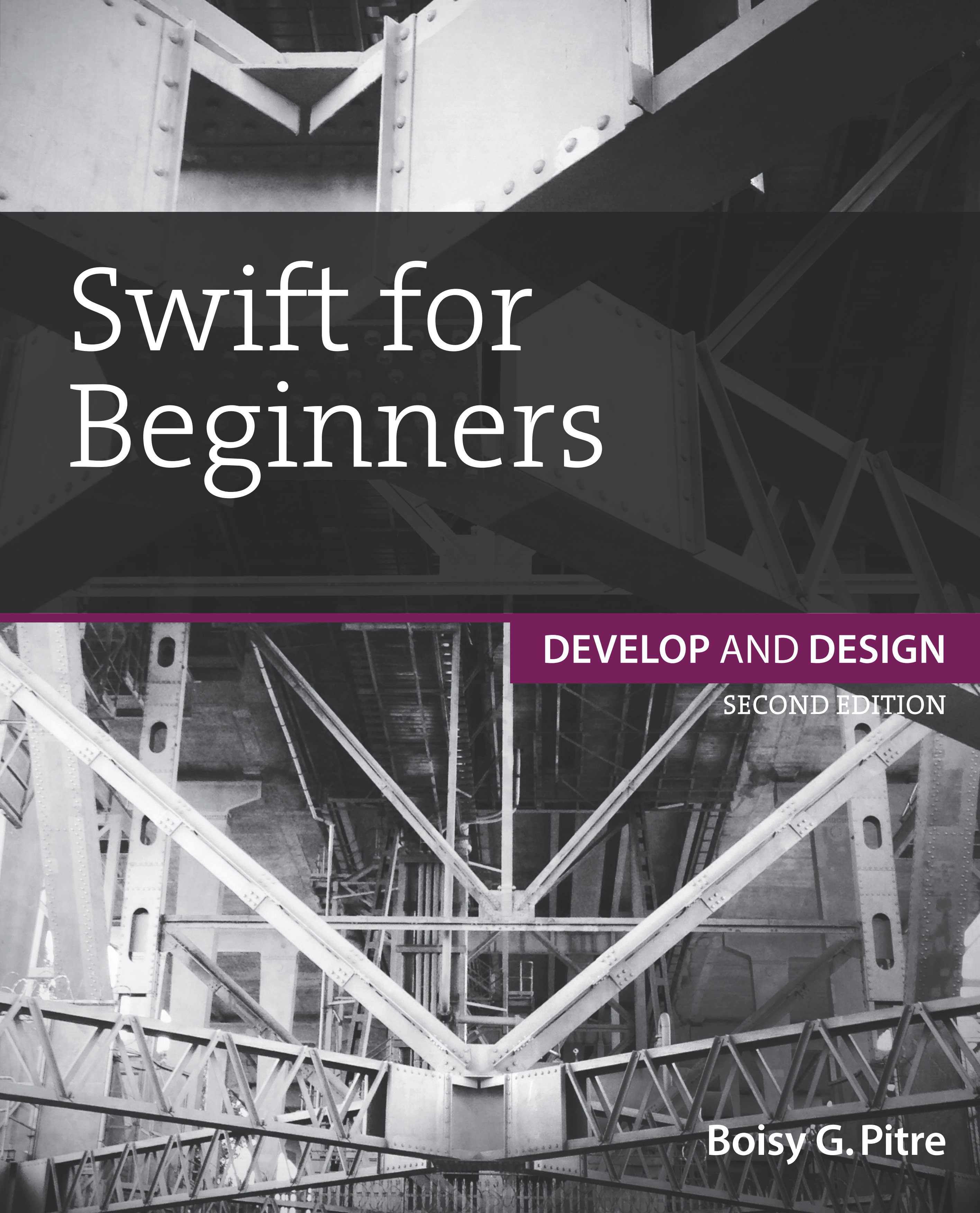


Swift for Beginners

DEVELOP AND **DESIGN**

SECOND EDITION

Boisy G. Pitre

Swift for Beginners

SECOND EDITION

DEVELOP AND DESIGN

Boisy G. Pitre

PEACHPIT PRESS
WWW.PEACHPIT.COM

Swift for Beginners: Develop and Design, Second Edition

Boisy G. Pitre

Peachpit Press
www.peachpit.com

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

Peachpit Press is a division of Pearson Education.

Editor: Connie Jeung-Mills
Production editor: David Van Ness
Development editor: Robyn G. Thomas
Copyeditor and proofreader: Scout Festa
Technical editor: Steve Phillips
Compositor: Danielle Foster
Indexer: Valerie Haynes Perry
Cover design: Aren Straiger
Interior design: Mimi Heft

ISBN-13: 978-0-13-428977-9
ISBN-10: 0-13-428977-3

9 8 7 6 5 4 3 2 1

Printed and bound in the United States of America

To the girls: Toni, Hope, Heidi, Lillian, Sophie, and Belle

This page intentionally left blank

ACKNOWLEDGMENTS

When Peachpit's executive editor Cliff Colby approached me about writing a second edition of *Swift for Beginners*, I readily agreed for several reasons. First, Apple has continued to evolve the Swift language to the point where a book update was necessary—the first edition was already outdated with respect to Swift 2 language enhancements and Xcode improvements. Second, I was eager to work with the same great team of people who were part of the first edition.

Shortly after the project started, Cliff left Peachpit and moved on to another adventure, but not before introducing me to Connie Jeung-Mills, who took over as executive editor. She brought together the original team from the first edition: Robyn Thomas as editor and Steve Phillips as technical editor. Rounding out the team was an addition, Scout Festa, who provided additional editorial support. Each one of them was indispensable and critical to the process, and I want to thank them for their assistance.

On the technical side, I continue to draw inspiration from the works of a number of friends who are authors in the iOS and Mac OS developer community: Chris Adamson, Bill Cheeseman, James Dempsey, Bill Dudney, Daniel Steinberg, and Richard Warren. Thanks go to MacTech's Ed Marczak and Neil Ticktin, as well as CocoaConf maestro Dave Klein, for the writing and speaking opportunities that they have provided me at those venues. My friends at Dave et Ray's Camp Jam/Supper Club always serve as inspiration for several of the coding examples I used in this edition. I also would like to thank Troy Deville for contributing the source code for his game Downhill Challenge.

Thanks also go to the minds at Apple for creating and enhancing Swift, currently in its second major release. The language has solidified since its introduction just over a year ago, and has already reached a popularity that is uncharacteristic for a computer language so young.

Lastly, my family and my wife, Toni, deserve special mention for the patience and encouragement they've shown while I worked on yet another book.

This page intentionally left blank

ABOUT THE AUTHOR

Boisy G. Pitre is Mobile Visionary and lead iOS developer at Affectiva, the leading emotion technology company and a spin-off of the MIT Media Lab. His work there has led to the creation of the first mobile SDK for delivering emotions to mobile devices. Prior to that he was a member of the Mac Products Group at Nuance Communications, where he worked with a team of developers on Dragon Dictate.

He also owns Tee-Boy, a software company focusing on Mac and iOS applications for the weather and data-acquisition markets, and he has authored the monthly "Developer to Developer" column in *MacTech* magazine.

Along with Bill Loguidice, Boisy co-authored the book *CoCo: The Colorful History of Tandy's Underdog Computer* (2013), published by Taylor & Francis.

Boisy holds a Master of Science in Computer Science from the University of Louisiana at Lafayette, is working toward his doctorate in computer science, and resides in the quiet countryside of Prairie Ronde, Louisiana. Besides Mac and iOS development, his hobbies and interests include retro-computing, ham radio, vending machine and arcade game restoration, farming, and playing the French music of South Louisiana.

This page intentionally left blank

CONTENTS

Introduction ... xvi
Welcome to Swift ... xviii

SECTION I THE BASICS ... 2

CHAPTER 1 **INTRODUCING SWIFT** ... 4
Evolutionary Yet Revolutionary ... 6
Preparing for Success ... 6
Tools of the Trade ... 7
Interacting with Swift ... 7
Ready, Set... ... 8
Diving into Swift ... 9
Help and Quit ... 10
Hello, World! ... 10
The Power of Declaration ... 11
Constants Are Consistent ... 13
This Thing Called a Type ... 15
Testing the Limits ... 16
Can a Leopard Change Its Stripes? ... 16
Being Explicit ... 18
Strings and Things ... 19
Stringing Things Together ... 19
Characters Have Character ... 20
Math and More ... 21
Expressions ... 22
Mixing Numeric Types ... 22
Numeric Representations ... 23
True or False ... 24
The Result ... 24
Printing Made Easy ... 26
Using Aliases ... 27
Grouping Data with Tuples ... 28
Optionals ... 29
Summary ... 31

CHAPTER 2 **WORKING WITH COLLECTIONS** 32
The Candy Jar 34
Birds of a Feather... 37
Extending the Array 38
Replacing and Removing Values 39
Inserting Values at a Specific Location 40
Combining Arrays 41
The Dictionary 42
Looking Up an Entry 43
Adding an Entry 45
Updating an Entry 46
Removing an Entry 46
Arrays of Arrays? 47
Starting from Scratch 50
The Empty Array 50
The Empty Dictionary 51
Iterating Collections 52
Array Iteration 52
Dictionary Iteration 54
Summary 55

CHAPTER 3 **TAKING CONTROL** 56
For What It's Worth 58
Counting on It 58
Inclusive or Exclusive? 59
For Old Time's Sake 61
Writing Shorthand 62
It's Time to Play 63
Making Decisions 66
The Decisiveness of "If" 66
When One Choice Is Not Enough 70
Switching Things Around 72
While You Were Away... 75
Inspecting Your Code 77
Give Me a Break! 80
Summary 81

CHAPTER 4 **WRITING FUNCTIONS AND CLOSURES** **82**

The Function 84

Coding the Function in Swift *84*

Exercising the Function *86*

More Than Just Numbers *87*

Parameters Ad Nauseam *89*

Functions Fly First Class *92*

Throw Me a Function, Mister *94*

A Function in a Function in a... .... *96*

Default Parameters *98*

What's in a Name? *100*

When It's Good Enough *102*

To Use or Not to Use? *102*

Don't Change My Parameters! *103*

The Ins and Outs *105*

Bringing Closure 106

Summing It Up 109

Stay Classy 109

CHAPTER 5 **ORGANIZING WITH CLASSES AND STRUCTURES** **110**

Objects Are Everywhere 112

Swift Objects Are Classy 113

Knock, Knock *114*

Let There Be Objects! *115*

Opening and Closing the Door *116*

Locking and Unlocking the Door *117*

Examining the Properties *120*

Door Diversity *120*

Painting the Door *123*

Inheritance 124

Modeling the Base Class *125*

Creating the Subclasses *128*

Instantiating the Subclass *130*

Convenience Initializers *136*

Enumerations 138

Structural Integrity 141

Value Types vs. Reference Types 143

Looking Back, Looking Ahead 145

CHAPTER 6 **FORMALIZING WITH PROTOCOLS AND EXTENSIONS** **146**
Following Protocol 148
Class or Protocol? *148*
More Than Just Methods *151*
Adopting Multiple Protocols *153*
Protocols Can Inherit, Too *155*
Delegation *156*
Extending With Extensions 159
Extending Basic Types *161*
Using Closures in Extensions *166*
Summary 167

SECTION II **DEVELOPING WITH SWIFT** **168**

CHAPTER 7 **WORKING WITH XCODE** **170**
Xcode's Pedigree 172
Creating Your First Swift Project 173
Diving Down 174
Interacting with the Project Window *176*
It's Alive! *178*
Piquing Your Interest 178
Making Room *179*
Building the UI *180*
Tidying Up *182*
Class Time *183*
Hooking It Up *188*
You Made an App! 189

CHAPTER 8 **MAKING A BETTER APP** **190**
It's the Little Things 192
Show Me the Money *192*
Remember the Optional? *195*
Unwrapping Optionals *195*
Looking Better *196*
Formatting: A Different Technique *196*
Compounding 201
Hooking Things Up *202*
Testing Your Work *205*

When Things Go Wrong 205
Where's the Bug? *206*
At the Breaking Point *206*
The Confounding Compound *210*
The Value of Testing 210
The Unit Test *211*
Crafting a Test *211*
When Tests Fail *215*
Tests That Always Run *215*
Wrapping Up 217

CHAPTER 9 **GOING MOBILE WITH SWIFT** **218**
In Your Pocket vs. on Your Desk 220
How's Your Memory? 220
Thinking About Gameplay *221*
Designing the UI *221*
Creating the Project 222
Building the User Interface 224
Creating the Buttons *225*
Running in the Simulator *227*
Setting Constraints *228*
The Model-View-Controller 230
Coding the Game 231
The Class *236*
Enumerations *236*
The View Objects *237*
The Model Objects *237*
Overridable Methods *238*
Game Methods *238*
Winning and Losing *242*
Back to the Storyboard 245
Time to Play 247

CHAPTER 10 **BECOMING AN EXPERT** 248
Memory Management in Swift 250
Value vs. Reference 250
The Reference Count 251
Only the Strong Survive 252
Put It in a Letter 253
The Test Code 254
Breaking the Cycle 256
Cycles in Closures 257
Thanks for the Memories 259
Thinking Logically 259
To Be or NOT to Be… 260
Combining with AND 261
One Way OR the Other 261
Generics 263
Overloading Operators 264
Equal vs. Identical 267
Error Handling 268
Throwing an Error 268
Catching the Error 270
Scripting and Swift 272
Creating the Script 273
Setting Permissions 274
Executing the Script 275
Examining How It Works 275
Calling S.O.S. 278
Game Time 279

CHAPTER 11 **HEADING DOWNHILL** 280
Gaming the System 282
GameKit 282
SpriteKit 282
It Starts with an Idea 283
Heading Downhill 283
Social Connectivity 283
Ready, Set… 284

How the Game Plays *284*
Take It for a Spin *285*
Inspecting the Project 287
Classes *288*
Assets *288*
Scenes *289*
Touring the Source 289
The Home Scene *289*
The Game Scene *293*
Game View Controller *298*
Taking It All In *300*
You Did It! 301
Study Apple's Frameworks *301*
Join Apple's Developer Program *301*
Become a Part of the Community *301*
Never Stop Learning *302*
Bon Voyage! *302*

Index 303

INTRODUCTION

Welcome to *Swift for Beginners*! Swift is Apple's new language for developing apps for iOS and Mac OS, and it is destined to become the premier computer language in the mobile and desktop space. As a new computer language, Swift has the allure of a shiny new car—everybody wants to see it up close, kick the tires, and take it for a spin down the road. That's probably why you're reading this book—you've heard about Swift and decided to see what all the fuss is about.

The notion that Swift is an easy language to use and learn certainly has merit, especially when compared to the capable but harder-to-learn programming language it's replacing: Objective-C. Apple has long used Objective-C as its language of choice for developing software on its platforms, but that is changing with the introduction of Swift.

WHO IS THIS BOOK FOR?

This book was written with the beginner in mind. In a sense, we're all beginners with Swift because it's such a new language. However, many of those who want to learn Swift as a first or second computer language haven't had any exposure to Objective-C or to the related languages C and C++.

Ideally, the reader will have some understanding and experience with a computer language; even so, the book is styled to appeal to the neophyte who is sufficiently motivated to learn. More experienced developers will probably find the first few chapters to be review material and light reading because the concepts are ubiquitous among many computer languages but nonetheless important to introduce Swift to the beginner.

HOW TO USE THIS BOOK

Like other books of its kind, *Swift for Beginners* is best read from start to finish. The material in subsequent chapters tends to build on the knowledge attained from previous ones. However, with a few exceptions, code examples are confined to a single chapter.

The book is sized to provide a good amount of material, but not so much as to overwhelm the reader. Interspersed in the text are a copious number of screenshots to guide the beginner through the ins and outs of Swift as well as the Xcode tool chain.

HOW YOU WILL LEARN

The best way to learn Swift is to use it, and using Swift is emphasized throughout the book with plenty of code and examples.

Each chapter contains code that builds on the concepts presented. Swift has two interactive environments you will use to test out concepts and gain understanding of the language: the REPL and playgrounds. Later, you'll build two simple but complete apps: a loan calculator for Mac OS and a memory game for iOS. In the final chapter, you will be introduced to the source code for a complete 2D game that uses several Apple gaming technologies.

Swift concepts will be introduced throughout the text—classes, functions, closures, and more, all the way to the very last chapter. You're encouraged to take your time and read each chapter at your leisure, even rereading if necessary, before moving on to the next one.

Source code for all the chapters is available at www.peachpit.com/swiftbeginners2. You can download the code for each chapter, which cuts down considerably on typing; nonetheless, I am a firm believer in typing in the code. By doing so, you gain insight and comprehension you might otherwise miss if you just read along and rely on the downloaded code. Make the time to type in all of the code examples.

For clarity, code and constructs such as class names are displayed in `monospace` font.

Highlighted code identifies the portions of the code that are intended for you to type:

```
  1> let candyJar = ["Peppermints", "Gooey Bears", "Happy Ranchers"]
candyJar: [String] = 3 values {
  [0] = "Peppermints"
  [1] = "Gooey Bears"
  [2] = "Happy Ranchers"
}
  2>
```

Bold code identifies an error returned from the REPL:

```
  8> x = y
repl.swift:8:5: error: cannot assign a value of type 'Double'
→ to a value of type 'Int'
x = y
    ^

  8>
```

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

> **NOTE:** Dictionary keys are not necessarily placed in alphabetical order. Swift will always use the order that is the most efficient for retrieval and access.

WHAT YOU WILL LEARN

Ultimately, this book will show you how to use Swift to express your ideas in code. When you complete the final chapter, you should have a good head start, as well as a solid understanding of what the language offers. Additionally, you'll have the skills to begin writing an app. Both iOS and Mac OS apps are presented as examples in the later chapters.

What this book does not do is provide an all-inclusive, comprehensive compendium on the Swift programming language. Apple's documentation is the best resource for that. Here, the emphasis is primarily on learning the language itself, with various Cocoa and CocoaTouch frameworks introduced to facilitate examples.

WELCOME TO SWIFT

Swift is a fun, new, and easy-to-learn computer language from Apple. With the knowledge you'll gain from this book, you can begin writing apps for iOS and Mac OS. The main tool you'll need to start learning Swift is the Xcode integrated development environment (IDE). Xcode includes the Swift compiler, as well as the iOS and Mac OS software development kits (SDKs) that contain the infrastructure required to support the apps you develop.

THE TECHNOLOGIES

The following technologies are all part of your journey into the Swift language.

SWIFT 2

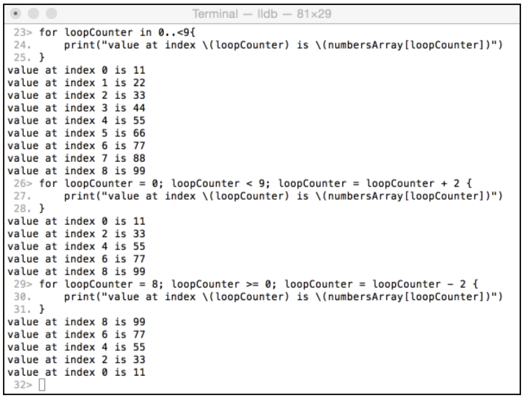
Swift 2 is the language you'll learn in this book. Swift is a modern language designed from the ground up to be easy to learn as well as powerful. It is the language that Apple has chosen to fuel the continued growth of the apps that make up their iOS, watchOS, tvOS, and Mac OS ecosystems.

XCODE 7

Xcode is Apple's premier environment for writing apps. It includes an editor, a debugger, a project manager, and the compiler tool chain needed to take Swift code and turn it into runnable code. You can download Xcode from Apple's Mac App Store.

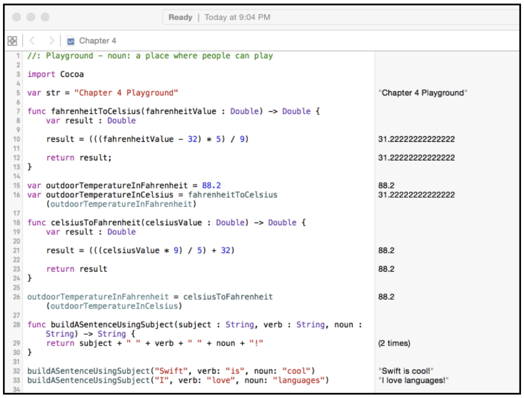
LLVM

Although it works behind the scenes within Xcode, LLVM is the compiler technology that powers the elegance of the Swift language and turns it into the digestible bits and bytes needed by the processors that power Apple devices.



THE REPL

The Read-Eval-Print-Loop (REPL) is a command-line tool you can use to try out Swift code quickly. You run it from the Terminal application on Mac OS.



PLAYGROUNDS

Their interactivity and immediate results make Xcode's playgrounds a great way to try out Swift code as you learn the language.

CHAPTER 4

Writing Functions and Closures

I’ve covered a lot up to this point in the book: variables, constants, dictionaries, arrays, looping constructs, control structures, and the like. You’ve used both the REPL command-line interface and now Xcode’s playgrounds feature to type in code samples and explore the language.

Up to this point, however, you have been limited to mostly experimentation: typing a line or three here and there and observing the results. Now it’s time to get more organized with your code. In this chapter, you’ll learn how to tidy up your Swift code into nice clean reusable components known as functions.

Let’s start this chapter with a fresh new playground file. If you haven’t already done so, launch Xcode and create a new playground by choosing File > New > Playground, and name it **Chapter 4.playground**. You’ll explore this chapter’s concepts with contrived examples in similar fashion to earlier chapters.

THE FUNCTION

Think back to your school years again. This time, remember high school algebra. You were paying attention, weren't you? In that class your teacher introduced the concept of the *function.* In essence, a function in arithmetic parlance is a mathematical formula that takes one or more inputs, performs a calculation, and provides a result, or output.

Mathematical functions have a specific notation. For example, to convert a Fahrenheit temperature value to the equivalent Celsius value, you would express that function in this way:

$$f\left(x\right)=\frac{\left(x-32\right)*5}{9}$$

The important parts of the function are:

- Name: In this case the function's name is *f*.
- Input, or independent variable: Contains the value that will be used in the function. Here it's *x*.
- Expression: Everything to the right of the equals sign.
- Result: Considered to be the value of *f(x)* on the left side of the equals sign.

Functions are written in mathematical notation but can be described in natural language. In English, the sample function could be described as:

> A function whose independent variable is x and whose result is the difference of the independent variable and 32, with the result being multiplied by 5, with the result being divided by 9.

The expression is succinct and tidy. The beauty of functions is that they can be used over and over again to perform work, and all they need to do is be called with a parameter. So how does this relate to Swift? Obviously I wouldn't be talking about functions if they didn't exist in the Swift language. And as you'll see, they can perform not just mathematical calculations but a whole lot more.

CODING THE FUNCTION IN SWIFT

Swift's notation for establishing the existence of a function is a little different than the mathematical one you just saw. In general, the syntax for declaring a Swift function is:

```
func funcName(paramName : type, ...) -> returnType
```

Take a look at an example to help clarify the syntax. **Figure 4.1** shows the code in the Chapter 4.playground file, along with the function defined on lines 7 through 13. This is the function discussed earlier, but now in a notation that the Swift compiler can understand.

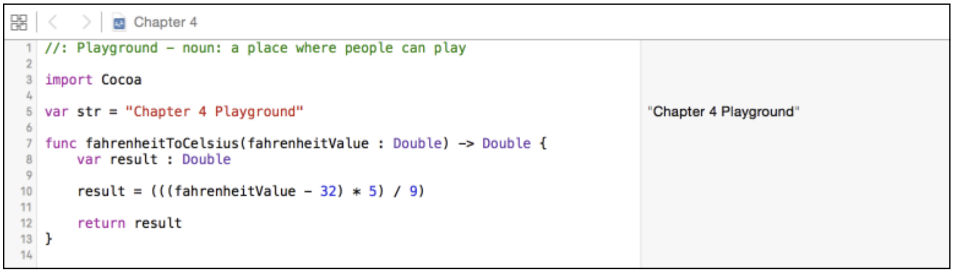


FIGURE 4.1 Temperature conversion as a Swift function

Start by typing in the following code.

```
func fahrenheitToCelsius(fahrenheitValue : Double) -> Double {
  var result : Double

  result = (((fahrenheitValue - 32) * 5) / 9)

  return result
}
```

As you can see on line 7, there is some new syntax to learn. The `func` keyword is Swift's way to declare a function. That is followed by the function name (`fahrenheitToCelsius`), and the independent variable's name, or parameter name, in parentheses. Notice that the `fahrenheitValue` parameter's type is explicitly declared as `Double`.

Following the parameter are the two characters `->`, which denote that this function is returning a value of a type (in this case, a `Double` type), followed by the open curly brace, which indicates the start of the function.

On line 8, you declare a variable of type `Double` named `result`. This will hold the value that will be given back to anyone who calls the function. Notice that it is the same type as the function's return type declared after the `->` on line 7.

The actual mathematical function appears on line 10, with the result of the expression assigned to `result`, the local variable declared in line 8. Finally on line 12, the result is returned to the caller using the `return` keyword. Anytime you wish to exit a function and return to the calling party, you use `return` along with the value being returned.

The Results sidebar doesn't show anything in the area where the function was typed. That's because a function by itself doesn't *do* anything. It has the potential to perform some useful work, but it must be called by a caller. That's what you'll do next.

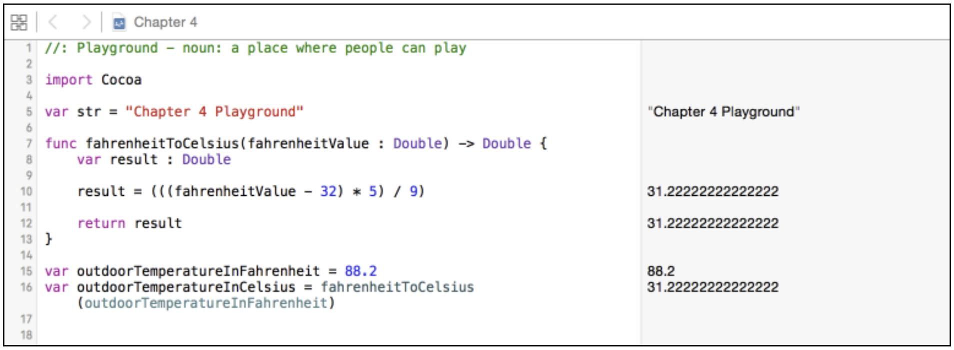


FIGURE 4.2 The result of calling the newly created function

EXERCISING THE FUNCTION

Now it's time to call on the function you just created. Type in the following two lines of code, and pay attention to the Results sidebar in **Figure 4.2**.

```
var outdoorTemperatureInFahrenheit = 88.2
var outdoorTemperatureInCelsius = fahrenheitToCelsius(outdoorTemperature
→ InFahrenheit)
```

On line 15, you've declared a new variable, `outdoorTemperatureInFahrenheit`, and set its value to 88.2 (remember, Swift infers the type in this case as a `Double`). That value is then passed to the function on line 16, where a new variable, `outdoorTemperatureInCelsius`, is declared, and its value is captured as the result of the function.

The Results sidebar shows that 31.222222 (repeating decimal) is the result of the function, and indeed, 31.2 degrees Celsius is equivalent to 88.2 degrees Fahrenheit. Neat, isn't it? You now have a temperature conversion tool right at your fingertips.

Now, here's a little exercise for you to do on your own: Write the inverse method, `celsiusToFahrenheit`, using the following formula for that conversion:

$$f(x) = \frac{x * 9}{5} + 32$$

Go ahead and code it up yourself, but resist the urge to peek ahead. Don't look until you've written the function, and then check your work against the following code and in **Figure 4.3**.

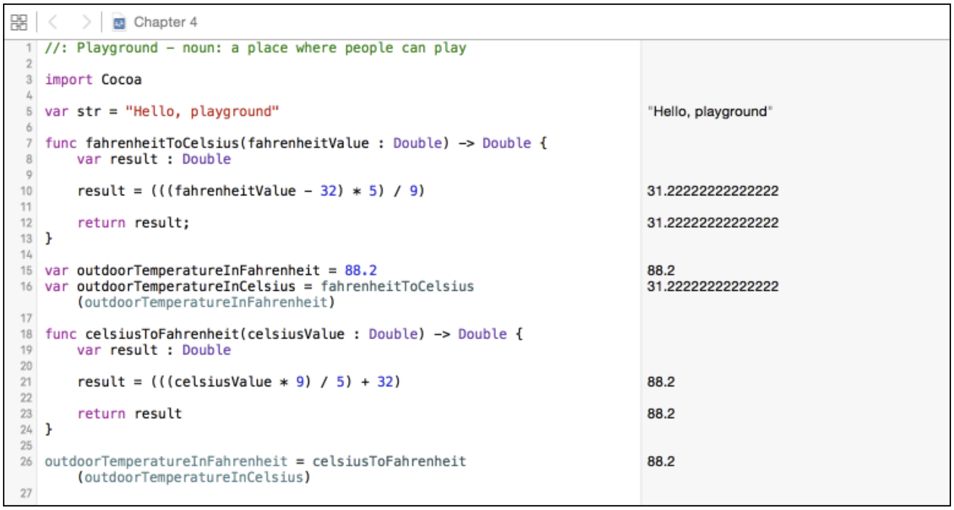


FIGURE 4.3 Declaring the inverse function, `celsiusToFahrenheit`

```
func celsiusToFahrenheit(celsiusValue : Double) -> Double {
  var result : Double

  result = (((celsiusValue * 9) / 5) + 32)

  return result
}

outdoorTemperatureInFahrenheit = celsiusToFahrenheit(outdoorTemperature
→ InCelsius)
```

The inverse function on lines 18 through 24 simply implements the Celsius to Fahrenheit formula and returns the result. Passing in the Celsius value of 31.22222 on line 26, you can see that the result is the original Fahrenheit value, 88.2.

You've just created two functions that do something useful: temperature conversions. Feel free to experiment with other values to see how they change between the two related functions.

MORE THAN JUST NUMBERS

The notion of a function in Swift is more than just the mathematical concept I have discussed. In a broad sense, Swift functions are more flexible and robust in that they can accept more than just one parameter, and even accept types other than numeric ones.

Consider creating a function that takes more than one parameter and returns something other than a `Double` (**Figure 4.4**).

FIGURE 4.4 A multi-parameter function


```
func buildASentenceUsingSubject(subject : String, verb : String, noun : String)
→ -> String {
  return subject + " " + verb + " " + noun + "!"
}

buildASentenceUsingSubject("Swift", verb: "is", noun: "cool")
buildASentenceUsingSubject("I", verb: "love", noun: "languages")
```

After typing in lines 28 through 33, examine your work. On line 28, you declared a new function, `buildASentence`, with not one but three parameters: `subject`, `verb`, and noun, all of which are `String` types. The function also returns a `String` type as well. On line 29, the concatenation of those three parameters, interspersed with spaces to make the sentence readable, is what is returned.

To demonstrate the utility of the function, it is called twice on lines 32 and 33, resulting in the sentences in the Results sidebar.

If you are familiar with the C language and how parameters are passed to functions, the notation on lines 32 and 33 may appear confusing at first. Swift enforces the notion of named parameters on all but the first parameter of a function. The names that were declared in the function on line 28 (verb and noun) are specified on this line right alongside the actual string values.

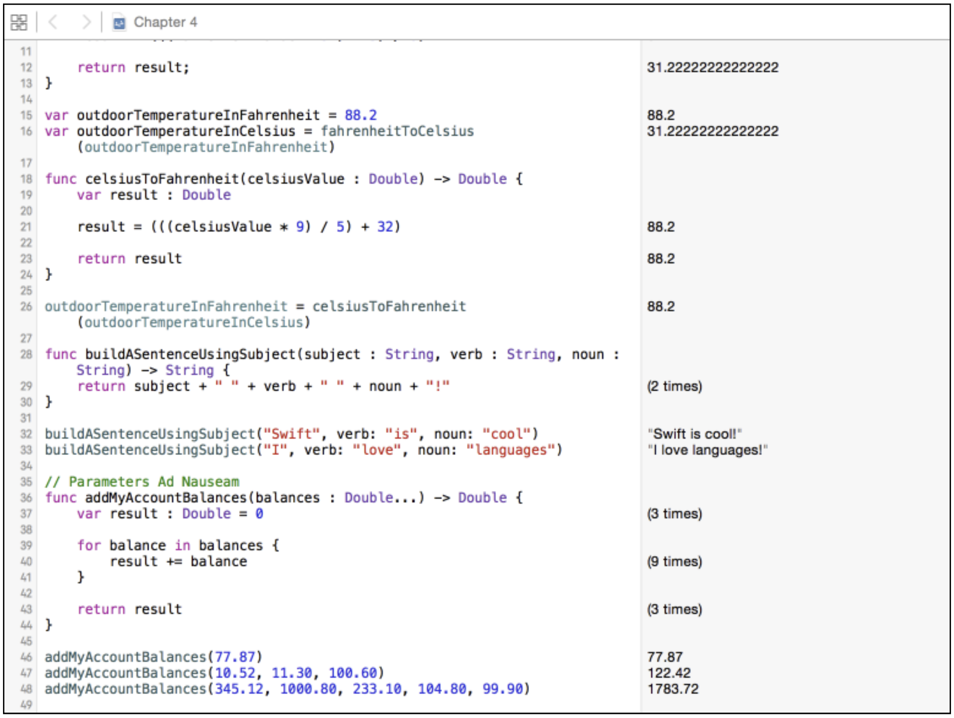


FIGURE 4.5 Variable parameter passing in a function

Swift enforces the notion of named parameters, which is a legacy of Objective-C. Named parameters bring clarity to your source code by documenting exactly what is being passed. From the code, you can clearly see that the verb and noun are the second and third parameters, respectively.

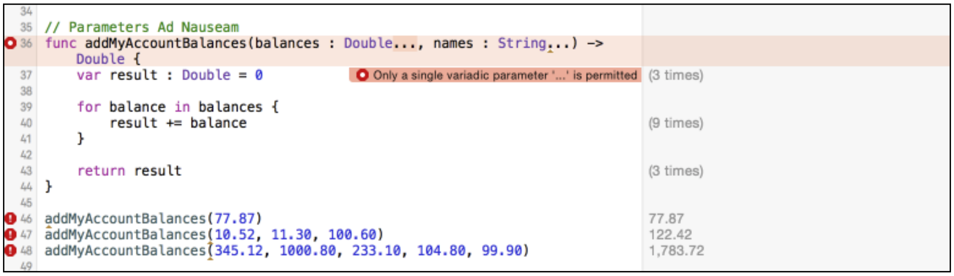
Feel free to replace the parameters with values of your own liking and view the results interactively.

PARAMETERS AD NAUSEAM

Imagine you're writing the next big banking app for the Mac, and you want to create a way to add some arbitrary number of account balances. Something so mundane can be done a number of ways, but you want to write a Swift function to do the addition. The problem is you don't know how many accounts will need to be summed at any given time.

Enter Swift's variable parameter passing notation. It provides you with a way to tell Swift, "I don't know how many parameters I'll need to pass to this function, so accept as many as I will give." Type in the following code, which is shown in action on lines 35 through 48 in **Figure 4.5**.

FIGURE 4.6 Adding additional variable parameters




```
// Parameters Ad Nauseam
func addMyAccountBalances(balances : Double...) -> Double {
  var result : Double = 0

  for balance in balances {
     result += balance
  }

  return result
}

addMyAccountBalances(77.87)
addMyAccountBalances(10.52, 11.30, 100.60)
addMyAccountBalances(345.12, 1000.80, 233.10, 104.80, 99.90)
```

This function's parameter, known as a *variadic parameter*, can represent an unknown number of parameters.

On line 36, your `balances` parameter is declared as a `Double` followed by the ellipsis (`...`) and returns a `Double`. The presence of the ellipsis is the clue: It tells Swift to expect *one or more* parameters of type `Double` when this function is called.

The function is called three times on lines 46 through 48, each with a different number of bank balances. The totals for each appear in the Results sidebar.

You might be tempted to add additional variadic parameters in a function. **Figure 4.6** shows an attempt to extend `addMyAccountBalances` with a second variadic parameter, but it results in a Swift error.

This is a no-no, and Swift will quickly shut you down with an error. Only *one* parameter of a function may contain the ellipsis to indicate a variadic parameter. All other parameters must refer to a single quantity.

Since we're on the theme of bank accounts, add two more functions: one that will find the largest balance in a given list of balances, and another that will find the smallest balance. Type the following code, which is shown on lines 50 through 75 in **Figure 4.7**.

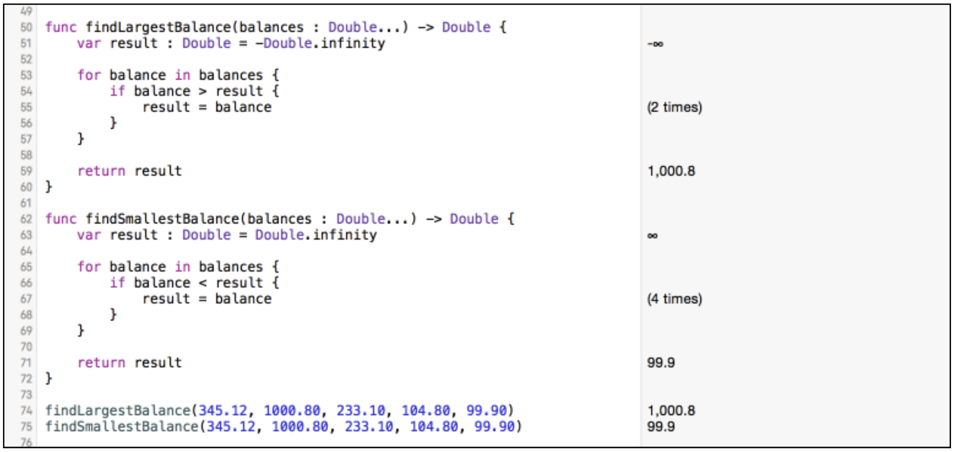


FIGURE 4.7 Functions to find the largest and smallest balance

```
func findLargestBalance(balances : Double...) -> Double {
  var result : Double = -Double.infinity

  for balance in balances {
    if balance > result {
      result = balance
    }
  }

  return result
}

func findSmallestBalance(balances : Double...) -> Double {
  var result : Double = Double.infinity

  for balance in balances {
    if balance < result {
      result = balance
    }
  }

  return result
}

findLargestBalance(345.12, 1000.80, 233.10, 104.80, 99.90)
findSmallestBalance(345.12, 1000.80, 233.10, 104.80, 99.90)
```

Both functions iterate through the parameter list to find the largest and smallest balance. Unless you have an account with plus or minus infinity of your favorite currency, these functions will work well. On lines 74 and 75, both functions are tested with the same balances used earlier, and the Results sidebar confirms their correctness.

FUNCTIONS FLY FIRST CLASS

One of the powerful features of Swift functions is that they are *first-class objects*. Sounds pretty fancy, doesn't it? What that really means is that you can handle a function just like any other value. You can assign a function to a constant, pass a function as a parameter to another function, and even return a function from a function!

To illustrate this idea, consider the act of depositing a check into your bank account, as well as withdrawing an amount. Every Monday, an amount is deposited, and every Friday, another amount is withdrawn. Instead of tying the day directly to the function name of the deposit or withdrawal, use a constant to point to the function for the appropriate day. The code on lines 77 through 94 in **Figure 4.8** provides an example.

```
var account1 = ("State Bank Personal", 1011.10)
var account2 = ("State Bank Business", 24309.63)

func deposit(amount : Double, account : (name : String, balance : Double)) ->
→ (String, Double) {
  let newBalance : Double = account.balance + amount
  return (account.name, newBalance)
}
func withdraw(amount : Double, account : (name : String, balance : Double)) ->
→ (String, Double) {
  var newBalance : Double = account.balance - amount
  return (account.name, newBalance)
}

let mondayTransaction = deposit
let fridayTransaction = withdraw

let mondayBalance = mondayTransaction(300.0, account: account1)
let fridayBalance = fridayTransaction(1200, account: account2)
```

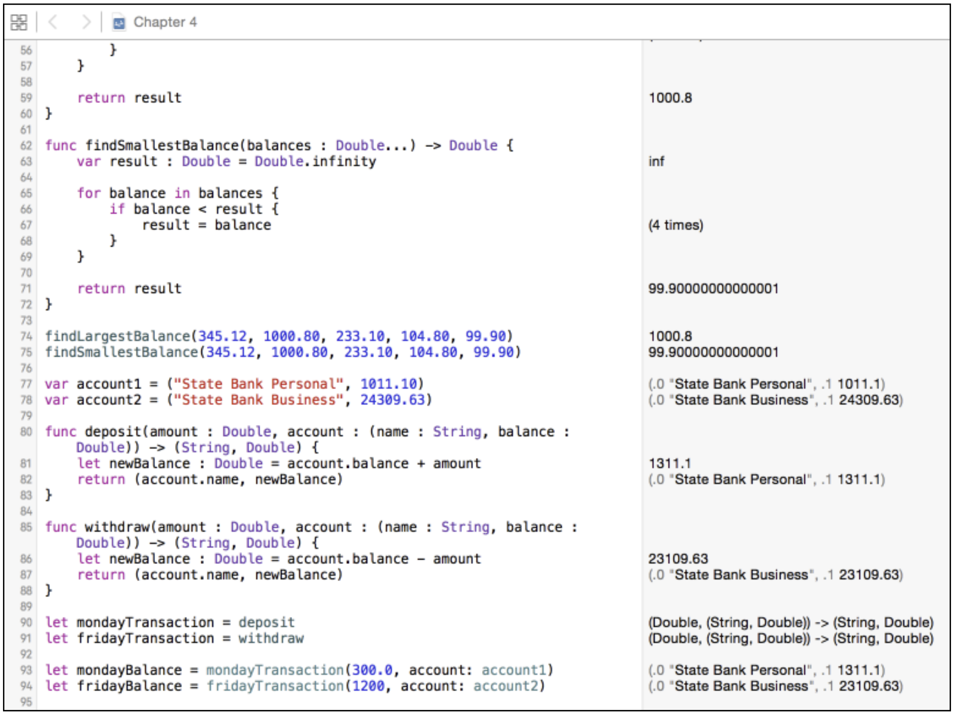


FIGURE 4.8 Demonstrating functions as first-class types

For starters, you create two accounts on lines 77 and 78. Each account is a tuple consisting of an account name and balance.

On line 80, a function named `deposit` is declared, and it takes two parameters: the `amount` (a `Double`) and a tuple named `account`. The tuple has two members: `name`, which is of type `String`, and `balance`, which is a `Double` that represents the funds in that account. The same tuple type is also declared as the return type.

At line 81, a variable named `newBalance` is declared, and its value is assigned the sum of the `balance` member of the `account` tuple and the `amount` variable that is passed. The tuple result is constructed on line 82 and returned.

The function on line 85 is named differently (`withdraw`) but is effectively the same, save for the subtraction that takes place on line 86.

On lines 90 and 91, two new constants are declared and assigned to the functions respectively by name: `deposit` and `withdraw`. Since deposits happen on a Monday, the `mondayTransaction` is assigned the `deposit` function. Likewise, withdrawals are on Friday, and the `fridayTransaction` constant is assigned the `withdraw` function.

Lines 93 and 94 show the results of passing the `account1` and `account2` tuples to the `mondayTransaction` and `fridayTransaction` constants, which are in essence the functions `deposit` and `withdraw`. The Results sidebar bears out the result, and you've just called the two functions by referring to the constants.

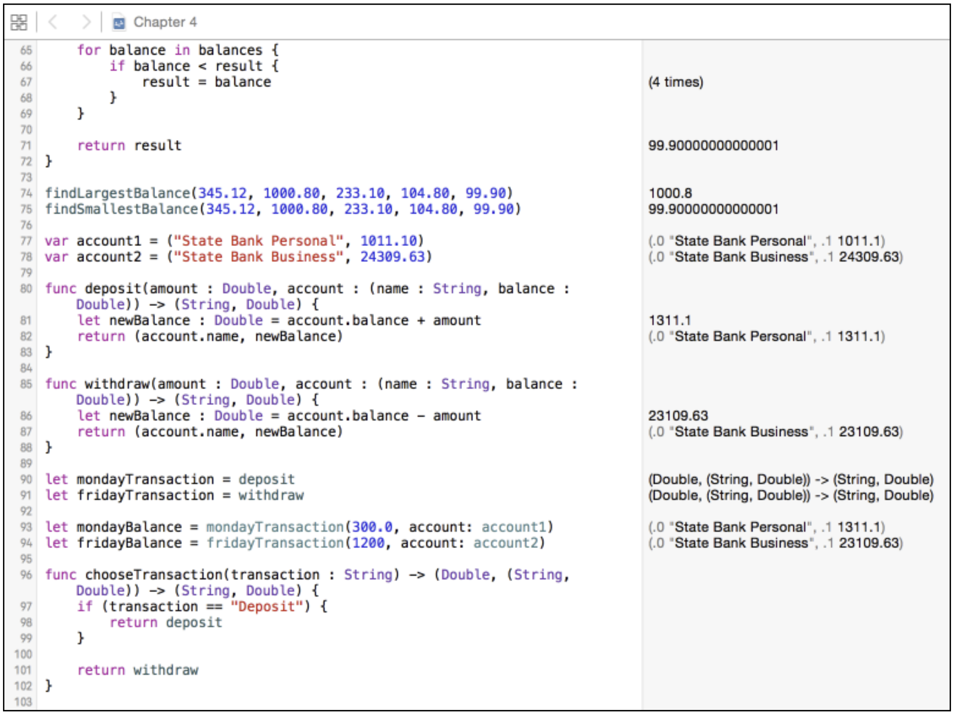


FIGURE 4.9 Returning a function from a function

THROW ME A FUNCTION, MISTER

Just as a function can return an `Int`, `Double`, or `String`, a function can also return another function. Your head starts hurting just thinking about the possibilities, doesn't it? Actually, it's not as hard as it sounds. Check out lines 96 through 102 in **Figure 4.9**.

```
func chooseTransaction(transaction: String) -> (Double, (String, Double)) -> (String, Double) {
  if transaction == "Deposit" {
    return deposit
  }

  return withdraw
}
```

On line 96, the function `chooseTransaction` takes a `String` as a parameter, which it uses to deduce the type of banking transaction. That same function returns a function, which itself takes a `Double`, and a tuple of `String` and `Double`, and returns a tuple of `String` and `Double`. Phew!

```
Chapter 4
72  }
73
74  findLargestBalance(345.12, 1000.80, 233.10, 104.80, 99.90)              1,000.8
75  findSmallestBalance(345.12, 1000.80, 233.10, 104.80, 99.90)             99.9
76
77  var account1 = ("State Bank Personal", 1011.10)                         (.0 "State Bank Personal", .1 1,011.1)
78  var account2 = ("State Bank Business", 24309.63)                        (.0 "State Bank Business", .1 24,309.63)
79
80  func deposit(amount : Double, account : (name : String, balance :
        Double)) -> (String, Double) {
81      let newBalance : Double = account.balance + amount                  (2 times)
82      return (account.name, newBalance)                                   (2 times)
83  }
84
85  func withdraw(amount : Double, account : (name : String, balance :
        Double)) -> (String, Double) {
86      let newBalance : Double = account.balance - amount                  (2 times)
87      return (account.name, newBalance)                                   (2 times)
88  }
89
90  let mondayTransaction = deposit                                         (Double, (String, Double)) -> (String, Double)
91  let fridayTransaction = withdraw                                        (Double, (String, Double)) -> (String, Double)
92
93  let mondayBalance = mondayTransaction(300.0, account: account1)         (.0 "State Bank Personal", .1 1,311.1)
94  let fridayBalance = fridayTransaction(1200, account: account2)          (.0 "State Bank Business", .1 23,109.63)
95
96  func chooseTransaction(transaction : String) -> (Double, (String,
        Double)) -> (String, Double) {
97      if (transaction == "Deposit") {
98          return deposit                                                  (Double, (String, Double)) -> (String, Double)
99      }
100
101     return withdraw                                                     (Double, (String, Double)) -> (String, Double)
102 }
103
104 // option 1: capture the function in a constant and call it
105 let myTransaction = chooseTransaction("Deposit")                        (Double, (String, Double)) -> (String, Double)
106 myTransaction(225.33, account2)                                         (.0 "State Bank Business", .1 24,534.96)
107
108 // option 2: call the function reuslt directly
109 chooseTransaction("Withdraw")(63.17, account1)                          (.0 "State Bank Personal", .1 947.93)
110
```

FIGURE 4.10 Calling the returned function in two different ways

That's a mouthful. Let's take a moment to look at that line more closely and break it down a bit. The line begins with the definition of the function and its sole parameter, `transaction`, followed by the `->` characters indicating the return type:

```
func chooseTransaction(transaction: String) ->
```

After that is the return type, which is a function that takes two parameters—the `Double`, and a tuple of `Double` and `String`—as well as the function return characters `->`:

```
(Double, (String, Double)) ->
```

And finally, the return type of the returned function, a tuple of `String` and `Double`.

What functions did you write that meet these criteria? The `deposit` and `withdraw` functions, of course! Look at lines 80 and 85. Those two functions are bank transactions that were used earlier. Since they are defined as functions that take two parameters (a `Double` and a tuple of `String` and `Double`) and return a tuple of `Double` and `String`, they are appropriate candidates for return values in the `chooseTransaction` function on line 96.

Back to the `chooseTransaction` function: On line 97, the `transaction` parameter, which is a `String`, is compared against the constant string `"Deposit"` and if a match is made, the `deposit` function is returned on line 98; otherwise, the `withdraw` function is returned on line 101.

OK, so you have a function which itself returns one of two possible functions. How do you use it? Do you capture the function in another variable and call it?

Actually, there are two ways this can be done (**Figure 4.10**).

```
// option 1: capture the function in a constant and call it
let myTransaction = chooseTransaction("Deposit")
myTransaction(225.33, account2)

// option 2: call the function result directly
chooseTransaction("Withdraw")(63.17, account1)
```

On line 105 you can see that the returned function for making deposits is captured in the constant `myTransaction`, which is then called on line 106 with `account2` increasing its amount by $225.33.

The alternate style is on line 109. There, the `chooseTransaction` function is being called to gain access to the `withdraw` function. Instead of assigning the result to a constant, however, the returned function is immediately pressed into service with the parameters `63.17` and the first account, `account1`. The results are the same in the Results sidebar: The `withdraw` function is called and the balance is adjusted.

A FUNCTION IN A FUNCTION IN A...

If functions returned by functions and assigned to constants isn't enough of an enigma for you, how about declaring a function inside another function? Yes, such a thing exists. They're known as *nested functions.*

Nested functions are useful when you want to isolate, or hide, specific functionality that doesn't need to be exposed to outer layers. Take, for instance, the code in **Figure 4.11**.

```
// nested function example
func bankVault(passcode : String) -> String {
  func openBankVault(_: Void) -> String {
     return "Vault opened"
  }
  func closeBankVault() -> String {
     return "Vault closed"
  }
  if passcode == "secret" {
     return openBankVault()
  }
  else {
     return closeBankVault()
  }
}

print(bankVault("wrongsecret"))
print(bankVault("secret"))
```

Chapter 4

```
 89
 90 let mondayTransaction = deposit                                        (Double, (String, Double)) -> (String, Double)
 91 let fridayTransaction = withdraw                                       (Double, (String, Double)) -> (String, Double)
 92
 93 let mondayBalance = mondayTransaction(300.0, account: account1)        (.0 "State Bank Personal", .1 1,311.1)
 94 let fridayBalance = fridayTransaction(1200, account: account2)         (.0 "State Bank Business", .1 23,109.63)
 95
 96 func chooseTransaction(transaction : String) -> (Double, (String,
        Double)) -> (String, Double) {
 97     if (transaction == "Deposit") {
 98         return deposit                                                 (Double, (String, Double)) -> (String, Double)
 99     }
100
101     return withdraw                                                    (Double, (String, Double)) -> (String, Double)
102 }
103
104 // option 1: capture the function in a constant and call it
105 let myTransaction = chooseTransaction("Deposit")                       (Double, (String, Double)) -> (String, Double)
106 myTransaction(225.33, account2)                                        (.0 "State Bank Business", .1 24,534.96)
107
108 // option 2: call the function reuslt directly
109 chooseTransaction("Withdraw")(63.17, account1)                         (.0 "State Bank Personal", .1 947.93)
110
111 // nested function example
112 func bankVault(passcode : String) -> String {
113     func openBankVault(_: Void) -> String {
114         return "Vault opened"                                          "Vault opened"
115     }
116     func closeBankVault() -> String {
117         return "Vault closed"                                          "Vault closed"
118     }
119     if (passcode == "secret") {
120         return openBankVault()                                         "Vault opened"
121     }
122     else {
123         return closeBankVault();                                       "Vault closed"
124     }
125 }
126
127 print(bankVault("wrongsecret"))                                        "Vault closed"
128 print(bankVault("secret"))                                             "Vault opened"
129
```

FIGURE 4.11 Nested functions in action

On line 112, a new function, `bankVault`, is defined. It takes a single parameter, `passcode`, which is a `String`, and returns a `String`.

Lines 113 and 116 define two functions inside the `bankVault` function: `openBankVault` and `closeBankVault`. Both of these functions take no parameter and return a `String`.

On line 119, the `passcode` parameter is compared with the string `"secret"` and if a match is made, the bank vault is opened by calling the `openBankVault` function. Otherwise, the bank vault remains closed.

INTO THE VOID

On line 113 you'll notice a new Swift keyword: `Void`. It means exactly what you might think: emptiness. The `Void` keyword is used mostly as a placeholder when declaring empty parameter lists, and is optional in this case. The underscore that precedes it is known as an "unnamed parameter," which is essentially an anonymous variable name. On line 116, you declare the `closeBankVault` function without any parameter, which assumes `Void`. In any case, functions that have no parameters can simply be declared without any parameters, and they're used here only for illustrative purposes. In fact, both function definitions on line 113 and 116 are equivalent for all practical purposes.

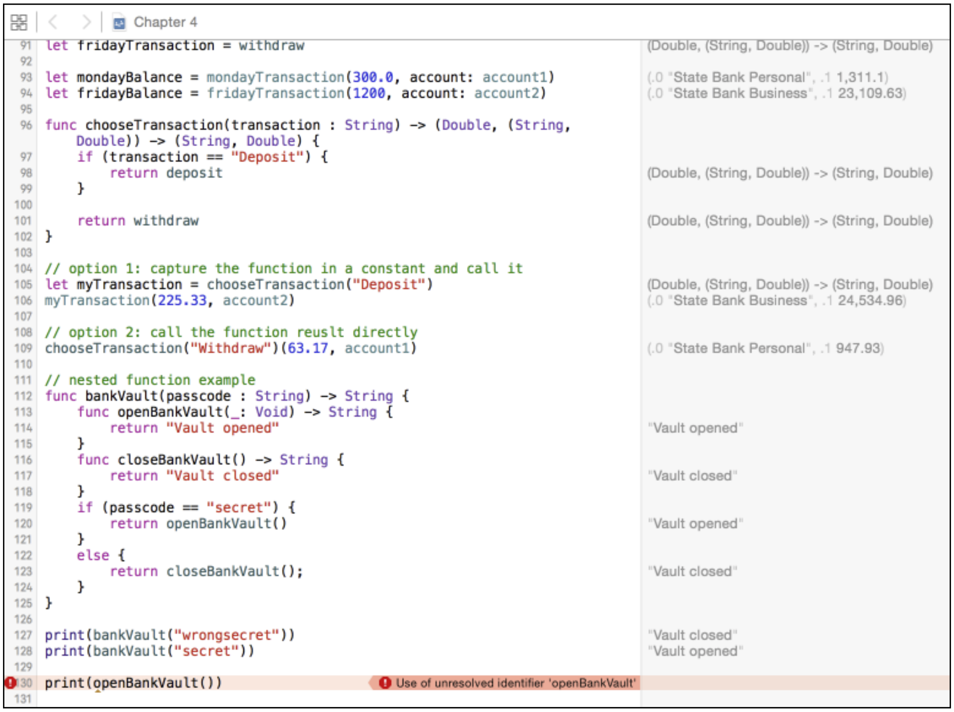


FIGURE 4.12 The result of attempting to call a nested function from a different scope

Lines 127 and 128 show the result of calling the `bankVault` method with an incorrect and correct passcode. What's important to realize is that the `openBankVault` and `closeBankVault` functions are "enclosed" by the `bankVault` function, and are not known outside of that function.

If you were to attempt to call either `openBankVault` or `closeBankVault` outside of the `bankVault` function, you would get an error. That's because those functions are not in *scope*. They are, in effect, hidden by the `bankVault` function and are unable to be called from the outside. **Figure 4.12** illustrates an attempt to call one of these nested functions.

In general, the obvious benefit of nesting functions within functions is that it prevents the unnecessary exposing of functionality. In Figure 4.12, the `bankVault` function is the sole gateway to opening and closing the vault, and the functions that perform the work are isolated within that function. Always consider this when designing functions that are intended to work together.

DEFAULT PARAMETERS

As you've just seen, Swift functions provide a rich area for utility and experimentation. A lot can be done with functions and their parameters to model real-world problems. Functions provide an interesting feature known as *default parameter values*, which allow you to declare functions that have parameters containing a "prefilled" value.

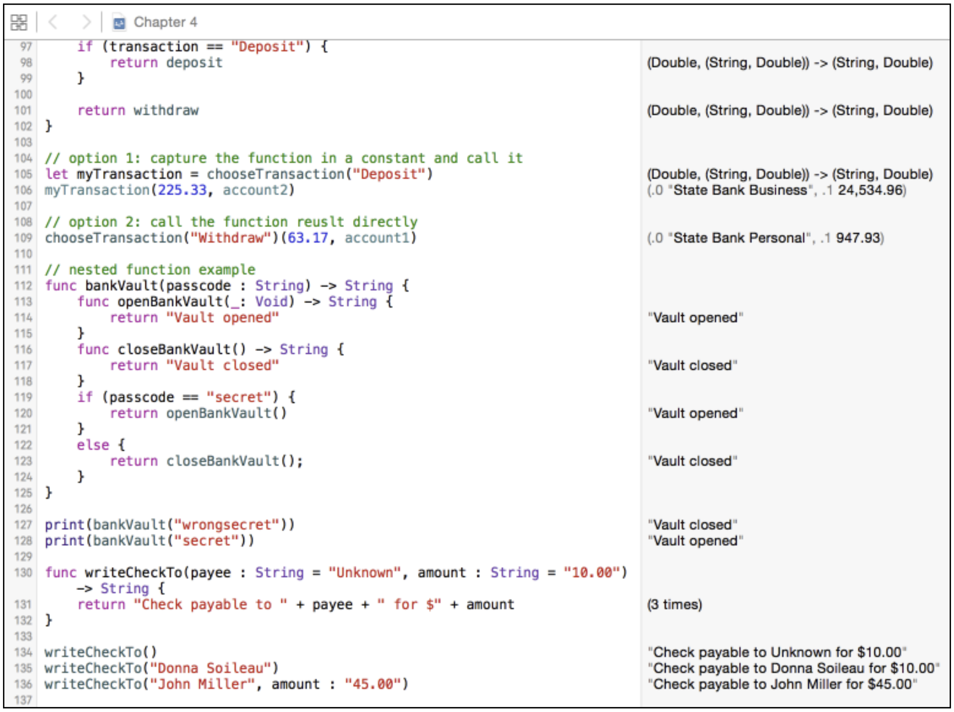


FIGURE 4.13 Using default parameters in a function

Let's say you want to create a function that writes checks. Your function would take two parameters: a payee (the person or business to whom the check is written) and the amount. Of course, in the real world, you always want to know these two pieces of information, but for now, think of a function that would assume a default payee and amount in the event the information wasn't passed.

Figure 4.13 shows such a function on lines 130 through 132. The `writeCheckTo` function takes two `String` parameters, the payee and `amount`, and returns a `String` that is simply a sentence describing how the check is written.

```
func writeCheckTo(payee : String = "Unknown", amount : String = "10.00") ->
→ String {
  return "Check payable to " + payee + " for $" + amount
}

writeCheckTo()
writeCheckTo("Donna Soileau")
writeCheckTo("John Miller", amount : "45.00")
```

Take note of the declaration of the function on line 130:

```
func writeCheckTo(payee : String = "Unknown", amount : String = "10.00") ->
→ String
```

What you haven't seen before now is the assignment of the parameters to actual values (in this case, payee is being set to "Unknown" by default and amount is being set to "10.00"). This is how you can write a function to take default parameters—simply assign the parameter name to a value!

So how do you call this function? Lines 134 through 136 show three different ways:

- Line 134 passes no parameters when calling the function.
- Line 135 passes a single parameter.
- Line 136 passes both parameters, with the second parameter following its parameter name amount.

In the case where no parameters are passed, the default values are used to construct the returned String. In the other two cases, the passed parameter values are used in place of the default values, and you can view the results of the calls in the Results sidebar.

Recall that Swift enforces the requirement that the parameter name must be passed for all but the first parameter. On line 135, only one parameter is used, so the name is not passed:

```
writeCheckTo("Donna Soileau")
```

On line 136, two parameter names are used, and the parameter name is specified prior to the amount string:

```
writeCheckTo("John Miller", amount : "45.00")
```

Default parameters give you the flexibility of using a known value instead of taking the extra effort to pass it explicitly. They're not necessarily applicable for every function out there, but they do come in handy at times.

WHAT'S IN A NAME?

As Swift functions go, declaring them is easy, as you've seen. In some cases, however, what really composes the function name is more than just the text following the keyword func.

As I touched on earlier, each parameter in a Swift function has the parameter name preceding the parameter. This gives additional clarity and description to a function name. Up to this point, you've been told that it must be passed when calling the function. Although it is good practice, it is not entirely necessary. When declaring a function, an *implicit external parameter name* can be notated with an underscore preceding the parameter name. Consider another check writing function in **Figure 4.14**, lines 138 through 140.

```
func writeCheckFrom(payer : String, _ payee : String, _ amount : Double) ->
→ String {
  return "Check payable from \(payer) to \(payee) for $\(amount)"
}

writeCheckFrom("Dave Johnson", "Coz Fontenot", 1_000.0)
```

```
137
138 func writeCheckFrom(payer : String, _ payee : String, _ amount : Double)
        ->String {
139     return "Check payable from \(payer) to \(payee) for $\(amount)"
140 }
141
142 writeCheckFrom("Dave Johnson", "Coz Fontenot", 1_000.0)
143
```

```
"Check payable from Dave Johnson to Coz Fontenot for $1000.0"
"Check payable from Dave Johnson to Coz Fontenot for $1000.0"
```

FIGURE 4.14 A function with an implicit external parameter name

```
143
144 func writeBetterCheckFrom(payer : String, payee : String, amount :
        Double) -> String {
145     return "Check payable from \(payer) to \(payee) for $\(amount)"
146 }
147
148 writeBetterCheckFrom("Fred Charlie", payee: "Ryan Hanks", amount:
        1350.0)
149
```

```
"Check payable from Fred Charlie to Ryan Hanks for $1350.0"
"Check payable from Fred Charlie to Ryan Hanks for $1350.0"
```

FIGURE 4.15 A function called with parameter names

This function is different from the earlier check writing function on lines 130 through 132 in two ways:

- An underscore and a space precede the parameters named payee and amount
- There are no default parameters

On line 142, the new `writeCheckFrom` function is called with three parameters: two `String` values and a `Double` value. From the name of the function, its purpose is clearly to write a check. When writing a check, you need to know several things: who the check is being written for, who is writing the check, and the amount the check is for. A good guess is that the `Double` parameter is the amount, which is a number. But without actually looking at the function declaration itself, how would you know what the two `String` parameters actually mean? Even if you were to deduce that they are the payer and payee, how do you know which is which, and in which order to pass the parameters?

Swift's default behavior of insisting on the use of parameter names solves this problem and makes the intent of your code easier to understand; it makes very clear to anyone reading the calling function what the intention is and the purpose of each parameter. **Figure 4.15** illustrates this.

```
func writeBetterCheckFrom(payer : String, payee : String, amount : Double) -> String {
  return "Check payable from \(payer) to \(payee) for $\(amount)"
}

writeBetterCheckFrom("Fred Charlie", payee : "Ryan Hanks", amount : 1350.0)
```

On line 144, you declare a function, `writeBetterCheckFrom`, which takes the same number of parameters as the function on line 138. However, each of the parameters in the new function omits the underscore.

The extra bit of typing pays off when the `writeBetterCheckFrom` function is called. Looking at that line of code alone, the order of the parameters and what they indicate is clear: Write a check *from* Fred Charlie *to* Ryan Hanks for a *total* of $1350.

WHEN IT'S GOOD ENOUGH

Parameter names bring clarity to functions, as you've just seen. In addition, Swift allows *external parameter names* to decorate a function declaration. This can be useful if you want to bring additional clarity to your function.

Line 150 of **Figure 4.16** shows this in action. The new method, `writeBestCheck` has dropped the `From` in the name. Instead, it has moved to the first parameter as an external parameter name. Other external parameter names in this function declaration are `to` and `total`.

On line 154, the parameter names are used as external parameter names to call the function, and the use of those names clearly shows what the function's purpose and parameter order is: a check written *from* Bart Stewart *to* Alan Lafleur for a *total* of $101. Note that when using external parameter names, the first parameter also requires the parameter name to be passed. This is different from what you saw earlier when your earlier functions weren't using external parameter names.

```
func writeBestCheck(from payer : String, to payee : String,
→ total amount : Double) -> String {
  return "Check payable from \(payer) to \(payee) for $\(amount)"
}

writeBestCheck(from: "Bart Stewart", to: "Alan Lafleur", total: 101.0)
```

TO USE OR NOT TO USE?

Parameter names bring clarity to functions, but they also require more typing on the part of the coder who uses your functions. Since they are optional parts of a function's declaration, when should you use them?

In general, if the function in question can benefit from the additional clarity of having parameter names provided for each parameter, by all means use them. The check writing example is such a case. Avoid parameter ambiguity in the cases where it might exist. On the other hand, if you're creating a function that just adds two numbers (see lines 156 through 160 in **Figure 4.17**), parameter names add little to nothing of value for the caller. You can just use the underscore (recall implicit external parameter names) and avoid passing the parameter name altogether.

```
func addTwoNumbers(number1 : Double, _ number2 : Double) -> Double {
  return number1 + number2
}

addTwoNumbers(33.1, 12.2)
```

```
func writeBestCheck(from payer : String, to payee : String, total
    amount : Double) -> String {
    return "Check payable from \(payer) to \(payee) for $\(amount)"      "Check payable from Bart Stewart to Alan Lafleur for $101.0"
}

writeBestCheck(from: "Bart Stewart", to: "Alan Lafleur", total: 101.0)      "Check payable from Bart Stewart to Alan Lafleur for $101.0"
```

FIGURE 4.16 Using the external parameter name syntax

```
func addTwoNumbers(number1 : Double, _ number2 : Double) -> Double {
    return number1 + number2                                              45.3
}

addTwoNumbers(33.1, 12.2)                                                 45.3
```

FIGURE 4.17 When parameter names are not necessary

```
161
162 func cashCheck(from : String, to : String, total : Double) -> String {
163     if to == "Cash" {
164         to = from                          Cannot assign to value: 'to' is a 'let' constant
165     }
166     return "Check payable from \(from) to \(to) for $\(total) has been
            cashed"
167 }
168
169 cashCheck("Jason Guillory", to: "Cash", total: 103.0)
170
```

FIGURE 4.18 Assigning a value to a parameter results in an error.

DON'T CHANGE MY PARAMETERS!

Functions are prohibited from changing the values of parameters passed to them, because parameters are passed as constants and not variables. Consider the function `cashCheck` on lines 162 through 169 in **Figure 4.18**.

```
func cashCheck(from : String, to : String, total : Double) -> String {
  if to == "Cash" {
    to = from
  }
  return "Check payable from \(from) to \(to) for $\(total) has been cashed"
}

cashCheck("Jason Guillory", to: "Cash", total: 103.00)
```

The function takes the same parameters as your earlier check writing function: who the check is from, who the check is to, and the total. On line 163, the `to` variable is checked for the value `"Cash"` and if it is equal, it is reassigned the contents of the variable `from`. The rationale here is that if you are writing a check to "Cash," you're essentially writing it to yourself.

Notice the error: `Cannot assign to value: 'to' is a 'let' constant`. Swift is saying that the parameter `to` is a constant, and since constants cannot change their values once assigned, this is prohibited and results in an error.

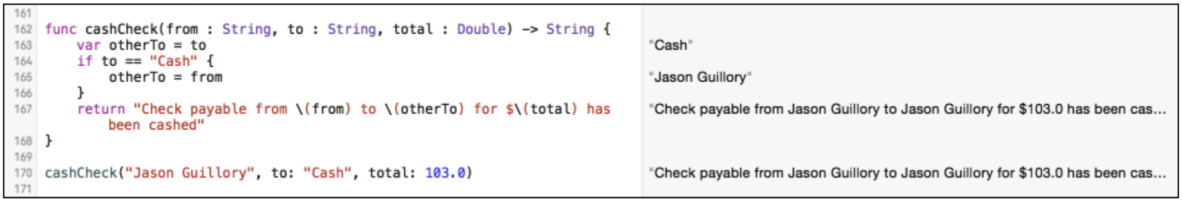


FIGURE 4.19 A potential workaround to the parameter change problem

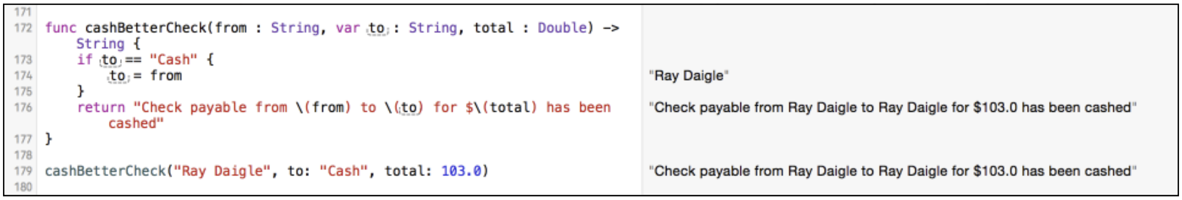


FIGURE 4.20 Using variable parameters to allow modifications

To get around this error, you could create a temporary variable, as done in **Figure 4.19**. Here, a new variable named `otherTo` is declared on line 163 and assigned to the `to` variable, and then possibly to the `from` variable on line 165, assuming the condition on line 164 is met. This is clearly acceptable and works fine for your purposes, but Swift gives you a better way.

With a `var` declaration on a parameter, you can tell Swift that the parameter is intended to be variable and can change within the function. All you need to do is add the keyword before the parameter name (or external parameter name in case you have one of those). **Figure 4.20** shows a second function, `cashBetterCheck`, which declares the `to` parameter as a variable parameter. Now the code inside the function can modify the `to` variable without receiving an error from Swift, and the output is identical to the workaround function above it.

```
func cashBetterCheck(from : String, var to : String, total : Double) ->
→ String {
  if to == "Cash" {
    to = from
  }
    return "Check payable from \(from) to \(to) for $\(total) has been cashed"
}

cashBetterCheck("Ray Daigle", to: "Cash", total: 103.00)
```

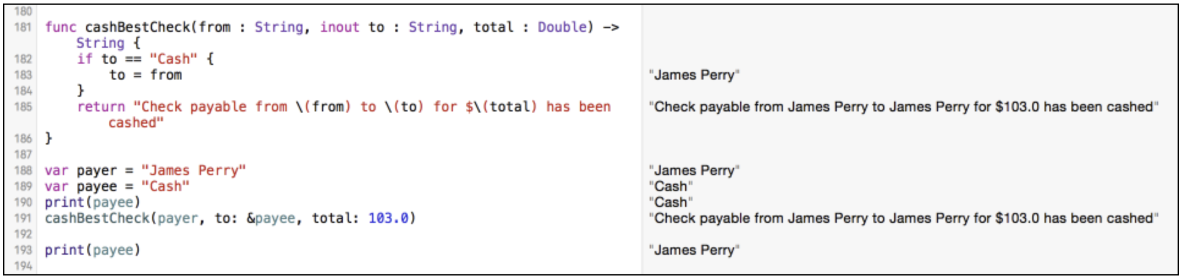


FIGURE 4.21 Using the `inout` keyword to establish a modifiable parameter

THE INS AND OUTS

As you've just seen, a function can be declared to modify the contents of one or more of its passed variables. The modification happens inside the function itself, and the change is not reflected back to the caller.

Sometimes having a function change the value of a passed parameter so that its new value is reflected back to the caller is desirable. For example, in the `cashBetterCheck` function on lines 172 through 177, having the caller know that the `to` variable has changed to a new value would be advantageous. Right now, that function's modification of the variable is not reflected back to the caller. Let's see how to do this in **Figure 4.21** using Swift's `inout` keyword.

```
func cashBestCheck(from : String, inout to : String, total : Double) ->
→ String {
  if to == "Cash" {
     to = from
  }
  return "Check payable from \(from) to \(to) for $\(total) has been cashed"
}

var payer = "James Perry"
var payee = "Cash"
print(payee)
cashBestCheck(payer, to: &payee, total: 103.00)

print(payee)
```

Lines 181 through 186 define the `cashBestCheck` function, which is virtually identical to the `cashBetterCheck` function on line 172, except the second parameter `to` is no longer a variable parameter—the `var` keyword has been replaced with the `inout` keyword. This new keyword tells Swift that the parameter's value can be expected to change in the function and that the change should be reflected back to the caller. With that exception, everything else is the same between the `cashBetterCheck` and `cashBestCheck` functions.

On lines 188 and 189, two variables are declared: `payer` and `payee`, with both being assigned `String` values. This is done because `inout` parameters must be passed a variable. A constant value will not work, because constants cannot be modified.

On line 190, the `payee` variable is printed, and the Results sidebar for that line clearly shows the variable's contents as `"Cash"`. This is to make clear that the variable is set to its original value on line 189.

On line 191, you call the `cashBestCheck` function. Unlike the call to `cashBetterCheck` on line 179, you are passing variables instead of constants for the `to` and `from` parameters. More so, for the second parameter (`payee`), we are prepending the ampersand character (&) to the variable name. This is a direct result of declaring the parameter in `cashBestCheck` as an `inout` parameter. You are in essence telling Swift that this variable is an `inout` variable and that you expect it to be modified once control is returned from the called function.

On line 193, the `payee` variable is again printed. This time, the contents of that variable do not match what was printed on line 190 earlier. Instead, `payee` is now set to the value `"James Perry"`, which is a direct result of the assignment in the `cashBestCheck` function on line 183.

BRINGING CLOSURE

Functions are great, and in the earlier code you've written, you can see just how versatile they can be for encapsulating functionality and ideas. Although the many contrived examples you went through may not give you a full appreciation of how useful they can be in every scenario, this will change as you proceed through the book. Functions are going to appear over and over again both here and in your code, so understand them well. You may want to re-read this chapter to retain all the ins and outs of functions.

I've got a little more to talk about before I close this chapter, however. Your tour of functions would not be complete without talking about another significant and related feature of functions in Swift: *closures*.

In layman's terms, a closure is essentially a block of code, like a function, that "closes in" or "encapsulates" all the "state" around it. All variables and constants declared and defined before a closure are "captured" in that closure. In essence, a closure preserves the state of the program at the point that it is created.

Computer science folk have another word for closures: *lambdas*. In fact, the very notion of the function you have been working with throughout this chapter is actually a special case of a closure—a function is a closure with a name.

So if functions are actually special types of closures, then why use closures? It's a fair question, and the answer can be summed up this way: Closures allow you to write simple and quick code blocks that can be passed around just like functions, but without the overhead of naming them.

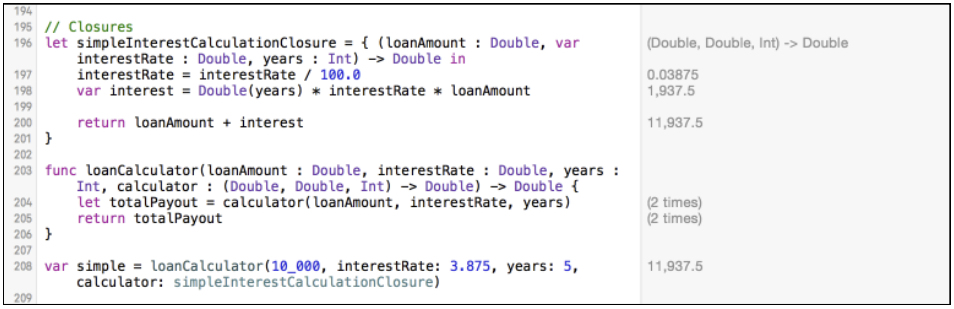


FIGURE 4.22 Using a closure to compute simple interest

In essence, closures are anonymous blocks of executable code.

Swift closures have the following structure:

```
{ (parameters) -> return_type in
  statements
}
```

This almost looks like a function, except that the keyword `func` and the name is missing, the curly braces encompass the entire closure, and the keyword `in` follows the return type.

Let's see closures in action. **Figure 4.22** shows a closure being defined on lines 196 through 201. The closure is being assigned to a constant named `simpleInterestCalculationClosure`. The closure takes three parameters: `loanAmount`, `interestRate` (both `Double` types), and `years` (an `Int` type). The code computes the future value of a loan over the term and returns it as a `Double`.

```
// Closures
let simpleInterestCalculationClosure = { (loanAmount : Double,
→ var interestRate : Double, years : Int) -> Double in
  interestRate = interestRate / 100.0
  var interest = Double(years) * interestRate * loanAmount

  return loanAmount + interest
}

func loanCalculator(loanAmount : Double, interestRate : Double, years :
→ Int, calculator : (Double, Double, Int) -> Double) -> Double {
  let totalPayout = calculator(loanAmount, interestRate, years)
  return totalPayout
}

var simple = loanCalculator(10_000, interestRate: 3.875, years: 5, calculator:
→ simpleInterestCalculationClosure)
```

FIGURE 4.23 Adding a second closure that computes compound interest

```
194
195 // Closures
196 let simpleInterestCalculationClosure = { (loanAmount : Double, var
        interestRate : Double, years : Int) -> Double in
197     interestRate = interestRate / 100.0
198     var interest = Double(years) * interestRate * loanAmount
199
200     return loanAmount + interest
201 }
202
203 func loanCalculator(loanAmount : Double, interestRate : Double, years :
        Int, calculator : (Double, Double, Int) -> Double) -> Double {
204     let totalPayout = calculator(loanAmount, interestRate, years)
205     return totalPayout
206 }
207
208 var simple = loanCalculator(10_000, interestRate: 3.875, years: 5,
        calculator: simpleInterestCalculationClosure)
209
210 let compoundInterestCalculationClosure = { (loanAmount : Double, var
        interestRate : Double, years : Int) -> Double in
211     interestRate = interestRate / 100.0
212     var compoundMultiplier = pow(1.0 + interestRate, Double(years))
213
214     return loanAmount * compoundMultiplier
215 }
216
217 var compound = loanCalculator(10_000, interestRate: 3.875, years: 5,
        calculator: compoundInterestCalculationClosure)
218
```

196	(Double, Double, Int) -> Double
197	0.03875
198	1,937.5
200	11,937.5
204	(2 times)
205	(2 times)
208	11,937.5
210	(Double, Double, Int) -> Double
211	0.03875
212	1.20935884128769
214	12,093.5884128769
217	12,093.5884128769

The formula for simple interest calculation is:

*futureValue = presentValue * interestRate * years*

Lines 203 through 206 contain the function `loanCalculator`, which takes four parameters: the same three that the closure takes, and an additional parameter, `calculator`, which is a closure that takes two `Double` types and an `Int` type and returns a `Double` type. Not coincidentally, this is the same parameter and return type signature as your previously defined closure.

On line 208, the function is called with four parameters. The fourth parameter is the constant `simpleInterestCalculationClosure`, which will be used by the function to compute the total loan amount.

This example becomes more interesting when you create a second closure to pass to the `loanCalculator` function. Since you've already computed simple interest, you can now write a closure that computes the future value of money using the compound interest formula:

futureValue = presentValue (1 + interestRate)years

Figure 4.23 shows the compound interest calculation closure defined on lines 210 through 215, which takes the exact same parameters as the simple calculation closure on line 196. On line 217, the `loanCalculator` function is again called with the same parameters as before, except the `compoundInterestCalculationClosure` is passed as the fourth parameter. As you can see in the Results sidebar, compound interest yields a higher future value of the loan than simple interest does.

```
let compoundInterestCalculationClosure = { (loanAmount : Double,
→ var interestRate : Double, years : Int) -> Double in
  interestRate = interestRate / 100.0
  var compoundMultiplier = pow(1.0 + interestRate, Double(years))
```

```
  return loanAmount * compoundMultiplier
}

var compound = loanCalculator(10_000, interestsRate: 3.875, years: 5,
→ calculator: compoundInterestCalculationClosure)
```

On line 212 you may notice something new: a reference to a function named `pow`. This is the power function, and it is part of Swift's math package. The function takes two `Double` parameters: the value to be raised and the power to raise it to. It returns the result as a `Double` value.

SUMMING IT UP

I've spent the entire chapter discussing functions and their use. Toward the end, you learned about closures and how they are essentially nameless functions that can be passed around to do useful work. As I indicated earlier, functions and closures are the foundations on which Swift apps are written. They appear everywhere and are an integral part of the development process. Knowing how they work and when to use them is a skill you will acquire over time.

In fact, there are even more things about functions and closures that I didn't touch on in this chapter. There's no need to overload you on every possible feature they have; I'll cover those extras later in the book. For now, you have enough of the basics to start doing useful programming.

Also, feel free to work with the code in the playground for this chapter. Change it, modify it, add to it, and make a mess of it if you want. That's what playgrounds are for, after all!

STAY CLASSY

With functions and closures covered, I'll turn your attention to the concept of the *class*. If you are familiar with object-oriented programming (OOP), Swift's notion of a class is similar to that of Objective-C and C++. If you're new to the idea of objects and OOP, don't worry—I'll explain all that terminology in the next chapter.

Meanwhile, feel free to take a break and review the notes and code in this chapter, as well as experiment with your playground file. When you're ready, proceed to Chapter 5, and I'll get down and dirty with classes.

INDEX

NUMBERS

2 × 2 matrix, 264–266

600 × 600 view space, 224

SYMBOLS

./ prefix, using with shell scripts, 275

..< syntax, using, 59

-> characters, using with functions, 85

+ (addition) operation, performing, 21

&& (AND) logical operator, 261

: (colon), using with variables and constants, 18

, (commas), using with arrays, 35

// (comments), converting lines into, 209

/ (division) operation, performing, 21–22

. (dot) notation, using with methods, 117

== (double equal) sign, 67–69

= (equal to) comparison, 24

! (exclamation mark)
- ending strings with, 10
- using with optionals, 195
- in Xcode, 186

> (greater than) comparison, 24, 69

>= (greater than or equal) comparison, 24, 69

< (less than) comparison, 24, 69

<= (less than or equal) comparison, 24, 69

* (multiplication) operation, performing, 21–22

! (NOT) logical operator, 260

!= (not equal to) comparison, 24, 69

% (modulo) operation, performing, 21

|| (OR) logical operator, 261–262

.. (periods), using with for-in loops, 58–59

+ (plus sign) operator, using with strings, 19

? (question mark), using with dictionaries, 43

– (subtraction) operation, performing, 21

_ (underscore)
- using with numeric representations, 23
- using with parameter names, 100–101
- using with Void keyword, 97

A

action methods, using in Xcode, 187. *See also* methods

actions and outlets, connecting, 188–189

addition (+) operation, performing, 21

addition and multiplication code, 266

advanceGame method, 242

aliases, using, 27

analyzing tools in IDE, 172

AND (&&) logical operator, 261

animateWithDuration method, 239–240

AppDelegate.swift file, 288

AppDelegate.swift source file, 176–177

append method, using with extensions, 163–164

Apple
- Developer Program, 301
- Game Center, 283

applicationDidFinishLaunching method, 176

`arguments` parameter, including in shell scripts, 276
arrays. *See also* empty array
  adding values to, 36
  combining, 41–42
  dictionaries of, 47–50
  extending, 38–39
  including in candy jar, 34–36
  interrogating for values, 35–36
  iterating, 52–54, 60
  mutability, 37
  removing values, 39–40
  replacing values, 39–40
  reviewing contents of, 40
  using commas (,) with, 35
  value types, 37
Attributes inspector. *See also* properties
  "Shows touch on highlight" option, 246
  using with buttons, 225–226

B

`backgroundColor` property, 239–240
balances, finding in banking app, 91
bank account, depositing check into, 92–94
banking app, 89–92
`bankVault` function, 97–98
base class, modeling, 125–128. *See also* classes
Bash shell, 272
`bayWindow` object, 130–131
binary notation, 23
`/bin/sh` in pathlist, 274
`Bool` type, 15, 24–26, 159. *See also* `someCondition` Boolean expression
Boolean expressions, comparing, 261
Bourne shell, 274
`break` statement, 73, 80–81
breakpoints
  encountering, 207, 256–257
  setting, 206, 259
bugs. *See also* debug area in Xcode
  analyzing `calculate` method, 206
  asking questions about, 205
  locating, 206
  setting breakpoints, 206–209
button sequence, highlighting, 239
`buttonByColor` method, 238–239
buttons for FollowMe game. *See also* `randomButton` method
  creating, 226–227
  duplicating, 226
  keeping centered on screen, 228–230
  positioning, 226–227
  resizing, 226
  selecting, 226
  using Attributes inspector with, 225–226

C

C (Celsius), computing, 162–163
`calculate` method, analyzing, 206
Calculate Simple button, naming, 202
candy jar example
  arrays, 34–36
  as container, 34
  `let` keyword, 35
  `String` constants, 35
  values, 34–35
`cashBestCheck` function, 105–106

cashBetterCheck function, 104–106
cashCheck function, 103
catching errors, 270–272
celsiusToFahrenheit method, 86–87
Character type, 15, 20
checkValidPassword() function, 269–271
chooseTransaction function, 94
class keyword, defining objects with, 115
class methods, 239
classes. *See also* base class; superclasses and subclasses
  components of, 113
  creating for interest calculator, 183–188
  vs. protocols, 148–151
  vs. structures, 142
  turning into objects, 115–116
  using, 252
closeBankVault function, declaring, 97
closures
  using, 252
  using in extensions, 166–167
  using with functions, 106–109
Cocoa frameworks, availability of, 197
CocoaConf, 301
code, inspecting, 77–80
Coin.atlas folder, 288
ColaMachine class, 159–161
collections
  iterating, 52–55
  iterating through, 60
  non-ordinal traversal, 61
colon (:), using with variables and constants, 18
color, changing for newBackDoor object, 123–124
colorTouched constant, 241
CombinationDoor class, 132–135
commands
  :help, 10
  let, 14
  :quit, 10
  referencing, 9
  retyping automatically, 10
commas (,), using with arrays, 35
comments (//), converting lines into, 209
comparisons, making, 24, 67–68
compiler errors, resource for, 302
compiler in IDE, 172
compound interest, checking calculation of, 210
compoundButtonClicked method, 203–204
compoundInterestCalculationClosure, 108
compoundInterestCalculator method, 203
computed properties, 162. *See also* properties
concatenating strings, 19–20
connecting
  actions and outlets, 188–189
  buttons, 203
constants
  on cases, 74
  using, 13–14
  versus variables, 14
constraints, setting for FollowMe game, 228–230
container, candy jar as, 34
convenience initializers, 136–138. *See also* init method
copyVar structure, 144
currency format, displayed, 196

D

data, grouping with tuples, 28–29
debug area in Xcode, 175, 207, 256. *See also* bugs
debug toolbar, 208
debugger in IDE, 172
declarations
  making, 11–13
  using long forms of, 51
decrement, post, 63
`default` keyword, 73
default parameter values, 98–100
`deinit` method, 254
delegation design pattern, using with protocols, 156–159
`deposit` function, declaring, 93
design patterns, 230
Developer Program, joining, 301
dictionaries. *See also* empty dictionary
  ? (question mark) used with, 43
  adding entries, 45
  of arrays, 47–50
  declaring and ordering, 43
  invalid values, 44
  iterating, 54–55
  keys, 42–43
  looking up entries, 43–44
  names and values, 42–43
  `nil` values, 44
  order of content, 55
  placing entries in, 42–43
  removing entries, 46–47
  updating entries, 46
`didReceiveMemoryWarning` method, 238
division (/) operation, performing, 21–22
`Door` object
  `close` and `open` methods, 116–117
  creating, 114–115
  instantiating, 116
  `lock` and `unlock` methods, 117–120, 134
dot (`.`) notation, using with methods, 117
`do/try/catch` construct, 270
double equal (==) sign, 67–69
`Double` type, 12–13, 15, 20, 193–194
Downhill Challenge game. *See also* gameplay
  Assets group, 288
  Classes group, 288
  contents of folder, 284
  Game Over scene, 286
  Game scene, 286, 293–298
  gameplay, 284–285
  `GameViewController.swift` class, 298–300
  going through source code, 300
  Home scene, 286, 289–293
  Leaderboard scene, 286, 291
  playing, 285–286
  premise, 283
  project files, 287
  running in simulator, 285–286
  Scenes group, 289
  `SKNode` class, 298
  snowman player, 283–284
  social connectivity, 283
  `viewOldLoad()` function, 299

E

editor area in Xcode, 175, 198
editor in IDE, 172
`else` clause, executing, 68
empty array, declaring, 50–51. *See also* arrays
empty dictionary, creating, 51–52. *See also* dictionaries
`enumerate()` method
  using with arrays, 54
  using with dictionaries, 54–55
enumerations, 138–141, 236
equal vs. identical objects, 267–268
equality, testing for, 68
error type, 268–269
errors
  auto correcting, 231
  catching, 270–272
  `do/try/catch` construct, 270
  throwing, 12, 268–270
exclamation mark (!)
  ending strings with, 10
  using with optionals, 195
  in Xcode, 186
exiting loops, 80–81
extending types, 161–165
extensions
  `append` method, 163–164
  computed properties, 162
  `Double` type, 193–194
  for extending `Int` type, 161
  form of, 160
  `prepend` method, 163–164
  `String` type, 163
  temperature units, 162–163
  using closures in, 166–167
external parameter names, 102

F

F (Fahrenheit), computing, 162–163
`fahrenheitToCelsius` function, 85
`false` and `true` values, 24
first-class objects, functions as, 92–94. *See also* objects
`firstClassLetter` object, 254
`Float` type, 15
floating point number, 11–12
FollowMe game. *See also* gameplay
  600 × 600 view, 224
  adjusting difficulty, 247
  `advanceGame` method, 242
  `animateWithDuration` method, 239–240
  `backgroundColor` property, 239–240
  `buttonByColor` method, 238–239
  `ButtonColor` enumeration, 236
  buttons, 225–227
  `buttonTouched` action method, 246
  coding, 231–235
  `colorTouched` constant, 241
  constraints, 228–230
  container view, 228
  creating, 222–223
  `didReceiveMemoryWarning` method, 238
  enumerations, 236
  game methods, 238–242
  `highlightColor` variable, 239
  highlighting button sequence, 239
  importing `UIKit`, 236

FollowMe game (*continued*)
improving playability, 247
index for next button, 237
iPhone 5s simulator, 227
keeping track of turns, 237
`Main.storyboard` file, 224, 245
model objects, 237
optional chaining, 241
overridable methods, 238
playing, 247
`playSequence` method, 239–240, 244
`randomButton` method, 244
randomness, 243
restarting, 244
rounds, 237, 244
`switch/case` construct, 238–239
UI design, 224–230
`UIButton tag` property, 241
view objects, 237
winning and losing, 242–244
`winningNumber` variable, 237
`for` loop, variation of, 61
`for-in` loops
as enumeration mechanism, 58–59
nesting, 55
`frontDoor` object, 116–117
`func` keyword, 85, 100
functions. *See also* methods
-> characters used with, 85
calling, 86–87
calling within parameter names, 101
closures, 106–109
coding, 84–86
as first-class objects, 92–94
mathematical notation, 84
with multiple parameters, 87–88
naming, 100–101
nesting, 96–98
results of calling, 86
returning values of types, 85
returning from functions, 94

G

Game Center, 283
game methods, 238–242
Game Over scene, 286
Game scene, 286, 293–298
`didBeginContact()` method, 295
`didEvaluateActions()` method, 297–298
`didMoveToView()` method, 294–295
`GameLogic` object, 293
`NewObject` class, 293
score and help nodes, 294
`setPlayer()` method, 295
`snowmanAnimate()` method, 295
`update()` method, 296–297
GameKit framework, 282
`GameLogic.swift` class, 288
`GameOverScene.swift` class, 288
gameplay. *See also* Downhill Challenge game; FollowMe game
button arrangement, 221–222
elements, 221
losing, 221
play flow, 221
playability, 221

randomness, 221
UI design, 221–222
`winning`, 221
`GameScene.swift` class, 288
`GameViewController.swift` class, 288
`SKNode` class, 298–299
`SKView` class, 300
generic method, 263–264. *See also* methods
`{ get set }`, using with protocols, 152–153
gigabytes (gb), converting `Int` to, 161–162
Go menu, 8
greater than (>) comparison, 24, 69
greater than or equal (>=) comparison, 24, 69
gutter, clicking lines in, 206

H

hash bang syntax, 274–276
Hello, World! 10–11
`:help` command, typing, 10
Help menu in Xcode, 278
hexadecimal notation, 23
`highlightColor` variable, 239
Home scene, 286, 289–293
`homeMailBox` object, 254
`HomeScene.swift` class, 288
`House` constant, 150
HUD (heads-up display) window, appearance of, 188

I

`@IBAction` tag, appearance of, 188
`@IBOutlet` tag, appearance of, 188
IDE (integrated development environment), components of, 172
identical vs. equal objects, 267–268
`if` statements
comparing numbers in, 69
multiple, 70–72
using, 66–70
using in playground, 67
immutable `String` values, 36–37
implicit external parameter name, 100–101
implicitly unwrapped optional, 186–187. *See also* optionals
`import` statement
in shell scripts, 276
using in Xcode, 177, 184–185
increment, post, 63
inheritance
and protocols, 155–156
superclasses and subclasses, 124–125
`init` method, 116, 121–123, 136, 159, 253, 290. *See also* convenience initializers
`inout` keyword, using to modify parameters, 105
input leniency, 200
`insert()` method, using with arrays, 41
inspecting code, 77–80
inspector icons, locating, 225
instantiation, 115–116
`NiceDoor class`, 131
subclasses, 130–136

`Int` type, 15–16, 20
  converting, 161–162
  explained, 11
  extending, 161–162
  optionals, 29–30
interactivity, benefits of, 7
interest calculator. *See also* simple interest
  adding Result label, 182–183
  Calculate Simple button, 202
  `compoundButtonClicked` method, 203–204
  `compoundInterestCalculator` method, 203
  creating classes, 183–188
  displaying in editor area, 198
  encountering bugs, 205–210
  file types, 184
  formatted input, 199
  inputs and outputs, 179
  labels, 181
  optimizing window size, 183
  Push Button element, 180
  renaming button title, 182
  `SimpleInterest` class, 184–185
  testing, 205
  text fields, 182
  UI (user interface), 180–182
`interestRate` variable, 208–209
iOS apps. *See* FollowMe game
iPhone
  5s simulator, 227
  aspect ratio, 222
iterating collections, 52–55

K

K (Kelvin), computing, 162–163
kilobytes (kb), converting `Int` to, 161–162

L

labels, creating for interest calculator, 181
lambdas, using with functions, 106–109
large number notation, 23
`launch` method, using with shell scripts, 276–277
`LaunchScreen.xib` file, 288
lazy property, 258. *See also* properties
Leaderboard scene, 286, 291
Left Arrow key, using, 10
leniency, turning on, 200
less than (<) comparison, 69
less than or equal (<=) comparison, 69
`let` command
  in candy jar example, 35
  using, 14
`Letter` class, 253–254
limits, upper and lower, 16
lines in gutter
  clicking, 206
  converting to comments, 209
  stepping over, 208
LLDB debugger, typing commands in, 207–208
LLVM (low level virtual machine), 250
`loanCalculator` function, 108
`lock` and `unlock` methods, 117–120, 134, 150

logical operators, 259–260
  AND (&&), 261
  OR (| |), 261
  NOT (!), 260
loops, exiting, 80–81

M

MacTech Conference, 301
Mailbox class, 253–254
MailChecker class, 257–258
MainMenu.xib file, 179
Main.storyboard file, 224, 245, 288
math
  binary notation, 23
  expressions, 22
  hexadecimal notation, 23
  large number notation, 23
  mixing numeric types, 22
  numeric representations, 23
  octal notation, 23
  operations, 21
  scientific notation, 23
matrix addition and multiplication code, 266
megabytes (mb), converting Int to, 161–162
memory address, 250
memory leaks, 251–252
memory management
  LLVM (low level virtual machine), 250
  value vs. reference, 250–251
methods, 113. *See also* action methods; functions; generic method; type methods
  specifying as mutating, 164–165
  using dot (.) notation with, 117
.minor and .max, adding to types, 16
mobile app. *See* FollowMe game
modulo (%) operation, performing, 21
multiplication (*) operation
  and addition code, 266
  performing, 21–22
mutable array
  creating, 37
  using, 38
mutating, specifying methods as, 164–165
MVC (model-view-controller), 230–231, 237
MyFirstSwiftApp project, 175, 184

N

navigator area in Xcode, 175
nested functions, 96–98
nesting for-in loops, 55
newBackDoor object, changing color of, 123–124
newBalance variable, declaring, 93
NewDoor class, 117–118
  init method, 121–123
  self keyword, 122
newFrontDoor object, properties of, 120
NewLockUnlockProtocol, 152
NiceDoor class
  creating, 128–129
  instantiating, 131
NiceWindow class, creating, 128–129
nil value, 29–30, 186, 194–195
NOT (!) logical operator, 260
not equal to (!=) comparison, 24, 69
notifications, using in Cocoa, 177

NSNumberFormatter class, 192, 194, 196–197, 208
Attributes inspector, 199
locating, 198
NSString method, 20
NULL value, 29
number formatting. *See* NSNumberFormatter class
numbers, comparing in if statements, 69
numeric representations, 23
numeric types
mixing, 22
upper and lower limits, 16

O

objects. *See also* first-class objects
defining with class keyword, 115
equal vs. identical, 267–268
properties and behaviors, 112
testing for identity, 268
turning classes into, 115–116
Object.swift class, 288
octal notation, 23
OOP (object-oriented programming), 112
base class, 125–128
inheritance, 124
subclasses, 128–136
operator overloading, 264–266
optional chaining, 241
optional Int, 29–30
optionals. *See also* implicitly unwrapped optional
explained, 186
unwrapping, 195
OR (| |) logical operator, 261–262
outlets and actions, connecting, 188–189
override keyword, 134

P

parameter names
best practices, 102–103
external, 102
implicit external, 100–101
parameter values, prohibited changing of, 103–105
parameters. *See also* unnamed parameter
defaults, 98–100
passing to functions, 88–89
setting values for, 100
using temporary variables with, 104–105
variadic, 90
passcode parameter, 97
passwords, checking and trying, 270–272
peppers example. *See* dictionaries
periods (.), using with for-in loops, 58–59
playground
Timeline pane, 167
using in Xcode, 64–65
playSequence method, 239–240, 244
plus sign (+) operator, using with strings, 19
Portal class, 125–128, 148
post increment and decrement, 63
prepend method, using with extensions, 163–164
print method, using, 26–27, 35
print() method, using, 10–11
profiling tools in IDE, 172
project manager in IDE, 172

project window in Xcode, 174–178
projects
  MyFirstSwiftApp, 175
  saving in Xcode, 174
properties, 112. *See also* Attributes inspector; computed properties; lazy property
protocols
  adding variables to, 151
  adopting multiple, 153–154
  vs. classes, 148–151
  delegation design pattern, 156–159
  and inheritance, 155–156
  using, 151–153
Push Button element, creating for interest calculator, 180

Q

question mark (?), using with dictionaries, 43
`:quit` command, typing, 10
quitting REPL, 34

R

$R3? temporary variable, 25
`randomButton` method, 244. *See also* buttons for FollowMe game
randomness, including in games, 243
raw values, using with enumerations, 139. *See also* values
`Rectangle` structure, using with protocols, 154
reference cycle
  breaking, 256–257
  in closures, 257–259
  explained, 252
  `firstClassLetter` object, 254–256
  `homeMailBox` object, 254–256
  `Letter` and `Mailbox` classes, 253–254
  `MailChecker` class, 257–258
  test code, 254–256
reference types vs. value types, 143–145, 250–252
`repeat-while` loop, 76–77
REPL (Read-Eval-Print-Loop) tool
  commands, referencing, 10
  quitting, 34
  temporary variable, 25
Results sidebar, contents of, 77–80
returned functions, calling, 94–96

S

safety, emphasis on, 36
saving projects in Xcode, 174
scientific notation, 23
Scoville units, 42
`securityDoor` object, 134
`self` keyword, 122, 162
`setLabel()` method, calling for Home scene, 291
shell scripts
  `./` prefix, 275
  `arguments` parameter, 276
  `/bin/sh`, 274
  creating, 272–274
  executing, 275
  hash bang syntax, 274–276
  `import` statement, 276
  `launch` method, 276–277

shell scripts (*continued*)
- permissions, 274–275
- type methods, 276
- `waitUntilExit` method, 277

`showLeaderboard()` method, 291–292
Simon electronic game, 220
simple interest, computing, 209. *See also* interest calculator
`SimpleInterest` class, creating, 184–185
`simpleInterestCalculationClosure` constant, 107–109
`SKNode` class, 298–299
`Snowball.atlas` folder, 288
snowman in Downhill Challenge, 283–284
`Snowman.atlas` folder, 288
`SnowMass.sks` file, 289
`SnowParticle.sks` file, 289
`Snow.sks` file, 289
`someCondition` Boolean expression, 76–77. *See also* `Bool` type
Spotlight, using, 8
SpriteKit framework, 282–283
stackoverflow.com, 302
storyboard file
- locating for FollowMe game, 224
- revising, 245–246

`strcat()` function, 20
`String` type, 15, 20
- in candy jar example, 35
- extending, 163–164
- immutability of, 36–37

`stringFromNumber` method, 194–196
strings
- casting into `Int`s and `Double`s, 17
- comparing, 70
- concatenating, 19–20
- declaring, 19
- testing equality of, 25

`stringWithFormat:`, using for concatenation, 20
strong references, 252
structures, 141–142
subclasses
- creating, 128–129
- instantiating, 130–136

subtraction (–) operation, performing, 21
`sunRoomDoor` object, 130–131
superclasses and subclasses, 124–125. *See also* classes
Swift app, running, 178
`switch/case` construct, 72–75, 78–79, 238–239
system requirements, 7

T

<T> generic placeholder, 263–264
`tag` property
- setting for red button, 245
- using with `UIButton`, 241

targets in Xcode, 175
technology conferences, attending, 301–302
temperature conversion, 85
temperature units, extending, 162–163
temporary variable, 25. *See also* variables

Terminal application
launching, 8
typing commands in, 9
test subsystem in IDE, 172
testing
importance of, 210
interest calculator, 205
unit tests, 211–212
text fields, creating for interest calculator, 182
throwing errors, 268–270
Timeline pane, displaying in playground, 167
toolbar in Xcode, 174
`touchesBegan()` method, calling for Home scene, 292
`Tractor` class, convenience initializers in, 136–138
`TriangleProtocol`, 156
`triple` function, 165
`TruckParticle.sks` file, 289
`true` and `false` values, 24
`tryPassword` function, 270
tuples, grouping data with, 28–29
type aliases, using, 27
type conversion, 17
type methods, 239, 276. *See also* methods
type promotion, 22
types
adding `.minor` and `.max` to, 16
associating with variables, 15
extending, 161–165
interactions between, 16–18
upper and lower limits, 16

U

`UIButton` `tag` property, 241
`UInt` types, 15–16
underscore (_)
using with numeric representations, 23
using with parameter names, 100–101
using with `Void` keyword, 97
unit tests
creating, 211–214
forcing failure of, 215
invoking, 215–216
passing, 214
`unlock` and `lock` methods, 117–120, 134, 150
unnamed parameter, 97. *See also* parameters
unsigned integers, 16
utilities area in Xcode, 175

V

value types
including in arrays, 37
vs. reference types, 143–145, 250–252
values. *See also* raw values
adding to arrays, 36
in candy jar example, 34–35
inserting at locations, 40–41
removing from arrays, 39–40
replacing in arrays, 39–40
`var` declaration, using on parameters, 104
variables. *See also* temporary variable
adding numbers to, 62–63
adding to protocol definitions, 151
assigning values to, 12

variables (*continued*)
versus constants, 14
declaring, 11
declaring as implicitly unwrapped optionals, 187
declaring explicitly, 18–19
naming, 13
parameter passing notation, 89–92
subtracting numbers from, 62–63
types, 15
using temporarily with parameters, 104–105
variadic parameters, 90
`Vehicle` structure, 142, 150
vending machine, modeling, 157–159
`VendingMachineProtocol`, 159
`ViewController.swift` file, replacing contents of, 231–235
`viewOldLoad()` function, 299
`Void` keyword, 97

W

`waitUntilExit` method, using with shell scripts, 277
weak references, 257
`while` loops, 75–79
`withdraw` function, declaring, 93
`writeBetterCheckFrom` function, 101
`writeCheckFrom` function, 101
`writeCheckTo` function, 99

X

Xcode, playground, 64–65
Xcode IDE (integrated development environment), 172
! (exclamation mark), 186
action methods, 187
AppDelegate.swift source file, 176
`applicationDidFinishLaunching` method, 176
context-sensitive help, 197
Continue Program Execution button, 256
debug area, 175, 207
documentation browser, 278–279
editor area, 175, 198
Help menu, 278–279
implicitly unwrapped optional, 186–187
`import` statement, 177
inspector icons, 225
launching, 173
`MainMenu.xib` file, 179
navigator area, 175
optionals, 186
project window, 176–178
releases of, 172
saving projects, 174
shell scripts, 272–274
targets, 175
toolbar, 174
utilities area, 175

This page intentionally left blank

Apple Pro Training Series

Apple offers comprehensive certification programs for creative and IT professionals. The Apple Pro Training Series is both a self-paced learning tool and the official curriculum of the Apple Training and Certification program, used by Apple Authorized Training Centers around the world.

To see a complete range of Apple Pro Training Series books, videos and apps visit: www.peachpit.com/appleprotraining

Apple
Certified