Learning Swift
Second Edition

Swift is Apple's new programming language and the future of iOS and OS X app development. It is a high-performance language that feels like a modern scripting language. Swift is easy to jump into, but its complex underpinnings are critical to becoming proficient at turning an idea into reality.

This book begins by giving you an overview of the key features through practical examples and progresses to more advanced topics. It covers important concepts such as Variables, Optionals, Closures, Generics, and Memory Management. Mixed in with those concepts, it also helps you learn the art of programming, such as maintainability, useful design patterns, and resources to further your knowledge. This all culminates in writing a basic iOS app that will get you well on your way to turning your own app ideas into reality.

What you will learn from this book

- Form a solid understanding of the Swift 2 language
- Get to know the practical aspects of how a computer program actually works
- Understand the paradigms used by Apple’s frameworks so you are not intimidated by them
- Utilize the vast resources written in Objective-C to better inform your Swift programming
- Develop a basic portfolio of Swift code by learning the critical concepts
- Experience both object-oriented and functional programming
- Get to know the new coding techniques made available by Swift 2
- Discover resources to ensure you never stop becoming a better developer

Who this book is written for

If you are looking to build iOS or OS X apps using the most modern technology, this book is ideal for you. You will find this book especially useful if you are new to programming or if you are yet to develop for iOS or OS X. No prior programming experience is required.

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 4 'To Be or Not To Be – Optionals'
- A synopsis of the book’s content
- More information on *Learning Swift Second Edition*
About the Author

Andrew J Wagner is a software developer who concentrates on iOS development and backend web services. He has a degree in computer engineering from Rensselaer Polytechnic Institute, New York. Currently, he works for a development shop named Chronos Interactive based in Denver, CO. He has experience of working with and for large-scale and small-scale companies, as well as running his own contracting and app companies. He is passionate about using computers as a creative outlet and writing software that is beautiful in implementation, functionality, and experience.

When he isn't working or spending time with friends and family, he writes for his blog at http://drewag.me.
Preface

This book will help you to get started with Swift in no time. It helps you understand the nuances of iOS programming not only from a conceptual but also from an implementation perspective. This book is an invaluable resource if you are looking forward to exploring the world of iOS application programming.

What this book covers

Chapter 1, *Introducing Swift*, will take the reader through the process of installing Swift and running their first Swift program, in order to expose its power right away.

Chapter 2, *Building Blocks - Variables, Collections, and Flow Control*, introduces you to the various built-in mechanisms Swift has for representing complex information in expressive and accessible ways, with the help of a real-world example.

Chapter 3, *One Piece at a Time - Types, Scopes, and Projects*, introduces the tools necessary to closely model the real world with code. It will teach you how to define your own custom types using structures, classes, and enumerations. It also explores the concept of scope and access control.

Chapter 4, *To Be or Not To Be - Optionals*, focuses on a special and critical type in Swift, called optionals. It includes a detailed explanation of how optionals work and how they can be used, which turns a seemingly complex topic into a very intuitive concept.

Chapter 5, *A Modern Paradigm - Closures and Functional Programming*, introduces you to a new way of thinking about code called functional programming. We learn how Swift supports this technique and how we can apply it to our programs to make it even more understandable and expressive.

Chapter 6, *Make Swift Work For You - Protocols and Generics*, describes what generics and protocols are and how they can provide power and safety at the same time.
Chapter 7, *Everything Is Connected - Memory Management*, dives deeper into the inner-workings of Swift. We discuss how a computer stores information and how we can use that knowledge in combination with some new tools in Swift, to ensure that our code remains responsive and minimizes its effect on battery life.

Chapter 8, *Paths Less Traveled – Error Handling*, goes into gracefully handling error situations in Swift with error throwing and catching.

Chapter 9, *Writing Code the Swift Way - Design Patterns and Techniques*, introduces the reader to the art of programming by taking them through a number of specific design patterns that help reduce the complexity of code.

Chapter 10, *Harnessing the Past - Understanding and Translating Objective-C*, develops a basic understanding of Objective-C with a focus on how it compares to Swift. This allows the reader to make use of the vast resources that exist in Objective-C to help with their Swift development.

Chapter 11, *A Whole New World - Developing an App*, focuses on explaining the process of creating a real world iOS application, with the help of an example.

Chapter 12, *What’s Next? - Resources, Advice, and the Next Steps*, discusses how to move forward to become the best app developer you possibly can. It provides a list of resources and advice the reader can use to continue their Swift and app development learning process.
As we discussed in Chapter 2, Building Blocks – Variables, Collections, and Flow Control, all variables and constants must always have a value before they are used. This is a great safety feature because it prevents you from creating a scenario where you forget to give a variable an initial value. It may make sense for some number variables, such as the number of sandwiches ordered to start at zero, but it doesn't make sense for all variables. For example, the number of bowling pins standing should start at 10, not zero. In Swift, the compiler forces you to decide what the variable should start at, instead of providing a default value that could be incorrect.

However, there are other scenarios where you will have to represent the complete absence of a value. A great example is if you have a dictionary of word definitions and you try to lookup a word that isn’t in the dictionary. Normally, this will return a String, so you could potentially return an empty String, but what if you also need to represent the idea that a word exists without a definition? Also, for another programmer who is using your dictionary, it will not be immediately obvious what will happen when they look up a word that doesn't exist. To satisfy this need to represent the absence of a value, Swift has a special type called an **optional**.

In this chapter, we will cover the following topics:

- Defining an optional
- Unwrapping an optional
- Optional chaining
- Implicitly unwrapped optionals
- Debugging optionals
- The underlying implementation
Defining an optional

So we know that the purpose of optionals in Swift is to allow the representation of the absence of a value, but what does that look like and how does it work? An optional is a special type that can “wrap” any other type. This means that you can make an optional `String`, optional `Array`, and so on. You can do this by adding a question mark (`?`) to the type name, as shown:

```swift
var possibleString: String?
var possibleArray: [Int]?
```

Note that this code does not specify any initial values. This is because all optionals, by default, are set to no value at all. If we want to provide an initial value we can do so similar to any other variable:

```swift
var possibleInt: Int? = 10
```

Also, note that if we left out the type specification (`: Int?`), `possibleInt` would be inferred to be of type `Int` instead of an optional `Int`.

Now, it is pretty verbose to say that a variable lacks a value. Instead, if an optional lacks a variable, we say it is `nil`. So both `possibleString` and `possibleArray` are `nil`, while `possibleInt` is `10`. However, `possibleInt` is not truly `10`. It is still wrapped in an optional.

You can see all the forms a variable can take by putting the following code into a playground:

```swift
var actualInt = 10
var possibleInt: Int? = 10
var nilInt: Int?
    print(actualInt) // 10
    print(possibleInt) // Optional(10)
    print(nilInt) // nil
```

As you can see, `actualInt` prints out just as we expected, but `possibleInt` prints out as an optional that contains the value `10` instead of just `10`. This is a very important distinction because an optional cannot be used as the value it is wrapping. `nilInt` just reports that it is `nil`. At any point, you can update the value within an optional; this includes assigning it a value for the first time, using the assignment operator (`=`):

```swift
nilInt = 2
print(nilInt) // Optional(2)
```
You can even remove the value within an optional by assigning it to nil:

```swift
nilInt = nil
print(nilInt) // nil
```

So we have this wrapped form of a variable that may or may not contain a value. What do we do if we need to access the value within an optional? The answer is that we must unwrap it.

### Unwrapping an optional

There are multiple ways to unwrap an optional. All of them essentially assert that there is truly a value within the optional. This is a wonderful safety feature of Swift. The compiler forces you to consider the possibility that an optional lacks any value at all. In other languages, this is a very commonly overlooked scenario that can cause obscure bugs.

### Optional binding

The safest way to unwrap an optional is to use something called **optional binding**. With this technique, you can assign a temporary constant or variable to the value contained within the optional. This process is contained within an `if` statement, so that you can use an else statement when there is no value. Optional binding looks similar to the following code:

```swift
if let string = possibleString {
    print("possibleString has a value: \(string)")
} else {
    print("possibleString has no value")
}
```

An optional binding is distinguished from an `if` statement primarily by the `if let` syntax. Semantically, this code is saying, "if you can let the constant `string` be equal to the value within `possibleString`, print out its value; otherwise, print that it has no value." The primary purpose of an optional binding is to create a temporary constant that is the normal (non-optional) version of the optional.

We can also use a temporary variable in an optional binding:

```swift
possibleInt = 10
if var actualInt = possibleInt {
    actualInt *= 2
    print(actualInt) // 20
}
print(possibleInt) // Optional(10)
```
Note that an asterisk (*) is used for multiplication in Swift. You should also notice something important about this code. If you put it into a playground, even though we multiplied the `actualInt` by 2, the value within the optional does not change. When we print out `possibleInt` later, the value is still `Optional(10)`. This is because even though we made `actualInt` a variable (otherwise known as mutable), it is simply a temporary copy of the value within `possibleInt`. No matter what we do with `actualInt`, nothing will get changed about the value within `possibleInt`. If we have to update the actual value stored within `possibleInt`, we simply assign `possibleInt` to `actualInt` after we are done modifying it:

```swift
possibleInt = 10
if var actualInt = possibleInt {
    actualInt *= 2
    possibleInt = actualInt
}
print(possibleInt) // Optional(20)
```

Now, the value wrapped inside `possibleInt` has actually been updated.

A common scenario that you will probably come across is the need to unwrap multiple optional values. One option is to simply nest the optional bindings:

```swift
if let actualString = possibleString {
    if let actualArray = possibleArray {
        if let actualInt = possibleInt {
            print(actualString)
            print(actualArray)
            print(actualInt)
        }
    }
}
```

However, this can be a pain, as it increases the indentation level each time to keep the code organized. Instead, you can actually list multiple optional bindings into a single statement separated by commas:

```swift
if let actualString = possibleString,
    let actualArray = possibleArray,
    let actualInt = possibleInt
{
    print(actualString)
    print(actualArray)
    print(actualInt)
}
```
This generally produces more readable code.

Another great way to do a concise optional binding within functions is to use the guard statement. This way, you can do a series of unwrapping without increasing the indent level of the code at all:

```swift
func someFunc2() {
    guard let actualString = possibleString,
        let actualArray = possibleArray,
        let actualInt = possibleInt
    else {
        return
    }

    print(actualString)
    print(actualArray)
    print(actualInt)
}
```

This construct allows us to access the unwrapped values after the guard statement, because the guard statement guarantees that we would have exited the function before reaching that code, if the optional value was nil.

This way of unwrapping is great, but saying that optional binding is the safest way to access the value within an optional, implies that there is an unsafe way to unwrap an optional. This way is called **forced unwrapping**.

**Forced unwrapping**

The shortest way to unwrap an optional is to use forced unwrapping. It is done using an exclamation mark (!) after the variable name when being used:

```swift
possibleInt = 10
possibleInt! *= 2
print(possibleInt) // "Optional(20)"
```

However, the reason it is considered unsafe is that your entire program will crash if you try to unwrap an optional that is currently nil:

```swift
nilInt! *= 2 // fatal error
```

The complete error you get is **unexpectedly found nil while unwrapping an optional value**. This is because the forced unwrapping is essentially your personal guarantee that the optional truly does hold a value. That is why it is called "forced".
Therefore, forced unwrapping should be used in limited circumstances. It should never be used just to shorten up the code. Instead, it should only be used when you can guarantee from the structure of the code that it cannot be nil, even though it is defined as an optional. Even in that case, you should see if it is possible to use a non-optional variable instead. The only other place you may use it is if your program truly could not recover from an optional being nil. In those circumstances, you should at least consider presenting an error to the user, which is always better than simply having your program crash.

An example of a scenario where it may be used effectively is with lazily calculated values. A lazily calculated value is the one that is not created until the first time it is accessed. To illustrate this, let's consider a hypothetical class that represents a file system directory. It will have a property listing its contents that is lazily calculated. The code will look similar to the following code:

```swift
class FileSystemItem {}
class File: FileSystemItem {}
class Directory: FileSystemItem {
    private var realContents: [FileSystemItem]?
    var contents: [FileSystemItem] {
        if self.realContents == nil {
            self.realContents = self.loadContents()
        }
        return self.realContents!
    }

    private func loadContents() -> [FileSystemItem] {
        // Do some loading
        return []
    }
}
```

Here, we have defined a superclass called `FileSystemItem` that both `File` and `Directory` inherit from. The content of a directory is a list of `FileSystemItem`. We define `contents` as a calculated variable and store the real value within the `realContents` property. The calculated property checks if there is a value loaded for `realContents`; if there isn't, it loads the contents and puts them into the `realContents` property. Based on this logic, we know for 100% certainty that there will be a value within `realContents` by the time we get to the return statement, so it is perfectly safe to use forced unwrapping.
Nil coalescing

In addition to optional binding and forced unwrapping, Swift also provides an operator called the **nil coalescing operator** to unwrap an optional. This is represented by a double question mark (??). Basically, this operator lets us provide a default value for a variable or operation result, in case it is nil. This is a safe way to turn an optional value into a non-optional value and it would look similar to the following code:

```swift
var possibleString: String? = "An actual string"
print(possibleString ?? "Default String") // "An Actual String"
```

Here, we are asking the program to print out `possibleString` unless it is nil; in which case, it will just print "Default String". Since we did give it a value, it printed out that value and it is important to note that it printed out as a regular variable, not an optional. This is because one way or another, an actual value was going to be printed.

This is a great tool for concisely and safely unwrapping an optional when a default value makes sense.

Optional chaining

A common scenario in Swift is to have an optional that you must calculate something from. If the optional has a value, you will want to store the result of the calculation on it, but if it is nil, the result should just be set to nil:

```swift
var invitee: String? = "Sarah"
var uppercaseInvitee: String?
if let actualInvitee = invitee {
    uppercaseInvitee = actualInvitee.uppercaseString
}
```

This is pretty verbose. To shorten this up in an unsafe way, we could use forced unwrapping:

```swift
uppercaseInvitee = invitee!.uppercaseString
```

However, optional chaining will allow us to do this safely. Essentially, it allows optional operations on an optional. When the operation is called, if the optional is nil, it immediately returns nil; otherwise, it returns the result of performing the operation on the value within the optional:

```swift
uppercaseInvitee = invitee?.uppercaseString
```
To Be or Not To Be – Optionals

So in this call, `invitee` is an optional. Instead of unwrapping it, we use optional chaining by placing a question mark (?) after it, followed by the optional operation. In this case, we are asking for the `uppercaseInvitee` property on it. If `invitee` is `nil`, `uppercaseInvitee` is immediately set to `nil` without even trying to access `uppercaseString`. If it actually does contain a value, `uppercaseInvitee` gets set to the `uppercaseString` property of the contained value. Note that all optional chains return an optional result.

You can chain as many calls as you want, both optional and non-optional, together in this way:

```swift
var invitees: [String]? = ["Sarah", "Jamison", "Marcos", "Roana"]
invitees?.first?.uppercaseString.hasPrefix("A")
```

This code checks if the first element of the `invitees`-list starts with the letter `A`, even if it is a lowercase `a`. First, it uses an optional chain in case `invitees` is `nil`. Then the call to `first` uses an additional optional chain because that method returns an optional `String`. We then call `uppercaseString`, which does not return an optional, allowing us to access `hasPrefix` on the result without having to use another optional chain. If at any point any of the optionals are `nil`, the result will be `nil`. This can happen for two different reasons:

- `invitees` is `nil`
- `first` returns `nil` because the array is empty

If the chain makes it all the way to `uppercaseString`, there is no longer a failure path and it will definitely return an actual value. You will notice that there are exactly two question marks being used in this chain and there are two possible failure reasons.

At first, it can be hard to understand when you should and should not use a question mark to create a chain of calls; the rule is to always use a question mark if the previous element in the chain returns an optional. However, so you are prepared, let’s take a look at what happens if you use an optional chain improperly:

```swift
invitees?.first // Error
```

In this case, we try to call a method directly on an optional without a chain, so we get an error that says **Value of optional type `[String]` not unwrapped; did you mean to use `!' or `?'**. Not only does it tell us that the value is not unwrapped, it even suggests two common ways of dealing with the problem: forced unwrapping or optional chaining.
We also have the case where we try to use an optional chain inappropriately:

```swift
var otherInvitees = ["Kai", "Naya"]
otherInvitees?.first // Error
```

Here, we get an error that says **Cannot use optional chaining on non-optional value of type `[String]`**. It is great to have a good sense of the errors you might see when you make mistakes; so that you can correct them quickly because we all make silly mistakes from time-to-time.

Another great feature of optional chaining is that it can be used for method calls on an optional that does not actually return a value:

```swift
invitees?.removeAll()
```

In this case, we only want to call `removeAll` if there is truly a value within the optional array. So with this code, if there is a value, all the elements are removed from it; otherwise, it remains nil.

In the end, option chaining is a great choice for writing a concise code that still remains expressive and understandable.

**Implicitly unwrapped optionals**

There is a second type of optional called an **implicitly unwrapped optional**. There are really two ways to look at what an implicitly unwrapped optional is; one way is to say that it is a normal variable that can also be nil; the other way is to say that it is an optional that you don't have to unwrap to use. The important thing to understand about them is that, similar to optionals, they can be nil, but you do not have to unwrap them like a normal variable.

You can define an implicitly unwrapped optional with an exclamation mark (`!`) instead of a question mark (`?`) after the type name:

```swift
var name: String!
```

Similar to regular optionals, implicitly unwrapped optionals do not need to be given an initial value because they are nil by default.

At first it may sound like it is the best of both worlds, but in reality it is more like the worst of both worlds. Even though an implicitly unwrapped optional does not have to be unwrapped, it will crash your entire program if it is nil when used:

```swift
name.uppercaseString // Crash
```
A great way to think about them is that every time it is used, it is implicitly doing a forced unwrapping. The exclamation mark is placed in its type declaration, instead of each time it is used. This can be problematic because it appears the same as any other variable except for how it is declared. That means it is very unsafe to use, unlike a normal optional.

So if the implicitly unwrapped optionals are the worst of both worlds and are so unsafe, why do they even exist? The reality is that in rare circumstances, they are necessary. They are used in circumstances where a variable is not truly optional, but you also cannot give an initial value to it. This is almost always the case for custom types that have a member variable that is non-optional but cannot be set during initialization.

A rare example of this is with a view in iOS. UIKit, as we discussed before, is the framework Apple provides for iOS development. In it, Apple has a class called UIView that is used to display content on the screen. Apple also provides a tool in Xcode called Interface Builder that lets you design these views in a visual editor instead of in code. Many views designed in this way will need references to other views that can be accessed later, programmatically. When one of these views is loaded, it is initialized without anything connected and then all the connections are made. Once all of the connections are made, a function called awakeFromNib is called on the view. This means that these connections are not available to be used during initialization but are available once awakeFromNib is called. This order of operations also ensures that awakeFromNib is always called before anything actually uses the view. This is a circumstance where it is necessary to use an implicitly unwrapped optional. A member variable may not be able to be defined until after the view is initialized, when it is completely loaded:

```swift
import UIKit
class MyView: UIView {
    @IBOutlet var button: UIButton!
    var buttonOriginalWidth: CGFloat!

    override func awakeFromNib() {
        self.buttonOriginalWidth = self.button.frame.size.width
    }
}
```
Notice that we have actually declared two implicitly unwrapped optionals. The first is a connection to a button. We know that this is a connection because it is preceded by @IBOutlet. This is declared as an implicitly unwrapped optional because connections are not set up until after initialization, but they are still guaranteed to be set up before any other methods are called on the view.

This then leads us to unwrapping our second variable, buttonOriginalWidth, implicitly because we need to wait until the connection is made before we can determine the width of the button. After awakeFromNib is called, it is safe to treat both button and buttonOriginalWidth as non-optional.

You may have noticed that we had to dive pretty deep into app development to find a valid use case for implicitly unwrapped optionals and this is arguably only because UIKit is implemented in Objective-C, as we will learn more about in Chapter 10, Harnessing the Past – Understanding and Translating Objective-C. This is another testament to the fact that they should be used sparingly.

**Debugging optionals**

We have already seen a couple of the compiler errors we will commonly see because of optionals. If we try to call a method on an optional that we intended to call on the wrapped value, we will get an error. If we try to unwrap a value that is not actually optional, we will also get an error. We also need to be prepared for the runtime errors that optionals can cause.

As we have discussed, optionals cause runtime errors that are also referred to as crashes, if you try to forcefully unwrap one that is nil. This can happen with both explicit and implicitly forced unwrapping. If you have followed my advice so far in this chapter, this should be a rare occurrence. However, we all end up working with a third party code and maybe they were lazy or maybe they use forced unwrapping to enforce their expectations about how their code should be used.
Also, we all suffer from being lazy from time to time. It can be exhausting or discouraging to worry about all the edge cases when you are excited about programming the core functionality of your app. We may use forced unwrapping temporarily while we worry about that main functionality and plan to come back to handle it later. After all, during development it is better to have a forced unwrapping crash the development version of your app than it is for it to fail silently if you have not yet handled that edge case. We may even decide that an edge case is not worth the development effort of handling because everything about developing an app is a trade off. Either way, we need to recognize a crash from forced unwrapping quickly so we don't waste extra time trying to figure out what went wrong.

When an app tries to unwrap a nil value, if you are currently debugging the app, Xcode will show you the line that is trying to do the unwrapping. The line will report that there was an `EXC_BAD_INSTRUCTION` error and you will also get a message in the console saying `fatal error: unexpectedly found nil while unwrapping an Optional value`:

![Image showing Xcode call stack](image)

You will also sometimes have to look at what code is currently calling the code that failed. To do that, you can use the call stack in Xcode. The call stack is the full path of all function calls that got to this location. So, if you have `function1` call `function2`, which then calls `function3`, `function3` will be at the top and `function1` will be at the bottom. Once the execution exits `function3`, it will be removed from the stack so you will just have `function2` on top of `function1`.

When your program crashes, Xcode will automatically display the call stack, but you can also manually show it by navigating to **View | Navigators | Show Debug Navigator**. It will look similar to the following screenshot:
Here, you can click around different levels of code to see the state of things. This will become even more important if the program is crashing within one of Apple's framework, where you do not have access to the code. In that case, you will want to move up the call stack to the point where your code called into the framework. You may also be able to look at the names of the functions to help you figure out what may have gone wrong.

Anywhere on the call stack, you can look at the state of the variables in the debugger, as shown:

If you do not see this variable's view, you can display it by clicking on the button in the bottom-right corner of the screen, second from the right that will be grayed out. Here, you can see that `invitee` is indeed nil, which is what caused the crash.
As powerful as the debugger is, if you find that it isn't helping you find the problem, you can always put print statements in important parts of the code. It is always safe to print out an optional, as long as you don't forcefully unwrap it as shown in the preceding example. As we have seen before, when an optional is printed, it will print nil if it doesn't have a value, or it will print Optional(<value>) if it has a value.

Debugging is an extremely important part of becoming a productive developer because we all make mistakes and create bugs. Being a great developer means that you can identify problems quickly and understand how to fix them soon after that. This will largely come from practice, but it will also come from having a firm grasp of what is really happening with your code versus simply adapting some code you find online to fit your needs through trial and error.

The underlying implementation

At this point, you should have a pretty strong grasp of what an optional is and how to use and debug it, but it will be valuable to look a little deeper at optionals to see how they actually work.

In reality, the question mark syntax for optionals is just special shorthand. Writing String? is equivalent to writing Optional<String>. Writing String! is equivalent to writing ImplicitlyUnwrappedOptional<String>. The Swift compiler has the shorthand versions because they are so commonly used. This allows the code to be more concise and readable.

If you declare an optional using the long form, you can see Swift's implementation by holding Command and clicking on the word Optional. Here, you can see that Optional is implemented as an enumeration. Simplifying the code a little, we have:

```swift
enum Optional<T> {
    case None
    case Some(T)
}
```

So we can see that an optional really has two cases: None and Some. None stands for the nil case, while the Some case has an associated value, which is the value wrapped inside the optional. Unwrapping is the process of retrieving the associated value out of the Some case.

The one part of this that you have not seen yet is the angled bracket syntax (<T>). This is called a generic and it essentially allows the enumeration to have an associated value of any type. We will cover generics in-depth in Chapter 6, Make Swift Work For You – Protocols and Generics.
Realizing that optionals are simply enumerations will help you understand how to use them. It also gives you some insight into how concepts are built on top of other concepts. Optionals seem really complex until you realize that they are just a two case enumeration. Once you understand enumerations, you can pretty easily understand optionals as well.

**Summary**

We have only covered a single concept, optionals, in this chapter, but we have seen that this is a pretty dense topic. We have seen that at the surface level, optionals are pretty straightforward. They are a way to represent a variable that has no value. However, there are multiple ways to get access to the value wrapped within an optional that have very specific use cases. Optional binding is always preferred, as it is the safest method, but we can also use forced unwrapping if we are confident that an optional is not nil. We also have a type called implicitly unwrapped optional, to delay the assigning of a variable that is not intended to be optional; however, we should use it sparingly because there is almost always a better alternative.

Now that we have a firm understanding of optionals, we can begin to look at something else that may appear minor on the surface but actually opens up a whole world of possibilities. All functions in Swift are actually variables or constants themselves. We will explore what this means in the next chapter.
Where to buy this book

You can buy Learning Swift (Second Edition) from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.