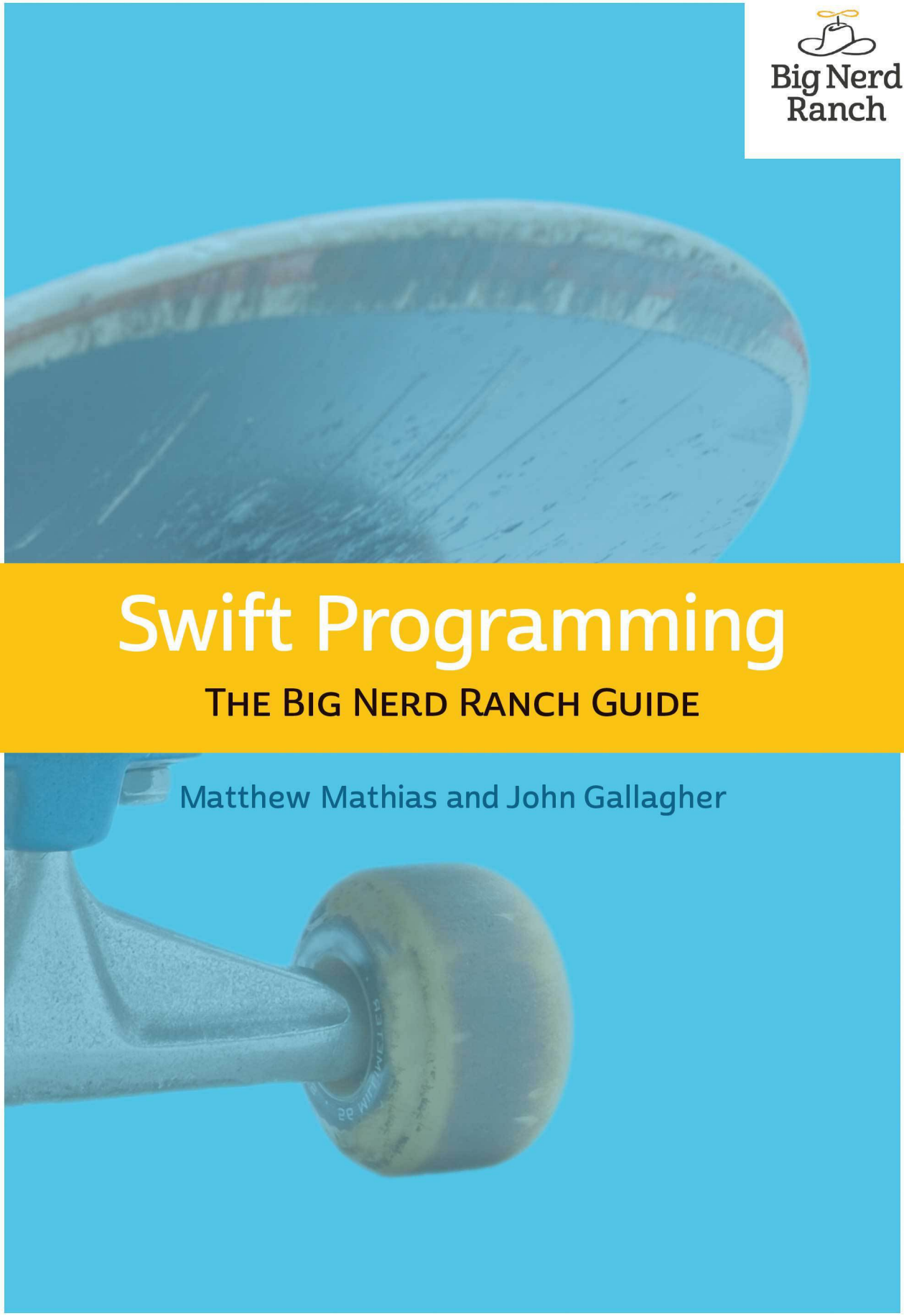


Big Nerd Ranch

Swift Programming

The Big Nerd Ranch Guide

Matthew Mathias and John Gallagher

Swift Programming: The Big Nerd Ranch Guide

by Matthew Mathias and John Gallagher

Big Nerd Ranch, LLC
200 Arizona Ave NE
Atlanta, GA 30307
(770) 817-6373
http://www.bignerdranch.com/
book-comments@bignerdranch.com

Exclusive worldwide distribution of the English edition of this book by

Pearson Technology Group
800 East 96th Street
Indianapolis, IN 46240 USA
http://www.informit.com

ISBN-10 0134398041
ISBN-13 978-0134398044

First edition, first printing, December 2015
Release D.1.1.1

Dedication

For my wife, who is smart, strong, and virtuous. And for my family, who has given me every opportunity to live a good life.

— M.M.

For my wife and best friend; you are "s'wonderful." And for my daughters, who bring me joy every day.

— J.G.

This page intentionally left blank

Acknowledgments

We received a lot of help in writing this book. Without it, this book would not be what it is, and it may have never even happened. Thanks are due.

First, we need to say thank you to our colleagues at Big Nerd Ranch. Thank you to Aaron Hillegass for providing us with the opportunity to write this book. It has been immensely gratifying to learn and teach Swift. Big Nerd Ranch provided us with the time and space to work on this project. We hope that this book lives up to the trust and the support that we have received.

Particular thanks are also due to our colleagues in the Cocoa Pod at Big Nerd Ranch. Your careful teaching revealed many bugs in the text, and your thoughtful recommendations led to many improvements in our approach. Those of you who are not instructors helped to review the materials, vetted our approach, and provided countless suggestions that we never thought of. It is truly wonderful to have colleagues such as you. Thank you Pouria Almassi, Matt Bezark, Nate Chandler, Step Christopher, Kynerd Coleman, Matthew Compton, Joseph Dixon, Robert Edwards, Sean Farrell, Brian Hardy, Florian Harr, Tom Harrington, Bolot Kerimbaev, Christian Keur, JJ Manton, Bill Monk, Chris Morris, Adam Preble, Scott Richie, Jeremy Sherman, Steve Sparks, Rod Strougo, TJ Usiyan, Zach Waldowski, Thomas Ward, and Mike Zornek.

Our colleagues in operations and sales are instrumental. Classes would literally never be scheduled without their work. Thank you Shannon Coburn, Nicole Rej, Heather Brown, Tasha Schroader, Mat Jackson, and Chris Kirksey for all of your hard work. We cannot do what you do.

Second, we need to acknowledge the many talented folks who worked on the book with us.

Elizabeth Holaday, our editor, helped refine the book, crystallize its strengths, and diminish its weaknesses.

Simone Payment, our copy-editor, found and corrected errors and ultimately made us look smarter than we are.

Ellie Volckhausen designed our cover; that skateboard looks pretty rad.

Chris Loper designed and produced the print book and the EPUB and Kindle versions.

Finally, thank you to our students. We learned with you and for you. Teaching is part of the greatest thing that we do, and it has been a pleasure working with you. We hope that the quality of this book matches your enthusiasm and determination.

This page intentionally left blank

Table of Contents

This page intentionally left blank

Introduction

Learning Swift

Apple's World Wide Developers Conference is an annual landmark event for its developer community. It is a big deal every year, but 2014 was particularly special: Apple introduced an entirely new language called Swift for the development of iOS and OS X applications.

As a new language, Swift represents a fairly dramatic shift for Mac OS X and iOS developers. More experienced iOS developers have something new to learn, and new developers cannot rely on a venerable community for tried and true answers and patterns. Naturally, this shift creates some uncertainty.

But this is also an exciting time to be a Mac OS X and iOS developer. There is a lot to learn in a new language, and this is especially true for Swift. The language has evolved quite a bit since its beta release in the summer of 2014, and it continues to evolve.

We are all at the forefront of this language's development. As new features are added to Swift, its users can collaboratively determine its best practices. You can directly contribute to this conversation, and your work with this book will start you on your way to becoming a contributing member of the Swift community.

Whither Objective-C?

So, what about Objective-C, Apple's previous *lingua franca* for its platforms? Do you still need to know that language? For the time being, we think that answer is an unequivocal "Yes." Apple's `Cocoa` library, which you will use extensively, is written in Objective-C, so debugging will be easier if you understand that language. Moreover, most learning materials and existing Mac and iOS apps are written in Objective-C. Indeed, Apple has made it easy, and sometimes preferable, to mix and match Objective-C with Swift in the same project. As an iOS or Mac developer, you are bound to encounter Objective-C, so it makes sense to be familiar with the language.

But do you need to know Objective-C to learn Swift? Not at all. Swift coexists and interoperates with Objective-C, but it is its own language. If you do not know Objective-C, it will not hinder you in learning Swift. (We will only use Objective-C directly in one chapter toward the end of this book, and even then it will not be important for you to understand the language.)

Prerequisites

We have written this book for all types of iOS and Mac OS X developers, from platform experts to first-timers. For readers just starting software development, we will highlight and implement best practices for Swift and programming in general. Our strategy is to teach you the fundamentals of programming while learning Swift. For more experienced developers, we believe this book will serve as a helpful introduction to your platform's new language. So while having some development experience will be helpful, we do not believe that it is necessary in order to have a good experience with this book.

We have also written this book with numerous examples so that you can refer to it in the future. Instead of focusing on abstract concepts and theory, we have written in favor of the practical. Our approach

favors using concrete examples to unpack the more difficult ideas and also to expose the best practices that make code more fun to write, more readable, and easier to maintain.

How This Book Is Organized

This book is organized in six parts. Each is designed to accomplish a specific set of goals that build on each other. By the end of the book, you will have built your knowledge of Swift from that of a beginner to a more advanced developer.

Getting Started	This part of the book focuses on the tools that you will need to write Swift code and introduces Swift's syntax.
The Basics	*The Basics* introduces the fundamental data types that you will use every day as a Swift developer. This part of the book also covers Swift's *control flow* features that will help you to control the order in which your code executes.
Collections and Functions	You will often want to gather related data in your application. Once you do, you will want to operate on that data. Swift offers *collections* and *functions* to help with these tasks.
Enumerations, Structures, and Classes	This part of the book covers how you will model your data in your own development. We cover the differences between these types and make some recommendations on when to use each.
Advanced Swift	As a modern language, Swift provides a number of more advanced features that enable you to write elegant, readable, and effective code. This part of the book discusses how to use these elements of Swift to write idiomatic code that will set you apart from more casual Swift developers.
Event-Driven Applications	This part of the book walks you through writing your first Mac OS X and iOS applications. For readers working with older Mac OS X or iOS applications, we conclude this part of the book by discussing how to interoperate between Objective-C and Swift.

How to Use This Book

Programming can be tough, and this book is here to make it easier. How can we help you with that? Follow these steps:

- Read the book. Really! Do not just browse it nightly before going to bed.
- Type out the examples as you read along. Part of learning is muscle memory. If your fingers know where to go and what to type without too much thought on your part, then you are on your way to becoming a more effective developer.

- Make mistakes! In our experience, the best way to learn how things work is to first figure out what makes them not work. Break our code examples and then make them work again.
- Experiment as your imagination sees fit. Whether that means tinkering with the code you find in the book or going off in your own direction, the sooner you start solving your own problems with Swift, the faster you will become a better developer.
- Do the challenges we have included in most chapters. As we mentioned, it is important to begin solving problems with Swift as soon as possible. Doing so will help you to start thinking like a developer.

More experienced developers may not need to go through some of the earlier parts of the book. *Getting Started* and *The Basics* may be very familiar to some developers.

One caveat: In *The Basics*, do not skip the chapter on Optionals as they are at the heart of Swift, and in many ways they define what is unique about the language.

Subsequent chapters like Arrays, Dictionaries, Functions, Enumerations, and Structs and Classes may seem like they will not present anything new to the practiced developer, but we feel that Swift's approach to these topics is unique enough that every reader should at least skim these chapters.

Last, remember that learning new things takes time. Dedicate some time to going through this book when you are able to avoid distractions. You will get more out of the text if you can.

Challenges

Many of the chapters conclude with an exercise for you to work through on your own. These are an excellent opportunity for you to challenge yourself. In our experience, truly deep learning is accomplished when you solve problems in your own way.

For the More Curious

Relatedly, we include sections entitled "For the More Curious" at the end of many chapters. These sections address questions that may have occurred to the curious reader working through the chapter. Sometimes, we discuss how a given language feature's underlying mechanics work, or we may explore a programming concept not quite related to the heart of the chapter.

Typographical Conventions

You will be writing a lot of code as you work through this book. To make things easier, we use a couple of conventions to identify what text is old, what should be added, and what should be removed. For example, in the function implementation below, you are deleting the text `print("Hello")` and adding `print("Goodbye")`.

```
func talkToMe() {
    ~~print("Hello")~~
    print("Goodbye")
}
```

Necessary Hardware and Software

To build and run the applications in this book, you will need a Mac running OS X Yosemite (10.10) or newer. You will also need to install Xcode, Apple's *integrated development environment* (IDE), which is available on the App Store. Xcode includes the Swift compiler as well as other development tools you will use throughout the book.

Swift is still under rapid development. This book is written for Swift 2.0 and Xcode 7.0. Many of the examples will not work as written if you are using an older version of Xcode. If you are using a newer version of Xcode, it is possible there may have been changes in the language that will cause some examples to fail.

As this book is moving into the printing process, Xcode 7.1 Beta is available. The code samples in the book work with the latest beta version we have been able to use. If future versions of Xcode do cause problems, take heart – the vast majority of what you learn will continue to be applicable to future versions of Swift even though there may be changes in syntax or names. You can also check out our forums at `http://forums.bignerdranch.com` for help.

Before We Begin

We hope to show you how much fun it can be to make applications for the Apple ecosystem. While writing code can be extremely frustrating, it can also be gratifying. There is something magical and exhilarating about solving a problem, not to mention the special joy that comes out of making an app that helps people and brings them happiness.

The best way to improve at anything is with practice. If you want to be a developer, then let's get started! If you find that you do not think you are very good at it, who cares? Keep at it and we are sure that you will surprise yourself. Your next steps lie ahead. Onward!

This page intentionally left blank

3
Conditionals

In previous chapters your code led a relatively simple life: you declared some simple constants and variables and then assigned them values. But of course, an application really comes to life – and programming becomes a bit more challenging – when the application makes decisions based on the contents of its variables. For example, a game may let players leap a tall building *if* they have eaten a power-up. You use conditional statements to help applications make these kind of decisions.

if/else

`if/else` statements execute code based on a specific logical condition. You have a relatively simple either/or situation and depending on the result one branch of code or another (but not both) runs. Consider Knowhere, your small town from the previous chapter, and imagine that you need to buy stamps. Either Knowhere has a post office or it does not. If it has a post office, you will buy stamps there. If it does not have a post office, you will need to drive to the next town to buy stamps. Whether there is a post office is your logical condition. The different behaviors are "get stamps in town" and "get stamps out of town."

Some situations are more complex than a binary yes/no. You will see a more flexible mechanism called `switch` in Chapter 5. But for now, let's keep it simple.

Create a new OS X playground and name it **`Conditionals`**. Enter the code below, which shows the basic syntax for an `if/else` statement:

Listing 3.1 Big or small?

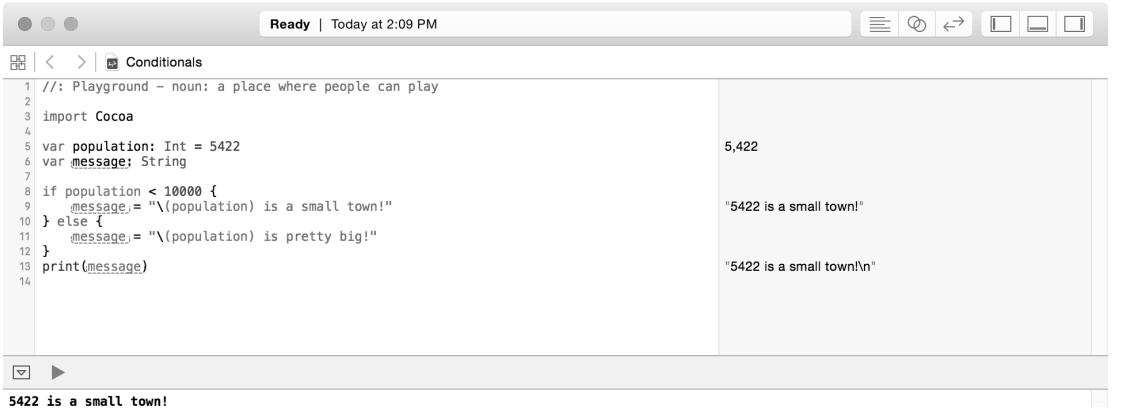
```
import Cocoa

var population: Int = 5422
var message: String

if population < 10000 {
    message = "\(population) is a small town!"
} else {
    message = "\(population) is pretty big!"
}

print(message)
```

You first declare `population` as an instance of the **`Int`** type and then assign it a value of 5422. Next, you declare a variable called `message` that is of the **`String`** type. You leave this variable uninitialized at first, meaning that you do not assign it a value.

Next comes the conditional `if/else` statement. This is where `message` is assigned a value based on whether the "if" statement evaluates to true. (Notice that you use *string interpolation* to put the population into the `message` string.)

Figure 3.1 shows what your playground should look like. The console and the results sidebar show that `message` has been set to be equal to the string literal assigned when the conditional evaluates to true. How did this happen?

Figure 3.1 Conditionally describing a town's population



The condition in the `if/else` statement tests whether your town's population is less than 10,000 via the < comparison operator. If the condition evaluates to true, then `message` is set to be equal to the first string literal ("X is a small town!"). If the condition evaluates to false – if the population is 10,000 or greater – the `message` is set to be equal to the second string literal ("X is pretty big!"). In this case, the town's population is less than 10,000, so `message` is set to "5422 is a small town!"

Table 3.1 lists Swift's comparison operators.

Table 3.1 Comparison operators

Operator	Description
<	Evaluates whether the number on the left is smaller than the number on the right.
<=	Evaluates whether the number on the left is smaller than or equal to the number on the right.
>	Evaluates whether the number on the left is greater than the number on the right.
>=	Evaluates whether the number on the left is greater than or equal to the number on the right.
==	Evaluates whether the number on the left is equal to the number on the right.
!=	Evaluates whether the number on the left is not equal to the number on the right.
===	Evaluates whether the two instances point to the same reference.
!==	Evaluates whether the two instances do not point to the same reference.

You do not need to understand all of the operators' descriptions right now. You will see many of them in action as you move through the book, and they will become clearer as you use them. Refer back to this table as a reference if you have questions.

Sometimes you only care about one aspect of the condition that is under evaluation. That is, you want to execute code if a certain condition is met and do nothing if it is not. Enter the code below. (Notice that new code, shown in bold, appears in two places.)

Listing 3.2 Is there a post office?

```
import Cocoa

var population: Int = 5422
var message: String
var hasPostOffice: Bool = true

if population < 10000 {
    message = "\(population) is a small town!"
} else {
    message = "\(population) is pretty big!"
}

print(message)

if !hasPostOffice {
    print("Where do we buy stamps?")
}
```

Here, you add a new variable called `hasPostOffice`. This variable has the type **`Bool`**, short for "Boolean." Boolean types can take one of two values: `true` or `false`. In this case, the Boolean `hasPostOffice` variable keeps track of whether the town has a post office. You set it to `true`, meaning that it does.

The `!` is called a *logical operator*. This operator is known as "logical not." It tests whether `hasPostOffice` is false. You can think of `!` as inverting a **`Boolean`** value: true becomes false, and false becomes true.

The code above first sets `hasPostOffice` to true, then asks whether it is false. If `hasPostOffice` is false, you do not know where to buy stamps, so you ask. If `hasPostOffice` is true, you know where to buy stamps and do not have to ask, so nothing happens.

Because the town *does* have a post office (because `hasPostOffice` was initialized to `true`), the condition `!hasPostOffice` is false. That is, it is *not* the case that `hasPostOffice` is false. Therefore, the **`print()`** function never gets called.

Table 3.2 lists Swift's logical operators.

Table 3.2 Logical operators

Operator	Description
&&	Logical AND: true if and only if both are true (false otherwise)
||	Logical OR: true if either is true (false only if both are false)
!	Logical NOT: true becomes false, false becomes true

Ternary Operator

The *ternary operator* is very similar to an `if/else` statement, but has more concise syntax. The syntax looks likes this: `a ? b : c`. In English, the ternary operator reads something like, "If `a` is true, then do `b`. Otherwise, do `c`."

Let's rewrite the town population check that used `if/else` using the ternary operator instead.

Listing 3.3 Using the ternary operator

```
...
if population < 10000 {
    message = "\(population) is a small town!"
} else {
    message = "\(population) is pretty big!"
}

message = population < 10000 ? "\(population) is a small town!" :
                    "\(population) is pretty big!"
...
```

The ternary operator can be a source of controversy: some programmers love it; some programmers loathe it. We come down somewhere in the middle. This particular usage is not very elegant. Your assignment to `message` requires more than a simple `a ? b : c`. The ternary operator is great for concise statements, but if your statement starts wrapping to the next line, we think you should use `if/else` instead.

Hit Command-Z to undo, removing the ternary operator and restoring your `if/else` statement.

Listing 3.4 Restoring `if/else`

```
...
message = population < 10000 ? "\(population) is a small town!" :
                    "\(population) is pretty big!"
if population < 10000 {
    message = "\(population) is a small town!"
} else {
    message = "\(population) is pretty big!"
}
...
```

Nested ifs

You can nest `if` statements for scenarios with more than two possibilities. You do this by writing an `if/else` statement inside the curly braces of another `if/else` statement. To see this, nest an `if/else` statement within the `else` block of your existing `if/else` statement.

Listing 3.5 Nesting conditionals

```
import Cocoa

var population: Int = 5422
var message: String
var hasPostOffice: Bool = true

if population < 10000 {
    message = "\(population) is a small town!"
} else {
    if population >= 10000 && population < 50000 {
        message = "\(population) is a medium town!"
    } else {
        message = "\(population) is pretty big!"
    }
}

print(message)

if !hasPostOffice {
    print("Where do we buy stamps?")
}
```

Your nested `if` clause makes use of the `>=` *comparator* (comparison operator) and the `&&` logical operator to check whether `population` is within the range of 10,000 to 50,000. Because your town's `population` does not fall within that range, your `message` is set to "5422 is a small town!" as before.

Try bumping up the population to exercise the other branches.

Nested `if/else` statements are common in programming. You will find them out in the wild, and you will be writing them as well. There is no limit to how deeply you can nest these statements. However, the danger of nesting them too deeply is that it makes the code harder to read. One or two levels are fine, but beyond that your code becomes less readable and maintainable.

There are ways to avoid nested statements. Next, you are going to *refactor* the code that you have just written to make it a little easier to follow. Refactoring means changing code so that it does the same work but in a different way. It may be more efficient, or may just look prettier or be easier to understand.

else if

The `else if` conditional lets you chain multiple conditional statements together. `else if` allows you to check against multiple cases and conditionally executes code depending on which clause evaluates to true. You can have as many `else if` clauses as you want. Only one condition will match.

To make your code a little easier to read, extract the nested `if/else` statement to be a standalone clause that evaluates whether your town is of medium size.

Listing 3.6 Using `else if`

```
import Cocoa

var population: Int = 5422
var message: String
var hasPostOffice: Bool = true

if population < 10000 {
    message = "\(population) is a small town!"
} else if population >= 10000 && population < 50000 {
    message = "\(population) is a medium town!"
}  else {
    if population >= 10000 && population < 50000 {
        message = "\(population) is a medium town!"
    } else {
        message = "\(population) is pretty big!"
    }
    message = "\(population) is pretty big!"
}

print(message)

if !hasPostOffice {
    print("Where do we buy stamps?")
}
```

You are using one `else if` clause, but you could have chained many more. This block of code is an improvement over the nested `if/else` above. If you find yourself with lots of `if/else` statements, you may want to use another mechanism, such as `switch` described in Chapter 5. Stay tuned.

Bronze Challenge

Add an additional `else if` statement to the town-sizing code to see if your town's `population` is very large. Choose your own population thresholds. Set the `message` variable accordingly.

This page intentionally left blank

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

T

U

V

W

X