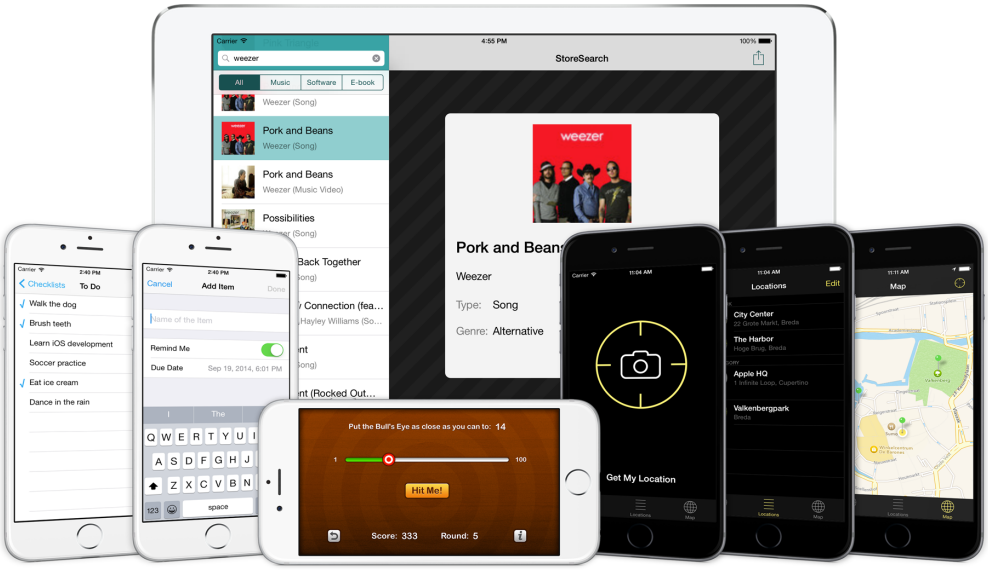
Chapter 1: Introduction

Hi, welcome to *The iOS Apprentice: Beginning iOS Development with Swift, Sixth Edition*, the swiftest way (pardon the pun) to iOS development mastery!

In this book you'll learn how to make your own iPhone and iPad apps using Apple's Swift 4.0 programming language (and Xcode 9), by building four interesting iOS apps.



The apps you'll be making in The iOS Apprentice

Everybody likes games, right? So, you'll start by building a simple but fun iPhone game named *Bull's Eye*. It will teach you the basics of iPhone programming, and the other apps will build on what you learn there.

Taken together, the four apps you'll build cover everything you need to know to make your own apps. By the end of the book you'll be experienced enough to turn your ideas into real apps that you can put on the App Store!

Even if you've never programmed before or if you're new to iOS, you should be able to follow along with the step-by-step instructions and understand how these apps are made. Each chapter has a ton of illustrations to prevent you from getting lost. Not everything might make sense right away, but hang in there and all will become clear in time.

Writing your own iOS apps is a lot of fun, but it's also hard work. If you have the imagination and perseverance, there is no limit to what you can make your apps do. It is my sincere belief that this book can turn you from a complete newbie into an accomplished iOS developer, but you do have to put in the time and effort. By writing this book, I've done my part. The rest is up to you...

About this book

The iOS Apprentice will help you become an excellent iOS developer, but only if you let it. Here are some tips that will help you get the most out of this book.

Learn through repetition

You're going to make several apps in this book. Even though the apps will start out quite simple, you may find the instructions hard to follow at first – especially if you've never done any computer programming before – because I will be introducing a lot of new concepts.

It's OK if you don't understand everything right away, as long as you get the general idea. As you proceed through the book, you'll go over many of these concepts again and again until they solidify in your mind.

Follow the instructions yourself

It is important that you not just read the instructions but also actually **follow them**. Open Xcode, type in the source code fragments, and run the app in the Simulator. This helps you to see how the app gets built step by step.

Even better, play around with the code and with the Xcode settings. Feel free to modify any part of the app and see what the results are - make a small change to the code and see how it affects the entire app. Experiment and learn! Don't worry about breaking stuff – that's half the fun. You can always find your way back to the beginning. But better still, you might even learn something from simply breaking the code and learning how to fix it.

Don't panic – bugs happen!

You will run into problems, guaranteed. Your programs will have strange bugs that will leave you stumped. Trust me, I've been programming for 30 years and that still happens to me too. We're only humans and our brains have a limited capacity to deal with complex programming problems. In this book, I will give you tools for your mental toolbox that will allow you to find your way out of any hole you have dug for yourself.

Understanding beats copy-pasting

Too many people attempt to write iOS apps by blindly copy-pasting code that they find on blogs and other websites, without really knowing what that code does or how it should fit into their program.

There is nothing wrong with looking on the web for solutions – I do it all the time – but I want to give you the tools and knowledge to understand what you're doing and why. That way you'll learn quicker and write better programs.

This is hands-on practical advice, not a bunch of dry theory (although we can't avoid *some* theory). You are going to build real apps right from the start and I'll explain how everything works along the way, with lots of pictures that illustrate what is going on.

I will do my best to make it clear how everything fits together, why we do things a certain way, and what the alternatives are.

Do the exercises

I will also ask you to do some thinking of your own – yes, there are exercises! It's in your best interest to actually do these exercises. There is a big difference between knowing the path and walking the path... And the only way to learn programming is to do it.

I encourage you to not just do the exercises but also to play with the code you'll be writing. Experiment, make changes, try to add new features. Software is a complex piece of machinery and to find out how it works you sometimes have to put some spokes in the wheels and take the whole thing apart. That's how you learn!

Have fun!

Last but not least, remember to have fun! Step by step you will build up your understanding of programming while making fun apps. By the end of this book you'll have learned the essentials of Swift and the iOS development kit. More importantly, you should have a pretty good idea of how everything goes together and how to think like a programmer.

It is my aim that by the time you reach the end of the book you will have learned enough to stand on your own two feet as a developer. I am confident that eventually you'll be able to write any iOS app you want as long as you get those basics down. You still may have a lot to learn, but when you're through with *The iOS Apprentice*, you can do without the training wheels.

Who this book is for

This book is great whether you are completely new to programming, or whether you come from a different programming background and are looking to learn iOS development.

If you're a complete beginner, don't worry – this book doesn't assume you know anything about programming or making apps. Of course, if you do have programming experience, that helps. Swift is a new programming language but in many ways it's similar to other popular languages such as PHP, C#, or JavaScript.

If you've tried iOS development before with the old language, Objective-C, then its low-level nature and strange syntax may have put you off. Well, there's good news: now that we have a modern language in Swift, iOS development has become a lot easier to pick up.

It is not my aim with this book to teach you all the ins and outs of iOS development. The iOS SDK (Software Development Kit) is huge and there is no way we can cover everything. Fortunately, we don't need to. You just need to master the essential building blocks of Swift and the iOS SDK. Once you understand these fundamentals, you can easily find out by yourself how the other parts of the SDK work and learn the rest on your own terms.

The most important thing I'll be teaching you, is how to think like a programmer. That will help you approach any programming task, whether it's a game, a utility, a mobile app that uses web service, or anything else you can imagine.

As a programmer you'll often have to think your way through difficult computational problems and find creative solutions. By methodically analyzing these problems you will be able to solve them, no matter how complex. Once you possess this valuable skill, you can program anything!

iOS 11 and better only

The code in this book is aimed exclusively at iOS version 11 and later. Each new release of iOS is such a big departure from the previous one that it just doesn't make sense anymore to keep developing for older devices and iOS versions. Things move fast in the world of mobile computing!

The majority of iPhone, iPod touch, and iPad users are pretty quick to upgrade to the latest version of iOS anyway, so you don't need to be too worried that you're leaving potential users behind.

Owners of older devices, such as the iPhone 4S, iPhone 5, or the first iPads, may be stuck with older iOS versions but this is only a tiny portion of the market. The cost of supporting these older iOS versions for your apps is usually greater than the handful of extra customers it brings you.

It's ultimately up to you to decide whether it's worth making your app available to users with older devices, but my recommendation is that you focus your efforts where they matter most. Apple as a company always relentlessly looks towards the future – if you want to play in Apple's backyard, it's wise to follow their lead. So back to the future it is!

What you need

It's a lot of fun to develop for the iPhone and iPad, but like most hobbies (or businesses!) it will cost some money. Of course, once you get good at it and build an awesome app, you'll have the potential to make that money back many times.

You will have to invest in the following:

iPhone, iPad, or iPod touch. I'm assuming that you have at least one of these. iOS 11 runs on the following devices: iPhone 5s or newer, iPad 5th generation or newer, iPad mini 2 or newer, 6th generation iPod touch - basically any device which has a 64-bit processor. With iOS 11, Apple dropped support for 32-bit processors and apps. If you have an older device, then this is a good time to think about getting an upgrade. But don't worry if you don't have a suitable device: you can do most of your testing on the Simulator.

> **Note:** Even though I mostly talk about the iPhone in this book, everything I say applies equally to the iPad and iPod touch. Aside from small hardware differences, they all use iOS and you program them in exactly the same way. You should also

> be able to run the apps from this book on your iPad or iPod touch without problems.

Mac computer with an Intel processor. Any Mac that you've bought in the last few years will do, even a Mac mini or MacBook Air. It needs to have at least macOS 10.12.4 Sierra. Xcode, the development environment for iOS apps, is a memory-hungry tool. So, having 4 GB of RAM in your Mac is no luxury. You might be able to get by with less, but do yourself a favor and upgrade your Mac. The more RAM, the better. A smart developer invests in good tools!

With some workarounds it is possible to develop iOS apps on Windows or a Linux machine, or a regular PC that has macOS installed (a so-called "Hackintosh"), but you'll save yourself a lot of hassle by just getting a Mac.

If you can't afford to buy the latest model, then consider getting a second-hand Mac from eBay. Just make sure it meets the minimum requirements (Intel CPU, preferably more than 2 GB RAM). Should you happen to buy a machine that has an older version of OS X (10.11 El Capitan or earlier), you can upgrade to the latest version of macOS from the online Mac App Store for free.

Apple Developer Program account. You can download all the development tools for free and you can try out your apps on your own iPhone, iPad, or iPod touch while you're developing, so you don't have to join the Apple Developer Program just yet. But to submit finished apps to the App Store you will have to enroll in the paid developer program. This will cost you $99 per year.

See developer.apple.com/programs/ for more info.

Xcode

The first order of business is to download and install Xcode and the iOS SDK.

Xcode is the development tool for iOS apps. It has a text editor where you'll type in your source code and it has a visual editor for designing your app's user interface.

Xcode transforms the source code that you write into an executable app and launches it in the Simulator or on your iPhone. Because no app is bug-free, Xcode also has a debugger that helps you find defects in your code (unfortunately, it won't automatically fix them for you, that's still something you have to do yourself).

You can download Xcode for free from the Mac App Store (apple.co/2wzi1L9). This requires at least an up-to-date version of macOS Sierra (10.12.4), so if you're still running an older version of macOS, you'll first have to upgrade to the latest version of macOS (also available for free from the Mac App Store). Get ready for a big download, as the full Xcode package is about 5 GB.

Important: You may already have a version of Xcode on your system that came pre-installed with your version of macOS. That version could be hopelessly outdated, so don't use it. Apple puts out new releases on a regular basis and you are encouraged to always develop with the latest Xcode and the latest available SDK on the latest version of macOS.

I wrote this revision of this book with **Xcode version 9** and the **iOS 11** SDK on macOS Sierra (10.12.6). By the time you're reading this, the version numbers might have gone up again. I will do my best to keep the PDF versions of the book up-to-date with new releases of the development tools and iOS versions but don't panic if the screenshots don't correspond 100% to what you see on your screen. In most cases, the differences will be minor.

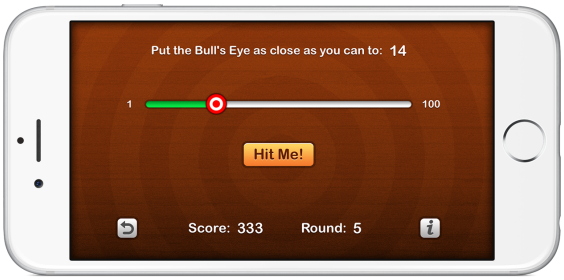
Many older books and blog posts (anything before 2010) talk about Xcode 3, which is radically different from Xcode 9. More recent material may mention Xcode versions 4, 5, 6, 7, or 8, which at first glance are similar to Xcode 9 but differ in many of the details. So if you're reading an article and you see a picture of Xcode that looks different from yours, they might be talking about an older version. You may still be able to get something out of those articles, as the programming examples are still valid. It's just Xcode that is slightly different.

What's ahead: an overview

The iOS Apprentice is spread across four apps, moving from beginning to intermediate topics. For each app, you will build it from start to finish, from scratch! Let's take a look at what's ahead.

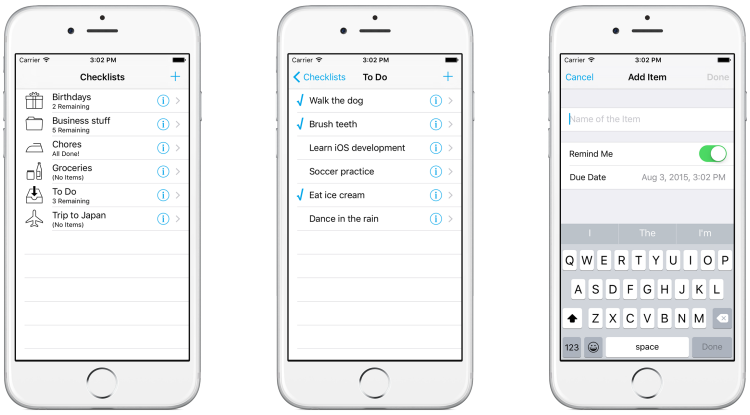
App 1: Bull's Eye

You'll start off by building a game called *Bull's Eye*. You'll learn how to use Xcode, Interface Builder, and Swift in an easy to understand manner.



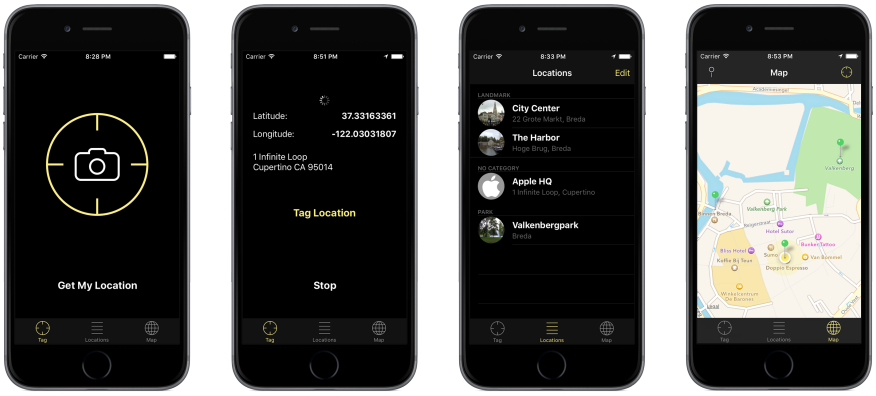
App 2: Checklists

For your next app, you'll create your own to-do list app. You'll learn about the fundamental design patterns that all iOS apps use, and about table views, navigation controllers, and delegates. Now you're making apps for real!



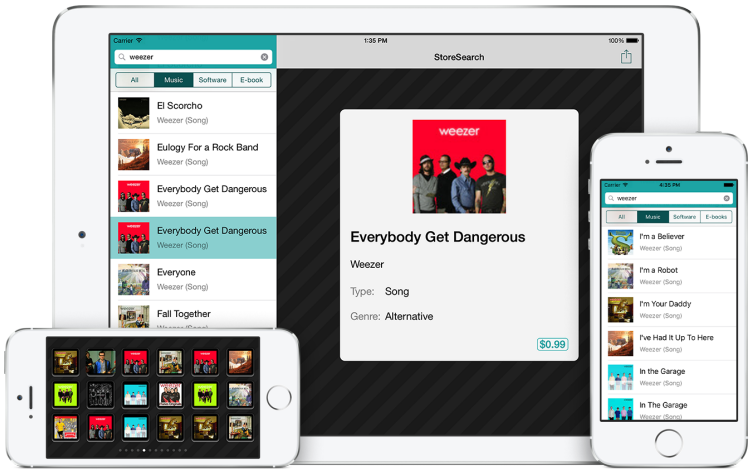
App 3: MyLocations

For your third app, you'll develop a location-aware app that lets you keep a list of spots that you find interesting. In the process, you'll learn about Core Location, Core Data, Map Kit, and much more!



App 4: StoreSearch

Mobile apps often need to talk to web services and that's what you'll do in you final app. You'll make a stylish app that lets you search for products on the iTunes store using HTTP requests and JSON.



Let's get started and turn you into a real iOS developer!

The language of the computer

The iPhone may pretend that it's a phone but it's really a pretty advanced computer that also happens to have the ability to make phone calls.

Like any computer, the iPhone works with ones and zeros. When you write software to run on the iPhone, you somehow have to translate the ideas in your head into those ones and zeros that the computer can understand.

Fortunately, you don't have to write any ones and zeros yourself. That would be a bit too much to ask of the human brain. On the other hand, everyday English is not precise enough to use for programming computers.

So, you will use an intermediary language, Swift, that is a little bit like English so it's reasonably straightforward for us humans to understand, while at the same time it can be easily translated into something the computer can understand as well.

This is an approximation of the language that the computer speaks:

```
Ltmp96:
    .cfi_def_cfa_register %ebp
    pushl   %esi
    subl    $36, %esp
Ltmp97:
    .cfi_offset %esi, -12
    calll   L7$pb
L7$pb:
    popl    %eax
    movl    16(%ebp), %ecx
    movl    12(%ebp), %edx
    movl    8(%ebp), %esi
    movl    %esi, -8(%ebp)
    movl    %edx, -12(%ebp)
    movl    %ecx, (%esp)
    movl    %eax, -24(%ebp)
    calll   _objc_retain
    movl    %eax, -16(%ebp)
    .loc    1 161 2 prologue_end
```

Actually, what the computer sees is this:

```
00011001010011110100100011001111100101000
00101000100111101011011100111010110100100
01010001110011111101011011000011100011000
10010000011100010100110100111100110011100
```

The `movl` and `calll` instructions are just there to make things more readable for humans. I don't know about you, but for me it's still hard to make much sense out of it.

It certainly is possible to write programs in that arcane language – that is what people used to do in the old days when computers cost a few million bucks apiece and took up a whole room – but I'd rather write programs that look like this:

```
func handleMusicEvent(command: Int, noteNumber: Int, velocity: Int) {

  if command == NoteOn && velocity != 0 {
    playNote(noteNumber + transpose, velocityCurve[velocity] / 127)

  } else if command == NoteOff ||
          (command == NoteOn && velocity == 0) {
    stopNote(noteNumber + transpose, velocityCurve[velocity] / 127)

  } else if command == ControlChange {
    if noteNumber == 64 {
      damperPedal(velocity)
    }
  }
}
```

The above code snippet is from a sound synthesizer program. It looks like something that almost makes sense. Even if you've never programmed before, you can sort of figure out what's going on. It's almost English.

Swift is a hot new language that combines traditional object-oriented programming with aspects of functional programming. Fortunately, Swift has many things in common with other popular programming languages, so if you're already familiar with C#, Python, Ruby, or JavaScript you'll feel right at home with Swift.

Swift is not the only option for making apps. Until recently, iOS apps were programmed in Objective-C, which is an object-oriented extension of the tried-and-true C language. Because of its heritage, Objective-C has some rough edges and is not really up to the demands of modern developers. That's why Apple created a new language.

Objective-C will still be around for a while but it's obvious that the future of iOS development is Swift. All the cool kids are using it already.

C++ is another language that adds object-oriented programming to C. It is very powerful but as a beginning programmer you probably want to stay away from it. I only mention it because C++ can also be used to write iOS apps, and there is an unholy marriage of C++ and Objective-C named Objective-C++ that you may come across from time to time.

I could have started *The iOS Apprentice* with an in-depth exploration of the features of Swift, but you'd probably fall asleep halfway. So instead, I will explain the language as we go along, very briefly at first but more in-depth later.

In the beginning, the general concepts – what is a variable, what is an object, how do you call a method, and so on – are more important than the details. Slowly but surely, all the arcane secrets of the Swift language will be revealed to you!

Are you ready to begin writing your first iOS app?

Chapter 2: The One-Button App

There's an old Chinese proverb that "A journey of a thousand miles begins with a single step." You are about to take that first step on your journey to iOS developer mastery. And you will take that first step by creating the *Bull's Eye* game.

This chapter covers the following:

- **The Bull's Eye game:** An introduction to the first app you'll make.
- **The one-button app:** Creating a simple one-button app where the button can take an action based on a tap on the button.
- **The anatomy of an app:** A brief explanation as to the inner-workings of an app.

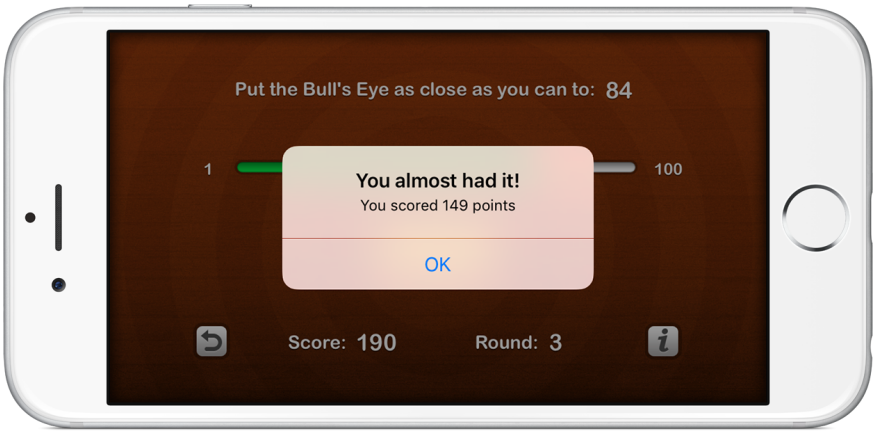
The Bull's Eye game

This is what the *Bull's Eye* game will look like when you're finished:

The finished Bull's Eye game

The objective of the game is to put the bull's eye, which is on a slider that goes from 1 to 100, as close to a randomly chosen target value as you can. In the screenshot above, the aim is to put the bull's eye at 84. Because you can't see the current value of the slider, you'll have to "eyeball" it.

When you're confident of your estimate, you press the "Hit Me!" button and a popup, also known as an alert, will tell you what your score is:



An alert popup shows the score

The closer to the target value you are, the more points you score. After you dismiss the alert popup by pressing the OK button, a new round begins with a new random target. The game repeats until the player presses the "Start Over" button (the curly arrow in the bottom-left corner), which resets the score to 0.

This game probably won't make you an instant millionaire on the App Store, but even future millionaires have to start somewhere!

Make a programming to-do list

> **Exercise:** Now that you've seen what the game will look like and what the gameplay rules are, make a list of all the things that you think you'll need to do in order to build this game. It's OK if you draw a blank, but give it a shot anyway.

I'll give you an example:

The app needs to put the "Hit Me!" button on the screen and show an alert popup when the user presses it.

Try to think of other things the app needs to do – it doesn't matter if you don't actually know how to accomplish these tasks. The first step is to figure out *what* you need to do; *how* to do these things is not important yet.

Once you know what you want, you can also figure out how to do it, even if you have to ask someone or look it up. But the "what" comes first. (You'd be surprised at how many people start writing code without a clear idea of what they're actually trying to achieve. No wonder they get stuck!)

Whenever I start working on a new app, I first make a list of all the different pieces of functionality I think the app will need. This becomes my programming to-do list. Having a list that breaks up a design into several smaller steps is a great way to deal with the complexity of a project.

You may have a cool idea for an app but when you sit down to write the program the whole thing can seem overwhelming. There is so much to do… and where to begin? By cutting up the workload into small steps you make the project less daunting – you can always find a step that is simple and small enough to make a good starting point and take it from there.

It's no big deal if this exercise is giving you difficulty. You're new to all of this! As your understanding grows of how software and the development process works, it will become easier to identify the different parts that make up a design, and to split it into manageable pieces.

This is what I came up with. I simply took the gameplay description and cut it into very small chunks:

- Put a button on the screen and label it "Hit Me!"
- When the player presses the Hit Me button the app has to show an alert popup to inform the player how well they did. Somehow you have to calculate the score and put that into this alert.
- Put text on the screen, such as the "Score:" and "Round:" labels. Some of this text changes over time, for example the score, which increases when the player scores points.
- Put a slider on the screen and make it go between the values 1 and 100.
- Read the value of the slider after the user presses the Hit Me button.
- Generate a random number at the start of each round and display it on the screen. This is the target value.
- Compare the value of the slider to that random number and calculate a score based on how far off the player is. You show this score in the alert popup.

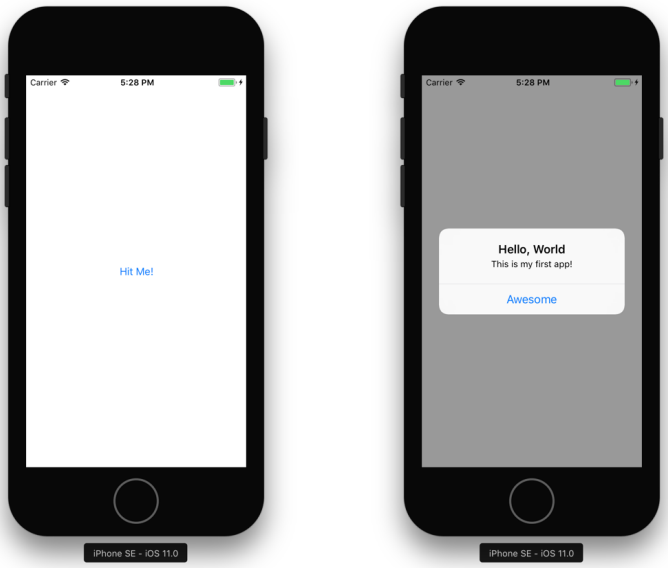
- Put the Start Over button on the screen. Make it reset the score and put the player back into the first round.
- Put the app in landscape orientation.
- Make it look pretty. :]

I might have missed a thing or two, but this looks like a decent list to start with. Even for a game as basic as this, there are quite a few things you need to do. Making apps is fun, but it's definitely a lot of work too!

The one-button app

Let's start at the top of the list and make an extremely simple first version of the game that just displays a single button. When you press the button, the app pops up an alert message. That's all you are going to do for now. Once you have this working, you can build the rest of the game on this foundation.

The app will look like this:



The app contains a single button (left) that shows an alert when pressed (right)

Time to start coding! I'm assuming you have downloaded and installed the latest version of the SDK and the development tools at this point.

In this book, you'll be working with **Xcode 9.0** or better. Newer versions of Xcode may also work but anything older than version 9.0 probably would be a no-go.

Because Swift is a very new language, it tends to change between versions of Xcode. If your Xcode is too old – or too new! – then not all of the code in this book may work properly. (For this same reason you're advised not to use beta versions of Xcode, only the official one from the Mac App Store.)

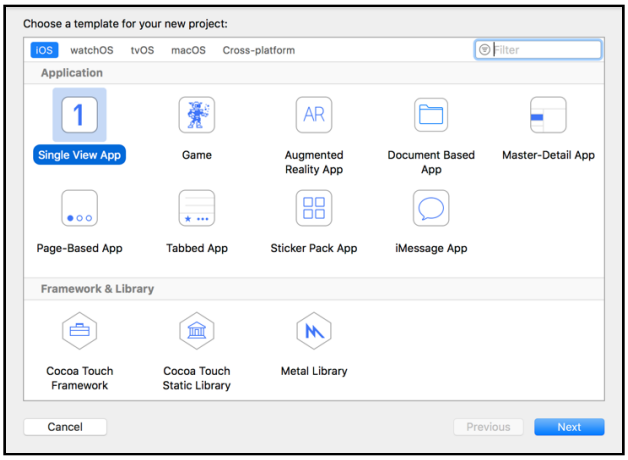
Create a new project

➤ Launch Xcode. If you have trouble locating the Xcode application, you can find it in the folder **/Applications/Xcode** or in your Launchpad. Because I use Xcode all the time, I placed its icon in my dock for easy access.

Xcode shows the "Welcome to Xcode" window when it starts:

Xcode bids you welcome

➤ Choose **Create a new Xcode project**. The main Xcode window appears with an assistant that lets you choose a template:

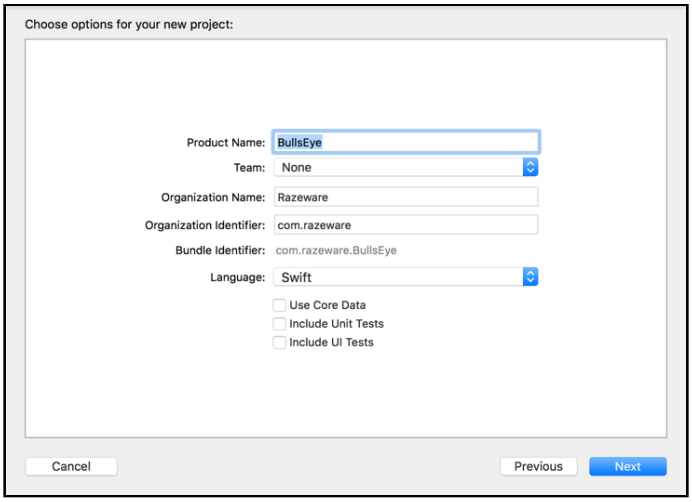


Choosing the template for the new project

Xcode has bundled templates for a variety of application styles. Xcode will make a pre-configured project for you based on the template you choose. The new project will already include some of the source files you need. These templates are handy because they can save you a lot of typing. They are ready-made starting points.

➤ Select **Single View Application** and press **Next**.

This opens a screen where you can enter options for the new app:



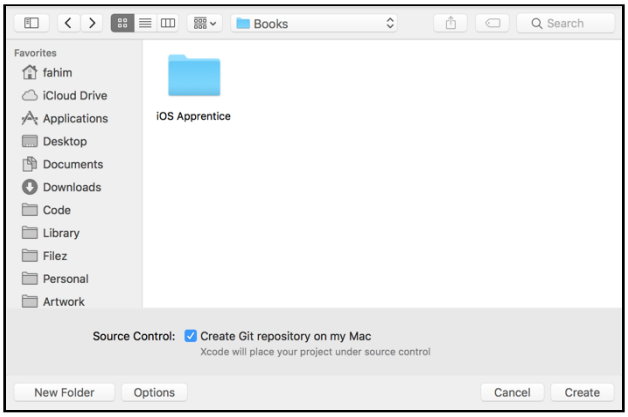
Configuring the new project

➤ Fill out these options as follows:

- Product Name: **BullsEye**. If you want to use proper English, you can name the project Bull's Eye instead of BullsEye, but it's best to avoid spaces and other special characters in project names.
- Team: If you already are a member of the Apple Developer Program, this will show your team name. For now, it's best to leave this setting alone; we'll get back to this later on.
- Organization Name: Fill in your own name here or the name of your company.
- Organization Identifier: Mine says "com.razeware". That is the identifier I use for my apps. As is customary, it is my domain name written in reverse. You should use your own identifier here. Pick something that is unique to you, either the domain name of your website (but backwards) or simply your own name. You can always change this later.
- Language: **Swift**

Make sure the three options at the bottom – Use Core Data, Include Unit Tests, and Include UI Tests – are *not* selected. You won't be using those in this project.

➤ Press **Next**. Now Xcode will ask where to save your project:



Choosing where to save the project

➤ Choose a location for the project files, for example the Desktop or your Documents folder.

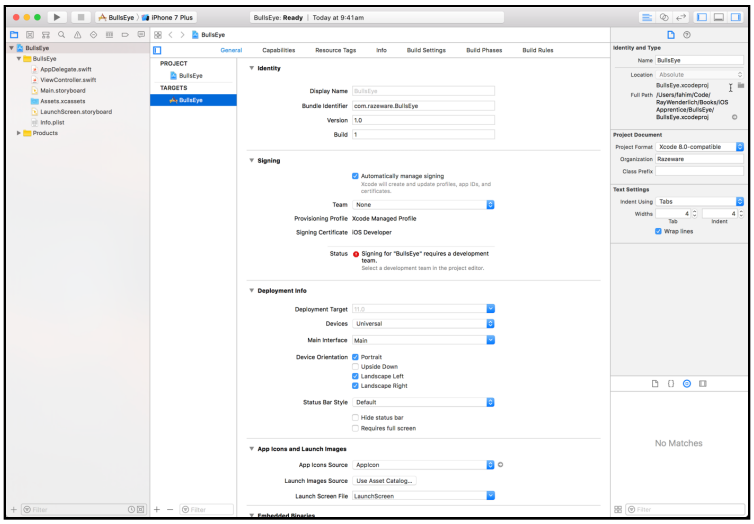
Xcode will automatically make a new folder for the project using the Product Name that you entered in the previous step (in your case BullsEye), so you don't need to make a new folder yourself.

At the bottom there is a checkbox that says, "Create Git repository on My Mac". You can ignore this for now. You'll learn about the Git version control system later on.

➤ Press **Create** to finish.

Xcode will now create a new project named BullsEye, based on the Single View Application template, in the folder you specified.

When it is done, the screen should look something like this:



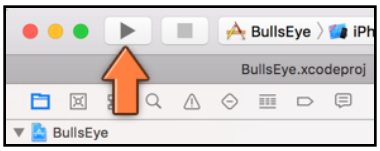
The main Xcode window at the start of your project

There may be small differences with what you're seeing on your own computer if you're using a version of Xcode newer than my version. Rest assured, any differences will only be superficial.

> **Note:** If you don't see a file named ViewController.swift in the list on the left but instead have ViewController.h and ViewController.m, then you picked the wrong language (Objective-C) when you created the project. Start over and be sure to choose Swift as the programming language.

Run your project

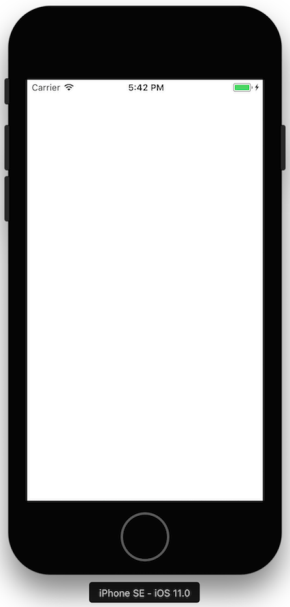
➤ Press the **Run** button in the top-left corner:



Press Run to launch the app

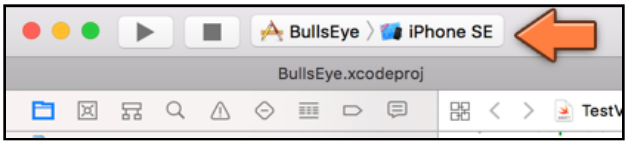
> **Note:** If this is the first time you're using Xcode, it may ask you to enable developer mode. Click **Enable** and enter your password to allow Xcode to make these changes.
>
> Also, make sure that you do not have your iPhone or iPad plugged in at this point to your computer, for example, for charging. If you do, it might switch to the actual device instead of the Simulator for running the app and since you are not yet set up for running on device, this could result in errors that might leave you scratching your head :]

Xcode will labor for a bit and then launch your brand new app in the iOS Simulator. The app may not look like much yet – and there is not anything you can do with it either – but this is an important first milestone in your journey!



What an app based on the Single View Application template looks like

If Xcode says "Build Failed" or "Xcode cannot run using the selected device" when you press the Run button, then make sure the picker at the top of the window says **BullsEye > iPhone SE** (or any other model number) and not **Generic iOS Device**:



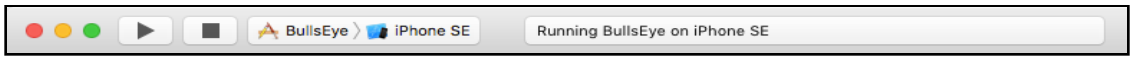
Making Xcode run the app on the Simulator

If your iPhone is currently connected to your Mac via USB cable, Xcode may have attempted to run the app on your iPhone and that may not work without some additional setting up. I'll show you how to get the app to run on your iPhone so you can show it off to your friends soon, but for now just stick with the Simulator.

➤ Next to the Run button is the **Stop** button (the square thingy). Press that to exit the app.

On your phone (or even the simulator) you'd use the home button to exit an app (on the Simulator you could also use the **Hardware → Home** item from the menu bar or use the handy ⇧+⌘+**H** shortcut), but that won't actually terminate the app. It will disappear from the Simulator's screen but the app stays suspended in the Simulator's memory, just as it would on a real iPhone.

Until you press Stop, Xcode's activity viewer at the top says "Running BullsEye on iPhone SE":



The Xcode activity viewer

It's not really necessary to stop the app, as you can go back to Xcode and make changes to the source code while the app is still running. However, these changes will not become active until you press Run again. That will terminate any running version of the app, build a new version, and launch it in the Simulator.

What happens when you press Run?

Xcode will first *compile* your source code – that is: translate it – from Swift into machine code that the iPhone (or the Simulator) can understand. Even though the programming language for writing iOS apps is Swift or Objective-C, the iPhone itself doesn't speak those languages. A translation step is necessary.

The compiler is the part of Xcode that converts your Swift source code into executable binary code. It also gathers all the different components that make up the app – source files, images, storyboard files, and so on – and puts them into the "application bundle".

This entire process is also known as *building* the app. If there are any errors (such as spelling mistakes for method names), the build will fail. If everything goes according to plan, Xcode copies the application bundle to the Simulator or the iPhone and launches the app. All that from a single press of the Run button.

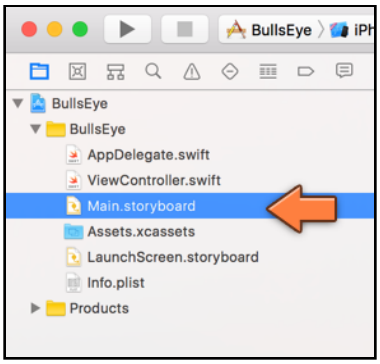
Add a button

I'm sure you're as unimpressed as I am with an app that just displays a dull white screen :] So, let's make it a bit more interesting by adding a button to it.

The left pane of the Xcode window is named the **Navigator area**. The row of icons along the top lets you select a specific navigator. The default navigator is the **Project navigator**, which shows the files in your project.

The organization of these files roughly corresponds to the project folder on your hard disk, but that isn't necessarily always so. You can move files around and put them into new groups and organize away to your heart's content. We'll talk more about the different files in your project later.

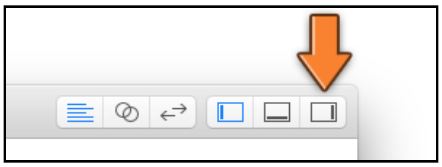
➤ In the **Project navigator**, find the item named **Main.storyboard** and click it once to select it:



The Project navigator lists the files in the project

Like a superhero changing his/her clothes in a phone booth, the main editing pane now transforms into the **Interface Builder**. This tool lets you drag-and-drop user interface components such as buttons to create the UI of your app. (OK, bad analogy, but Interface Builder is a super tool in my opinion.)

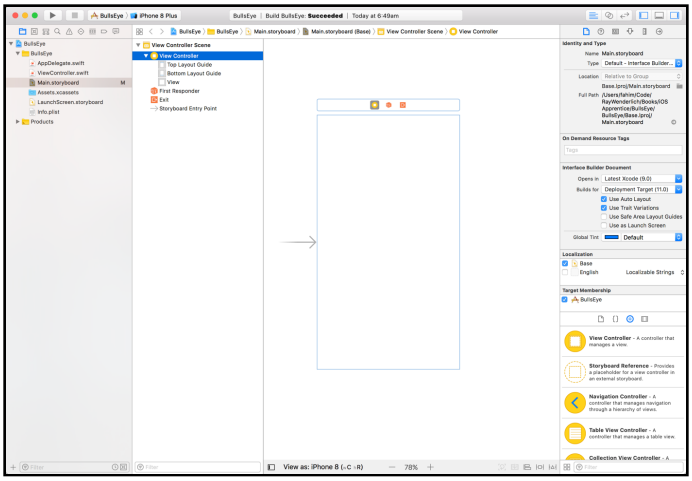
➤ If it's not already blue, click the **Hide or show utilities** button in Xcode's toolbar:



Click this button to show the Utilities pane

These toolbar buttons change the appearance of Xcode. This one in particular opens a new pane on the right side of the Xcode window.

Your Xcode should now look something like this:



Editing Main.storyboard in Interface Builder

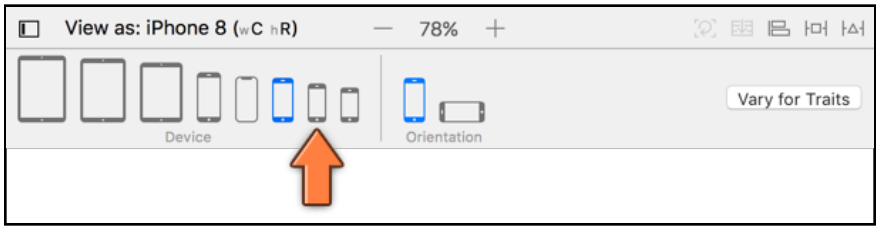
This is the *storyboard* for your app. The storyboard contains the designs for all of your app's screens, and shows the navigation flow in your app from one screen to another.

Currently, the storyboard contains just a single screen or *scene*, represented by a rectangle in the middle of the Interface Builder canvas.

> **Note:** If you don't see the rectangle labeled "View Controller" but only an empty white canvas, then use your mouse or trackpad to scroll the storyboard around a bit. Trust me, it's in there somewhere! Also make sure your Xcode window is large enough. Interface Builder takes up a lot of space...

The scene currently has the size of an iPhone 8. To keep things simple, you will first design the app for the iPhone SE, which has a slightly smaller screen. Later you'll also make the app fit on the larger iPhone models.

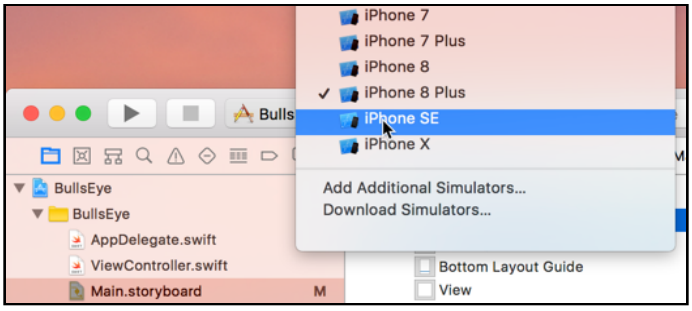
➤ At the bottom of the Interface Builder window, click **View as: iPhone 8** to open up the following panel:



Choosing the device type

Select the **iPhone SE**, the second smallest iPhone, thus resizing the preview UI you see in Interface Builder to be set to that of an iPhone SE. You'll notice that the scene's rectangle now becomes a bit smaller, corresponding to the screen size of the iPhone 5, iPhone 5s, and iPhone SE models.

➤ In the Xcode toolbar, make sure it says **BullsEye > iPhone SE** (next to the Stop button). If it doesn't then click it and pick iPhone SE from the list:

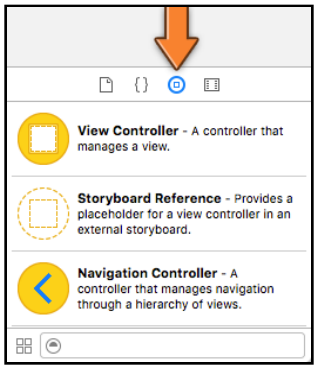


Switching the Simulator to iPhone SE

Now when you run the app, it will run on the iPhone SE Simulator (try it out!).

Back to the storyboard:

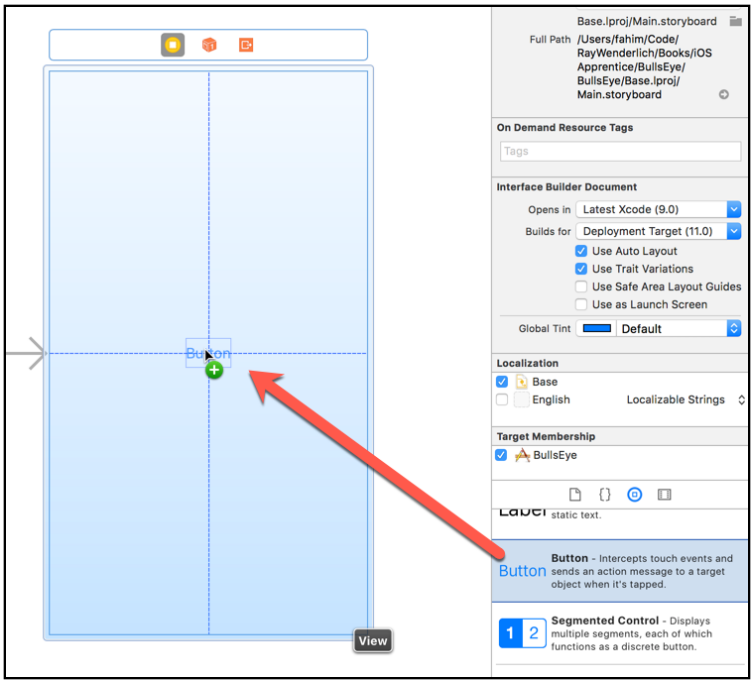
➤ At the bottom of the Utilities pane you will find the **Object Library** (make sure the third button, the one that looks like a circle, is selected):



The Object Library

Scroll through the items in the Object Library's list until you see **Button**. (Alternatively, you can type the word "button" in to the search/filter box at the bottom of the Object Library.)

➤ Click on **Button** and drag it into the working area, on top of the scene's rectangle.



Dragging the button on top of the scene

That's how easy it is to add new buttons, just drag & drop. That goes for all other user interface elements too. You'll be doing a lot of this, so take some time to get familiar with the process.

➤ Drag-and-drop a few other controls, such as labels, sliders, and switches, just to get the hang of it.

This should give you some idea of the UI controls that are available in iOS. Notice that the Interface Builder helps you to layout your controls by snapping them to the edges of the view and to other objects. It's a very handy tool!

➤ Double-click the button to edit its title. Call it Hit Me!

The button with the new title

It's possible that your button has a border around it:

The button with a bounds rectangle

This border is not part of the button, it's just there to show you how large the button is. You can turn these rectangles on or off using the **Editor → Canvas → Show Bounds Rectangles** menu option.

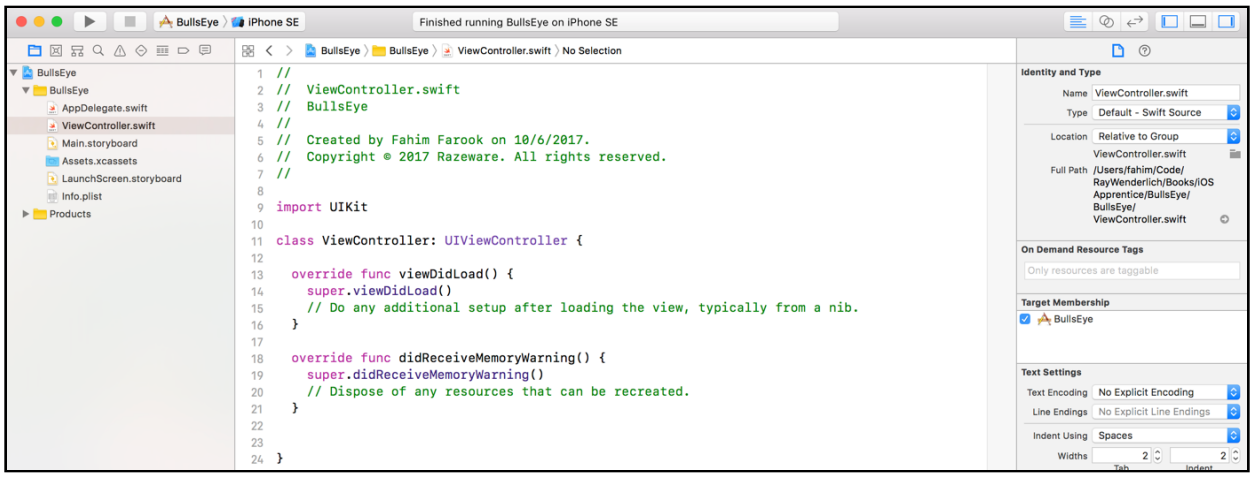
When you're done playing with Interface Builder, press the Run button from Xcode's toolbar. The app should now appear in the Simulator, complete with your "Hit Me!" button. However, when you tap the button it doesn't do anything yet. For that you'll have to write some Swift code!

The source code editor

A button that doesn't do anything when tapped is of no use to anyone. So, let's make it show an alert popup. In the finished game the alert will display the player's score, but for now we shall limit ourselves to a simple text message (the traditional "Hello, World!").

➤ In the **Project navigator**, click on **ViewController.swift**.

The Interface Builder will disappear and the editor area now contains a bunch of brightly colored text. This is the Swift source code for your app:



The source code editor

> **Note**: If your Xcode editor window does not show the line numbers as in the screenshot above, and you'd actually like to see the line numbers, from the menu bar choose **Xcode → Preferences... → Text Editing** and go to the **Editing** tab. There, you should see a **Line numbers** checkbox under **Show** - check it.

➤ Add the following lines directly above the very last } bracket in the file:

```
@IBAction func showAlert() {
}
```

The source code for **ViewController.swift** should now look like this:

```
//
//  ViewController.swift
//  BullsEye
//
//  Created by <you> on <date>.
//  Copyright © <year> <your organization>. All rights reserved.
//

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.
  }

  @IBAction func showAlert() {
  }
}
```

How do you like your first taste of Swift? Before I can tell you what this all means, I have to introduce the concept of a view controller.

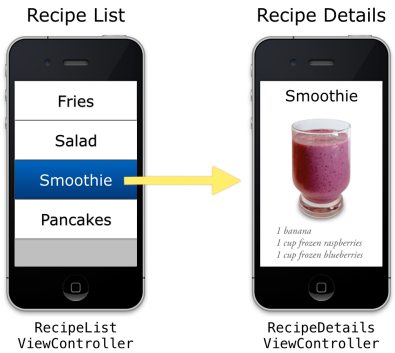
> **Xcode will autosave**
>
> You don't have to save your files after you make changes to them because Xcode will automatically save any modified files when you press the Run button. Nevertheless, Xcode isn't the most stable piece of software out there and occasionally it may crash on you before it has had a chance to save your changes, so I still like to press ⌘+**S** on a regular basis to save my files.

View controllers

You've edited the **Main.storyboard** file to build the user interface of the app. It's only a button on a white background, but a user interface nonetheless. You also added source code to **ViewController.swift**.

These two files – the storyboard and the Swift file – together form the design and implementation of a *view controller*. A lot of the work in building iOS apps is making view controllers. The job of a view controller is to manage a single screen in your app.

Take a simple cookbook app, for example. When you launch the cookbook app, its main screen lists the available recipes. Tapping a recipe opens a new screen that shows the recipe in detail with an appetizing photo and cooking instructions. Each of these screens is managed by a view controller.



The view controllers in a simple cookbook app

What these two screens do is very different. One is a list of several items; the other presents a detail view of a single item.

That's why you need two view controllers: one that knows how to deal with lists, and another that can handle images and cooking instructions. One of the design principles of iOS is that each screen in your app gets its own view controller.

Currently *Bull's Eye* has only one screen (the white one with the button) and thus only needs one view controller. That view controller is simply named "ViewController" and the storyboard and Swift file work together to implement it. (If you are curious, you can check the connection between the screen and the code for it by switching to the Identity inspector on the right sidebar of Xcode in the storyboard view. The class value shows the current class associated with the storyboard scene.)

Simply put, the Main.storyboard file contains the design of the view controller's user interface, while ViewController.swift contains its functionality – the logic that makes the user interface work, written in the Swift language.

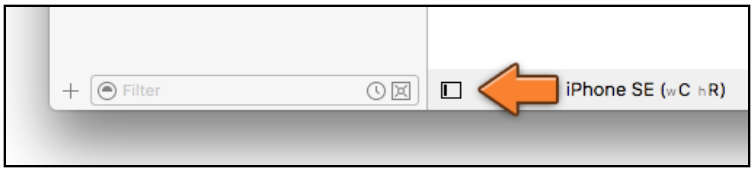
Because you used the Single View Application template, Xcode automatically created the view controller for you. Later you will add a second screen to the game and you will create your own view controller for that.

Make connections

The line of source code you have just added to ViewController.swift lets Interface Builder know that the controller has a "showAlert" action, which presumably will show an alert popup. You will now connect the button on the storyboard to that action in your source code.

➤ Click **Main.storyboard** to go back into Interface Builder.

In Interface Builder, there should be a second pane on the left, next to the navigator area, the **Document Outline**, that lists all the items in your storyboard. If you do not see that pane, click the small toggle button in the bottom-left corner of the Interface Builder canvas to reveal it.



The button that shows the Document Outline pane

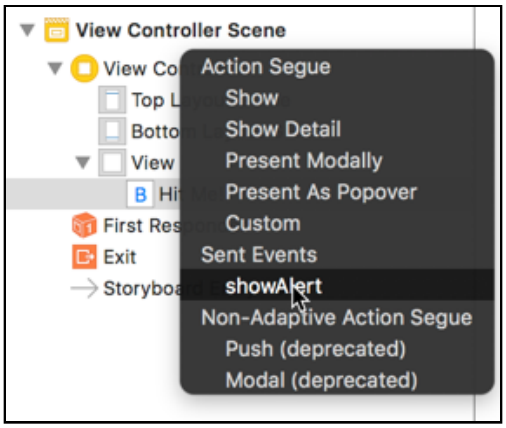
➤ Click the **Hit Me button** once to select it.

With the Hit Me button selected, hold down the **Control** key, click on the button and drag up to the **View Controller** item in the Document Outline. You should see a blue line going from the button up to View Controller.

(Instead of holding down Control, you can also right-click and drag, but don't let go of the mouse button before you start dragging.)

Ctrl-drag from the button to View Controller

Once you're on View Controller, let go of the mouse button and a small menu will appear. It contains two sections, "Action Segue" and "Sent Events", with one or more options below each. You're interested in the **showAlert** option under Sent Events. The Sent Events section shows all possible actions in your source code that can be hooked up to your storyboad and **showAlert** is the name of the action that you added earlier in the source code of **ViewController.swift**.



The popup menu with the showAlert action

➤ Click on **showAlert** to select it. This instructs Interface Builder to make a connection between the button and the line `@IBAction func showAlert()`.

From now on, whenever the button is tapped the `showAlert` action will be performed. That is how you make buttons and other controls do things: you define an action in the view controller's Swift file and then you make the connection in Interface Builder.

You can see that the connection was made by going to the **Connections inspector** in the Utilities pane on the right side of the Xcode window.

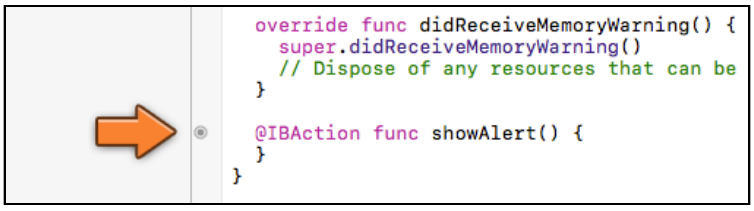
➤ Click the small arrow-shaped button at the top of the pane to switch to the Connections inspector:

The inspector shows the connections from the button to any other objects

In the Sent Events section, the "Touch Up Inside" event is now connected to the showAlert action. You can also see the connection in the Swift file.

➤ Select **ViewController.swift** to edit it.

Notice how to the left of the line with `@IBAction func showAlert()`, there is a solid circle? Click on that circle to reveal what this action is connected to.



A solid circle means the action is connected to something

Act on the button

You now have a screen with a button. The button is hooked up to an action named `showAlert` that will be performed when the user taps the button.

Currently, however, the action is empty and nothing will happen (try it out by running the app again, if you like). You need to give the app more instructions.

➤ In **ViewController.swift**, modify `showAlert` to look like the following:

```
@IBAction func showAlert() {
  let alert = UIAlertController(title: "Hello, World",
                                message: "This is my first app!",
                          preferredStyle: .alert)

  let action = UIAlertAction(title: "Awesome", style: .default,
                             handler: nil)

  alert.addAction(action)

  present(alert, animated: true, completion: nil)
}
```

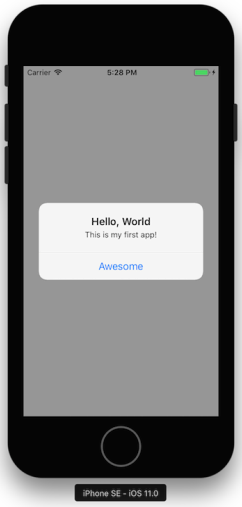
The new lines of code implement the actual alert display functionality.

The commands between the `{ }` brackets tell the iPhone what to do, and they are performed from top to bottom.

The code in `showAlert` creates an alert with a title "Hello, World", a message "This is my first app!" and a single button labeled "Awesome".

If you're not sure about the distinction between the title and the message: both show text, but the title is slightly bigger and in a bold typeface.

➤ Click the **Run** button from Xcode's toolbar. If you didn't make any typos, your app should launch in the Simulator and you should see the alert box when you tap the button.



The alert popup in action

Congratulations, you've just written your first iOS app! What you just did may have been mostly gibberish to you, but that shouldn't matter. We take it one small step at a time.

You can strike off the first two items from the to-do list already: putting a button on the screen and showing an alert when the user taps the button.

Take a little break, let it all sink in, and come back when you're ready for more! You're only just getting started...

> **Note:** Just in case you get stuck, I have provided the complete Xcode projects which are snapshots of the project as at the beginning and end of each chapter. That way you can compare your version of the app to mine, or – if you really make a mess of things – continue from a version that is known to work.

You can find the project files for each chapter in the corresponding folder.

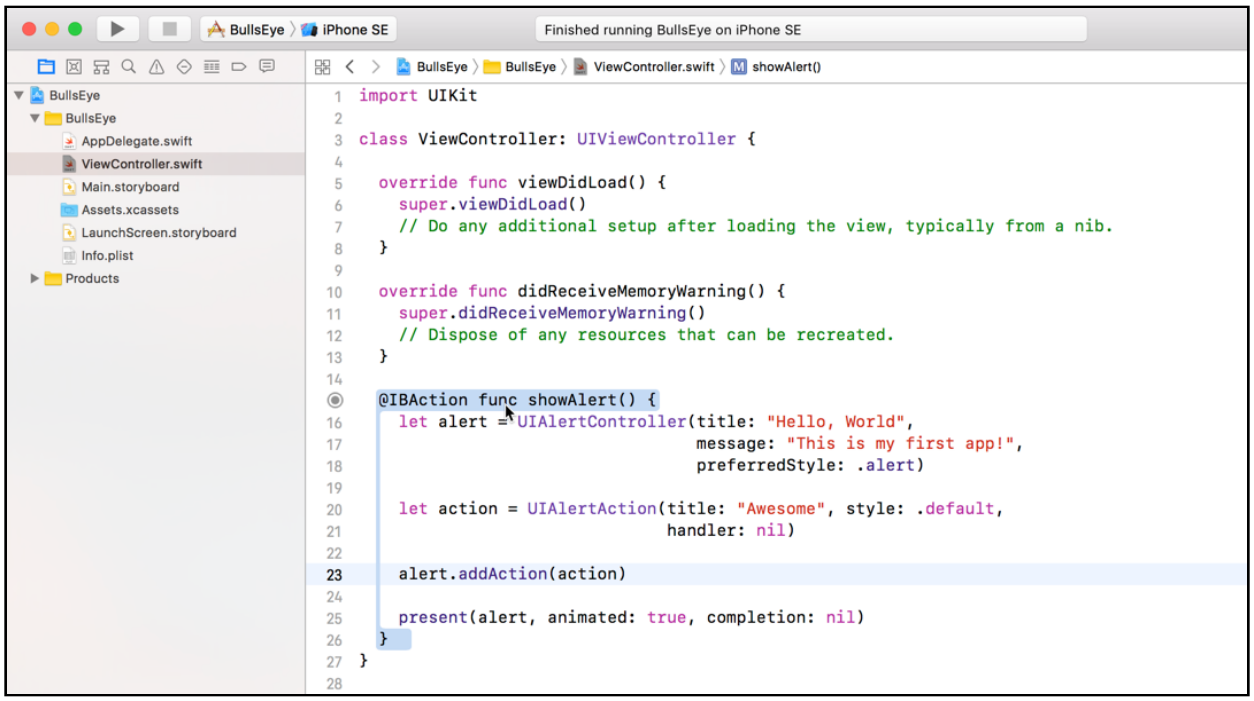
Problems?

If Xcode gives you a "Build Failed" error message after you press Run, then make sure you typed in everything correctly. Even the smallest mistake could potentially confuse Xcode. It can be quite overwhelming at first to make sense of the error messages that Xcode spits out. A small typo at the top of a source file can produce several errors elsewhere in that file.

Typical mistakes are differences in capitalization. The Swift programming language is case-sensitive, which means it sees `Alert` and `alert` as two different names. Xcode complains about this with a "<something> undeclared" or "Use of unresolved identifier" error.

When Xcode says things like "Parse Issue" or "Expected <something>" then you probably forgot a curly bracket `}` or parenthesis `)` somewhere. Not matching up opening and closing brackets is a common error.

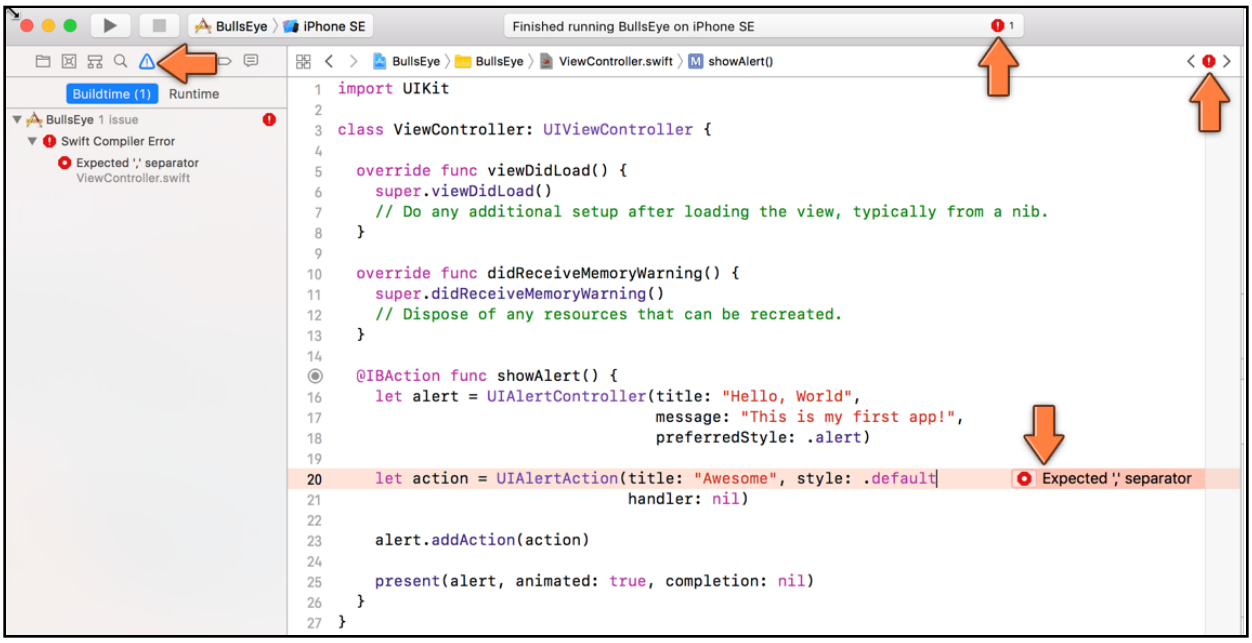
Tip: In Xcode, there are multiple ways to find matching brackets to see if they line up. If you move the text cursor over a closing bracket, Xcode will highlight the corresponding opening bracket, or vice versa. You could also hold down the ⌘ key and move your mouse cursor over a line with a curly bracket and Xcode will highlight the full block from the opening curly bracket to the closing curly bracket (or vice versa) - nifty!



Xcode shows you the complete block for curly brackets

Tiny details are very important when you're programming. Even one single misplaced character can prevent the Swift compiler from building your app.

Fortunately, such mistakes are easy to find.



Xcode makes sure you can't miss errors

When Xcode detects an error it switches the pane on the left from the Project navigator, to the **Issue navigator**, which shows all the errors and warnings that Xcode has found. (You can go back to the project files list using the small buttons at the top.)

In the above screenshot, apparently, I forgot a comma somewhere.

Click on the error message in the Issue navigator and Xcode takes you to the line in the source code with the error. Sometimes, it even suggests a fix to resolve it:

Fix-it suggests a solution to the problem

Sometimes it's a bit of a puzzle to figure out what exactly you did wrong when your build fails - fortunately, Xcode lends a helping hand.

Errors and warnings

Xcode makes a distinction between errors (red) and warnings (yellow). Errors are fatal. If you get one, you cannot run the app till the error is fixed. Warnings are informative. Xcode just says, "You probably didn't mean to do this, but go ahead anyway."

> In my opinion, it is best to treat all warnings as if they were errors. Fix the warning before you continue and only run your app when there are zero errors and zero warnings. That doesn't guarantee the app won't have any bugs, but at least they won't be silly ones :]

The anatomy of an app

It might be good at this point to get some sense of what goes on behind the scenes of an app.

An app is essentially made up of **objects** that can send messages to each other. Many of the objects in your app are provided by iOS, for example the button – a `UIButton` object – and the alert popup – a `UIAlertController` object. Some objects you will have to program yourself, such as the view controller.

These objects communicate by passing messages to each other. When the user taps the Hit Me button in the app, for example, that `UIButton` object sends a message to your view controller. In turn the view controller may message more objects.

On iOS, apps are *event-driven*, which means that the objects listen for certain events to occur and then process them.

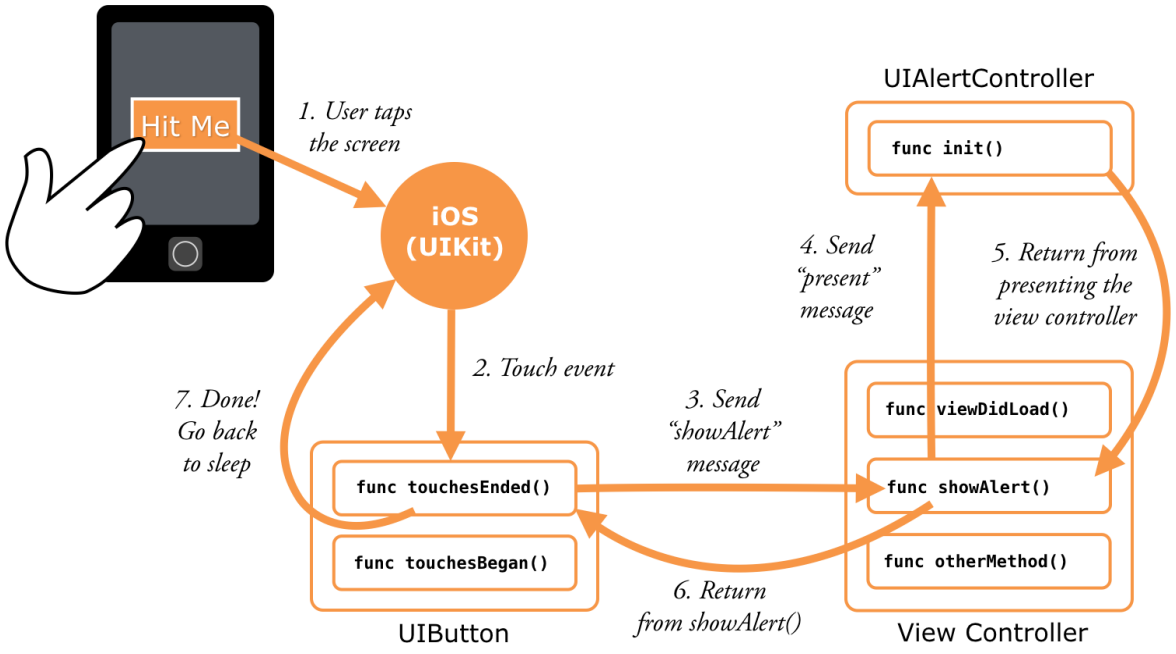
As strange as it may sound, an app spends most of its time doing... absolutely nothing. It just sits there waiting for something to happen. When the user taps the screen, the app springs to action for a few milliseconds and then it goes back to sleep again until the next event arrives.

Your part in this scheme is that you write the source code for the actions that will be performed when your objects receive the messages for such events.

In the app, the button's Touch Up Inside event is connected to the view controller's `showAlert` action. So when the button recognizes it has been tapped, it sends the `showAlert` message to your view controller.

Inside `showAlert`, the view controller sends another message, `addAction`, to the `UIAlertController` object. And to show the alert, the view controller sends the `present` message.

Your whole app will be made up of objects that communicate in this fashion.



The general flow of events in an app

Maybe you have used PHP or Ruby scripts on your web site. This event-based model is different from how a PHP script works. The PHP script will run from top-to-bottom, executing the statements one-by-one until it reaches the end and then it exits.

Apps, on the other hand, don't exit until the user terminates them (or they crash!). They spend most of their time waiting for input events, then handle those events and go back to sleep.

Input from the user, mostly in the form of touches and taps, is the most important source of events for your app, but there are other types of events as well. For example, the operating system will notify your app when the user receives an incoming phone call, when it has to redraw the screen, when a timer has counted down, and many more.

Everything your app does is triggered by some event.

You can find the project files for the app up to this point under **02 - The One-Button App** in the Source Code folder.