



# Swift Translation Guide for Objective-C Developers

**DEVELOP AND DESIGN**

**Maurice Kelly**

# 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](http://www.peachpit.com)

To report errors, please send a note to [errata@peachpit.com](mailto: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

### **Notice of Rights**

All rights reserved. No part of this book may be reproduced or transmitted in any form by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. For information on getting permission for reprints and excerpts, contact [permissions@peachpit.com](mailto:permissions@peachpit.com).

### **Notice of Liability**

The information in this book is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of the book, neither the author nor Peachpit shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the instructions contained in this book or by the computer software and hardware products described in it.

### **Trademarks**

Apple, Cocoa, Cocoa Touch, Objective-C, OS X, and Xcode are registered trademarks of Apple Inc., registered in the U.S. and other countries. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Peachpit was aware of a trademark claim, the designations appear as requested by the owner of the trademark. All other product names and services identified throughout this book are used in editorial fashion only and for the benefit of such companies with no intention of infringement of the trademark. No such use, or the use of any trade name, is intended to convey endorsement or other affiliation with this book.

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, Aoibhinn 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 Hruđa, Duncan Grazier, Jason Ordway, and Miriam Kerbach. 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

	A History .....	xii
	Welcome to Swift .....	xiv
<b>CHAPTER 1</b>	<b>CREATING SWIFT PROJECTS .....</b>	<b>2</b>
	Your First Swift Project .....	4
	Fundamental Differences .....	5
	<i>No More Header Files</i> .....	5
	<i>Main Is Missing</i> .....	5
	<i>Semicolons?</i> .....	6
	Using Swift and Objective-C Together .....	6
	Wrapping Up .....	7
<b>CHAPTER 2</b>	<b>PLAYGROUNDS AND THE SWIFT REPL .....</b>	<b>8</b>
	Swift Playgrounds .....	10
	<i>A Quick Tour of Swift Playgrounds</i> .....	10
	<i>What Can You Do with Playgrounds?</i> .....	14
	<i>Creating Your Own Playgrounds</i> .....	14
	The Swift REPL .....	15
	<i>How to Use the REPL</i> .....	15
	<i>Possibilities</i> .....	16
	<i>Limitations</i> .....	17
	Wrapping Up .....	17
<b>CHAPTER 3</b>	<b>LANGUAGE BASICS .....</b>	<b>18</b>
	Types .....	20
	<i>Primitive Types</i> .....	20
	<i>Strings</i> .....	22
	<i>Functions</i> .....	22
	<i>Blocks</i> .....	22
	<i>Tuples</i> .....	22
	<i>Classes, Structs, and Enumerations</i> .....	23
	<i>Collections</i> .....	24
	Variable Declarations .....	24
	<i>var and let</i> .....	24
	<i>Mutability</i> .....	25
	<i>Type Inference</i> .....	26
	Syntax .....	27
	<i>Semicolons</i> .....	27
	<i>Where Is @?</i> .....	27
	<i>No Longer Hip to Be Square</i> .....	28

	<i>Control Blocks</i>	28
	<i>The Post-Preprocessor Age</i>	29
	<i>Comments</i>	30
	<i>Operators</i>	31
	<i>Identical</i>	32
	<i>Different</i>	32
	<i>New</i>	35
	<i>Tuples</i>	38
	<i>Creating and Using Tuples</i>	38
	<i>Reusing Tuples</i>	39
	<i>Wrapping Up</i>	39
<b>CHAPTER 4</b>	<b>CONTROL STRUCTURES</b>	<b>40</b>
	<i>General Changes</i>	42
	<i>Parentheses Are Optional</i>	42
	<i>Braces Are Mandatory</i>	42
	<i>Boolean Conditions</i>	43
	<i>Loops</i>	44
	<i>for and for-in</i>	44
	<i>while and do-while</i>	45
	<i>Conditionals</i>	46
	<i>if</i>	46
	<i>switch</i>	46
	<i>To Integers and Beyond!</i>	47
	<i>Strings</i>	47
	<i>Enumerations</i>	48
	<i>Ranges</i>	48
	<i>Pattern Matching</i>	49
	<i>Safety Features</i>	54
	<i>Wrapping Up</i>	55
<b>CHAPTER 5</b>	<b>OPTIONALS</b>	<b>56</b>
	<i>Why Do We Need Optionals?</i>	58
	<i>Declaring an Optional</i>	58
	<i>Using an Optional</i>	59
	<i>Unwrapping</i>	59
	<i>Optional Binding</i>	60
	<i>Implicit Unwrapping</i>	61
	<i>Optional Chaining</i>	61
	<i>Things to Watch Out For</i>	62
	<i>nil Coalescing Operator</i>	63
	<i>No Need to Unwrap</i>	63
	<i>Wrapping Up</i>	63



<b>CHAPTER 6</b>	<b>FUNCTIONS</b>	<b>64</b>
	Calling Functions	66
	Defining Functions	67
	Parameter Naming	67
	Default Parameter Values	69
	Return Values	70
	Modifying Function Parameters	71
	Variadic Parameters	73
	Function Scopes	75
	Global Scope	75
	Scope of Methods	75
	Nested Functions	75
	Using Functions	76
	Function Types	76
	Using Functions as Parameter Values	78
	Using Functions as Return Values	78
	Wrapping Up	79
<b>CHAPTER 7</b>	<b>BLOCKS AND CLOSURES</b>	<b>80</b>
	Defining Closures	82
	Creating the Definition	82
	Assigning to Variables	83
	Receiving as Function Parameters	83
	Creating Type Aliases	84
	Executing Closures	84
	Optimizing Closures	85
	Implicit Returns	85
	Type Inference	85
	Shorthand Argument Names	85
	Trailing Closure Syntax	86
	Operator Functions	86
	Capturing Values	87
	Wrapping Up	87
<b>CHAPTER 8</b>	<b>STRINGS</b>	<b>88</b>
	Constructing Strings	90
	Manipulating Strings	91
	Examining String Values	91
	Comparing Strings	92
	Modifying String Content	93
	Interacting with NSString	95
	Substrings	95
	Conversions	96

	<i>Path Modification and URL Methods</i> .....	96
	<i>Explicitly Creating an NSString</i> .....	96
	Unicode .....	97
	Wrapping Up .....	98
<b>CHAPTER 9</b>	<b>CLASSES</b> .....	<b>100</b>
	Creating Classes .....	102
	Methods .....	102
	Properties .....	103
	<i>Stored Properties</i> .....	103
	<i>Read-only Properties</i> .....	107
	<i>Think About Your Self</i> .....	108
	Initializers .....	108
	<i>Creating Initializers</i> .....	109
	<i>Using Convenience and Designated Initializers</i> .....	109
	<i>Deinitializing</i> .....	110
	Inheritance .....	111
	<i>Overriding a Method</i> .....	111
	<i>Calling to the Superclass</i> .....	112
	Access Control .....	112
	<i>Private</i> .....	113
	<i>Internal</i> .....	114
	<i>Public</i> .....	114
	Subscripting .....	114
	Wrapping Up .....	115
<b>CHAPTER 10</b>	<b>STRUCTURES AND ENUMERATIONS</b> .....	<b>116</b>
	Structures .....	118
	<i>Defining a Structure</i> .....	118
	<i>Modifying a Structure</i> .....	120
	Enumerations .....	122
	<i>Creating Enumerations</i> .....	122
	<i>Using Enumerations</i> .....	123
	<i>Associated Values</i> .....	124
	<i>Methods and Computed Properties</i> .....	126
	Nesting Types .....	126
	Wrapping Up .....	127
<b>CHAPTER 11</b>	<b>MEMORY MANAGEMENT</b> .....	<b>128</b>
	Swift Memory Management .....	130
	When to Use Weak and Unowned References .....	131
	Closures .....	132
	Wrapping Up .....	133

<b>CHAPTER 12</b>	<b>COLLECTIONS</b>	<b>134</b>
	General Comparisons	136
	Arrays	136
	<i>Creating Arrays</i>	137
	<i>Reading Array Contents</i>	137
	<i>Manipulating Arrays</i>	138
	Dictionaries	141
	<i>Creating Dictionaries</i>	141
	<i>Reading from Dictionaries</i>	142
	<i>Manipulating Dictionaries</i>	143
	Mutability	144
	Wrapping Up	145
<b>CHAPTER 13</b>	<b>PROTOCOLS, EXTENSIONS, AND GENERICS</b>	<b>146</b>
	Protocols	148
	<i>Creating</i>	148
	<i>Conforming to Protocols</i>	150
	Extensions	151
	Generics	152
	<i>Generic Functions</i>	152
	<i>Generic Types</i>	153
	Wrapping Up	154
<b>CHAPTER 14</b>	<b>THERE ISN'T A WORD FOR THAT</b>	<b>156</b>
	Only in Objective-C	158
	<i>Exception Handling</i>	158
	KVO	158
	<i>Reflection</i>	159
	<i>Dynamic Dispatch</i>	159
	New to Swift	160
	<i>Namespaces</i>	160
	<i>Custom Operators</i>	160
	<i>filter, map, and reduce</i>	161
	Wrapping Up	163
<b>CHAPTER 15</b>	<b>INTERACTING WITH OBJECTIVE-C</b>	<b>164</b>
	Introducing Swift to Objective-C	166
	<i>Adding Swift Files</i>	166
	<i>Using Swift Code in Objective-C</i>	167
	Using Objective-C with Swift	168
	<i>Adding Objective-C Files</i>	168

<i>Syntax Changes</i> .....	169
<i>Type Changes</i> .....	169
<i>Working with C Code</i> .....	170
Frameworks .....	171
Wrapping Up .....	171
Index .....	172

# 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.

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

---

**Objective-C**

---

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

---

**Swift**

---

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](http://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.

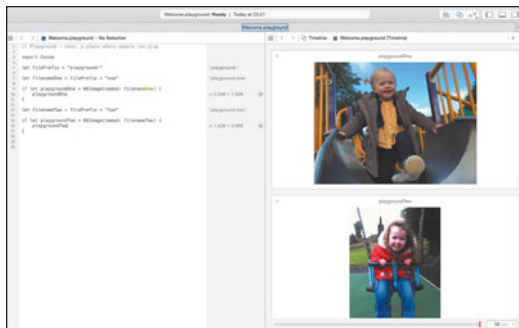


### POWERFUL

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

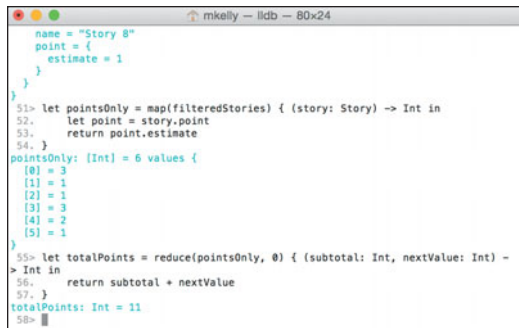
Xcode 6 comes with a pair of new tools to help you experiment with Swift: playgrounds and the REPL.



### 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:

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

---

Objective-C

---

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:

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

---

Swift

---

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:

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

---

**Swift**

---

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:

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

---

**Swift**

---

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 top-most view—indicated by the `Superview` property being `nil`. In Objective-C, you would do the following:

```
UIView *currentView = aView;
```

---

**Objective-C**

---

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

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

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

---

**Swift**

---

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:

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

---

**Objective-C**

---

Using Swift, this boils down to:

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

---

**Swift**

---

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:

```
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"
}
```

---

**Swift**

---

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 `UITableViewCell`s based on their position in the table view:

```
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) {
```

---

**Objective-C**

---

```

    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.

```

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")
}

```

---

Swift

---

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)")  
}
```

---

Swift

---

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
```

- 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.

---

Swift

---

## 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

## SYMBOLS

~ operator, 35  
- operator, 32  
-- operator, 32  
.. operator, 35  
/ operator, 32  
| | operator, 32–33  
| operator, 32  
+ operator, 32  
~= operator, 35  
-= operator, 32  
= operator, 32  
== operator, 33  
=== operator, 35  
. operator, 33  
! operator, 32–33, 35  
% operator, 32  
& operator, 32–34  
\* operator, 32  
? operator, 33, 35  
[ ] operator, 35  
^ operator, 32  
< operator, 32  
> operator, 32  
@ (at) sign, absence of, 27–28  
{ } (curly braces), requirement of, 42–43  
( ) (parentheses)  
    omission of, 29, 42  
    using with functions, 66, 76–77  
; (semicolon), absence of, 6, 27  
[ ] (square) brackets, use of, 28

## A

access control  
    internal, 114  
    private, 113  
    public, 114  
addition  
    and assignment operator, 32  
    operator, 32  
    with overflow operator, 33  
AnyObject primitive type, 21

Apple WWDC demo, downloading, 14  
application delegate, using, 5–6  
array contents, reading, 137–138  
arrays, 24  
    adding objects, 138–139  
    creating, 137  
    implementing, 144  
    manipulating, 138–141  
    modifying, 145  
    removing objects from, 139–140  
    replacing objects in, 140  
    sorting, 141  
    subscripting, 114  
    Swift versus C, 136  
as operator, 35  
assertions, using in exception handling, 158  
assignment operator, 32, 34  
Assistant Editor, features of, 13  
at (@) sign, absence of, 27–28

## B

Balloons demo, downloading, 14  
bit shift operator, 32  
bitwise operator, 32  
blocks. *See also* closures  
    versus closures, 22  
    creating references to, 83  
    defining as parameters, 83–84  
    inline creation of, 83  
    passing into methods, 82  
    receiving, 82  
B00L and Bool primitive types, 21  
Boolean conditions, 43  
bridging header, creating, 166

## C

C arrays, 24  
C code, working with, 170  
C preprocessor, use of, 29  
capture list, syntax for, 132

- case conditions, grouping, 55
- case statement, including where clause in, 53–54
- cells, highlighting, 53–54
- char primitive type, 21
- Character primitive type, 21
- characters
  - inserting, 94
  - removing, 94–95
  - replacing ranges of, 94
- Class primitive type, 21
- class property, using with instances, 103
- class versus instance methods, 102
- classes
  - and access control, 112–114
  - creating, 102
  - creating instances of, 108
  - features of, 23
  - and inheritance, 111–112
  - and initializers, 108–111
  - instantiating, 110
  - and methods, 102–103
  - and properties, 103–108
  - as reference types, 132
  - and subscripting, 114–115
- closure syntax, trailing, 86
- closures. *See also* blocks
  - assigning to variables, 83
  - creating definitions, 82
  - creating type aliases, 84
  - executing, 84–85
  - implicit returns, 85
  - operator functions, 86–87
  - optimizing, 85–87
  - receiving as function parameters, 83–84
  - as reference types, 132
  - shorthand argument names, 85–86
  - syntax for capture list, 132
  - terminology, 87
  - trailing closure syntax, 86
  - type inference, 85
- code, debugging, 11, 14
- collection types, customizing, 153
- collections, 24
  - arrays, 135–141
  - dictionaries, 135, 141–144
  - mutability, 144–145
  - types, 136

- comments, using, 29–31
- computed properties, using, 104–105
- conditionals
  - if, 46
  - switch, 46–47
- console output view, displaying, 13
- constants
  - syntax for, 25
  - using, 25
- control blocks, 28–29
- control structures
  - behavior of, 42
  - for and for-in loops, 44–45
  - while and do-while loops, 45
- convenience initializers, using, 109–110
- curly braces ({}), requirement of, 42–43
- custom operators, using in Swift, 160–161.
  - See also* operators

## D

- debugging code, 11, 14
- decrement operator, 32
- dictionaries, 24
  - adding and replacing objects, 143
  - creating and initializing, 141–142
  - implementing, 144
  - key type, 142
  - listing keys and values, 143
  - modifying, 145
  - reading from, 142–143
  - removing objects, 143–144
  - value type, 142
- divide and ignore overflow operator, 33
- division operator, 32
- documentation, interactivity of, 14
- dot-syntax, use of, 28
- Double and double primitive types, 21
- dynamic dispatch, using in Objective-C, 159

## E

- enumerations. *See also* nested types
  - associated values, 124–125
  - computed properties, 126
  - creating, 122
  - features of, 23
  - methods, 126

- enumerations (*continued*)
  - overview, 122
  - raw values, 124
  - shortened name syntax, 123
  - using, 48, 123–124
- equality operator, 32–33
- exception handling, using in Objective-C, 158
- extensions
  - adding, 152
  - adding protocol conformance, 152
  - creating, 151
  - documenting with comments, 151
  - explained, 23, 151

## F

- fallthrough keyword, adding, 54–55
- filter function, using in Swift, 162
- Float and float primitive types, 21
- flow control statements. *See* control structures
- for and for-in loops, 44–45
- frameworks, including Swift code in, 171
- function scopes
  - global, 75
  - of methods, 75
  - nested, 75–76
- function types
  - external parameter names, 77
  - stating, 76
  - using, 77
- functions. *See also* generic functions; nested
  - functions; parameters
    - calling, 66–67
    - comma-separating parameters, 66–67
    - comparator, 79
    - defining, 67
    - versus methods, 22
  - methods as, 102
  - modifying parameters, 71–73
  - parameter naming, 67–69
    - as parameter values, 78
    - passing to functions, 78
    - as return values, 78–79
  - return values, 70–71
  - storing, 76

## G

- generic functions, defining, 152. *See also* functions
- generic types
  - instantiating, 153
  - support for type constraints, 154
  - using, 153–154
- generics, explained, 147
- global scope, 75
- greater-than comparisons, 32

## H

- header files, absence of, 5
- Hello, playground, appearance of, 11

## I

- @IBAction attribute, 6
- @IBOutlet attribute, 6
- id primitive type, 21
- if conditional, 43
- if statements
  - nesting, 50
  - versus switch statements, 48
- IMP primitive type, 21
- importing modules, 5
- increment operator, 32
- inheritance
  - calling to superclasses, 112
  - overriding methods, 111–112
  - overview, 111
- initializers
  - convenience and designated, 109–110
  - creating, 109
  - default, 108
  - deinitializing, 110–111
- in-out parameters, 72–73
- instance versus class methods, 102
- instances of classes, creating, 108
- Int8 primitive type, 21
- integers. *See also* pattern matching
  - enumerations, 48
  - ranges, 48–49
  - strings, 47

interactive documentation, use of, 14  
iOS playground  
    import statement, 11  
    string variable, 11  
is operator, 35

## K

KVO (key-value observing), using in  
    Objective-C, 158–159

## L

Language pop-up menu, clicking, 4  
lazy stored properties, using, 105–106  
less-than comparisons, 32  
let declaration, 24–25  
logical operators, 32–33  
long primitive types, 21  
loops  
    for and for-in, 44–45  
    while and do-while, 45

## M

main.m, absence of, 5  
map function, using in Swift, 163  
member access operator, 33  
memory management  
    object references, 130  
    overview, 130  
methods  
    executing, 102  
    versus functions, 102  
    including in protocols, 148–150  
    overriding, 111–112  
    receiving blocks, 82  
modules, importing, 5  
multiplication operator, 32  
multiply and ignore overflow operator, 33  
mutability, 25–26  
mutable arrays  
    adding objects to, 138–139  
    declaring, 144  
mutating keyword, using with structures, 121

## N

named parameters. *See also* parameters  
    differentiated, 68–69  
    matching, 68  
namespaces, using in Swift, 160  
nested functions. *See also* functions  
    explained, 65  
    using, 75–76  
nested types, 126–127. *See also* enumerations;  
    structures  
nil coalescing operator, 63  
nil objects, sending messages to, 61  
None primitive type, 21  
not equal operator, 33  
NSError, using, 170  
NSFastEnumeration protocol, counterpart to, 44  
NSInteger primitive type, 21  
NSString. *See also* strings  
NSString  
    comparing to Swift strings, 90  
    conversions, 96  
    creating explicitly, 96–97  
    path modification, 96  
    substrings, 95  
    URL methods, 96  
NSUInteger primitive type, 21

## O

@objc attribute, using, 167  
object methods, 102  
object references, strong and weak, 130–131.  
    *See also* reference types  
object types, nesting, 23  
Objective-C  
    adding Swift files, 166  
    dynamic dispatch, 159  
    exception handling, 158  
    KVO (key-value observing), 158–159  
    reflection, 159  
    using Swift code in, 167–168  
Objective-C and Swift. *See also* Swift  
    bridging header, 166  
    C code, 170  
    frameworks, 171  
    syntax changes, 169  
    type changes, 169  
    using together, 6

- Objective-C files, adding, 168
- OO (object-oriented programming), 101
- operators. *See also* custom operators
  - assignment, 34
  - behaviors, 32
  - categories, 31
  - optional, 36
  - overflow, 33–34
  - pattern match, 36–37
  - range, 35–36
  - subscript, 37–38
  - type, 36–37
- optional binding, 60
- optional chaining, 61–62
- optional operators, 36
- optionals
  - declaring, 58
  - interpolation for embedding, 63
  - unwrapping, 59–61
  - use of, 58
  - using, 59
- out-parameter, explained, 23
- overflow, taking into account, 33–34

## P

- parameter naming, using, 67–69. *See also* named parameters
- parameter values, default, 69–70
- parameters. *See also* functions; named parameters
  - modifying, 71–73
  - in-out, 72–73
  - separating in functions, 66–67
  - unnamed, 67
  - variable, 71–72
  - variadic, 73–74
- parentheses (())
  - omission of, 29, 42
  - using with functions, 66, 76–77
- pattern matching. *See also* integers
  - customizing, 53–54
  - operator, 37
  - tuples, 49–51
  - using, 49
  - value bindings, 52–53
  - where clauses, 53–54
  - wildcards, 51–52

- playgrounds. *See* Swift playgrounds
- pointers, using Swift types with, 170
- #pragma mark directive, use of, 29
- primitive types
  - extending capabilities of, 20
  - as objects, 20
  - type aliases, 20
- projects, creating, 4
- properties
  - defining in protocols, 149–150
  - getting, 34–35
  - read-only, 107–108
  - stored, 103–107
  - syntax, 103
- property observers, using, 106–107
- protocols
  - conforming to, 150–151
  - creating, 148–150
  - defining properties in, 149–150
  - including methods in, 148–150
  - restricting, 148
  - use of, 147
  - using as type constraints, 154

## Q

- Quick Look popover, appearance of, 12

## R

- range operator, 35
- ranges, 48–49
- read-only properties, using, 107–108
- reduce function, using in Swift, 163
- reference types. *See also* object references
  - classes, 132
  - closures, 132
- reflection, using in Objective-C, 159
- remainder operator, 32–33
- REPL. *See* Swift REPL
- Results sidebar
  - contents of, 11–12
  - modifying existing statements, 11
  - updates in, 11

## S

- safety features, 54–55
- SceneKit animation framework, use of, 14

- scope of methods, 75
- scrubber, using with Timeline, 13
- SEL primitive type, 21
- Selector primitive type, 21
- `self.method()`, using, 108
- `self.property`, using, 108
- semicolon (;), absence of, 6, 27
- SequenceType protocol, 44
- short primitive type, 21
- square ([ ]) brackets, use of, 28
- stored properties
  - computed properties, 104–105
  - declaring, 103
  - lazy, 105–106
  - property observers, 106–107
  - setting values, 104
- Story class, using in Swift, 162
- string content, modifying, 93
- String type
  - convenience methods, 90
  - design of, 89
  - implementation as structure, 89
  - lack of length property, 92
  - subscript notation, 91
- String.Index
  - using, 91
  - warning about values, 92
- strings. *See also* NSString; Unicode
  - advance function, 91–92
  - in alphabetical comparisons, 93
  - appending, 93
  - as Character values, 91
  - comparing, 92–93
  - comparisons, 93
  - constructing, 90
  - empty status of, 93
  - inserting, 94
  - literal syntax, 90
  - matching against, 47
  - moving indexes along, 91
  - versus NSString, 90
  - overview, 22
  - prefixes and suffixes, 93
  - removing characters from, 94–95
  - startIndex and endIndex, 91
- strong reference type
  - explained, 130
  - using, 131–132
- structures, 23. *See also* nested types
  - defining, 118–120
  - features of, 118
  - modifying, 120–121
- subscript operator, 37–38
- subscripting, 114–115
- subtraction operator, 32–33
- superclass, calling to, 112
- Swift. *See also* Objective-C
  - custom operators, 160–161
  - declaring variables in, 16
  - filter function, 162
  - map function, 163
  - namespaces, 160
  - reduce function, 163
  - Story class, 162
- Swift and Objective-C. *See also* Objective-C
  - bridging header, 166
  - C code, 170
  - frameworks, 171
  - syntax changes, 169
  - type changes, 169
  - using together, 6
- Swift code
  - including in frameworks, 171
  - using in Objective-C, 167–168
  - working with C code, 170
- .swift extension, explained, 5
- Swift files, adding to Objective-C projects, 166
- Swift header for Objective-C
  - generating, 168
  - locating, 166
- Swift playgrounds
  - creating, 10, 14–15
  - creation options, 10
  - interactive documentation, 14
  - limitations, 14
  - Node module for playgrounds, 15
  - Results sidebar, 11–12
  - source editor, 10
  - Timeline feature, 13
  - uses of, 14
- Swift REPL
  - accessing from Xcode, 17
  - command line, 17
  - comparing to playgrounds, 17
  - defining and referencing variables, 16–17
  - editing facilities, 16–17

- Swift REPL (*continued*)
  - entering code into, 15
  - explained, 15
  - limitations, 17
  - multiline statements, 16
  - testing script code, 17
  - using, 15–16
- Swift types, using with pointers, 170
- Swift versus Objective-C
  - header files, 5
  - main.m, 5
  - semicolons, 6
- switch conditional, 46–47
- switch statements versus `if`, 48
- syntax
  - @ (at) sign, 27–28
  - [ ] (square) brackets, 28
  - comments, 30–31
  - control blocks, 28–29
  - operators, 31–38
  - post-preprocessor, 29
  - semicolons (;), 27
  - tuples, 38–39

## T

- template options dialog, displaying, 4
- ternary operator, 33, 35
- Timeline feature, using, 13
- tuples
  - creating and using, 38–39
  - defining, 23
  - explained, 22
  - pattern matching, 49–51
  - reusing, 39
- type inference, 26
- type methods
  - calling, 103
  - explained, 102
- type operators, 36–37
- type safety, 26, 169
- typedefs, using with blocks, 84
- types
  - blocks, 22
  - classes, 23
  - collections, 24

- displaying, 25
- enumerations, 23
- functions, 22
- immutable, 26
- mutable, 26
- primitive, 20–21
- strings, 22
- structs, 23
- tuples, 22–23

## U

- UInt\* primitive types, 21
- UIView
  - creating, 12
  - Quick Look preview of, 13
- Unicode, 97–98. *See also* strings
- unsigned char primitive type, 21
- unsigned long long primitive type, 21
- unsigned primitive type, 21
- unsigned short primitive type, 21
- unwrapping optionals, 59–61, 63

## V

- value bindings, pattern matching, 52–53
- values, capturing with closures, 87
- variable declarations
  - mutability, 25–26
  - type inference, 26
  - var and let, 24–25
- variable parameters, 71–72
- variables, declaring in Swift, 16
- variadic parameters, 73–74
- Void and void primitive types, 21

## W

- weak reference type
  - explained, 130
  - using, 131–132
- where clause, including in case statement, 53–54
- while and do-while loops, 45
- wildcards, pattern matching, 51–52