) + Double(ties))
    }
```

Ties are common to more than just hockey, so you could make that a protocol instead of coupling it to just hockey specifically:

```
protocol Tieable {
    var ties: Int { get }
}
```

With type constraints, you can also make a default implementation for `winningPercentage()`, specifically for types that are both a `TeamRecord` and `Tieable`:

```
extension TeamRecord where Self: TieableRecord {
    func winningPercentage() -> Double {
        return Double(wins) / (Double(wins) + Double(losses) + Double(ties))
    }
}
```

Now the `HockeyRecord`, or any type that is both a `TeamRecord` and `Tieable`, won't need to explicitly implement a `winningPercentage()` that factors in ties:

```
struct HockeyRecord: TeamRecord, Tieable {
    var wins: Int
    var losses: Int
    var ties: Int
}

let hockeyRecord: TeamRecord = HockeyRecord(wins: 8, losses: 7, ties: 1)
hockeyRecord.winningPercentage() // .500
```

You can see that with a combination of protocol extensions and *constrained* protocol extensions, you can provide default implementations that make sense for very specific cases.

Mini-exercise

Write a default implementation on `CustomStringConvertible` that will print the win/loss record in the format Wins – Losses for any `TeamRecord` type. For instance, if a team is 10 and 5, it should return 10 – 5.

Protocol-oriented benefits

You've seen a number of the benefits of protocol extensions, but what exactly are the benefits of protocol-oriented programming?

Programming to interfaces, not implementations

By focusing on protocols instead of implementations, you can apply code contracts

to any type—even those that don't support inheritance.

Suppose you were to implement TeamRecord as a base class:

```
class TeamRecordBase {
    var wins: Int
    var losses: Int

    func winningPercentage() -> Double {
        return Double(wins) / (Double(wins) + Double(losses))
    }
}

// Will not build.
// Inheritance is only possible with classes.
struct BaseballRecord: TeamRecordBase {

}
```

At this point, you'd be stuck working with classes as long as you were working with team records.

Similarly, if you wanted to add ties to the mix, you'd either have to add ties to your subclass:

```
class HockeyRecord: TeamRecordBase {
    var ties: Int

    override func winningPercentage() -> Double {
        return Double(wins) / (Double(wins) + Double(losses) + Double(ties))
    }
}
```

Or you'd have to create *yet another* base class and thus deepen your class hierarchy:

```
class TieableRecordBase: TeamRecordBase {
    var ties: Int

    override func winningPercentage() -> Double {
        return Double(wins) / (Double(wins) + Double(losses) + Double(ties))
    }
}

class HockeyRecord: TieableRecordBase {}

class CricketRecord: TieableRecordBase {}
```

Likewise, if you wanted to work with any records that have wins, losses and ties, then you'd generally code against the lowest-common denominator base class:

```
extension TieableRecordBase {
    func totalPoints() -> Int {
```

```
        return (2 * wins) + (1 * ties)
    }
```

This forces you to "code to implementation, not interface". If you wanted to compare the records of two teams, all you care about is that there are wins and losses. With classes though, you'd need to operate on the specific base class that happens to define wins and losses.

I'm sure you don't want to hear what would happen if you suddenly needed to support divisional wins and losses on some sports! :]

With protocols, you don't need to worry about the specific type or even whether the thing is a class or a struct; all you care about is the existence of certain common properties and methods.

Traits, mixins and multiple inheritance

Speaking of supporting one-off features such as a divisional win or loss, one of the real benefits of protocols is that they allow a form of multiple inheritance.

When creating a type, you can use protocols to decorate it with all the unique characteristics you want:

```
protocol TieableRecord {
    var ties: Int { get }
}

protocol DivisionalRecord {
    var divisionalWins: Int
    var divisionalLosses: Int
}

protocol PointableRecord {
    func totalPoints() -> Int
}

extension PointableRecord where Self: TieableRecord, Self: TeamRecord {
    func totalPoints() -> Int {
        return (2 * wins) + (1 * ties)
    }
}

struct HockeyRecord: TeamRecord,
                    TieableRecord,
                    DivisionalRecord,
                    CustomStringConvertible,
                    Equatable {
    var wins: Int
    var losses: Int
    var ties: Int
    var divisionalWins: Int
    var divisionalLosses: Int
}
```

```

        var description: String {
            return "(wins) - (losses) - (ties)"
        }

        func ==(lhs: HockeyRecord2, rhs: HockeyRecord2) -> Bool {
            return lhs.wins == rhs.wins &&
                lhs.ties == rhs.ties &&
                lhs.losses == rhs.losses
        }
    }
}

```

This `HockeyRecord` is a `TeamRecord`, it is a `TieableRecord`, it keeps track of divisional wins and losses, it can be compared with `==` and it defines its own `CustomStringConvertible` description!

When you use protocols this way, it's often described as using **mixins** or **traits**. These terms reflect that you can add protocols to a type such that they don't interdepend on each other.

- **Mixin:** A protocol such as `Equatable` or `CustomStringConvertible` that defines additional characteristics of a type, but that doesn't include any code or implementation.
- **Trait:** A protocol that adds "free" behavior to a type. The `PointableProtocol` above is an example of a trait, because you provide the ability to calculate `totalPoints()` simply by making `HockeyRecord` conform to `PointableProtocol`.

There's no limit to the number of behaviors you can adopt in a type. This kind of compositional approach lets you build your types logically out of small pieces defined by protocols.

Simplicity

When you write a method to calculate the winning percentage, you only need wins, losses and ties. When you write code to print the full name of a person, you only need a first and a last name.

If you were to write code to do these tasks inside of a more complex object, it could be easy to make the mistake of coupling it with unrelated code:

```

func winningPercentage() -> Double {
    var percent = Double(wins) / Double(wins) + Double(losses)

    // Oh no! Not relevant!
    self.above500 = percent > .500

    return percent
}

```

That `above500` property might be needed for some reason in cricket, but not in hockey. However, that makes the function very specific to a particular sport.

You saw how simple the protocol extension version of this function was: it handled one calculation and that was it. Having simple default implementations that can be used throughout your types keeps the common code in one place.

You don't need to know that the type that adopts a protocol is a `HockeyRecord`, or a `StudentAthlete`, or a class, struct or enum. Because the code inside your protocol extension operates only on the protocol itself, *any* type that conforms to that protocol will also conform to your code.

As you'll discover again and again in your coding life, simpler code means less buggy code. :]

Why Swift is a protocol-oriented language

You've learned about the capabilities of protocols and protocol extensions, but you may be wondering: What exactly does it mean that Swift is a protocol-oriented language?

Protocol extensions greatly affect your ability to write expressive and decoupled code—and many of the design patterns that protocol extensions enable are reflected in the Swift language itself.

To begin, you can contrast protocol-oriented programming with object-oriented programming. The latter is focused on the idea of *objects* and how they interact. Because of this, the class is at the center of any object-oriented language.

Though classes are a part of Swift, you'll find they are an *extremely* small part of the standard library. Instead, Swift is built primarily on a collection of structs and protocols:

- Classes: 6
- Enum: 8
- Structs: 103
- Protocol: 86

You can see the significance of this in many of Swift's core types, such as `Int` and `Array`. Consider the definition of `Array`:

```
public struct Array<Element> : CollectionType, Indexable, SequenceType,  
    MutableCollectionType, MutableIndexable, _DestructorSafeContainer {  
    // ...  
}
```

Though that's quite a mouthful, there are two important takeaways:

1. An `Array` is a struct;

2. It adopts *a lot* of protocols.

The fact that `Array` is a struct means it's a value type, of course, but it also means that it can't be subclassed and it can't be a subclass. This relates to the second takeaway, which is that instead of inheriting behaviors from common base classes, `Array` adopts protocols to define many of its more common capabilities.

Why is this significant? It allows Swift to "decorate" `Array` with many common Swift behaviors. For example, both `Dictionary` and `Array` adopt `CollectionType`, but only `Array` adopts `MutableIndexable`—meaning it has a `startIndex` and an `endIndex`. On the other hand, `Dictionary` adopts `DictionaryLiteralConvertible`—meaning it has a key and a value.

This decoration of defined behaviors allows `Array` and `Dictionary` to be alike in some respects and different in others. Had Swift used subclassing, `Dictionary` and `Array` would either share one common base class or none at all. With protocols and protocol-oriented programming, you can treat them *both* as a `CollectionType`.

With a design centered around protocols rather than specific classes, structs or enums, your code is instantly more portable and decoupled, because methods now apply to a range of types instead of one specific type. Your code is also more cohesive, because it operates only on the properties and methods within the protocol you're extending and its type constraints, ignoring the internal details of any type that conforms to it.

Understanding protocol-oriented programming is a powerful skill that will help you become a better Swift developer, and give you new ways to think about how to design your code.



Note: More neutral-minded Swift developers will call Swift a "multi-paradigm" language. You've already seen inheritance and object-oriented techniques and now protocol-oriented programming; Swift can support them both!

Stay tuned for Chapter 23, "Functional Programming" where you'll see how you can use Swift in yet another "paradigm" or style.

Key points

- **Protocol extensions** let you write implementation code for protocols, and even write default implementations on methods required by a protocol.
- Protocol extensions are the primary driver for **protocol-oriented programming**, and let you write code that will work on any type that conforms to a protocol.
- **Type constraints** on protocol extensions provide additional context and let you write more specialized implementations.
- You can decorate a type with **traits** and **mixins** to extend behavior without requiring inheritance.

Where to go from here?

This chapter on protocol-oriented programming wraps up the section on building your own types. You're ready to take on the world of value vs reference types, picking enumerations over structs, and designing with protocols!

In the final section of this book, you'll learn about some more advanced language features. But not before getting to the challenges to solidify your knowledge of protocols, of course.

Challenges

Challenge A: Protocol extension practice

Suppose you own a retail store. You have food items, household items, clothes and electronics. Begin with an `Item` protocol:

```
protocol Item {
    var name: String { get }
    var clearance: Bool { get }
    var msrp: Float { get }
    func totalPrice() -> Float()
}
```

Fulfill the following requirements using primarily what you've learned about protocol-oriented programming. In other words, minimize the code in your classes, structs or enums.

- Clothes do not have sales tax, but all other items have 7.5% sales tax.
- When on clearance, food items are discounted 50%, household and clothes are discounted 25% and electronics are discounted 5%.
- Items should implement `CustomStringConvertible` and return `name`. Food items should also print their expiration dates.

Challenge B: Randomization

Write a protocol extension on `SequenceType` named `randomize()` that will rearrange the elements in random order. You can test out your implementation on an ordinary `Array`, which implements `SequenceType`.

Hints:

- Your method signature should be `randomize() -> [Generator.Element]`. The type `[Generator.Element]` is an array of whatever type (such as a `String` or an `Int`) the `SequenceType` holds. (You will learn more about this in the Generics chapter)
- You can call the `arc4random_uniform()` method like `this:arc4random_uniform(2)` to randomly generate a 1 or 0 for your randomization algorithm.

Section IV: Advanced Topics

You've made it to the final section of this book! In this section, you'll delve into some advanced topics to round out your Swift apprenticeship:

- **Chapter 21, Error Handling** – Errors are an everyday fact of life in programming, and handling them gracefully is an important part of writing stable and user-friendly code.
- **Chapter 22, Generics** – You've been using generics all along in Swift, and this chapter will explain what they are and how they relate to the types and collections you're already familiar with.
- **Chapter 23, Functional Programming** – Swift supports several programming paradigms, such as object-oriented programming and protocol-oriented programming. In this chapter, you'll learn the basics of functional programming, the philosophy behind it and why you might use it.

Chapter 21: Error Handling

By Janie Clayton

You've made it to the advanced topics section of this book! At this point, you might look at the title of this chapter and scoff, "Hey, I'm a good programmer. I don't need to worry about handling errors!" But you'll soon see that, regardless of your talent, error handling will be a constant in your life as a developer. And that's a good thing!

All programmers, especially skilled ones, need to worry about error handling. It isn't about you being a bad programmer—it's about acknowledging that you don't control everything.

In this chapter, you'll learn the fundamentals of error handling: what it is, how you implement it and when you need to worry about it.

What is error handling?

Error handling is the art of failing gracefully. You have complete control of your code, but you don't have complete control of anything outside of your code. This includes user input, network connections and any external files your app need to access.

Imagine you're in the desert and you decide to surf the Internet. You're miles away from the nearest Wi-Fi spot. You have no cellular signal. You open your Internet browser. What happens? Does your browser hang there forever with a spinning a wheel of death, or does it immediately alert you to the fact that you have no Internet access?

These are things you need to consider when you're designing the user experience for your apps, as well as the interfaces of your classes and structs. Think about what can go wrong and how you want your app to respond to it.

Start-Up-Opoly

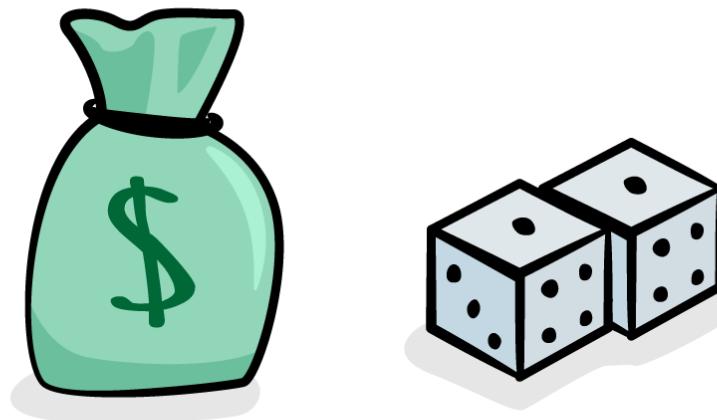
Start-Up-Opoly is a riff on the classic game of Monopoly. Here are the rules that are important for this exercise:

1. You get to travel around the board only as long as you have funding.
2. You lose funding if you roll doubles.
3. After you lose your funding, you can't move around the board until you raise another round of funding, which you get by, again, rolling doubles.

These rules are slightly different from the original Monopoly, but they'll work well to illustrate how to handle errors. Generally speaking, you expect to be able to move around the board. However, there are times when that action will fail, and you need your app to be able to deal with those situations.

Through the first half of this chapter, you'll use the game of Start-Up-Opoly to acquaint yourself with the basics of error handling in Swift. But first, let's look at the protocol that will make that possible.

In the start-up game, you get funded or...



YOU DIE!!!

The `ErrorType` protocol

Swift includes the `ErrorType` protocol, which works with the other parts of the error-handling architecture. It signals to the compiler that any data structure that complies to the protocol can be used for error handling.

The protocol can apply to any type you define, but it's especially well-suited to enumerations. As you learned in Chapter 16, "Enumerations," enumerations use case statements to specify all possible values, and the enum can only have one of those values, exclusive of the others. This is ideal for returning a specific error type.

In your example, you have two different error scenarios: You rolled a double, or you have no funding.

You can create an error enum in your code:

```
enum RollingError: ErrorType {  
    case Doubles  
    case OutOfFunding  
}
```

The cases define your two error scenarios.

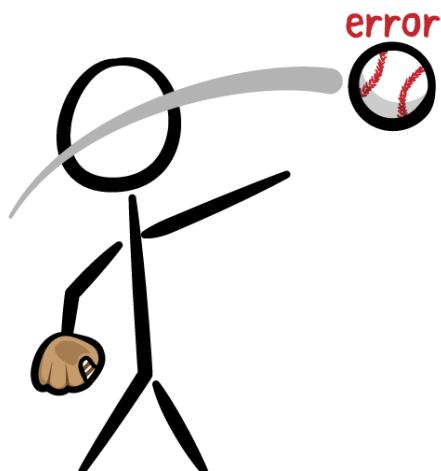
Now that you have your custom error type in place, you can begin implementing it in your functions.

Throwing errors

You've created a nice enum of errors that can happen when you try to move after rolling your dice. Cool, but what does your program do with these errors? It throws them, of course! That's the actual terminology you'll see: **throwing** errors, and then **catching** them.

has funding == false

"Hey boys! We've got another
error coming in!"



After your program throws an error, you need to handle that error. There are two ways to approach this problem: You can handle your errors immediately, or you can bubble them up to another level.

To choose your approach, you need to think about where it makes the most sense to handle the error. If it makes sense to handle the error immediately, then do so. If, however, you're in a situation where you have to alert the user and have her take action, but you're several function calls away from a user interface element, then it makes sense to bubble up the error until you reach the point where you can alert the user.

It's entirely up to you when to handle the error, but *not* handling it isn't an option. Swift requires you to handle your error at some point in the chain, or your program won't compile.

Roll function

The dice roll is a fundamental component of Start-Up-Opoloy. In fact, it's the only component! Since you need to determine how to proceed after a roll, whatever the result, the function managing that is the perfect place to set up your error handling:

```
var hasFunding = true

func roll(firstDice: Int, secondDice: Int) throws {
    let error: RollingError

    if firstDice == secondDice && hasFunding { // 1
        error = .Doubles
        hasFunding = false
        throw error
    } else if firstDice == secondDice && !hasFunding { // 2
        hasFunding = true
        print("Huzzah! You raise another round of funding!")
    } else if !hasFunding { // 3
        error = .OutOfFunding
        throw error
    } else { // 4
        print("You moved \(firstDice + secondDice) spaces")
    }
}
```

Your funding status is a piece of state you need to continuously track. You start the game with funding intact, so you can begin playing immediately.

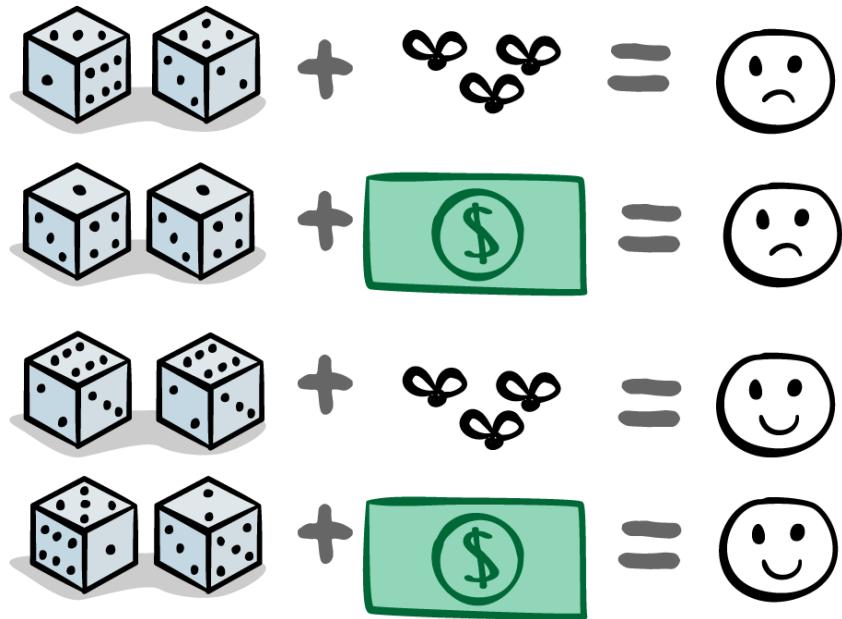
As you know, from here on, doubles control funding. If you roll a double, you lose funding, and you don't get it back until you roll another double, which means you've raised more funding.

There are four cases to check:

1. The first `if` statement checks if you rolled doubles and already have funding. In

that case, you need to change your funding status.

2. If you don't have funding and you need it, then rolling doubles is good!
3. If you don't have funding, then you can't move until you do.
4. Lastly, if you've checked every case where your roll might fail, you get to add a default case that handles most of your rolls.



The throws keyword

Your function doesn't only take two "dice" as a parameter—it also includes the word `throws`. If you're writing an error-throwing function, you need to use the keyword `throws` before the return type. If you're calling an error-throwing function but not handling the errors in that function, you also need to include `throws` in your method declaration.

However, when you get to the function that's catching and handling the errors, you *don't* need to include `throws` in the declaration. Since you aren't bubbling up the error any longer, it's no longer a throwing function.

Move function

Since your last function was `roll`, it's time to implement the `move` function:

```
func move(firstDice: Int, secondDice: Int) -> String {
    do {
        try roll(firstDice, secondDice: secondDice)
        return "Successful roll."
    } catch RollingError.Doubles {
        return "You rolled doubles and have lost your funding"
    } catch RollingError.OutOfFunding {
```

```
    return "You need to do another round of funding."  
} catch {  
    return "Unknown error"  
}  
}
```

You're only calling one function in here, which is `roll`. Since `roll` can throw errors, you embed it in a `do` block. That also introduces a new scope, but more importantly sections off the code that can potentially throw errors.

If the function doesn't throw an error, you're done with your logic. But if `roll` does throw an error, you need to catch it. There are two possible errors you need to catch and handle: `Doubles` and `OutOfFunding`.

In this example, you alert the user that she can't move on the board and tell her why. Soon enough, you'll learn more complex error-handling tricks you can master.

Mini-exercise

With the above code in place, try calling `move` a few times. What do you think the results will be from the following calls?

```
move(1, secondDice: 2)  
move(4, secondDice: 4)  
move(1, secondDice: 6)
```

Advanced error handling

Cool, you know how to handle an error! That's neat, but how do you expand it to an entire class or app? How do you scale your error handling and think about it in the larger context of a complex app?

In the rest of this chapter, you'll learn how to handle optional values and failable initializers, and how to chain errors together to create a complex web of actions that must all be true for your function to succeed.

PugBot

The sample project you'll work with in this second half of the chapter is a **PugBot**. The PugBot is cute and friendly, but sometimes it gets lost and confused. As the programmer of the PugBot, it's your responsibility to make sure it doesn't get lost on the way home from your PugBot lab.



The PugBot will
Love You FOREVER!

You'll learn how to make sure that the PugBot you're working with is one that *you* programmed, and how to fail gracefully if it isn't. You'll also learn how to handle the unfortunate situation where you have to come to the rescue of a lost PugBot.

First, you need to set up an enum containing all of the directions in which your PugBot will move:

```
enum Direction {  
    case Left  
    case Right  
    case Forward  
}
```

Failable Initializers

The first thing you have to worry about, when you're starting out with your PugBot class, is the possibility that the lost PugBot isn't able to get back home. You can get around this by declaring your instance to be optional, but that doesn't quite resolve your problem. How do you deal with initializing an optional class instance?

You use a special kind of initializer—a **failable initializer**—that can fail without crashing your app. The failable initializer for your PugBot class looks like this:

```
class PugBot {  
    let name: String  
    let correctPath: [Direction]  
    var currentStepInPath = 0  
  
    init? (name: String, correctPath: [Direction]) {  
        self.correctPath = correctPath
```

```
    self.name = name  
  
    // 1  
    guard (correctPath.count > 0) else {return nil}  
  
    // 2  
    switch name {  
        case "Delia", "Olive", "Frank", "Otis", "Doug":  
            break  
        default:  
            return nil  
    }  
}
```

You have two pieces of information you need to worry about with your PugBot: its name and a set of directions to get it home. There are two ways that instantiating a PugBot can fail:

1. If you don't receive any directions for your PugBot, then you can't ensure it gets home, because it lacks the ability to do so.
2. You have five PugBots you're responsible for. If you initialize a PugBot with the name of one of these pugs, then you're responsible for making sure it gets home. If it doesn't get one of those names, then your PugBot will fail to initialize and you won't be able to do anything about it.

You can check your initializer using the code below:

```
let rightDirections: [Direction] =  
    [.Forward, .Left, .Forward, .Right]  
let wrongDirections: [Direction] =  
    [.Left, .Left, .Left, .Forward]  
  
let invalidPug = PugBot(name: "Lassie", correctPath: rightDirections)  
let myPugBot = PugBot(name: "Delia", correctPath: rightDirections)!  
let wrongPugBot = PugBot(name: "Delia", correctPath: wrongDirections)!
```

Since Lassie isn't a pug or a name associated with your PugBot class, the `invalidPug` variable is `nil`. Delia is a valid pug and you're able to handle her progress home.

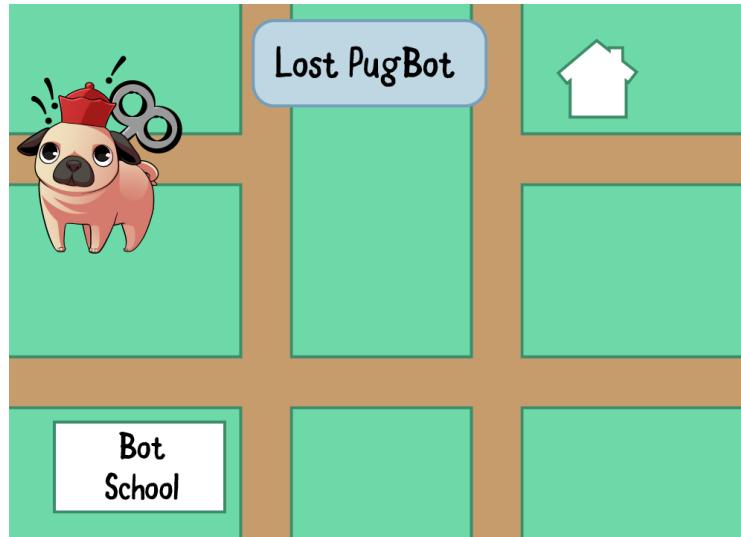
You'll notice that there are two sets of directions, one that's right and one that's wrong. You might be wondering how you can know the directions you're receiving are the right ones. Just keep reading to get the answer.

Chaining error-throwing functions

In the Start-Up-Opoly example, you learned how to handle errors from a single-step operation. There are also times you'll need a series of successful operations to be sure your app is working the way you'd like.

In your PugBot app, your PugBot will need to make a series of moves to get home

successfully. If your PugBot takes a wrong turn somewhere, it will be lost and will need you to rescue it.



First you need an associated error enum that will connect to each direction in which the PugBot can move:

```
enum PugBotError: ErrorType {
    case DidNotTurnLeft(directionMoved: Direction)
    case DidNotTurnRight(directionMoved: Direction)
    case DidNotGoForward(directionMoved: Direction)
    case EndOfPath
}
```

Since you're tracking both where your PugBot is supposed to go and where it's actually going, you can also keep track of the direction your PugBot went if it took a wrong turn. That's pretty cool!

Last but not least, you need to set up some methods in your PugBot class to correlate to the errors and directions:

```
func turnLeft() throws {
    guard (currentStepInPath < correctPath.count) else {
        throw PugBotError.EndOfPath
    }

    let direction = correctPath[currentStepInPath]
    if direction != .Left {
        throw PugBotError.DidNotTurnLeft(directionMoved: direction)
    }
    currentStepInPath++
}

func turnRight() throws {
    guard (currentStepInPath < correctPath.count) else {
        throw PugBotError.EndOfPath
    }
    let direction = correctPath[currentStepInPath]
```

```
if direction != .Right {
    throw PugBotError.DidNotTurnRight(directionMoved: direction)
}
currentStepInPath++
}

func moveForward() throws {
    guard (currentStepInPath < correctPath.count) else {
        throw PugBotError.EndOfPath
    }
    let direction = correctPath[currentStepInPath]

    if direction != .Forward {
        throw PugBotError.DidNotGoForward(directionMoved: direction)
    }
    currentStepInPath++
}
```

In your initializer, you create an array of directions to be passed into the PugBot instance, and a variable to keep track of where your PugBot is in the directions. Each of these methods can fail and throw an error in two cases:

1. In the first case, the method fails if you try to get your PugBot to continue moving after it gets home. You check the number of steps your pug has currently taken versus where it is in the array of passed-in directions. If you have more steps than you do directions, you throw an error specifying that your pug has already reached the end of the path.
2. The second case is if the direction isn't the one you expect. Not only does your thrown error have the direction in which the PugBot was *supposed* to move, the associated value also includes the direction in which your PugBot *actually* moved. These pieces of information will help you narrow down where your PugBot went awry.

Finally, you can construct your goHome method, which is where you'll be giving your PugBot the set of directions it needs to follow to get home:

```
func goHome() throws {
    try moveForward()
    try turnLeft()
    try moveForward()
    try turnRight()
}
```

You create a somewhat complex set of instructions to "program" into your PugBot to tell it how to get home. If the PugBot follows the directions to the letter, it will get home, no problem. But if, sadly, the PugBot gets lost, your method will throw an error immediately.

Every single one of these commands must pass for the method to complete successfully. The second your chain throws an error, your PugBot will stop trying to get home and will stay put until you come and rescue it.

Wrapping and handling multiple errors

Since you're a smart developer, you've noticed that you're not handling your errors in `goHome()`. You've also probably noticed that you're handling all of your directional methods in the same way. If you try to handle each and every method you call in `goHome()`, you'll get up to 20 or 30 calls, and your code will become very cluttered.

You can avoid all of this by wrapping your error handling in its own function:

```
func movePugBotSafely(move:() throws -> ()) -> String {
    do {
        try move()
        return "Completed move successfully."
    } catch PugBotError.DidNotTurnLeft(let directionMoved) {
        return "The PugBot was supposed to turn left, but turned \
(directionMoved) instead."
    } catch PugBotError.DidNotTurnRight(let directionMoved) {
        return "The PugBot was supposed to turn right, but turned \
(directionMoved) instead."
    } catch PugBotError.DidNotGoForward(let directionMoved) {
        return "The PugBot was supposed to move forward, but turned \
(directionMoved) instead."
    } catch PugBotError.EndOfPath() {
        return "The PugBot tried to move past the end of the path."
    } catch {
        return "An unknown error occurred"
    }
}
```

This function takes a function or closure as a parameter, as you saw in Chapters 7 and 8. You tell your function that it accepts another function that takes no parameters, returns no value, and throws an error.

This creates a nice, generic function that you can reuse and plug in without repeating any code. The signature matches all of the turn methods you wrote in your PugBot class, so you can pass in all of them as parameters. Your function then returns a string.

In your `do` block, you try the function that you pass in as a parameter. If it doesn't throw an error, you return a success message. If, however, it does throw an error, you now need to handle it.

Here's where you're able to go through and catch each individual error. For the directional errors, you're able to catch each error and its associated value. You can take those pieces of information and generate a specific error message about the direction the pug went and the direction it was supposed to go. You also catch the last error in your enum if your PugBot tries to move past the end of the path.

You might notice that you have to add a default case at the end. What gives? You've exhausted the cases in your `PugBotError` enum, so why is the compiler hassling you?

Unfortunately, at this point, Swift's Do-Try-Catch system isn't type-specific. There's no way to tell the compiler that it should only expect PugBotErrors. To the compiler, that isn't exhaustive, because it doesn't handle each and every possible error that it knows about, so you still need to have a default case even though the method will never call it.

Now you can use your function to handle movement safely:

```
movePugBotSafely {  
    try myPugBot.goHome()  
    try myPugBot.moveForward()  
}
```

Thanks to trailing closure syntax, your movement calls are cleanly wrapped in the call to `movePugBotSafely(_:_)`. Here, your PugBot will go home safely.



From there, if you try to move forward then the PugBot will be moving beyond the end of the path, which generates an error.

Key points

- You can create an enum that conforms to the **ErrorType** protocol that works with Swift's error-handling paradigm.
- Any function that can throw an error or calls a function that can throw an error has to be marked with **throws**.
- When calling an error-throwing function, you must embed it in a **do** block. Within that block, you **try** the function, and if it fails, you **catch** it.
- If you have a struct that might fail when being instantiated, you can use a **failable initializer** to ensure that the structure can return `nil` without crashing the app.

- You can chain a complex series of commands together knowing that if any link in the chain fails, the rest of the commands won't execute.

Where to go from here?

Handling errors is a necessary part of programming. You never know when things will go wrong, and it's important to make sure that both end users of your app and fellow programmers using your code have a good error-handled experience!

With `try` and `throws` and other language features, Swift makes it very explicit where things can go wrong. It's up to you now to *try* writing error-free code, but to keep in mind where failures can happen and make sure you signal those problems when they occur.

Chapter 22: Generics

By Alexis Gallagher

The truth is, you already know about generics. Every time you use a Swift array, you're using generics. This might even give the impression that generics are *about* collections, but that impression is both incorrect and misleading.

In this chapter, you'll learn about the fundamentals of what generics *are*. This will give you a solid foundation for understanding how to define your own generic code. Finally, you'll loop back to look at generic types in the Swift standard library—arrays, dictionaries and optionals—using this new perspective.

Introducing generics

To get started, you'll consider a stylized contrast that highlights the essential nature of generics, which is this: generics provide a means of expressing systematic variation *at the level of types, as opposed to the level of values*.

Values defined by other values

Suppose you're running a pet shop that sells only dogs and cats, and you want to use a Swift playground to model that business.

To start, you define a type, `PetKind`, that can hold two possible values corresponding to the two kinds of pets that you sell, dogs and cats:

```
enum PetKind {  
    case Cat  
    case Dog  
}
```

So far, so good. Now suppose you want to model not just the animals but also the employees, the pet keepers who look after the pets. Your employees are highly specialized. Some keepers only look after cats, and others only dogs.

So you define a `KeeperKind` type, as follows:

```
struct KeeperKind {  
    var keeperOf: PetKind  
}
```

Then you can initialize a `catKeeper` and `dogKeeper` in the following way:

```
let catKeeper = KeeperKind(keeperOf: .Cat)  
let dogKeeper = KeeperKind(keeperOf: .Dog)
```

There are two points to note about how you're modeling your shop.

First, you're representing the different kinds of pets and keepers by *varying the values of types*. There's only one type for pet kinds—`PetKind`—and one type for keeper kinds—`KeeperKind`. Different kinds of pets are represented only by distinct values of the `PetKind` type, just as different kinds of keepers are represented by distinct values of the `KeeperKind` type.

Second, *one range of possible values determines another range of possible values*. Specifically, the range of possible `KeeperKind` values directly mirrors the range of possible `PetKind` values.

If your store started selling birds, you'd simply add a `.Bird` member to the `PetKind` enumeration, and you'd immediately be able to initialize a value describing a bird keeper, `KeeperKind(keeperOf: .Bird)`. And if you started selling a hundred different kinds of pets, you'd immediately be able to represent a hundred different kinds of keepers.

In contrast, you could have defined a second unrelated enumeration instead of `KeeperKind`:

```
enum EnumKeeperKind {  
    case CatKeeper  
    case DogKeeper  
}
```

In this case, nothing would enforce this relationship except your diligence in always updating one type to mirror the other. If you added `PetKind.Snake` but forgot to add `EnumKeeperKind.SnakeKeeper`, then things would get out of whack.

But with `KeeperKind`, you explicitly established the relationship via a property of type `PetKind`. Every possible `PetKind` value implied a corresponding `KeeperKind` value. Or you could say, the set of possible `PetKind` values defines the set of possible `KeeperKind` values.

To summarize, you can depict the relationship like so:

PetKind values	KeeperKind values
.Cat	KeeperKind(keeperOf:.Cat)
.Dog	KeeperKind(keeperOf:.Dog)
etc.	etc.

Types defined by other types

The model above fundamentally works by varying the *values of types*. Now consider another way to model the pet-to-keeper system, by varying *the types themselves*.

Suppose that instead of defining a single type PetKind that represents all kinds of pets, you chose to define a distinct type for every kind of pet you sell. This is quite a plausible choice if you're working in an object-oriented style, where you model the pets' behaviors with different methods for each pet. Then you'd have the following:

```
struct Cat {}
struct Dog {}
```

Now how do you represent the corresponding kinds of keepers? You could simply write the following:

```
struct KeeperForCats {}
struct KeeperForDogs {}
```

But that's no good. It has *exactly* the same problem as does manually defining a parallel enum of KeeperKind values—namely, that it relies on you to enforce the required domain relationship that there's one kind of keeper for every kind of pet.

What you'd really like is a way to *declare* a relationship just like the one you established for values. You'd like to declare that every possible pet type implies the existence of a corresponding keeper type, a correspondence that you'd depict like so:

Pet types	Keeper types
Cat	Keeper (of Cat...)
Dog	Keeper (of Dog...)
etc.	etc.

That is, you'd like to establish that for every possible pet type, there is defined a corresponding Keeper type. But you don't want to have to do this manually; you want a way to automatically define a set of new types for all the keepers.

This, it turns out, is exactly what generics are for!

Anatomy of generic types

Generics provide a mechanism for using one set of types to define a new set of types.

In your example, you can define a **generic type** for keepers, like so:

```
struct Keeper<T> {}
```

This definition immediately defines all the corresponding keeper types, as desired:

Pet types	Keeper types
Cat	Keeper<Cat>
Dog	Keeper<Dog>

You can verify these types are real by creating values of them, specifying the entire type in the initializer:

```
var aCatKeeper = Keeper<Cat>()
```

What's going on here? First, `Keeper` is the name of a generic type.

But you might say that a generic type isn't really a type at all. It's more like a recipe for making real types, or **concrete types**. One sign of this is the error you get if you try to instantiate it in isolation:

```
var aThirdKeeper = Keeper() // compile-time error!
```

The compiler complains here because it doesn't know what kind of keeper you want. That `T` in angle brackets is the **type parameter** that specifies the type for the kind of animal you're keeping.

If you remember back to arrays and optionals, for example, the long-style declaration looked like this:

```
var myStrings: Array<String>
let age: Optional<Int>
```

It wouldn't make sense to declare an `Array` or `Optional` without the type parameter. An array of what? An optional value that might contain what? The type parameters are required right at compile time to say it's an array of strings, or an optional integer.

Once you provide the required type parameter, as in `Keeper<Cat>`, the generic `Keeper` has now become a new concrete type. `Keeper<Cat>` is different from `Keeper<Dog>`, even though they started from the same generic type; these resulting concrete types are called **specializations** of the generic type.

To summarize the mechanics, in order to define a generic type like `Keeper<T>`, you only need to choose the name of the generic type and of the type parameter. In one stroke this defines a *set* of new types. Those are all the specializations of `Keeper<T>` implied by all possible concrete types that one could substitute for the type parameter `T`.

Notice that the type `Keeper` doesn't currently store anything at all, or even use the type `T` in any way. Essentially, generics are a way to systematically define sets of types.

Using type parameters

Usually, though, you'll want to *do* something with type parameters.

Suppose you want to keep better track of individuals. First, you enrich your type definitions to include identifiers—that is, names. This lets every value represent the identity of an individual animal or keeper:

```
struct Cat {  
    var name: String  
}  
  
struct Dog {  
    var name: String  
}  
  
struct Keeper<T> {  
    var name: String  
}
```

You also want to track which keeper looks after which animals. Suppose every keeper is responsible for one animal in the morning and another in the afternoon. You can express this by adding properties for the morning and afternoon animals. But what type should those properties have?

Clearly, if a particular keeper only manages dogs, then the properties must only hold dogs. And if cats, then cats. In general, if it's a keeper of `T`, then the morning and afternoon animal properties should be of type `T`.

To express this, you merely need to *use* the type parameter that previously only distinguished the nature of your keeper types:

```
struct Keeper<T> {  
    var name: String  
    var morningAnimal: T  
    var afternoonAnimal: T  
}
```

By using `T` in the body of the generic type definition above, you can express that the morning and afternoon animals must be the kind of animal the keeper knows best.

Just as function parameters become constants to use within the function body, you can use type parameters such as `T` throughout your type definitions. You can use the type parameter anywhere in the definition of `Keeper<T>`, not only for stored properties, but also for computed properties, method signatures or nested types.

Now when you instantiate a `Keeper`, Swift will make sure the types all match up:

```
let jason = Keeper(name: "Jason",
                    morningAnimal: Cat(name: "Whiskers"),
                    afternoonAnimal: Cat(name: "Sleepy"))
```

Here, the keeper Jason manages the cat Whiskers in the morning and the cat Sleepy in the afternoon. The type of `jason` is `Keeper<Cat>`.

Mini-exercises

- Try instantiating another `Keeper`, but this time for dogs.
- What do you think would happen if you tried to instantiate a `Keeper` with a dog in the morning and a cat in the afternoon?

Naming your type parameters

Generic type parameters are a bit like function parameters, which means you can understand them better by looking at the parallels with how you understand parameters in other contexts.

One parallel concerns basic naming style. Variables, in general, are *named placeholders* for undetermined values, and there are two naming conventions. Sometimes, you use neutral identifiers, like `x` and `y`, to suggest the generality of a variable that could mean anything or take any value. Other times, you use meaningful identifiers, like `velocity` and `name`.

Type parameters are treated in the same way. Sometimes you'll see neutral identifiers, like the capitals `T`, `U`, `V`, and at other times, meaningful names like `Key`, `Value` or `Element`.

Type inference with generics

In general, you provide the compiler all the additional information it needs in the context of use, that is, in the call site of the type's initializer or method, or at the point of a declaration of a variable, constant or property.

Here's an example where you explicitly provide all the types:

```
let favoriteIntsExplicit = Array<Int>(arrayLiteral: 8, 9, 42)
```

You have an `Array` with type parameter `Int` specified for the array's generic element type.

Note: If you look at how Swift arrays are defined in the generated headers, they use a meaningful name: `Array<Element>`

You can also rely on Swift's type inference. Since the type parameter `Element` appears in `init(arrayLiteral elements: Element...)`, the Swift compiler will infer that the `Element` is equal to the type of whatever value you use there. Since you provide `Int` values there, you don't need to explicitly pass `Int` into the type generic parameter list:

```
let favoriteIntsInferred = Array(arrayLiteral: 8, 9, 42)
```

However, if you wanted to declare that your numeric literals 8, 9 and 42 should be interpreted as another type, you could again explicitly pass a type parameter to do that:

```
let favoriteFloatsInferred = Array<Float>(arrayLiteral: 8, 9, 42)
```

In all these examples, you're providing the type information at the call site of the initializer. The same syntax, and the same choice of explicit vs inferred, are available for type annotations in variable, constant and property declarations:

```
let favoriteIntsExplicit2: Array<Int>      = [8, 9, 42]
let favoriteIntsInferred2: Array              = [8, 9, 42]
let favoriteFloatsInferred2: Array<Float> = [8, 9, 42]
```

Note: Since arrays are so common, Swift offers the shorthand notation you're probably used to by now: `[Element]` rather than `Array<Element>`. We're using the full generic syntax here just to make it clear what's going on behind the scenes.

Arrays are pretty simple: they take a single type parameter, and that type can be seemingly anything. You've seen arrays of integers, strings, and other custom types you've defined.

Dictionaries

However, Swift generics also allow for multiple type parameters and for more complex sets of restrictions on them. These let you use generic types and protocols with associated types to model complex algorithms and data structures. A straightforward example of this is dictionaries.

`Dictionary` has two type parameters in the comma-separated generic parameter list that falls between the angle brackets, as you can see in its declaration:

```
struct Dictionary<Key : Hashable, Value> // etc..
```

Key and Value represent the types of the dictionary's keys and values. But the annotation on Key, Key : Hashable, says more. Everything after that colon is a **type constraint**. A type constraint indicates a required supertype, and/or a required protocol or list of protocols, for any type that will serve as the argument for that type parameter.

For instance, the type constraint for Dictionary requires that any type that will serve as the key for the dictionary be hashable, because the dictionary is a hash map, which must hash its keys to enable fast lookup.

For types such as Dictionary with multiple type parameters, just provide a comma-separated list:

```
let intNames: Dictionary<Int, String> = [42: "forty-two"]
```

As with arrays, dictionaries get some special treatment in Swift since they're built-in and so common. You've already seen the type style [KeyType: ValueType], and you can also use type inference:

```
let intNames2: [Int: String] = [42: "forty-two", 7: "seven"]
let intNames3 = [42: "forty-two", 7: "seven"]
```

Optionals

Finally, no discussion of generics would be complete without mentioning optionals. In Chapter 16, "Enumerations" you learned that optionals are implemented as enumerations, but they're also "just another" generic type, which you could have defined yourself.

Suppose you were writing an app that let a user enter her birthdate but didn't require it. You might find it handy to define an enum type, as follows:

```
enum OptionalDate {
    case None
    case Some(NSDate)
}
```

Similarly, if another form allowed but didn't require the user to enter her last name, you might define the following type:

```
enum OptionalString {
    case None
    case Some(String)
}
```

Then you could capture all the information a user did or did not enter into a struct with properties of those types:

```
struct Person {  
    // other properties here  
    var birthday: OptionalDate  
    var lastName: OptionalString  
}
```

And if you found yourself doing this repeatedly for new types of data the user might not provide, then at some point you'd want to generalize this into a generic type that represented the concept of "a value of a certain type that might be present", and you'd write:

```
enum Optional<T> {  
    case None  
    case Some(T)  
}
```

At this point, you would have reproduced Swift's own `Optional<T>` type, since this is exactly the definition in the Swift standard library! It turns out, `Optional<T>` is close to being a plain old generic type, like one you could write yourself.

Why do I say "close"? It would be a plain old generic, like one you could write yourself, if you interacted with optionals only by writing out their full enum-based types, like so:

```
var birthdate: Optional<NSDate> = Optional<NSDate>.None  
if birthdate == Optional<NSDate>.None {  
    // no birthdate  
}
```

But, of course, it's more common and conventional to write something like this:

```
var birthdate: NSDate? = nil  
if birthdate == nil {  
    // no birthdate  
}
```

In fact, those two code blocks are saying exactly the same thing. The second is just relying on special language support for generics: The `T?` shorthand syntax for specifying the optional type `Optional<T>`, and `nil`, which can stand for the `.None` value of an `Optional<T>` specialized on any type.

As with arrays and dictionaries, optionals get a privileged place in the language with this syntax to make using them more concise. But all of these features provide more convenient ways to access the underlying type, which is just a generic type.

Generic functions

So far, you've looked at definitions of generic types, the types being classes, structs or enums. But you can also define standalone generic functions.

Instead of the angle-bracketed generic parameter list that comes after the type name, the generic parameter list comes after the function name. You can then use the generic parameter in the body of the function definition. This function takes two arguments and swaps their order:

```
func swapped<T, U>(x: T, _ y: U) -> (U, T) {
    return (y, x)
}

swapped(33, "Jay") // returns ("Jay", 33)
```

A generic function definition demonstrates a confusing aspect about the syntax: having both type parameters and function parameters. You have both the generic parameter list of type parameters `<T, U>`, and the list of function parameters `(x: T, _ y: U)`.

Think of the type parameters as arguments *for the compiler*, which it uses to define one possible function. Just as your generic Keeper struct meant the compiler could make dog keepers and cat keepers and any other kind of keeper, the compiler can now make a non-generic specialized swapped function for any two types for you to use.

Key points

- Generics express systematic variation at the level of types, via a **type parameter** that ranges over possible concrete type values.
- Generics are like functions *for the compiler*. They are evaluated at compile time and result in new types which are specializations of the generic type.
- A generic type is not a real type on its own, but more like a recipe, program, or template for defining new types.
- Generics are everywhere in Swift, in optionals, arrays, dictionaries, other collection structures, and so on.

Where to go from here?

In Swift, generics are everywhere! Every Swift array and every optional is a generic. Every time you write `nil`, you're using a generic. Even the most basic operators like `+` and `==` are all defined with generics.

So it's fair to say, it's impossible to write Swift without using generics. However, because they're integrated so gracefully into the language, it's surprisingly easy to write Swift without really understanding what generics are. But they are worth understanding, because it helps a great deal when it's time to dig into the standard library or write your own generic code.

Generics are fundamentally a mechanism for allowing systematic variation at the level of types, as opposed to values. They let you write code that defines an infinite or a carefully circumscribed set of types or functions in one expression. But this isn't magic. To understand how generics do this, it helps to think of them as functions that are run not by your program, but by the compiler when it builds your program.

If you simply squint a little as you keep this perspective in mind, all the regular ways you think about parameters, variables, constraints and evaluation apply directly to generics.

Challenges

Challenge A: Building a collection

Consider the pet and keeper example from earlier in the chapter:

```
struct Cat {  
    var name: String  
}  
  
struct Dog {  
    var name: String  
}  
  
struct Keeper<T> {  
    var name: String  
    var morningAnimal: T  
    var afternoonAnimal: T  
}
```

Imagine that instead of looking after only two animals, every keeper looks after a changing number of animals throughout the day. It could be one, two, ten animals per keeper instead of just morning and afternoon ones.

You'd want to do things like this:

```
let christine = Keeper<Cat>(name: "Christine")  
  
christine.append(someCat)  
christine.append(anotherCat)
```

Then you'd want to access the count of all of animals for a keeper like `christine.count` and access them by index like `christine.animalAtIndex(50)`.



Of course, you're describing your old friend the array type, `Array<Element>`!

Your challenge is to update the `Keeper` struct to have this kind of interface. You'll probably want to include a private array inside `Keeper` and then provide methods and properties on `Keeper` to allow outside access to the array.

Chapter 23: Functional Programming

By Alexis Gallagher

Swift was invented to replace Objective-C, a language created and designed to support object-oriented programming. Swift was also built to develop apps for OS X and iOS, which requires intimate interaction with Cocoa or Cocoa Touch, two app development frameworks that are deeply object-oriented.

However, Swift is more than that. As its designer Chris Lattner observed in an August 18, 2014 post on the Apple Developer Forum, Swift "dramatically expands the design space through the introduction of generics and **functional programming** concepts."

The truth is, Swift is a hybrid language. It supports multiple programming styles. But because Swift itself is new, unlike other mainstream languages, it's had the opportunity to incorporate ideas from functional programming into its most basic structure.

In this chapter, you'll learn the basics of functional programming:

- To start, you'll acquaint yourself with the classic higher-order function, `map`.
- Then you'll zoom out and consider what functional programming *is* and how `map` supports it.
- After that, you'll dive into the other two classic higher-order functions, `filter` and `reduce`, using them to demonstrate simple data manipulation.

Higher-order functions: `map`

Any function that takes another function as a parameter is called a **higher-order function**. But there are three higher-order functions in particular—you can call them the Big Three—that are considered the classics.

`map` is the first of these. It's more convenient than sliced bread and easier to use

than a doorknob. Take the old example of a pet store. Suppose you're printing signs for your shop, and you have an array of strings listing your inventory:

```
let animals = ["cat", "dog", "sheep", "dolphin", "tiger"]
```

For your signage, you'd like an array that has all those names capitalized. You've already defined a function `capitalize` that takes a `String` and returns a capitalized `String`:

```
func capitalize(s: String) -> String {  
    return s.uppercaseString  
}
```

So how do you create an array of capitalized names? One way is simply to create the new array and populate it:

```
var uppercaseAnimals: [String] = []  
  
for animal in animals {  
    let uppercaseAnimal = capitalize(animal)  
    uppercaseAnimals.append(uppercaseAnimal)  
}  
  
uppercaseAnimals // ["CAT", "DOG", "SHEEP", "DOLPHIN", "TIGER"]
```

First, you create the empty output array `uppercaseAnimals` before iterating through every animal name in the input array. For every name, you call `capitalize(_:)` and add the return value to the output array.

This works great. But a few days later, you decide you want pictures in your sign. In Swift, a `Character` can represent most Unicode characters, including emojis. You learned this back in Chapter 4, "Strings".

As it happens, all your uppercase animal names are standard Unicode character names for emojis. You've already defined the following function, which takes a `String` and returns a `Character` with the emoji of the named animal:

```
func characterForCharacterName(c: String) -> Character {  
    let curlyBracedCharacterName = "\N{\(c)}"  
    let charStr = curlyBracedCharacterName.stringByApplyingTransform(  
        NSStringTransformToUnicodeName, reverse: true)  
    return charStr!.characters.first!  
}
```

Since Unicode characters are helpfully named, this function changes a character name such as "DOG" into the corresponding emoji.

So how do you create an array of these pictures? Just like with capitalization, one way is simply to create the new array and populate it:

```
var animalEmojis: [Character] = []

for uppercaseAnimal in uppercaseAnimals {
    let emoji = characterForCharacterName(uppercaseAnimal)
    animalEmojis.append(emoji)
}

animalEmojis // ["🐱", "🐶", "🐑", "🐬", "🐯"]
```

Here, you create the empty output array `animalEmojis` before iterating through every animal name in the input array. For every name, you call `characterForCharacterName(_:)`, and add the return value to the output array.

Does this code, and explanation, sound familiar? That's the key point! These two snippets are similar to each other, just as they are similar to thousands of loops written over the years. They both start with an *input array* and then create an *output array*, which they populate by iterating over every input element and applying a *transform function* to each.

Aside from the names of the variables, the only real thing that differs are the values and types of the input array, the transform function and the output array. You can see the parallels in this table:

	capitalization	rendering
inputs	animals	uppercaseAnimals
outputs	uppercaseAnimals	animalEmojis
transform	capitalize	characterForCharacterName

In other words, both cases fit exactly into this pattern:

```
let inputs: [InputType] = /* some list of InputType */
var outputs: [OutputType] = []

for inputItem in inputs {
    let outputItem = transform(inputItem)
    outputs.append(outputItem)
}

outputs // /* some list of OutputType values */
```

As you saw in the previous chapter on Generics, you could treat this as the body of a generic function definition:

```
func map<InputType, OutputType>(inputs: [InputType],
    transform: (InputType)->(OutputType)) -> [OutputType]
{
    var outputs:[OutputType] = []

    for inputItem in inputs {
        let outputItem = transform(inputItem)
```

```
    outputs.append(outputItem)
}

return outputs
}
```

This generic function has two generic types, one for input and one for output. As function parameters, you need to pass in the input array as well as another function to do the transformation from `InputType` to `OutputType`.

All that's left to do is the familiar iteration and building the output array.

And with this function, you can perform the earlier operations much more simply.

```
let uppercaseAnimals2 = map(animals, transform: capitalize)
let animalEmojis2 = map(uppercaseAnimals, transform:
characterForCharacterName)
```

Congratulations! You've just defined `map`, the gateway to functional programming.

To put it into one sentence: `map` takes an array of values and a transform function, and returns a new array generated by applying the transform function to every element of the input array.

The essential abstraction here is that all of these loops are the same, and only the transform function changes. So you put your own logic into the transform and factor out the standard boilerplate part into `map`.

Swift's version of `map`

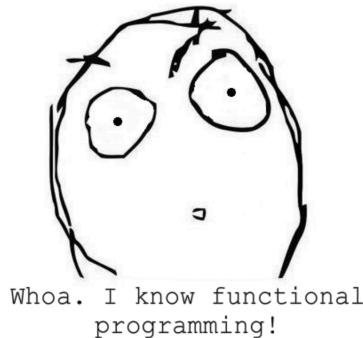
While the version of `map` you've defined above is quite serviceable, Swift's built-in `map` is different in a few ways that make it safer and handier. First, it isn't defined as a function at all but as a method. So you call it as a method on the input array in the familiar object-oriented style, like so:

```
let uppercaseAnimals3 = animals.map(capitalize)
let animalEmojis3 = uppercaseAnimals.map(characterForCharacterName)
```

Second, it isn't defined as a method on `Array<T>`, but on the protocol `SequenceType`, as well as on various other types. So if you implement your own sequence type or collection type, `map` will already be there waiting for you.

Introducing functional programming?

That was `map`. Boom. Welcome to functional programming.



But given that anyone can implement `map` in a couple of lines of code, and it's perhaps merely a convenience for removing the boilerplate of a loop, how can `map` demonstrate anything? To answer these questions, you must first tackle the wider and amorphous question of what *is* functional programming.

Philosophy of functional programming

To put it simply, functional programming is a *programming style* that treats the *mathematical function* as the primary unit of abstraction, in order to allow *clear reasoning* about programs. Let's take these points of emphasis one by one.

The functional style of programming

First, functional programming is a *style* of programming, just like object-oriented programming is a style. Style is about the way that you do something, not the thing that you do.

Now it's true that some languages are designed to support some styles more than others. For instance, C isn't designed to support object-oriented programming. However, you can still write C in an object-oriented style, and in fact Apple's entire Core Foundation framework is designed to enable this.

A "functional programming language" is simply a language that has been designed to include support for the functional programming style. But fundamentally, a programming style is about how you approach your problem.

So what is the functional programming style? It's based on the function. Just as an object-oriented programming style aims to solve problems by representing them in terms of objects, the functional programming style tries to describe problems and their solutions in terms of functions.

Specifically, the aim is to use *mathematical functions*, by which is meant not "functions that use math" but "the function as it is understood in mathematics".

Note: More technically, in the context of functional programming, these are called *pure functions*, but you'll call them mathematical functions here to emphasize the point.

Mathematical functions versus computational functions

What's the difference between functions in mathematics and the computational functions you use in programming languages? The key difference is just this: mathematical functions are not *instructions about what to do*, but instead are *assertions of a relationship* between their arguments and their returned values.

Say you're writing a function that describes how many miles you'll drive in an hour if you're driving 60 miles per hour. You might write:

```
d(t) = 60 * t
```

This asserts a relationship, namely, that the distance in miles traveled, d , is sixty times the time elapsed in hours, t . As long as you keep going in a straight line at sixty miles an hour, that relationship continues to be true.

From that relationship, it follows that you travel 120 miles in 2 hours, or $d(2) = 120$. And it will *always be true* that $d(2) = 120$. There is no scenario in which $d(2) = 120$ the "first time" you calculate it, and then $d(2)$ equals some other value the "second time" you calculate it.

Because the equation simply describes a truth, there is no way that "checking on it" can change it, any more than you need to look up your friend's birthday more than once because you're worried someone changed it when you weren't looking. Truths just keep being true.

In contrast, suppose you import a Swift module containing a function with the signature `distanceTraveled(t: Float) -> Float`, which purports to tell you the distance traveled at 60 miles per hour. What about this? Is this a mathematical function?

In fact, you don't know!

It might be the case that the function is implemented in a straightforward manner, as follows:

```
func distanceTraveled(t: Float) -> Float {  
    return 60 * t  
}
```

In that case, the function is just like the algebraic function above. The distance depends only on time, and the same time value will always yield the same distance value.

But it might also be that the function does other things. In fact, there are just

about a million surprising things the function could be doing under the hood. What if it depends on some random number, like this:

```
func distanceTraveled(t: Float) -> Float {
    let booster = 1 + Float(arc4random_uniform(2))
    return booster * 60 * t
}
```

Then, `distanceTraveled(2)` = 120 half of the time, and the rest of the time `distanceTraveled(2)` = 240.

Even if a function isn't generating a random variable, any variable you don't control is just as bad for the purpose of getting a predictable result. For instance, what if you define the function like this:

```
func distanceTraveled(t: Float) -> Float {
    let booster = externalVariable
    return booster * 60 * t
}
```

Then the result depends on `externalVariable`. What is it? Does it change? Who controls it? Maybe whoever controls it is populating it with a new random number, every half a second! The point is, these are all questions you wouldn't need to ask if that variable weren't there.

In the functional programming style, the idea is to avoid external state and to have functions give you the same result every time you give it the same input.

Clear reasoning

As you can see, there are a multitude of things you can *do* in a computational function that might make its behavior surprising. The idea of writing in a functional programming style is simply to eschew those things.

Instead, as much as possible, write a function so that its inputs completely determine its outputs, so that the same inputs always produce the same outputs, and so that it has no side effects anywhere else.

If you ensure these things, then your computational function will behave like a plain old mathematical function. Another way to think of it is like this: If you can pretend that your function is simply looking things up in some vast constant dictionary, then it's a mathematical function. And the benefits of this are that it will be easier to understand and easier to use within a larger system.

For example, you can imagine the distance calculation function from above as this kind of lookup. Imagine if you built a massive dictionary of values:

```
[ 1: 60, 1.5: 90, 2: 120, 3: 180, ... ]
```

If this dictionary were fine-grained enough, you wouldn't be able to tell whether asking for the distance was coming from a dictionary like this, or from a function

that does the calculation.

Generally speaking, your mathematical-style functions do less than computational functions. Ideally, they *do* nothing at all.

So how do you build a significant program, and define a programming style, based on this sort of restriction? Now that you have a loose definition of the functional programming style, your friend `map` provides a good answer.

Map as philosophy in practice

In practice, the functional programming style described in the last section, which aims to use the mathematical function as the primary unit of abstraction, boils down to a few basic programming practices.

Prefer immutable values

The first practice is to *prefer immutable values*. In Swift, this means preferring constants to variables. If you say `let x = 5` instead of `var x = 5`, when you're using a type with value semantics like `Int`, then you're defining an `x` that cannot change as your program executes.

In other words, you've defined your computational variable so that it behaves just like a mathematical variable, and you can pretend it's like you've written the mathematical equation $x = 5$.

One way `map` helps you use immutable values is by letting you define the output array as a constant. Look at again at your Dark Ages-style loop for generating an array of emoji:

```
var animalEmojis: [Character] = []
for uppercaseAnimal in uppercaseAnimals {
    let emoji = characterForCharacterName(uppercaseAnimal)
    animalEmojis.append(emoji)
}

animalEmojis // ["🐱", "🐶", "🐑", "🐬", "🐯"]
```

Here the output array `animalEmojis` must be a variable since you change it within the loop. But probably you don't want to change it outside that context. You want to treat it as a constant value, like the 5 in `let x = 5`.

Because `map` abstracts the process of building the array, it lets you define `animalEmojis` as a constant, directly representing the core idea that you're starting with one (mathematical) value, `uppercaseAnimals`, and using it to generate another (mathematical) value, `animalEmojis`:

```
let animalEmojis = uppercaseAnimals.map(characterForCharacterName)
```

Use pure functions

The second basic practice for programming in a functional style is to use *pure functions*. A pure function is the formal name for a function that behaves like a mathematical function: Its output depends only on its inputs, and it doesn't modify its inputs or anything else outside the function.

If you can pretend that the function is doing nothing but looking up a value in a dictionary, then it's a pure function.

`map` works best when its argument, the transform function, is a pure function. When the transform function is pure, then `map` itself acts as a pure function.

Since `characterForCharacterName` is a pure function and doesn't modify the `String` passed into it or any other state in the system, you know that the above line won't modify `uppercaseAnimals` even if `uppercaseAnimals` is a variable.

Treat functions as values

A third basic practice for functional programming is *treating functions as values*. This simply means passing functions as arguments, storing them into variables and returning them from other functions.

You've already learned about functions and closures as values in chapters 7–8, and you've seen it again here with `map`: you pass in a small piece of logic, a function that takes one element type to another element type.

For this purpose, it's useful and common to use closures to make things more concise. For instance, you can remove `capitalize` and rewrite the snippet that calculates uppercase animal names more tersely, like so:

```
let uppercaseAnimals4 = animals.map({ $0.uppercaseString })
```

You'll switch to this closure-based style for the rest of the chapter.

Two more classics: filter and reduce

`map` is the first of the Big Three higher-order functions. As you've seen, `map` takes a transform function that operates on elements, and provides a kind of `transform` method that operates on collections like arrays. Now let's meet the other two.

Filter

The second one is `filter`. `filter` takes a **predicate function**, a function that takes a single value element and returns either true or false, indicating if the element passes some test.

For instance, suppose you wanted to see only those animals with three-character

names. You could use filter as follows:

```
let threeCharacterAnimals = animals.filter() {
    $0.characters.count == 3
}

threeLetterAnimals // => ["cat", "dog"]
```

The closure passed into filter here returns true only if the string in the collection has three characters. filter calls this predicate for each element in the collection, and returns a new collection with the filtered results.

Reduce

The third of the Big Three functions is reduce, which is more versatile compared to its siblings. While map and filter both transform a collection into another collection, reduce transforms a collection into *any* type of your choosing.

reduce takes a **combining function**, which combines an element of the input array with a value of the final result type. Then reduce combines the elements from the input array, one at a time, with successive result values, until it's processed the entire input array, and then it returns the final result value.

For instance, you can think of the operation of summing an array of numbers as the operation of reducing them:

```
func sum(items: [Int]) -> Int {
    return items.reduce(0, combine: +)
}

let total = sum([1, 2, 3])
total // => (((0 + 1) + 2) + 3) == 6
```

reduce takes an initial value as the first parameter. In this case, since you're calculating the sum, the initial value is 0. The combining function is simply the + operator.

Or you can think of concatenating an array of strings as the operation of reducing them, with an initial result value of the empty string and the combining function of string concatenation, also represented by the + operator:

```
func concatenate(items: [String]) -> String {
    return items.reduce("", combine: +)
}

let phrase = concatenate(["Hello", " ", "World"])
phrase // => ((("" + "Hello") + " ") + "World") == "Hello World"
```

If map is easier to use than a doorknob, then you might say reduce is more versatile than a Swiss Army knife. If you had to take one higher-order function to a desert island, it should be reduce. You can actually use it to build map, filter and many other functions, as well!

Key points

- Functional programming is a programming style, like object-oriented or protocol-oriented programming.
- Many Swift language features exist to support a functional programming style, and originated in functional programming languages.
- Most fundamentally, a functional programming style boils down to three practices: preferring immutable values, preferring pure functions, and making use of higher-order functions.

Where to go from here?

You've discovered the "big idea" behind the functional programming style and discussed basic practices for following that style in Swift. You've looked at `map` in detail, including building an implementation of it yourself, and also taken a quick look at `filter` and `reduce`.

Functional programming is a large and vibrant topic, which is unfortunately impossible to illustrate with short concrete examples, even though `map`, `filter` and `reduce` can give a tantalizing taste of it.

Swift's language features have been strongly influenced by functional programming. But this fact is part of a wider trend of functional programming language features becoming more widespread. As a result, one could say that more and more programmers are doing functional programming without much ceremony around it, or without even realizing it.

But if you're curious, there's a much wider world of ideas and methods to explore, and Swift is an excellent vehicle for doing so.

Challenges

Challenge A: Using reduce

Suppose the worst came to pass: you were in fact stranded on a desert island with only `reduce`.

How would you define `map` and `filter` using only `reduce`, and without using any iteration constructs?



Conclusion

We hope this book has helped you get up to speed with Swift. Swift is filled with language features and programming paradigms, and we hope you now feel comfortable enough with the language to move on to building bigger things.

With the language fundamentals under your belt, you're ready to crack your knuckles with iOS apps, OS X apps, the latest killer tvOS app and who knows what else!

If you have any questions or comments as you continue to use Swift, please stop by our forums at <http://www.raywenderlich.com/forums>.

Thank you again for purchasing this book. Your continued support is what makes the tutorials, books, videos, conferences and other things we do at raywenderlich.com possible—we truly appreciate it!

Wishing you all the best in your continued Swift adventures,

– Matt, Erik, Eli, Ben, Alexis, Janie, Jawwad, Rui, Audrey, Tim, Bradley and Greg

The *Swift Apprentice* team