Swift is Apple's new innovative and user-friendly development language, which is packed with modern features to make programming easier, fun, and flexible.

This book begins by giving you a solid introduction to the Swift programming language so you can quickly begin developing applications using this interesting language. It also covers advanced topics such as Objective-C interoperability, ARC, closures, and concurrency. Each concept covered is backed up with example code and demonstrates how to properly execute it. Next, you will be taught about all of the advanced features of Swift, and its interaction with Apple's APIs and libraries. You’ll then learn to interact with REST-based web services using Swift. We will conclude the book by getting equipped to design and build applications using established design patterns.

Who this book is written for

If you are a developer that learns best by looking at, and working with, code, then this book is for you. A basic understanding of Apple’s tools is beneficial but not mandatory.

What you will learn from this book

- Prototype and test code in a Playground
- Understand the basics of Swift, including operators, collections, control flows, and functions
- Create and use Classes, Structures, and Enums, including object-oriented topics such as inheritance, protocols, and Extensions
- Dwell into Subscripts, Optionals, and closures with real-world scenarios
- Employ Grand Central Dispatch to add concurrency to your applications
- Study the Objective-C interoperability with mix and match
- Access network resources using Swift

Implement various standard design patterns in the Swift language

Mastering Swift

Master Apple's new Swift programming language by following the best practices to write efficient and powerful code.
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Taking the First Steps with Swift'
- A synopsis of the book’s content
- More information on Mastering Swift
About the Author

Jon Hoffman has over 20 years of experience in the information technology field. Over those 20 plus years, Jon has worked in the areas of system administration, network administration, network security, development, and architecture. Currently, he works as a software engineer for Syntech Systems.

Over the past 7 years, Jon has developed extensively for the iOS platform. This includes several apps that he has published on the App Store, apps that he has written for third parties, and numerous enterprise applications. Packt Publishing also published Jon's first book, iOS and OS X Network Programming Cookbook.

What really drives Jon are the challenges in the information technology field, and there is nothing more exhilarating for him than overcoming a challenge. Some of Jon's other interests are watching baseball (go Sox!) and basketball (go Celtics!). Jon also really enjoys fishing, kayaking, and just being out on the lake with his family.
Preface

Swift is Apple's new programming language that was introduced at the WWDC (Worldwide Developers Conference) 2014, alongside the integrated development environment Xcode 6 and iOS 8. Swift was arguably the most significant announcement at WWDC 2014 and very few people, including Apple "insiders", were aware of the project's existence prior to it being announced.

Swift can be thought of as Objective-C reimagined using modern concepts and safe programming patterns. In Apple's own words, Swift is like "Objective-C without the C". Chris Lattner, the creator of Swift, said Swift took language ideas from Objective-C, Rust, Haskell, Ruby, Python, C#, CLU, and far too many others to list.

In Apple's own words "Swift is a successor to the C and Objective-C languages"; therefore, it is imperative for iOS and OS X developers who want to keep their skills up to date to not only learn but also master the Swift programming language.

The first five chapters of this book will introduce the reader to the Swift programming language. These chapters will give the reader a solid understanding of the Swift programming language. The remainder of the book will cover more advance topics such as concurrency, network development, and design patterns and will help the reader master the swift programming language.

This book is written in an example-based approach where each topic covered is backed by examples written to reinforce the topic and to show how to implement it within the reader's code.

Since Swift is constantly changing and evolving, the author has started a blog at http://masteringswift.blogspot.com to keep readers up to date with what is new in Swift. The blog will also be used to enhance and expand on the material in the book.
Preface

What this book covers

Chapter 1, *Taking the First Steps with Swift*, introduces the reader to the Swift programming language and discusses what inspired Apple to create Swift. We also go over the basic syntax of Swift and how to use Playgrounds to experiment and test Swift code.

Chapter 2, *Learning about Variables, Constants, Strings, and Operators*, explains to the reader about variables and constants in Swift and when to use them. There are brief overviews of the most common variable types, with examples of how to use them. We conclude this chapter by showing examples of how to use the most common operators in the Swift language.

Chapter 3, *Using Collections and Cocoa Data Types*, introduces Swift's Array and Dictionary collection types and shows examples of how to use them. We will also show how to use the Cocoa and Foundation data types with Swift.

Chapter 4, *Control Flow and Functions*, explains how to use Swift's control flow statements. These include looping, conditional, and control transfer statements. The second half of the chapter is all about functions and how to define and use them.

Chapter 5, *Classes and Structures*, explains Swift's classes and structures in detail. We will look at what makes them similar and what makes them different. We will also look at access controls and object-oriented design. We will close this chapter out by looking at memory management in Swift.

Chapter 6, *Working with XML and JSON Data*, starts off by discussing what XML and JSON data is and their uses. We then show several examples of how to parse and build XML and JSON data using Apple's frameworks.

Chapter 7, *Custom Subscripting*, examines how we can use custom subscripts in our classes, structures, and enumerations. Subscripts in Swift can be used to access elements in a collection. We can also define custom subscripts for our classes, structures, and enumerations.

Chapter 8, *Using Optional Type and Optional Chaining*, looks at what optional types really are, various ways to unwrap them, and optional chaining. For a developer just learning Swift, optional types can be one of the most confusing items to learn.

Chapter 9, *Working with Generics*, allows us to write very flexible and reusable code that avoids duplication. In this chapter, we will examine how Swift implements generics. We will also look at the proper ways to use generics in our code.
Chapter 10, Working with Closures, examines how to define and use closures in our code. We will conclude this chapter with a section on how to avoid strong reference cycles with closures. Closures in Swift are similar to blocks in Objective-C, except that they have a much cleaner and easier-to-use syntax.

Chapter 11, Using Mix and Match, examines mix and match and demonstrates how we can include Swift code in our Objective-C projects and Objective-C code in our Swift projects. With all of the apps and frameworks written in Objective-C, it was important to allow Swift and Objective-C code to work together.

Chapter 12, Concurrency and Parallelism in Swift, shows how to use both Grand Central Dispatch (GCD) and Operation Queues to add concurrency and parallelism to our applications. Understanding and knowing how to add concurrency and parallelism to our apps can significantly enhance the user experience.

Chapter 13, Swift Formatting and Style Guide, defines a style guide for the Swift language that can be a template for enterprise developers that need to create a style guide. Most enterprises have style guides for the various languages that they develop in.

Chapter 14, Network Development with Swift, looks at the Apple APIs to connect to remote servers and how to best use them. Network development can be both fun and challenging.

Chapter 15, Adopting Design Patterns in Swift, examines how to implement some of the more common design patterns in Swift. A design pattern identifies a common software development problem and provides a strategy for dealing with it.
Ever since I was 12 years old and wrote my first program in the basic programming language, programming has been a passion for me. Even as programming became my career, it always remained more of a passion than a job, but over the past few years, that passion has waned. I was unsure why I was losing that passion. I attempted to recapture it with some of my side projects but nothing really brought back the excitement that I used to have. Then, something wonderful happened, Apple announced Swift. Swift is such an exciting and progressive language that it has brought a lot of that passion back and made programming fun for me again.

In this chapter, you will learn:

- What is Swift?
- What are some of the features of Swift?
- What are Playgrounds?
- How do you use Playgrounds?
- What are the basic syntaxes of the Swift language?

What is Swift?

Swift is Apple's new programming language that was introduced at the Worldwide Developers Conference (WWDC) in 2014 alongside the integrated development environment Xcode 6 and iOS 8. Swift was arguably the most significant announcement at WWDC 2014 and very few people, including Apple insiders, were aware of the project's existence prior to it being announced. It was amazing, even by Apple's standards, that they were able to keep Swift a secret for as long as they did and that no one suspected they were going to announce a new development language.
Swift can be thought of as Objective-C reimagined using modern concepts and safe programming patterns. In Apple's own words, Swift is like Objective-C without the C. Chris Lattner, the creator of Swift, said Swift took language ideas from Objective-C, Rust, Haskell, Ruby, Python, C#, CLU, and far too many others to list.

At WWDC 2014, Apple really stressed that Swift was Safe by Default. Swift is designed to eliminate many common programming errors, making applications more secure and less prone to bugs. As we look at Swift throughout this book, we will point out many ways in which Swift is safer than not only Objective-C, but also safer than most other modern languages.

The development of Swift started in 2010 by Chris Lattner. He implemented much of the basic language structure with only a few people being aware of its existence. It wasn't until late 2011 that other developers began to really contribute to Swift, and in July of 2013, it became a major focus of the Apple Developer Tools group.

Chris Lattner started at Apple in the summer of 2005. He has held several positions in the Developers Tools group and is currently the Director and Architect of that group. On his home page (http://www.nondot.org/sabre/), Chris notes that Xcode's Playground (more on Playgrounds a little later in this chapter) became a personal passion of his because it makes programming more interactive and approachable. We will be using Playgrounds a lot in the book as a test and experimentation platform.

There are a lot of similarities between Swift and Objective-C. Swift adopts the readability of Objective-C's named parameters and dynamic object model. Swift also provides seamless access to existing Cocoa frameworks. This gives Objective-C developers a certain amount of familiarity when they begin to learn Swift.

While there are a lot of similarities between Swift and Objective-C, there are also significant differences between them as well. Swift's syntax and formatting are a lot closer to Python than Objective-C, but Apple did keep the curly braces. I know Python people would disagree with me, and that is alright because we all have different opinions, but I like the curly braces. Swift actually makes the curly braces required for control statements, such as if and while, and eliminating bugs such as the Goto Fail bug in Apple's SSL library.

**Mix and match** allows us to create applications that contain both Objective-C and Swift files that can communicate with each other. This allows us to systematically update current Objective-C applications with Swift classes. It also allows us to use current Objective-C libraries/frameworks in our Swift applications.
Mix and match lets Objective-C and Swift files coexist in the same project. This allows us to begin using Swift without throwing away our existing Objective-C code base or projects.

Swift was also built to be fast. At WWDC 2014, Apple showed a number of benchmarks that showed Swift significantly outperformed Objective-C. Swift uses the LLVM compiler, which is included with Xcode 6 to transform the Swift code into highly optimized native code that is tuned to get the most out of Apple's modern hardware.

If you are an iOS or OS X developer and you are still not convinced that learning Swift is a good idea, then maybe this one paragraph from Apple's Swift page (https://developer.apple.com/swift/) will help convince you:

Swift is a successor to the C and Objective-C languages. It includes low-level primitives such as types, flow control, and operators. It also provides object-oriented features such as classes, protocols, and generics, giving Cocoa and Cocoa Touch developers the performance and power they demand.

The first line in that paragraph that says, "Swift is a successor to the C and Objective-C languages" is the most important line. This line and other Apple documentation tells us that Apple sees the Swift language as its application and systems programming language of the future. While Objective-C is not going away anytime soon, it sounds like it will be taking a backseat to Swift in the very near future.

**Swift features**

When Apple said that Swift is Objective-C without the C, they are really only telling us half of the story. Objective-C is a superset of C and provides object-oriented capabilities and a dynamic runtime to the C language. This meant that Apple needed to maintain compatibility with C, which limited the enhancements it could make to the Objective-C language. As an example, Apple could not change how the switch statement functioned and still maintain the C compatibility.

Since Swift does not need to maintain the same C compatibility of Objective-C, Apple was free to add any feature/enhancement to the language. This allowed Apple to include the best features from many of today's most popular and modern languages, such as Objective-C, Python, Java, Ruby, C#, Haskell, and many others.
The following chart shows a list of some of the most exciting enhancements that Swift includes:

<table>
<thead>
<tr>
<th>Swift feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type inference</td>
<td>Swift can automatically deduce the type of the variable or constant based on the initial value.</td>
</tr>
<tr>
<td>Generics</td>
<td>Generics allow us to write code once to perform identical tasks for different types of objects.</td>
</tr>
<tr>
<td>Collection mutability</td>
<td>Swift does not have separate objects for mutable or non-mutable containers. Instead, you define mutability by defining the container as a constant or a variable.</td>
</tr>
<tr>
<td>Closure syntax</td>
<td>Closures are self contained blocks of functionality that can be passed around and used in our code.</td>
</tr>
<tr>
<td>Optionals</td>
<td>Optionals define a variable that might not have a value.</td>
</tr>
<tr>
<td>Switch statement</td>
<td>The Switch statement has been drastically improved. This is one of my favorite improvements.</td>
</tr>
<tr>
<td>Multiple return types</td>
<td>Functions can have multiple return types using tuples.</td>
</tr>
<tr>
<td>Operator overloading</td>
<td>Classes can provide their own implementation of existing operators.</td>
</tr>
</tbody>
</table>

Before we begin our journey into the wonderful world of Swift development, let's take a detour and visit a place that I have loved ever since I was a kid: The playground.

**Playgrounds**

When I was a kid, the best part of the school day was going to the playground. It really did not matter what we were playing as long as we were on the playground, I knew it would be fun. When Apple introduced Playgrounds as part of Xcode 6, I was excited just by the name, but I wondered if Apple could make its Playground as fun as the playgrounds from my youth. While Apple's Playgrounds might not be as fun as playing kickball when I was nine years old, it definitely brings a lot of fun back to experimenting and playing with code.

**Getting started with Playgrounds**

Playgrounds are an interactive work environment that lets us write code and see the results immediately in the sidebars. As changes are made to the code, the results in the sidebar also change in real time. This means that Playgrounds are a great way to learn and experiment with Swift.
Playgrounds also make it incredibly easy to try out new APIs, prototype new algorithms, and demonstrate how code works. We will be using Playgrounds throughout this book to show how our sample code works. Therefore, before we really get into Swift development, let's spend some time learning and getting comfortable with Playgrounds.

If the Swift code does not make a lot of sense right now, do not worry; as we go through the book, this code will begin to make sense. We are simply trying to get a feel of Playgrounds right now.

A Playground has three sections, as follows:

- **Coding Area**: This is where you enter your Swift code.
- **Results Sidebar**: This is where the results of your code are shown. Each time you type in a new line of code, the results are re-evaluated and the results sidebar is updated with the new results.
- **Timeline Sidebar**: This sidebar displays various objects depending on what your code does. Later in this chapter, we will show you how to display an image in the timeline sidebar.

The following screenshot shows how the sections are arranged in a Playground:
Let's start a new Playground. The first thing we need to do is to start Xcode. Once Xcode has started, we can select the **Get started with a playground** option, as shown in the following screenshot:

Alternatively, we can navigate to **File | New | Playground** from the top menu bar, as shown in the following screenshot:
Next, we should see a screen similar to the following screenshot that lets us name our Playground and select whether the Playground is an iOS or OS X Playground. For most of the examples in this book, it is safe to assume that you can select either iOS or OS X unless it is otherwise noted.

Finally, we are asked for the location to save our Playground too. After we select the location, the Playground will open up and look similar to the following screenshot:
From the preceding screenshot, we can see that the coding area of the Playground looks like the coding area for an Xcode project. What is different is the sidebar on the right-hand side. This sidebar is where the results of our code are shown. The code in the previous screenshot imports iOS's UIKit framework and sets a variable named `str` to the string `Hello, playground`. You can see the content of the `str` string in the sidebar to the right of the code.

By default, a new Playground does not open the timeline sidebar. You can open it manually by pressing the `Command`, `Option`, and `Enter` keys together. We will show you why the timeline sidebar is so important later in this chapter.

**iOS and OS X Playgrounds**

When you start a new iOS Playground, the Playground imports UIKit (Cocoa Touch). This gives us access to the UIKit framework that provides the core infrastructure for iOS applications. When we start a new OS X Playground, the Playground imports Cocoa. This gives us access to the OS X Cocoa framework.

What the last paragraph means is, if we want to experiment with specific features of either UIKit or Cocoa, we need to open the correct Playground. As an example, if we have an iOS Playground open and we want to create an object that represents a color, we would use a `UIColor` object. If we had an OS X playground open, we would use an `NSColor` object to represent a color.

**Showing images in a Playground**

As you will see throughout this book, Playgrounds are great at showing the results of code as text in the results sidebar. Playgrounds can also do a lot more than just text; they can also do images, graphs, and display views. Let's take a look at how we would show an image in a Playground. The first thing we need to do is to load the image into the resource directory of our Playground. The following steps show how to load an image into the resource directory:
1. Let's begin by showing the project navigator sidebar. To do this, in the top menu bar navigate to View | Navigators | Show Project Navigator, or you can use the Command-1 keyboard shortcut. The project navigator looks like this:

![Project Navigator Sidebar]

2. Next, we need to drag the image into the Resources folder so that we can access it from our code. Once we drag the image file over it, it will appear in the Resources folder.
3. Now we can access the image that is in our Resources folder with our code. The following screenshot shows how we will do this. The actual code that we use to access the image is not as important at this time as knowing how to access resources within a playground:

4. To view the image, we need to hover our cursor in the results sidebar over the section that shows the width and height of the image. In our example, the width and height section looks like this: **w 256 h 256**. Once we hover the mouse pointer of the width and height, we should see two symbols, as shown in the following screenshot:
5. We can press either of the symbols to show the image. The one that is shaped like a circle with a plus sign in it will display the image within the playground's code section, while the one that looks like an eye will pop the image up outside of the playground. The following screenshot shows what it looks like if we press the symbol that looks like a circle with a plus sign in it.
Displaying graphs in Playgrounds

We can also graph the value of numeric variables over time with the timeline sidebar. This feature is really useful when we are prototyping new algorithms. It allows us to see the value of the variable throughout the course of the calculations. To see how graphing works, take a look at the following Playground:

In this playground, we set the variable \( j \) to 1. We then create a for loop that assigns numbers 1 through 5 to the variable \( i \). At each step in the for loop, we set the value of the variable \( j \) to the current value of \( j \) multiplied by \( i \). The graph in the timeline sidebar shows the value of the variable \( j \) at each step of the for loop. We will be covering for loops in detail later in this book.

To bring up the graph, click on the symbol that is shaped like a circle with a dot in it. We can then move the timeline slider to see the values of variable \( j \) at each step of the for loop.
What Playgrounds are not

There is a lot more that we can do with Playgrounds, and we have only scratched the surface in our quick introduction here. As we proceed through the book, we will be using Playgrounds for almost all of the sample code and will demonstrate other features of Playgrounds as they are used.

Before we leave this brief introduction, let’s take a look at what Playgrounds are not, so we can understand when not to use Playgrounds:

- Playgrounds should not be used for performance testing: The performance you see from any code that is run in a Playground is not representative of how fast the code will run when it is in your projects.
- Playgrounds do not support user interaction: Users cannot interact with code that is run in a Playground.
- Playgrounds do not support on-device execution: You cannot run the code that is present in a Playground outside of the Playground. This means you cannot run it as an external application or on an external device.

Swift language syntax

If you are an Objective-C developer, and you are not familiar with modern languages like Python or Ruby, the code in the previous screenshots probably looked pretty strange. The Swift language syntax is a huge departure from Objective-C, which was based largely on Smalltalk and C.

The Swift language uses very modern concepts and syntax to create very concise and readable code. There was also a heavy emphasis on eliminating common programming mistakes. Before we get into the Swift language itself, let’s take a look at some of the basic syntax of the Swift language.

Comments

Writing comments in Swift code is a little different from writing comments in Objective-C code. We still use the double slash // for single line comments and the /* and */ for multiline comments.
What has changed is how we document the parameters and the return value. To document any parameter, we use the `:param:` field, and for the return value, we use the `:return:` field. The following Playground shows examples of both single line and multiline comments to properly comment a function:

![Playground screenshot](image)

To write good comments, I would recommend using single line comments within a function to give quick one-line explanations of your code. We will then use the multiline comments outside of functions and classes to explain what the function and class does. The preceding Playground shows a good use of comments. By using proper documentation, as we did in the preceding screenshot, we can use the documentation feature within Xcode. If we hold down the option key, and then click on the function name anywhere in our code, Xcode will display a popup with the description of the function. This next screenshot shows what that popup would look like:
Semicolons

You probably noticed from the code samples so far, that we are not using semicolons at the end of lines. The semicolons are optional in Swift; therefore, both lines in the following Playground are valid in Swift. You can see the results of the code in the results sidebar, as shown in the following screenshot:

For style purposes, it is strongly recommended that you do not use semicolons in your Swift code. If you are really set on using semicolons in your code (I would not recommend it), then I would recommend just being consistent and using it on every line of code; however, Swift will not warn you if you forget them.
Parentheses

In Swift, parentheses around conditional statements are optional, for example, both if statements in the following Playground are valid. You can see the results of the code in the sidebar, as shown in the following screenshot:

For style purposes, it is recommended that you do not include the parentheses in your code unless you have multiple conditional statements on the same line. For readability purposes, it is good practice to put parentheses around the individual conditional statements that are on the same line. See the following Playground for samples:
Curly braces

In Swift, unlike most other languages, the curly bracket is required after statements. This is one of the safety features that are built into Swift. Arguably, there have been numerous security bugs that may have been prevented if the developer would have used curly braces. A good example of this is Apple's Goto Fail bug. These bugs could also have been prevented by other means, as well as unit testing and code reviews, but requiring developers to use curly braces, in my opinion, is a good security standard. The following Playground shows you what error you get if you forget to include the curly braces:

Assignment operator (=) does not return a value

In most other languages, the following line of code is valid, but it probably is not what the developer meant to do:

```swift
if (x = 1) {
}
```
In Swift, this statement is not valid. Using an assignment operator (=) in a conditional statement (if and while) will throw an error. This is another safety feature built into Swift. It prevents the developer from forgetting the second equals sign (=) in a comparison statement. This error is shown in the following Playground:

Spaces are optional in conditional and assignment statements
For both conditional (if and while) and assignment (=) statements, the white spaces are optional. Therefore, in the following Playground, both the i block and j block of code are valid.
For style purposes, I would recommend adding the white spaces (such as the `j` block for readability purposes), but as long as you pick one style and be consistent, either style should be acceptable.

**Hello world**

All good computer books that are written to teach a computer language have a section that shows a user how to write a Hello World application. This book is no exception. In this section, we will show you how to write two different Hello World applications.

Our fist Hello World application will be the traditional Hello World application that simply prints Hello World to the console. Let's begin by creating a new Playground and naming it `Chapter_1_Hello_World`. The Playground can be either an iOS or an OS X Playground.

In Swift, there are two functions that will print messages to the console. These functions are `print()` and `println()`. The `print()` function will print the textural representation of the object passed to it, while the `println()` function will also print the textural representation of the object passed to it, and will also add a new line character to the end.

The `println()` function is overloaded to receive either an object to print or no value, in which case, we will print only a newline character to the console.
Let's add the `println()` function, as shown in the following Playground, to our Playground to print the Hello World message to the console. We will then need to open the Timeline sidebar (keyboard shortcut: `Command-option` and `Enter`) to see what is being printed to the console. Our Playground should look like the following Playground once we are done:

As we can see in the Timeline sidebar, our application will print the message **Hello World!** to the console.

Now, let's see if we can spice this up a bit and include a variable. We will be covering variables a lot more in the next chapter, so, for right now, it is alright if you do not fully understand how variables in Swift work; we mainly want to show you how to include variables in a `println()` function.

There are two ways in which we can add a string to another string. We can concatenate them or we can include them inline. To concatenate two strings, we use the `+` operator to add them together. The following code is an example of concatenating two strings:

```swift
var stringC = stringA + stringB
```

To include a string inline, we use a special sequence of characters `\( )`. The following code shows how to include a string inline with another string:

```swift
var name = "Jon"
var stringC = "Hello \(name)"
```
The following Playground shows examples of how to concatenate strings and how to include strings inline:

For the `message1` variable, we concatenate the strings together, while for the `message2` variable, we include the other strings inline. We then use the `println()` function to display both messages to the console.

**Summary**

In this chapter, we showed you how to start and use Playgrounds to experiment with Swift programming. We also covered the basic Swift language syntax and discussed proper language styles. The chapter concluded with two Hello World examples. In the next chapter, we will see how to use variables and constants in Swift. We will also look at the various data types and how to use operators in Swift.
Where to buy this book

You can buy Mastering Swift 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.