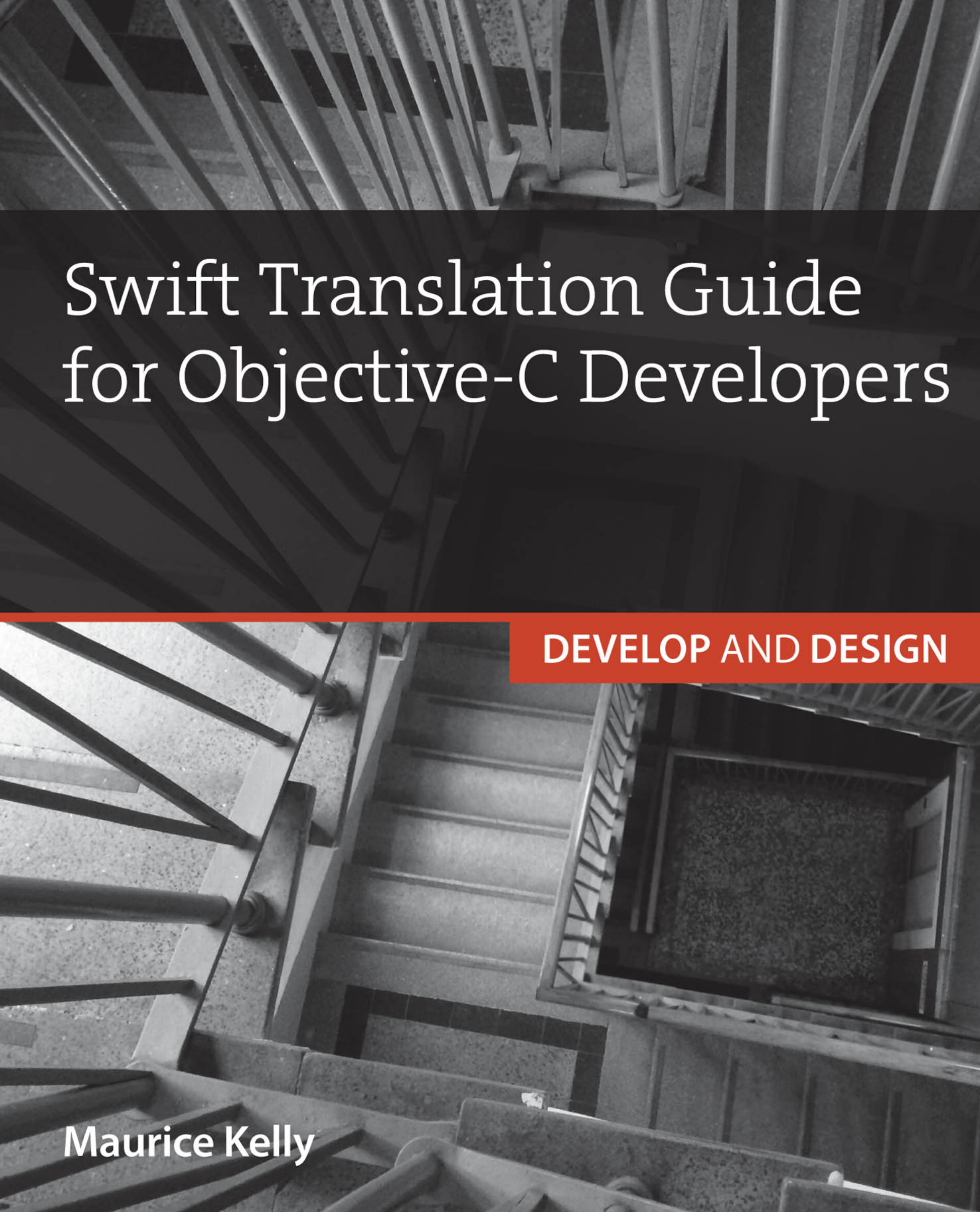


Swift Translation Guide for Objective-C Developers

DEVELOP AND **DESIGN**

Maurice Kelly

Swift Translation Guide for Objective-C Developers

DEVELOP AND DESIGN

Maurice Kelly

PEACHPIT PRESS
WWW.PEACHPIT.COM

Swift Translation Guide for Objective-C Developers: Develop and Design
Maurice Kelly

Peachpit Press

www.peachpit.com
To report errors, please send a note to errata@peachpit.com

Peachpit Press is a division of Pearson Education.

Copyright 2015 by Maurice Kelly

Editor: Robyn G. Thomas
Copyeditor: Darren Meiss
Proofreader: Nancy Bell
Technical editor: Mark Goody
Compositor: Danielle Foster
Indexer: Valerie Haynes Perry
Cover design: Aren Straiger
Interior design: Mimi Heft

ISBN-13: 978-0-134-04469-9
ISBN-10: 0-134-04469-X

9 8 7 6 5 4 3 2 1

Printed and bound in the United States of America

Once more, I couldn't have completed this project without the support, encouragement, and love of my wife, Fiona, and our wonderful girls, Aoibhínn and Caoimhe. "Mank ooo" for giving me lots of smiles and hugs every time I finished writing.

To my parents, brother, sister, and our extended families—thanks for looking interested when I talked about Swift, and for putting up with all the excuses that began with "But I'm working on the book..."

ACKNOWLEDGMENTS

I would like to thank the engineering management chain at my employers, ShopKeep Inc., particularly Stepan Hruda, Duncan Grazier, Jason Ordway, and Miriam Kerbache. They supported this endeavor by giving me the permission and space to work on it, and by giving me an enjoyable and progressive engineering environment to return to.

Mark Goody deserves thanks on many levels for his help with this book: As a friend, technical editor, and oft-inspirator, it feels wrong to boil it down to just one word, but all I can say is thanks.

It was a pleasure to work again with Robyn Thomas on this book. Robyn puts up with my poor language skills, despite the constant ribbing about U.S. English, and for that she is forgiven for her predilection toward *z*, and aversion to *u*. Thanks also to Clifford Colby for the wonderful lunch in San Francisco and for taking a chance on me with another book. I hope to buy you lunch in return sometime.

ABOUT THE AUTHOR

Maurice Kelly has been engineering software since leaving university in 2001. After spending many years working on server software for mobile telecoms, he took a change of direction to work at the user-facing end by becoming an iOS developer. He has a love for synthesizers and music, and dreams of owning a Land Rover Defender someday. He lives with his wife and children just outside Dromara, a small village in Northern Ireland.

CONTENTS

INTRODUCTION

When Apple introduced Swift at the Apple WWDC (Worldwide Developers Conference) in 2014, the audience, packed full of developers for the Apple platforms, was stunned and silent. Swift is the first truly new programming language to be introduced for the Mac development platform in its history; Objective-C, C++, and C were all existing languages. Swift was built from the ground up, by Apple and for Apple.

Apple described the new Swift language as "Objective-C without the C," but that doesn't really do it justice. Swift is more of a completely new programming language than simply Objective-C with the C heritage extracted.

A HISTORY

For many Apple-centric developers, especially those developing for the iOS platform, the introduction of a new language is a seismic change. Objective-C has been our go-to programming language for years, and it often feels like it has been the only language we have ever used. Yet it wasn't always that way...

While Objective-C has existed since the 1980s, it wasn't adopted for Mac OS development until after the acquisition of NeXT, a company founded by Steve Jobs in the mid-80s, by Apple in 1996. NeXT had selected Objective-C as the primary programming language for its NeXTSTEP operating system. NeXTSTEP was a major influence on the development of Mac OS X, and it was perhaps inevitable that Objective-C would come to be a dominant language in the OS X ecosystem.

Before the acquisition of NeXT, a number of different languages were popularly used to develop Mac applications. Early Mac developers commonly used Pascal to write their apps, and Apple would later introduce Object Pascal as an object-oriented extension of the language.

When Mac hardware switched from 68K to PowerPC, Apple took the opportunity to adopt C++ and rewrite its MacApp framework. Prior to the NeXT acquisition, most Mac applications were written in C++. The adoption of NeXTSTEP, and its Objective-C-based frameworks, as the foundation for Mac OS X would have been a serious bump in the road, because developers would have needed to re-implement their applications in Objective-C.

Thankfully, Apple introduced not just the Cocoa framework, which of course was based on Objective-C, but also a peer framework named Carbon that was programmed in C++. This offered existing developers a smoother path to porting their classic Mac OS apps to Mac OS X. Though Carbon has been deprecated for a number of years, it existed for quite some time, so it is fair to say that C++ was a very popular language for OS X development alongside Objective-C and not just before it. Even today, C++ is still available to use in Xcode as both a pure language and as Objective-C++. Maybe it is not surprising that in many ways Swift has many C++ "flavors" to it, such as overloading and generics.

Despite being introduced as "Objective-C without the C," Swift is still very much a member of the extended C family of languages. As a result, it should still feel somewhat familiar to developers coming from C, C++, and Java, and especially to developers coming from Objective-C.

HOW TO USE THIS BOOK

This is not a traditional technical book for a new programming language: This is a translation guide, with an express purpose of helping existing developers of OS X and iOS applications in Objective-C migrate their skill sets to the new language.

HOW YOU WILL LEARN

We begin by examining the core constructs of the Objective-C language, picking apart its syntax, and showing you how it differs from Swift. The book focuses mainly on short samples of code from both languages, illustrating how the syntax of Objective-C translates to Swift. The languages are interspersed, and you'll find labels in the margins to help you quickly distinguish between Objective-C and Swift code.

Objective-C

```
[greetingMaker produceGreeting:^NSString *(NSString *format, NSString *name) {
    return [NSString stringWithFormat:format, name];
}];
```

Swift

```
greetingMaker.produceGreeting( { (format: String, name: String) -> (String) in
    return NSString(format:format, name)
} )
```

You'll also find notes containing additional information about the topics.

> **NOTE:** Properties must always be declared as `var` in the protocol, even if you do not intend to use them as a variable in the conforming type. When you implement a protocol, you are free to redefine the property as a constant if you wish.

Swift and Xcode 6 present a completely new way to experiment with code in the form of Swift playgrounds. The examples in this book are as short as possible so you can type them into a Swift playground in order to see the results, and to modify the code for the purposes of experimentation. Most of the code examples are collated into Swift playground files and available for download at the website to accompany the book at http://swift-translation.guide.

WHAT YOU WILL LEARN

This book takes the existing syntax, constructs, and patterns from Objective-C and shows how they can be translated into Swift. We'll strive to point out potential pitfalls along the way, highlight shortcomings of the new language, and illustrate new ways of doing things that were never before possible in Objective-C.

This book assumes that you are familiar with Apple's Foundation, Cocoa, and/or CocoaTouch frameworks. Although the programming language may be changing, these frameworks make developing for OS X and iOS such a rich experience and will largely remain the same.

At the time of this writing, Swift was at version 1.1 but the language is evolving rapidly. If you have any problems with the code samples, please look in the online documentation for the standard library (bit.ly/apple-swift-docs) to check that the syntax is actually the latest.

WELCOME TO SWIFT

You most likely have the tools you need already installed to develop your iOS or OS X apps in Swift rather than Objective-C. For iOS apps, you need at least Xcode 6.0; for OS X development, you need at least Xcode 6.1. If you're not running the right version, upgrade using the Mac App Store, or download a version from the Apple Developer Center at http://developer.apple.com.

SWIFT'S GOALS

When introducing Swift to the world, Apple had three goals for the new language: It had to be safe, modern, and powerful.

SAFE

Swift added and removed programming concepts to make it a safer language than Objective-C. In the new additions column are optional variables, constant references, and numerous syntactical changes that reduce the scope for programming logic errors. Under removals, say goodbye to pointers and the ability to send messages to `nil` objects.

MODERN

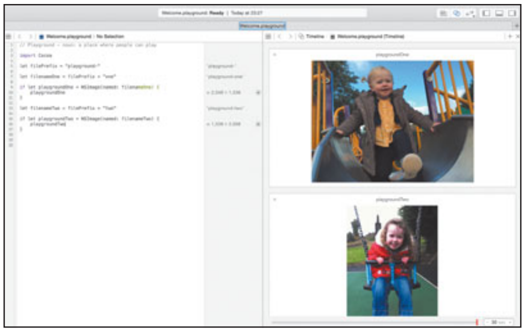
Swift is a decidedly more modern language than Objective-C. Functions are treated as first class types, generics provide a more extensible base for custom collections, and even subtle changes like type inference can leave Objective-C looking increasingly dated. Structures and enumerations have been overhauled, taking them from their humble C origins and making them an alternative to classes for many data needs.

POWERFUL

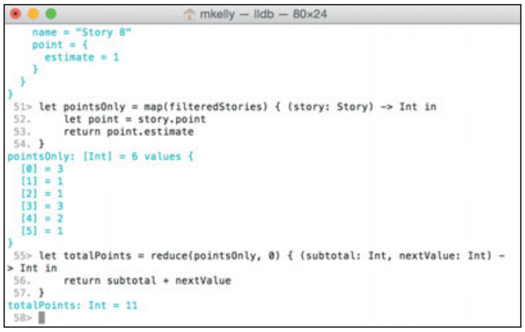
Apple has made bold claims about Swift's performance in comparison to Objective-C and other high-level languages, some of which appear to be starting to hold water in real-world tests. Swift has a powerful feature set and syntax, which allow iOS and OS X developers to do more than they ever have before.

NEW TOOLS

Xcode 6 comes with a pair of new tools to help you experiment with Swift: playgrounds and the REPL.



```
// Playground - noun: a place where people can play

import Cocoa

let filePrefix = "playground-"

let filenameOne = filePrefix + "one"

if let playgroundOne = NSImage(named: filenameOne) {
    playgroundOne
}

let filenameTwo = filePrefix + "two"

if let playgroundTwo = NSImage(named: filenameTwo) {
    playgroundTwo
}
```



```
    name = "Story 8"
    point = {
      estimate = 1
    }
  }
}
 51> let pointsOnly = map(filteredStories) { (story: Story) -> Int in
 52.     let point = story.point
 53.     return point.estimate
 54. }
pointsOnly: [Int] = 6 values {
  [0] = 3
  [1] = 1
  [2] = 1
  [3] = 3
  [4] = 2
  [5] = 1
}
 55> let totalPoints = reduce(pointsOnly, 0) { (subtotal: Int, nextValue: Int) -
> Int in
 56.     return subtotal + nextValue
 57. }
totalPoints: Int = 11
 58> 
```

PLAYGROUNDS

Swift playgrounds bring a new way of experimentation to Xcode.

Playgrounds are interactive sandbox documents in which you can experiment with Swift's new ideas without worrying about creating new projects or dirtying your existing projects. They can work with text, images, or even complex SpriteKit animations.

REPL

The Swift REPL allows interactive programming from a command-line interface.

A Read-Eval-Print-Loop (REPL) tool gives you access to a playground-like environment from the command line. They are a great way to test syntax or a new idea quickly, without the overhead of having to run Xcode.

CHAPTER 4

Control Structures

Working with data is difficult if you have to do it in a completely linear manner, which makes control structures a vital part of any programming language. On the surface, the control structures in Swift are the same as those in Objective-C, but once you start to dig a little deeper you will begin to find that they go beyond what was possible before in a number of ways.

GENERAL CHANGES

Like Objective-C, Swift has four main control structures:

- `for` and `for-in` loops
- `while` and `do-while` loops
- `if` conditional blocks
- `switch` conditional blocks

They behave in a similar manner to their Objective-C counterparts but with some key differences you should be aware of.

PARENTHESES ARE OPTIONAL

The parentheses that surround the conditionals of the `if`, `for`, and `while` structures are no longer mandatory. Similarly the parentheses that surround the `switch` expression are also no longer mandatory.

This `for` loop:

Objective-C

```
for (var i = 0; i < 3; i++) { ... }
```

can now be written with less punctuation as:

Swift

```
for var i = 0; i < 3; i++ { ... }
```

BRACES ARE MANDATORY

As if to provide a counterbalance, the Swift developers have made curly braces mandatory for control structure blocks that contain a single line of code. An `if` statement in Objective-C that takes advantage of this shortcut:

Objective-C

```
if (number < 0)
    NSLog(@"Negative number");
else if (number > 0)
    NSLog(@"Positive number");
else
    NSLog(@"Zero");
```

has to be fully fleshed out in Swift:

Swift

```
if number < 0 {
    println("Negative number")
} else if number > 0 {
    println("Positive number")
} else {
    println("Zero")
}
```

Enforcing the use of braces may be a source of annoyance to developers who like to keep single-line if statements compact, but should result in safer code for many others.

BOOLEAN CONDITIONS

The if conditional in C (and thus Objective-C) behaves in a manner summarized very simply: If the supplied expression evaluates to true then the if block is executed. Where things get complicated is in the number of ways an expression can evaluate to true. The following list shows the variety of positive evaluations:

- A Boolean expression that evaluates to true
- A Boolean expression that evaluates to YES
- A numerical value that is nonzero (positive or negative)
- An object pointer that is not nil
- A pointer that is not NULL

In Objective-C, this allows us to take some convenient shortcuts. For example, testing for nonzero numbers is a common pattern:

Objective-C

```
if (integerValue) {
    //  Perform an action if not zero
}
```

Swift relies heavily on protocols to define behavior, and one such protocol is BooleanType. An expression can be used only as a conditional if it conforms to this protocol, and even the humble Int does not conform by default. To use a conditional, you just need to be more specific about what exactly it is you are testing:

Swift

```
if integerValue != 0 {
    //  Perform an action if not zero
}
```

This may seem like a pain at first, but is actually a useful way to learn some good programming practices. When we relied on the fact that a nonzero number equated to true, we also assumed that the reader of the code knew our exact meaning. This is acceptable with precise numbers, but could easily get confusing when working with enums or any other types where an assumption is being made that the reader knows the underlying values. The Swift core principle of safety is the likely motive behind these changes; making conditionals more explicit means less likelihood of using them incorrectly and getting unexpected results.

Now that we've covered the general differences, let's look into the four control structures in detail.

LOOPS

As with Objective-C, Swift has two flavors of loops—`for` and `while`—each of which has its own two varieties. The `for` statement supports the traditional C-style `for` statement with initializer, conditional, and increment expressions, and the more modern `for-in` style that works with collections and other iterable constructs. The `while` statement supports condition-block execution, and the `do-while` supports block-condition execution.

FOR AND FOR-IN

The traditional `for` loop works exactly as you would expect. The general format is still `for` ***`initializer`***`;` ***`conditional`***`;` ***`increment`*** with the only differences being in how the initializer and conditional are expressed:

- You must define variables using the `var` keyword; you can't subsequently increment a constant.
- You can rely on type inference to set the correct type for newly defined variables.
- You must use a conditional that conforms to the `BooleanType` protocol.

The `for-in`-style of loop has seen more in the way of changes with Swift. `for-in` was introduced with *fast enumeration* in Objective-C and is available to any classes that implement the `NSFastEnumeration` protocol. Objective-C developers most frequently use `for-in` loops with `NSArray`, `NSSet`, and `NSDictionary` collections, though any `NSEnumerator`-based class will also conform to `NSFastEnumeration`.

In Swift, the counterpart to the `NSFastEnumeration` protocol is `SequenceType`, and so anything that conforms to `SequenceType` is iterable. As you would expect, the default collections (array and dictionary) do so, but a few other types also implement the protocol:

- Foundation collection types—Apple has performed the requisite magic to allow the collection types you know and love to behave as iterable collections in Swift.
- Strings—A string is, after all, a collection of characters, and so the Swift String type supports iterating over the characters in a string.
- Half-open and closed ranges—Both types of ranges appear to the `for-in` loop as a collection of integers allowing it to iterate over the items in the collection.
- `strides`—The Swift `stride` function can be used to produce more complex ranges, including configurable increments and descending ranges.

Being able to use ranges with a `for-in` loop means fewer reasons than ever before to use the traditional `for` loop. Where you once used the following to perform a block of actions 10 times:

Objective-C

```
for var i = 1; i <= 10; i++ { ... }
```

you can now use a range with the `for-in` variant instead:

Swift

```
for i in 1...10 { ... }
```

Note how this improves readability; there is less chance of off-by-one errors, and if you're the type of developer who sweats the naming of your variables, you can even use a wildcard expression to show that you don't care what the variable is:

Swift

```
for _ in 1...10 { ... }
```

If you want to make your own object types iterable, you need to implement the `SequenceType` protocol. We will look at protocols in more detail in Chapter 13.

WHILE AND DO-WHILE

There is not much difference between `while` and `do-while` loops in Objective-C and Swift. Aside from the general changes described earlier (regarding conditions, parentheses, and braces), the only other difference you are likely to encounter is in the use of optional binding, which can be used as an alternative to using a `!=` `nil` test in a conditional expression.

For example, say you wanted to travel up a `UIView` hierarchy to determine the topmost view—indicated by the `superview` property being `nil`. In Objective-C, you would do the following:

Objective-C

```
UIView *currentView = aView;

while (currentView.superview != nil) {
    currentView = currentView.superview;
}
```

Using optional binding in Swift, you can instead use this code:

Swift

```
var possibleView: UIView? = aView
while let actualView = possibleView?.superview {
    possibleView = actualView
}
```

Don't worry if the excessive question marks seem a bit strange—we introduce optionals and optional binding in Chapter 5.

CONDITIONALS

Conditionals in Swift also come in the same two types as Objective-C: the largely unchanged if statement and the radically overhauled switch statement.

IF

Like while loops, the if conditional statement in Swift has mainly the same behaviors as its counterpart in Objective-C. As with the other control statements, it is subject to the changes in parentheses and braces, and again requires the conditional expression to conform to the BooleanType. Also like the while statement, if can be used with optional binding to guard against optional variables whose current value is nil.

SWITCH

Of all the flow control statements, the humble switch has probably changed the most. In Objective-C, describing switch as an alternative way of writing a series of if statements would be fair. For example, consider this sequence of if statements:

Objective-C

```
if (i == 1) {
    //  React to i == 1
} else if (i == 2) {
    //  React to i == 2
} else if (i == 3) {
    //  React to i == 3
} else {
    //  Handle every other case
}
```

The alternative in terms of the switch statement is:

Objective-C

```
switch (i) {
    case 1:
        //  React to i == 1
        break;
    case 2:
        //  React to i == 2
        break;
    case 3:
        //  React to i == 3
        break;
    default:
        //  Handle every other case
}
```

Two major changes to the way switch works in Swift have greatly enhanced its powers: an ability to switch on more than just integer values, and the ability to employ pattern-matching techniques. Swift also brings a number of smaller changes that make switch statements a little less error-prone than before.

TO INTEGERS AND BEYOND!

While switch statements have always been useful for dealing with integers, if you couldn't boil your data down to numbers, you had to move back to clunkier if constructs, which has long been a source of frustration for developers.

STRINGS

Quite possibly one of the most sought after capabilities is that of being able to match against strings. A common pattern in Objective-C is:

Objective-C

```
if ([stringValue isEqualToString:@"MatchA"]) {
    //  Handle for "MatchA"
} else if ([stringValue isEqualToString:@"MatchB"]) {
    //  Handle for "MatchB"
} else {
    //  Handle for all other possibilities
}
```

Using Swift, this boils down to:

Swift

```
switch stringValue {
    case "MatchA":
        //  Handle for MatchA
    case "MatchB":
        //  Handle for MatchB
    default:
        //  Handle for all other possibilities
}
```

While the length of the two alternatives is not significantly different, the need to state the stringValue variable just once makes refactoring easier and safer, and the Swift code block is arguably easier to comprehend while scanning the code.

ENUMERATIONS

Of course in Objective-C, switching based on the value of an enum is an integral part of development; after all, an Objective-C enum is nothing more than a "coded" integer, and all switch understands is integers.

However, over in the enlightened world of Swift, enumerations are now an object type, and thus no longer have to be plain old integers. We will look at enumerations in detail in Chapter 10, but for now all you need to know is that the underlying value can be as simple as an integer or as complex as a structure, an object, or even nothing at all! Thankfully Swift has you covered.

iOS developers who have worked with table views will be familiar with the UITableViewCellAccessoryType enumeration. If you wanted to check the value of the cell accessory type and perform different behaviors based on the value, you can do the following:

Swift

```
var cell = UITableViewCell()
switch cell.accessoryType {
    case UITableViewCellAccessoryType.None: println("None")
    case UITableViewCellAccessoryType.DisclosureIndicator:
    → println("Disclosure Indicator")
    case UITableViewCellAccessoryType.DetailDisclosureButton:
    → println("Disclosure Button")
    case UITableViewCellAccessoryType.Checkmark: println("Checkmark")
    case UITableViewCellAccessoryType.DetailButton: println("Detail Button")
}
```

RANGES

One area in Objective-C where the if statement had the advantage over switch was number ranges. If you wanted to execute different code based on more than individual values, if was your only option. For example:

Objective-C

```
NSString *grade;
NSUInteger testScore = getTestScore();
if (testScore >= 0 && testScore < 40) {
    grade = @"F";
} else if (testScore >= 40 && testScore < 60) {
    grade = @"C";
} else if (testScore >= 60 && testScore < 80) {
    grade = @"B";
} else {
    grade = @"A";
}
```

Earlier in this chapter, we looked at the concept of number ranges and in particular the half-open and closed ranges that Swift gives us. Using ranges and a switch statement you could easily reimplement this in Swift as:

Swift

```
var testScore = getTestScore()
var grade = ""
switch testScore {
    case 0..<40: grade = "F"
    case 40..<60: grade = "C"
    case 60..<80: grade = "B"
    default: grade = "A"
}
```

You can also use switch statements with Swift's other primitive types, including floating point numbers, optionals, and even object types. If you are in doubt about using a type with a switch statement, just create a new playground and try it out. And if you're having no luck, there's always pattern matching.

PATTERN MATCHING

In Chapter 3, we briefly covered the pattern match operator (~=) and specifically mentioned that it could be used to great effect with switch statements. The pattern match operator is even implicitly added to your conditional statement. There are a number of different pattern types you can match with, and you've already looked at ranges in the previous section. Here, we will examine some more types, including tuples and wildcards, as well as looking at where clauses and value binding.

TUPLES

As you learned in Chapter 3, a tuple is a simple set of ordered data such as a pair of coordinates, a list of places, or a sequence of numbers. Because tuples do not need to be defined in advance, you can assemble them from disparate pieces of data to make complex logic decisions. As an example, consider the following piece of code for customizing UITableViewCells based on their position in the table view:

Objective-C

```
if (indexPath.section == 0) {
    //  Configuring all cells in section 0 the same way
} else if (indexPath.section == 1) {
    if (indexPath.row == 0) {
        //  Configuring cell in row 0 distinct from the rest of section 1
    } else {
        //  Configuring remaining cells in section 1 the same way
    }
} else if (indexPath.section == 2) {
    if (indexPath.row == 3 || indexPath.row == 4 || indexPath.row == 5) {
        //  Configuring cells in rows 3, 4, and 5 distinct from rest of section 2
    } else {
        //  Configuring remaining cells in section 2 the same way
    }
} else {
    if (indexPath.row == 0) {
        //  Configuring row 0 in any other sections distinct from the rest of
            → the section
    } else {
        //  Configuring remaining cells in any other sections the same way
    }
}
```

At the minute, it isn't too hard to follow, but with time and an expanding feature set, this set of nested `if` statements will start to become harder to read, debug, and maintain. You could reimplement this as a flatter structure using Swift's ability to match on tuples:

Swift

```
switch (indexPath.section, indexPath.row) {
    case (0, 0):
        println("Configuring one row in section 0")
    case (1, 0):
        println("Configuring cell in row 0 distinct from the rest of section 1")
    case (1, 1):
        println("Configuring cell in row 1 distinct from the rest of section 1")
    case (2, 0):
        println("Configuring general cell in section 2")
    case (2, 1):
        println("Configuring general cell in section 2")
    case (2, 2):
        println("Configuring general cell in section 2")
    case (2, 3):
        println("Configuring cells in rows 3, 4, and 5 distinct from rest of
        → section 2")
    case (2, 4):
        println("Configuring cells in rows 3, 4, and 5 distinct from rest of
        → section 2")
    case (2, 5):
        println("Configuring cells in rows 3, 4, and 5 distinct from rest of
        → section 2")
    case (2, 6):
        println("Configuring general cell in section 2")
    case (3, 0):
        println("Configuring cell in section 3")
    case (4, 0):
        println("Configuring cell in section 3")
}
```

This solution has the advantage of being visually flatter. Unfortunately, it isn't as maintainable; the addition of new rows and sections will require constant updates. The conditional logic of the `if` statements, and particularly the `else` blocks, are sorely missing. Luckily there is an answer in the form of wildcards.

WILDCARDS

You first encountered the wildcard pattern very briefly in the previous section on `for` loops where you used a wildcard to discard the values you were iterating over. The wildcard pattern can also be used with tuples and case statements to match an entire series of entries with one statement. The wildcard operator (_) can be used in any position in the tuples supplied to a case statement, meaning you can match logical groupings like table sections.

Swift

```
switch (indexPath.section, indexPath.row) {
    case (0, _):
        println("Configuring section 0")
    case (1, 0):
        println("Configuring cell in row 0 distinct from the rest of section 1")
    case (1, _):
        println("Configuring remaining cells in section 1 the same way")
    case (2, 3):
        println("Configuring cells in rows 3, 4, and 5 distinct from rest of
        → section 2")
    case (2, 4):
        println("Configuring cells in rows 3, 4, and 5 distinct from rest of
        → section 2")
    case (2, 5):
        println("Configuring cells in rows 3, 4, and 5 distinct from rest of
        → section 2")
    case (2, _):
        println("Configuring remaining cells in section 2 the same way")
    case (_, 0):
        println("Configuring row 0 in any other sections distinct from the rest
        → of the section")
    case (_, _):
        println("Configuring remaining cells in any other sections the same way")
}
```

When using the wildcard operator, bear in mind that if multiple case statements result in a match, the first case is the selected match; if we had placed the last entry (with the double wildcard) anywhere else, it would have a negative impact on the logic. Try to order your case statements from most to least specific, regardless of what type of pattern matching you use!

While this code is more readable, an unfortunate bit of duplication exists in the handling of rows 3, 4, and 5 in section 2. Swift allows you to use ranges in combination with tuples and wildcards. You can refactor the code to a more manageable version by using a range for handling rows 3, 4, and 5. The highlighted line in the following code replaces three case statements:

Swift

```
switch (indexPath.section, indexPath.row) {
    case (0, _):
        println("Configuring section 0")
    case (1, 0):
        println("Configuring cell in row 0 distinct from the rest of section 1")
    case (1, _):
        println("Configuring remaining cells in section 1 the same way")
    case (2, 3...5):
        println("Configuring cells in rows 3, 4, and 5 distinct from rest of
        → section 2")
    case (2, _):
        println("Configuring remaining cells in section 2 the same way")
    case (_, 0):
        println("Configuring row 0 in any other sections distinct from the rest
        → of the section")
    case (_, _):
        println("Configuring remaining cells in any other sections the same way")
}
```

VALUE BINDINGS

Within conditional statements you often want to have access to the values used to make the decisions. Continuing with the table view cell customization example, you might actually want to know the row and section numbers handled by the wildcard patterns in the last case, the use case being that you want to display the number of the row and section in the cell.

Of course, these values are already available in the form of `indexPath.row` and `indexPath.section`, but Swift and UIKit still use the concept of zero-based indexes, which isn't conducive to a good user experience. Creating new values based on the `indexPath` as part of the case statement block is possible, but you can also use a feature of Swift called *value bindings*; these allow values from the case to be bound to temporary variables or narrowly scoped constants.

You can now rewrite the last case statement to take advantage of value bindings:

```
switch (indexPath.section, indexPath.row) {
    case (0, _):
        println("Configuring section 0")
    case (1, 0):
        println("Configuring cell in row 0 distinct from the rest of section 1")
    case (1, _):
        println("Configuring remaining cells in section 1 the same way")
    case (2, 3...5):
        println("Configuring cells in rows 3, 4, and 5 distinct from rest of
        → section 2")
    case (2, _):
        println("Configuring remaining cells in section 2 the same way")
    case (_, 0):
        println("Configuring row 0 in any other sections distinct from the rest
        → of the section")
    case (var section, let row):
        section++
        println("Configuring cell \(row + 1) in section \(section)")
}
```

By assigning to a variable, you are able to modify the section without worrying about changing the original tuple passed to the `switch` statement.

Note that the wildcard operators are no longer included in the case statement, but it still retains the same behavior. Think of the wildcard operator as a throwaway variable you do not care about; the value bindings, on the other hand, signify that you are interested in the values they contain.

WHERE CLAUSES

As if using tuples, ranges, and even wildcards didn't give enough flexibility already, Swift adds even more capability by providing the facility to include `where` clauses in the `case` statement.

Another common use case when working with table views is the need to place some sort of highlight on a cell to indicate state, often through the use of color or a checkmark disclosure indicator. In our seemingly never-ending example, the sections of the table covered by the last `case` statement could have a highlight if their index paths are in a separately maintained array of index paths.

You could achieve this using nested conditional statements within the last `case`. Alternatively, you can make a copy of the `case` and use a `where` clause on the first one; remember, you want the `case` statements to go from most to least specific. A `where` clause takes a regular conditional expression, and it can use variables and constants from outside the `switch` statement as well as value bindings from within the case statement itself.

Using a where clause, the code now becomes:

Swift

```
switch (indexPath.section, indexPath.row) {
    case (0, _):
        println("Configuring section 0")
    case (1, 0):
        println("Configuring cell in row 0 distinct from the rest of section 1")
    case (1, _):
        println("Configuring remaining cells in section 1 the same way")
    case (2, 3...5):
        println("Configuring cells in rows 3, 4, and 5 distinct from rest of
        → section 2")
    case (2, _):
        println("Configuring remaining cells in section 2 the same way")
    case (_, 0):
        println("Configuring row 0 in any other sections distinct from the rest
        → of the section")
    case (var section, let row) where contains(highlightedIndexPaths, indexPath):
        section++
        println("Configuring highlighted cell \(row + 1) in section \(section)")
    case (var section, let row):
        section++
        println("Configuring cell \(row + 1) in section \(section)")
}
```

CUSTOM PATTERN MATCHING

There will always be a use case the developers of Swift cannot foresee, and so they have provided the ability to create custom pattern matching behavior by overloading the pattern match (~=) operator. If you wanted to compare a `String` object to an `Int`, or one custom type to another, you will need to supply your own way of comparing the two types. For more information on operator overloading, see Chapter 14.

SAFETY FEATURES

The improvements to `switch` are not just about usability; the efforts to make Swift a safer programming language have extended to the `switch` statement as well.

- No fall through by default: This is a fundamental change between the Objective-C and Swift behaviors. If you look at any of the examples in this chapter, you'll notice that none of them feature a break statement. Unlike C and Objective-C, Swift does not allow code to fall through from one case statement to another by default: You have to explicitly request this behavior by adding the `fallthrough` keyword to the end of your `case` block. C and

Objective-C developers are often bitten by the accidental omission of a break statement, but this rarely caused an actual error—just time spent debugging strange data problems.

- Multiple conditions can apply to a single case: Now that relying on automatic fall through isn't possible, multiple case conditions can be grouped on one or more lines when separated by commas. If you wanted to match on the values 1, 3, and 5, you can use `case 1, 3, 5:`. This is more efficient than putting successive lines doing something like the following code sample. This sample is legal but dangerous in Swift—accidentally removing the fallthrough keyword on one line may break a number of conditions.

```
case 1:
    fallthrough
case 3:
    fallthrough
case 5:
    //  Action
```

Swift

- The case list must be exhaustive: In Swift, writing a switch statement where none of the supplied case statements are a match isn't possible. If the compiler detects that such a condition has arisen, it reports an error. You can avoid this scenario by ensuring that you create case statements for every possible value, or set of values, by using wildcards or by including a default case.

WRAPPING UP

Given how important control statements are to programming, the major changes in Swift, especially to the switch statement, aren't that surprising. Most of the changes are for the better, and while some may cause a few "style guide" conflicts in the short term, the added safety and convenience in some of the constructs are very welcome.

The next chapter takes a look at optionals—a new feature that has the potential to change some of the code patterns we take for granted and make for more compact and readable source code.

INDEX