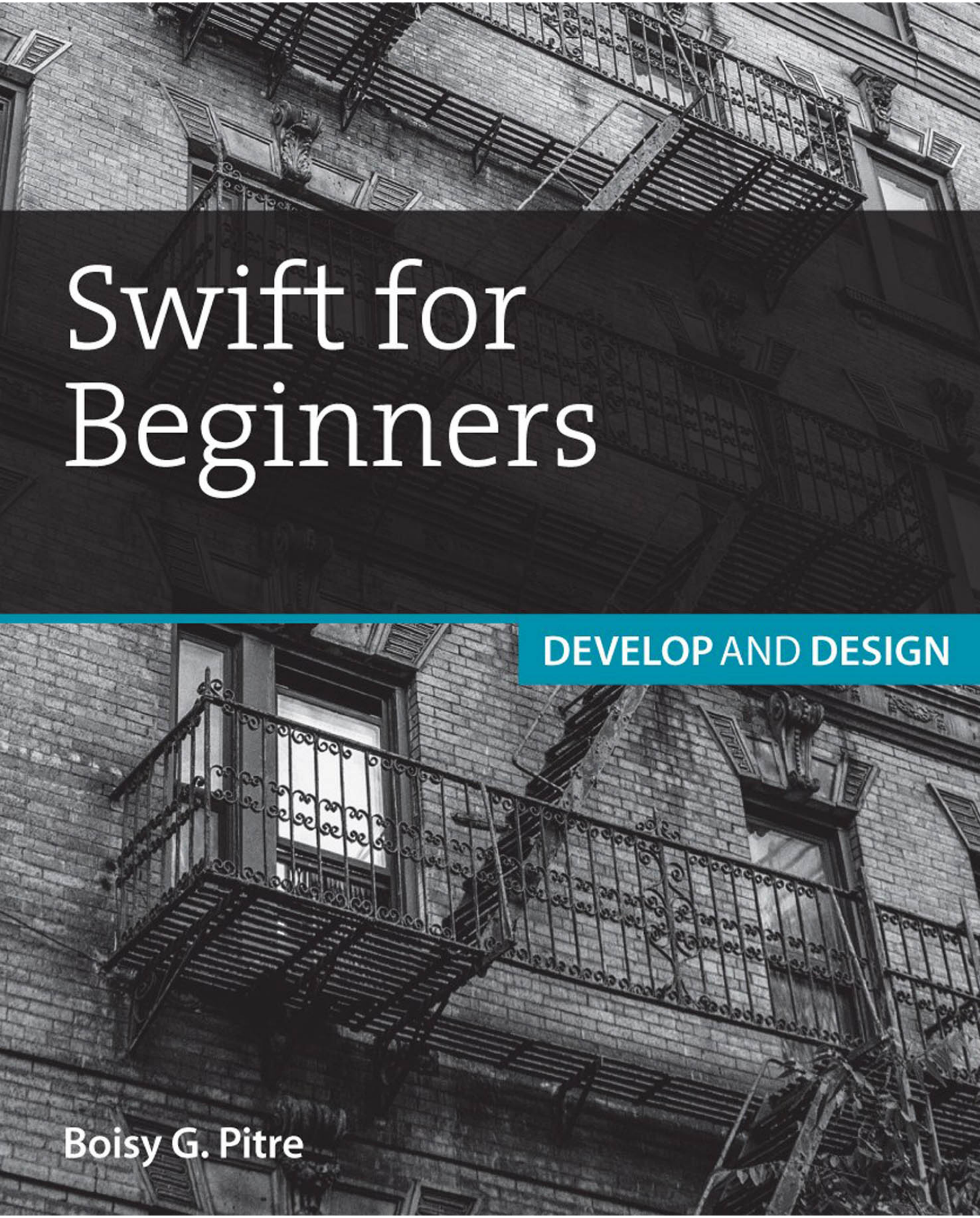

Swift for Beginners
DEVELOP AND DESIGN
Boisy G. Pitre

Swift for Beginners

DEVELOP AND DESIGN

Boisy G. Pitre

PEACHPIT PRESS
WWW.PEACHPIT.COM

Swift for Beginners: Develop and Design

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: Robyn G. Thomas
Copyeditor: Darren Meiss
Proofreader: Nancy Bell
Technical editor: Steve Phillips
Compositor: Danielle Foster
Indexer: Valerie Haynes Perry
Cover design: Aren Straiger

ISBN-13: 978-0-134-04470-5
ISBN-10: 0-134-04470-3

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

ACKNOWLEDGMENTS

When creating a book, it truly "takes a village," and I could not have had better support from the staff at Peachpit Press. Many thanks go to executive editor, Cliff Colby at Peachpit for providing me the opportunity; editor Robyn Thomas for her immensely helpful and invaluable editing skills and for keeping the project on track; technical editor Steve Phillips who meticulously commented on drafts of the book for technical correctness and whose contributions undoubtedly made this work better.

During my writing, I've drawn inspiration from the works of a number of friends who are authors in the iOS and Mac OS developer community: Chris Adamson, Bill Cheeseman, Bill Dudney, Daniel Steinberg, and Richard Warren.

I would be remiss if I didn't mention the MacTech Conference dynamic duo 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.

A tip of the hat to James Dempsey, whose band, the Breakpoints, and their excellent *Backtrace* album fueled several long writing and review sessions with their rollicking tunes. Java Square Café in downtown Opelousas, Louisiana provided a great place to write as well as tasty lattes. Also, thanks to *Dave et Ray's Camp Jam/Supper Club* and my friends there who served as inspiration for several of the coding examples I used.

Much appreciation goes to the minds at Apple for creating Swift, along with a host of great products over the years that have enhanced my own productivity and that of many, many others.

Finally, many thanks to my family, especially my wife, Toni, whose patience and encouragement while I worked on this book was abundant.

ABOUT THE AUTHOR

Boisy G. Pitre is Affectiva's Mobile Visionary and lead iOS developer, where his work has led to the creation of the first mobile SDK for delivering emotions to mobile devices for the leading emotion technology company and spin-off of the MIT Media Lab. 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 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, 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.

CONTENTS

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.

Not only is Swift easy to learn, it's extremely powerful. You'll get a taste of some of that power here in this book.

WHO IS THIS BOOK FOR?

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

No matter your skill level or prior experience, *Swift for Beginners* will appeal to anyone who wants to learn about Swift.

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 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 between 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.

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.

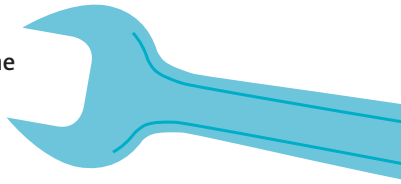
Source code for all the chapters is available at www.peachpit.com/swiftbeginners. You can download the code for each chapter, which cuts down considerably on typing; nonetheless, I am a firm believer in typing in the code. By doing so, you gain insight and comprehension you might otherwise miss if you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

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

Highlighted code throughout the book 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>
```

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



NOTE: To print or not to print? Remember, simply typing the name of the item without using `print` or `println` is acceptable when using the REPL. When doing so, the result of the expression is assigned to a temporary variable—in this case `$R0`.

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; various Cocoa frameworks are touched on where necessary to facilitate the examples but are not heavily emphasized for their own sake.

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 all 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

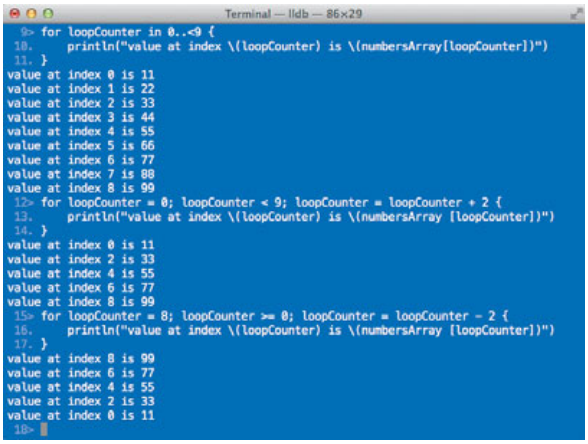
Swift is the language you'll learn in this book. Swift is a modern language designed from the ground up to be easy to learn as well as powerful. It is language that Apple has chosen to fuel the continued growth of apps, which make up their iOS and Mac OS ecosystem.

XCODE

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

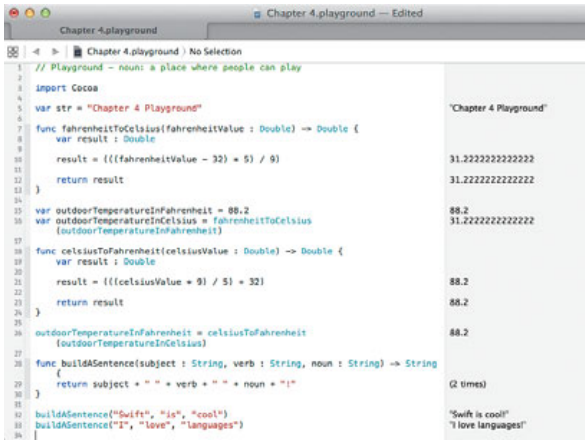
LLVM

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



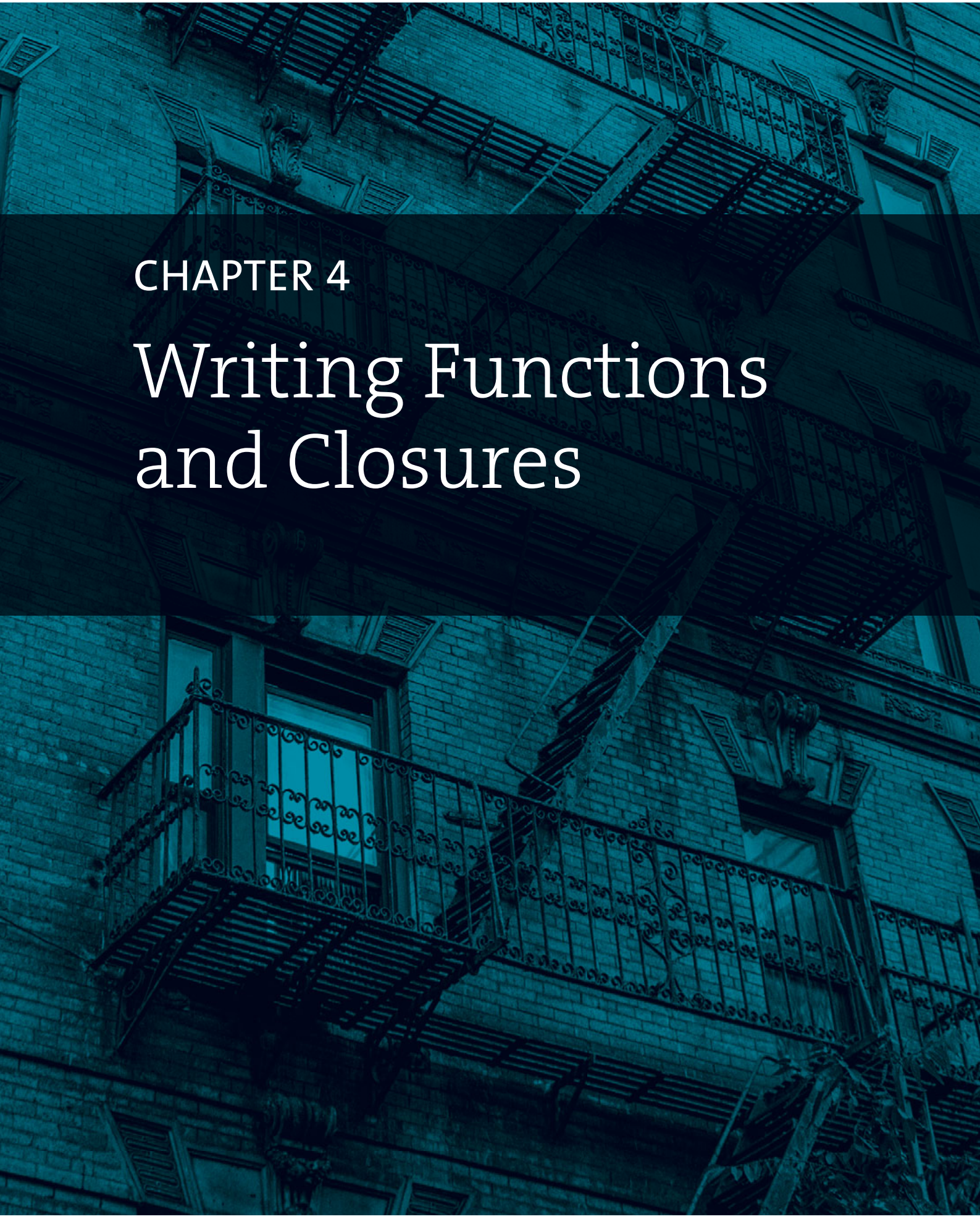
THE REPL

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

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



CHAPTER 4

Writing Functions and Closures

We'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 6'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 called functions.

Let's start this chapter with a fresh, new playground file. If you haven't already done so, launch Xcode 6 and create a new playground by choosing File > New > Playground, and name it **Chapter 4**. We'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 an input, 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:

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

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 on 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 this function, and functions in general, 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 the Swift compiler can understand.

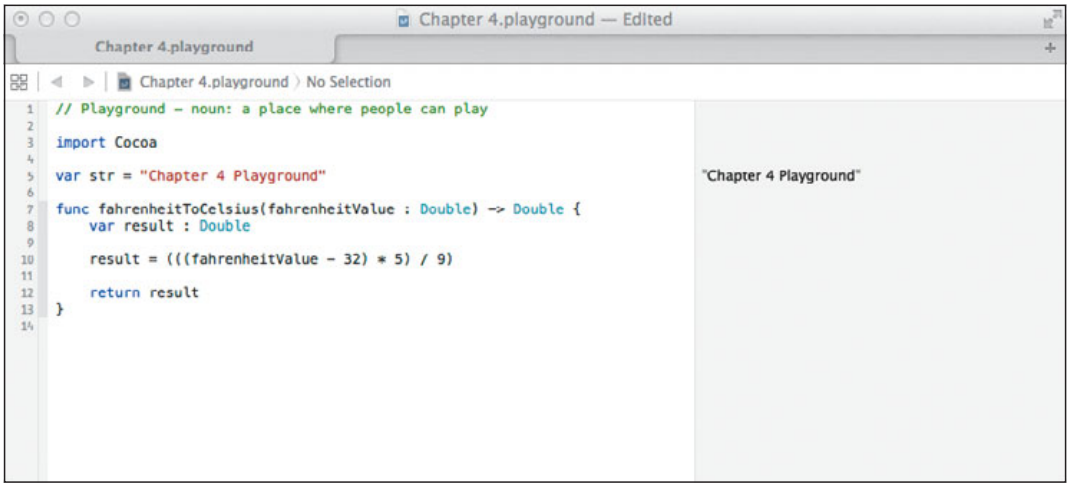


FIGURE 4.1 Temperature conversion as a Swift function

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 parameter's type is explicitly declared as `Double`.

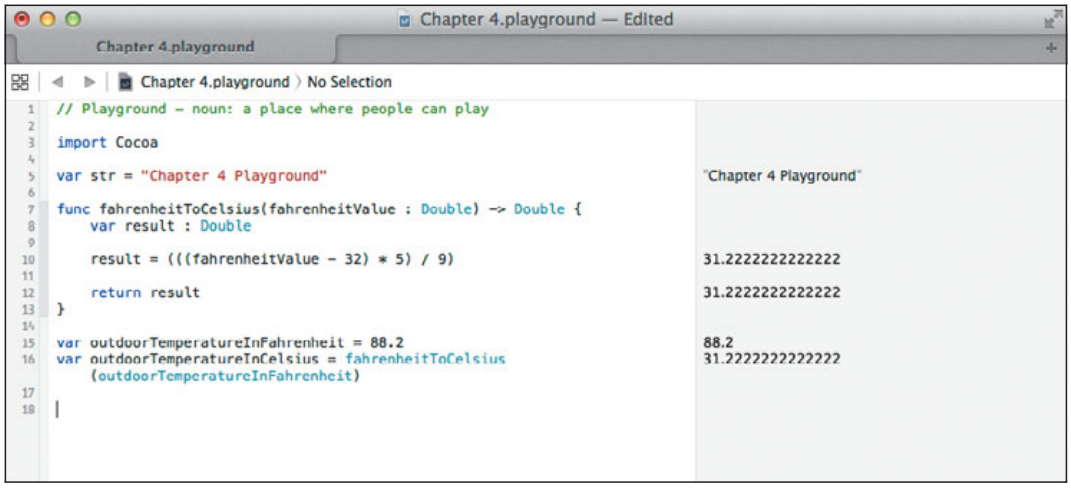
Following the parameter are the two characters `->`, which denote that this function is returning a value of a type (in this case, a `Double` type), followed by the open curly brace, which indicates the start of the function.

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 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. Any time 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**.

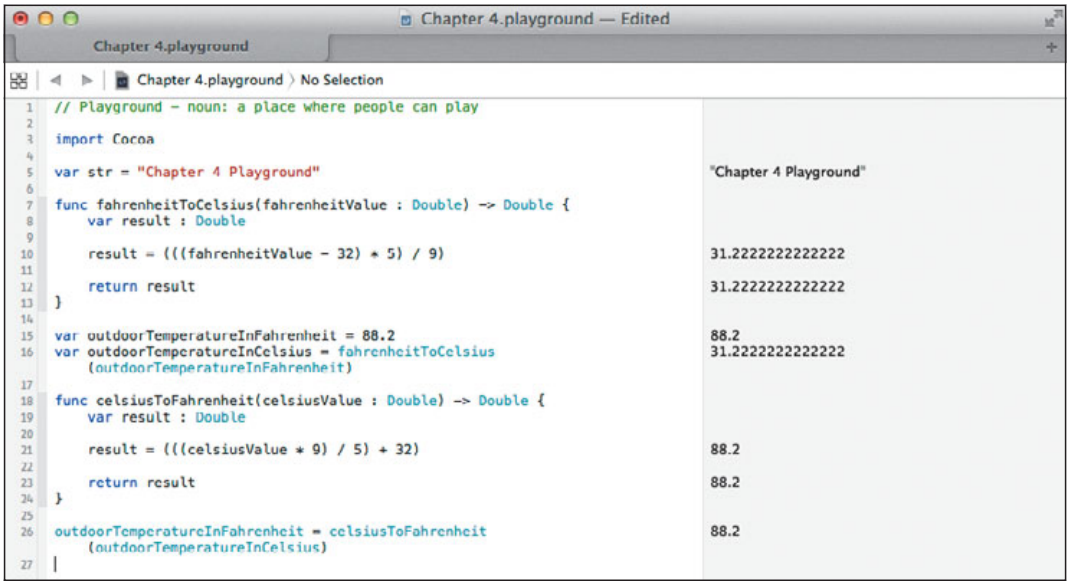


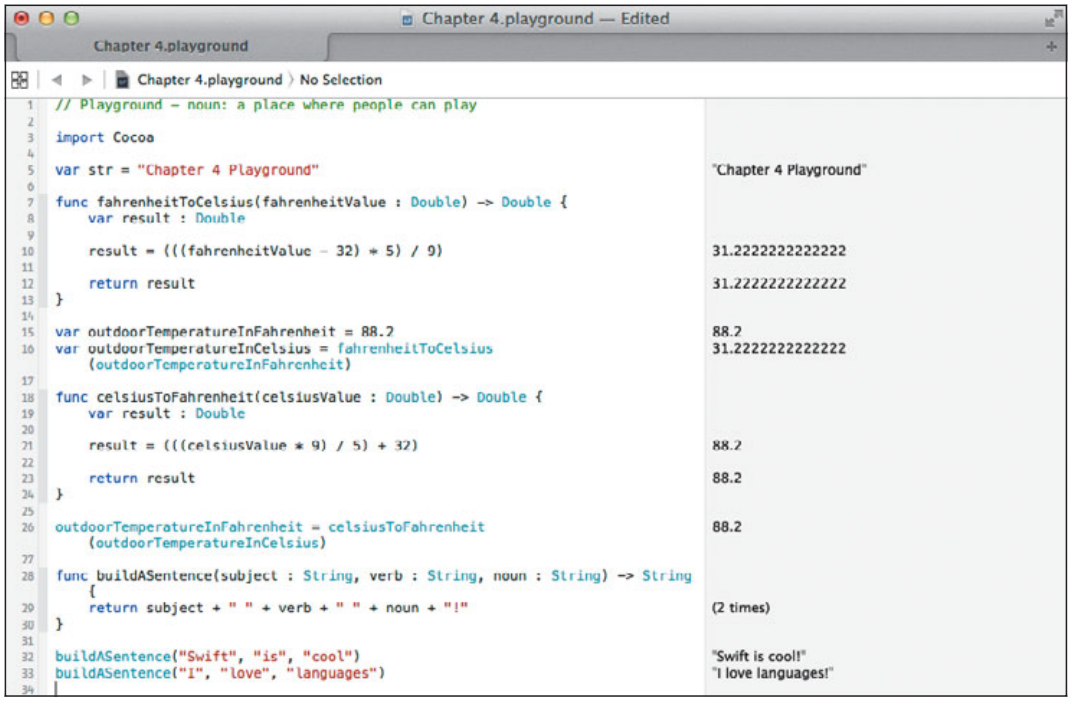
FIGURE 4.3 Declaring the inverse function, `celsiusToFahrenheit`

```
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.

FIGURE 4.4 A multi-parameter function



MORE THAN JUST NUMBERS

The notion of a function in Swift is more than just the mathematical concept we 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**).

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

buildASentence("Swift", "is", "cool")
buildASentence("I", "love", "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.

For clarity, the function is called twice on lines 32 and 33, resulting in the sentences in the Results sidebar. Feel free to replace the parameters with values of your own liking and view the results interactively.

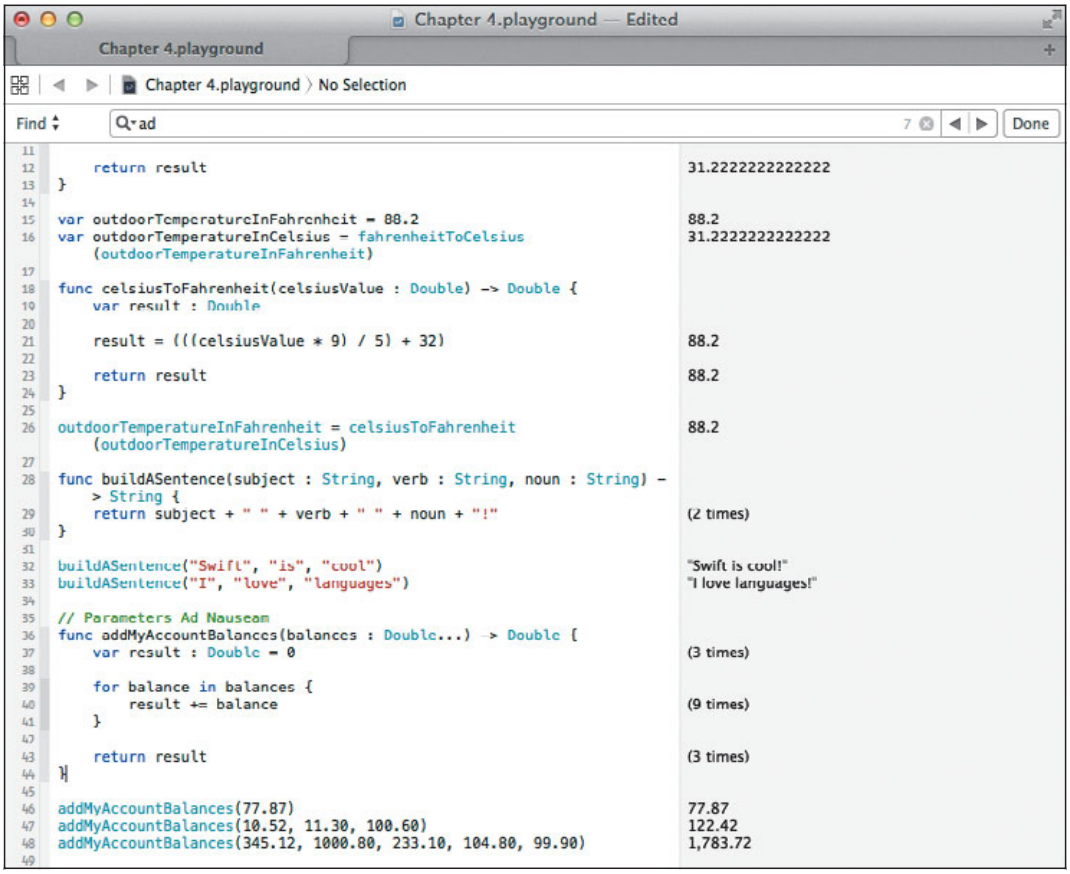


FIGURE 4.5 Variable parameter passing in a function

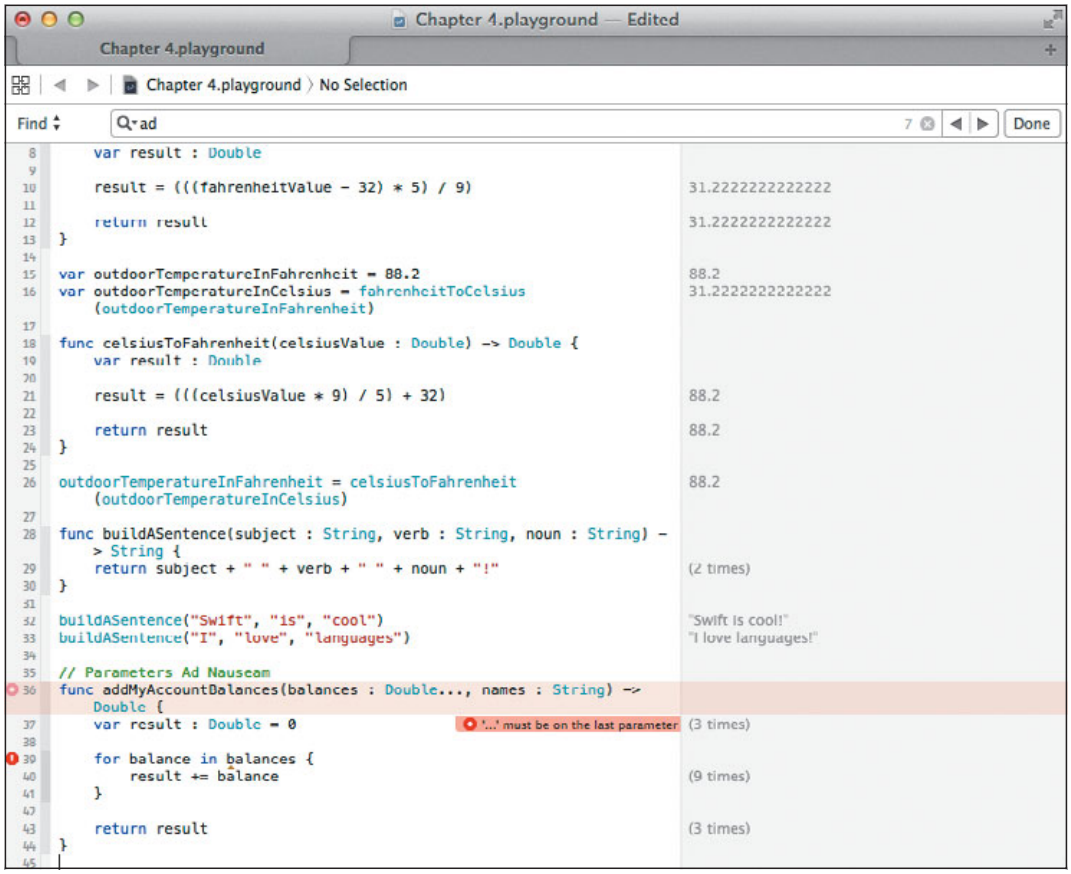
PARAMETERS AD NAUSEAM

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**.

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

   for balance in balances {
      result += balance
   }
```

FIGURE 4.6 Adding additional variable parameters results in an error.




```
    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, our `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.

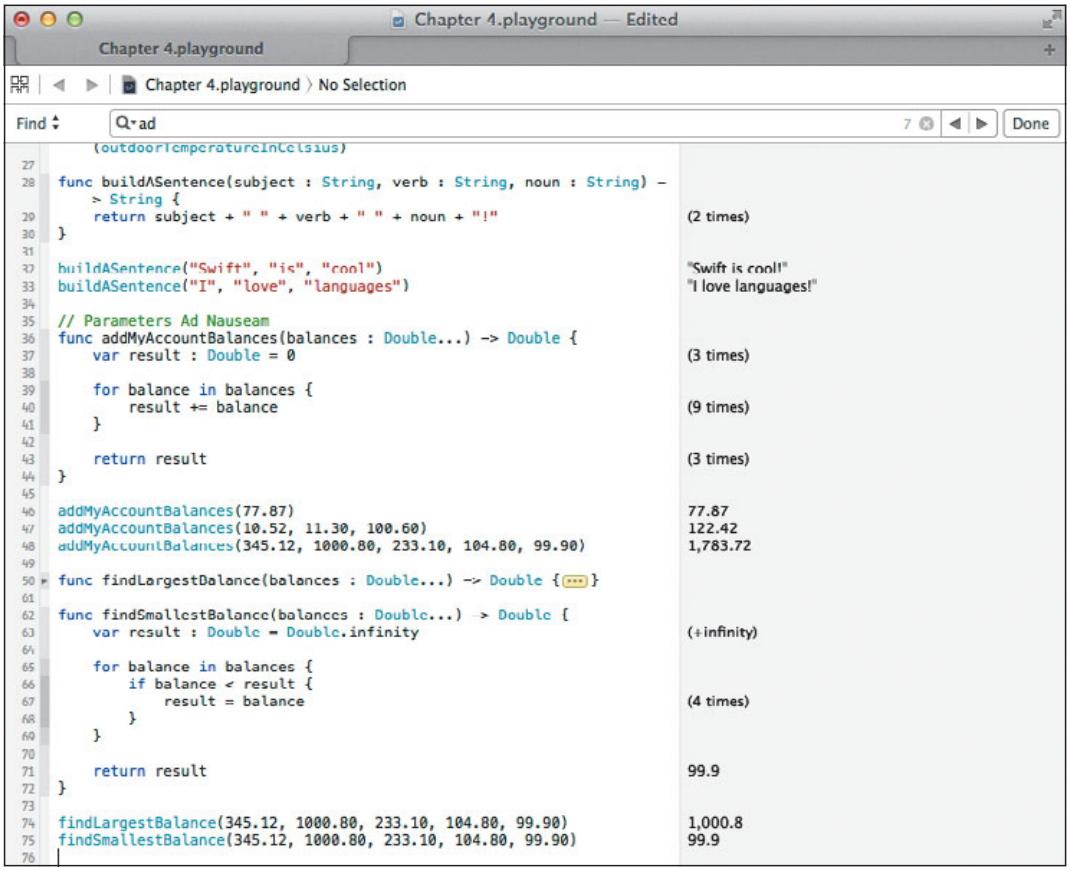


FIGURE 4.7 Functions to find the largest and smallest balance

This is a no-no, and Swift will quickly shut you down with an error. Only the *last* 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**.

```
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) {
   var newBalance : Double = account.balance + amount
   return (account.name, newBalance)
}
```

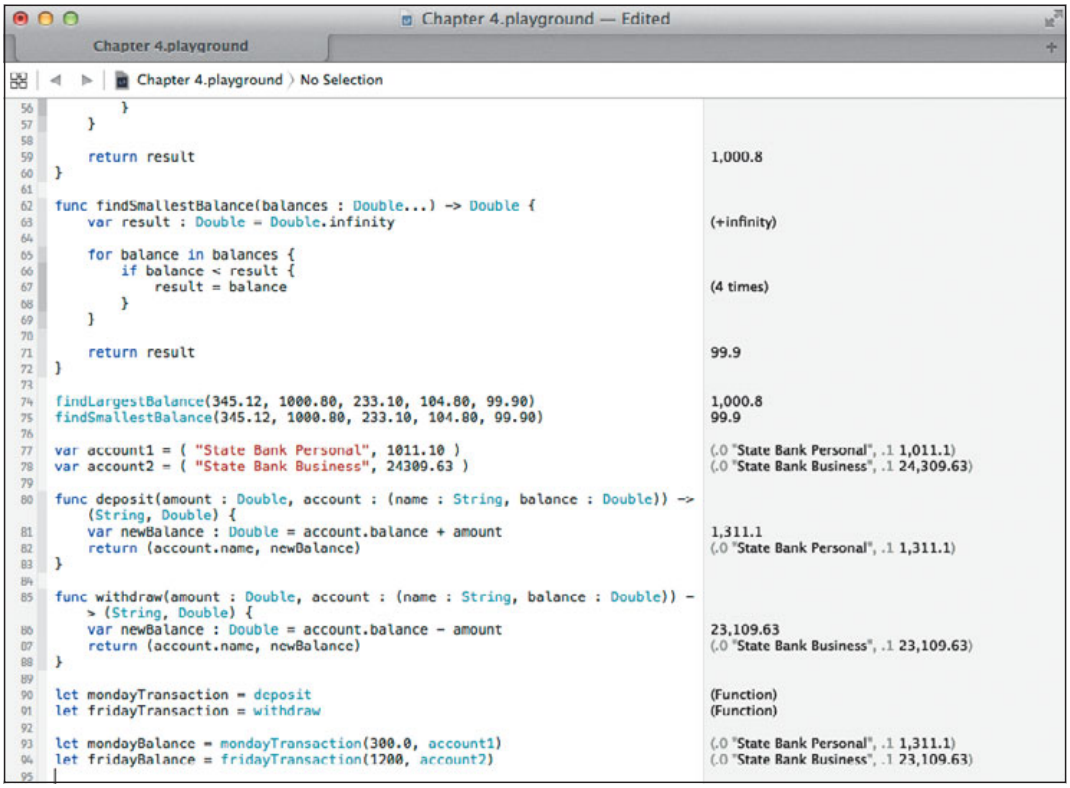


FIGURE 4.8
Demonstrating functions as first-class types

```
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, account1)
let fridayBalance = fridayTransaction(1200, account2)
```

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 is declared named `deposit` that 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.

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!

That's a mouthful. Let's take a moment to look at that line closer 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`.

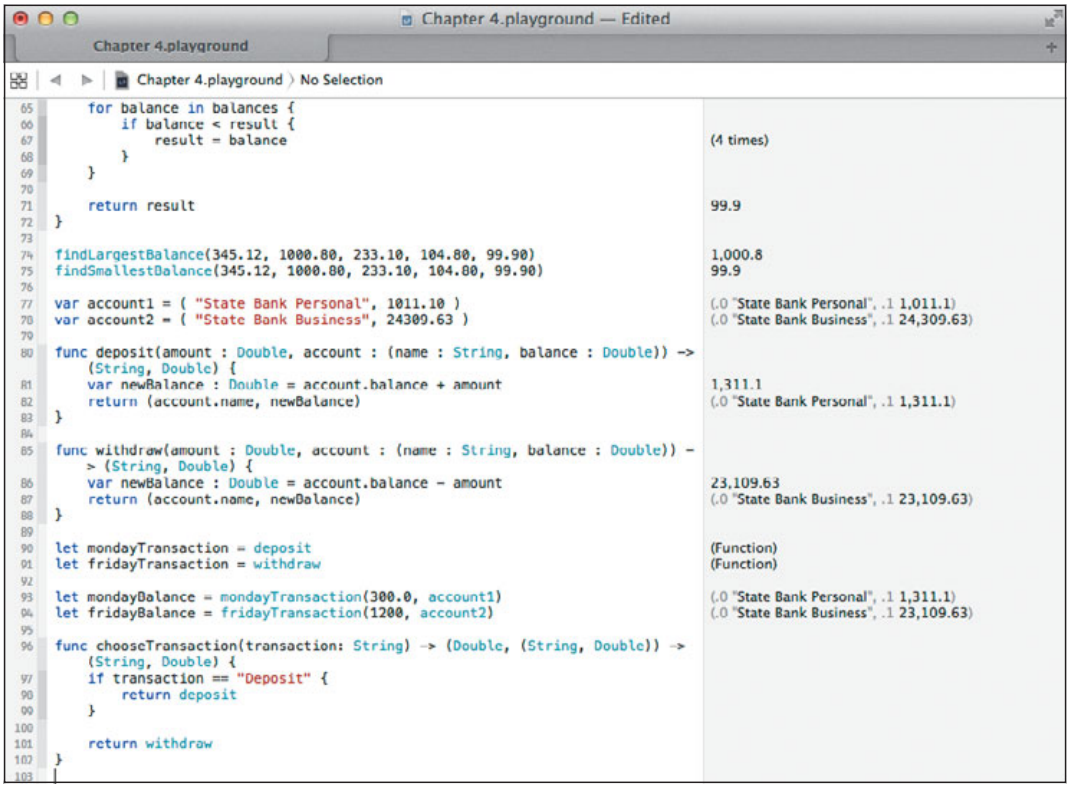


FIGURE 4.9
Returning a function from a function

What functions did you write that meet this 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.

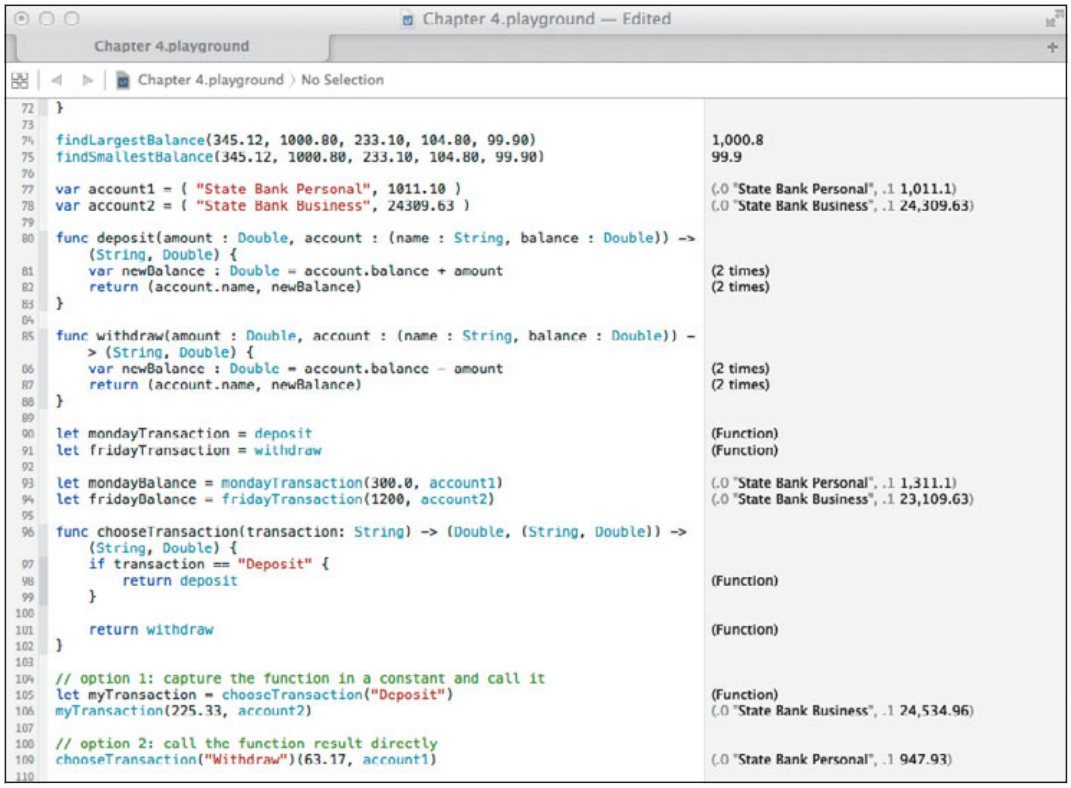
Back to the `chooseTransaction` function: On line 97, the `transaction` parameter, which is a `String`, is compared against the constant string `"Deposit"` and if a match is made, the `deposit` function is returned on line 98; otherwise, the `withdraw` function is returned on line 101.

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)
```

FIGURE 4.10 Calling the returned function in two different ways



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 of another function? Yes, such a thing exists. They're called *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"
   }
```

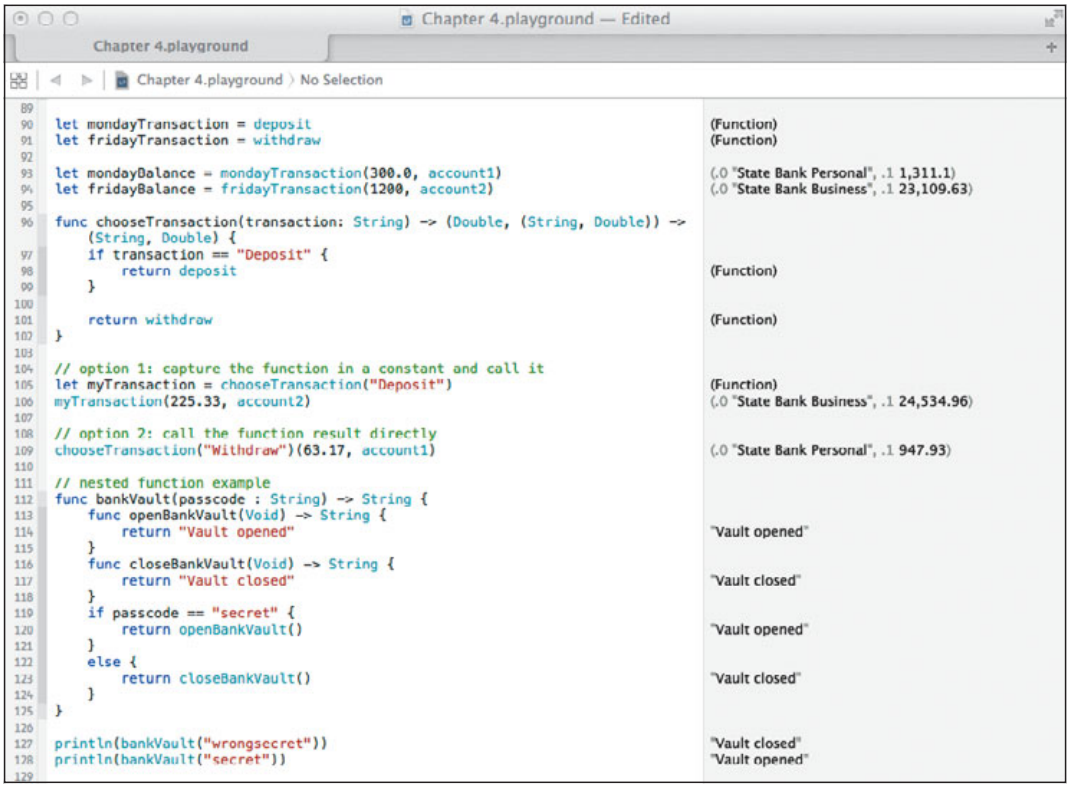


FIGURE 4.11 Nested functions in action

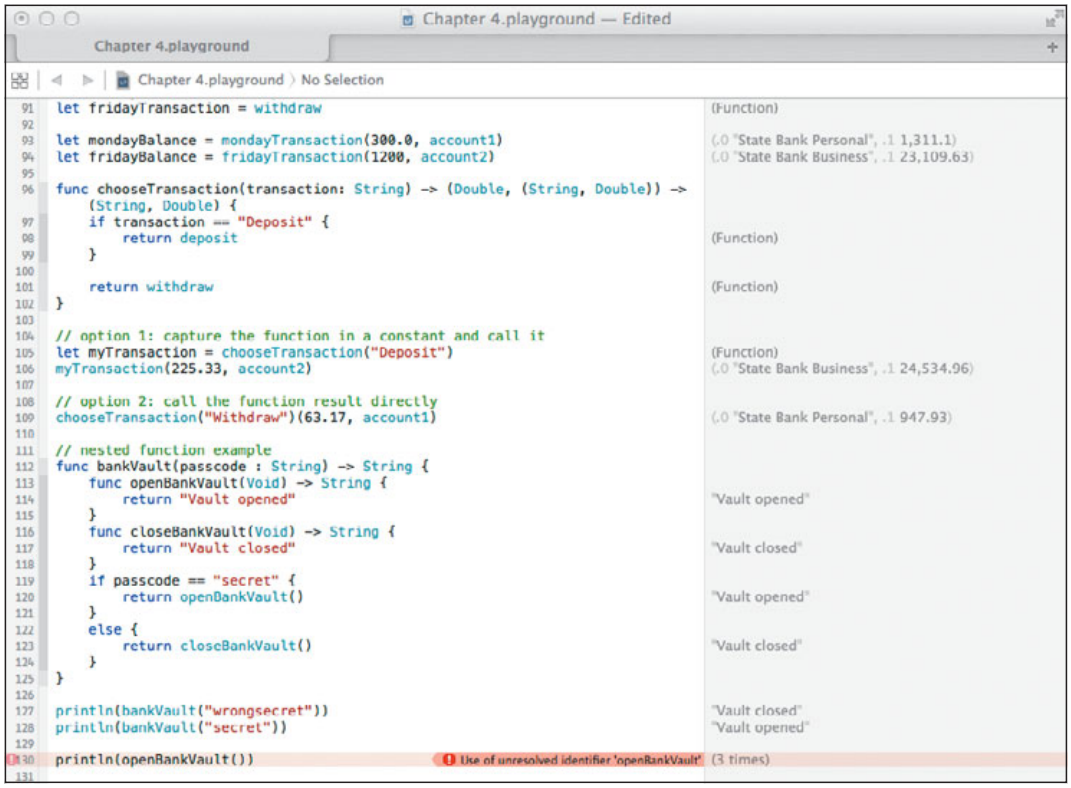
```
    func closeBankVault(Void) -> String {
        return "Vault closed"
    }
    if passcode == "secret" {
        return openBankVault()
    }
    else {
        return closeBankVault()
    }
}

println(bankVault("wrongsecret"))
println(bankVault("secret"))
```

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 of 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.

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



INTO THE VOID

On lines 113 and 116, 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. However, be aware of it because you will be seeing more of it later.

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.

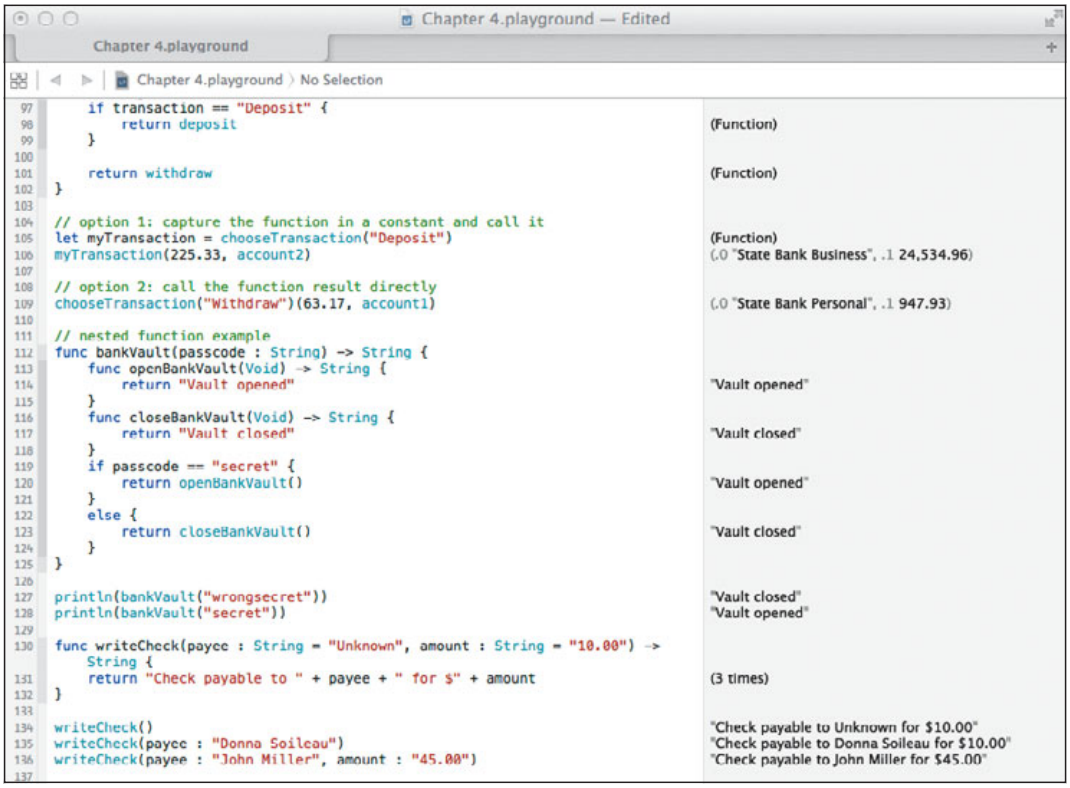


FIGURE 4.13 Using default parameters in a function

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 another interesting feature called *default parameter values*, which allow you to declare functions that have parameters containing a "prefilled" value.

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

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

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

```
func writeCheck(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.

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.

Another observation: When calling a function set up to accept default parameters, you must pass the parameter name and a colon as part of that parameter. On line 135, only one parameter is used:

```
writeCheck(payee : "Donna Soileau")
```

And on line 136, both parameter names are used:

```
writeCheck(payee : "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.

Each parameter in a Swift function can have an optional *external parameter* preceding the parameter name. External names give additional clarity and description to a function name. Consider another check writing function in **Figure 4.14**, lines 138 through 140.

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

writeCheck("Dave Johnson", "Coz Fontenot", 1000.0)
```

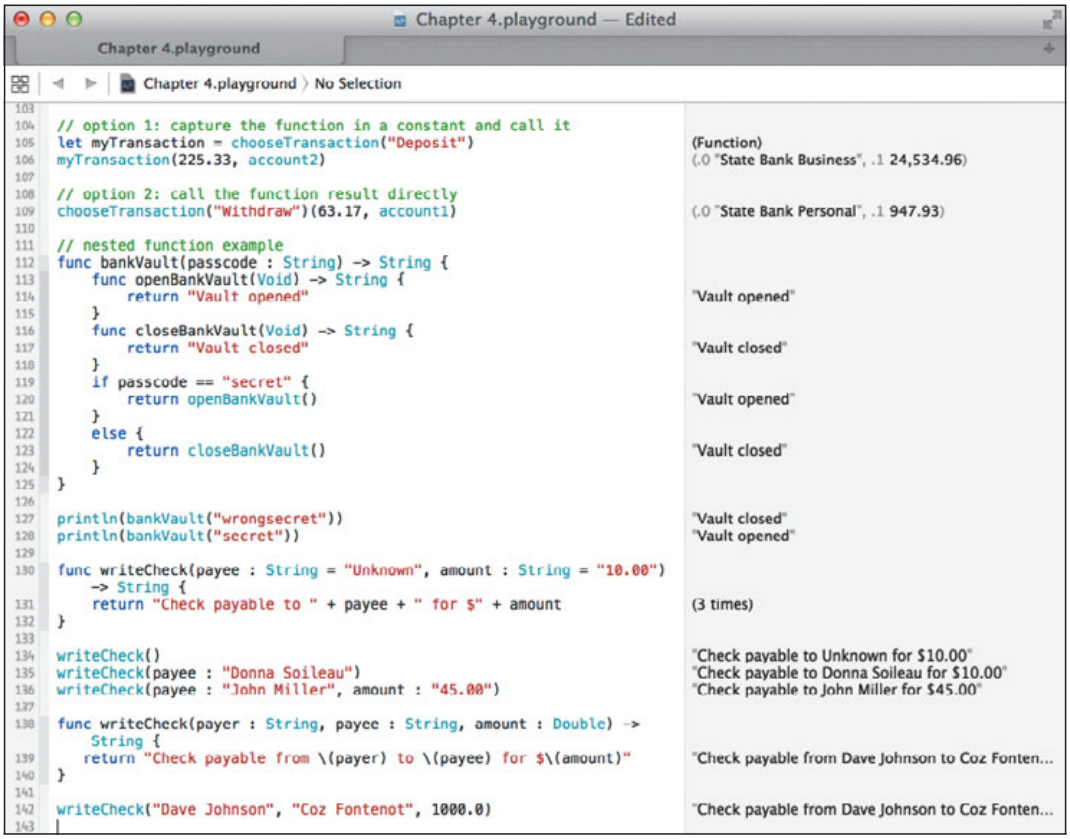


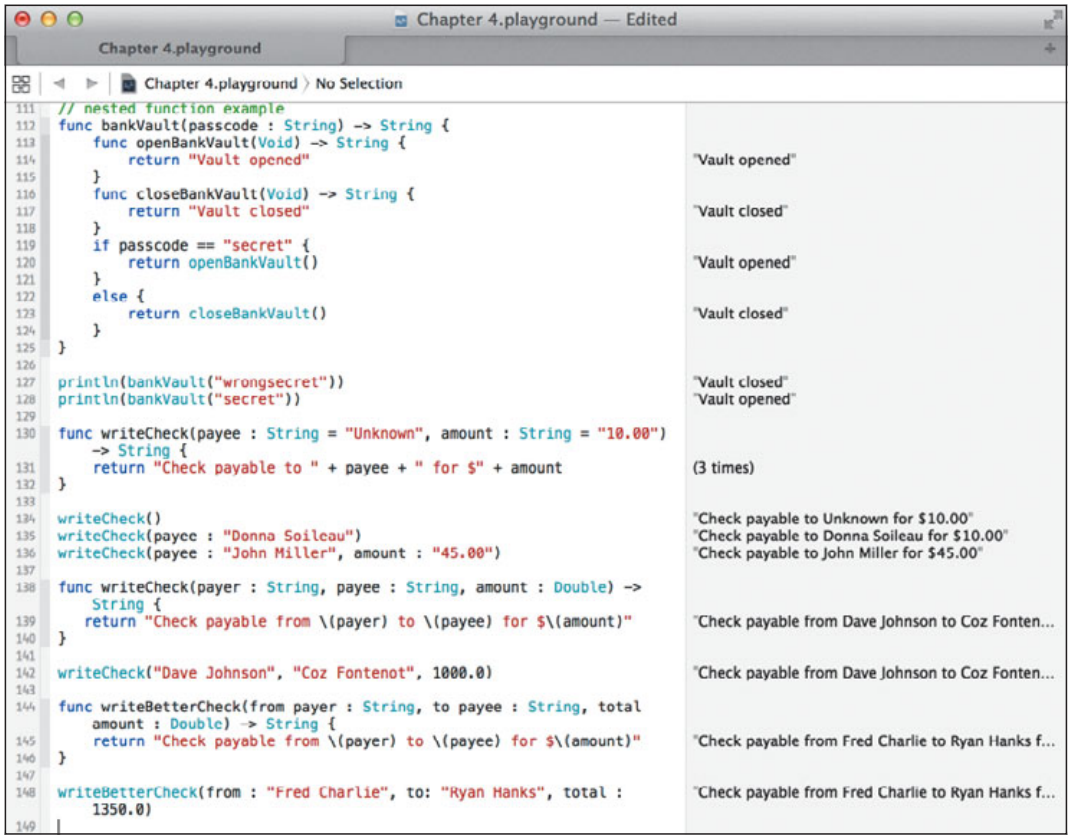
FIGURE 4.14 A function without external parameter names

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

- An additional parameter named payer to indicate who the check is coming from
- No default parameters

On line 142, the new `writeCheck` 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 for how much? 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?

FIGURE 4.15 A function with external parameter names



External parameter names solve this problem by adding an additional name to each parameter that *must* be passed when calling the function, which makes very clear to anyone reading the calling function what the intention is and the purpose of each parameter. **Figure 4.15** illustrates this quite well.

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

writeBetterCheck(from : "Fred Charlie", to: "Ryan Hanks", total : 1350.0)
```

On line 144, you declare a function, `writeBetterCheck`, which takes the same number of parameters as the function on line 138. However, each of the parameters in the new function now has its own *external* parameter: `from`, `to`, and `total`. The original parameter names are still there, of course, used inside the function itself to reference the assigned values.

This extra bit of typing pays off on line 148, when the `writeBetterCheck` 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.

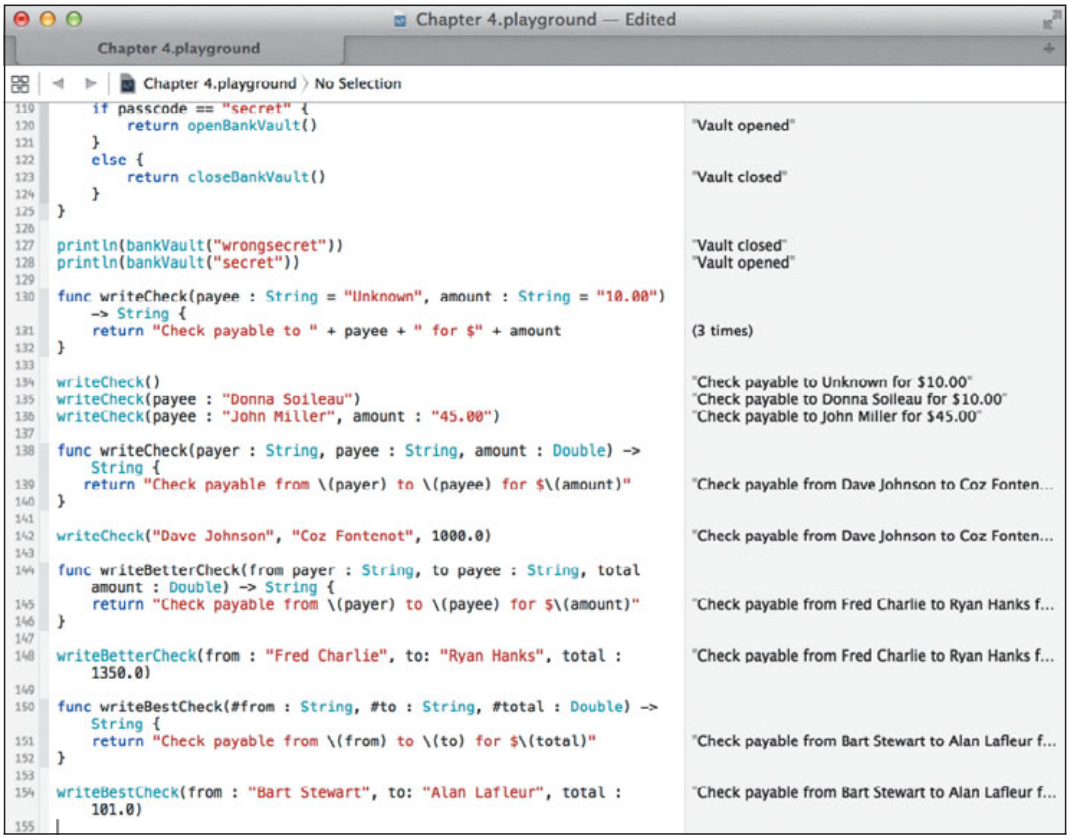


FIGURE 4.16 Using the shorthand external parameter syntax

WHEN IT'S GOOD ENOUGH

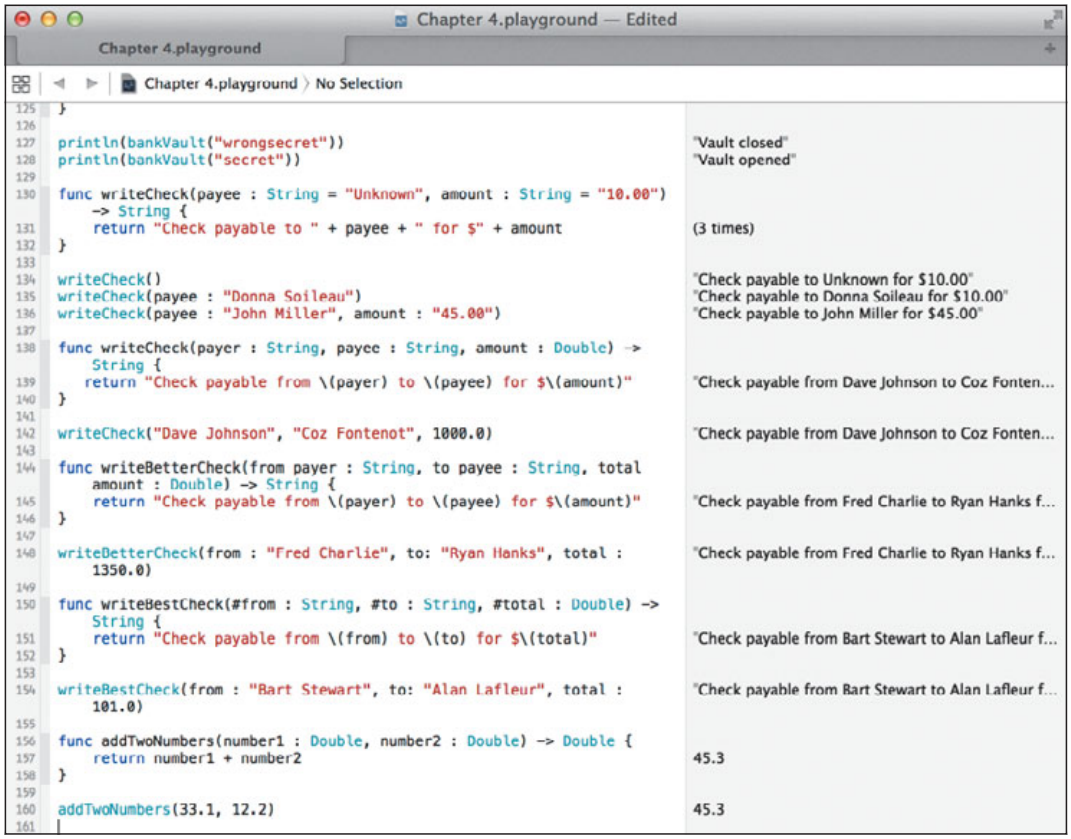
External parameter names bring clarity to functions, but they can feel somewhat redundant and clumsy as you search for something to accompany the parameter name. Actually, you may find that in certain cases, the parameter name is descriptive enough to act as the external parameter name. Swift allows you to use the parameter name as an external name, too, with a special syntax: the # character.

Instead of providing an external parameter name, simply prepend the # character to the parameter name, and use that name when calling the new function `writeBestCheck`, as done on line 150 in **Figure 4.16**. This is known as *shorthand external parameter naming*.

The three parameter names, `from`, `to`, and `total`, all are prepended with #. On line 154, the parameter names are used as external parameter names once again 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.

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

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

FIGURE 4.17 When external parameter names are not necessary



TO USE OR NOT TO USE?

External parameter names bring clarity to functions, but they also require more typing on the part of the caller 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 additional clarity of having external 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**), external parameter names add little to nothing of value for the caller.

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

addTwoNumbers(33.1, 12.2)
```

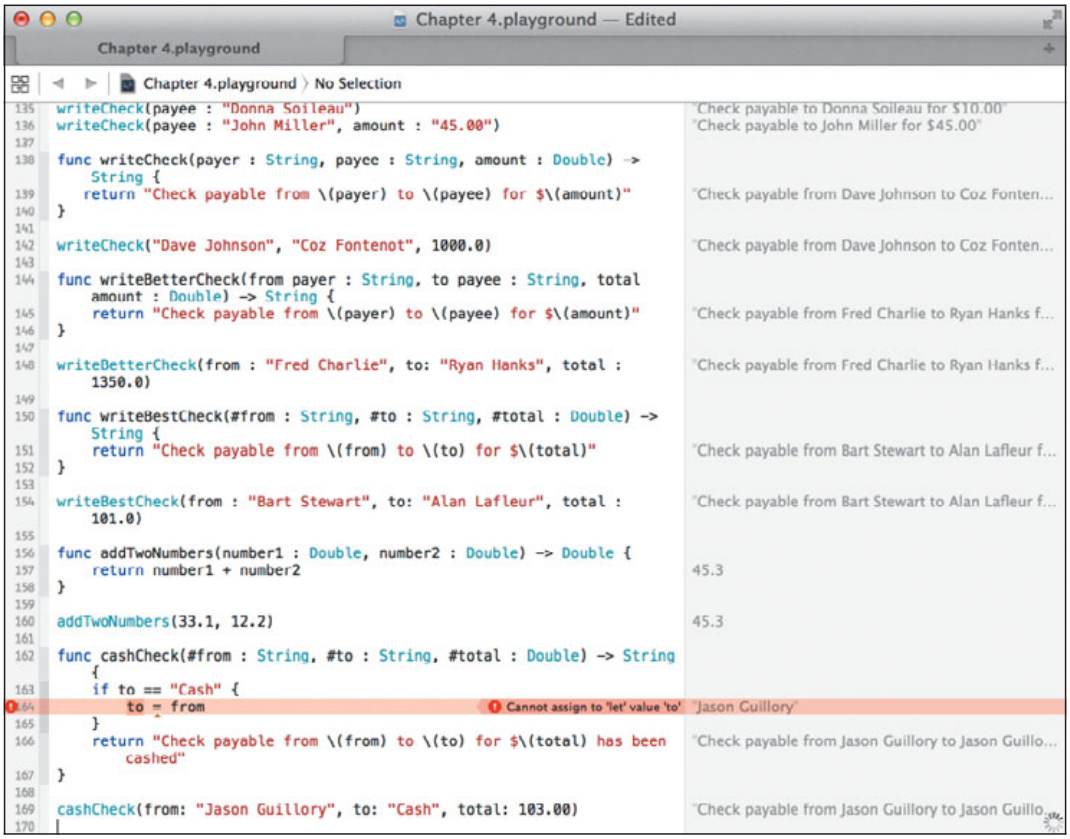


FIGURE 4.18 Assigning a value to a parameter results in an error.

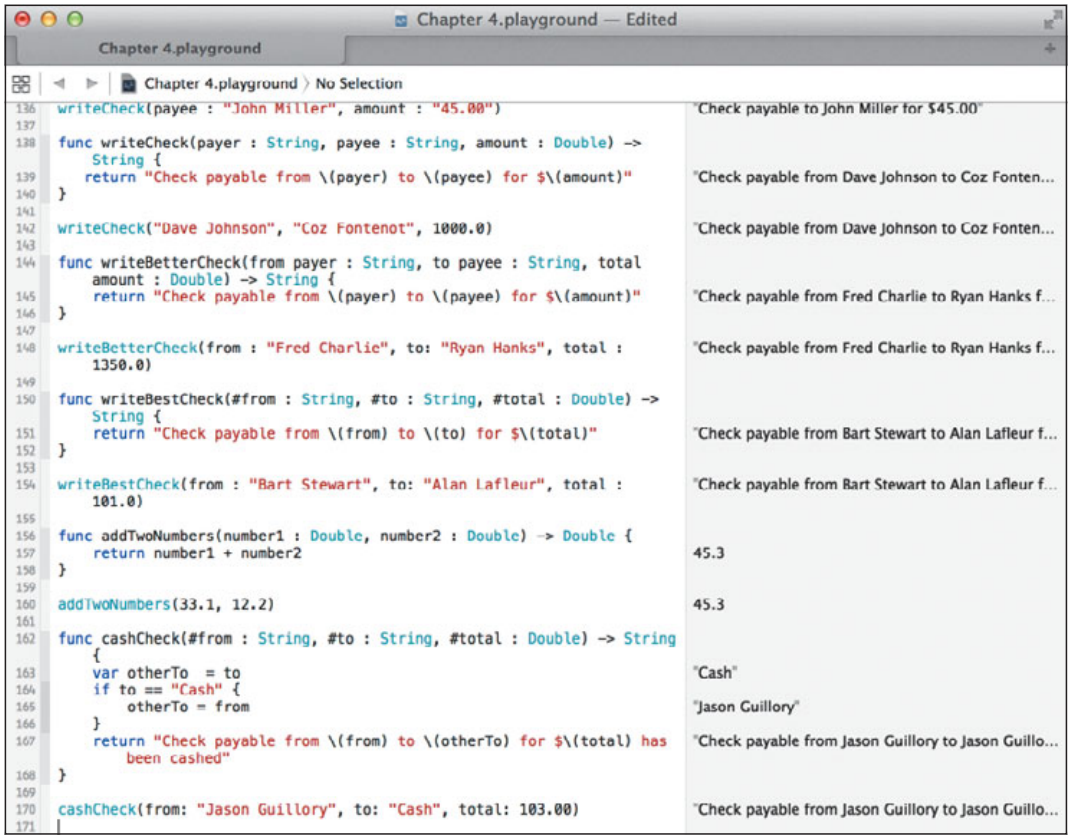
DON'T CHANGE MY PARAMETERS!

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

The function takes the same parameters as our 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.

FIGURE 4.19
A potential workaround to the parameter change problem



Notice the error: **Cannot assign to 'let' value 'to'**. 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.

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 assuming the condition on line 164 is met. This is clearly acceptable and works fine for our purposes, but Swift gives you a better way.

With a `var` declaration on a parameter, you can tell Swift 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.

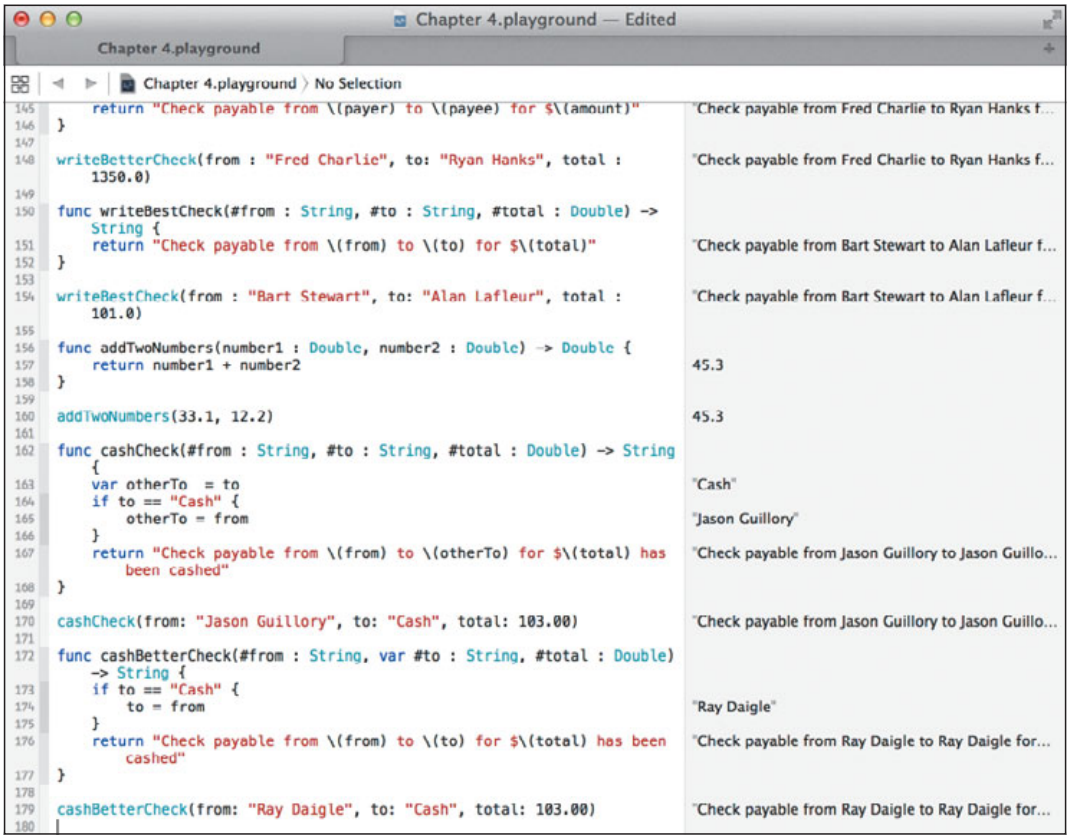
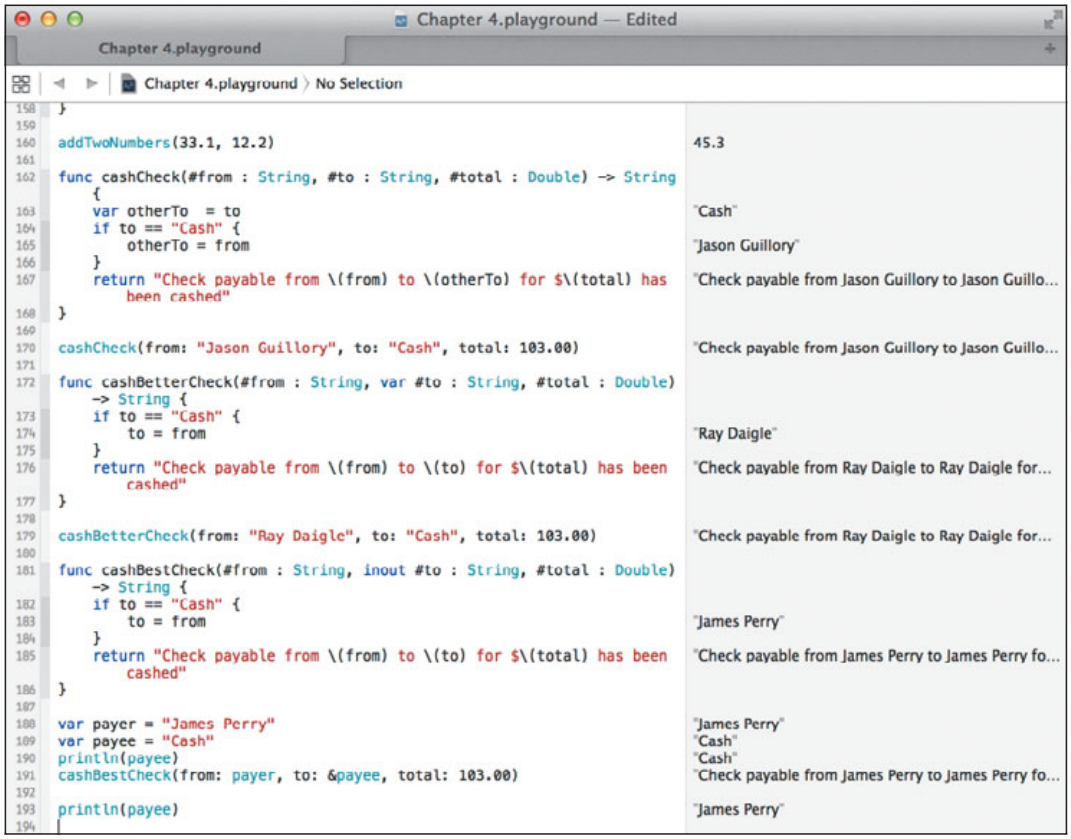


FIGURE 4.20 Using variable parameters to allow modifications

```
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(from: "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"
println(payee)
cashBestCheck(from: payer, to: &payee, total: 103.00)

println(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 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, we call the `cashBestCheck` function. Unlike the call to `cashBetterCheck` on line 179, we 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 in-out 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 189 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, that will change as you proceed through the book. Functions are going to appear over and over again both here and in your coding, so understand them well. You may want to re-read this chapter to retain all the ins and outs of functions.

We've got a little more to talk about before we close this chapter, however. Our 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, which "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.

In essence, they 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, 3.875, 5, simpleInterestCalculationClosure)
```

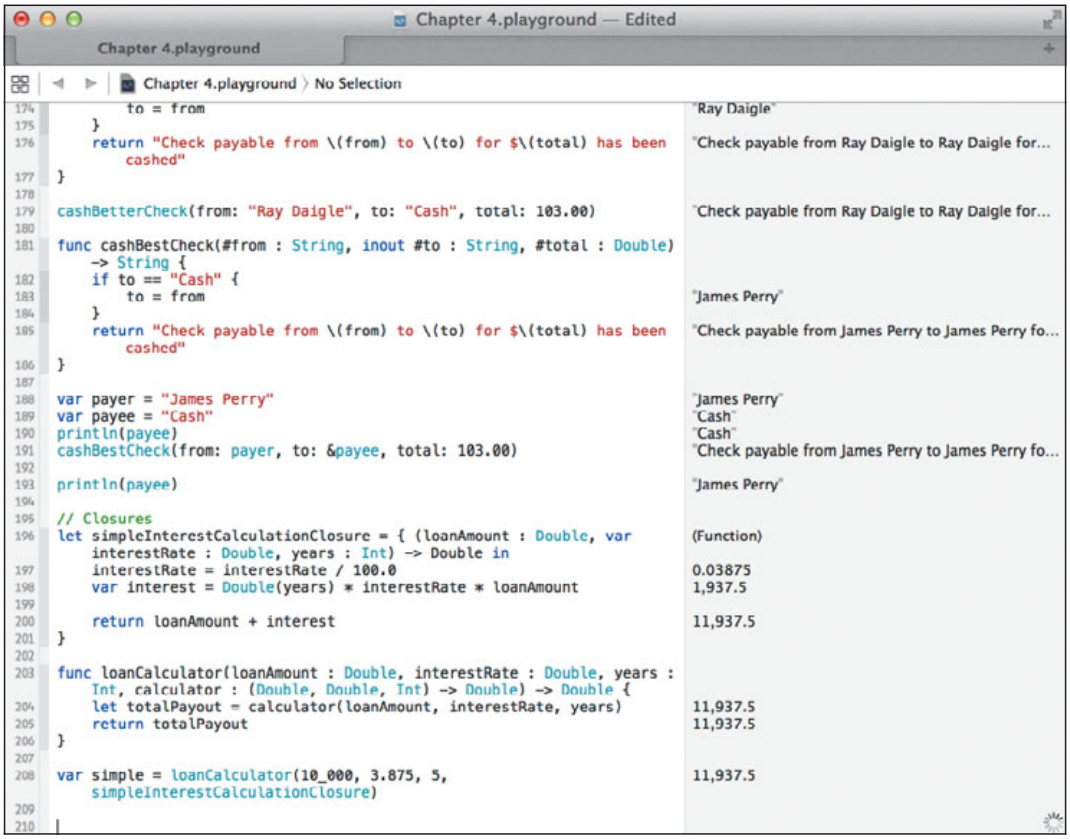


FIGURE 4.22 Using a closure to compute simple interest

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 our 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 computed simple interest, then write a closure that computes the future value of money using the compound interest formula:

$$futureValue = presentValue\ (1 + interestRate)^{years}$$

FIGURE 4.23 Adding a second closure that computes compound interest

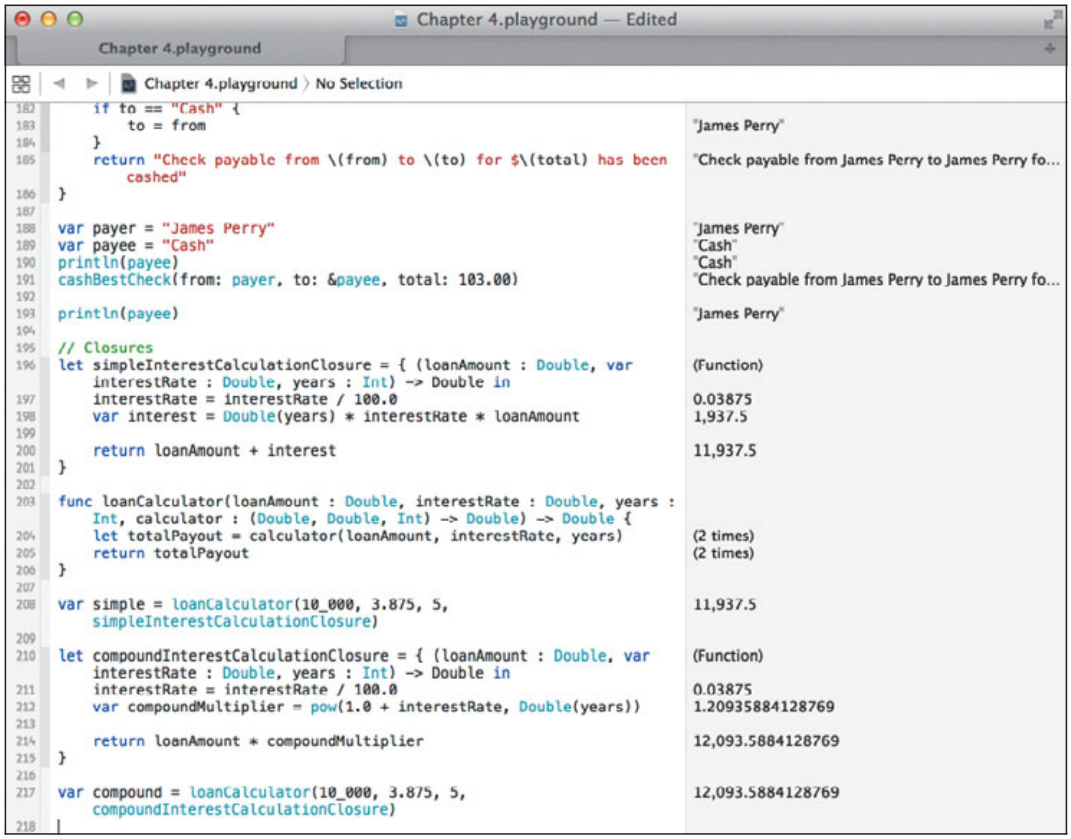


Figure 4.23 shows the compound interest calculation closure defined on lines 210 through 215, which takes the exact same parameters as the simple calculation closure on line 196. On line 217, the `loanCalculator` function is again called with the same parameters as before, except the `compoundInterestCalculationClosure` 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, 3.875, 5, compoundInterest
→ CalculationClosure)
```

On line 212 you may notice something new: a reference to a function named `pow`. This is the power function, and 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

We'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; we'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, we'll turn our 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, we'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 we'll get down and dirty with classes.

This page intentionally left blank

INDEX

SYMBOLS

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X