LEARNING Swift™ 2 PROGRAMMING

JACOB SCHATZ

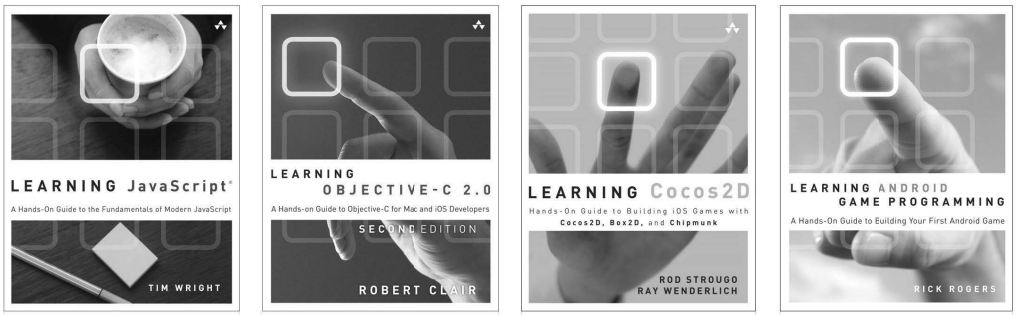
FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Learning Swift 2 Programming

Second Edition

Addison-Wesley Learning Series



Visit **informit.com/learningseries** for a complete list of available publications.

The **Addison-Wesley Learning Series** is a collection of hands-on programming guides that help you quickly learn a new technology or language so you can apply what you've learned right away.

Each title comes with sample code for the application or applications built in the text. This code is fully annotated and can be reused in your own projects with no strings attached. Many chapters end with a series of exercises to encourage you to reexamine what you have just learned, and to tweak or adjust the code as a way of learning.

Titles in this series take a simple approach: they get you going right away and leave you with the ability to walk off and build your own application and apply the language or technology to whatever you are working on.

Learning Swift 2 Programming

Second Edition

Jacob Schatz

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town • Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City • São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Learning Swift 2 Programming
Second Edition

ISBN-13: 978-0-13-443159-8

ISBN-10: 0-13-443159-6

Library of Congress Control Number: 2015957570

Printed in the United States of America

First Printing: December 2015

Trademarks
All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. The publisher cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer
Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Special Sales
For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Acquisitions Editor
Mark Taber

Managing Editor
Sandra Schroeder

Project Editor
Seth Kerney

Copy Editor
Cheri Clark

Indexer
Cheryl Lenser

Proofreader
Megan
Wade-Taxter

Technical Editor
Mike Keen

Editorial Assistant
Vanessa Evans

Designer
Chuti Prasertsith

Compositor
codeMantra

Contents at a Glance

Introduction 1

1 Getting Your Feet Wet: Variables, Constants, and Loops 5

2 Collecting Your Data: Arrays and Dictionaries 31

3 Making Things Happen: Functions 43

4 Structuring Code: Enums, Structs, and Classes 57

5 SpriteKit 75

6 Reusable Code: Closures 93

7 Creating Your Own Syntax: Subscripts and Advanced Operators 109

8 Protocols 129

9 Becoming Flexible with Generics 153

10 Games with SpriteKit 163

11 Making Games with Physics 187

12 Making Apps with UIKit 205

Index 219

Table of Contents

Introduction 1

1 Getting Your Feet Wet: Variables, Constants, and Loops 5

Building Blocks of Swift 6

Computed Properties (Getters and Setters) 6

Using Comments 8

Inference 8

Merging Variables into a `String` 10

Optionals: A Gift to Unwrap 11

Printing Your Results 14

Implicitly Unwrapped Optionals 14

Tuples 15

Number Types 16

From Objective-C to Swift 17

Control Flow: Making Choices 18

Switching It Up: `switch` Statements 25

Stop...Hammer Time 28

Summary 29

2 Collecting Your Data: Arrays and Dictionaries 31

Using Arrays 31

Your First Array the Long Way 31

A Quicker Array 32

Using `AnyObject` 32

Differences Between `NSArray`s and Swift Arrays 33

Modifying Arrays 33

Accessing Array Elements 33

Adding Elements to an Array 34

Removing Elements from Arrays 34

Iterating Over Arrays 35

Extra Bits of Arrays 35

Emptying an Array 36

Using Dictionaries 36

Adding, Removing, and Inserting with Dictionaries 37

Iterating Over Dictionaries 37
Extra Bits of Dictionaries 38
Emptying a Dictionary 38
Testing Dictionaries for the Presence of Values 38
Putting It All Together 39
Summary 41

3 Making Things Happen: Functions 43
Defining Functions 44
Return Types 45
Multiple Return Values 46
More on Parameters 47
External Parameter Names 47
Default Parameter Values 48
Variadic Parameters 49
In-Out Parameters 50
Functions as Types 51
Putting It All Together 52
Summary 55

4 Structuring Code: Enums, Structs, and Classes 57
Enums 58
Which Member Was Set? 59
Associated Values 59
Raw Values 60
Structs 61
Defining Methods in Structs 63
Structs Are Always Copied 64
Mutating Methods 65
Classes 66
Initialization 66
What Is a Reference Type? 68
Do I Use a Struct or a Class? 68
Forgot Your Pointer Syntax? 69
Property Observers 69
Methods in Classes 70
Summary 74

5 SpriteKit 75
Introducing SpriteKit 75
The `SKNode` and `SKSpriteNode` 75
Creating a Game 76
The New Project Screen 76
The Game 85
Summary 92

6 Reusable Code: Closures 93
What Are Closures? 93
Closures in Other Languages 94
How Closures Work and Why They're Awesome 95
The Closure Syntax 96
Inferring Using Context 96
Arguments Have a Shorthand, Too 97
Sorting a Custom `Car` Class 97
Closures Are Reference Types 98
Automatic Reference Counting 99
Strong Reference Cycles 100
Trailing Closures 106
Summary 107

7 Creating Your Own Syntax: Subscripts and Advanced Operators 109
Writing Your First Subscript 110
Bits and Bytes with Advanced Operators 113
Bitwise `NOT` 114
Bitwise `AND` 115
Bitwise `OR` 116
Bitwise `XOR` 117
Shifting Bits 118
`UInt8`, `UInt16`, `UInt32`, `Int8`, `Int16`, `Int32`, and So On 119
Value Overflow and Underflow 119
Customizing Operators 120
Making Your Own Operators 122
Bits and Bytes in Real Life 123
Summary 127

8 Protocols 129
Writing Your First Protocol 129
Properties 131
`Animizable` and `Humanizable` 134
Methods 135
Delegation 136
Protocols as Types 138
Protocols in Collections 139
Protocol Inheritance 140
Protocol Composition 141
Protocol Conformity 143
Optional Protocol Prerequisites 145
Optional Chaining 146
Back to Optional Protocol Requisites 148
Useful Built-in Swift Protocols 149
Summary 151

9 Becoming Flexible with Generics 153
The Problem That Generics Solve 153
Other Uses for Generics 155
Generics for Protocols 157
The `where` Clause 158
Summary 162

10 Games with SpriteKit 163
The Game 163
The Setup 163
Tour the Code 164
The Game 164
Step 1: Create the World 165
Step 2: Making Things Move 176
Summary 185

11 Making Games with Physics 187
Making a Physics-Based Game 187
Creating the Project 188
Adding the Assets 189
Adding the Levels 189

Generating the Levels 190
Making a Playable Game 197
Creating the Cage 199
Summary 204

12 Making Apps with UIKit 205
Application Types 205
Single-View Applications 206
Creating the User Interface 208
Adding Constraints 209
Hooking Up the UI to Code 211
Writing the Code 212
The `TableView` 216
Summary 218

Index 219

About the Author

Jacob Schatz is a senior software engineer with more than eight years of experience writing code for the masses. His code is often used by millions of people, and his advice is often sought. Jacob also goes by the name Skip Wilson and has a popular YouTube channel currently covering Swift and Python. Jacob is always selectively consuming the latest programming trends. He has a passion for making a difference and is constantly solving problems. Lately he has been deep into Swift, but he also writes tons of JavaScript, Python, Objective-C, and other languages. He is always learning more languages and thoroughly enjoys making new things. He is, at heart, a pedagogue, and he enjoys teaching and finding new ways to explain advanced concepts.

Dedication

For Tiffany and Noa

Acknowledgments

I could not have written this book without the help of many people. Thank you to the following:

Logan Wright, who wrote tons of YouTube tutorials with me and helped me with this book.

Cody Romano, who graciously helped me write and proofread, and whose endless knowledge has helped me debug more than a few bugs.

Mike Keen, who tirelessly proofread chapters and tried all my examples to make sure they were legit. He also provided an endless source of inspiration.

Mom and Dad, who, even though they had no idea what they were reading, sat there and read this book thoroughly, providing sage advice.

My wife, who put up with me spending countless hours in front of my computer, and through the process of this book has become an advanced programmer.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

We welcome your comments. You can email or write directly to let us know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that we cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail we receive, we might not be able to reply to every message.

When you write, please be sure to include this book's title and author, as well as your name and phone or email address.

Email: errata@informit.com

Mail: Addison-Wesley/Prentice Hall Publishing
ATTN: Reader Feedback
330 Hudson Street
7th Floor
New York, New York, 10013

Reader Services

Register your copy of *Learning Swift 2 Programming* (ISBN 978-0-13-443159-8) at informit.com/register for convenient access to downloads, updates, and corrections as they become available.

This page intentionally left blank

Introduction

WELCOME TO *LEARNING SWIFT* 2 *PROGRAMMING*, SECOND EDITION. This book will launch you into the world of iOS programming using the exciting new Swift programming language. This book covers Swift from start to finish, in a quick but complete way.

This Introduction covers the following:

- Who should read this book
- Why you should read this book
- What you will be able to achieve using this book
- What Swift is and why it is awesome
- How this book is organized
- Where to find the code examples

Ready?

Who Should Read This Book

This book is for those who already have one or many programming languages under their belt. You may be able to get through this book with Swift as your first language, but you'll find it easier if you can relate it to other languages. If you have experience with iOS programming with Objective-C, you should really be able to take to Swift quickly. This book often relates Swift concepts to those of other popular programming languages, including JavaScript, Python, Ruby, C, and Objective-C.

Why You Should Read This Book

This book will teach you all aspects of Swift programming so you can start writing high-quality apps as quickly as possible. However, it is not an exhaustive reference; it is a complete yet easy-to-digest initiation into Swift. This book will make you a better developer; because Swift is a mixture of many languages, you are bound to learn new concepts here. Swift is very robust on its own, and at the same time it allows you to mix in Objective-C.

If you are reading this book, you've probably heard people talking about Swift's amazing features. You've heard about its advanced design, how fast it runs, and how much easier your

development will be. This book shows you all those features of the Swift language, as well as some very exciting discoveries I've made with it. Now is the perfect time to jump right in. This book will get you fully immersed and provide everything you need in order to get up and running as quickly as possible.

What You Will Learn from This Book

Reading this book will make you an official Swift programmer and allow you to write real-world, production-quality apps. You'll write apps that take advantage of the most advanced features of Swift, so you'll be writing refined, clean code. After reading this book, you'll be able to create any app you want in Swift. Here are just a few things you will learn while reading this book:

- How to combine existing Objective-C code into new Swift applications
- How to use advanced features like generics to write less code
- How to create optionals as a quicker way to make sure your code doesn't crash at runtime due to nonexistent values
- How to write closures to pass around little blocks of functionality, which can be written in as little as four characters
- How to create a 2D side-scrolling game using SpriteKit
- How to create a 3D game using SceneKit
- How to read bits and bytes so you can do things like read a PDF

What Is Swift?

Swift is a new programming language from Apple that replaces and also works alongside languages like C and Objective-C. The idea with Swift is to make it easier to write apps for iOS with a language that is fresh and new. The Swift language relates to many other languages. It is also so customizable that you can write Swift in many ways. For example, Swift allows you to define what square brackets do; instead of always using them for array and dictionary access, you can technically make them do whatever you want. Swift allows you to define your own operators and override existing ones. If you want to make a new triple incrementor (such as `+++`) that increments twice instead of once, then you can do that. Plus, you can create custom operators to work with your custom classes, which means you'll write less code and therefore make your life easier. For example, if you were to write a program about automobiles, you could define what would happen if you were to add two cars instances to each other. Normally you can only add numbers to each other, but in Swift you can override the + operator to do whatever you want.

Swift is well structured and completely compatible with Objective-C. All the libraries available in Objective-C are also available in Swift. Swift allows you to create bridges that connect languages.

How This Book Is Organized

This book is divided into 12 chapters, which cover the language itself and walk you through creating a few apps:

- Chapters 1–4 cover basic language syntax, including variables, constants, arrays, dictionaries, functions, classes, enums, and structs. These are the basic building blocks of the Swift language.
- Chapter 5 takes a break from the language syntax and helps you create a basic game of tic-tac-toe.
- Chapters 6–9 cover more advanced language features, including closures, subscripts, advanced operators, protocols and extensions, generics, and programming on the bit and byte levels.
- Chapters 10–12 show you how to create real-world apps using the knowledge you've gained from previous chapters.

Enjoy the Ride

My goal was to make this book fun to read, and I had a lot of fun writing it. I want to show you how exciting learning a new language can be.

When a new language comes out, often not a whole lot of knowledge is out there about it. This book aims to give you direct access to knowledge that is hard to find, and it is an easy-to-read version of a lot of knowledge that is hard to read. Searching online for answers can be difficult because Swift evolves and we all are still figuring out Swift together. There are, of course, bugs in the language, and I'm sure there will continue to be bugs. I wrote this book while Swift was still in beta (and constantly changing) and finished it up as Swift became version 2.0. Swift will continue to change and improve as more people use it and report bugs as time goes on. This book has been tested against the latest version of Swift (as of this writing), but that doesn't mean that Swift won't change. I hope you enjoy learning to use Swift.

This page intentionally left blank

1

Getting Your Feet Wet: Variables, Constants, and Loops

Swift is a new programming language created by Apple, with the intention of making development of software for Apple products significantly easier. If you have experience in C and Objective-C, you should find Swift to be a walk in the park. All the classes and types that are available to you in C and Objective-C are ported over and available in their exact incarnations in Swift.

If, however, you come from a Ruby or Python background, you will find Swift's syntax to be right up your alley. Swift borrows and iterates on many ideas from Python and Ruby.

If you come from the JavaScript world, you will be pleased to know that Swift also doesn't ask you to declare types, as old strict Java does. You will also be pleased to know that Swift has its own version of `indexOf` and many other familiar JavaScript functions. If they aren't the exact replicas of said functions, they will at least be familiar.

If you come from the Java world, you will be happy to know that even though Swift does not force you to declare types, you still can and Swift most certainly enforces those types, very strictly.

These are all just basic syntax comparisons; the real magic evolves from Swift's chameleon-like capability to be written in any way that makes you the programmer comfortable. If you want to write the tersest one-liner that does everything you ever needed in one fell swoop, Swift has you covered. If you want to write Haskell-like functional programming, Swift can do that, too. If you want to write beautiful object-oriented programming with classic design patterns, Swift will do that as well.

In the future (or now, depending on when you are reading this), Swift will be open source so that you can officially (theoretically) write Swift on Linux or Windows. Someone may even create a web framework like Ruby on Rails in Swift.

This chapter covers the basic building blocks of Swift. It starts with variables and constants. With this knowledge, you will be able to store whatever you'd like in memory. Swift has a special feature called *optionals,* which allows you to check for `nil` values in a smoother way than in other programming languages. As I briefly mentioned before, Swift has strong type inference; this allows you to have strict typing without needing to declare a type. This chapter also goes over how Swift handles loops and `if`/`else` statements.

Building Blocks of Swift

Swift allows you to use variables and constants by associating a name with a value of some type. For example, if you want to store the string `"Hi"` in a variable named `greeting`, you can use a variable or a constant. You create a variable by using the `var` keyword. This establishes an associated value that can be changed during the execution of the program. In other words, it creates a mutable storage. If you do not want mutable storage, you can use a constant. For example, you might record the number of login retries a user is allowed to have before being refused access to the site. In such a case, you would want to use a constant, as shown in this example:

```
var hiThere = "Hi there"
hiThere = "Hi there again"

let permanentGreeting = "Hello fine sir"
permanentGreeting = "Good morning sir"
```

Notice that you don't use a semicolon as you would in many other languages. Semicolons are not mandatory, unless you want to combine many statements together on the same line. In Swift you would not put a semicolon on the end of the line, even though Swift will not complain. Here is an example that shows you when you would use the semicolon in Swift when multiple lines are combined into one:

```
let numberOfRetries = 5; var currentRetries = 0
```

Also unique to Swift, you can use almost any Unicode character to name your variables and constants. Developers can name resources using Hebrew, Simplified Chinese, and even special Unicode characters, such as full-color koala emoji.

When declaring multiple variables, you can omit the `var` keyword. Here is an example:

```
var yes = 0, no = 0
```

Computed Properties (Getters and Setters)

In Swift you can also declare variables as computed properties. You would use this when you want to figure out the value of the variable at runtime. Here is an example of a getter, where

the value of the score is determined by how much time is left. In this example we are creating a read-only computed property.

```
var timeLeft = 30
var score:Int {
get{
    return timeLeft * 25
}
}
print(score)
```

In this example we can reference (or read) `score` anywhere because it is in the global scope. What is really interesting is that if we try to set the score, it will give us an error because we have created a read-only property. If we want to be able to set this property, we need to create a setter. You cannot create a setter without a getter. Aside from the fact that it would not make sense, it also just will not work. Let's create a setter to go along with our getter. It does not make sense for a setter to set the computed property directly because the value of the property is computed at runtime. Therefore, you use a setter when you want to set other values as a result of the setter being set. Also, setters work well in some sort of organizational unit, which we haven't covered yet, but it's worth diving into briefly. Here is a full Swift example, which includes many elements we have not covered yet.

```
import UIKit
struct Book {
    var size = CGSize()
    var numberOfPages = 100;
    var price:Float {
    get{
        return Float(CGFloat(numberOfPages) * (size.width * size.height))
    }
    set(newPrice){
        numberOfPages = Int(price / Float(size.width * size.height))
    }
    }
}

var book = Book(size: CGSize(width: 0.5, height: 0.5), numberOfPages: 400)
print(book.price)
book.price = 400
print(book.numberOfPages)
```

In this example we create a book `Struct`, which is a way to organize code so that it is reusable. I would not expect you to understand all of this example, but if you have ever coded in any other languages, you will notice that there is a lot of type casting going on here. Type casting is a something you do all the time in Objective-C and most other languages. We will cover all aspects of this code in this book, but you should know that we created a setter, which sets the number of pages in the book relative to the new price.

Using Comments

You indicate comments in Swift by using a double forward slash, exactly as in Objective-C. Here's an example:

```
// This is a comment about the number of retries
let numberOfRetries = 5 // We can also put a comment on the end of a line.
```

If you want to create comments that span multiple lines, you can use this /* */ style of comments, which also works well for documentation.

```
/* Comments can span
multiple lines */
```

Inference

Swift uses inference to figure out what *types* you are trying to use. Because of this, you do not need to declare a type when creating variables and constants. However, if you want to declare a type you may do so, and in certain situations, it is absolutely necessary. When declaring a variable, the rule of thumb is that Swift needs to know what type it is. If Swift cannot figure out the type, you need to be more explicit. The following is a valid statement:

```
var currentRetries = 0
```

Notice that Swift has to figure out what type of number this is. `currentRetries` may be one of the many types of numbers that Swift offers (Swift will infer this as an `Int` in case you are wondering, but more on that later). You could also use this:

```
var currentRetries:Int = 0
```

In this case, you explicitly set the type to `Int` by using the colon after the variable name to declare a type. Although this is legit, it is unnecessary because Swift already knows that `0` is an `Int`. Swift can and will infer a type on a variable that has an initial value.

When do you need to declare the type of a variable or constant? You need to declare the type of a variable or constant if you do not know what the initial value will be. For example:

```
var currentRetries:Int
```

In this case, you must declare `Int` because without it, Swift cannot tell what type this variable will be. This is called *type safety*. If Swift expects a string, you must pass Swift a string. You cannot pass an `Int` when a `String` is expected. This style of coding is a great time-saver. You will do a lot less typing with your fingers and a lot more thinking with your brain. Every default value you give a variable without a type will be given a type. Let's talk about numbers first.

For number types, Swift gives us the following:

- `Int` is available in 8, 16, 32, and 64 bits, but you will most likely stay with just `Int`. It's probably large enough for your needs. Here's what you need to know about `Int`:

  `Int` on 32-bit platforms is `Int32`.

  `Int` on 64-bit platforms is `Int64`.

That is, when you declare a variable as `Int`, Swift will do the work of changing that to `Int32` or `Int64`. You don't need to do anything on your end.

`Int` can be both positive and negative in value.

`Int` will be the default `type` when you declare a variable with a number and no decimals:

```
var someInt = 3 // this will be an Int
```

`UInt` is provided as an *unsigned* integer. An unsigned number must be positive, whereas a *signed* number (an `Int`) can be negative. For consistency, Apple recommends that you generally use `Int` even when you know that a value will never be negative.

- `Double` denotes 64-bit floating-point numbers. `Double` has a higher precision than `float`, with at least 15 decimal digits. `Double` will be the chosen type when you declare a variable that has decimals in it:

  ```
  var someDouble = 3.14 // this will be a double
  ```

  Combining any integer with any floating-point number results in a `Double`:

  ```
  3 + 3.14 // 6.14 works and will be a double
  var three = 3
  var threePointOne = 3.1
  three + threePointOne //Error because you can't mix types
  ```

- `Float` denotes 32-bit floating-point numbers. `Float` can have a precision as small as 6. Whether you choose `Float` or `Double` is completely up to you and your situation. Swift will choose `Double` when no type is declared.

Along with `Decimal` numbers, you can use `Binary`, `Octal`, and `Hexadecimal` numbers:

- `Decimal` is the default for all numbers, so no prefix is needed.
- Create a `Binary` number by adding a `0b` prefix.
- `Octal` uses a `0o` prefix.
- `Hexadecimal` uses a `0x` prefix.

You can check the type of the object by using the `is` keyword. The `is` keyword will return a Boolean. In this example we use the `Any` class to denote that `pi` can be anything at all until we type it as a `Float`:

```
var pi:Any?
pi = 3.141
pi is Double //true
pi is Float  //false
```

Notice that you declare this type as `Any?` in the preceding example. The question mark denotes an optional, which allows us to not set an initial value without causing an error. The `Any` type can be any type (exactly what it says). Objective-C is not as strict as Swift, and you need to

be able to intermingle the two languages. For this purpose, `Any` and `AnyObject` were created, which allows you to put any type in an object. Think about arrays in Objective-C, which can mix different types together; for that purpose you need to give Swift the ability to have arrays of different types. You'll learn more about this later in the chapter.

Swift is the only programming language (that I know of) that lets you put underscores in numbers to make them more legible. Xcode ignores the underscores when it evaluates your code. You might find using underscores especially useful with big numbers when you want to denote a thousand-comma separator, as in this case:

```
var twoMil = 2_000_000
```

Before you can add two numbers together, they must be made into the same type. For example, the following will not work:

```
var someNumA:UInt8 = 8
var someNumB:Int8 = 9
someNumA + someNumB
//Int8 is not convertible to UInt8
```

The reason this does not work is that `someNumA` is a `UInt8` and `someNumB` is an `Int8`. Swift is very strict about the combination of things.

To make this work, you must convert one of the types so that the two types are the same. To do this, use the *initializer* of the type. For example, you can use the initializer `UInt8`, which can convert `someNumB` to a `UInt8` for you:

```
someNumA + UInt8(someNumB)
```

Swift is strict and makes sure that you convert types before you can combine them.

We had to do a lot of conversions of types in a previous example.

Merging Variables into a `String`

When you want to combine a variable in a string there is a special syntax for that. Take an example in which you have a variable `message` and you want to mix it into a string. In Objective-C you would do something like this:

```
[NSString stringWithFormat:@"Message was legit: %@", message];
```

In JavaScript you would do something like this:

```
"Message was legit:" + message;
```

In Python you would do something like this:

```
"Message was legit: %s" % message
```

In Ruby you would do something like this:

```
"Message was legit: #{message}"
```

In Swift you do something like this:

```
"Message was legit: \(message)"
```

You use this syntax of `\()` to add a variable into a string. Of course, this will interpret most things you put in between those parentheses. This means you can add full expressions in there like math. For example:

```
"2 + 2 is \(2 + 2)"
```

This makes it very simple to add variables into a string. Of course, you could go the old-school way and concatenate strings together with the plus operator. In most situations you don't need to do this because the `\()` makes things so much easier. One thing to remember is that Swift has strict type inference, so if you try to combine a `String` with an `Int`, Swift will complain. The error it gives is not the easiest to decipher. For example:

```
"2 + 2 is " + (2 + 2)
```

This returns the following error (depending on your version of Swift and how you are running it):

```
<stdin>:3:19: error: binary operator '+' cannot be
applied to operands of type 'String' and 'Int'
print("2 + 2 is " + (2 + 2))
~~~~~~~~~~~ ^ ~~~~~~~
<stdin>:3:19: note: overloads for '+' exist with these
partially matching parameter lists: (Int, Int),
(String, String), (UnsafeMutablePointer<Memory>,
Int), (UnsafePointer<Memory>, Int)
print("2 + 2 is " + (2 + 2))
```

What this means is that you can't mix `String`s and `Int`s. So you have to convert the `Int` to a `String`.

```
"2 + 2 is " + String(2 + 2)
```

This works because you are now combining a `String` and an `Int`. One of the most important things to keep in mind when writing Swift is that you'll often do a lot of type conversion to deal with the strict typing.

Optionals: A Gift to Unwrap

In our tour through the basic building blocks of Swift, we come to optionals. Optionals are a unique feature of Swift, and they are used quite extensively. Optionals allow you to safely run code where a value may missing, which would normally cause errors. Optionals take some getting used to. Optionals help you achieve clean-looking code with fewer lines while also being stricter.

In many languages, you need to check objects to see whether they are `nil` or `null`. Usually, you write some pseudo-code that looks like the following. In this example we check for not null in JavaScript:

```
if(something != null) {...
```

In Swift, an optional either contains a value or it doesn't. In other languages, we often have to deal with missing values, such as a variable that once contained a value but no longer does. Or when a variable is initialized without a value. To mark something as optional, you just include a `?` next to the type of the object. For example, here's how you create a `String` optional:

```
var someString:String? = "Hey there!"
```

You can now say that `someString` is of type `String?` (a "`String` optional") and no longer just of type `String`. Try printing that variable as an optional string and then as a regular string. Notice the difference in their returned values.

```
var greetingOptional:String? = "hi there"
var greeting:String = "Hi"
print(greetingOptional) //Optional("hi there")
print(greeting) //"Hi"
```

If you choose to use an optional and it does contain a value, you must do something special to get raw value out. Optionals must be "unwrapped" in order to get their value back. There are a couple ways to get the value out of an optional. When you see a variable of type `String?`, you can say that this variable may or may not contain a value. You will test this `String` optional to find out whether it does in fact have a value. How do you test an optional? There are a couple of ways. First try to use *value binding*.

Value binding allows you to do two things. First, it allows you to test the optional to see whether it is `nil` (whether it contains a value). Second, if that variable is not `nil`, value binding allows you to grab the value out of the optional and have it passed into a constant as a locally scoped variable. To see this in action, take a look at an example, but before you can try it out, you first need to open a new playground:

1. Open Xcode.
2. Click Get started with a playground.
3. Save a new playground file by giving it a filename.

Now you can try out value binding with optionals:

```
var hasSomething:String? = "Hey there!"
if let message = hasSomething {
    "Message was legit: \(message)"
} else {
    "There was no message!"
}
```

A couple of new things are going on here. Let's go through this example one step at a time:

1. On the first line, you create a variable as usual, but you add the `?` to say that this is a `String` optional. This means that this `String` may contain a value or `nil`. In this case, it contains a value. That value is the string `"Hey there!"`.
2. Next, you write an `if` statement. You are testing whether the variable `hasSomething` is `nil`. At the same time, you are assigning that value of the optional to a constant `message`. If the variable contains a value, you get a new constant (available only in the local scope, so we call it a locally scoped constant), which is populated with the raw value of the optional. You will then enter into the `if` statement body.
3. If you do enter into that `if` statement, you now have a `message` to use. This constant will be available only in that `if` statement.

However, sometimes you are absolutely sure that your optional contains a value and is not empty. You can think of optionals as a gift that needs to be unwrapped. If an optional is `nil` inside, it will not throw an error when you use it. In other languages, trying to access something of `nil` (or `null`) value throws an error.

You can unwrap an optional by using an exclamation point. That is, you can get the value inside the optional by using an exclamation point. Let's look again at our earlier example:

```
var hasSomething:String? = "Hey there!"
print(hasSomething) // Optional("Hey there!")\n
// Now unwrap the optional with the "!"
print(hasSomething!) // "Hey there!\n"
```

If you were sure that the string contained a value, you could unwrap the optional with the "`!`." Now you can get the value out of the optional with one extra character. Remember how we said optionals are like wrapped-up presents? Well, it's sometimes good to think of them more like bombs in Minesweeper. If you are too young for Minesweeper, think of them as presents that could contain bombs. You want to unwrap an optional with the "!" only if you are absolutely sure it contains a value. You want to unwrap an optional with the "!" only if you are absolutely sure it does not contain `nil`. If you unwrap an optional that's `nil`, using "`!`," then you will throw a fatal error, and your program will crash:

```
var hasSomething:String? //declare the optional string with no initial
   value
// Now try and force it open
hasSomething! // fatal error:
Execution was interrupted, reason: EXC_BAD_INSTRUCTION...
```

When you get an `EXC_BAD_INSTRUCTION` somewhere, it means that your app is trying to access something that does not exist, which could be an error with an empty optional trying to unwrap with the "`!`."

Printing Your Results

When you use the playground to test your code, you have two options for printing data. You can simply just write it, like this:

```
var someString = "hi there"
someString //prints "hi there" in the output area
```

You can also use `print()`, which prints to the console output area. When you are making a full-fledged app, compiling code outside a playground, you'll want to use `print()`, like this, because just writing the variable will not do anything:

```
var someString = "hi there"
print(someString) //prints "hi there" in the console output
```

Implicitly Unwrapped Optionals

Sometimes you want to create an optional that gets unwrapped automatically. To do this, you assign the type with an exclamation point instead of a question mark:

```
var hasSomething:String! = "Hey there"// implicitly unwrapped optional string
hasSomething // print the implicitly unwrapped optional and get the
   unwrapped value.
```

You can think of implicitly unwrapped optionals as a present that unwraps itself. You should not use an implicitly unwrapped optional if a chance exists that it may contain `nil` at any point. You can still use implicitly unwrapped optionals in value binding to check their values.

So why should you create implicitly unwrapped optionals in the first place if they can be automatically unwrapped? How does that make them any better than regular variables? Why even use them in the first place? These are fantastic questions, and we will answer them later, after we talk about classes and structures in Chapter 4, "Structuring Code: Enums, Structs, and Classes." One quick answer is that sometimes we want to say that something has no value initially but we promise that it will have a value later. Properties of classes must be given a value by the time initialization is complete. We can declare a property with the exclamation point to say, in effect, "Right now it does not have a value, but we promise we will give this property a value at some point."

Also, sometimes you will have a constant that cannot be defined during initialization, and sometimes you will want to use an Objective-C API. For both of these reasons and more, you will find yourself using implicitly unwrapped optionals. The following example has two examples (with some concepts not covered yet) in which you would commonly use implicitly unwrapped optionals.

```
class SomeUIView:UIView {
    @IBOutlet var someButton:UIButton!
    var buttonWidth:CGFloat!
```

```
    override func awakeFromNib() {
        self.buttonOriginalWidth = self.button.frame.size.width
    }
}
```

In this example you have a button, which you cannot initialize yourself because the button will be initialized by Interface Builder. Also, the width of the button is unknown at the time of the creation of the class property, so you must make it an implicitly unwrapped optional. You will know the width of the button after `awakeFromNib` runs, so you promise to update it then.

Tuples

Using *tuples* (pronounced "TWO-pulls" or "TUH-pulls") is a way to group multiple values into one value. Think of associated values. Here is an example with URL settings:

```
let purchaseEndpoint = ("buy","POST","/buy/")
```

This tuple has a `String`, a `String`, and a `String`. This tuple is considered to be of type (`String`, `String`, `String`). You can put as many values as you want in a tuple, but you should use them for what they are meant for and not use them like an array or a dictionary. You can mix types in tuples as well, like this:

```
let purchaseEndpoint = ("buy","POST","/buy/",true)
```

This tuple has a `String`, a `String`, a `String`, and a `Bool`. You are mixing types here, and this tuple is considered to be of type (`String`, `String`, `String`, `Bool`). You can access this tuple by using its indexes:

```
purchaseEndpoint.1 // "POST"
purchaseEndpoint.2 // "/buy/"
```

This works well but there are some inconveniences here. You can guess what `POST` and `/buy/` are, but what does `true` stand for? Also, using indexes to access the tuple is not very pretty or descriptive. You need to be able to be more expressive with the tuple.

You can take advantage of Swift's capability to name individual elements to make your intentions clearer:

```
let purchaseEndpoint = (name: "buy", httpMethod: "POST",URL: "/buy/",useAuth: true)
```

This tuple has `String`, `String`, `String`, and `Bool` (true or false) values, so it is the same type as the previous tuple. However, now you can access the elements in a much more convenient and descriptive way:

```
purchaseEndpoint.httpMethod = "POST"
```

This is much better. It makes much more sense and reads like English.

You can *decompose* this tuple into multiple variables at once. Meaning you can take the tuple and make multiple constants or variables out of it in one fell swoop. So if you want to get the `name`, the `httpMethod`, and the `URL` into individual variables or constants, you can do so like this:

```
let (purchaseName, purchaseMethod, purchaseURL, _) = purchaseEndpoint
```

Here, you are able to take three variables and grab the meat out of the tuple and assign it right to those variables. You use an underscore to say that you don't need the fourth element out of the tuple. Only three out of the four properties of the tuple will be assigned to constants.

In Chapter 3, "Making Things Happen: Functions," you will use tuples to give functions multiple return values. Imagine having a function that returned a tuple instead of a string. You could then return all the data at once and do something like this:

```
func getEndpoint(endpoint:String) ->
    (description: String, method: String, URL: String) {
    return (description: endpoint, method: "POST", URL: "/\(endpoint)/")
}
let purchaseEndpoint = getEndpoint("buy")
print("You can access the
    \(purchaseEndpoint.description) endpoint at the URL \(purchaseEndpoint.URL)")
```

Number Types

Swift is interoperable with Objective-C, so you can use C, Objective-C, and Swift types and code all within Swift. As discussed earlier in the chapter, when you write a variable using an integer, Swift automatically declares it with a type `Int`, without your having to tell Swift you want an `Int`. In this example, you don't tell Swift to make this variable an `Int`:

```
let theAnswerToLifeTheUniverseAndEverything = 42
```

Rather, Swift infers that it is an `Int`. Remember that on 32-bit systems this `Int` will be an `Int32`, and on 64-bit systems it will be an `Int64`. If you don't remember that it won't matter because Swift will convert this for you automatically anyway. Even though you have many different `Int` types available to you, unless you need an `Int` of a specific size, you should stick with Swift's `Int`. When we say `Int32`, what we mean is a 32-bit integer. (This is similar to C.) You can also use `UInt` for unsigned (non-negative) integers, but Apple recommends that you stick with `Int` even if you know that your variable is going to be unsigned.

Again, when you write any type of floating-point number (a number with a decimal), and you don't assign a type, Swift automatically declares it with the type `Double`. Swift also gives you `Double` and `Float` types. The difference between them is that `Double` has a higher precision of around 15 decimal digits, whereas `Float` has around 6. Here is an example of a `Double` in Swift:

```
let gamma = 0.57721566490153286060651209008240243104215933593992
```

Swift is strict about its types and they get combined together. If something is meant to be a `String`, and you give it an `Int`, then you will get an error. Swift needs you to be explicit with types. For example, this will not work:

```
var someInt = 5 // Inferred to be an Int
someInt + 3.141 // throws an error
```

This throws an error because you can't combine an `Int` and a `Double`. If you want to combine an `Int` and a `Double`, you must first convert the `Int` to a `Double` or vice versa, depending on your preference. Here we combine an `Int` and a `Double` by converting the `Int` to a `Double`:

```
var someInt = 5 // Inferred to be an Int
Double(someInt) + 3.141 // 8.141

var someInt = 5 // Inferred to be an Int
Float(someInt) + 3.141 // In this case 3.141 will be inferred to be a Float so
// it can combine with a Float

var someInt = 5 // Inferred to be an Int
Float(someInt) + Double(3.141) //This will throw an error and will not work
```

You can use the initializer (`Float(someInt)` or `Double(someInt)`, etc.) of the number type to convert between types. For example, you can use `Float()` to convert any number type into a `Float`.

So again, when you want to perform any operations on two or more number types, all sides of the operation must be of the same type. You'll see this pattern often in Swift, and not just with numbers. For example, you cannot directly add a `UInt8` and a `UInt16` unless you first convert the `UInt8` to a `UInt16` or vice versa.

From Objective-C to Swift

If you are coming from the world of Objective-C and C, you know that you have many number types at your disposal. Number types like `CGFloat` and `CFloat` are necessary to construct certain objects. For example, SpriteKit has the `SKSpriteNode` as a position property, which uses a `CGPoint` with two `CGFloat`s.

What is the different between `CGFloat` and `Float`? In this specific case we found that `CGFloat` is just a `typealias` for `Double`. This is what the code actually says:

```
typealias CGFloat = Double
```

What is a `typealias`? Great question. A `typealias` is just a shortcut to get to an already existing type by giving it a substitute name. You could give `String` an alternative name type of `Text`, like this:

```
typealias Text = String
var hello:Text = "Hi there"
```

Now `hello` is of type `Text`, which never existed before this point. So if `CGFloat` is a `typealias` for a `Double`, this just means that when you make `CGFloat`s, you are really just making `Double`s. It's worth it to Command+click around and see what is mapping to what. For example, a `CFloat` is a `typealias` for `Float`, and a `CDouble` is a `typealias` for `Double`.

That does not mean that you can suddenly add them together. You still need to convert them to combine them. For example, this will not work:

```
var d = 3.141
var g = CGFloat(3.141)
print(d + g)
```

To fix this example we would need to do something like this:

```
var d = 3.141
var g = CGFloat(3.141)
print(CGFloat(d) + g)
```

Control Flow: Making Choices

Controlling the order in which your code executes is obviously a crucial aspect of any programming language. By building on the traditions of C and C-like languages, Swift's control flow constructs allow for powerful functionality while still maintaining a familiar syntax.

`for` Loops

At its most basic, a `for` loop allows you to execute code over and over again. This is also called "looping." How many times the code gets executed is up to you (maybe infinitely). In the Swift language, there are two distinct types of `for` loops to consider. There is the traditional `for-condition-increment` loop, and there is the `for-in` loop. `for-in` is often associated with a process known as *fast enumeration*—a simplified syntax that makes it easier to run specific code for every item. `for-in` loops give you far less code to write and maintain than your typical C `for` loops.

`for-condition-increment` Loops

You use a `for-condition-increment` loop to run code repeatedly until a condition is met. On each loop, you typically increment a counter until the counter reaches the desired value. You can also decrement the counter until it drops to a certain value, but that is less common. The basic syntax of this type of loop in Swift looks something like this:

```
for initialization; conditional expression; increment {
    statement
}
```

As in Objective-C and C, in Swift you use semicolons to separate the different components of the `for` loop. However, Swift doesn't group these components into parentheses. Aside from this slight syntactic difference, `for` loops in Swift function as they would in any C language.

Here's a simple example of a `for-condition-increment` loop that simply prints `Hello` a few times:

```
for var i = 0; i < 5; ++i {
    print("Hello there number \(i)")
}
// Hello there number 0
// Hello there number 1
// Hello there number 2
// Hello there number 3
// Hello there number 4
```

This is fairly straightforward, but notice the following:

- Variables or constants declared in the *initialization expression* only exist within the scope of the loop. If you need to access these values outside the scope of the `for` loop, then the variable must be declared prior to entering the loop, like this:

  ```
  var i = 0
  for i; i < 5; ++i {...
  ```

- If you're coming from another language, particularly Objective-C, you will notice that the last example uses `++i` instead of `i++`. Using `++i` increments `i` before returning its value, whereas using `i++` increments `i` after returning its value. Although this won't make much of a difference in the earlier example, Apple specifically suggests that you use the `++i` implementation unless the behavior of `i++` is explicitly necessary.

`for-in` Loops and Ranges

In addition to giving you the traditional `for-condition-increment` loop, Swift builds on the enumeration concepts of Objective-C and provides an extremely powerful `for-in` statement. You will most likely want to use `for-in` for most of your looping needs because the syntax is the most concise and also makes for less code to maintain.

With `for-in`, you can iterate numbers in a range. For example, you could use a `for-in` loop to calculate values over time. Here you can loop through 1 to 4 with less typing:

```
class Tire{}
var tires = [Tire]()
for i in 1...4 {
    tires.append(Tire())
}
print("We have \(tires.count) tires")
```

We haven't covered the class and array syntax yet, but maybe you can take a guess at what they do. This example uses a `...` range operator for a closed range. The range begins at the first number and includes all the numbers up to and including the second number.

Swift also provides you with the half-open range operator, which is written like this:

```
1..<4
```

This range operator includes all numbers from the first number up to but not including the last number. The previous example, rewritten to use the non-inclusive range operator, would look like this:

```
class Tire{}
var tires = [Tire]()
for i in 1..<5 {
    tires.append(Tire())
}
print("We have \(tires.count) tires")
```

As you can see, the results are almost identical and both examples provide concise and readable code. When you don't need access to `i`, you can disregard the variable altogether by replacing it with an underscore (_). The code now might look something like this:

```
class Tire { }
var tires = [Tire]()
// 1,2,3, and including 4

class Tire {}
var tires = [Tire]()
for _ in 1...4 {
    tires.append(Tire())
}
print("We have \(tires.count) tires")
```

Let's pretend that a bunch of tires have gone flat, and you need to refill each tire with air. We could use the `for` in loop to loop through all of our tires in the tire array. We can add on to our earlier example by giving the tires air. You could do something like this:

```
class Tire { var air = 0 }
var tires = [Tire]()
for _ in 1...4 {
    tires.append(Tire())
}
print("We have \(tires.count) tires")

for tire in tires {
    tire.air = 100
    print("This tire is filled \(tire.air)%")
}
print("All tires have been filled to 100%")
```

With this type of declaration, Swift uses *type inference* to assume that each object in an array of type `[Tire]` will be a `Tire`. This means it is unnecessary to declare the type `Tire` explicitly. In a situation in which the array's type is unknown, the implementation would look like this:

```
class Tire { var air = 0 }
var tires = [Tire]()
for _ in 1...4 {
    tires.append(Tire())
}
print("We have \(tires.count) tires")

for tire: Tire in tires {
    tire.air = 100
    print("This tire has been filled to \(tire.air)%")
}
```

In this example we told the loop that each tire in the array of tires is going to be specifically of type `Tire`. In this specific example there is not a good reason to do this, but you may come upon a situation in which the type is not set explicitly. Since Swift must know what types it is dealing with, you should make sure that you communicate that information to Swift.

Looping Through Other Types

In Swift, a `String` is really a collection of `Character` values in a specific order. You can iterate values in a `String` by using a `for-in` statement, like so:

```
for char in "abcdefghijklmnopqrstuvwxyz".characters {
    print(char)
}
// a
// b
// c
// etc....
```

As long as something conforms to the `SequenceType`, you can loop through it. You cannot loop through the string directly; you need to access its character property.

When looping, there will be situations in which you will need access to the index as well as the object. One option is to iterate through a range of indexes and then get the object at the index. You would write that like this:

```
let numbers  = ["zero", "one", "two", "three", "four"]
for idx in 0..<numbers.count {
    let numberString = array[idx]
    print("Number at index \(idx) is \(numberString)")
}
// Number at index 0 is zero
// Number at index 1 is one
// etc....
```

This works just fine, but Swift provides a much swifter way to do this. Swift gives you an `enumerate` method as part of the array, which makes this type of statement much more concise. Let's use the `for-in` statement in with the `enumerate` method:

```
let numbers = ["zero", "one", "two", "three", "four"]
for (i, numberString) in numbers.enumerate() {
    print("Number at index \(i) is \(numberString)")
}
// Number at index 0 is zero
// Number at index 1 is one
// etc....
```

This is much clearer, and you would use it when you require an array element and its accompanying index. You can grab the index of the loop and the item being iterated over!

Up to this point, all the loops we've covered know beforehand how many times they will iterate. For situations in which the number of required iterations is unknown, you'll want to use a `while` loop or a `do while` loop. The syntax to use these `while` loops is very similar to that in other languages. Here's an example:

```
var i = 0
while i < 10 {
    i++
}
```

This says that this loop should increment the value of `i` while it is less than 10. In this situation, `i` starts out at 0, and on each run of the loop, `i` gets incremented by 1. Watch out, though, because you can create an infinite loop this way. There will be times when you really need an infinite loop.

One way to create an infinite `while` loop is to use `while true`:

```
while true {
}
```

In this example you use a `while` loop that always evaluates to `true`, and this loop will run forever, or until you end the program or it crashes itself.

This next example uses some of the looping capabilities plus `if/else` statements to find the prime numbers. Here's how you could find the 200th prime number:

```
var primeList = [2.0]
var num = 3.0
var isPrime = 1
while primeList.count < 200 {
    var sqrtNum = sqrt(num)
    // test by dividing only with prime numbers
    for primeNumber in primeList {
        // skip testing with prime numbers greater
        // than square root of number
        if num % primeNumber == 0 {
```

```
            isPrime = 0
            break
        }
        if primeNumber > sqrtNum {
            break
        }
    }
    if isPrime == 1 {
        primeList.append(num)
    } else {
        isPrime = 1
    }
    //skip even numbers
    num += 2
}
print(primeList)
```

Grabbing `primeList[199]` will grab the 200th prime number because arrays start at 0. You can combine `while` loops with `for-in` loops to calculate prime numbers.

To `if` or to `else` (`if/else`)

It's important to be able to make decisions in code. It's okay for you to be indecisive but you wouldn't want that for your code. Let's look at a quick example of how to make decisions in Swift:

```
let carInFrontSpeed = 54
if carInFrontSpeed < 55 {
    print("I am passing on the left")
} else {
    print("I will stay in this lane")
}
```

You use Swift's `if` and `else` statements to make a decision based on whether a constant is less than 55. Since the integer 54 is less than the integer 55, you print the statement in the `if` section.

One caveat to `if` statements is that in some languages you can use things that are "truthy," like 1 or a non-empty array. That won't work in Swift. You must conform to the protocol `BooleanType`. To make this simple, you must use `true` or `false`. For example, here are some examples that will not work:

```
if 1 { // 1 in an Int and can't be converted to a boolean
    //Do something
}
var a = [1,2,3]
if a.count{ // a.count is an Int and can't be converted to a boolean
    print("YES")
}
```

If you want to make these examples work, you would have to convert those numbers to a `Bool`. For example, check out what happens when we convert some simple integers to Booleans.

```
print(Bool(1)) // true
print(Bool(2)) // true
print(Bool(3)) // true
print(Bool(0)) // false
```

If `Int`s can be converted into `Bool`s, you can check for `if` with a truthy value if you first convert it to a `Bool`. Let's rewrite our previous failing examples and make them work.

```
if Bool(1) { // 1 in an Int and can't be converted to a boolean
    print("Duh it works!")
}
var a = [1,2,3]
if Bool(a.count){ // a.count is an Int and can't be converted to a boolean
    print("YES")
}
```

We were able to check for 1 and an array `count` in the `if` statement because we first converted them to `Bool`s.

You may also want to check multiple statements to see whether they're `true` or `false`. You want to check whether the car in front of you slows down below 55 mph, whether there is a car coming, and whether there is a police car nearby. You can check all three in one statement with the `&&` operator. This operator states that both the statement to its left and the one to its right must be `true`. Here's what it looks like:

```
var policeNearBy = false
var carInLane3 = false
var carInFrontSpeed = 45
if !policeNearBy && !carInLane3 && carInFrontSpeed < 55 {
    print("We are going to pass the car.")
} else {
    print("We will stay right where we are for now.")
}
```

Your code will make sure that all three situations are `false` before you move into the next lane (the `else`).

Aside from the `and` operator, you also have the `or` operator. You can check to see whether any of the statements is `true` by using the `or` operator, which is written as two pipes: `||`. You could rewrite the preceding statement by using the `or` operator. This example just checks for the opposite of what the preceding example checks for:

```
var policeNearBy = false
var carInLane3 = false
var carInFrontSpeed = 45
if policeNearBy || carInLane3 || carInFrontSpeed > 55 {
    print("We will stay right where we are for now.")
```

```
} else {
    print("We are going to pass the car.")
}
```

If any of the preceding variables is `true`, you will stay where you are; you will not pass the car.

Aside from just `if` and `else`, you may need to check for other conditions. You might want to check multiple conditions, one after the other, instead of just going straight to an `else`. You can use `else if` for this purpose, as shown in this example:

```
var policeNearBy = false
var carInLane3 = false
var carInFrontSpeed = 45
var backSeatDriverIsComplaining = true
if policeNearBy || carInLane3 || carInFrontSpeed > 55 {
    print("We will stay right where we are for now.")
} else if backSeatDriverIsComplaining {
    print("We will try to pass in a few minutes")
}else {
    print("We are going to pass the car.")
}
```

You can group as many of these `else if`s together as you need. However, when you start grouping a bunch of `else if` statements together, it might be time to use the `switch` statement.

Switching It Up: `switch` Statements

You can get much more control and more readable code if you use a `switch` statement. Using tons of `if else` statements might not be as readable. Swift's `switch` statements are very similar to those in other languages with extra power added in. The first major difference with `switch` statements in Swift is that you do not use the `break` keyword to stop a condition from running through each `case` statement. Swift automatically breaks on its own when the condition is met.

Another caveat about `switch` statements is that they must be absolutely exhaustive. That is, if you are using a `switch` statement on an `int`, you need to provide a `case` for every `int` *ever*. This is not possible, so you can use the `default` statement to provide a match when nothing else matches. Here is a basic `switch` statement:

```
var num = 5
switch num {
case 2:print("It's two")
case 3:print("It's three")
default:print("It's something else")
}
```

This tests the variable `num` to see whether it is 2, 3, or something else. Notice that you must add a `default` statement. As mentioned earlier, if you try removing it, you will get an error because the `switch` statement must exhaust every possibility. Also note that `case 3` will not run if `case 2` is matched because Swift automatically breaks for you.

You can also check multiple values at once. This is similar to using the `or` operator (`||`) in `if else` statements. Here's how you do it:

```
var num = 5
switch num {
case 2,3,4:print("It's two") // is it 2 or 3 or 4?
case 5,6:print("It's five") // is it 5 or 6?
default:print("It's something else")
}
```

In addition, you can check within ranges. The following example determines whether a number is something between 2 and 6:

```
var num = 5
switch num {
// including 2,3,4,5,6
case 2...6:print("num is between 2 and 6")
default:print("None of the above")
}
```

You can use tuples in `switch` statements. You can use the underscore character (_) to tell Swift to "match everything." You can also check for ranges in tuples. Here's how you could match a geographic location:

```
var geo = (2,4)
switch geo {
//(anything, 5)
case (_,5):print("It's (Something,5)")
case (5,_):print("It's (5,Something)")
case (1...3,_):print("It's (1 or 2 or 3, Something)")
case (1...3,3...6):print("This would have matched but Swift already found a match")
default:print("It's something else")
}
```

In the first `case`, you are first trying to find a tuple whose first number is anything and whose second number is 5. The underscore means "anything," and the second number must be 5. Our tuple is 2,4 so that won't work because the second number in our tuple is 4.

In the second `case`, you are looking for the opposite of the first `case`. In this instance the first number must be 5 and the second number can be anything.

In the third `case`, you are looking for any number in the range 1 to 3, including 3, and the second number can be anything. Matching this `case` causes the `switch` to exit. We can use ranges to check numbers in switch statements, which makes them even more powerful.

The next `case` would also match, but because Swift has already found a match, it never executes. In this case we are checking two ranges.

Switch statements in Swift break on their own. If you've ever programmed in any other common language, you know you have to write `break` so that the case will stop. If you want that typical Objective-C, JavaScript functionality that will not use `break` by default (where the third `case` and fourth `case` will match), you can add the keyword `fallthrough` to the `case`, and the `case` will not break:

```
var geo = (2,4)
switch geo {
//(anything, 5)
case (_,5):print("It's (Something,5)")
case (5,_):print("It's (5,Something)")
case (1...3,_):
    print("It's (1 or 2 or 3, Something)")
    fallthrough
case (1...3,3...6):
    print("We will match here too!")
default:print("It's something else")
}
```

Now the third `case` and fourth `case` match, and you get both `print` statements:

```
It's (1 or 2 or 3, Something)
We will match here too!
```

Remember the value binding example from earlier? You can use this same idea in `switch` statements. Sometimes it's necessary to grab values from the tuple. You can even add in a `where` statement to make sure you get exactly what you want. Here is the kitchen-sink example of `switch` statements:

```
var geo = (2,4)
switch geo {
case (_,5):print("It's (Something,5)")
case (5,_):print("It's (5,Something)")
case (1...3,let x):
    print("It's (1 or 2 or 3, \(x))")
case let (x,y):
    print("No match here for \(x) \(y)")
case let (x,y) where y == 4:
    print("Not gonna make it down here either for \(x) \(y)")
default:print("It's something else")
}
```

You might get a warning here telling you that a `case` will never be executed, and that is okay. This is the mother of all `switch` statements. Notice that the last two cases will never run. You can comment out the third and fourth `switch` statements to see each run. We talked about the first `case` and second `case`. The third `case` sets the variable `x` (to `4`) to be passed into the

print if there is a match. The only problem is that this works like the underscore by accepting everything. You can solve this with the where keyword. In the fourth case, you can declare both x and y at the same time by placing the let outside the tuple. Finally, in the last case, you want to make sure that you pass the variables into the statement, and you want y to be equal to 4. You control this with the where keyword.

Stop...Hammer Time

It's important to have some control over your switch statements and loops. You can use break, continue, and labels to provide more control.

Using break

Using break stops any kind of loop (for, for in, or while) from carrying on. Say that you've found what you were looking for, and you no longer need to waste time or resources looping through whatever items remain. Here's what you can do:

```
var mystery = 5
for i in 1...8 {
    if i == mystery {
        break
    }
    print(i) // Will be 1, 2, 3, 4
}
```

The loop will never print 5 and will never loop through 6, 7, or 8.

Using continue

Much like break, continue will skip to the next loop and not execute any code below the continue. If you start with the preceding example and switch out break with continue, you will get a result of 1, 2, 3, 4, 6, 7, and 8:

```
var mystery = 5
for i in 1...8 {
    if i == mystery {
        continue
    }
    print(i) // Will be 1, 2, 3, 4, 6, 7, 8
}
```

Using Labeled Statements

break and continue are fantastic for controlling flow, but what if you had a switch statement inside a for in loop? You want to break the for loop from inside the switch statement, but you can't because the break you write applies to the switch statement and not the loop.

In this case, you can label the `for` loop so that you can tell the `for` loop to break and make sure that the `switch` statement does not break:

```
var mystery = 5
rangeLoop: for i in 1...8 {
    switch i {
    case mystery:
        print("The mystery number was \(i)")
        break rangeLoop
    case 3:
        print("was three. You have not hit the mystery number yet.")
    default:
        print("was some other number \(i)")
    }
}
```

Here, you can refer to the right loop or `switch` to `break`. You could also `break for` loops within `for` loops without returning a whole function. The possibilities are endless.

Summary

This chapter has covered a lot of ground. You can see that Swift isn't another version of Objective-C. Swift is a mixture of principles from a lot of languages, and it really is the best of many languages. It has ranges, which pull syntax straight out of Ruby. It has `for in` loops with `enumerate` and tuples, which both are straight out of Python. It has regular `for` loops with `i++` or `++i`, which come from C and many other languages. It also has optionals, which are Swift's own invention.

You'll see shortly that Swift has a lot of cool features that make it easy to use along with your Objective-C and C code. You have already gotten a small taste of arrays. Chapter 2, "Collecting Your Data: Arrays and Dictionaries," covers arrays and dictionaries in detail. You'll see how Swift's strong typing and optionals come into play.

This page intentionally left blank

Index

Symbols

& (ampersand)
  bitwise AND, 115-116
  calling functions by reference, 50
  value underflow/overflow, 119-120
<< (bitwise left shift operator), 118
>> (bitwise right shift operator), 118
^ (caret), bitwise XOR, 117
/* */ (comments), 8
&& (double ampersand), AND operator, 24
== (double equal sign), equality operator, 121, 149
// (double forward slash), comments, 8
|| (double pipe), OR operator, 24
... (ellipsis)
  closed ranges, 19
  variadic parameters, 49-50
= (equal sign), assignment operator, 122
! (exclamation point), optionals, 13-15
@objc attribute, 145-146
| (pipe), bitwise OR, 116-117
+ (plus sign), concatenating strings, 11
? (question mark), optionals, 9, 12
..< (range operator), half-open ranges, 20
; (semicolon), 6
[ ] (square-brackets notation)
  array elements, accessing, 33-34
  dictionaries, accessing, 37
  dictionary values, setting, 37
\() syntax, variables in strings, 10-11
~ (tilde), bitwise NOT, 114-115
_ (underscore)
  external function parameters, 48
  in numbers, 10
0b prefix, 113

A

access modifiers, property access modifiers, 71

accessing
classes via subscripts, 110
elements in arrays, 33-34, 110-111
indexes via for loops, 21-22
raw values in enums, 60-61
strings via for loops, 21
tuples, 15
values in dictionaries, 37-39, 111
adding
background images to games, 89-91
constraints, 209-211
elements to arrays, 34
values to dictionaries, 37
ampersand (&)
bitwise AND, 115-116
calling functions by reference, 50
value underflow/overflow, 119-120
anchor points, setting, 91-92
AND operator
bitwise operations, 115-116
double ampersand (&&), 24
animation
collision detection, 176-180
in physics-based game, 197-198
with SKAction class, 180-185
anonymous functions, 43
Any type, 9-10
AnyObject type, 32-33
AppDelegate class, 136
append() method, 32, 34
AppKit, 205
ApplicationDelegate, 176
apps
constraints, adding, 209-211
single-view applications, 206-207
Storyboard, 205
user interface
connecting to code, 211-212
creating, 208-209
writing code for, 212-218
ARC (Automatic Reference Counting), 99-100
strong reference cycles, 100-102
in closures, 104-106
unowned keyword, 102-104
weak keyword, 102
arguments. *See also* parameters
calling functions, 44-46
in closures, 97
arrays
AnyObject type in, 32-33
concatenating, 34
declaring, 31-32
with generics, 155-157
elements
accessing, 33-34, 110-111
inserting, 34
iterating over, 35
prepopulating, 35
removing, 34-35
emptying, 36
example code, 39-41
as function parameters, 49-50
multidimensional, 35-36
subscripts in, 111-113
mutable versus immutable, 33
types allowed, 31
as? keyword, 144-145
AspectFill scale mode, 82-84
AspectFit scale mode, 84
assert() method, 112
asset library for physics-based game, 189

assignment operator, 122
associated types, 157-158
associated values in enums, 59-60
atlases, storing images, 170-171
Automatic Reference Counting. *See* **ARC (Automatic Reference Counting)**

B

background images, adding to games, 89-91
Ball class, 191-192
BallManager class, 192-197
base of classes, finding, 78-79
binary numbers, calculating, 113-114
binary operators, 121
Binary type, 9
bits
binary number calculations, 113-114
shifting, 118
bitwise operations
examples of usage, 113-114
NOT operator, 114-115
AND operator, 115-116
OR operator, 116-117
reading GIF files example, 123-127
shifting bits, 118
signed versus unsigned types, 119
value underflow/overflow, 119-120
XOR operator, 117
Bluetooth, bitwise operations and, 113
Boolean type, 23-25
break statements, 28
bytes, binary number calculations, 113-114

C

C programming language, comparison with Swift, 5
calling functions
by reference, 50
by value, 44
capture lists, defining, 105
caret (^), bitwise XOR, 117
case statements in switch statements, 25-28
CDouble type, 18
CFloat type, 18
CGFloat type, 17
CGPoint class, 173
CGSize class, 84
changeStartButton() method, 213
class keyword, 66
classes
accessing via subscripts, 110
AppDelegate, 136
ARC (Automatic Reference Counting), 99-100
closures, 104-106
strong reference cycles, 100-102
unowned keyword, 102-104
weak keyword, 102
Ball, 191-192
BallManager, 192-197
CGPoint, 173
CGSize, 84
custom, sorting, 97-98
declaring, 66
Diamond, 170
finding base, 78-79
game managers, 85-86
GameHelper, 167

GameManager, 167-169
GameScene
collision detection, 199-203
files in, 164-165
setting up, 174-175
GameViewController, 79-84
changing scale mode, 82-84
ignoring sibling order, 81-82
SKView class and, 79-81
Ghost, 169-170
Hero, 169
methods
inheritance, 72-74
instance methods, 70-71
property access modifiers, 71
self keyword, 72
type methods, 71-72
multiple initializers, 66-68
NSDateComponentsFormatter, 218
properties. *See* properties
property observers, 69-70
as reference types, 68
SKAction, 76, 180-185
SKLabelNode, 77
SKNode, 75-76
SKPhysicsBody, 178-179
SKScene, 78-79
SKSceneScaleMode, 82-84
SKShapeNode, 77-78
SKSpriteNode, 76-77
SKView, 79-81
structs versus, 57, 61-62
StudySessionManager, 216
Tile, 169
ViewController, 206
when to use, 68-69

closures, 43, 93
custom classes, sorting, 97-98
declaring, 95-97
in JavaScript, 94-95
as reference types, 98-99
strong reference cycles in, 104-106
trailing closures, 106-107
code listings. *See* listings
collections, protocols in, 139-140
collision detection, 176-180, 190-191, 199-203. *See also* physics-based games
comments, 8
Comparable protocol, 97, 134, 149-150
comparison operators
Comparable protocol and, 149-150
equality operator, 121, 149
composition (protocols), 141-143
computed properties, 6-7, 131
concatenating
arrays, 34
strings, 11
conforming to protocols, 129-131, 143-145
connecting user interface to code, 211-212
constants
declaring, 6, 60
naming, Unicode characters, 6
constraints, adding, 209-211
constructors. *See* initializers
continue statements, 28
control flow
break statements, 28
continue statements, 28
for loops, 18-22

accessing indexes, 21-22
accessing strings, 21
for-condition-increment loops, 18-19
for-in loops, 19-21
if-else statements, 23-25
labeled statements, 28-29
switch statements, 25-28
while loops, 22-23
converting types, 10, 17
Boolean type, 23-24
Int type to String type, 11
count method
arrays, 34
dictionaries, 38
creating. *See* declaring
custom classes, sorting, 97-98
custom operators, 109
defining new, 122-123
overloading, 120-122
custom types, creating with generics, 155-157
CustomStringConvertible protocol, 150-151

D

data types. *See* types
DebugPrintable protocol, 151
Decimal type, 9
declaring
arrays, 31-32
with generics, 155-157
classes, 66
closures, 95-97
constants, 6, 60
dictionaries, 36-38
enums, 58
functions, 44-45
protocols, 129-131
method requirements, 135-136
structs, 62-63
subscripts, 110-113
types, 8
variables, 6
as computed properties, 6-7
in structs, 63
default parameters, 48-49
default statements, 25
defining
capture lists, 105
methods in structs, 63-64
operators, 122-123
deinit function, 101
delegates
collision detection, 176
described, 129
delegation, 136-138
deleting. *See* removing
Diamond class, 170
dictionaries
declaring, 36-38
emptying, 38
example code, 39-41
values
accessing, 37-39, 111
inserting, 37
iterating over, 37-38
removing, 37
testing for presence, 38-39
didBeginContact() method, 200
didSet() function, 70
directories, creating projects directory, 77

double ampersand (&&), AND operator, 24
double equal sign (==), equality operator, 121, 149
double forward slash (//), comments, 8
Double type, 9, 16-17
dynamic bodies, static bodies versus, 180

E

elements. *See also* values
  accessing, 33-34, 110-111
  inserting, 34
  iterating over, 35
  naming in tuples, 15
  prepopulating, 35
  removing, 34-36
ellipsis (.)
  closed ranges, 19
  variadic parameters, 49-50
else statements, 23-25
else-if statements, 25
emptying. *See also* removing
  arrays, 36
  dictionaries, 38
enum keyword, 58
enumerate function, 22, 35
enums
  declaring, 58
  members, 58
    associated values, 59-60
    determining which was set, 59
    raw values, 60-61
  purpose of, 57
equal sign (=), assignment operator, 122
equality (==) operator, 121, 149
Equatable protocol, 134, 149
  custom classes, sorting, 97
  custom operators, 121
  with generics, 157
errors
  conforming to protocols, 130-131
  EXC_BAD_INSTRUCTION, 13
example code. *See* listings
EXC_BAD_INSTRUCTION error, 13
exclamation point (!), optionals, 13-15
exclusive OR operator, 117
external parameters, 47-48

F

fadeInWithDuration() method, 181
fadeOutWithDuration() method, 181
fallthrough keyword, 27
file specifications, bitwise operations and, 113
files in GameScene class, 164-165
Fill scale mode, 84
finding base of classes, 78-79
Float type, 9, 16-17
for-condition-increment loops, 18-19
for-in loops, 19-21
  enumerate method, 22
for loops, 18-22
  accessing indexes, 21-22
  accessing strings, 21
  for-condition-increment loops, 18-19
  for-in loops, 19-21
formatting time, 218
func keyword, 44, 47
functions, 43. *See also* methods

anonymous functions, 43
calling
by reference, 50
by value, 44
closures. *See* closures
declaring, 44-45
deinit, 101
didSet(), 70
enumerate, 22, 35
example code, 52-55
generics in, 153-155
init(), 66
keys, 38
methods versus, 70
operators as, 93
parameters
arrays as, 49-50
in declarations, 44
default parameters, 48-49
external parameters, 47-48
in-out parameters, 50-51
variadic parameters, 49-50
return types, 45-46
multiple return values, 46-47
sort(), as closure, 95-97
as types, 51-52
values, 38
where keyword, 158-162
willSet(), 70

G

game managers, 85-86
GameHelper class, 167
GameManager class, 167-169
games
creating
adding background image, 89-91
deleting sample code, 85
game manager, 85-86
new project, creating in Xcode, 76-77, 163-164, 188-189
scene sizing, 86-89
setting anchor points, 91-92
physics-based games, 187-188
adding levels, 189-190
animation of ball, 197-198
asset library, 189
collision detection, 190-191, 199-203
creating game world, 190-197
running, 77
SpriteKit, 75
animation with SKAction class, 180-185
collision detection, 176-180
creating game world, 165-175
description of example game, 163
files in GameScene class, 164-165
finding base of classes, 78-79
GameViewController class, 79-84
initial code, 164
SKLabelNode class, 77
SKNode class, 75-76
SKShapeNode class, 77-78
SKSpriteNode class, 77
stage, 75
GameScene class
collision detection, 199-203
files in, 164-165
setting up, 174-175

GameViewController class, 79-84
changing scale mode, 82-84
ignoring sibling order, 81-82
SKView class and, 79-81
generics, 153
for protocols, 157-158
T type, 95-96, 154-155
usage
in custom types, 155-157
in functions, 153-155
where keyword, 158-162
getters, 6-7
in dictionaries, 111
as properties, 131-133
for subscripts, 110
Ghost class, 169-170
GIF files, reading (bitwise operations example), 123-127
global variables for timers, 212-213

H

Hero class, 169
Hexadecimal type, 9
home directory, creating projects directory, 77

I

if-else statements, 23-25
ignoring sibling order, 81-82
images, storing in atlases, 170-171
immutable arrays, 33
implicitly unwrapped optionals, 14-15
indexes
accessing via for loops, 21-22
tuples, accessing, 15
inference, 8-10, 16-17
in closures, 96
infinite while loops, 22
infix operators, 123
info.plist, 165-167, 189-190
inheritance, 72-74
of protocols, 140-141
init() function, 66
initial game code in SpriteKit, 164
initializers, 10, 17
for classes, 66
memberwise initializers, 64
multiple initializers, 66-68
initializing physics, 178-179
inline closures, 96
inout keyword, 50, 122
in-out parameters, 50-51
insert() method, 34
inserting
elements in arrays, 34
values in dictionaries, 37
instance methods, 70-71
instances, ARC (Automatic Reference Counting), 99-100
closures, 104-106
strong reference cycles, 100-102
unowned keyword, 102-104
weak keyword, 102
Int type, 8-9, 16-17
bitwise operations, 119
converting to String type, 11
Int32 type, 8, 16
Int64 type, 8, 16
internal keyword, 71
invalidate() method, 215

is keyword, 9, 144-145
isGameOver() method, 202
iterating
over arrays, 35
over dictionaries, 37-38

J

Java, comparison with Swift, 5
JavaScript
closures in, 94-95
comparison with Swift, 5

K

keys
dictionaries, declaring, 36-37
removing from dictionaries, 38
keys function, 38
keywords
as?, 144-145
class, 66
enum, 58
fallthrough, 27
func, 44, 47
inout, 50, 122
internal, 71
is, 9, 144-145
lazy, 104, 134
let, 33, 60, 131
mutating, 65
operator, 122
private, 71
protocol, 129
public, 71
self, 72
static, 71
subscript, 110
super, 73
typedef, 57
unowned, 102-104
var, 6, 33, 131
weak, 102
where, 28, 158-162

L

labeled statements, 28-29
landscape orientation, 174
lazy keyword, 104, 134
let keyword, 33, 60, 131
levels (in games), creating, 165-167, 189-190
listening for screen taps, 181-182
listings
array example code, 39
function example code, 52
loading tiles on screen, 171-173
logical shifting, 118
loops
break statements, 28
continue statements, 28
for loops, 18-22
accessing indexes, 21-22
accessing strings, 21
for-condition-increment loops, 18-19
for-in loops, 19-21
while loops, 22-23

M

main.storyboard, 205
members, 58

associated values, 59-60
determining which was set, 59
raw values, 60-61
memberwise initializers, 64
memory management, ARC (Automatic Reference Counting), 99-100
closures, 104-106
strong reference cycles, 100-102
unowned keyword, 102-104
weak keyword, 102
methods. *See also* functions
append(), 32, 34
assert(), 112
changeStartButton(), 213
count
arrays, 34
dictionaries, 38
defining in structs, 63-64
didBeginContact(), 200
fadeInWithDuration(), 181
fadeOutWithDuration(), 181
functions versus, 70
inheritance, 72-74
insert(), 34
instance methods, 70-71
invalidate(), 215
isGameOver(), 202
moveBy(), 181
moveByX(), 181
moveTo(), 180-181
mutating in structs, 65
print(), 14
property access modifiers, 71
in protocol declarations, 135-136
optional methods, 145-146
rawvalue, 60
removeAllActions(), 181
removeAtIndex(), 34
removeFromParent(), 183
removeLast(), 34
removeValueForKey(), 37
resizeToWidth(), 181
reverse(), 202
self keyword, 72
startStopTapped(), 214
tableView(), 216-218
timeIntervalSinceReferenceDate(), 184
timerTick(), 215
type methods, 71-72
updateValue(), 37
viewDidLoad(), 213
moveBy() method, 181
moveByX() method, 181
moveTo() method, 180-181
moving. *See* animation
multidimensional arrays, 35-36
subscripts in, 111-113
multi-line comments, 8
multiple initializers, 66-68
multiple protocols, 141-143
multiple return values for functions, 46-47
multiple variables, declaring, 6
mutable arrays, 33
mutable storage, 6
mutating keyword, 65
mutating methods
in protocol declarations, 136
in structs, 65

N

naming
constants, Unicode characters, 6
elements in tuples, 15
values in tuples, 46-47
variables, Unicode characters, 6
nested functions. *See* closures
nil, testing optionals for, 12-13
node trees, 79
nodes. *See also* SKNode class
described, 78
types of, 77-78
NOT operator, bitwise operations, 114-115
NSArray, arrays versus, 33
NSDateComponentsFormatter class, 218
NSMutableArray, arrays versus, 33
number types, 8-9, 16-17
typealiases, 17-18
underscore (_) in, 10

O

Objective-C, comparison with Swift, 5
Octal type, 9
open source, Swift as, 5
OpenGL, 75
OR operator
bitwise operations, 116-117
double pipe (||), 24
operator keyword, 122
operators
assignment operator, 122
binary operators, 121
bitwise operations
NOT operator, 114-115
AND operator, 115-116
OR operator, 116-117
reading GIF files example, 123-127
shifting bits, 118
value underflow/overflow, 119-120
XOR operator, 117
comparison operators
Comparable protocol and, 149-150
equality operator, 121, 149
custom operators, 109
defining new, 122-123
overloading, 120-122
as functions, 93
splat operator, 159
ternary operator, 122
unary operators, 121-122
optional methods for protocols, 145-146
optionals, 6, 11-15
! (exclamation point), 13-15
? (question mark), 9, 12
accessing dictionaries, 37-39
accessing enum raw values, 61
chaining, 146-148
unwrapping, 12-13
implicitly unwrapping, 14-15
organizing games, 85-86
overloading operators, 120-122
overriding, 74

P

parameters. *See also* arguments
arrays as, 49-50
in closures, 97
default parameters, 48-49
external parameters, 47-48
in function declarations, 44-45
in-out parameters, 50-51
variadic parameters, 49-50
Photoshop, bitwise operations and, 113
physics, initialization, 178-179
physics-based games, 187-188. *See also* collision detection
adding levels, 189-190
animation of ball, 197-198
asset library, 189
creating game world, 190-197
pipe (|), bitwise OR, 116-117
pixels in point system, 86-89
plus sign (+), concatenating strings, 11
point system, sizing scenes, 86-89
pointer syntax, 69
portrait orientation, changing to landscape, 174
postfix operators, 123
unary operators, 121-122
prefix operators, 123
unary operators, 121-122
prepopulating elements in arrays, 35
print() method, 14
Printable protocol, 134, 143, 150-151
printing variables, 14
private keyword, 71
projects, creating in Xcode, 76-77, 163-164, 188-189
projects directory, creating, 77
properties
computed properties, 6-7, 131
getters and setters, 131-133
lazy keyword, 104, 134
property access modifiers, 71
property observers, 69-70
read-only stored properties, creating, 133
stored properties, 131
property access modifiers, 71
property observers, 69-70
protocol keyword, 129
protocols
in collections, 139-140
Comparable, 97, 134, 149-150
composition, 141-143
conforming to, 129-131, 143-145
CustomStringConvertible, 150-151
DebugPrintable, 151
declaring, 129-131
method requirements, 135-136
delegation, 136-138
described, 129
Equatable, 134, 149
custom classes, sorting, 97
custom operators, 121
with generics, 157
generics for, 157-158
inheritance, 140-141
optional chaining, 146-148
optional methods, 145-146
Printable, 134, 143, 150-151
SequenceType, 162
as types, 138-139

UITableViewDataSource, 213
UITableViewDelegate, 213
public keyword, 71
Python, comparison with Swift, 5

Q

question mark (?), optionals, 9, 12

R

ranges
for-in loops, 19-21
in switch statements, 26
raw values in enums, 60-61
rawvalue method, 60
reading GIF files example (bitwise operators), 123-127
read-only computed properties, 7
read-only stored properties, creating, 133
reference types
classes as, 68
closures as, 98-99
value types versus, 69
removeAllActions() method, 181
removeAtIndex() method, 34
removeFromParent() method, 183
removeLast() method, 34
removeValueForKey() method, 37
removing. *See also* emptying
elements from arrays, 34-36
keys from dictionaries, 38
sample code in games, 85
values from dictionaries, 37-38
ResizeFill scale mode, 82, 84
resizeToWidth() method, 181
resting property, 203
restitution, 198
return types for functions, 45-46
multiple return values, 46-47
reverse() method, 202
Ruby, comparison with Swift, 5
running games, 77

S

sample code. *See* listings
scale modes, changing, 82-84
scaling scenes, 82-84
scenes
point system for sizing, 86-89
scaling, 82-84
screen
listening for taps, 181-182
loading tiles, 171-173
self keyword, 65, 72
semicolon (;), 6
SequenceType protocol, 162
setters, 6-7
in dictionaries, 111
as properties, 131-133
for subscripts, 110
setting anchor points, 91-92
shifting bits, 118
sibling order, ignoring, 81-82
signed types, unsigned versus, 119
single-view applications, 206-207
sizing scenes, 86-89
SKAction class, 76, 180-185
SKLabelNode class, 77
SKNode class, 75-76
SKPhysicsBody class, 178-179

SKPhysicsContactDelegate, 176
.sks files, 164-165
SKScene class, 78-79
SKSceneScaleMode class, 82-84
SKShapeNode class, 77-78
SKSpriteNode class, 76-77
SKView class, 79-81
sort() function, as closure, 95-97
sorting custom classes, 97-98
splat operator, 159
SpriteKit, 75
base of classes, finding, 78-79
example game creation
animation with SKAction class, 180-185
collision detection, 176-180
creating game world, 165-175
description of example game, 163
files in GameScene class, 164-165
initial game code, 164
GameViewController class, 79-84
changing scale mode, 82-84
ignoring sibling order, 81-82
SKView class and, 79-81
physics-based games, 187-188
adding levels, 189-190
animation of ball, 197-198
asset library, 189
collision detection, 190-191, 199-203
creating game world, 190-197
SKLabelNode class, 77
SKNode class, 75-76
SKShapeNode class, 77-78
SKSpriteNode class, 77
SpriteKit scene editor (GUI), 164-165
sprites. *See* SpriteKit
square-brackets notation ([])
array elements, accessing, 33-34
dictionaries, accessing, 37
dictionary values, setting, 37
stage (in games), 75
startStopTapped() method, 214
static bodies
described, 192
dynamic bodies versus, 180
static keyword, 71
stored properties, 131
storing images in atlases, 170-171
Storyboard, 205
strings
accessing via for loops, 21
concatenating, 11
converting Int type to, 11
variables in, 10-11
strong reference cycles, 100-102
in closures, 104-106
unowned keyword, 102-104
weak keyword, 102
strong references, 100
structs
classes versus, 57, 61-62
declaring, 62-63
methods
defining, 63-64
mutating, 65
multiple initializers, 66-68
self keyword, 65
as value types, 64
when to use, 68-69
StudySessionManager class, 216

subclasses, 72-74
subscript keyword, 110
subscripts, 109
array elements, accessing, 33-34, 110-111
classes, accessing, 110
declaring, 110-113
dictionaries, accessing, 37, 111
dictionary values, setting, 37, 111
in multidimensional arrays, 111-113
super keyword, 73
superclasses, 72-74
Swift
comparison with other languages, 5-6
described, 2
switch statements, 25-28
syntax. *See* declaring

T

T type, 95-96, 154-155
tables, updating, 216-218
tableView() method, 216-218
tapping screen, listening for, 181-182
ternary operator, 122
testing
optionals, 12-13
values for presence in dictionaries, 38-39
texture atlases, storing images, 170-171
tilde (~), bitwise NOT, 114-115
Tile class, 169
tiles, loading on screen, 171-173
time, formatting, 218
timeIntervalSinceReferenceDate() method, 184
timers, global variables for, 212-213
timerTick() method, 215
trailing closures, 106-107
troubleshooting EXC_BAD_INSTRUCTION error, 13
tuples, 15-16
accessing, 15
as function return types, 46-47
named values, 46-47
in switch statements, 26-27
type casting, 7
type methods, 71-72
in protocol declarations, 135
type safety, 8
typealiases, 17-18
typedef keyword, 57
types
in arrays, 31
mixing, 32-33
associated types, 157-158
checking, 9
converting, 10, 17
Boolean type, 23-24
Int type to String type, 11
custom types, creating with generics, 155-157
declaring, 8
functions as, 51-52
generics. *See* generics
inference, 8-10, 16-17
initializers, 10, 17
number types, 8-9, 16-17
optionals. *See* optionals
protocols as, 138-139
typealiases, 17-18

U

UIKit, 205
constraints, adding, 209-211
single-view applications, 206-207
Storyboard, 205
user interface
connecting to code, 211-212
creating, 208-209
writing code for, 212-218
UInt type, 9, 16
bitwise operations, 119
UITableViewDataSource protocol, 213
UITableViewDelegate protocol, 213
unary operators, 121-122
underscore (_)
external function parameters, 48
in numbers, 10
Unicode characters in variable names, 6
unowned keyword, 102-104
unsigned types, signed versus, 119
unwrapping optionals, 12-13
implicitly unwrapping, 14-15
updateValue() method, 37
updating tables, 216-218
user interface
connecting to code, 211-212
constraints, adding, 209-211
creating for apps, 208-209
writing code for, 212-218

V

value binding, 12-13
value types
reference types versus, 69
structs as, 64
value underflow/overflow, 119-120
values. *See also* elements
accessing in dictionaries, 37-39, 111
associated values in enums, 59-60
inserting in dictionaries, 37
iterating over in dictionaries, 37-38
naming in tuples, 46-47
raw values in enums, 60-61
removing from dictionaries, 37-38
testing for presence in dictionaries, 38-39
values function, 38
var keyword, 6, 33, 131
variables
declaring, 6
as computed properties, 6-7
in structs, 63
global variables for timers, 212-213
lazy keyword, 134
naming, Unicode characters, 6
optionals. *See* optionals
printing, 14
in strings, 10-11
tuples. *See* tuples
types
checking, 9
converting, 10, 17
inference, 8-10, 16-17
initializers, 10, 17
number types, 8-9, 16-17
typealiases, 17-18
variadic parameters, 49-50
view controllers, 206-207

ViewController class, 206
viewDidLoad() method, 213
views. *See* user interface

W

weak keyword, 102
where keyword, 28, 158-162
while loops, 22-23
willSet() function, 70
writing. *See* declaring

X

Xcode, creating projects, 76-77, 163-164, 188-189
XOR operator, bitwise operations, 117