Your First Swift App

Your First Swift App

Everything you need to know to build and submit your first iOS app.

Ash Furrow

This book is for sale at http://leanpub.com/yourfirstswiftapp

This version was published on 2016-02-10

Contents

Chapter 1: Introduction **1**
- Xcode 1
- Basic Project 3
- App Delegate 4
- Storyboard 11
- View Controller 12
- Other Files 15

Chapter 1: Introduction

Congratulations! You've bought my book and that's a great first step toward becoming a proficient iOS developer writing in Swift.

This chapter will give you a feet-first introduction to the ins and outs of the tools used to write iOS apps. We're going to begin with Xcode, then move onto basic Swift syntax. We'll take a look at the app delegate and create a very, very small app to poke around.

I'm going to assume you have *some* basic knowledge of object-oriented programming. If you'd like a refresher, Oracle has a great description of OOP to get you started.

We're going to be building an application called "Coffee Timer" to help users brew coffee and tea. Before we start anything, I'd like to present the Application Definition Statement.

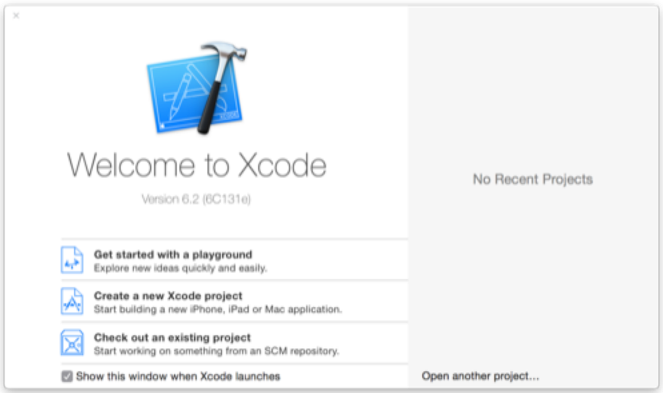
> Coffee Timer helps people time coffee and tea to brew delicious beverages.

We're going to start out small, then augment our application in each chapter until we have something worth submitting to the App Store.

Let's get started.

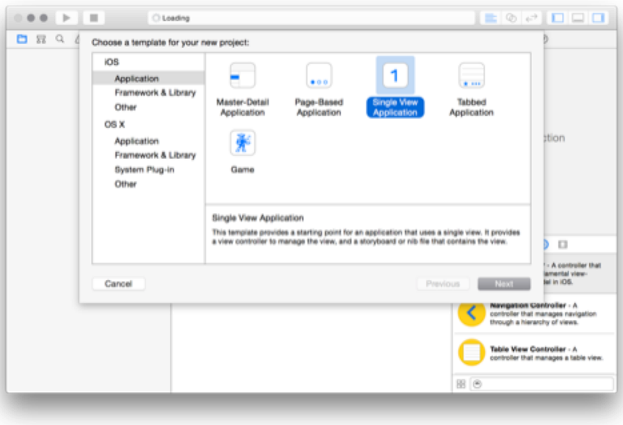
Xcode

Xcode is the Integrated Development Environment we're going to use to build our apps. Xcode 6, the version you need to write to write Swift is available from the Xcode downloads page. You'll see the welcome prompt.


Welcome to Xcode
Version 6.2 (6C131e)
Get started with a playground
Explore new ideas quickly and easily.
Create a new Xcode project
Start building a new iPhone, iPad or Mac application.
Check out an existing project
Start working on something from an SCM repository.
Show this window when Xcode launches
No Recent Projects
Open another project...

Xcode welcome prompt

Click on "Create a new Xcode project" to get started. This will bring up the new project dialogue window.


Loading
Choose a template for your new project:
iOS
Application
Framework & Library
Other
OS X
Application
Framework & Library
System Plug-in
Other
Master-Detail Application
Page-Based Application
Single View Application
Tabbed Application
Game
Cancel
Previous
Next
Navigation Controller - A controller that manages navigation through a hierarchy of views.
Table View Controller - A controller that manages a table view.

Single View Application

This template provides a starting point for an application that uses a single view. It provides a view controller to manage the view, and a storyboard or nib file that contains the view.

Xcode new project dialogue

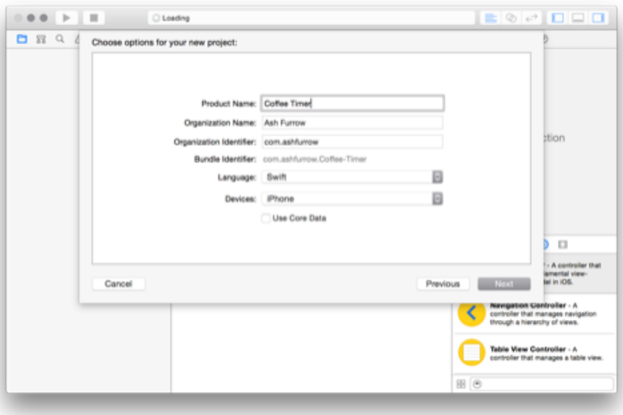
Make sure that "Application" under "iOS" is selected in the left hand pane. We're going to use the "Single View Application" template. Xcode is going to ask us for some details.

We'll give our app a name. This is going to be what's seen under the app icon on the home screen, so it's important to pick a great name. Don't worry – you can change this later if you have to. This book is going to focus on writing a timer to help people brew coffee, so we'll use the name "Coffee Timer" for our app.

The name you choose for your application will affect some directory names, some class names, as well as the copyright notices at the beginning of your files.

I use my name as an "Organization Name." This does not affect anything in the App Store at all. A "Company Identifier" is a required field and the convention is to use a reverse DNS name for your website. My website is "ashfurrow.com", so my company identifier is "com.ashfurrow" – again, you can change this later, so don't fret too much! The goal here is just to get started!

Make sure, of course, that you choose "Swift" as the programming language. Finally, select "iPhone" as the device and make sure that "Use Core Data" is *not* selected. I've filled in these values as shown in the next figure.

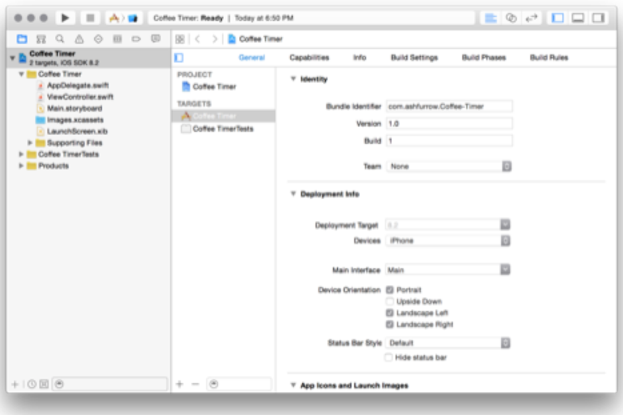

Loading
Cancel
Previous
Next
Navigation Controller - A controller that manages navigation through a hierarchy of views.
Table View Controller - A controller that manages a table view.

Choose options for your new project:

Product Name: Coffee Timer
Organization Name: Ash Furrow
Organization Identifier: com.ashfurrow
Bundle Identifier: com.ashfurrow.Coffee-Timer
Language: Swift
Devices: iPhone
☐ Use Core Data

Creating our first project

Save the project anywhere. If you want to use git to keep track of changes in your project, feel free to check the "Create local git repository" checkbox in the save dialogue. I won't be covering how to use git in this book. After saving, you'll have a (mostly) empty project.



Identity

Bundle Identifier: com.ashfurrow.Coffee-Timer
Version: 1.0
Build: 1
Team: None

Deployment Info

Deployment Target: 8.2
Devices: iPhone
Main Interface: Main
Device Orientation: ☑ Portrait ☐ Upside Down ☑ Landscape Left ☑ Landscape Right
Status Bar Style: Default
☐ Hide status bar

App Icons and Launch Images

Newly created project

Basic Project

Awesome! Let's build and run the app. You can either hit the "Run" button with the "Play" symbol on it, or press command-R to run the app. This will compile the code and install the app onto whatever device is specified in the "Scheme" drop-down menu (top left). For now, we're not going to worry about running the app on actual devices – it's a complicated process and you're better off diving into code as soon as possible. We'll cover testing your app on devices in the final chapter.

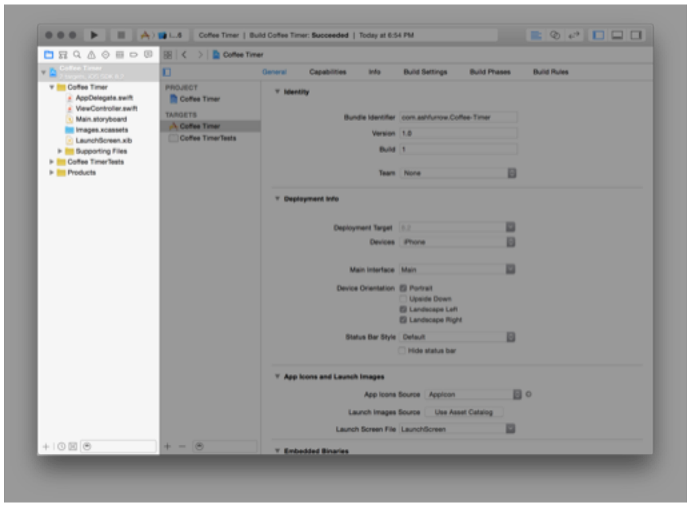
The app is shown here.

First run

This is a pretty boring app so far, but it is an app. You can hit the home button in the simulator (command-shift-H) to see the Home Screen of the simulator and there's your app. Not bad! Let's go back to Xcode to explore the parts of an app that are created for you.

App Delegate

Take a look at the files list on the left hand side of the Xcode window. You'll see a bunch of files. Let's go through each one, one at a time. Open the Project Navigator (command-1).


Coffee Timer | Build Coffee Timer: **Succeeded** | Today at 6:54 PM

Coffee Timer
- Coffee Timer
  - AppDelegate.swift
  - ViewController.swift
  - Main.storyboard
  - Images.xcassets
  - LaunchScreen.xib
  - Supporting Files
- Coffee TimerTests
- Products

PROJECT
Coffee Timer
TARGETS
Coffee Timer
Coffee TimerTests

General | Capabilities | Info | Build Settings | Build Phases | Build Rules

Identity

Bundle Identifier	com.ashfurrow.Coffee-Timer
Version	1.0
Build	1
Team	None

Deployment Info

Deployment Target	8.2
Devices	iPhone
Main Interface	Main
Device Orientation	✓ Portrait
	☐ Upside Down
	✓ Landscape Left
	✓ Landscape Right
Status Bar Style	Default
	☐ Hide status bar

App Icons and Launch Images

App Icons Source	AppIcon
Launch Images Source	Use Asset Catalog
Launch Screen File	LaunchScreen

Embedded Binaries

Project Navigator

Let's go through the files one-by-one and take a look at what they do. The first file is `AppDelegate.swift`.

```
//
//  AppDelegate.swift
//  Coffee Timer
//
//  Created by Ash Furrow on 2014-07-26.
//  Copyright (c) 2014 Ash Furrow. All rights reserved.
//

import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?

    func application(application: UIApplication, didFinishLaunchingWithOptions l\
aunchOptions: [NSObject : AnyObject]?) -> Bool {
        // Override point for customization after application launch.
        return true
    }

    func applicationWillResignActive(application: UIApplication) {
        // Sent when the application is about to move from active to inactive st\
ate. This can occur for certain types of temporary interruptions (such as an inc\
oming phone call or SMS message) or when the user quits the application and it b\
egins the transition to the background state.
        // Use this method to pause ongoing tasks, disable timers, and throttle \
down OpenGL ES frame rates. Games should use this method to pause the game.
    }

    func applicationDidEnterBackground(application: UIApplication) {
        // Use this method to release shared resources, save user data, invalida\
te timers, and store enough application state information to restore your applic\
ation to its current state in case it is terminated later.
        // If your application supports background execution, this method is cal\
led instead of applicationWillTerminate: when the user quits.
    }

    func applicationWillEnterForeground(application: UIApplication) {
        // Called as part of the transition from the background to the inactive \
state; here you can undo many of the changes made on entering the background.
    }

    func applicationDidBecomeActive(application: UIApplication) {
        // Restart any tasks that were paused (or not yet started) while the app\
lication was inactive. If the application was previously in the background, opti\
onally refresh the user interface.
    }

    func applicationWillTerminate(application: UIApplication) {
        // Called when the application is about to terminate. Save data if appro\
priate. See also applicationDidEnterBackground:.
    }


}
```

That is a lot of code! *Don't worry*, we're going to go through it line-by-line so you understand *exactly* what is going on.

The group of lines at the top, the comment, is a standard copyright notice Xcode inserts into each file you create. The next line is an `import` statement. This imports other framework files so we can reference the classes specified in them.

In this case, we've imported `UIKit`. UIKit is the framework for displaying user interfaces in iOS apps. We need this `import` so we can access `UIResponder`, `UIApplicationDelegate`, and `UIWindow` later on.

Note that classes that belong to most system frameworks, like UIKit, have a two-character "prefix" at the beginning of them. For example, we say "UIWindow" instead of just "Window". This is an archaic holdover from Objective-C, which doesn't have support for namespacing. Classes created in Swift *do not* have this prefix, so don't worry.

Next, we see the two lines below.

```
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
```

This is a little tricky. `@UIApplicationMain` indicates that this is the *entry point* for our application. Only one of them exists in the project. While it's important to have, Xcode will always put it in the right spot for you when you create a new project, so don't worry about it.

`class` means that we're declaring a new class, and that class' name is `AppDelegate`. The colon after the class name indicates the start of the type inheritance clause: `UIResponder, UIApplicationDelegate`.

This list contains the class that `AppDelegate` inherits from, as well as the list of protocols that it conforms to. Protocols are covered in more detail in Chapter 4, so we won't worry about them right now. In this list, the superclass must come first, so `AppDelegate` extends the `UIResponder` class. I'm not going to go into the specifics of what `UIResponder` *is* right now; just know that the app delegate extends it.

Classes in Swift do not need to have a superclass at all. These are called "pure" Swift classes. However, because the frameworks we use to build iOS apps are still compatible with Objective-C, we often subclass an Objective-C class, as is the case with our `AppDelegate`.

The rest of the type inheritance clause contains the list of protocols that the class conforms to, which is only `UIApplicationDelegate`. This is probably the *most important* part of this line of code.

Protocols are like interfaces in other languages like Java or C#: they are a guarantee that this object will implement certain methods. Other classes using `AppDelegate` can expect that it has these.

That's actually only mostly true. Protocols may specify *optional methods*, which is the case with `UIApplicationDelegate`. I'll describe in more detail what an application delegate is shortly. First, let's take a look at the rest of the Swift file.

The next line is `var window: UIWindow?`. This means that there is a property called "window" of a type `UIWindow?`. The `:` should be read as "is of type."

But what's with that question mark? Well, Swift has a concept called "Optional Types" that are similar to "Nullable Types" in C#. These are types that could be `nil`. See, unlike languages like Objective-C or Java, Swift has *compile-time* checks for objects being `nil`, which avoids different types of errors. If you're familiar with Java, you'll probably be familiar with the NullPointerException that occurs when you try and call a method on an object which is `null`. Getting these types of errors is incredibly rare in Swift. We're getting a little ahead of ourselves, but let's take a look at optional types.

Optional Types

An optional type is any type that ends in a question mark. Any type can become an optional type, included classes, integers, and strings. Consider our `window` property. If we want to call or access a property on `window`, we need to check if it is `nil`. There are two ways to do this. The first is to have an `if` statement that explicitly checks for `nil`, and then explicitly unwraps the `window` property.

```
if window != nil {
    var frame = window!.frame
}
```

We use the `!` operator to explicitly unwrap the `window` optional. If this unwrapping fails, because `window` is actually `nil`, then the app will crash, so always check first. While that works, it's super-ugly. We can do a little better.

```
if let window = window {
    var frame = window.frame
}
```

That's a bit better. We're defining a new `window` variable, which is local to the `if` statement. That statement will only be executed if the `window` property is not `nil`. Inside the `if` statement, the window is a regular `UIWindow` type, not an optional type any more.

That's still pretty ugly. Let's do better.

```
var frame = window?.frame
```

Here we see the most succinct way of unwrapping an optional, the `?` operator. If the `window` property is `nil`, then the line stops executing immediately.

Note that in the first two code samples, the type of `frame` is `CGRect`, but in the third sample, the type is `CGRect?`, an optional type itself. That's because in the first two samples, we assign the variable to the `frame` property on a *non-optional* `window`. In the last sample, we assign it to `window?.frame`, which might be `nil` if `window` is `nil`, which means `frame` could be `nil`, too.

Note that it is best to prefer non-optional types. We'll discuss "default values" in Chapter 3.

So far, we now know that this is a class named `AppDelegate` that extends `UIResponder` and conforms to the `UIApplicationDelegate` protocol, and that it has one property called `window`. But what on Earth is an application delegate?

An application delegate, or app delegate, is a special class used by the app. All apps have exactly *one* app delegate. The app delegate is responsible for being notified about certain important application events, like finishing launching or quitting. It's the job of the app delegate to let the rest of the app know when something important happens. Let's continue to look at the rest of the file to see what I mean.

The rest of the file contains a list of empty methods that Xcode has stubbed out for us. Let's look at what a method looks like.

```
func application(application: UIApplication, didFinishLaunchingWithOptions
    launchOptions: [NSObject : AnyObject]?) -> Bool {
    // Override point for customization after application launch.
    return true
}
```

`func` specifies that this is the beginning of a Swift function. The method name is a little weird because of its Objective-C roots. The abbreviated method name is `application(:, didFinish-LaunchingWithOptions:)`, and it returns a `Bool`. The other identifiers in the brackets, `application` and `launchOptions`, are the names of the parameters passed into this method. It's all a little tricky, but you'll get the hang of it. Instead of explaining the arcane and confusing rules of Swift function naming and Objective-C interoperability, we're just going to keep moving on. Don't worry, you'll develop an intuition around the syntax.

Next, let's look at the types of the parameters. `UIApplication` and `NSDictionary`. These give you access to the `UIApplication` instance for the app lifecycle, and any launch options (if the app was launched from, say, tapping on a push notification).

Let's take a look at all of the methods defined in this class.

- `application(application:, didFinishLaunchingWithOptions launchOptions:)`
- `applicationWillResignActive(application:)`
- `applicationDidEnterBackground(application:)`
- `applicationWillEnterForeground(application:)`
- `applicationDidBecomeActive(application:)`
- `applicationWillTerminate(application:)`

Xcode will also have filled these in with descriptions of what each stub does.

Application state is a little complicated. Once launched, an app is in the foreground until the user opens another app or returns to the Home Screen. Then it enters the background. From there, it stops executing until opened again, but it's still in memory. In the background, an application will either be returned to the foreground and become the active application again, or it will be terminated.

These methods should all be fairly straight forward, except for maybe `applicationWillResignActive()`. This is called when an application is still in the foreground, but isn't actively being used by the user. The best example of this scenario is when someone double-taps their Home Button to open the multitasking interface. The application is still in the foreground, but not actively responding to user events.

Why would you care? Well, games and videos should pause and OpenGL should throttle down its frame rate. Other than that, it's not terribly important.

So that's what an app delegate does out of the box: it responds to application events. For now, it won't matter much, but the app delegate is a central component of any application and deserves a good overview.

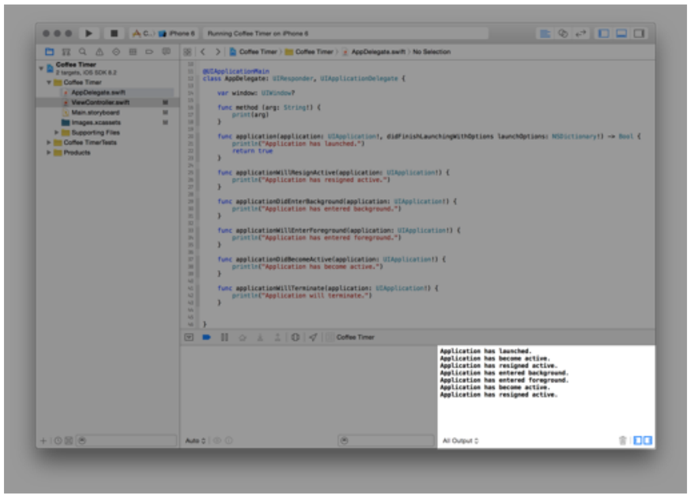
Let's throw some logging information into the app and re-run it. Add the following logging information to your methods.

```
func application(application: UIApplication, didFinishLaunchingWithOptions
launchOptions: [NSObject : AnyObject]?) -> Bool {

    print("Application has launched.")
    return true
}

func applicationWillResignActive(application: UIApplication) {
    print("Application has resigned active.")
}

func applicationDidEnterBackground(application: UIApplication) {
    print("Application has entered background.")
}

func applicationWillEnterForeground(application: UIApplication) {
    print("Application has entered foreground.")
}

func applicationDidBecomeActive(application: UIApplication) {
    print("Application has become active.")
}
```

```
func applicationWillTerminate(application: UIApplication) {
    print("Application will terminate.")
}
```

Re-run the app and play with it. Simulate a press of the Home Button (command-shift-H). Do it twice in a row, quickly, like you would double-tap the home button on a real device. See how the application moves from foreground to resigning its active state.

Logging information will be displayed in the console, which will be opened for you when something is printed there.



```
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?

    func method (arg: String!) {
        print(arg)
    }

    func application(application: UIApplication!, didFinishLaunchingWithOptions launchOptions: NSDictionary!) -> Bool {
        println("Application has launched.")
        return true
    }

    func applicationWillResignActive(application: UIApplication!) {
        println("Application has resigned active.")
    }

    func applicationDidEnterBackground(application: UIApplication!) {
        println("Application has entered background.")
    }

    func applicationWillEnterForeground(application: UIApplication!) {
        println("Application has entered foreground.")
    }

    func applicationDidBecomeActive(application: UIApplication!) {
        println("Application has become active.")
    }

    func applicationWillTerminate(application: UIApplication!) {
        println("Application will terminate.")
    }


}
```

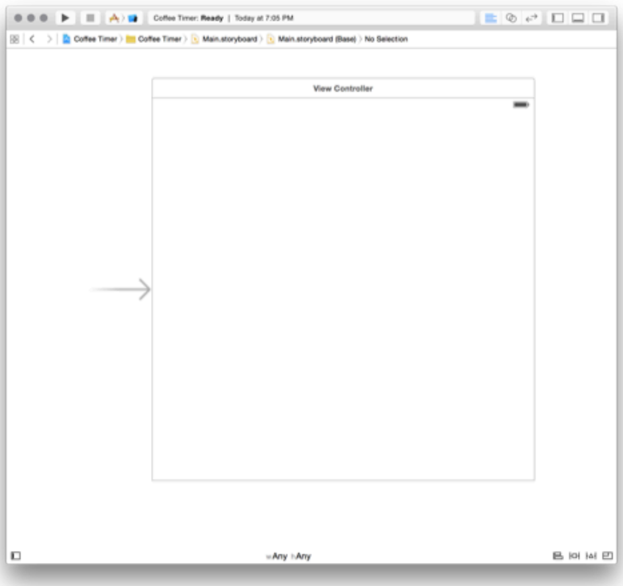
```
Application has launched.
Application has become active.
Application has resigned active.
Application has entered background.
Application has entered foreground.
Application has become active.
Application has resigned active.
```

Logging console

> Depending on the format you're reading this book in, and on what device, sometimes code is too long to fit on a single line. When this happens, sometimes a backslash (\) is inserted at the end of a line, and the line continues below.
>
> When this happens, it's really important to **not** type the backslash into Xcode. If you see a backslash at the end of a line of code, it's not meant to be typed in.

Storyboard

Next comes the fun part: the storyboard. The storyboard visually lays out everything the user is going to see and interact with inside of your app. For now, the storyboard looks a little boring.



Main storyboard

Your main storyboard is named `MainStoryboard.storyboard` in Xcode. Click it once to open it. The first thing I want you to do, and this is *very important*, is to open the File Inspector in the right hand pane (comand-alt-1) and find the checkbox labelled "Use Autolayout".

Uncheck this box.

When prompted, you'll want to "Disable Size Classes" as well. Autolayout is this hot new technology from Apple that's fairly complicated to use in practice, so we're going to stick with an older way of doing things for now. Once you master the basics of iOS development, I'd highly recommend you learn Autolayout.

Feel free to explore around the storyboard. Notice the buttons to zoom in and out. These will be handy once the storyboard becomes larger. Notice the hierarchy of the storyboard on the left hand side. You can hide it with the arrow on the left – useful for developing on smaller screens. The "View Controller" is the parent of the "View", as you might expect. Click on view controller in the hierarchy, then open the Identity inspector (command-alt-3). The Identity Inspect can also be shown by opening the View menu, then selecting Utilities and clicking "Identity Inspector".

What's that? The "Custom Class" is set as `ViewController`. Interesting … that's the name of the other class Xcode created for us.

View Controller

In fact, when iOS unpacks your storyboard, it uses the value in the Custom Class to know which class to use. It *instantiates* an instance of this class to use.

Let's explore the view controller files, then we'll come back to how it interacts with the storyboard. Let's take a look at the `ViewController.swift` file.

I'm omitting Xcode's copyright comment at the top of source files for the rest of this book.

```
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }

}
```

Here we have the definition of our `ViewController` class. It extends the `UIViewController` class and overrides two of the functions defined there. Those two functions are `viewDidLoad` and `didReceiveMemoryWarning`. We're going to ignore the memory warning function and focus for now on `viewDidLoad`.

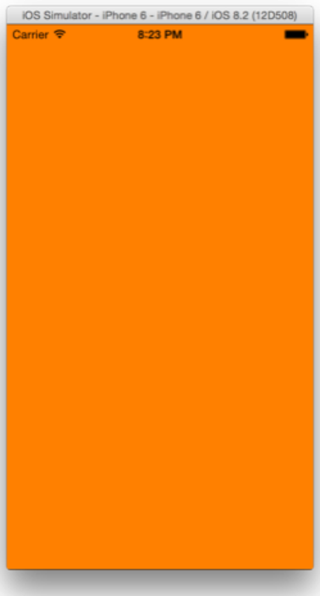
When the storyboard is woken up to be shown to the user, it needs to create a view controller. The class it uses for that controller is set by the Custom Class property in the storyboard, like we saw earlier.

Once that view controller is create, it "loads" the user interface shown in the storyboard. Once this view is loaded, the `viewDidLoad` function is called. This is your opportunity to customize the view *after* it has been loaded from the storyboard, but *before* it's first displayed to the user. Let me show you what I mean.

Replace the `viewDidLoad` implementation with this one:

```
override func viewDidLoad() {
    super.viewDidLoad()
    print("View is loaded.")
    view.backgroundColor = UIColor.orangeColor()
}
```

Hey! The background colour of the view is now orange.



Orange background set in `viewDidLoad`

Swift lets us reduce the verbosity of our code in a really neat way, using a technology called *type inference*. Swift knows that the `backgroundColor` property is a `UIColor`, so we can replace our method above with the following.

```
override func viewDidLoad() {
    super.viewDidLoad()
    print("View is loaded.")
~~    view.backgroundColor = UIColor.orangeColor()~~
    view.backgroundColor = .orangeColor()
}
```

There is another file that is central to your application: your asset library. It's a file called "Assets.xcassets", and it contains all of the images used throughout our app. If you open the asset catalogue, you'll notice that there is an image already defined for us: AppIcon. We'll cover that in Chapter 8.

And finally, there is “LaunchScreen.storyboard”. This is a quite complex topic that we’ll cover in Chapter 8, too.

Other Files

The files we’ve talked about so far are the main files that you’ll work with 99% of the time in Xcode.

The `Info.plist` is the file that specifies a lot of the metadata about our app. These are values like our version number, which interface orientations we support, and the default language. You don’t change these very often.

That’s it! You’ve created your first Xcode project and explored every nook and cranny of it. It’s not important to understand every piece – we’re going to do more in-depth exploring in the next chapter. What’s important is that you’re familiar with the over-arching concepts: what the app delegate is and what it’s for, that views are laid out in the storyboard, and that code to modify views goes in the view controller.