iOS Development with Swift

Jack D. Watson-Hamblin

MEAP

MANNING

MEAP Edition
Manning Early Access Program
iOS Development with Swift
Version 2

For more information on this and other Manning titles go to
www.manning.com

welcome

Thanks for purchasing the MEAP for *iOS 8 Development with Swift*. This book has been written to take most anyone from never doing iOS development to being a well-rounded iOS developer ready to start creating their first few apps.

Introducing you to programming in general and teaching you how to build an app would be too much for one book. This is why we assume you've been working with Object-Oriented Programming languages like Ruby, Java, or Python.

While working with start-ups trying to build the next big thing (of course), I was creating their back end services for the web app and iOS app. Suddenly our iOS developer was no longer available and the job of producing the app was thrust upon me. At that point iOS developers didn't have the tools or great documentation we have now, it was a tricky transition but an amazing change.

The path that has lead to writing this book has been long and full of some crazy jobs that have given me a wealth of experiences to share with you. For now we'll be sharing chapters 1-3. Chapter 2 is focused on making you familiar with Xcode 6, including creating interfaces with it. Chapters 1 & 3 will give you a solid foundation of Swift knowledge to get you ready for the rest of the book.

Throughout this book we will be covering the core aspects of iOS 8 and Swift. You will begin with understanding UIKit and Foundation, the two core iOS frameworks. After creating applications using these frameworks from Apple, you can dive deeper into other frameworks like Core Data for efficient local data storage; Core Location and MapKit for location-based apps; animations, networking, and the new App Extensions feature introduced in iOS 8.

A firm knowledge of your tools is essential for any professional. Throughout *iOS 8 Development with Swift* book you will be introduced to features of Xcode 6 and Swift that range from novice to power-user level. Your toolkit also wouldn't be complete without unit and performance testing tools and a great debugger either, which is why these are the focus of the final part of the book.

Swift and iOS 8 have brought some enormous changes and opportunities to the development community, and I want to bring more with *iOS 8 Development with Swift*. We cover many topics to make sure that you can build the apps you want to after, or even while reading it.

This is going to be cliché, but true. I wish a book like this had existed as I was learning Objective-C. Now I feel good knowing this book will exist for those joining the community through Swift.

Because your feedback is essential to creating the best book possible, I hope you'll be leaving comments in the Author Online forum. After all, I may already know how to do all this stuff, but I need to know if my explanations are working for you!

— Jack D. Watson-Hamblin

brief contents

1 Understanding Swift's impact

This chapter covers

- Why Swift is an important change
- Swift's safety features
- Benefits of Swift's functional features

Shortly after people started to look into the language syntax of Swift, there was a flurry of comments that certain things looked similar to their language of choice. It didn't seem to matter what someone's favorite language was Swift was welcoming and familiar for everyone interested in it.

While working on Swift, the team of developers drew inspiration from many modern languages like Rust, C#, Ruby, Haskell, and many more. The languages that are reflected in Swift are battle tested, and the people maintaining them have decades of combined learning about what makes a programming language easy and safe to use. The team behind Swift used that inspiration to shape it. The result was a language that's not only powerful and modern, but also familiar to developers with backgrounds in many other languages.

1.1 Why Swift?

Before Swift, iOS and Mac OS X applications were written using Apple's other programming language, Objective-C. As time has gone on Objective-C has had small changes and additions developed to try and keep up with the fast moving platforms it was being used for. The limitations of Objective-C meant that Apple had to rethink how to engineer a language that suits today's world, and from that Swift was born.

Though it will be sometime before Objective-C disappears, Apple's announcement of Swift was a statement that it should be the language of choice for iOS and Mac development from here on out.

1.1.1 A new level of interactivity

Apple's focus on developer tools has lead to an IDE that becomes increasingly more powerful with each release. A new tool called Playgrounds has been added in Xcode 6, which enables you to learn, experiment, test, and generally play with Swift code.

Swift Playgrounds are incredibly powerful for getting extremely fast feedback. Many developers working with compiled languages are realizing that Swift Playgrounds are the tool they always wished they'd had but didn't know until they saw it.

Many languages contain some kind of REPL (read-eval-print-loop), which you can program in and see the result immediately. Xcode 6 Swift Playgrounds provide a similar experience but combine this with the full power of an IDE to allow you to dive deeper into the results of your code.

When you open Xcode you'll see the Welcome to Xcode window shown in figure 1.1. It gives you the option to Get Started With a Playground.

Figure 1.1 The Welcome to Xcode window

> **NOTE** If this window doesn't come up when you launch Xcode, or if you're already doing something in Xcode, you can open the window by navigating to Window > Welcome to Xcode, or pressing Cmd-Shift-1.

No programming book would be complete without a quick Hello World example. Keeping up with tradition we are going to get a taste of both Swift and Swift Playgrounds with a basic Hello World example that will create a UILabel – the predominate way to display text in iOS – and make it display "Hello World" in big bold letters.

We create playgrounds by either selecting Get Started With a Playground in the Welcome to Xcode window or navigating to File > New > Playground from anywhere in Xcode 6. After creating a new playground, enter a name in the Name field of the dialog that comes up (see figure 1.2), click Next, and then choose a folder in which to save the playground file.

Figure 1.2 New playground dialog

TIP It's extremely handy to keep a playground file in your dock or open while you're working so that you can quickly paste in code and test ideas instead of creating a new one every time.

Once we've selected a place to save your playground file, we're presented with a new playground window (shown in figure 1.3). It has some Swift code already in it that creates a variable holding the string `"Hello, playground"` – the "Hello World" example for the Playgrounds tool.

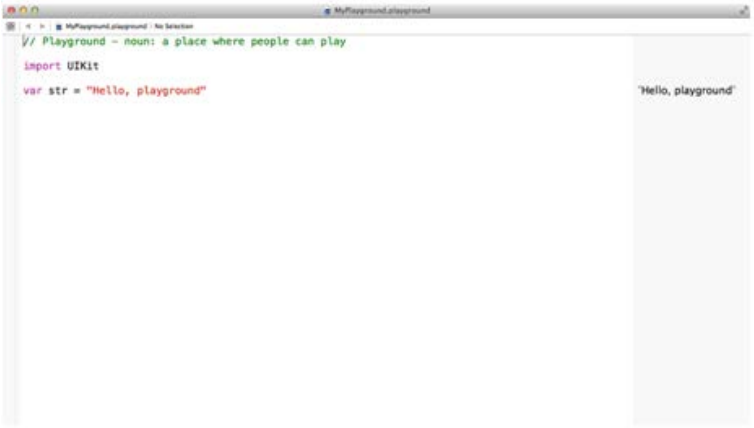

Figure 1.3 Newly created Swift playground

At the top of the code editor in your playground is a comment, and after is some executable Swift code. On line 3 in figure 1.3 is an `import` statement that imports `UIKit`, the framework used in iOS for building user interfaces. This gives you a way to test custom view and UI ideas in isolation. On line 5, you can see a variable called `str` being defined to contain the string `"Hello, playground"`.

Using the code editor in the playground, edit the code to match listing 1.1.

Listing 1.1 Creating a UILabel in a Swift Playground

```
// Playground - noun: a place where people can play

import UIKit

var str = "Hello, playground"
var label = UILabel()                                                    #A
label.text = str                                                         #B
label.font = UIFont.boldSystemFontOfSize(120)                            #C
label.sizeToFit()                                                        #D
label                                                                    #E
```

#A Initialize a new UILabel
#B Set the label's text
#C Font set to 120 and bold
#D Size the frame to fit the text
#E Used for the preview

You may notice that the gray sidebar on the right side of the playground updates as you write the code. It tells you the result of each line, as shown in figure 1.4. This feature of playgrounds is useful for us as developers because it allows us to see what our code is doing as it runs, from top to bottom. This feature removes the need to add logging statements in several places throughout the code to discover what a variable equals at a certain point in execution; instead it's all there in the sidebar.

Figure 1.4 Result of Swift code in the playground sidebar

The sidebar has some hidden buttons that let you get more information than you can see in the timeline. Hover over the last line in the sidebar, and two icons appear. The Quick Look icon (seen in figure 1.5) displays the result of a line of code in a tooltip-style overlay (see figure 1.6). This is an extremely helpful tool.

Figure 1.5 Quick Look icon in Swift Playgrounds

Figure 1.6 Using the Quick Look button to inspect the result of code

If you want to keep the result up so that you can see it update live as you change the code, you can instead use the Value History button. When clicked, Value History opens a new pane in the playground window. This pane displays the result of the value you added (see figure 1.7), which reloads live as you modify the code. Try it by changing the font size to a few different values.

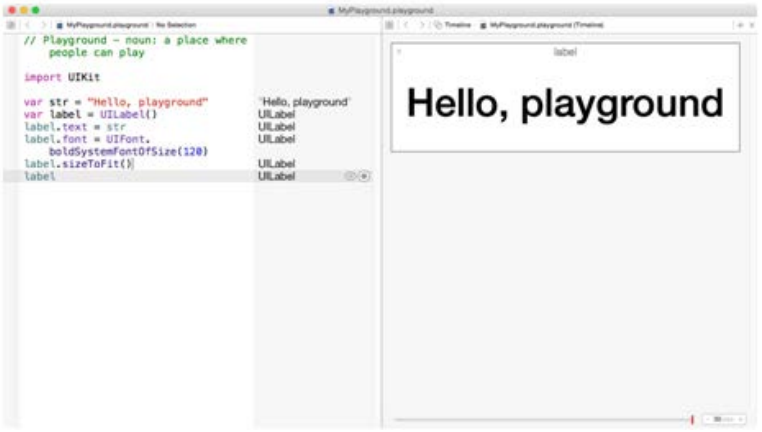

Figure 1.7 Displaying the code output in the Value History pane

Congratulations! You just wrote your first Swift code. You even used something that is a core part of almost every iOS application, a `UILabel`.

With that quick preview, you now have a basic understanding of a tool you can use to give you a tighter feedback loop when working on your applications. Throughout this book, many of the smaller examples will use playgrounds as the place to run and experiment further. As an iOS developer, you will find that playgrounds are core tools that you'll use to get your work done more quickly.

1.2 What makes Swift a safe language?

You may have some questions at this point:

- How is Swift a safe language?
- What are the features that make it modern?
- Does it really make my development process and apps faster?

Answering these questions is the focus of the rest of this chapter. I'll walk through some of the language and tooling features that have made it possible to achieve Swift's goals.

1.2.1 Some types of bugs can be forgotten

One of the biggest selling points of Swift has to be the safety it gives you for avoiding bugs while developing applications for iOS and Mac. Thanks to choices made while designing the language, and a smart and strict (without being overbearing) compiler, Swift removes entire categories of erroneous code by making them result in compile-time errors instead of runtime-errors meaning bugs are caught much earlier.

The number of crashes in iOS applications has been greatly reduced since iOS was opened up as a platform for developers. In the beginning, it had all kinds of issues. One of the most common was related to memory management: the `EXC_BAD_ACCESS` error code, which caused

applications to crash because they tried to access something that was no longer in memory. In the last few years, there have been enormous improvements in the tooling, the platform, and Objective-C, to decrease the number of issues developers were having. Now, with Swift, you don't have to worry about a number of common mistakes and hard-to-catch bugs.

One of the goals of Swift was to design a safe language, but what is it trying to save us from? Common issues that plague developers using many languages are bugs like these:

- Memory management and memory leaks
- Data structures containing unexpected data types in them
- Uninitialized variables
- Non-exhaustive `switch` statements
- `nil`/`null` working it's way into the data
- Control-flow statements with hanging expressions

All of these issues, except for memory management, which is mostly handled automatically with the occasional bit of assistance–are things stopped by the compiler. Errors caused by these issues, due to bad code or bad logic, are a thing of the past.

You may be getting worried: your past experience may have been with languages and compilers that try to fix such issues through heavy use of syntax. This isn't the case with Swift–it's far less syntax heavy, than its predecessor, Objective-C, and other similar languages like Java.

Thanks to features like type inference where the compiler can guess data types using information attained from method definitions your code has less chance of containing bugs without becoming bloated with crazy syntax. Type inference plays a deeper role in the language than you will first realize. Language constructs that we'll look at throughout this book such as optionals and typed collections, as well as the compiler's forced adherence to good programming practices, all lead to cleaner code with less bugs in it.

1.2.2 Comparing Swift and Objective-C

Let's take a look some comparisons of writing safe code in Objective-C and Swift. This is just a quick look; you'll learn about these safety features in depth throughout the book. A quick scan will show you how much cleaner safe code is in Swift.

EXAMPLE 1: BASIC TYPE SAFETY

Here are two examples of code that do exactly the same thing: the first (listing 1.2) is written in Objective-C, and the second (listing 1.3) is written in Swift. Both of these code examples declare a variable to start a greeting, and then create a dictionary (a collection of data organized into key/value pairs) that has a name as the key and an age as the value. After that, the code loops over the dictionary and prints to the console an introduction to each person.

Listing 1.2 Type safety in Objective-C

```
NSString *greeting = @"Hello this is %@, and they are %@ years old";
NSDictionary *people = @{
  "Jane Doe" : 23,
  "John Smith" : 37,
  "Kathy Lee": 31
};

for(NSString *name in people) {
  NSLog(greeting, name, people[name]);
}
```

Listing 1.3 Type safety in Swift

```
var greeting = "Hello this is %@, and they are %d years old"
var people = [
  "Jane Doe" : 23,
  "John Smith" : 37,
  "Kathy Lee" : 31
]

for (name, age) in people {
  NSLog(greeting, name, age)
}
```

Strongly typed compiled languages are generally safer than dynamic interpreted languages due to the number of bugs that can be caught at compile time just due to the type system. Where is the type system in Swift, though? You can see all the type syntax in the Objective-C example (listing 1.2). But in the Swift example, there isn't a single mention of a string, an integer, or a dictionary.

Swift has a feature called *type inference*. The Swift compiler can look at the definition of `greeting` and determine that because a `String` is being assigned to it that `greeting`'s type is `String`. You can still define the type of something–in some places, such as function definitions, you're required to do so. But by using type inference where you can, your Swift code becomes less syntax heavy than many strongly typed languages. When you do want to specify a type manually, you can do so by separating the variable name from the type with a colon.

Listing 1.4 Specifying a type in Swift

```
var greeting: String                                    #1
greeting = "Hello World!"

var myArray: [Int]                                      #2
myArray = [1, 2, 3, 4, 5]

myArray += ["Six"]                                      #3
```

#1 Specify a type by using a colon
#2 An array containing Int values
#3 This will cause a compile error

You can define exactly what type you want a variable to be, such as saying it should be a `String` #1 or that it's an `Array` containing values of the type `Int` #2. Type inference keeps you safe from bugs like the one in listing 1.4 #3. It safely assumes that if you have an `Array` with

only numbers in it, your code won't handle working with the text in a `String` if one sneaks into the `Array`. Type safety and inference go much further than this, as you'll see throughout this book.

EXAMPLE 2: AVOIDING NIL WITH OPTIONALS

Swift introduces a way to avoid the infestation of `nil` and `nil` checks in code. It does this through the use of a language feature called *optionals* that are covered in chapter 3. The examples in listings 1.5 and 1.6 try to retrieve data from a deeply nested chain of properties that may or may not contain a `nil` value somewhere along the way.

Listing 1.5 Checking for nil in Objective-C

```
User *user = User.findUserByEmail(@"info@example.com");

if (user != nil && user.address != nil && user.address.streetName != nil) {
  NSLog(
    @"The user with the email info@example.com lives on %@",
    user.address.streetName
  );
} else {
  NSLog(@"Could not find user's street name");
}
```

Listing 1.6 Checking for nil in Swift

```
var user = User.findUserByEmail("info@example.com")

if let streetName = user?.address?.streetName {
  NSLog("The user with the email info@example.com lives on %@", streetName)
} else {
  NSLog("Could not find user's street name")
}
```

This is an example of optional binding and optional chaining, which are incredibly powerful features. Unlike many other languages where `nil` is a value that can appear almost anywhere, in Swift an optional is a succinct way to model the idea that some functions and properties either will or won't have a value, while everything else is assumed to always have a value by default. The compiler enforces this, meaning it's not something you discover you missed at runtime, possibly on someone else's phone that you have no control over once you've launched your application in the iOS App Store.

1.3 Swift's modern features

Objective-C is a powerful and feature-rich language, but it has always been constrained by the limitations of C and a long history of codebases and libraries for which it has to maintain compatibility. Swift doesn't have either of these things holding it back; its removed the dependency on C and could be designed without existing code bases in mind due to it being a new language. The only limitations Swift has are miniscule in comparison.

You've already seen a brief glimpse of a modern feature of Swift that is also one of its strongest safety features: type inference. Many of the modern features impact each other, as you'll find out as you explore more advanced concepts in Swift.

1.3.1 Basic modern features

It's best that you start with the basics: things that are heavily taken for granted in Swift and that either are better executed in Swift or, until recently, weren't available in Objective-C.

First are the literals used in the language, such as `String`, `Integer`, `Array`, and `Dictionary`. Until recently, Objective-C didn't actually support these. Instead, Objective-C developers used classes like `NSArray` to create arrays. All of these classes come from the Foundation framework in Objective-C; they're not part of the language. Objective-C has added support for these literals now. It's kind of awkward, because they start with an `@` symbol–if you forget it then you're actually creating a C `String`, and that will lead to errors. This is part of the issue of Objective-C being a superset of C.

Swift has proper literals, and the good news is that they are fully backwards compatible with their Foundation counterparts. This is incredibly important because all of Apple's frameworks use these Foundation classes.

Coming from any other language, it's an odd idea that there could be no literals in a language. But for a long time this was the case; and even now, they're just literals for the Foundation classes. The following listing shows an example.

Listing 1.7 Swift data type literals

```
"Hello World!"                                          #A

123                                                     #B

[1.3, 2.7, 1.9]                                         #C

["name": "Jane Doe", "email": "info@example.com"]       #D

("Kathy Lee", "iOS Developers")                         #E
```

#A String
#B Integer
#C Array of Floats
#D Dictionary with String keys and values
#E Tuple with two Strings

Mutability is also part of the Swift language. The Foundation classes I mentioned actually have both mutable and immutable versions (for example, `NSString` and `NSMutableString`).

Swift has taken a modern route of making immutability available to developers by using keywords, while maintaining compatibility with Objective-C frameworks. Two keywords in the language are used to assign a value to a name: `let` and `var`. These will be introduced fully as you learn about Swift as a language during the rest of part 1 of this book. The quick explanation is that `let` means a value is immutable, and you can't change it or assign something new to this name. The opposite case is `var`, which means a value is mutable and so is the name pairing, meaning you can do what you like with it. Immutability is always preferred because of the safety it implies.

Listing 1.8 Mutable and immutable values

```
let name = "Swift"                                                    #A

var version = "1.0.0 (swift-600.0.51.4)"                              #B
```

#A An immutable value
#B A mutable value

1.3.2 More advanced features

Swift comes with much more modern features that weren't available to iOS developers until its release in 2014. These include features like type inference, generics, tuples, pattern matching, optionals, modules, and much more. Most of these features help you create cleaner, more flexible, and safer code, or help you encapsulate parts of your code better than Objective-C could.

1.4 Swift and functional programming

Many languages inspired how Swift was implemented. And though Swift is an object-oriented language, it has taken many ideas from functional programming languages like Haskell and Scala.

This is not a book about functional programming, but understanding the basic parts Swift has taken from the functional programming paradigm will make your Swift code even better. I'm going to use the 80/20 rule (20% of the work for 80% of the results) to teach you the basic concepts of functional programming. We'll look at the ones used in Swift so that you can understand the benefits of using them. You'll learn some of the theory in this section; over the course of this book, you will learn more of the practical side of Swift's use of functional programming.

1.4.1 Immutability

The core of functional programming, other than functions themselves, is immutability. Being able to differ between mutable and immutable values is commonplace in many object-oriented languages as well.

Immutability can seem like an over-the-top constraint on your code, especially given that the result of most applications is the mutation of state. For example, interacting with a database or changing the properties of your user interface requires mutation. Much of the core business logic in your application can be written using immutable values though.

One of the hardest places for developers to use immutability is a collection of data, such as an array. You'll often find yourself wanting to perform different mutating changes on an array, such as adding new values, transforming the values in the array into new values, or filtering out values that don't match certain criteria. Many of these actions also require mutable variables: for example, a counter in a loop. Many imperative programming techniques have taught us to rely heavily on mutable state.

The basic concept of immutability in functional programming is that instead of mutating something, you're creating a new, immutable value by using the old value. Instead of creating

lots of copies of your data, you manage it efficiently by using the old data as part of the new data.

Consider the benefit you get from using an immutable array. If you create a new immutable array based on the values in the old immutable array, you can be sure that the values in the old array won't change. They're immutable, so it's safe to use them as a reference for your new array. Swift takes care of this for you. In addition, you get the added benefit of the compiler being able to make great performance enhancements because it doesn't have to deal with mutable values. Immutability is core to the following topic: pure functions.

1.4.2 Pure functions

A pure function, given the same input, always gives the same output. It doesn't have any side effects like printing to the screen or changing a database. Pure functions especially don't mutate their input or use any global state to determine their output. They're pure in the sense that they do their job and don't interact with the world around them.

Take a second to think about this concept. Imagine if the majority of your application was made up of pure functions. How hard would it be to rationalize about what your application was doing? It wouldn't be hard at all, and that's the point.

You may be surprised that because pure functions are so ridiculously simple to comprehend, the majority of the time you can look at a function's name, parameter types, and the return type to work out exactly what it does.

This is where functional programming and mathematics begin their journey together. You can trust math. It's reliable at a whole new level. With that in mind I want to tell you a trait of functions in mathematics: they're pure. Think of functions like `sin`, `cos`, and `tan`. In math these functions are trusted completely and for a good reason: they will always give the same output when given the same input, and they never mess with the result of your calculations.

This may seem like an awfully big restriction, just as immutability did at first; but once again, experts have thought about these things carefully. They have shown us that by pushing to extract as many pure functions as possible from our code, anything that has to deal with mutable state becomes much easier to rationalize about. Having pure functions doesn't mean you must remove all mutable state from all of your functions; it just means that you should try to keep the mutable state separate from your pure functions.

1.4.3 First class and higher order functions

Functions and methods are important in many languages. The issue is that they're only seen as important because they encapsulate logic. This mindset is completely removed in the functional programming paradigm.

Swift can treat functions as first-class citizens, which means that functions can be passed around like data. This can be done with defined functions as well as anonymous functions; both play a role in creating *closures*. Having the ability to treat functions as data is incredibly important because it allows you to do something further with your functions: create functions that take functions or return functions.

A higher-order function is the next level of a pure function. These higher-order functions do one or both of two things.

1. Take one or more functions as input
2. Output a function

Having first-class functions and the ability to use higher-order functions allows you to create incredibly dynamic, easily understood code. Throughout the book, you'll be using first-class and higher-order functions extensively; you'll get plenty of practice that shows you the benefits they provide.

Depending on your past experience, you may have seen these in other languages like Ruby or JavaScript, with blocks and anonymous functions. Or as an example of a higher-order function, you may have used `map`, which takes a list of values and maps them to a new list of new values. The `map` function is core for doing things like transforming data.

1.4.4 Recursion

The trusty loop, whether it's a `while` or `for` loop or one of their relatives, is a core part of imperative programming and therefore most object-oriented programming as well. This isn't the case in functional programming–instead recursive functions are used.

A *recursive function* calls itself within its definition. The most famous of these is the function used to calculate Fibonacci sequences (e.g. 1, 2, 3, 5, 8, 13…).

Listing 1.8 Mathematics function for Fibonacci

```
fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2)
```

To calculate a Fibonacci number, you add the Fibonacci number of the input minus 1, to the Fibonacci number of the input minus 2. The function is recursive because it calls itself from within its definition.

But listing 1.8 also defines that when the `fib` function is passed 0, it returns 0; and when it's passed 1, it returns 1. These are the base cases–the seed values. Without these seed values, the recursion would never end. When crafting a recursive function its common to consider these base cases before you begin to write the recursive part. Doing this leads to much simpler functions.

Lets take the following listing as an example. This pseudo-code sums the numbers in an array using recursion instead of a loop (which is what you may be used to jumping straight to).

Listing 1.9 Recursive sum in pseudo-code

```
numbers = [1, 2, 3]

sum([]) = 0                                  #1
sum([x]) = x                                 #1
sum([head, tail]) = head + sum(tail)         #2

sum(numbers)                                 #3
```

#1 Base cases
#2 sum calls itself
#3 Returns 6

This function defines two bases cases #1 that return 0 if there is nothing in the array or, if there is one item in the array, return that item. For example calling `sum([])`returns 0 because there is nothing to sum, and calling `sum([3])` returns 3 because the sum of all the values in the array is 3 (there is only the one value in the array: 3).

What about this head and tail thing, though #2? What is that? The *head* is the first element in the array, and the *tail* is everything that isn't the head. By looking at an array like this, recursion becomes easy. Calling `sum([1, 2, 3])` expands to `1 + sum([2, 3])`, which expands out to `1 + 2 + sum([3])`. The base case says that when there is just one value, it should be returned. So the full expression is `1 + 2 + 3`, which gives you 6 just as expected.

Why not use a loop? It comes back to trusting mathematics and pure functions. Nowhere in either `fib` or `sum` is there any mutable state unlike in a loop, where a variable would keep track of the current sum and change on each iteration of the loop. Without state, and by being thoughtful about how you construct recursive functions, you end up with code that can be optimized better, rationalized about more easily, and tested more easily. And, best of all, you can create lots of smaller functions or use higher-order functions to reuse code more effectively and keep your code lean as your application requirements grow.

The last thing I haven't mentioned about functional programming techniques is concurrency. Without mutable state, everything you've read so far can work more effectively in a multithreaded environment, which iOS is. I didn't mention concurrency until now because it's a large focus of why people should use functional programming techniques. Focusing solely on that takes away from the beauty of applying your trust of mathematics to your programming. Trusting your code is more important than concurrency issues.

1.4.5 The type system

Strongly typed languages are a mainstay when it comes to trusting your code, and this is doubly true in Swift. It has a powerful and strict compiler, although, as demonstrated earlier, it's not as overbearing as some other language's compilers.

You may wonder why I'm mentioning the type system in a section about functional programming. Functions have input and output types, of course, but that's not a big deal–unless you stop to think about the fact that the types in the parameters and return value actually define a new type.

For example you may have a function that takes two objects of the same type as its parameters and returns a Boolean. You're likely to assume that the function compares the two objects. The type definition of the function has some meaning, and that's why Swift has type definitions for functions. This one would then be `(SomeClass, SomeClass) -> Bool`. You will learn more about this later.

This is important for functional programming because it means we can restrict the kinds of functions passed into higher-order functions. Using `map` as an example, any function that is

passed to map should take a collection of values and return a collection of values. It wouldn't make sense to pass the function I described before into map, so Swift won't let you do it. It treats the types for the input and output of your functions as a type definition. This way, you can define the type of function your higher-order function accepts, just as you define the type of input any function will accept.

1.5 Summary

As a language, Swift has made programming for iOS a much more approachable idea. Many developers who were scared off by Objective-C are flocking to Swift because it's much more familiar to them.

In the past few pages, you've learned about many of Swift's improvements over Objective-C, it's safety features, and some of the niceties Swift has for you as a developer. You've also learned about some of the basic concepts Swift takes from functional programming languages to allow you to write safer code that is easier to rationalize about.

In this chapter you've read a lot about Swift, which is expected–after all, this is a book about Swift. What I haven't discussed, but you'll learn about throughout this book, is the fact that Swift isn't the only tool Apple has given us to make things easier and to help us be more effective developers. Xcode 6 and many of it's features help us greatly too. You'll quickly start to see how Swift enables you to be a swift developer.

With your introduction to the impact that Swift has made, it's time to begin writing your first application using Xcode 6, Swift, and the Cocoa Touch frameworks in chapter 2.