Swift Essentials

Swift is a new and powerful programming language that represents an essential new programming tool for iOS and OSX applications and builds upon the power of Objective-C while streamlining the developer experience.

Swift Essentials is a fast-paced, practical guide showing you the quickest way to put Swift to work in the real world. It guides you concisely through the basics of syntax and development before pushing ahead to explore Swift’s higher features through practical programming projects.

By the end of the book, you will be able to use Xcode’s graphical interface builder, create interactive applications, and communicate with network services.

Who this book is written for

Whether you are a seasoned Objective-C developer or new to the Xcode platform, Swift Essentials will provide you with all you need to know to get started with the language. Prior experience with iOS development is not necessary, but will be helpful to get the most out of the book.

What you will learn from this book

- Explore the nuts and bolts of the Swift syntax
- Test Swift code interactively with the REPL
- Display graphics with QuickLook in the Swift playground
- Present data in master-detail applications
- Use the Swift storyboard to manage multi-screen applications
- Create graphical UIViews with Swift
- Parse JSON and XML data from network sources
- Build a standalone iOS application from start to finish

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 2 “Playing with Swift”
- A synopsis of the book’s content
- More information on Swift Essentials

About the Author

Dr Alex Blewitt has over 20 years of experience in Objective-C and has been using Apple frameworks since NeXTSTEP 3.0. He upgraded his NeXTstation for a TiBook when Apple released Mac OS X in 2001 and has been developing on it ever since.

Alex currently works for a financial company in London and writes for the online technology news site InfoQ. He has authored two other books for Packt Publishing. He also has a number of apps on the App Store through Bandlem Limited. When he's not working on technology and the weather is nice, he likes to go flying from the nearby Cranfield airport.

Alex writes regularly on his blog http://alblue.bandlem.com as well tweets regularly on Twitter, @alblue.
Acknowledgments

This book would not have been possible without the ongoing love and support of my wife, Amy, who has helped me through the highs and lows of life. She gave me the freedom to work during the many late nights and weekends that it takes to produce a book and its associated code repository. She truly is the gem of my life.

I'd also like to thank my parents, Ann and Derek, for their encouragement and support during my formative years. It was their work ethics that allowed me to start my career in technology as a teenager and to incorporate my first company before I was 25. I'd also like to congratulate them on their 50th wedding anniversary in 2015, and I look forward to reaching this milestone with Amy.

Thanks is due especially to the reviewers of the book, Nate Cook, James Robert, Arvid Gerstmann, and Anil Varghese, who provided excellent feedback on the contents of this book during development and caught many errors in both the text and code. Any remaining errors are my own.

I'd also like to thank CodeClub, with whom I have been volunteering to teach young children how to code, and Akeley Wood, for allowing me to be a part of it. I hope both Sam and Holly enjoy it as much as I do.

Finally, I'd like to thank Ben Moseley and Eren Kotan who introduced me to NeXT in the first place and set my career going on a twenty-year journey to this book.
Swift Essentials

*Swift Essentials* provides an overview of the Swift language and the tooling necessary to write iOS applications. From simple Swift commands on the command line to interactively testing graphical content in the Playground editor, the Swift language and syntax is introduced by examples.

The book also introduces end-to-end iOS application development by showing you how a simple iOS application can be created, followed by how to use storyboards and custom views to build a more complex networked application.

The book concludes by providing a worked example from scratch that builds up a GitHub repository browser.

**What This Book Covers**

*Chapter 1, Exploring Swift*, presents the Swift read-evaluate-print-loop (REPL) and introduces the Swift language through examples on standard data types, functions, and looping.

*Chapter 2, Playing with Swift*, demonstrates Swift Playgrounds as a means to interactively play with the Swift code and obtain graphical results. It also introduces the playground format and shows how playgrounds can be created automatically from Markdown and AsciiDoc files.

*Chapter 3, Creating an iOS Swift App*, shows you how to create and test an iOS application built in Swift using Xcode, along with an overview of the Swift classes, protocols, and enums.

*Chapter 4, Storyboard Applications with Swift and iOS*, introduces the concept of Storyboards as a means to create a multiscreen iOS application and shows how views in Interface Builder can be wired to Swift outlets and actions.

*Chapter 5, Creating Custom Views in Swift*, covers custom views in Swift using custom table views, laying out nested views, drawing custom graphics, and layered animations.

*Chapter 6, Parsing Networked Data*, demonstrates how Swift can talk to networked services, using both HTTP and custom stream-based protocols.

*Chapter 7, Building a Repository Browser*, uses the techniques described in this book to build a repository browser that can display information about users' GitHub repositories.

*Appendix* provides additional references and resources to continue learning about Swift.
Playing with Swift

Xcode ships with both a command line interpreter (covered in Chapter 1, Exploring Swift) and a graphical interface called **playground** that can be used to prototype and test Swift code snippets. Code typed into the playground is compiled and executed interactively, which permits a fluid style of development. In addition, the user interface can present a graphical view of variables as well as a timeline, which can show how loops are executed. Finally, playgrounds can mix and match code and documentation, leading to the possibility of providing example code as playgrounds and using playgrounds to learn how to use existing APIs and frameworks.

This chapter will present the following topics:

- How to create a playground
- Displaying values in the timeline
- Presenting objects with Quick Look
- Running asynchronous code
- Using playground live documentation
- Generating playgrounds with Markdown and Asciidoc
- Limitations of playgrounds
Playing with Swift

Getting started with playgrounds
When Xcode is started, a welcome screen is shown with various options, including the ability to create a playground. Playgrounds can also be created from the File | New | Playground menu.

Creating a playground
Using either the Xcode welcome screen (which can be opened by navigating to Window | Welcome to Xcode) or navigating to File | New | Playground, create MyPlayground in a suitable location targeting iOS. Creating the playground on the Desktop will allow easy access to test Swift code, but it can be located anywhere on the filesystem.

Playgrounds can be targeted either towards OS X applications or towards iOS applications. This can be configured when the playground is created, or by switching to the Utilities view by navigating to View | Utilities | Show File Inspector or pressing Command + Option + 1 and changing the dropdown from OS X to iOS or vice versa.
When initially created, the playground will have a code snippet that looks as follows:

```swift
// Playground - noun: a place where people can play
import UIKit
var str = "Hello, playground"
```

Playgrounds targeting OS X will read `import Cocoa instead.`

On the right-hand side, a column will show the value of the code when each line is executed. In the previous example, the word **Hello, playgr...** is seen, which is the result of the string assignment. By grabbing the vertical divider between the Swift code and the output, the output can be resized to show the full text message:

Alternatively, by moving the mouse over the right-hand side of the playground, the **Quick Look** icon (the eye symbol) will appear; if clicked on, a pop-up box will show the full details:
Playing with Swift

Viewing the console output

The console output can be viewed on the right-hand side by opening the Assistant Editor. This can be opened by pressing Command + Option + Enter or by navigating to View | Assistant Editor | Show Assistant Editor. This will show the result of any println statements executed in the code.

Add a simple for loop to the playground and show the Assistant Editor:

```swift
for i in 1...12 {
    println("I is \(i)")
}
```

The output is shown on the right-hand side:

The assistant editor can be configured to be displayed in different locations, such as at the bottom, or stacked horizontally or vertically by navigating to the View | Assistant Editor menu.
Viewing the timeline

The timeline shows what other values are displayed as a result of executing the code. In the case of the print loop shown previously, the output was displayed as **Console Output** in the timeline. However, it is possible to use the playground to inspect the value of an expression on a line, without having to display it directly. In addition, results can be graphed to show how the values change over time.

Add another line above the `println` statement to calculate the result of executing an expression, `(i-6)*(i-7)`, and store it in a variable, `j`:

```swift
for i in 1...12 {
    var j = (i-7) * (i-6)
    println("I is \(i)")
}
```

On the line next to the variable definition, click on the add variable history symbol (+), which is in the right-hand column (visible when the mouse moves over that area). After it is clicked on, it will change to a (o) symbol and display the graph on the right-hand side. The same can be done for the `println` statement as well:
Playing with Swift

The slider at the bottom, indicated by the red tick mark, can be used to slide the vertical bar to see the exact value at certain points:

```swift
for i in 1...12 {
    var j = (i-7) * (i-6)
    var k = i
    println("I is \(i)")
}
```

When the slider is dragged, both values will be shown at the same time.

To show several values at once, use additional variables to hold the values and display them in the timeline as well:

```swift
for i in 1...12 {
    var j = (i-7) * (i-6)
    var k = i
    println("I is \(i)")
}
```

When the slider is dragged, both values will be shown at the same time.
Displaying objects with QuickLook

The playground timeline can display objects as well as numbers and simple strings. It is possible to load and view images in a playground using classes such as UIImage (or NSImage on OS X). These are known as QuickLook supported objects, and by default include:

- Strings (attributed and unattributed)
- Views
- Class and struct types (members are shown)
- Colors

It is possible to build support for custom types in Swift, by implementing a debugQuickLookObject method that returns a graphical view of the data.

Showing colored labels

To show a colored label, a color needs to be obtained first. When building against iOS, this will be UIColor; but when building against OS X, it will be NSColor. The methods and types are largely equivalent between the two, but this chapter will focus on the iOS types.

A color can be acquired with an initializer or by using one of the predefined colors that are exposed in Swift using methods:

```swift
import UIKit // AppKit for OS X
let blue = UIColor.blueColor() // NSColor.blueColor() for OS X
```

The color can be used in a UILabel, which displays a text string in a particular size and color. The UILabel needs a size, which is represented by a CGRect, and can be defined with an x and y position along with a width and height. The x and y positions are not relevant for playgrounds and so can be left as zero:

```swift
let size = CGRect(x:0,y:0,width:200,height:100)
let label = UILabel(frame:size) // NSLabel for OS X
```

Finally, the text needs to be displayed in blue and with a larger font size:

```swift
label.text = str // from the first line of the code
label.textColor = blue
label.font = UIFont.systemFontOfSize(24) // NSFont for OS X
```
When the playground is run, the color and font are shown in the timeline and available for quick view. Even though the same UILabel instance is being shown, the timeline and the QuickLook values show a snapshot of the state of the object at each point, making it easy to see what has happened between changes.

```swift
var str = "Hello, playground"

for i in 1...12 {
    var j = (i-7) * (i-6)
    var k = i
    println("I is \(i)\)
}

let blue = UIColor.
    blueColor()
let size = CGRect(x:0,y:0,width:200,height:100)
let label = UILabel(frame:size)
label.text = str
label.textAlignment = NSTextAlignment.center
label.textColor = blue
label.font = UIFont.
    systemFont(ofSize:24)
```

**Showing images**

Images can be created and loaded into a playground using the UIImage constructor (or NSImage on OS X). Both take a named argument, which is used to find an image with the given name from the playground's Resources folder.

To download a logo, open Terminal.app and run the following commands:

```
$ mkdir MyPlayground.playground/Resources
$ curl http://alblue.bandlem.com/images/AlexHeadshotLeft.png >
    MyPlayground.playground/Resources/logo.png
```
An image can now be loaded in Swift with:

```swift
let logo = UIImage(named:"logo")
```

The location of the Resources associated with a playground can be seen in the File Inspector utilities view, which can be opened by pressing Command + Option + 1.

The loaded image can be displayed using QuickLook or by adding it to the value history:

It is possible to use a URL to acquire an image by creating an NSURL with NSURL(string: "http://..."), then loading the contents of the URL with NSData(contentsOfURL:), and finally using UIImage(data:) to convert it to an image. However, as Swift will keep re-executing the code over and over again, the URL will be hit multiple times in a single debugging session without caching. It is recommended that NSData(contentsOfURL:) and similar networking classes be avoided in playgrounds.
Advanced techniques

The playground has its own framework, XCPlayground, which can be used to perform certain tasks. For example, individual values can be captured during loops for later analysis. It also permits asynchronous code to continue to execute once the playground has finished running.

Capturing values explicitly

It is possible to explicitly add values to the timeline by importing the XCPlayground framework and calling XCPCaptureValue with a value that should be displayed in the timeline. This takes an identifier, which is used both as the title and for group-related data values in the same series. When the value history button is selected, it essentially inserts a call to XCPCaptureValue with the value of the expression as the identifier.

For example, to add the logo to the timeline automatically:

```
import XCPlayground
XCPCaptureValue("logo", logo)
```

```
let label = UILabel(frame: size)
label.textColor = blue
label.font = UIFont.systemFont(ofSize: 24)

let logo = UIImage(named: "logo")

import XCPlayground
XCPCaptureValue("logo", logo)
```
It is possible to use an identifier to group the data that is being shown in a loop with the identifier representing categories of the values. For example, to display a list of all even and odd numbers between 1 and 6, the following code could be used:

```swift
for n in 1...6 {
    if n % 2 == 0 {
        XCPCaptureValue("even", n)
        XCPCaptureValue("odd", 0)
    } else {
        XCPCaptureValue("odd", n)
        XCPCaptureValue("even", 0)
    }
}
```

The result, when executed, will look as follows:

![Graph showing even and odd numbers between 1 and 6]

[45]
Running asynchronous code

By default, when the execution hits the bottom of the playground, the execution stops. In most cases, this is desirable, but when asynchronous code is involved, execution might need to run even if the main code has finished executing. This might be the case if networking data is involved or if there are multiple tasks whose results need to be synchronized.

For example, wrapping the previous even/odd split in an asynchronous call will result in no data being displayed:

```swift
dispatch_async(dispatch_get_main_queue()) {
    for n in 1...6 {
        // as before
    }
}
```

This uses one of Swift's language features: the `dispatch_async` method is actually a two-argument method that takes a queue and a block type. However, if the last argument is a block type, then it can be represented as a trailing closure rather than an argument.

To allow the playground to continue executing after reaching the bottom, add the following call:

```swift
XCPSetExecutionShouldContinueIndefinitely()
```

Although this suggests that the execution will run forever, it is limited to 30 seconds of runtime, or whatever is the value displayed at the bottom-right corner of the screen. This timeout can be changed by typing in a new value or using the + and − buttons to increase/decrease time by one second.
Playgrounds and documentation

Playgrounds can contain a mix of code and documentation. This allows a set of code samples and explanations to be mixed in with the playground itself. Although there is no way of using Xcode to add sections in the UI at present, the playground itself is an XML file that can be edited using an external text editor such as TextEdit.app.

Learning with playgrounds

As playgrounds can contain a mixture of code and documentation, it makes them an ideal format for viewing annotated code snippets. In fact, Apple's Swift Tour book can be opened as a playground file.
Xcode documentation can be searched by navigating to the **Help | Documentation and API Reference** menu, or by pressing **Command + Shift + 0**. In the search field that is presented, type **Swift Tour** and then select the first result. The Swift Tour book should be presented in Xcode's help system:

A link to download and open the documentation as a playground is given in the first section; if this is downloaded, it can be opened in Xcode as a standalone playground. This provides the same information, but allows the code examples to be dynamic and show the results in the window:
A key advantage of learning through playground-based documentation is that the code can be experimented with. In the **Simple Values** section of the documentation, where `myVariable` is assigned, the right-hand side of the playground shows the values. If the literal numbers are changed, the new values will be recalculated and shown on the right-hand side.

Some examples are presented solely in playground form; for example, the Balloons demo, which was used in the introduction of Swift in the WWDC 2014 keynote, is downloadable as a playground from https://developer.apple.com/swift/resources/.

![Note that the Balloons playground requires OS X 10.10 and Xcode 6.1 to run.]

### Understanding the playground format

A playground is an OS X **bundle**, which means that it is a directory that looks like a single file. If a playground is selected either in `TextEdit.app` or in `Finder`, then it looks like a regular file:

![Under the covers, it is actually a directory:]

```
$ ls -F
MyPlayground.playground/
```

Inside the directory, there are a number of files:

```
$ ls -1 MyPlayground.playground/*
MyPlayground.playground/Resources
MyPlayground.playground/contents.xcplayground
MyPlayground.playground/section-1.swift
MyPlayground.playground/timeline.xctimeline
```
The files are as follows:

- The Resources directory, which was created earlier to hold the logo image
- The contents.xcplayground file, which is an XML table of contents of the files that make up the playground
- The section-1.swift file, which is the Swift file created by default when a new playground is created, and contains the code that is typed in for any new playground content
- The timeline.xctimeline file, which is an automatically generated file containing timestamps of execution, which the runtime generates when executing a Swift file and the timeline is open

The table of contents file defines which runtime environment is being targeted (for example, iOS or OS X), a list of sections, and a reference to the timeline file:

```xml
<playground version='3.0' sdk='iphonesimulator'>
  <sections>
    <code source-file-name='section-1.swift'/>
  </sections>
  <timeline fileName='timeline.xctimeline'/>
</playground>
```

This file can be edited to add new sections, provided that it is not open in Xcode at the same time.

An Xcode playground directory is deleted and recreated whenever changes are made in Xcode. Any Terminal.app windows open in that directory will no longer show any files. As a result, using external tools and editing the files in place might result in changes being lost. In addition, if you are using ancient versions of control systems, such as SVN and CVS, you might find your version control metadata being wiped out between saves. Xcode ships with the industry standard Git version control system, which should be preferred instead.
Adding a new documentation section

To add a new documentation section, ensure that the playground is not open in Xcode and then edit the contents.xcplayground file. The file itself can be opened by right-clicking on the playground in Finder and choosing Show Package Contents:

This will open up a new Finder window, with the contents displayed as a top-level set of elements. The individual files can then be opened for editing by right-clicking on the contents.xcplayground file, choosing Open With | Other..., and selecting an application, such as TextEdit.app.

Alternatively, the file can be edited from the command line using an editor such as pico, vi, or emacs.

Although there are few technology debates more contentious than whether vi or emacs is better, the recommended advice is to learn how to be productive in at least one of them. Like learning to touch-type, being productive in a command-line editor is something that will pay dividends in the future if the initial learning challenge can be overcome. For those who don't have time, pico (also known as nano) can be a useful tool in command-line situations, and the on-screen help makes it easier to learn to use. Note that the carat symbol (^) means control, so ^X means Control + X.
Playing with Swift

To add a new documentation section, create a directory called Documentation, and inside it, create a file called hello.html. The HTML file is an HTML5 document, with a declaration and a body. A minimal file looks like:

```html
<!DOCTYPE html>
<html>
<body>
  <h1>Welcome to Swift Playground</h1>
</body>
</html>
```

The content needs to be added to the table of contents (contents.xcplayground) in order to display it in the playground itself, by adding a documentation element under the sections element:

```xml
<playground version='3.0' sdk='iphonesimulator'>
  <sections>
    <code source-file-name='section-1.swift'/>
    <documentation relative-path='hello.html'/>
  </sections>
</playground>
```

The relative-path attribute is relative to the Documentation directory.

```
All content in the Documentation directory is copied between saves in the timeline and can be used to store other text content such as CSS files. Binary content, including images, should be stored in the Resources directory.
```

When viewed as a playground, the content will be shown in the same window as the documentation:
If the content is truncated in the window, then a horizontal rule can be added at the bottom with `<hr/>`, or the documentation can be styled, as shown in the next section.

### Styling the documentation

As the documentation is written in HTML, it is possible to style it using CSS. For example, the background of the documentation is transparent, which results in the text overlapping both the margins as well as the output.

To add a style sheet to the documentation, create a file called `stylesheet.css` in the `Documentation` directory and add the following content:

```css
body {
    background-color: white
}
```

To add the style sheet to the HTML file, add a style sheet link reference to the `head` element in `hello.html`:

```html
<head>
    <link rel="stylesheet" type="text/css" href="stylesheet.css"/>
</head>
```

Now when the playground is opened, the text will have a solid white background and will not be obscured by the margins:

![Welcome to Swift Playground](image)
Adding resources to a playground

Images and other resources can also be added to a playground. Resources need to be added to a directory called Resources, which is copied as is between different versions of the playground.

To add an image to the document, create a Resources folder and then insert an image. For example, earlier in this chapter, an image was downloaded by using the following commands:

$ mkdir MyPlayground.playground/Resources
$ curl http://alblue.bandlem.com/images/AlexHeadshotLeft.png >
MyPlayground.playground/Resources/logo.png

The image can then be referred to in the documentation using an img tag and a relative path from the Documentation directory:

```html
<img src="../Resources/logo.png" alt="Logo"/>
```

Other supported resources (such as JPEG and GIF) can be added to the Resources folder as well. It is also possible to add other content (such as a ZIP file of examples) to the Resources folder and provide hyperlinks from the documentation to the resource files.

```html
<a href="../Resources/AlexBlewitt.vcf">Download contact card</a>
```

Additional entries in the header

The previous example showed the minimum amount of content required for playground documentation. However, there are other meta elements that can be added to the document that have specific purposes and which might be found in other playground examples on the internet. Here is a more comprehensive example of using meta elements:

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8"/>
  <link rel="stylesheet" type="text/css" href="stylesheet.css"/>
  <title>Welcome to Swift Playground</title>
  <meta name="xcode-display" content="render"/>
  <meta name="apple-mobile-web-app-capable" content="yes"/>
  <meta name="viewport" content="width=device-width,maximum-scale=1.0"/>
</head>
```
In this example, the document is declared as being written in English (`lang="en"` on the `html` element) and in the UTF-8 character set.

The `<meta charset="utf-8"/>` should always be the first element in the HTML head section, and the UTF-8 encoding should always be preferred for writing documents. If this is missed, it will default to a different encoding, such as ISO-8859-1, which can lead to strange characters appearing. Always use UTF-8 for writing HTML documents.

The `link` and `title` are standard HTML elements that associate the style sheet (from before) and the title of the document. The title is not displayed in Xcode, but it can be shown if the HTML document is opened in a browser instead. As the documentation is reusable between playgrounds and the web, it makes sense to give it a sensible title.

The `link` should be the second element after the `charset` definition. In fact, all externally linked resources—such as style sheets and scripts—should occur near the top of the document. This allows the HTML parser to initiate the download of external resources as soon as possible. This also includes the HTML5 `prefetch` link type, which is not supported in Safari or playground at the time of writing.

The `meta` tags are instructions to Safari to render it in different ways (Safari is the web engine that is used to present the documentation content in playground). Safari-specific `meta` tags are described at https://developer.apple.com/library/safari/documentation/Applications/Reference/SafariHTMLRef/Articles/MetaTags.html and include the following:

- The `xcode-display=render` meta tag, which indicates that Xcode should show the content of the document instead of the HTML source code when opening in Xcode
- The `apple-mobile-web-app-capable=yes` meta tag, which indicates that Safari should show this fullscreen if necessary when running on a mobile device
- The `viewport=width=device-width,maximum-scale=1.0` meta tag, which allows the document body to be resized to fit the user’s viewable area without scaling
Generating playgrounds automatically
The format of the playground files are well known, and several utilities have been created to generate playgrounds from documentation formats, such as Markdown or AsciiDoc. These are text-based documentation formats that provide a standard means to generate output documents, particularly HTML-based ones.

Markdown
Markdown (a word play on markup) was created to provide a standard syntax to generate web page documentation with links and references in a plain text format. More information about Markdown can be found at the home page (http://daringfireball.net/projects/markdown/), and more about the standardization of Markdown into CommonMark (used by StackOverflow, GitHub, Reddit, and others) can be found at http://commonmark.org.

Embedding code in documentation is fairly common in Markdown. The file is treated as a top-level document, with sections to separate out the documentation and the code blocks. In CommonMark, these are separated with back ticks (````), often with the name of the language to add different script rendering types:

```swift
println("Welcome to Swift")
```

Other text and other blocks can follow below.

The most popular tool for converting Markdown/CommonMark documents into playgrounds (at the time of writing) is Jason Sandmeyer's swift-playground-builder at https://github.com/jas/swift-playground-builder/. The tool uses Node to execute JavaScript and can be installed using the npm install -g swift-playground-builder command. Both Node and npm can be installed from http://nodejs.org.

Once installed, documents can be translated using playground --platform ios --destination outdir --stylesheet stylesheet.css. If code samples should not be editable, then the --no-refresh argument should be added.
AsciiDoc

AsciiDoc is similar in intent to Markdown, except that it can render to more backends than just HTML5. AsciiDoc is growing in popularity for documenting code, primarily because the standard is much more well defined than Markdown is. The de facto standard translation tool for AsciiDoc is written in Ruby and can be installed using the `sudo gem install asciidoctor` command.

Code blocks in AsciiDoc are represented by a [source] block. For Swift, this will be [source, swift]. The block starts and ends with two hyphens (--):

```swift
println("Welcome to Swift")
```

Other text and other code blocks can follow below --.

AsciiDoc files typically use the ad extension, and the ad2play tool can be installed from James Carlson's repository at https://github.com/jxxcarlson/ad2play. Saving the preceding example as example.ad and running ad2play example.ad will result in the generation of the example.playground file.


Limitations of playgrounds

Although playgrounds can be very powerful for interacting with code, there are some limitations that are worth being aware of. There is no debugging support in the playground. It is not possible to add a breakpoint and use the debugger and find out what the values are. Given that the UI allows tracking values—and that it's very easy to add new lines with just the value to be tracked—this is not much of a hardship.

Other limitations of playgrounds include:

- Only the simulator can be used for the execution of iOS-based playgrounds. This prevents the use of hardware-specific features that might only be present on a device.
The performance of playground scripts is mainly driven based on how many lines are executed and how much output is saved by the debugger. It should not be used to test the performance of performance-sensitive code.

Although the playground is well suited to present user interface components, it cannot be used for user input.

Anything requiring entitlements (such as in-app purchases or access to iCloud) is not possible in playground at the time of writing.

Note that while earlier releases of playground did not support custom frameworks, Xcode 6.1 permits frameworks to be loaded into playground, provided that the framework is built and marked as `public` and that it is in the same workspace as the playground.

Summary

This chapter presented playgrounds, an innovative way of running Swift code with graphical representations of values and introspection of running code. Both expressions and the timeline were presented as a way of showing the state of the program at any time, as well as graphically inspecting objects using QuickLook. The XCPlayground framework can also be used to record specific values and allow asynchronous code to be executed.

Being able to mix code and documentation into the same playground is also a great way of showing what functions exist, and how to create self-documenting playgrounds was presented. In addition, tools for the creation of such playgrounds using either AsciiDoc or Markdown (CommonMark) were introduced.

The next chapter will look at how to create an iOS application with Swift.
Where to buy this book
You can buy Swift Essentials 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.