

Up to date for iOS 10,
Xcode 8 & Swift 3



ios Apprentice

FIFTH EDITION

Tutorial 1: Getting Started

By Matthijs Hollemans

iOS Apprentice

Matthijs Hollemans

Copyright ©2016 Razeware LLC.

Notice of Rights

All rights reserved. No part of this book or corresponding materials (such as text, images, or source code) may be reproduced or distributed by any means without prior written permission of the copyright owner.

Notice of Liability

This book and all corresponding materials (such as source code) are provided on an “as is” basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose, and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in action of contract, tort or otherwise, arising from, out of or in connection with the software or the use of other dealing in the software.

Trademarks

All trademarks and registered trademarks appearing in this book are the property of their own respective owners.

License

By purchasing *iOS Apprentice*, you have the following license:

- You are allowed to use and/or modify the source code in *iOS Apprentice* in as many apps as you want, with no attribution required.
- You are allowed to use and/or modify all art, images and designs that are included in *iOS Apprentice* in as many apps as you want, but must include this attribution line somewhere inside your app: “Artwork/images/designs: from *iOS Apprentice* book, available at www.raywenderlich.com”.
- The source code included in *iOS Apprentice* is for your personal use only. You are NOT allowed to distribute or sell the source code in *iOS Apprentice* without prior authorization.
- This book is for your personal use only. You are NOT allowed to sell this book without prior authorization, or distribute it to friends, coworkers or students; they would need to purchase their own copies.

About the author



Matthijs Hollemans is a mystic who lives at the top of a mountain where he spends all of his days and nights coding up awesome apps. Actually he lives below sea level in the Netherlands and is pretty down-to-earth but he does spend too much time in Xcode. Check out his website at www.matthijshollemans.com.

About the cover

Striped dolphins live to about 55-60 years of age, can travel in pods numbering in the thousands and can dive to depths of 700 m to feed on fish, cephalopods and crustaceans. Baby dolphins don't sleep for a full a month after they're born. That puts two or three sleepless nights spent debugging code into perspective, doesn't it? :]

Table of Contents: Extended

Tutorial 1: Getting Started.....	6
The language of the computer.....	13
The Bull's Eye game	16
The one-button app.....	18
How does an app work?	36
Working our way down the to-do list	38
Objects, data and methods	42
Adding the rest of the controls.....	45
Enough playing around... let's make a game!.....	56
Calculating the score.....	72
Polishing the game.....	85
Adding the About screen.....	95
Making it look good	104
Running the game on your device	144
The end... or the beginning?	150

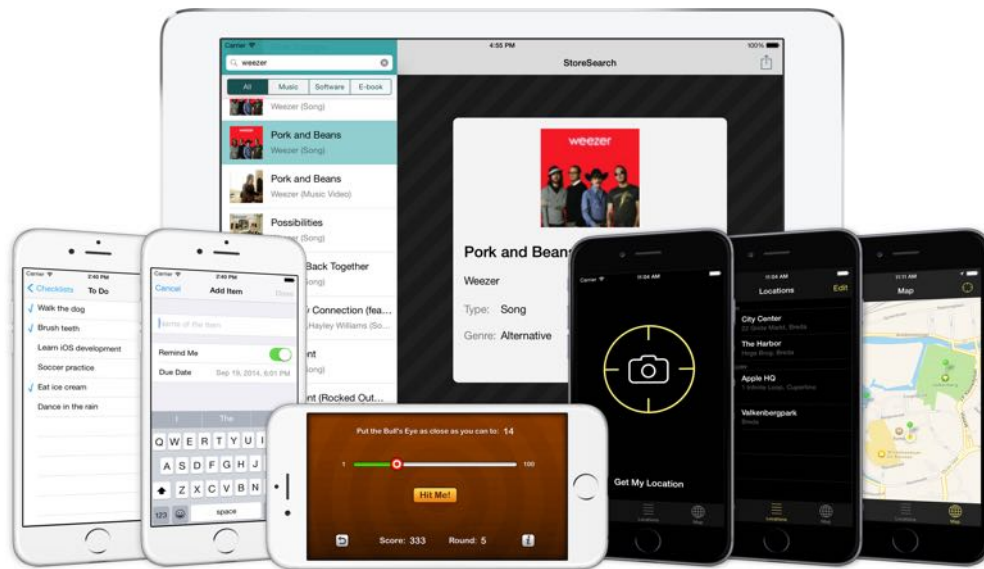
Tutorial 1: Getting Started

By Matthijs Hollemans

Hi! I am Matthijs Hollemans, a full-time iOS developer and tutorial team member at www.raywenderlich.com.

You're about to read the first tutorial from my book *The iOS Apprentice: Beginning iOS Development with Swift, Fifth Edition*.

In this book you will learn how to make your own iPhone and iPad apps with Apple's Swift 3.0 programming language, through a series of four epic-length hands-on tutorials.



The apps you'll be making in The iOS Apprentice

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

The best part is you can read it here in its entirety for free!

Each tutorial in this book describes a new app in full detail, and together they cover everything you need to know to make your own apps. By the end of the series 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 tutorial 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 iPhone and iPad 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 these cool devices do. It is my sincere belief that this series 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 these tutorials I've done my part, now it's up to you...

Enjoy the first tutorial! If it works out for you, then I hope you'll get the rest of the book from www.raywenderlich.com/store/ios-apprentice or Amazon.com.

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 a lot of apps in this book. Even though the apps will start out quite simple, you may find the tutorials 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. In the subsequent tutorials from this series you'll go over many of these concepts 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. Feel free to modify any part of the app and see what the results are. Experiment and learn! Don't worry about breaking stuff – that's half the fun. You can always find your way back to the beginning.

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 course, 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 iPhone 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 instruments 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 just 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 the series 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 after these tutorials 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 series to teach you all the ins and outs of iPhone and iPad development. The iOS SDK (Software Development Kit) is huge and there is no way we can cover everything – but 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 10 and better only

The tutorials in this series are aimed exclusively at iOS version 10 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 or the first iPads, may be stuck with iOS version 9 or earlier but this is only a tiny portion of the market. The cost of supporting these older iOS versions with 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 10 runs on the following devices: iPhone 5 or newer, iPad 4th generation or newer, iPad mini 2 or newer, 6th generation iPod touch. 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 everything in the Simulator.

Note: Even though I mostly talk about the iPhone in this tutorial series, 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 these tutorials 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 OS X 10.11 El Capitan or macOS 10.12 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 1 GB RAM). Should you happen to buy a machine that has an older version of OS X (10.10 Yosemite 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 (Software Development Kit).



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 (itunes.apple.com/app/xcode/id497799835). This requires at least OS X El Capitan (10.11), so if you're still running OS X Yosemite or even Mavericks 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 OS X. That version is 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 OS X.

I wrote the latest revision of this book with **Xcode version 8.0** and the **iOS 10.0** SDK on macOS Sierra (10.12). By the time you're reading this the version numbers have no doubt gone up again. I will do my best to keep the PDF versions of the tutorials 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 8. More recent material may mention Xcode versions 4, 5, 6, or 7, which at first glance are similar to Xcode 8 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're 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 the tool that is slightly different.

What's ahead: an overview

The *iOS Apprentice* is split into 4 tutorials, moving from beginning to intermediate topics. In each tutorial you will build a complete app, from scratch! Let's take a look at what's ahead.

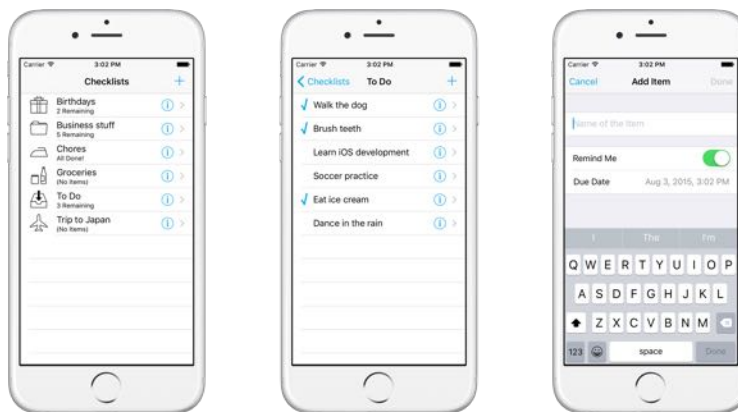
Tutorial 1: Getting Started

In the first tutorial 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 easygoing manner.



Tutorial 2: Checklists

In the second tutorial in the series, 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!



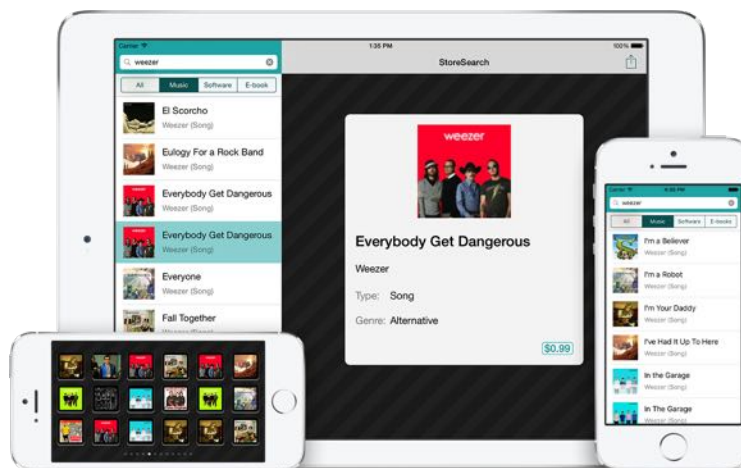
Tutorial 3: MyLocations

In the third tutorial in the series, 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!



Tutorial 4: StoreSearch

Mobile apps often need to talk to web services and that's what you'll do in this final tutorial of the series. 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 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.

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 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:

```
000110010100111101001000110011111001010
001010001001111010110111001110101101001
010100011100111110101110110000111000110
1001000001110001010011010011111001100111
```

The `movl` and `calll` instructions are just there to make things more readable for humans. Well, 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 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, iPhone and iPad 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 treatise on 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 secrets of the Swift language will be revealed to you.

Are you ready to begin writing your first iOS app?

The Bull's Eye game

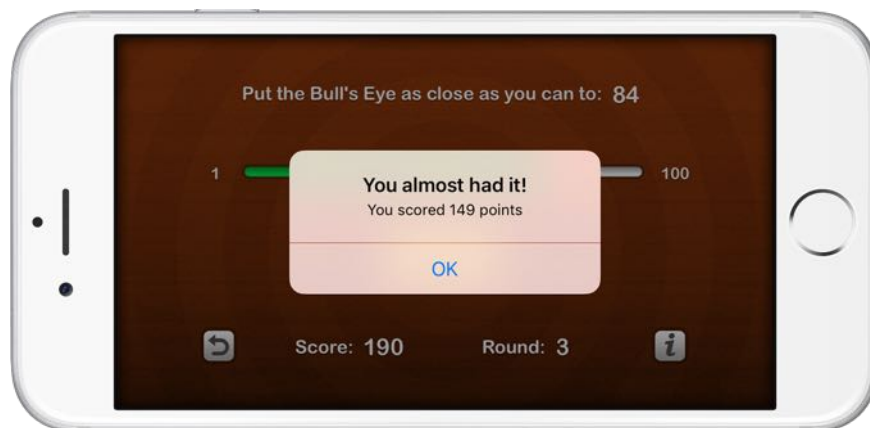
In this first lesson you're going to create a game called Bull's Eye. This is what the game will look like when you're finished:



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 22. 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!

Making 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 – no 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 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 she 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.

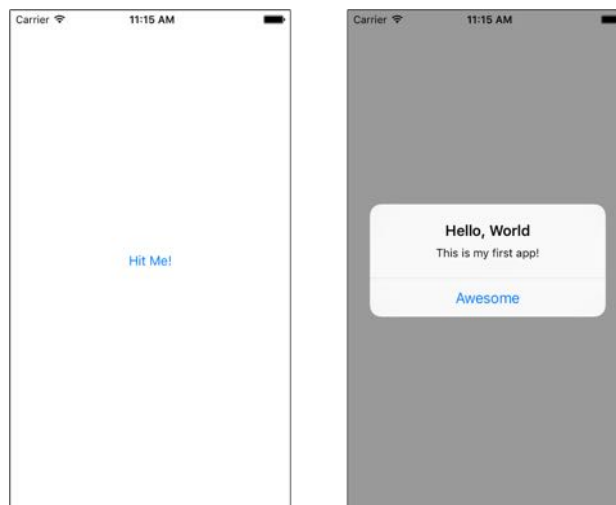
- Put a slider on the screen and make it go between the values 1 and 100.
- Read the value of the slider after the user presses the Hit Me button.
- Generate a random number at the start of each round and display it on the screen. This is the target value.
- Compare the value of the slider to that random 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 already 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

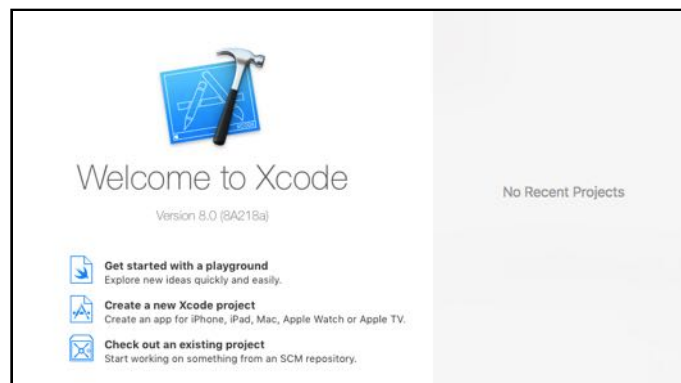
version of the SDK and the development tools at this point.

In this tutorial, you'll be working with **Xcode 8.0** or better. Newer versions of Xcode may also work but anything older than version 8.0 is 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.)

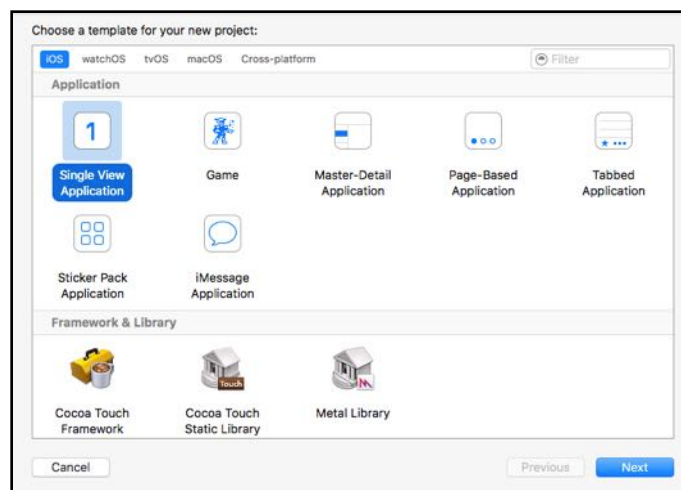
► **Launch Xcode.** If you have trouble locating the Xcode application, you can find it in the folder **/Applications/Xcode** or in your Launchpad. Because I use Xcode all the time, I placed its icon in my dock for easy access.

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:

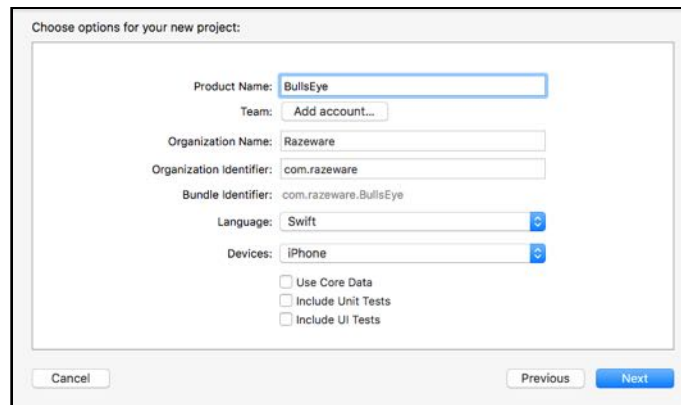


Choosing the template for the new project

There are 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 many 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:



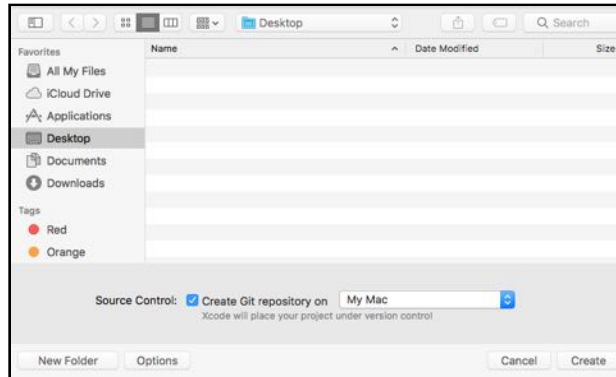
Configuring the new project

➤ 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 in the tutorial.
- 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 the other way around. 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**
- Devices: **iPhone**

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.

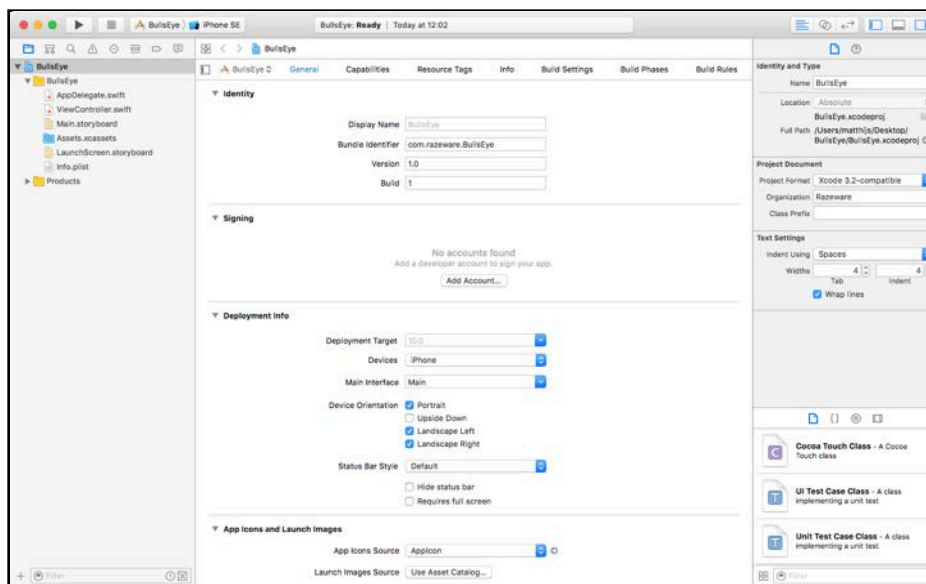
Xcode will automatically make a new folder for the project using the Product Name that you entered in the previous step (in your case BullsEye), so you don't need to make a new folder yourself.

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 in one of the next tutorials.

- 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 looks 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 8.0. 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 when you made the project (Objective-C). Start over and be sure to choose Swift as the programming language.

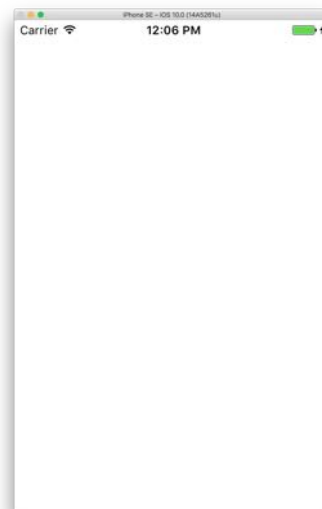
➤ Press the **Run** button in the top-left corner:



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.

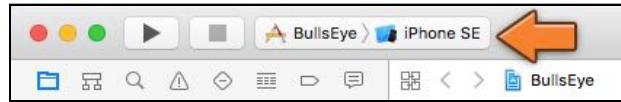
Xcode will labor for a bit and then it launches your brand new app in the iOS Simulator. The app may not look like much yet – and there is not anything you can do with it either – but this is an important first milestone in your journey.



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**:



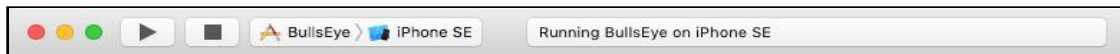
Making Xcode run the app on the Simulator

If your iPhone is currently connected to your Mac with the USB cable, Xcode may have attempted to run the app on your iPhone and that may not work without some additional setting up. At the end of this tutorial I'll show you how to get the app to run on your iPhone so you can show it off to your friends, 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 you'd use the home button to exit an app (on the Simulator choose the **Hardware** → **Home** item from the menu bar), 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 a machine code that the iPhone (or the Simulator) can understand. Even though the programming language for writing iPhone 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 so-called "application bundle".

This entire process is also known as *building* the app. If there are any errors

(such as spelling mistakes), 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.

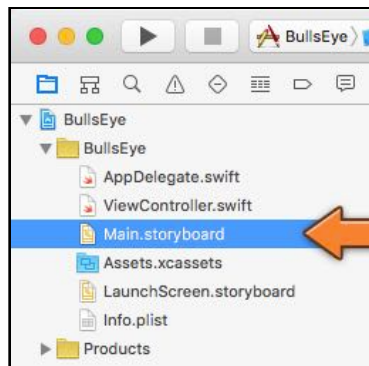
Adding the button

I'm sure you're as little impressed as I am with an app that just displays a dull white screen, so let's add a button to it.

The left-hand side of the Xcode window is named the **Navigator area**. The row of icons at the top determines which navigator is visible. Currently that is the **Project navigator**, which shows the list of 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 to your heart's content. We'll talk more about the different files that your project has later.

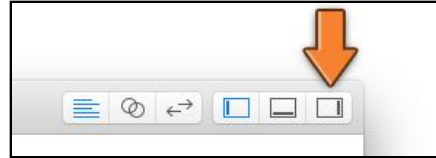
➤ In the **Project navigator**, find the item named **Main.storyboard** and click it once to select it:



The Project navigator lists the files in the project

Like a superhero changing his 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 into the app. (OK, bad analogy, but Interface Builder is a super tool in my opinion.)

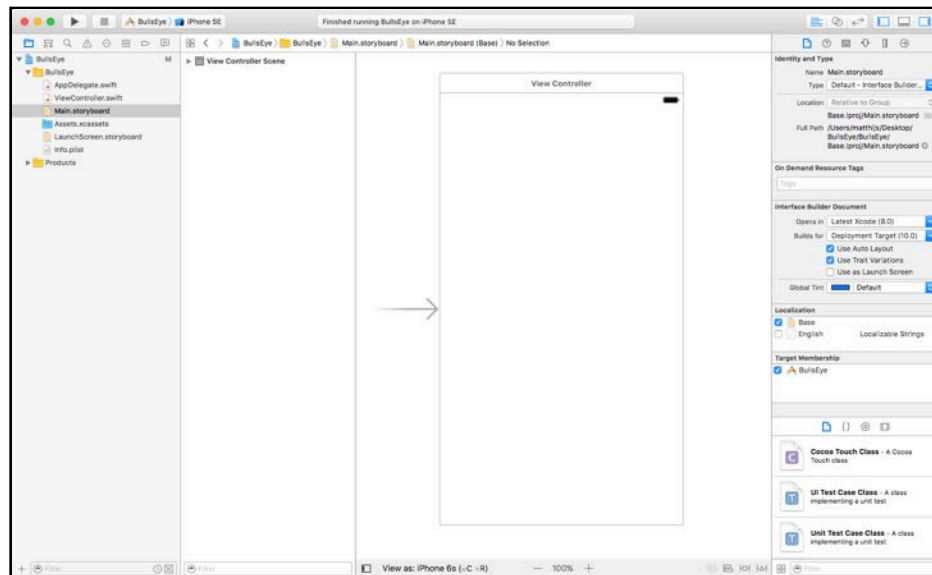
➤ If it's not already blue, click the **Hide or show utilities** button in Xcode's toolbar:



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

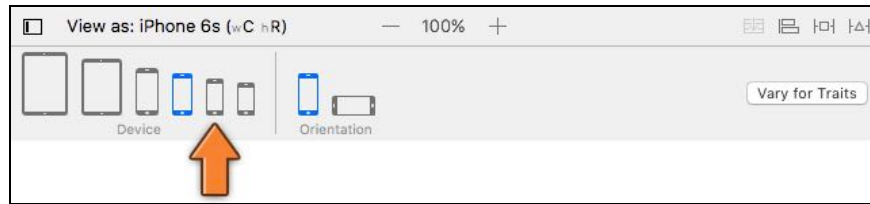
This is the *storyboard* for your app. The storyboard contains the designs for all of your app's screens, and shows how the app goes from one screen to another with big pointy arrows.

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 6s or iPhone 7. 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 6s, 7, and Plus.

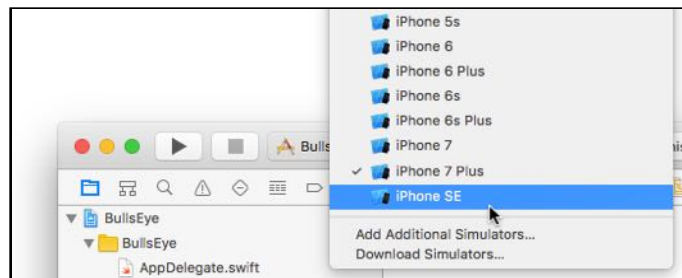
► At the bottom of the Interface Builder window, click **View as: iPhone 6s** to open up the following panel:



Choosing the device type

Select the **iPhone SE**, the second smallest iPhone. The scene's rectangle now becomes a bit smaller too. This corresponds to the screen size of the iPhone 5, iPhone 5s, and iPhone SE models.

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

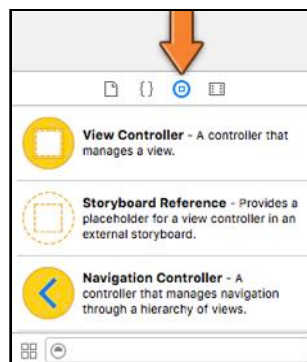


Switching the Simulator to iPhone SE

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

Back to the storyboard:

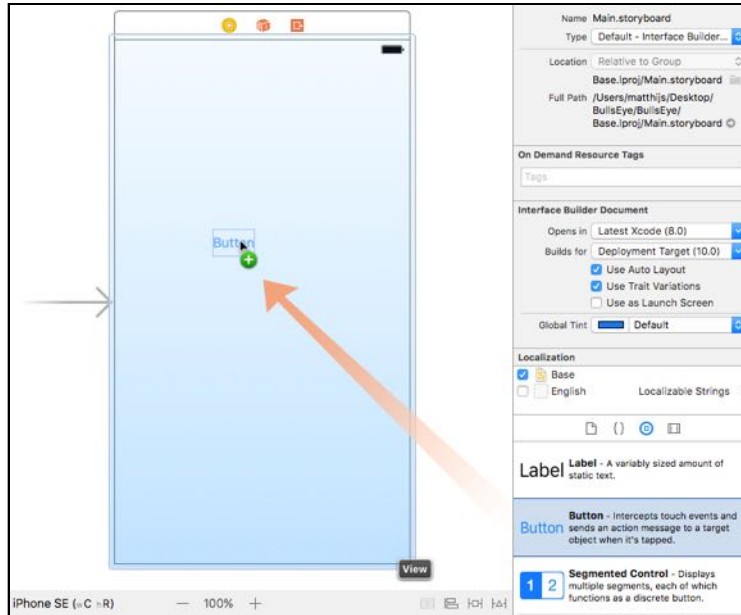
► At the bottom of the Utilities pane you will find the **Object Library** (make sure the third button, the one that looks like a circle, is selected):



The Object Library

Scroll through the items in the Object Library's list until you see **Button**.

► 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.

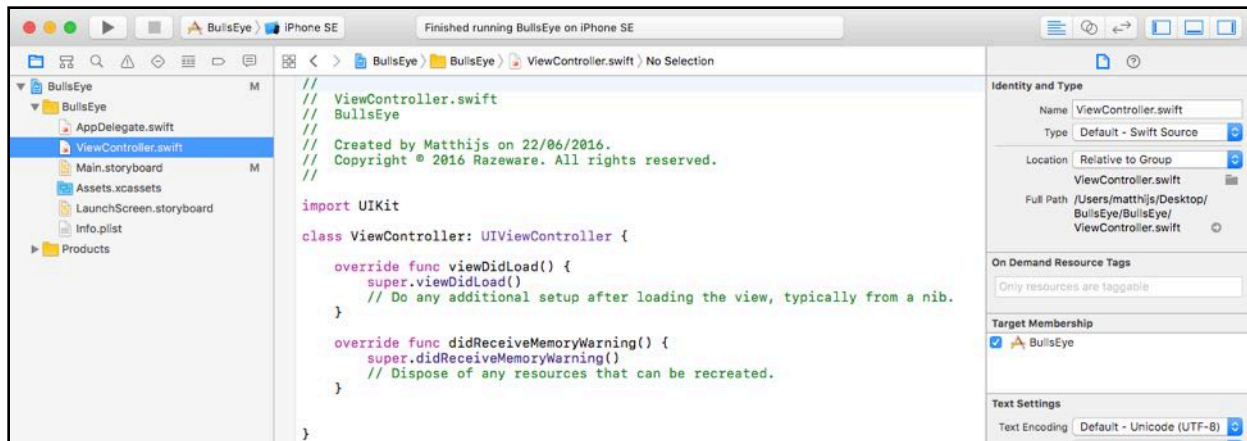
When you're done playing with Interface Builder, press the Run button from Xcode's toolbar. The app should now appear in the Simulator, complete with your "Hit Me!" button. However, when you tap the button it doesn't do anything yet. For that you'll have to write some Swift code!

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

► 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> <you>. 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 first 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.

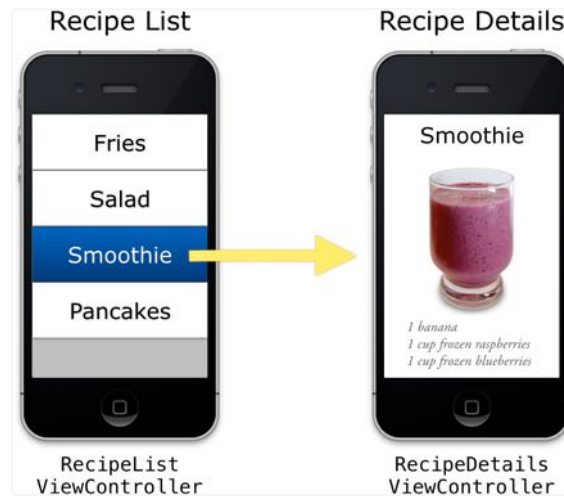
View controllers

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 from 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 its own 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 also 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 on top) 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.

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.

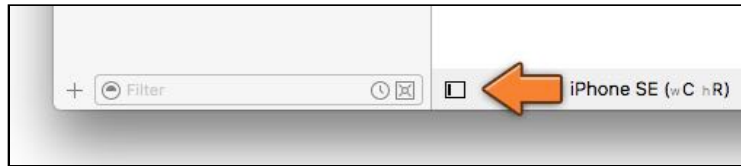
Making connections

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 to that action.

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

There should be a pane on the left, the **Outline pane**, 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.

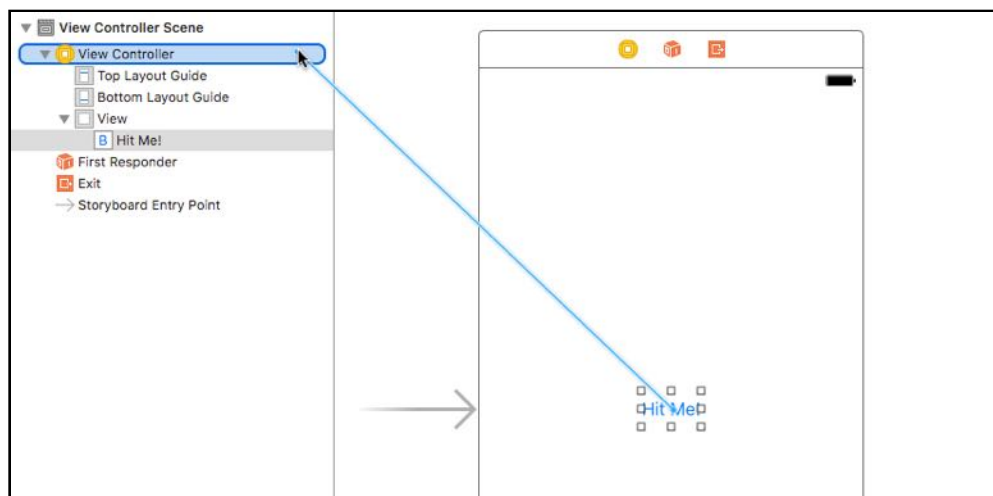


The button that shows the Outline pane

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

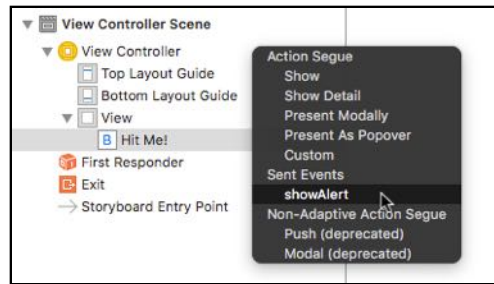
With the Hit Me button selected, hold down the **Ctrl** key, click on the button and drag up to the **View Controller** item in the Outline pane. You should see a blue line going from the button up to View Controller.

(Instead of holding down Ctrl, 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. This is the name of the action that you added earlier in the source code of **ViewController.swift**.



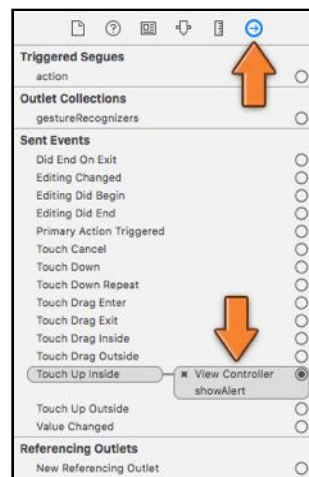
The popup menu with the showAlert action

► 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:

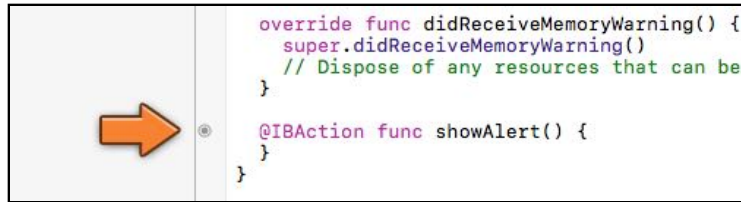


The inspector shows the connections from the button to any other objects

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

Acting 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). You need to give the app more instructions.

► In **ViewController.swift**, add the following lines to `showAlert`:

```
@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)  
}
```

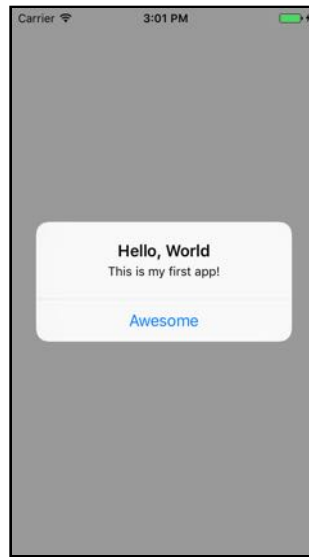
These new lines provide the actual functionality of this action.

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 for several checkpoints in this tutorial inside the Source Code folder that comes with this tutorial. 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 the app you've made thus far in the **01 - One Button App** 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 will totally confuse Xcode. It can be quite overwhelming to make sense out of the error messages. A small typo at the top of the source code 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

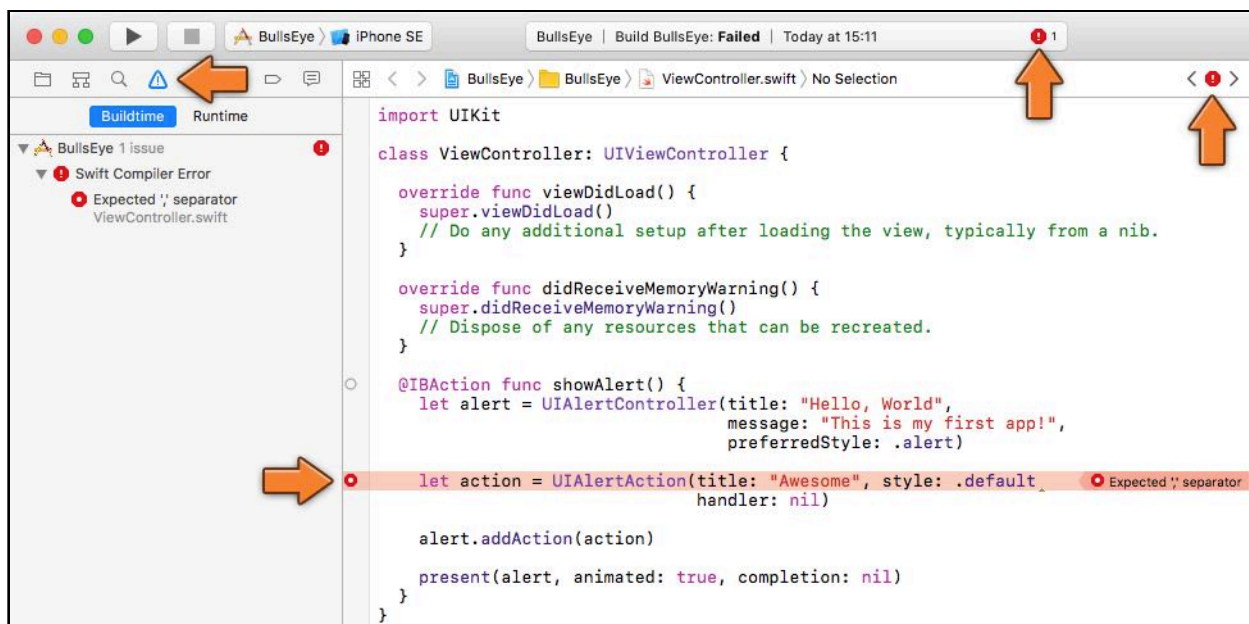
identifier" error.

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: If you move the text cursor over a closing bracket, Xcode will highlight the corresponding opening bracket.)

Tiny details like this 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.

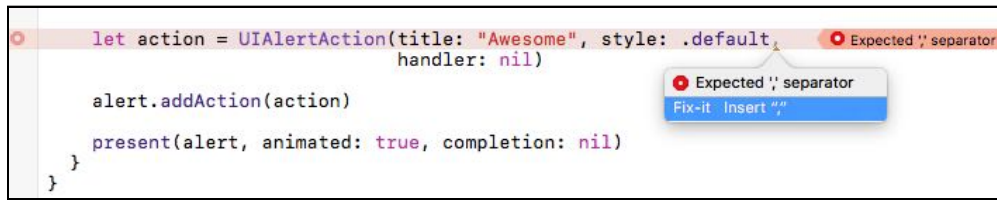


Xcode makes sure you can't miss errors

When Xcode detects an error it switches the pane on the left, where your project files used to be, to the **Issue navigator**. This list shows all the errors and warnings that Xcode has found. (You can go back to the project files with the small buttons at the top.)

Apparently, I forgot a comma somewhere.

Click on the error message and Xcode takes you to the line in the source code with the error. It even suggests what you need to do 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, but 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 are not allowed to run the app. 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 it won't be silly ones.

How does an app work?

It will 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.

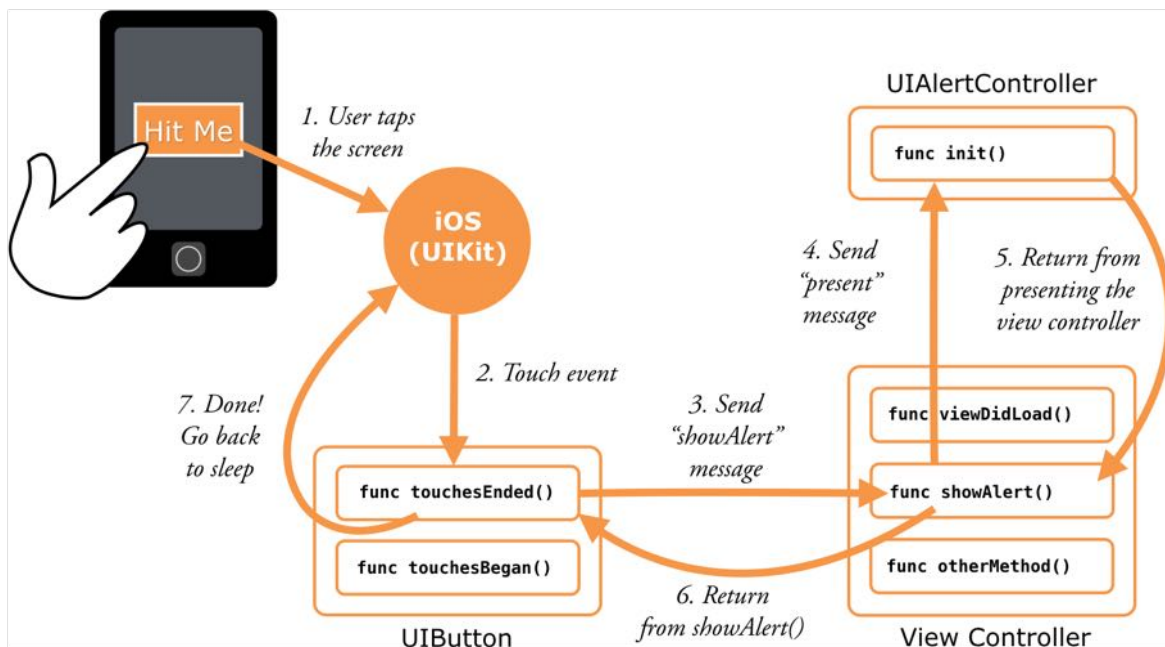
As strange as it may sound, an app spends most of its time doing... absolutely nothing. It just sits there waiting for something to happen. When the user taps the screen, the app springs to action for a few milliseconds and then it goes back to sleep again until the next event arrives.

Your part in this scheme is that you write the source code 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.

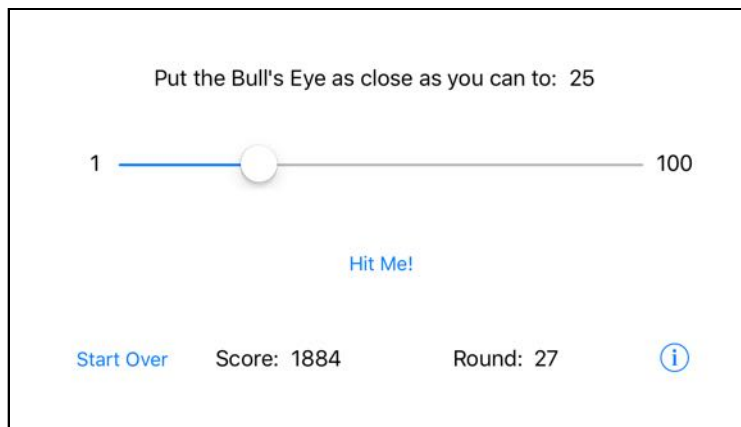
Working our way down the to-do list

Now that you have accomplished the first task of putting a button on the screen and making it show an alert, you'll simply go down the list and tick off the other items.

You don't really have to do this in any particular order, although some things make sense to do before others. For example, you cannot read the position of the slider if you don't have a slider yet.

So let's add the rest of the controls – the slider and the text labels – and turn this app into a real game!

When you're done, the app will look like this:



The game screen with standard UIKit controls

Hey, wait a minute... that doesn't look nearly as pretty as the game I promised you! The difference is that these are the standard UIKit controls. This is what they look like straight out of the box.

You've probably seen this look before because it is perfectly suitable for regular apps. But because the default look is a little boring for a game, you'll put some special sauce on top later in this lesson.

UIKit and other frameworks

iOS offers a lot of building blocks in the form of frameworks or "kits". The UIKit framework provides the user interface controls such as buttons, labels and navigation bars. It manages the view controllers and generally takes care of anything else that deals with your app's user interface. (That is what UI stands for: User Interface.)

If you had to write all that stuff from scratch, you'd be busy for a while. Instead, you can build your app on top of the system-provided frameworks and take advantage of all the work the Apple engineers have already done for

you.

Any object you see whose name starts with UI, such as UIButton, comes from UIKit. When you're writing iOS apps, UIKit is the framework you'll spend most of your time with but there are others as well.

Examples of other frameworks are Foundation, which provides many of the basic building blocks for building apps; Core Graphics for drawing basic shapes such as lines, gradients and images on the screen; AVFoundation for playing sound and video; and many others.

The complete set of frameworks for iOS is known collectively as Cocoa Touch.

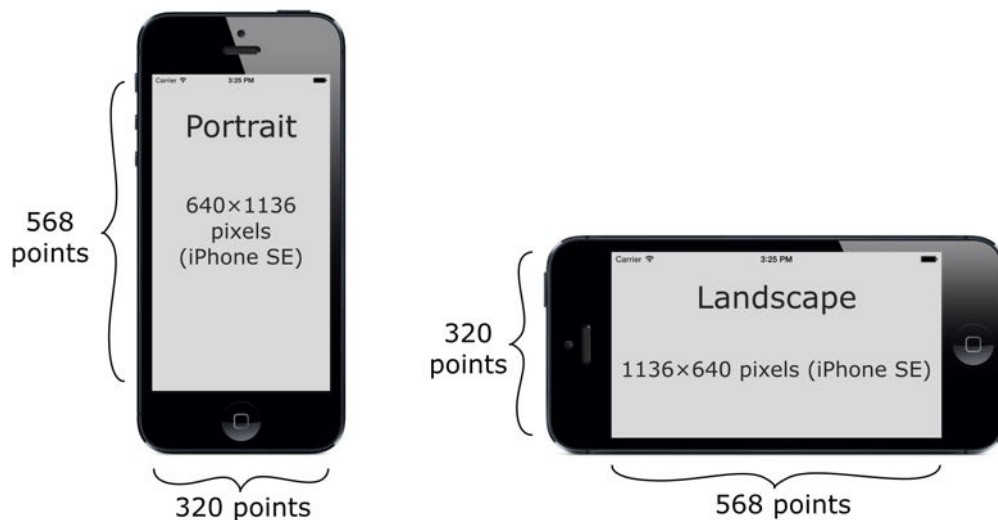
Portrait vs. landscape

Notice that the dimensions of the app have changed: the iPhone is tilted on its side and the screen is wider but less tall. This is called *landscape* orientation.

You've no doubt seen landscape apps before on the iPhone. It's a common display orientation for games but many other types of apps work in landscape mode too, usually in addition to the regular "upright" *portrait* orientation.

For instance, many people prefer to write emails with their device flipped over because the wider screen allows for a bigger keyboard and easier typing.

In portrait orientation, the iPhone SE screen consists of 320 points horizontally and 568 points vertically. For landscape these dimensions are switched.



Screen dimensions for portrait and landscape orientation

So what is a *point*?

On older devices – up to the iPhone 3GS and corresponding iPod touch models, as well as the first iPads – one point corresponds to one pixel. As a result, these low-

resolution devices don't look very sharp because of their big, chunky pixels.

I'm sure you know what a pixel is. In case you don't, it's the smallest element that a screen is made up of. The display of your iPhone is a big matrix of pixels that each can have their own color, just like a TV screen. Changing the color values of these pixels produces a visible image on the display. The more pixels, the better the image looks.

On the high-resolution Retina display of the iPhone 4 and later models, one point actually corresponds to two pixels horizontally and vertically, so four pixels in total. It packs a lot of pixels in a very small space, making for a much sharper display, which accounts for the popularity of Retina devices.

On the Plus it's even crazier: it has a 3x resolution with *nine* pixels for every point. Insane! You need to be eagle-eyed to make out the individual pixels on this fancy Retina HD display. It becomes almost impossible to make out where one pixel ends and the next one begins, that's how miniscule they are.

It's not only the number of pixels that differs between the various iPhone models. Over the years they have received different form factors, from the small 3.5-inch screen in the beginning all the way up to 5.5-inches on the iPhone 6s Plus and 7 Plus.

The form factor determines the width and height of the screen in points:

Device	Form factor	Screen dimension in points
iPhone 4S and older	3.5"	320×480
iPhone 5, 5c, 5s, SE	4"	320×568
iPhone 6, 6s, 7	4.7"	375×667
iPhone 6, 6s, 7 Plus	5.5"	414×736
iPad	7.9" and 9.7"	768×1024
iPad Pro	12.9"	1024×1366

In the early days of iOS, there was only one screen size. But those days of "one size fits all" are long gone. Now we have a variety of screen sizes to deal with.

Remember that UIKit works with points instead of pixels, so you only have to worry about the differences between the screen sizes measured in points. The actual number of pixels is only important for graphic designers because images are still measured in pixels.

Developers work in points, designers work in pixels.

The difference between points and pixels can be a little confusing, but if that is the only thing you're confused about right now then I'm doing a pretty good job. ;-)

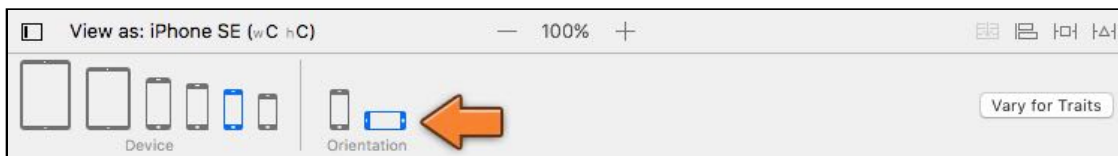
In this tutorial you'll initially work with just the iPhone SE screen size of 320×568 points – just to keep things simple. Later on in the tutorial you'll also make the game fit on the other types of iPhones.

Converting the app to landscape

To turn the app from portrait into landscape, you have to do two things:

1. Make the view in **Main.storyboard** landscape instead of portrait.
2. Change the **Supported Device Orientations** settings of the app.

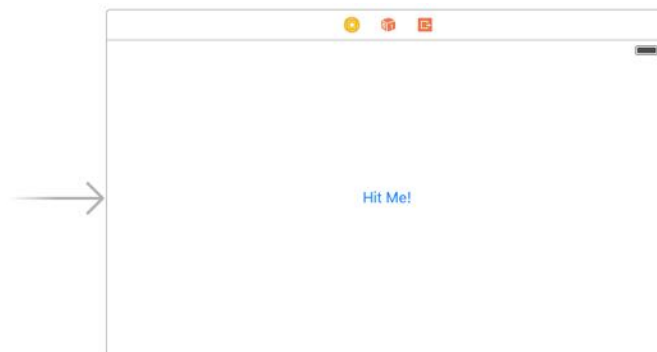
► Open **Main.storyboard** in the Interface Builder. In the **View as: iPhone SE** panel, change **Orientation** to landscape:



Changing the orientation in Interface Builder

This changes the dimensions of the view controller. It also puts the button in an awkward place.

► Move the button back to the center of the view because an untidy user interface just won't do in this day and age.



The view in landscape orientation

That takes care of the view layout.

► Run the app on the iPhone SE Simulator. The screen does not show up as landscape yet, and the button is no longer in the center either.

However, if you rotate the Simulator to landscape, then everything will look as it should.

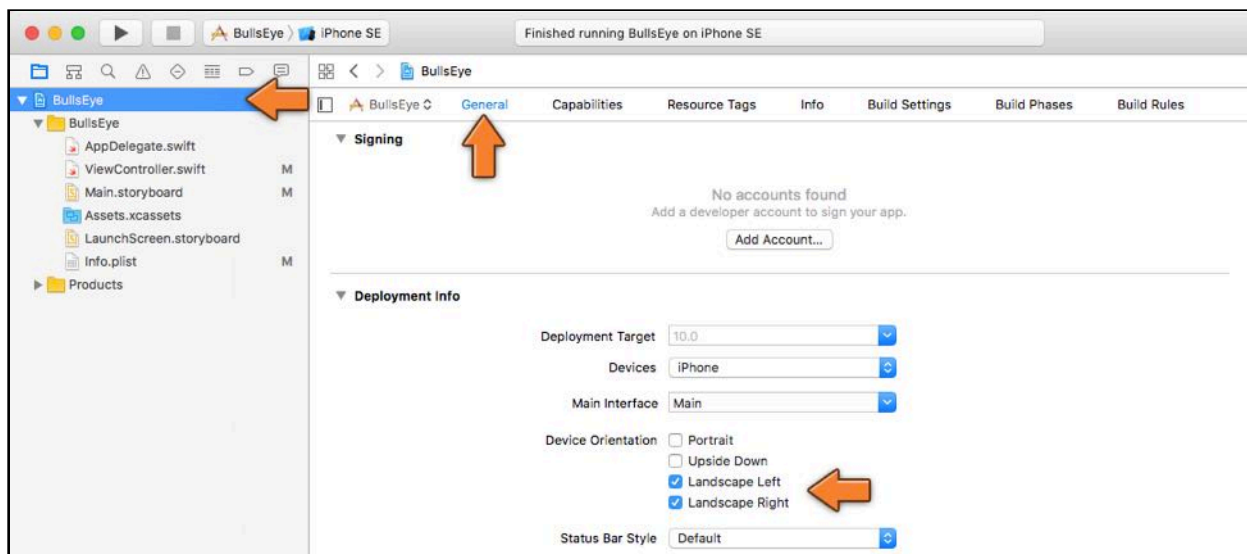
► Choose **Hardware** → **Rotate Left** or **Rotate Right** from the Simulator's menu bar at the top of the screen, or hold ⌘ and press the left or right arrow keys on your keyboard. This will flip the Simulator around.

Notice that in landscape orientation the app no longer shows the iPhone's status bar. This gives apps more room for their user interfaces.

You should do one more thing. There is a configuration option that tells iOS what orientations your app supports. New apps that you make from the template always support both portrait and landscape orientation.

► Click the blue **BullsEye** project icon at the top of the **Project navigator**. The editor pane of the Xcode window now reveals a bunch of settings for the project.

► Make sure that the **General** tab is selected:



The settings for the project

In the section **Deployment Info**, there is an option for **Device Orientation**.

► Check only the **Landscape Left** and **Landscape Right** options and leave the Portrait and Upside Down options unchecked.

Run the app again and it properly launches in the landscape orientation right from the beginning.

Objects, data and methods

Time for some programming theory. Yes, you cannot escape it.

Swift is a so-called “object-oriented” programming language, which means that most of the stuff you do involves objects of some kind. I already mentioned a few

times that an app consists of objects that send messages to each other.

When you write an iOS app, you'll be using objects that are provided for you by the system, such as the `UIButton` object from `UIKit`, and you'll be making objects of your own, such as view controllers.

So what exactly *is* an object? Think of an object as a building block of your program.

Programmers like to group related functionality into objects. *This* object takes care of parsing a file, *that* object knows how to draw an image on the screen, and that object over there can perform a difficult calculation.

Each object takes care of a specific part of the program. In a full-blown app you will have many different types of objects (tens or even hundreds).

Even your small starter app already contains several different objects. The one you have spent the most time with so far is `ViewController`. The Hit Me button is also an object, as is the alert popup. And the texts that you put on the alert – “Hello, World” and “This is my first app!” – are also objects.

The project also has an object named `AppDelegate`, even though you're going to ignore that for this lesson (but feel free to look inside its source file if you're curious). These object thingies are everywhere!

An object can have both *data* and *functionality*:

- An example of data is the Hit Me button that you added to the view controller earlier. When you dragged the button into the storyboard, it actually became part of the view controller's data. Data *contains* something. In this case, the view controller contains the button.
- An example of functionality is the `showAlert` action that you added to respond to taps on that button. Functionality *does* something.

The button itself also has data and functionality. Examples of button data are the text and color of its label, its position on the screen, its width and height, and so on. The button also has functionality: it can recognize that the user taps on it and will trigger an action in response.

The thing that provides functionality to an object is commonly called a *method*. Other programming languages may call this a “procedure” or “subroutine” or “function”. You will also see the term function used in Swift; a method is simply a function that belongs to an object.

Your `showAlert` action is an example of a method. You can tell it's a method because the line says `func` (short for “function”) and the name is followed by parentheses:

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

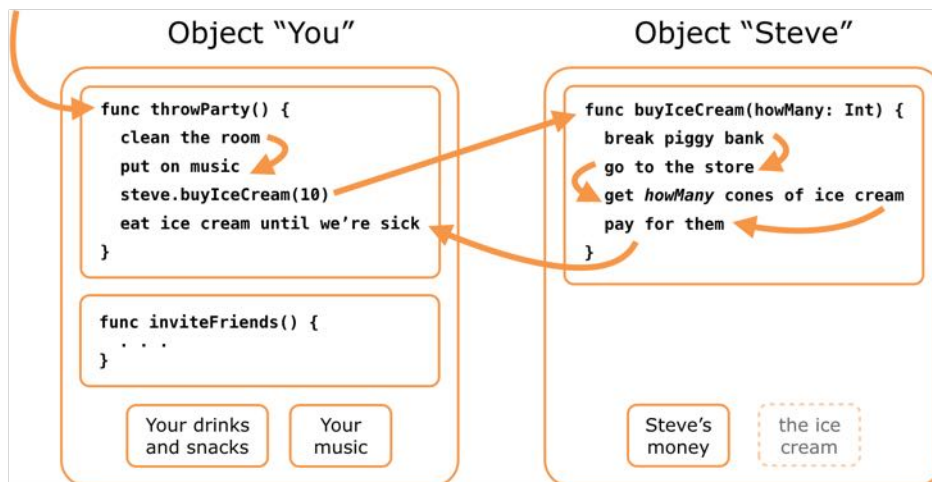


All method definitions start with the word func and have parentheses

If you look through the rest of **ViewController.swift** you'll see several other methods, such as `viewDidLoad()` and `didReceiveMemoryWarning()`.

These currently don't do much; the Xcode template placed them there for your convenience. These specific methods are often used by view controllers, so it's likely that you will need to fill them in at some point.

The concept of methods may still feel a little weird, so here's an example:



Every party needs ice cream!

You (or at least an object named "You") want to throw a party but you forgot to buy ice cream. Fortunately, you have invited the object named Steve who happens to live next door to a convenience store. It won't be much of a party without ice cream, so at some point during your party preparations you send object Steve a message asking him to bring some ice cream.

The computer now switches to object Steve and executes the commands from his `buyIceCream()` method, one by one, from top to bottom.

When his method is done, the computer returns to your `throwParty()` method and continues with that, so you and your friends can eat the ice cream that Steve brought back with him.

The Steve object also has data. Before he goes to the store he has money. At the store he exchanges this money data for other, much more important, data: ice cream! After making that transaction, he brings the ice cream data over to the party (if he eats it all along the way, your program has a bug).

“Sending a message” sounds more involved than it really is. It’s a good way to think conceptually of how objects communicate, but there really aren’t any pigeons or mailmen involved. The computer simply jumps from the `throwParty()` method to the `buyIceCream()` method and back again.

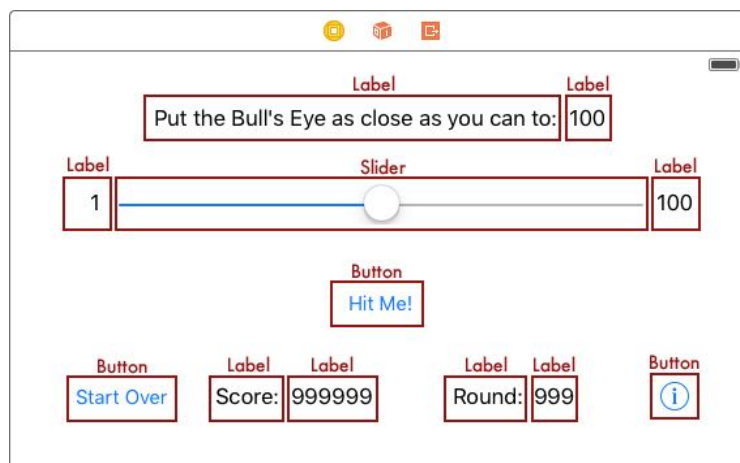
Often the terms “calling a method” or “invoking a method” are used instead. That means the exact same thing as sending a message: the computer jumps to the method you’re calling and returns to where it left off when that method is done.

The important thing to remember is that objects have methods (the steps involved in buying ice cream) and data (the actual ice cream and the money to buy it with).

Objects can look at each other’s data (to some extent anyway, just like Steve may not approve if you peek inside his wallet) and can ask other objects to perform their methods. That’s how you get your app to do things.

Adding the rest of the controls

Your app already has a button but you still need to add the rest of the UI controls, also known as “views”. Here is the screen again, this time annotated with the different types of views:

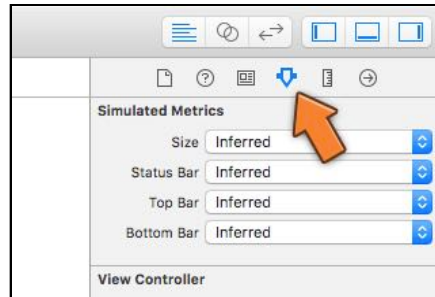


The different views in the game screen

As you can see, I put placeholder values into some of the labels (for example, “999999”). That makes it easier to see how the labels will fit on the screen when they’re actually used. The score label could potentially hold a large value, so you’d better make sure the label has room for it.

► Try to re-create this screen on your own by dragging the various controls from the Object Library. You'll need a few new Buttons, Labels, and a Slider. You can see in the screenshot above how big the items should (roughly) be. It's OK if you're a few points off.

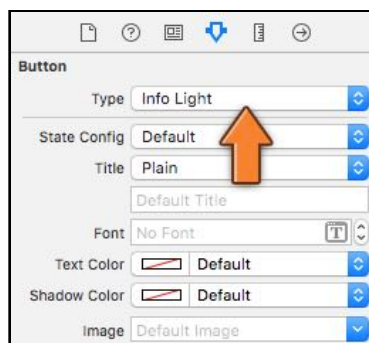
To tweak the settings of these views, you use the **Attributes inspector**. You can find this inspector in the pane on the right-hand side of the Xcode window:



The Attributes inspector

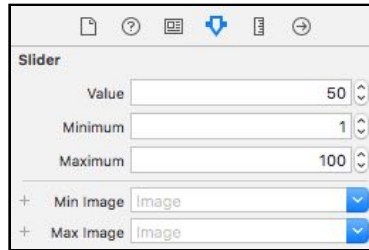
The inspector area shows various aspects of the item that is currently selected. The Attributes inspector, for example, lets you change the background color of a label or the size of the text on a button. You've already seen the Connections inspector that showed the button's actions. As you become more proficient with Interface Builder, you'll be using all of these inspector panes to configure your views.

► The **(i)** button is actually a regular Button, but its **Type** is set to **Info Light** in the Attributes inspector:



The button type lets you change the look of the button

► Also use the Attributes inspector to configure the **slider**. Its minimum value should be 1, its maximum 100, and its current value 50.



The slider attributes

When you're done, you should have 12 user interface elements in your scene: one slider, three buttons and a whole bunch of labels. Excellent.

► Run the app and play with it for a minute. The controls don't really do much yet (except for the button that should still pop up the alert), but you can at least drag the slider around.

You can tick a few more items off the to-do list, all without any programming! That is going to change really soon, because you will have to write Swift code to actually make the controls do anything.

The slider

The next item on your to-do list is: "Read the value of the slider after the user presses the Hit Me button."

If, in your messing around in Interface Builder, you did not accidentally disconnect the button from the showAlert action, you can modify the app to show the slider's value in the alert popup. (If you did disconnect the button, then you should hook it up again first.)

Remember how you added an action to the view controller in order to recognize when the user tapped the button? You can do the same thing for the slider. This new action will be performed whenever the user drags the slider's knob.

The steps for adding this action are largely the same as what you did before.

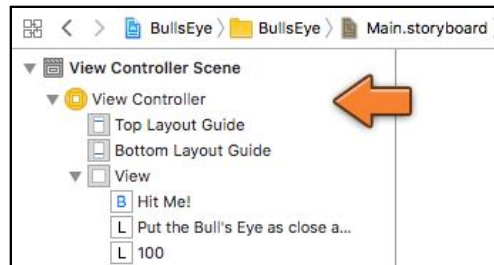
► First, go to **ViewController.swift** and add the following at the bottom, just before the final closing bracket:

```
@IBAction func sliderMoved(_ slider: UISlider) {  
    print("The value of the slider is now: \(slider.value)")  
}
```

► Second, go to the storyboard and Ctrl-drag from the slider to View Controller in the Outline pane. Let go of the mouse button and select **sliderMoved:** from the popup. Done!

Just to refresh your memory, the Outline pane sits on the left-hand side of the Interface Builder canvas. It shows the view hierarchy of the storyboard. Here you

can see that the View Controller contains a white view (simply named View) that spans the size of the scene, which in turn contains the sub-views you've added: the buttons and labels.



The Outline pane shows the view hierarchy of the storyboard

Remember, if the Outline pane is not visible, click the little icon at the bottom to reveal it:

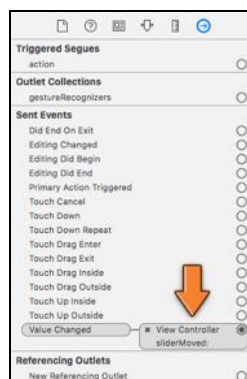


This button shows or hides the Outline pane

When you connect the slider, make sure to Ctrl-drag to View Controller (with the yellow icon), not View Controller Scene (gray icon). If you don't see the yellow icon, then click the arrow in front of View Controller Scene to expand it.

If all went well, the `sliderMoved:` action is now hooked up to the slider's Value Changed event. This means the `sliderMoved()` method will be called every time the user drags the slider to the left or right.

You can verify that the connection was made by selecting the slider and looking at the **Connections inspector**:

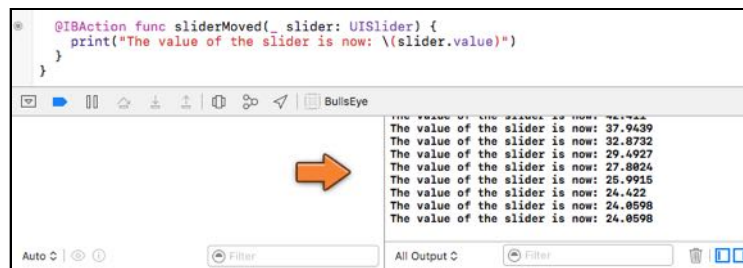


The slider is now hooked up to the view controller

Note: Did you notice that the `sliderMoved:` action has a colon in its name but `showAlert` does not? That's because the `sliderMoved()` method takes a single parameter, `slider`, while `showAlert()` does not have any parameters. If an action method has a parameter, Interface Builder adds a `:` to the name. You'll learn more about using parameters soon.

► Run the app and drag the slider.

As soon as you start dragging, the Xcode window opens a new pane at the bottom, the so-called **Debug area**, showing a list of messages:



Printing messages in the Debug area

If you swipe the slider all the way to the left, you should see the value go down to 1. All the way to the right, the value should be 100.

The `print()` function is a great help to show you what is going on in the app. Its entire purpose is to write a text message to the Debug area. Here, you used it to verify that you properly hooked up the action to the slider and that you can read its value as the slider is moved.

I often use `print()` to make sure my apps are doing the right thing before I add more functionality. Printing a message to the Debug area is quick and easy.

Note: You may see a bunch of other messages in the Debug area too. This is debug output from UIKit and the iOS Simulator. You can safely ignore these messages.

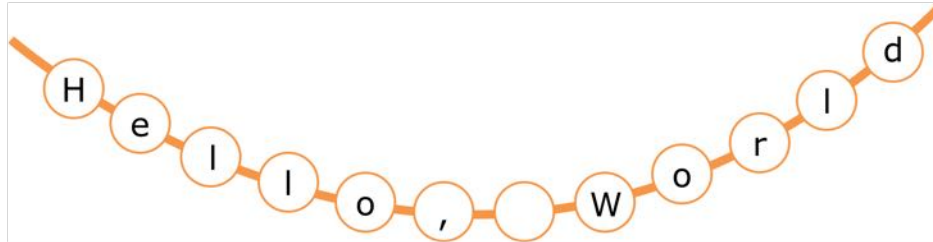
Strings

To put text in your app, you use something called a "string". The strings you have used so far are:

```
"Hello, World"
"This is my first app!"
"Awsome"
"The value of the slider is now: \(slider.value)"
```


The first three were used to make the `UIAlertController`; the last one you used with `print()` above.

Such a chunk of text is called a string because you can visualize the text as a sequence of characters, as if they were beads on a piece of string (sorry, it doesn't have anything to do with underwear):



A string of characters

Working with strings is something you need to do all the time when you're writing apps, so over the course of this tutorial series you'll get quite experienced with it.

To create a string, simply put the text in between double quotes. In other languages you can often use single quotes as well, but in Swift they must be double quotes. And they must be plain double quotes, not typographic "sixes and nines".

To summarize:

```
// This is the proper way to make a Swift string:
"I am a good string"

// These are wrong:
'I should have double quotes'
''Two single quotes do not make a double quote''
"My quotes are too fancy"
@"I am an Objective-C string"
```

Anything between the characters `\(...)` inside a string is special. The `print()` statement used the string, "The value of the slider is now: `\(slider.value)`". Think of the `\(...)` as a placeholder: "The value of the slider is now: X", where X will be replaced by the value of the slider.

Filling in the blanks is a very common way to build up strings in Swift.

Introducing variables

Printing information with `print()` to the Debug pane is very useful during development of the app, but it's absolutely useless to the user because they can't see any of this.

Let's improve this action method and make it show the value of the slider in the

alert popup. So how do you get the slider's value into `showAlert()`?

When you read the slider's value in `sliderMoved()`, that piece of data disappears when the action method ends. It would be handy if you could remember this value until the user taps the Hit Me button.

Fortunately, Swift has a building block exactly for this purpose: the *variable*.

► Open **ViewController.swift** and add the following at the top, directly below the line that says `class ViewController`:

```
var currentValue: Int = 0
```

You have now added a variable named `currentValue` to the view controller object.

The code should look like this (I left out the insides of the methods):

```
import UIKit

class ViewController: UIViewController {
    var currentValue: Int = 0

    override func viewDidLoad() {
        . . .
    }

    override func didReceiveMemoryWarning() {
        . . .
    }

    @IBAction func showAlert() {
        . . .
    }

    @IBAction func sliderMoved(_ slider: UISlider) {
        . . .
    }
}
```

It is customary to add the variables above the methods, and to indent everything with a tab or two to four spaces. Which one you use is largely a matter of personal preference. I like to use two spaces. (You can configure this in Xcode's preferences panel. From the menu bar choose Xcode → Preferences... → Text Editing and go to the Indentation tab.)

Remember when I said that a view controller, or any object really, could have both data and functionality? The `showAlert()` and `sliderMoved()` actions are examples of functionality, while the `currentValue` variable is part of its data.

A variable allows the app to remember things. Think of a variable as a temporary storage container for a single piece of data. There are containers of all sorts and sizes, just as data comes in all kinds of shapes and sizes.

You don't just put stuff in the container and then forget about it. You will often

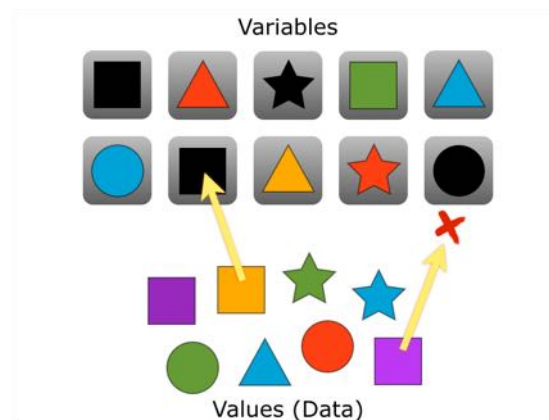
replace its contents with a new value. When the thing that your app needs to remember changes, you take the old value out of the box and put in the new value.

That's the whole point behind variables: they can *vary*. For example, you will update `currentValue` with the new position of the slider every time the slider is moved.

The size of the storage container and the sort of values the variable can remember are determined by its *data type*, or just *type*.

You specified the type `Int` for the `currentValue` variable, which means this container can hold whole numbers (also known as “integers”) between at least minus two billion and plus two billion. `Int` is one of the most common data types but there are many others and you can even make your own.

Variables are like children's toy blocks:



Variables are containers that hold values

The idea is to put the right shape in the right container. The container is the variable and its type determines what “shape” fits. The shapes are the possible values that you can put into the variables.

You can change the contents of each box later. For example, you can take out the blue square and put in a red square, as long as both are squares.

But you can't put a square in a round hole: the data type of the value and the data type of the variable have to match.

I said a variable is a *temporary* storage container. How long will it keep its contents? Unlike meat or vegetables, variables won't spoil if you keep them for too long – a variable will hold onto its value indefinitely, until you put a new value into that variable or until you destroy the container altogether.

Each variable has a certain lifetime (also known as its *scope*) that depends on

exactly where in your program you defined that variable. In this case, `currentValue` sticks around for just as long as its owner, `ViewController`, does. Their fates are intertwined.

The view controller, and thus `currentValue`, is there for the duration of the app. They don't get destroyed until the app quits. Soon you'll also see variables that live much shorter (so-called "local" variables).

Enough theory, let's make this variable work for us.

► Change the contents of the `sliderMoved()` method in **`ViewController.swift`** to the following:

```
@IBAction func sliderMoved(_ slider: UISlider) {  
    currentValue = lroundf(slider.value)  
}
```

You removed the `print()` statement and replaced it with this line:

```
currentValue = lroundf(slider.value)
```

What is going on here?

You've seen `slider.value` before, which is the slider's position at that moment. This is a value between 1 and 100, possibly with digits behind the decimal point. And `currentValue` is the name of the variable you have just created.

To put a new value into a variable, you simply do this:

```
variable = the new value
```

This is known as "assignment". You *assign* the new value to the variable. It puts the shape into the box. Here, you put the value that represents the slider's position into the `currentValue` variable.

Easy enough, but what is the `lroundf` thing? Recall that the slider's value can have numbers behind the decimal point. You've seen this with the `print()` output in the Debug pane as you moved the slider.

However, this game would be really hard if you made the player guess the position of the slider with an accuracy that goes behind the decimal point. That will be nearly impossible to get right!

It is more fair to use whole numbers only. That is why `currentValue` has data type `Int`, because that type stores *integers*, a fancy term for whole numbers.

You use the function `lroundf()` to round the decimal number to the nearest whole number and you then store that rounded-off number into `currentValue`.

Functions and methods

You've already seen that methods provide functionality, but *functions* are another way to put functionality into your apps (the name sort of gives it away). Functions and methods are how Swift programs combine multiple lines of code into single, cohesive units.

The difference between the two is that a function doesn't belong to an object while a method does. In other words, a method is exactly like a function – that's why you use the `func` keyword to define them – except that you need to have an object to use the method. But regular functions, or *free functions* as they are sometimes called, can be used anywhere.

Swift provides your programs with a large library of useful functions. The function `lroundf()` is one of them and you'll be using a few others during this lesson as well. `print()` is also a function, by the way. You can tell because the function name is always followed by parentheses that possibly contain one or more parameters.

➤ Now change the `showAlert()` method to the following:

```
@IBAction func showAlert() {  
    let message = "The value of the slider is: \(currentValue)"  
  
    let alert = UIAlertController(title: "Hello, World",  
                                message: message,           // changed  
                                preferredStyle: .alert)  
  
    let action = UIAlertAction(title: "OK",                 // changed  
                               style: .default, handler: nil)  
  
    alert.addAction(action)  
  
    present(alert, animated: true, completion: nil)  
}
```

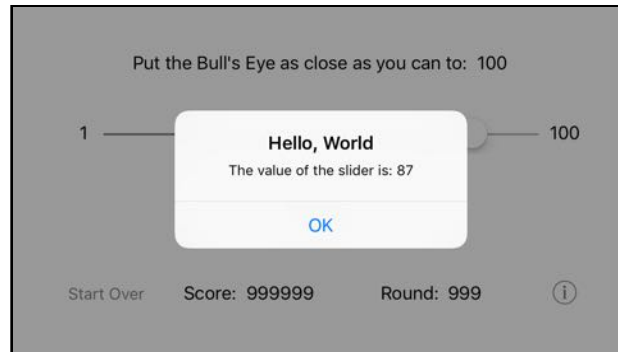
The line with `let message =` is new. Also note the other two small changes.

As before, you create and show a `UIAlertController`, except this time its message says: "The value of the slider is: X", where X is replaced by the contents of the `currentValue` variable (a whole number between 1 and 100).

Suppose `currentValue` is 34, which means the slider is about one-third to the left. The new code above will convert the string "The value of the slider is: \(currentValue)" into "The value of the slider is: 34" and puts that into a new object named `message`.

The old `print()` did something similar, except that it printed the result to the Debug pane. Here, however, you do not wish to print the result but show it in the alert popup. That is why you tell the `UIAlertController` that it should now use this new string as the message to display.

► Run the app, drag the slider, and press the button. Now the alert should show the actual value of the slider.



The alert shows the value of the slider

Cool. You have used a variable, `currentValue`, to remember a particular piece of data, the rounded-off position of the slider, so that it can be used elsewhere in the app, in this case in the alert's message text.

If you tap the button again without moving the slider, the alert will still show the same value. The variable keeps its value until you put a new one into it.

Your first bug

There is a small problem with the app, though. Maybe you've noticed it already. Here is how to reproduce the problem:

► Press the Stop button in Xcode to completely terminate the app, then press Run again. Without moving the slider, immediately press the Hit Me button.

The alert now says: "The value of the slider is: 0". But the slider's knob is obviously at the center, so you would expect the value to be 50. You've discovered a bug!

Exercise: Think of a reason why the value would be 0 in this particular situation (start the app, don't move the slider, press the button). ■

Answer: The clue here is that this only happens when you don't move the slider. Of course, without moving the slider the `sliderMoved()` message is never sent and you never put the slider's value into the `currentValue` variable.

The default value for the `currentValue` variable is 0, and that is what you are seeing here.

► To fix this bug, change the declaration of `currentValue` to:

```
var currentValue: Int = 50
```

Now the starting value of `currentValue` is 50, which should be the same value as the slider's initial position.

► Run the app again and verify that the bug is solved.

You can find the project files for the app up to this point under **02 - Slider and Variables** in the tutorial's Source Code folder.

Enough playing around... let's make a game!

You've built the user interface and you know how to find the position of the slider. That already knocks quite a few items off the to-do list.

The big remaining items are generating the random value for the target and calculating how well the player did. But first, there's still an improvement to make on the slider.

Outlets

You managed to store the value of the slider into a variable and show it on the alert. That's good but you can still improve on it a little.

What if you decide to set the initial value of the slider in the storyboard to something other than 50, say 1 or 100? Then `currentValue` would be wrong again because the app always assumes it will be 50 at the start. You'd have to remember to also fix the code to give `currentValue` a new initial value.

Take it from me, those kinds of small things are hard to remember, especially when the project becomes bigger and you have dozens of view controllers to worry about, or when you haven't looked at the code for weeks.

Therefore, to fix this issue once and for all, you're going to do some work inside the `viewDidLoad()` method in **ViewController.swift**. That method currently looks like this:

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

When you created this project based on Xcode's template, Xcode already put the `viewDidLoad()` method into the source code. You will now add some code to it.

The `viewDidLoad()` message is sent by UIKit as soon as the view controller loads its user interface from the storyboard file. At this point, the view controller isn't visible yet, so this is a good place to set instance variables to their proper initial values.

► Change `viewDidLoad()` to the following:

```
override func viewDidLoad() {  
    super.viewDidLoad()
```

```
    currentValue = lroundf(slider.value)
}
```

The idea is that you take whatever value is set on the slider in the storyboard (whether it is 50, 1, 100, or anything else) and use that as the initial contents of `currentValue`.

Recall that you need to round off the number, because `currentValue` is an `Int` and integers cannot take digits behind the decimal point.

Unfortunately, Xcode complains about these changes when you press Run. Try it for yourself.

► Try to run the app.

Xcode says “Build Failed”, followed by something like: “Error: Use of unresolved identifier ‘slider’”.

That happens because `viewDidLoad()` does not know anything named `slider`.

Then why did this work earlier, in `sliderMoved()`? Let’s take a look at that method again:

```
@IBAction func sliderMoved(_ slider: UISlider) {
    currentValue = lroundf(slider.value)
}
```

Here you do the exact same thing: you round off `slider.value` and put it into `currentValue`. So why does it work here but not in `viewDidLoad()`?

The difference is that `slider` is a so-called *parameter* of the `sliderMoved()` method. Parameters are the things inside the parentheses following a method’s name. In this case there’s a single parameter named `slider`, which refers to the `UISlider` object that sent this action message.

Action methods can have a parameter that refers to the UI control that triggered the method. That is convenient when you wish to use that object in the method, just as you did here (the object in question being the `UISlider`).

When the user moves the slider, the `UISlider` object basically says, “Hey view controller, I’m a slider object and I just got moved. By the way, here’s my phone number so you can get in touch with me.”

The `slider` parameter contains this “phone number” but it is only valid for the duration of this particular method.

In other words, `slider` is *local*; you cannot use it anywhere else.



Locals

When I first introduced variables, I mentioned that each variable has a certain lifetime, known as its *scope*. The scope of a variable depends on where in your program you defined that variable.

There are three possible scope levels in Swift:

1. **Global scope.** These objects exist for the duration of the app and are accessible from anywhere.
2. **Instance scope.** This is for variables such as `currentValue`. These objects are alive for as long as the object that owns them stays alive.
3. **Local scope.** Objects with a local scope, such as the `slider` parameter of `sliderMoved()`, only exist for the duration of that method. As soon as the execution of the program leaves this method, the local objects are no longer accessible.

Let's look at the top part of `showAlert()`:

```
@IBAction func showAlert() {  
    let message = "The value of the slider is: \(currentValue)"  
  
    let alert = UIAlertController(title: "Hello, World",  
                                message: message, preferredStyle: .alert)  
  
    let action = UIAlertAction(title: "OK", style: .default,  
                              handler: nil)  
    . . .  
}
```

Because the `message`, `alert`, and `action` objects are created inside the method, they are locals. They only come into existence when the `showAlert()` action is performed and cease to exist when the action is done.

As soon as the `showAlert()` method completes, i.e. when there are no more statements for it to execute, the computer destroys the `message`, `alert`, and `action` objects. Their storage space is no longer needed.

The `currentValue` variable, however, lives on forever.. or at least for as long as the `ViewController` does (which is until the user terminates the app). This type of variable is named an *instance variable*, because its scope is the same as the scope of the object instance it belongs to.

In other words, you use instance variables if you want to keep a certain value around, from one action event to the next.



The solution is to store a reference to the slider as a new instance variable, just like you did for `currentValue`. Except that this time, the data type of the variable is not `Int`, but `UISlider`. And you're not using a regular instance variable but a special form called an *outlet*.

► Add the following line to **ViewController.swift**:

```
@IBOutlet weak var slider: UISlider!
```

It doesn't really matter where this line goes, just as long as it is somewhere inside the brackets for `class ViewController`. I usually put outlets with the other instance variables.

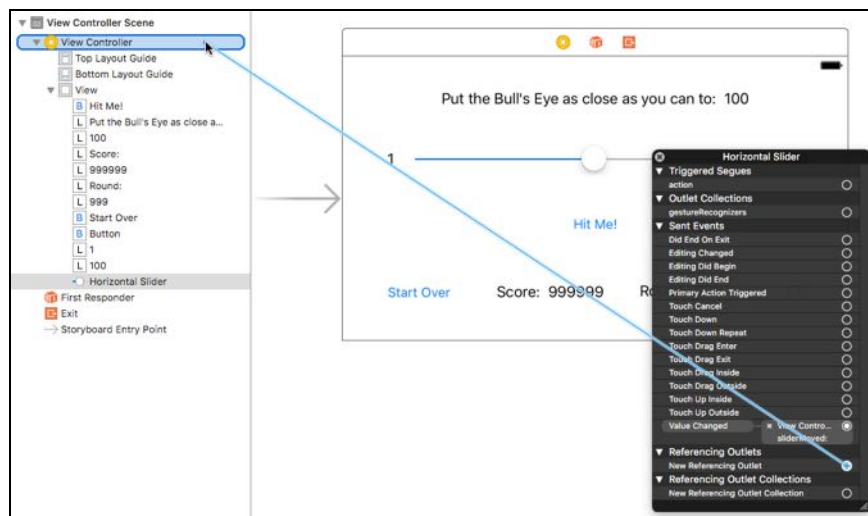
This line tells Interface Builder that you now have a variable named `slider` that can be connected to a `UISlider` object. Just as Interface Builder likes to call methods "actions", it calls these variables outlets. Interface Builder doesn't see any of your other variables, only the ones marked with `@IBOutlet`.

Don't worry about `weak` or the exclamation point for now. Why these are necessary will be explained in the next tutorials. For now just remember that a variable for an outlet needs to be declared as `@IBOutlet weak var` and has an exclamation point at the end. (Sometimes you'll see a question mark instead; all this hocus pocus will be explained in due time.)

► Open the storyboard. Hold **Ctrl** and click on the **slider**. Don't drag anywhere just yet: let go of the mouse button and a menu pops up that shows all the connections for this slider. (Instead of Ctrl-clicking you can also right-click once.)

This popup menu works exactly the same as the Connections inspector. I just wanted to show you that it exists as an alternative.

► Click on the open circle next to **New Referencing Outlet** and drag to **View Controller**:



Connecting the slider to the outlet

► In the popup that appears, select **slider**.

This is the outlet that you just added to the object. You have successfully connected the slider object from the storyboard to the view controller's `slider` outlet.

Now that you have done all this setup work, you can refer to the slider object from anywhere inside the view controller using the `slider` variable.

With these changes in place, it no longer matters what you choose for the initial value of the slider in Interface Builder. When the app starts, `currentValue` will always correspond to that setting.

► Run the app and immediately press the button. It correctly says: "The value of the slider is: 50". Stop the app, go into Interface Builder and change the initial value of the slider to something else, say, 25. Run the app again and press the button. The alert should read 25 now.

Put the slider's starting position back to 50 when you're done playing.

Exercise: Give `currentValue` an initial value of 0 again. Its initial value is not longer important – it will be overwritten in `viewDidLoad()` anyway – but Swift demands that all variables always have some value and 0 is as good as any. ■



Comments

You've seen green lines that begin with `//` a few times now. These are comments. You can write any text you want after the `//` symbol as the compiler will ignore such lines completely.

```
// I am a comment! You can type anything here.
```

Anything between the `/*` and `*/` markers is considered a comment as well. The difference between `//` and `/* */` is that the former only works on a single line, while the latter can span multiple lines.

```
/*  
    I am a comment as well!  
    I can span multiple lines.  
*/
```

The `/* */` comments are often used to temporarily disable whole sections of the source code, usually when you're trying to hunt down a pesky bug, a practice known as "commenting out".

The best use for comment lines is to explain how your code works. Well-written source code is self-explanatory but sometimes additional clarification is useful.

Explain to who? To yourself, mostly.

Unless you have the memory of an elephant, you'll probably have forgotten exactly how your code works when you look at it six months later. Use comments to jog your memory.

As you have seen, Xcode automatically adds a comment block with copyright information at the top of the source code files. Personally, I don't care much for these copyright blocks. Feel free to remove those lines if you don't like them either.



Generating the random number

You still have quite a ways to go before the game is playable, so let's get on with the next item on the list: generating a random number and displaying it on the screen.

Random numbers come up a lot when you're making games because often games need to have some element of unpredictability. You can't really get a computer to generate numbers that are truly random and unpredictable, but you can employ a so-called *pseudo-random generator* to spit out numbers that at least appear that way. You'll use my favorite one, the `arc4random_uniform()` function.

A good place to generate this random number is when the game starts.

► Add the following line to `viewDidLoad()` in **ViewController.swift**:

```
targetValue = 1 + Int(arc4random_uniform(100))
```

The complete `viewDidLoad()` should now look like this:

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    currentValue = lroundf(slider.value)  
    targetValue = 1 + Int(arc4random_uniform(100))  
}
```

What did you do here? First, you're using a new variable, `targetValue`. You haven't actually defined this variable yet, so you'll have to do that in a minute.

You are also calling the function `arc4random_uniform()` to deliver an arbitrary integer (whole number) between 0 and 100.

Actually, the highest number you will get is 99 because `arc4random_uniform()` treats the upper limit as exclusive. It only goes up-to 100, not up-to-and-including. To get a number that is truly in the range 1 - 100, you need to add 1 to the result of `arc4random_uniform()`.

You still have to add the variable `targetValue` to the view controller, otherwise Xcode will complain that it doesn't know anything about this variable.

If you don't tell the compiler what kind of variable `targetValue` is, then it doesn't know how much storage space to allocate for it, nor can it check if you're using the variable properly everywhere.

► Add the new variable at the top of **ViewController.swift**, with the other variables:

```
var targetValue: Int = 0
```

Variables in Swift must always have a value, so here you give it the initial value 0. That 0 is never used in the game; it will always be overwritten by the random value in `viewDidLoad()`.

Note: Up until you made this latest change, Xcode may have pointed out that it did not know the `targetValue` variable. That error message should now have disappeared.

Xcode tries to be helpful and it analyzes the program for mistakes as you're typing. Sometimes you may see temporary warnings and error messages that will go away when you complete the changes that you're making.

Don't be too intimidated by these messages; they are only short-lived while the code is in a state of flux.

I hope the reason is clear why you made `targetValue` an instance variable.

You want to calculate the random number in one place – in `viewDidLoad()` – and then remember it until the user taps the button, in `showAlert()`.

► Change `showAlert()` to the following:

```
@IBAction func showAlert() {  
    let message = "The value of the slider is: \(currentValue)" +  
                  "\n\nThe target value is: \(targetValue)"  
  
    let alert = . . .  
}
```

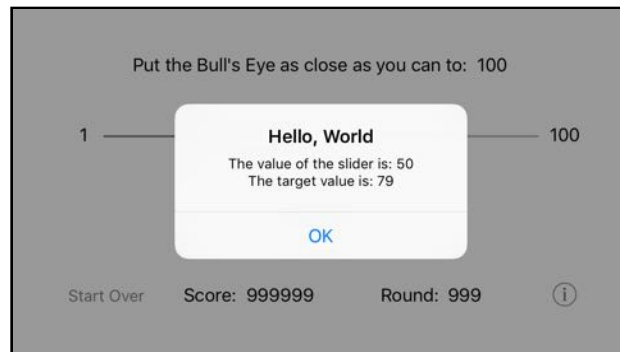
Tip: Whenever you see `. . .` in a source code listing I mean that as shorthand for: this part didn't change. (Don't go replacing what was there with an actual ellipsis!)

You've simply added the random number, which is now stored in `targetValue`, to the message string. This should look familiar to you by now: the `\(targetValue)` placeholder is replaced by the actual random number.

The `\n` character sequence is new. It means that you want to insert a special "new line" character at that point, which will break up the text into two lines so the

message is a little easier to read.

► Run the app and try it out!



The alert shows the target value on a new line

Note: Earlier you've used the `+` operator to add two numbers together (just like how it works in math) but here you're also using `+` to glue different bits of text into one big string.

Swift allows the use of the same operator symbol for different tasks, depending on the data types involved. If you have two integers, `+` adds them up. But with two strings, `+` concatenates them into a larger string.

Programming languages often use the same symbols for different purposes, depending on the context. (There are only so many symbols to go around.)

Adding rounds to the game

If you press the Hit Me button a few times, you'll notice that the random number never changes. I'm afraid the game won't be much fun that way.

This happens because you generate the random number in `viewDidLoad()` and never again afterwards. The `viewDidLoad()` method is only called once when the view controller is created during app startup.

The item on the to-do list actually said: "Generate a random number *at the start of each round*". Let's talk about what a round means in terms of this game.

When the game starts, the player has a score of 0 and the round number is 1. You set the slider halfway (to value 50) and calculate a random number. Then you wait for the player to press the Hit Me button. As soon as she does, the round ends.

You calculate the points for this round and add them to the total score. Then you increment the round number and start the next round. You reset the slider to the halfway position again and calculate a new random number. Lather, rinse, repeat.

Whenever you find yourself thinking something along the lines of, "At this point in

the app we have to do such and so,” then it makes sense to create a new method for it. This method will nicely capture that functionality in a unit of its own.

➤ With that in mind, add the following new method to **ViewController.swift**.

```
func startNewRound() {
    targetValue = 1 + Int(arc4random_uniform(100))
    currentValue = 50
    slider.value = Float(currentValue)
}
```

It doesn't really matter where you put it, as long as it is inside the brackets of class ViewController, so that the compiler knows it belongs to the ViewController object.

It's not very different from what you did before, except that you moved the logic for setting up a new round into its own method, `startNewRound()`. The advantage of doing this is that you can use this logic from more than one place.

First you'll call this new method from `viewDidLoad()` to set up everything for the very first round. Recall that `viewDidLoad()` happens just once when the app starts up, so this is a great place to begin the first round.

➤ Change `viewDidLoad()` to:

```
override func viewDidLoad() {
    super.viewDidLoad()
    startNewRound()
}
```

Note that you've removed the existing statements from `viewDidLoad()` and replaced them with just the call to `startNewRound()`.

You will also call `startNewRound()` after the player pressed the Hit Me button, from within `showAlert()`.

➤ Make the following change to `showAlert()`:

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

    startNewRound()
}
```

The call to `startNewRound()` goes at the very end, right after `present(alert, ...)`.

Until now, the methods from the view controller have been invoked for you by UIKit when something happened: `viewDidLoad()` is performed when the app loads, `showAlert()` is performed when the player taps the button, `sliderMoved()` when the player drags the slider, and so on. This is the event-driven model we talked about earlier.

It is also possible to call methods by hand, which is what you're doing here. You are

sending a message from one method in the object to another method in that same object.

In this case, the view controller sends the `startNewRound()` message to itself in order to set up the new round. The iPhone will then go to that method and execute its statements one-by-one. When there are no more statements in the method, it returns to the calling method and continues with that – either `viewDidLoad()` if this is the first time or `showAlert()` for every round after.

Sometimes you may see method calls written like this:

```
self.startNewRound()
```

That does the exact same thing as just `startNewRound()` without “self.” in front. Recall how I just said that the view controller sends the message to itself? Well, that’s exactly what “self” means.

To call a method on an object you’d normally write:

```
receiver.methodName(parameters)
```

The receiver is the object you’re sending the message to. If you’re sending the message to yourself, then the receiver is `self`. But because sending messages to `self` is very common, you can also leave off this special keyword.

To be fair, this isn’t exactly the first time you’ve called methods. `addAction()` is a method on `UIAlertController` and `present()` is a method that all view controllers have, including yours.

When you write Swift programs, a lot of what you do is calling methods on objects, because that is how the objects in your app communicate.

I hope you can see the advantage of putting the “new round” logic into its own method. If you didn’t, the code for `viewDidLoad()` and `showAlert()` would look like this:

```
override func viewDidLoad() {
    super.viewDidLoad()

    targetValue = 1 + Int(arc4random_uniform(100))
    currentValue = 50
    slider.value = Float(currentValue)
}

@IBAction func showAlert() {
    . . .

    targetValue = 1 + Int(arc4random_uniform(100))
    currentValue = 50
    slider.value = Float(currentValue)
}
```


Can you see what is going on here? The same functionality is duplicated in two places. Sure, it is only three lines, but often the code you would have to duplicate will be much larger.

And what if you decide to make a change to this logic (as you will shortly)? Then you will have to make this change in two places as well.

You might be able to remember to do so if you recently wrote this code and it is still fresh in memory, but if you have to make that change a few weeks down the road, chances are that you'll only update it in one place and forget about the other.

Code duplication is a big source of bugs, so if you need to do the same thing in two different places consider making a new method for it.

The name of the method also helps to make it clear what it is supposed to be doing. Can you tell at a glance what the following does?

```
targetValue = 1 + Int(arc4random_uniform(100))
currentValue = 50
slider.value = Float(currentValue)
```

You probably have to reason your way through it: "It is calculating a new random number and then resets the position of the slider, so I guess it must be the start of a new round."

Some programmers will use a comment to document what is going on, but in my opinion the following is much clearer:

```
startNewRound()
```

This line practically spells out for you what it will do. And if you want to know the specifics of what goes on in a new round, you can always look up the `startNewRound()` method and look inside.

Well-written source code speaks for itself. I hope I have convinced you of the value of making new methods!

► Run the app and verify that it calculates a new random number between 1 and 100 after each tap on the button.

You should also have noticed that after each round the slider resets to the halfway position. That happens because `startNewRound()` sets `currentValue` to 50 and then tells the slider to go to that position. That is the opposite of what you did before (you used to read the slider's position and put it into `currentValue`), but I thought it would work better in the game if you start from the same position in each round.

Exercise: Just for fun, modify the code so that the slider does not reset to the halfway position at the start of a new round. ■

By the way, you may have been wondering what `Float(...)` and `Int(...)` do in these lines:

```
targetValue = 1 + Int(arc4random_uniform(100))  
slider.value = Float(currentValue)
```

Swift is a so-called *strongly typed* language, meaning that it is really picky about the shapes that you can put into the boxes. For example, if a variable is an `Int` you cannot put a `Float` into it, and vice versa.

The value of a `UISlider` happens to be a `Float`, which is a number that can have digits after the decimal point – you’ve seen this when you printed out the value of the slider – but `currentValue` is an `Int`. So this won’t work:

```
slider.value = currentValue
```

The compiler considers this an error. Some programming languages are happy to convert the `Int` into a `Float` for you, but Swift wants you to be explicit about such conversions.

When you say `Float(currentValue)`, the compiler takes the integer number that’s stored in `currentValue`’s box and puts it into a new `Float` value that it can give to the `UISlider`.

Something similar happens with `arc4random_uniform()`, where the random number gets converted to an `Int` first before it can be placed into `targetValue`.

Because Swift is stricter about this sort of thing than most other programming languages, it is often a source of confusion for newcomers to the language. Unfortunately, Swift’s error messages aren’t always very clear about what part of the code is wrong or why.

Just remember, if you get an error message saying, “cannot assign value of type ‘something’ to type ‘something else’” then you’re trying to mix incompatible data types. The solution is to explicitly convert one type to the other, as you’ve been doing here.

Putting the target value into the label

Great, you figured out how to calculate the random number and how to store it in an instance variable, `targetValue`, so that you can access it later.

Now you are going to show that target number on the screen. Without it, the player won’t know what to aim for and that would make the game impossible to win...

When you made the storyboard, you already added a label for the target value (top-right corner). The trick is to put the value from the `targetValue` variable into this label. To do that, you need to accomplish two things:

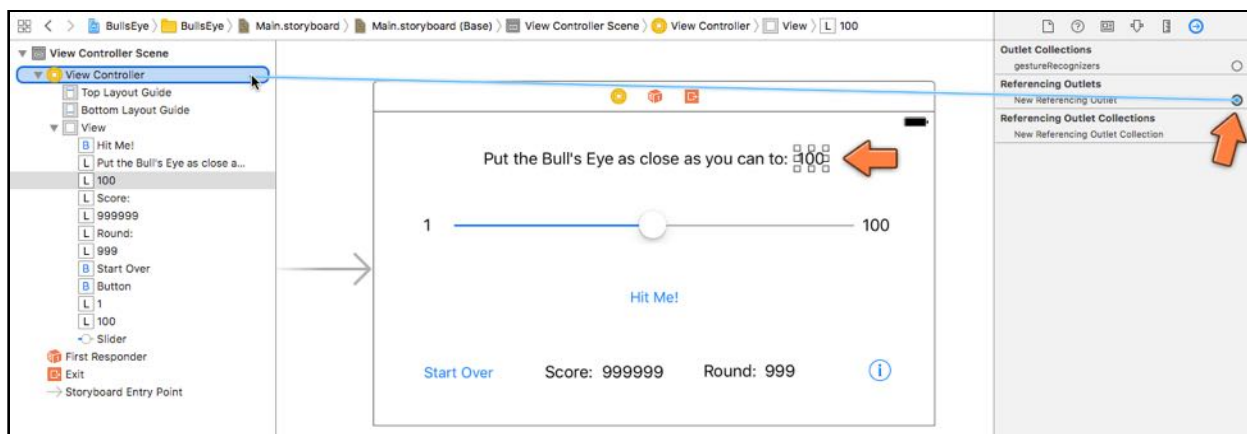
1. Create an outlet for the label so you can send it messages
2. Give the label new text to display

This will be very similar to what you did with the slider. Recall that you added an `@IBOutlet` variable so you could reference the slider anywhere from within the view controller. Using this outlet variable you could ask the slider for its value, through `slider.value`. You'll do the same thing for the label.

- In **ViewController.swift**, add the following line below the other outlet:

```
@IBOutlet weak var targetLabel: UILabel!
```

- In **Main.storyboard**, click to select the label (the one at the top that says "100").
- Go to the **Connections inspector** and drag from **New Referencing Outlet** to **View Controller**.



Connecting the target value label to its outlet

- Select **targetLabel** from the popup, and the connection is made.
- Now on to the good stuff. Add the following method below `startNewRound()` in **ViewController.swift**:

```
func updateLabels() {
    targetLabel.text = String(targetValue)
}
```

You're putting this logic into its own method because it's something you might use from different places.

The name of the method makes it clear what it does: it updates the contents of the labels. Currently it's just setting the text of a single label, but later on you will add code to update the other labels as well (total score, round number).

The code inside `updateLabels()` should have no surprises for you, although you may wonder why you cannot simply do:

```
targetLabel.text = targetValue
```

The answer is that you cannot put a value of one data type into a variable of another type. The square peg doesn't fit into the round hole.

The `targetLabel` outlet references a `UILabel` object. The `UILabel` object has a `text` property, which is a string object. You can only put string values into `text` but the above line tries to put `targetValue` into it, which is an `Int`. That won't fly because an `Int` and a string are two very different kinds of things.

So you have to convert the `Int` into a string, and that is what `String(targetValue)` does. It's similar to what you've seen before with `Float(...)` and `Int(...)`.

Just in case you were wondering, you can also write it as a string with a placeholder like you've done before:

```
targetLabel.text = "\\(targetValue)"
```

Which one you like better is a matter of taste. Either approach will work fine.

Notice that `updateLabels()` is a regular method – it is not attached to any UI controls as an action – so it won't do anything until you actually call it. (You can tell because it doesn't say `@IBAction` anywhere.)

The logical place to call `updateLabels()` would be after each call to `startNewRound()`, because that is where you calculate the new target value.

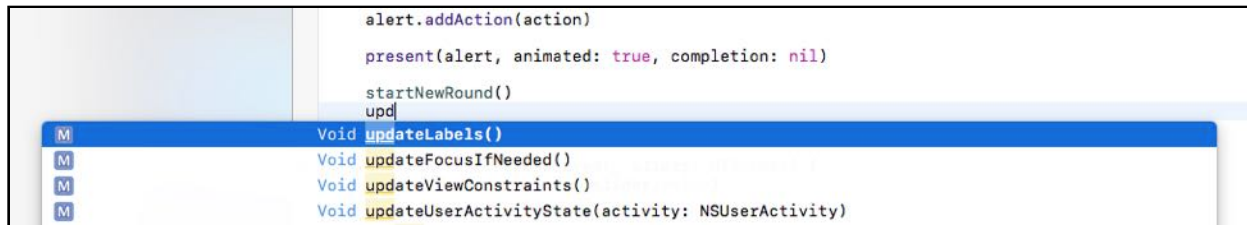
Currently, you send the `startNewRound()` message from two places: `viewDidLoad()` and `showAlert()`, so let's update these methods.

► Change `viewDidLoad()` and `showAlert()` to:

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    startNewRound()  
    updateLabels()      // add this line  
}
```

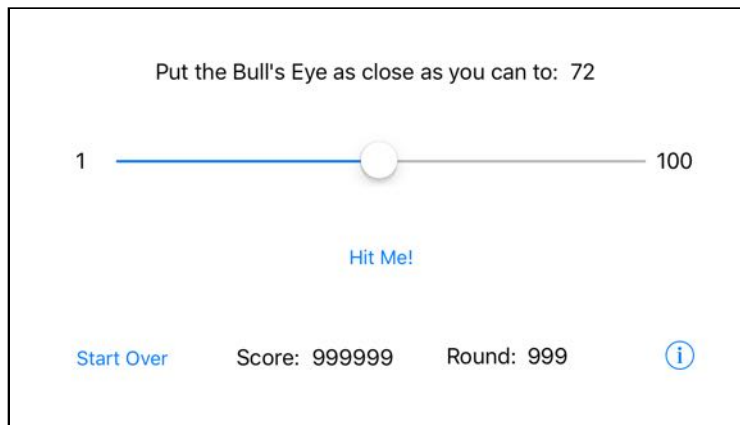
```
@IBAction func showAlert() {  
    . . .  
    startNewRound()  
    updateLabels()      // add this line  
}
```

You should be able to type just the first few letters of the method name, **upd**, and Xcode will complete the rest. Press **Enter** to accept the suggestion:



Xcode autocomplete offers suggestions

► Run the app and you'll actually see the random value on the screen. That should make it a little easier to aim for.



The label in the top-right corner now shows the random value

You can find the project files for the app up to this point under **03 - Outlets** in the tutorial's Source Code folder.



Action methods vs. normal methods

So what is the difference between an action method and a regular method?

Answer: Nothing.

An action method is really just the same as any other method. The only special thing is the `@IBAction` specifier. This allows Interface Builder to see the method so you can connect it to your buttons, sliders, and so on.

Other methods, such as `viewDidLoad()`, do not have the `@IBAction` specifier. This is a good thing because all kinds of mayhem would occur if you hooked such methods up to your buttons.

This is the simple form of an action method:

```
@IBAction func showAlert()
```

You can also ask for a reference to the object that triggered this action, using a parameter:

```
@IBAction func sliderMoved(_ slider: UISlider)
@IBAction func buttonTapped(_ button: UIButton)
```

But the following method cannot be used as an action from Interface Builder:

```
func updateLabels()
```

It is not marked as @IBAction and as a result Interface Builder can't see it. To use updateLabels(), you will have to call it yourself.



If you've made it this far, then I'm guessing you like what you're reading. :-)

This is only the first tutorial from my book *The iOS Apprentice: Beginning iOS Development with Swift*. The full book has three more of these huge tutorials – and in each you will develop a complete app from scratch.

The tutorials move from beginning to intermediate topics. Each new app is a little more advanced than the one before. Together they cover most of what you need to know to make your own apps.

By the end of the series you'll have learned the essentials of Swift and the iOS development kit. More importantly, you should have a pretty good idea of how all the different parts fit together and how to solve problems like a pro developer.

I'm confident that after working through these tutorials you'll be able to go out on your own and turn your ideas into real apps!



You might not know everything yet, but you will be able to stand on your own two feet as a developer – and you’ll be prepared for your journey into the exciting world of iPhone and iPad development.

Some highlights of what *The iOS Apprentice* will teach you:

- **How to program in Swift**, even if you’ve never programmed before or if the thought of learning a new language scares you.
- **How to think like a programmer**. You are more than a code monkey who just punches source code into an editor. As a programmer you’ll have to think through difficult computational problems and find creative solutions. Once you possess this valuable skill, you can program anything!
- **Experience with the SDK**. The iOS SDK is huge and there is no way we can cover everything – but we don’t need to. You just need to master the essential building blocks, such as navigation controllers and table views. You’ll also learn how to use web services from your apps and how to make iPad apps. Once you understand these fundamentals, you can easily find out for yourself how the rest of the SDK works.
- **How to make apps look and feel great**. There is more to making apps than just programming. We’ll discuss user interface design as well as graphics and animation techniques. You’ll already get a taste of that in this first tutorial when you give the game a makeover and add support for different iPhone models.
- **The latest and greatest**. We will take full advantage of Swift and the latest iOS features such as Auto Layout and Universal Storyboards. There is no point in teaching you the *old* way of iOS development that was current when a lot of other programming books were written, but is now hopelessly out-of-date. Every new version of iOS adds improved development techniques and you’ll use these to your benefit.

This is not just a bunch of dry theory but *hands-on practical advice*! I’ll explain how everything works along the way while you’re making real apps, with lots of pictures that clearly illustrate what is going on.

If this sounds like your idea of a fun time, then hop to raywenderlich.com/store/ios-apprentice to get the full *iOS Apprentice* series.

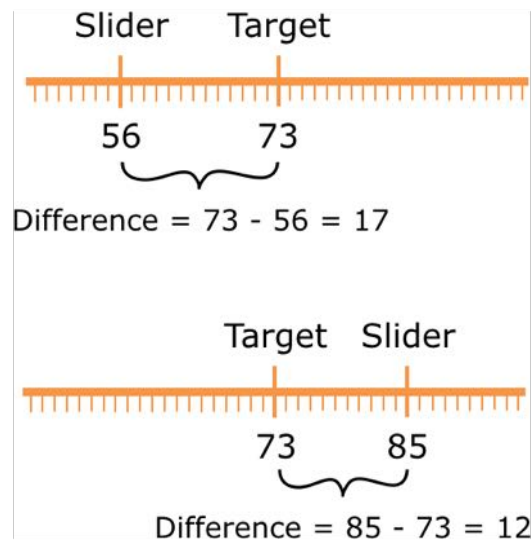
Calculating the score

Now that you have both the target value (the random number) and a way to read the slider’s position, you can calculate how many points the player scored.

The closer the slider is to the target, the more points for the player.

To calculate the score for this round, you look at how far off the slider’s value is

from the target:



Calculating the difference between the slider position and the target value

A simple approach to find the distance between the target and the slider is to subtract `currentValue` from `targetValue`.

Unfortunately, that gives a negative value if the slider is to the right of the target because now `currentValue` is greater than `targetValue`.

You need some way to turn that negative distance into a positive value – or you end up subtracting points from the player's score (unfair!).

Always doing the subtraction the other way around – `currentValue` minus `targetValue` – won't solve things because then the difference will be negative if the slider is to the left of the target instead of the right.

Hmm, it looks like we're in trouble here...

Exercise: How would you frame the solution to this problem if I asked you to solve it in natural language? Don't worry about how to express it in computer language for now, just think of it in plain English. ■

I came up with something like this:

- *If the slider's value is greater than the target value, then the difference is: slider value minus the target value.*
- *However, if the target value is greater than the slider value, then the difference is: target value minus the slider value.*
- *Otherwise, both values must be equal, and the difference is zero.*

This will always lead to a difference that is a positive number, because you always subtract the smaller number from the larger one.

Do the math:

If the slider is at position 60 and the target value is 40, then onscreen the slider is to the right of the target value, and the difference is $60 - 40 = 20$.

However, if the slider is at position 10 and the target is 30, then the slider is to the left of the target and has a smaller value. The difference here is $30 - 10 =$ also 20.



Algorithms

What you've just done is come up with an *algorithm*, which is a fancy term for a series of mechanical steps for solving a computational problem. This is only a very simple algorithm, but it is one nonetheless.

There are many famous algorithms, such as *quicksort* for sorting a list of items and *binary search* for quickly searching through such a sorted list. Other people have already invented many algorithms that you can use in your own programs, so that saves you a lot of thinking!

However, in all the programs that you write you'll probably have to come up with a few algorithms of your own. Some are simple such as the one above; others can be pretty hard and might cause you to throw up your hands in despair. But that's part of the fun of programming.

The academic field of Computer Science concerns itself largely with studying algorithms and finding better ones.

You can describe any algorithm in plain English. It's just a series of steps that you perform to calculate something. Often you can perform that calculation in your head or on paper, the way you did above. But for more complicated algorithms doing that might take you forever, so at some point you'll have to convert the algorithm to computer code.

The point I'm trying to make is this: if you ever get stuck and you don't know how to make your program calculate something, take a piece of paper and try to write out the steps in English. Set aside the computer for a moment and think the steps through. How you would perform this calculation by hand?

Once you know how to do that, writing the algorithm in computer code should be a piece of cake.



It is possible you came up with a different way to solve this little problem, and I'll show you two alternatives in a minute, but let's convert this one to computer code first:

```
var difference: Int
if currentValue > targetValue {
    difference = currentValue - targetValue
} else if targetValue > currentValue {
    difference = targetValue - currentValue
} else {
    difference = 0
}
```

The "if" construct is new. It allows your code to make decisions and it works much like you would expect from English. Generally, it works like this:

```
if something is true {
    then do this
} else if something else is true {
    then do that instead
} else {
    do something when neither of the above are true
}
```

You put a so-called *logical condition* after the if keyword. If that condition turns out to be true, for example currentValue is greater than targetValue, then the code in the block between the { } brackets is executed.

However, if the condition is not true, then the computer looks at the else if condition and evaluates that. There may be more than one else if, and it tries them one by one from top to bottom until one proves to be true.

If none of the conditions are found to be valid, then the code in the else block is executed.

In the implementation of this little algorithm you first create a local variable named difference to hold the result. This will either be a positive whole number or zero, so an Int will do:

```
var difference: Int
```

Then you compare the currentValue against the targetValue. First you determine if currentValue is greater than targetValue:

```
if currentValue > targetValue {
```

The > is the *greater-than* operator. The condition currentValue > targetValue is

considered true if the value stored in the `currentValue` variable is at least one higher than the value stored in the `targetValue` variable. In that case, the following line of code is executed:

```
difference = currentValue - targetValue
```

Here you subtract `targetValue` (the smaller one) from `currentValue` (the larger one) and store the difference in the `difference` variable.

Notice how I chose variable names that clearly describe what kind of data the variable contains. Often you will see code such as this:

```
a = b - c
```

It is not immediately clear what this is supposed to mean, other than that some arithmetic is taking place. The variable names "a", "b" and "c" don't give any clues as to their intended purpose.

Back to the if-statement. If `currentValue` is equal to or less than `targetValue`, the condition is untrue (or *false* in computer-speak) and the program will skip the code block until it reaches the next condition:

```
} else if targetValue > currentValue {
```

The same thing happens here as before, except that now the roles of `targetValue` and `currentValue` are reversed. The computer will only execute the following line when `targetValue` is the greater of the two values:

```
difference = targetValue - currentValue
```

This time you subtract `currentValue` from `targetValue` (i.e. the other way around) and store the result in the `difference` variable.

There is only one situation you haven't handled yet, and that is when `currentValue` and `targetValue` are equal. If this happens, the player has put the slider exactly on top of the random number, a perfect score. In that case the difference is 0:

```
} else {  
    difference = 0  
}
```

At this point you've already determined that one value is not greater than the other, nor is it smaller, leaving you only one conclusion to draw: the numbers must be equal.

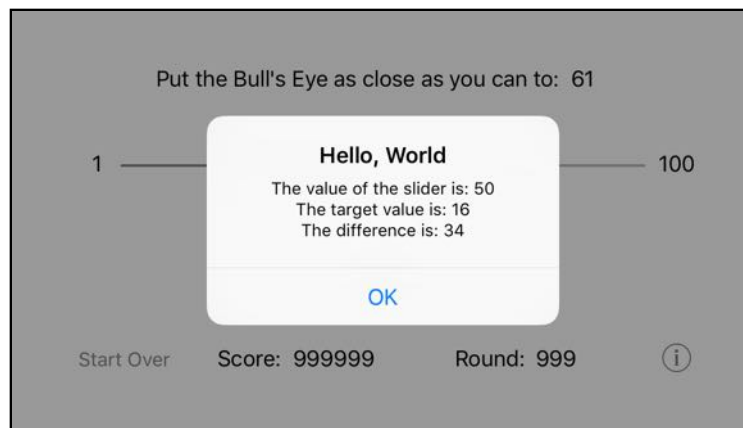
► Let's put this algorithm into action. Add it to the top of `showAlert()`:

```
@IBAction func showAlert() {  
    var difference: Int  
    if currentValue > targetValue {  
        difference = currentValue - targetValue
```

```
} else if targetValue > currentValue {  
    difference = targetValue - currentValue  
} else {  
    difference = 0  
}  
  
let message = "The value of the slider is: \(currentValue)"  
              + "\nThe target value is: \(targetValue)"  
              + "\nThe difference is: \(difference)"  
    . . .  
}
```

Just so you can see that it works, you have added the difference value to the alert message as well.

► Run it and see for yourself.



The alert shows the difference between the target and the slider

Alternative ways to calculate the difference

I mentioned earlier that there are other ways to calculate the difference between `currentValue` and `targetValue` as a positive number. The above algorithm works well but it is eight lines of code. I think we can come up with a simpler approach that takes up fewer lines.

The new algorithm goes like this:

1. *Subtract the target value from the slider's value.*
2. *If the result is a negative number, then multiply it by -1 to make it a positive number.*

Now you're no longer avoiding the negative number, as computers can work just fine with negative numbers, but you simply turn it into a positive number.

Exercise: Convert this algorithm into source code. Hint: the English description of the algorithm contains the words "if" and "then", which is a pretty good indication

you'll have to use the if-statement. ■

You should have arrived at something like this:

```
var difference = currentValue - targetValue
if difference < 0 {
    difference = difference * -1
}
```

This is a pretty straightforward translation of the new algorithm.

You first subtract the two variables and put the result into the difference variable.

Notice that you can create the new variable and assign it the result of a calculation, all in one line. You don't need to put it onto two different lines, like so:

```
var difference: Int
difference = currentValue - targetValue
```

Also, in the one-liner version you didn't have to tell the compiler that difference takes Int values. Because both currentValue and targetValue are Ints, Swift is smart enough to figure out that difference should also be an Int.

This feature is called *type inference* and it's one of the big selling points of Swift.

Once you have the subtraction result, you use an if-statement to determine whether difference is negative, i.e. less than zero. If it is, you multiply by -1 and put the new result – now a positive number – back into the difference variable.

When you write,

```
difference = difference * -1
```

the computer first multiplies difference's value by -1. Then it puts the result of that calculation back into difference. In effect, this overwrites difference's old contents (the negative number) with the positive number.

Because this is a common thing to do, there is a handy shortcut:

```
difference *= -1
```

The *= operator combines * and = into a single operation. The end result is the same: the variable's old value is gone and it now contains the result of the multiplication.

You could also have written this algorithm as follows:

```
var difference = currentValue - targetValue
if difference < 0 {
    difference = -difference
}
```

Instead of multiplying by -1, you now use the negation operator to ensure difference's value is always positive. This works because negating a negative number makes it positive again. (Ask a math professor if you don't believe me.)

► Give these new algorithms a try. You should replace the old stuff at the top of `showAlert()` as follows:

```
@IBAction func showAlert() {  
    var difference = currentValue - targetValue  
    if difference < 0 {  
        difference = difference * -1  
    }  
  
    let message = . . .  
}
```

When you run this new version of the app (try it!), it should work exactly the same as before. The result of the computation does not change, only the technique you used.

The final alternative algorithm I want to show you uses a function.

You've already seen functions a few times before: you used `arc4random_uniform()` when you made random numbers and `lroundf()` for rounding off the slider's decimals.

To make sure a number is always positive, you can use the `abs()` function.

If you took math in school you might remember the term "absolute value", which is the value of a number without regard to its sign.

That's exactly what you need here and the standard library contains a convenient function for it, which allows you to reduce this entire problem to a single line:

```
let difference = abs(targetValue - currentValue)
```

It really doesn't matter whether you subtract `currentValue` from `targetValue` or the other way around. If the number is negative, `abs()` turns it positive. It's a handy function to remember.

► Make the change to `showAlert()` and try it out:

```
@IBAction func showAlert() {  
    let difference = abs(targetValue - currentValue)  
  
    let message = . . .  
}
```

It doesn't get much simpler than that!

Exercise: Something else has changed... can you spot it? ■

Answer: You wrote **let** difference instead of **var** difference.

Swift makes a distinction between variables and so-called *constants*. Unlike a variable, the value of a constant cannot change (you probably guessed it from the name).

You can only put something into the box of a constant once but not replace it with something else afterwards.

The keyword `var` creates a variable while `let` creates a constant. That means difference is now a constant, not a variable.

In the previous algorithms, the value of difference could possibly change. If it was negative, you turned it positive. That required difference to be a variable, because only variables can be assigned new values.

Now that you can calculate the whole thing in a single line, difference will never have to change once you've given it a value. In that case, it's better to make it a constant with `let`. (Why is that better? It makes your intent clear, which in turn helps the Swift compiler understand your program better.)

By the same token, `message`, `alert`, and `action` are also constants (and have been all along!). Now you know why you declared these objects with `let` instead of `var`. Once they've been given a value, they never need to change.

Constants are very common in Swift. Often you only need to hold onto a value for a very short time. If in that time the value never has to change, it's best to make it a constant (`let`) and not a variable (`var`).

What's the score?

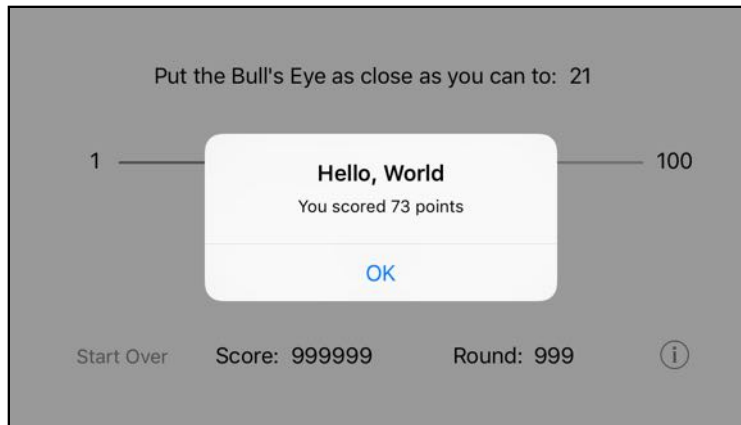
Now that you know how far off the slider is from the target, calculating the player's score for this round is easy.

► Change `showAlert()` to:

```
@IBAction func showAlert() {  
    let difference = abs(targetValue - currentValue)  
    let points = 100 - difference  
  
    let message = "You scored \(points) points"  
    . . .  
}
```

The maximum score you can get is 100 points if you put the slider right on the target and the difference is zero. The further away from the target you are, the fewer points you earn.

► Run the app and score some points!



The alert with the player's score for the current round

Exercise: Because the maximum slider position is 100 and the minimum is 1, the biggest difference is $100 - 1 = 99$. That means the absolute worst score you can have in a round is 1 point. Explain why this is so. (Eek! It requires math!) ■

Keeping track of the player's total score

In this game, you want to show the player's total score on the screen. After every round, the app should add the newly scored points to the total and then update the score label.

Because the game needs to keep the total score around for a long time, you will put it in an instance variable.

► Add a new score instance variable to **ViewController.swift**:

```
class ViewController: UIViewController {  
    var currentValue: Int = 0  
    var targetValue: Int = 0  
    var score = 0           // add this line  
}
```

Hey, what's that? Unlike for the other two instance variables, you did not state that score is an Int.

If you don't specify a data type, Swift uses *type inference* to figure out what type you meant. Because 0 is a whole number, Swift assumes that score should be an integer, and therefore automatically gives it the type Int. Handy!

In fact, you don't need to specify Int for the other instance variables either:

```
var currentValue = 0  
var targetValue = 0
```

► Make these changes.

Thanks to type inference, you only have to list the name of the data type when

you're not giving the variable an initial value. But most of the time, you can safely make Swift guess at the type.

I think type inference is pretty sweet! It will definitely save you some, uh, typing (in more ways than one!).

Now `showAlert()` can be amended to update this score variable.

► Make the following changes:

```
@IBAction func showAlert() {  
    let difference = abs(targetValue - currentValue)  
    let points = 100 - difference  
  
    score += points           // add this line  
  
    let message = "You scored \(points) points"  
    . . .  
}
```

Nothing too shocking here. You just added the following line:

```
score += points
```

This adds the points that the user scored in this round to the total score. You could also have written it like this:

```
score = score + points
```

Personally, I prefer the shorthand `+=` version but either one is okay. Both accomplish exactly the same thing.

Showing the score on the screen

You're going to do exactly the same thing that you did for the target label: hook up the score label to an outlet and put the score value into the label's text property.

Exercise: See if you can do the above without my help. You've already done these things before for the target value label, so you should be able to repeat these steps by yourself for the score label. ■

You should have done the following. You added this line to **ViewController.swift**:

```
@IBOutlet weak var scoreLabel: UILabel!
```

Then you went into the storyboard and connected the label (the one that says 999999) to the new `scoreLabel` outlet.

Unsure how to connect the outlet? There are several ways to make connections from user interface objects to the view controller's outlets:

- Ctrl-click on the object to get a context-sensitive popup menu. Then drag from

New Referencing Outlet to View Controller (you did this with the slider).

- Go to the Connections Inspector for the label. Drag from New Referencing Outlet to View Controller (you did this with the target label).
- Ctrl-drag from View Controller to the label (give this one a try now). Make sure you do it in this order; ctrl-dragging from the label to the view controller won't work.

There is more than one way to skin a cat, uh, connect outlets.

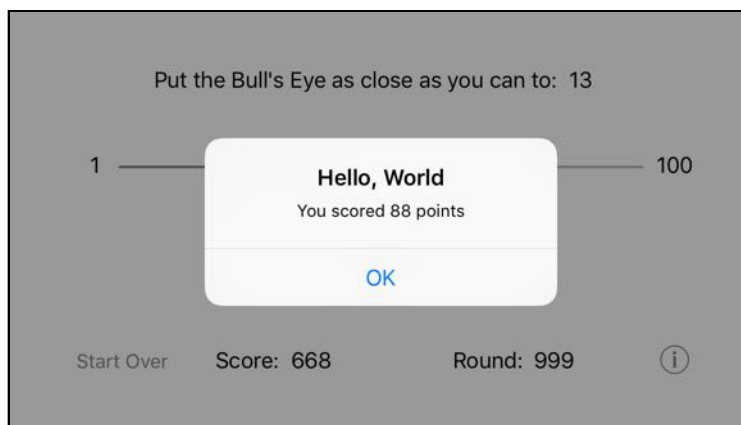
Great, that gives you a `scoreLabel` outlet that you can use to put text into the label. Now where in the code shall you do that? In `updateLabels()`, of course.

► Back in **ViewController.swift**, change `updateLabels()` to the following:

```
func updateLabels() {  
    targetLabel.text = String(targetValue)  
    scoreLabel.text = String(score)  
}
```

Nothing new here. You convert the score – which is an `Int` – into a `String` and then give that string to the label's `text` property. In response to that, the label will redraw itself with the new score.

► Run the app and verify that the points for this round are added to the total score label whenever you tap the button.



The score label keeps track of the player's total score

One more round...

Speaking of rounds, you also have to increment the round number each time the player starts a new round.

Exercise: Keep track of the current round number (starting at 1) and increment it when a new round starts. Display the current round number in the corresponding label. I may be throwing you into the deep end here, but if you've been able to

follow the instructions so far, then you've already seen all the pieces you will need to pull this off. Good luck! ■

If you guessed that you had to add another instance variable, then you were right. You should have added the following line to the source code:

```
var round = 0
```

It's also OK if you included the name of the data type, even though that is not strictly necessary:

```
var round: Int = 0
```

Also an outlet for the label:

```
@IBOutlet weak var roundLabel: UILabel!
```

As before, you should have connected the label to this outlet in Interface Builder.

Don't forget to make those connections

Forgetting to make the connections in Interface Builder is an often-made mistake, especially by yours truly.

It happens to me all the time that I make the outlet for a button and write the code to deal with taps on that button, but when I run the app it doesn't work. Usually it takes me a few minutes and some head scratching to realize that I forgot to connect the button to the outlet or the action method.

You can tap on the button all you want, but unless that connection exists your code will not respond.

Finally, `updateLabels()` should now look like this:

```
func updateLabels() {  
    targetLabel.text = String(targetValue)  
    scoreLabel.text = String(score)  
    roundLabel.text = String(round)  
}
```

Did you also figure out where to increment the `round` variable?

I'd say the `startNewRound()` method is a pretty good place. After all, you call this method whenever you start a new round. It makes sense to increment the round counter there.

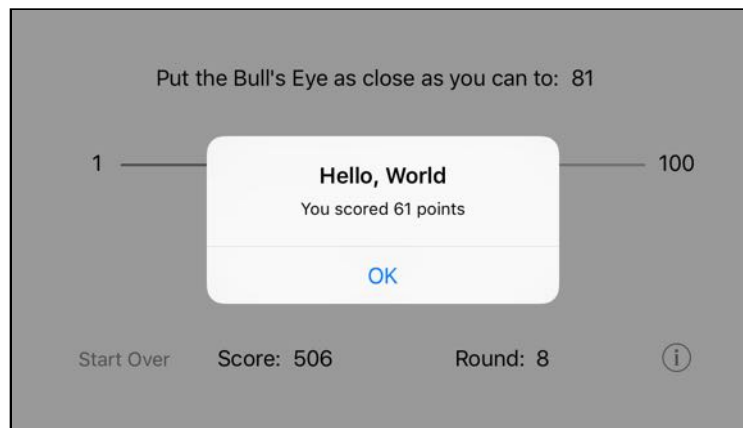
► Change `startNewRound()` to:

```
func startNewRound() {  
    round += 1 // add this line
```

```
targetValue = 1 + Int(arc4random_uniform(100))
currentValue = 50
slider.value = Float(currentValue)
}
```

Note that when you declared the round instance variable, you gave it a default value of 0. Therefore, when the app starts up, round is initially 0. When you call `startNewRound()` for the very first time, it adds 1 to this initial value and as a result the first round is properly counted as round 1.

► Run the app and try it out. The round counter should update whenever you press the Hit Me button.



The round label counts how many rounds have been played

You can find the project files for the app up to this point under **04 - Rounds and Score** in the tutorial's Source Code folder. If you get stuck, compare your version of the app with those source files to see if you missed anything.

Polishing the game

You could leave it at this and have a playable game. The gameplay rules are all implemented and the logic doesn't seem to have any big flaws. As far as I can tell, there are no bugs. But there is still some room for improvement.

Obviously, the game is not very pretty yet and you will get to work on that soon. In the mean time, there are a few smaller tweaks you can make.

Unless you already changed it, the title of the alert still says "Hello, World!" You could give it the name of the game, "Bull's Eye", but I have a better idea. What if you change the title depending on how well the player did?

If the player put the slider right on the target, the alert could say: "Perfect!" If the slider is close to the target but not quite there, it could say, "You almost had it!" If the player is way off, the alert could say: "Not even close..." And so on. This gives

players a little more feedback on how well they did.

Exercise: Think of a way to accomplish this. Where would you put this logic and how would you program it? Hint: there are an awful lot of “if’s” in the preceding sentences. ■

The right place for this logic is `showAlert()`, because that is where you create the `UIAlertController`. You already do some calculations to make the message text and now you will do something similar for the title text.

➤ Here is the changed method in its entirety:

```
@IBAction func showAlert() {
    let difference = abs(targetValue - currentValue)
    let points = 100 - difference
    score += points

    // add these lines
    let title: String
    if difference == 0 {
        title = "Perfect!"
    } else if difference < 5 {
        title = "You almost had it!"
    } else if difference < 10 {
        title = "Pretty good!"
    } else {
        title = "Not even close..."
    }

    let message = "You scored \(points) points"

    let alert = UIAlertController(title: title,           // change this
                                message: message,
                                preferredStyle: .alert)

    let action = UIAlertAction(title: "OK", style: .default, handler: nil)
    alert.addAction(action)
    present(alert, animated: true, completion: nil)

    startNewRound()
    updateLabels()
}
```

You create a new local string named `title`, which will contain the text that goes at the top of the alert. Initially, this title doesn’t have any value.

To decide which title text to use, you look at the difference between the slider position and the target:

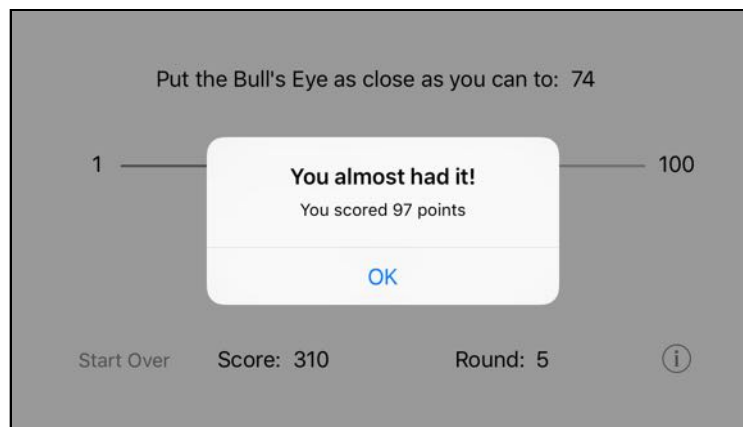
- If it equals 0, then the player was spot-on and you put the text “Perfect!” into `title`.
- If the difference is less than 5, you use the text “You almost had it!”
- A difference less than 10 is “Pretty good!”

- However, if the difference is 10 or greater, then you consider the player's attempt "Not even close..."

Can you follow the logic here? It's just a bunch of if-statements that consider the different possibilities and choose a string in response.

When you create the UIAlertController object, you now give it this title string instead of a fixed text.

Run the app and play the game for a bit. You'll see that the title text changes depending on how well you're doing. That if-statement sure is handy!



The alert with the new title

Exercise: Give the player an additional 100 bonus points when she has a perfect score. This will encourage players to really try to place the bull's eye right on the target. Otherwise, there isn't much difference between 100 points for a perfect score and 98 or 95 points if you're close but not quite there.

Now there is an incentive for trying harder – a perfect score is no longer worth just 100 but 200 points. Maybe you can also give the player 50 bonus points for being just one off. ■

► Here is how I would have made these changes:

```
@IBAction func showAlert() {
    let difference = abs(targetValue - currentValue)
    var points = 100 - difference    // change let to var

    let title: String
    if difference == 0 {
        title = "Perfect!"
        points += 100                // add this line
    } else if difference < 5 {
        title = "You almost had it!"
        if difference == 1 {        // add these lines
            points += 50
        }
    }
}
```

```
    } else if difference < 10 {  
        title = "Pretty good!"  
    } else {  
        title = "Not even close..."  
    }  
  
    score += points // move this line here  
    . . .  
}
```

You should notice a few things:

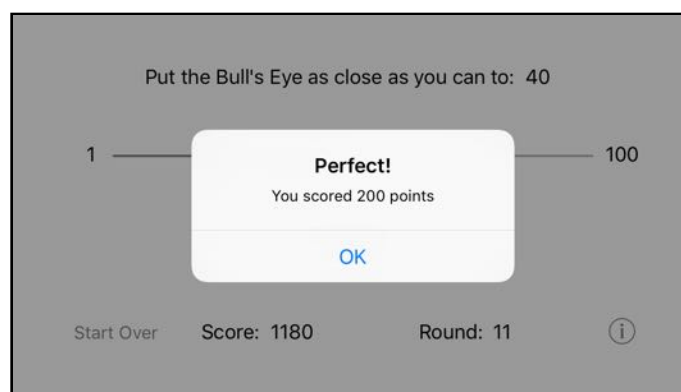
- In the first if you'll see a new statement between the curly brackets. When the difference is equal to zero, you now not only set the title to "Perfect!" but also award an extra 100 points.
- The second if has changed too. There is now an if inside another if. Nothing wrong with that! You want to handle the case where difference is 1 in order to give the player bonus points. That happens inside the new if-statement.

After all, if the difference is more than 0 but less than 5, it could also be 1 (but not necessarily all the time). Therefore, you perform an additional check to see if the difference truly was 1, and if so, add 50 extra points.

- Because these new if-statements add extra points, points can no longer be a constant; it now needs to be a variable. That's why you changed it from `let` into `var`.
- Finally, the line `score += points` has moved below the ifs. This is necessary because the app might update the points variable inside those if-statements and you want those additional points to count towards the score as well.

If you did it slightly differently, then that's fine too, as long as it works! There is often more than one way to program something, and if the results are the same then each way is equally valid.

► Run the app to see if you can score some bonus points!



Raking in the points...



Local variables recap

I would like to point out one more time the difference between local variables and instance variables. As you should know by now, a local variable only exists for the duration of the method that it is defined in, while an instance variable exists as long as the view controller (the object that owns it) exists. The same thing is true for constants.

In `showAlert()`, there are six locals and you use three instance variables:

```
let difference = abs(targetValue - currentValue)
var points = 100 - difference
let title = . . .
score += points
let message = . . .
let alert = . . .
let action = . . .
```

Exercise: Point out which are the locals and which are the instance variables in the `showAlert()` method. Of the locals, which are variables and which are constants? ■

Answer: Locals are easy to recognize, because the first time they are used inside a method their name is preceded with `let` or `var`:

```
let difference = . . .
var points = . . .
let title = . . .
let message = . . .
let alert = . . .
let action = . . .
```

This syntax creates a new variable (`var`) or constant (`let`). Because these variables and constants are created inside the method, they are locals.

Those six items – `difference`, `points`, `title`, `message`, `alert`, and `action` – are restricted to the `showAlert()` method and do not exist outside of it. As soon as the method is done, the locals cease to exist.

You may be wondering how `difference`, for example, can have a different value every time the player taps the Hit Me button, even though it is a constant – after all, aren't constants given a value just once, never to change afterwards?

Here's why: each time a method is invoked, its local variables and constants are created anew. The old values have long been forgotten and you get all new ones.

When `showAlert()` is called, it creates a completely new instance of `difference` that is unrelated to the previous one. That particular constant value is only used until

the end of `showAlert()` and then it is forgotten again.

The next time `showAlert()` is called after that, it creates yet another new instance of `difference` (as well as new instances of the other locals `points`, `title`, `message`, `alert`, and `action`). And so on... There's some serious recycling going on here!

But inside a single invocation of `showAlert()`, `difference` can never change once it has its value. The only local that can change is `points`, because it's a `var`.

The instance variables, on the other hand, are defined outside of any method. It is common to put them at the top of the file:

```
class ViewController: UIViewController {  
  
    var currentValue = 0  
    var targetValue = 0  
    var score = 0  
    var round = 0  
  
}
```

As a result, you can use these variables from any method, without the need to declare them again, and they will keep their values.

If you were to do this,

```
@IBAction func showAlert() {  
    let difference = abs(targetValue - currentValue)  
    var points = 100 - difference  
  
    var score = score + points    // doesn't work!  
    . . .  
}
```

then things wouldn't work as you'd expect them to. Because you now put `var` in front of `score`, you have made it a new local variable that is only valid inside this method.

In other words, this won't add points to the *instance variable* `score` but to a new *local variable* that also happens to be named `score`. The instance variable `score` never gets changed, even though it has the same name.

Obviously that is not what you want to happen here. Fortunately, the above won't even compile. Swift knows there's something fishy about that line.

Note: To make a distinction between the two types of variables, so that it's always clear at a glance how long they will live, some programmers prefix the names of instance variables with an underscore.

They would name the variable `_score` instead of just `score`. Now there is less confusion because names beginning with an underscore won't be mistaken for being locals. This is only a convention. Swift doesn't care one way or the other how you spell your instance variables.

Other programmers use different prefixes, such as "m" (for member) or

"f" (for field) for the same purpose. Some even put the underscore *behind* the variable name. Madness!



Waiting for the alert to go away

There is something that bothers me about the game. You may have noticed it too...

As soon as you tap the Hit Me button and the alert pops up, the slider immediately jumps back to its center position, the round number increments, and the target label already gets the new random number.

What happens is that the new round already gets started while you're still watching the results of the last round. That's a little confusing.

It would be better to wait with starting the new round until *after* the player has dismissed the alert popup. Only then is the current round truly over.

Maybe you're wondering why this isn't already happening? After all, in `showAlert()` you only call `startNewRound()` after you've shown the alert popup:

```
@IBAction func showAlert() {  
    . . .  
  
    let alert = UIAlertController(. . .)  
    let action = UIAlertAction(. . .)  
    alert.addAction(action)  
  
    // Here you make the alert visible:  
    present(alert, animated: true, completion: nil)  
  
    // Here you start the new round:  
    startNewRound()  
    updateLabels()  
}
```

Contrary to what you may expect, `present(alert, ...)` doesn't hold up execution of the rest of the method until the alert popup is dismissed. That's how alerts on other platforms tend to work, but not on iOS.

Instead, `present(alert, ...)` puts the alert on the screen and immediately returns. The rest of the `showAlert()` method is executed right away, and the new round already starts before the alert popup has even finished animating.

In programmer-speak, alerts work *asynchronously*. Much more about that in a later tutorial, but what it means for you right now is that you don't know in advance

when the alert will be done. But you can bet it will be well after `showAlert()` has finished.

So if you can't wait in `showAlert()` until the popup is dismissed, then how do you wait for it to close?

The answer is simple: events! As you've seen, a lot of the programming for iOS involves waiting for specific events to occur – buttons being tapped, sliders being moved, and so on. This is no different. You have to wait for the "alert dismissed" event somehow. In the mean time, you simply do nothing.

Here's how it works:

For each button on the alert, you have to supply a `UIAlertAction` object. This object tells the alert what the text on the button is – "OK" – and what the button looks like (you're using the default style here):

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

The third parameter, `handler`, tells the alert what should happen when the button is pressed. This is the "alert dismissed" event you've been looking for.

Currently `handler` is `nil`, which means nothing happens. To change this, you'll need to give the `UIAlertAction` some source code to perform when the button is tapped. When the user finally taps OK, the alert will remove itself from the screen and jump to your code. That's your cue to take it from there.

This is also known as the *callback* pattern. There are several ways this pattern manifests on iOS. Often you'll be asked to create a new method to handle the event. But here you'll use something new: a *closure*.

► Change the bottom bit of `showAlert()` to:

```
@IBAction func showAlert() {  
    . . .  
    let alert = UIAlertController(. . .)  
  
    let action = UIAlertAction(title: "OK", style: .default,  
                              handler: { action in  
                                  self.startNewRound()  
                                  self.updateLabels()  
                              })  
  
    alert.addAction(action)  
    present(alert, animated: true, completion: nil)  
}
```

Two things have happened here:

1. You removed the calls to `startNewRound()` and `updateLabels()` from the bottom of the method. (Don't forget this part!)
2. You placed them inside a block of code that you gave to `UIAlertAction`'s handler

parameter.

Such a block of code is called a closure. You can think of it as a method without a name. This code is not performed right away, only when the OK button is tapped. This particular closure tells the app to start a new round and update the labels when the alert is dismissed.

► Run it and see for yourself. I think the game feels a lot better this way.

Self

You may be wondering why in the handler block you did `self.startNewRound()` instead of just writing `startNewRound()` like before.

The `self` keyword allows the view controller to refer to itself. That shouldn't be too strange a concept. When you say, "I want ice cream," you use the word "I" to refer to yourself. Similarly, objects can talk about (or to) themselves as well.

Normally you don't need to use `self` to send messages to the view controller, even though it is allowed. The exception: inside closures you *do* have to use `self` to refer to the view controller.

This is a rule in Swift. If you forget `self` in a closure, Xcode doesn't want to build your app (try it out). This rule exists because closures can "capture" variables, which comes with surprising side effects. You'll learn more about that in the other tutorials.

Starting over

No, you're not going to throw away the source code and start this project all over! I'm talking about the game's "Start Over" button. This button is supposed to reset the score and put the player back into the first round.

One use of the Start Over button is for playing against another person. The first player does ten rounds, then the score is reset and the second player does ten rounds. The player with the highest score wins.

Exercise: Try to implement this Start Over button on your own. You've already seen how you can make the view controller react to button presses, and you should be able to figure out how to change the score and round variables. ■

How did you do? If you got stuck, then follow the instructions below.

First, add a method to **ViewController.swift** that starts a new game. I suggest you put it near `startNewRound()` because the two are conceptually related.

► Add the new method:

```
func startNewGame() {  
    score = 0
```

```
round = 0
startNewRound()
}
```

This method resets the score and round number, and starts a new round as well.

Notice that you set round to 0 here, not to 1. You use 0 because incrementing the value of round is the first thing that startNewRound() does.

If you were to set round to 1, then startNewRound() would add another 1 to it and the first round would actually be labeled round 2.

So you begin at 0, let startNewRound() add one and everything will work out fine.

(It's probably easier to figure this out from the code than from my explanation. This should illustrate why we don't program computers in English.)

You also need an action method to handle taps on the Start Over button.

➤ Add the action method to **ViewController.swift**:

```
@IBAction func startOver() {
    startNewGame()
    updateLabels()
}
```

It doesn't really matter where you place this method, but below the other action methods is a nice place for it.

When the Start Over button is pressed, the startOver() action method first calls startNewGame() to start a new game. (See, if you choose method names that make sense, then reading source code really isn't that hard.)

Because startNewGame() changes the contents of the instance variables you also call updateLabels() to update the text of the score, round and target labels.

Just to make things consistent, in viewDidLoad() you should replace the call to startNewRound() by startNewGame(). Because score and round are already 0 when the app starts, it won't really make any difference to how the app works but it does make the intention of the source code clearer.

➤ Make this change:

```
override func viewDidLoad() {
    super.viewDidLoad()
    startNewGame()      // this line changed
    updateLabels()
}
```

Finally, you need to connect the Start Over button to the action method.

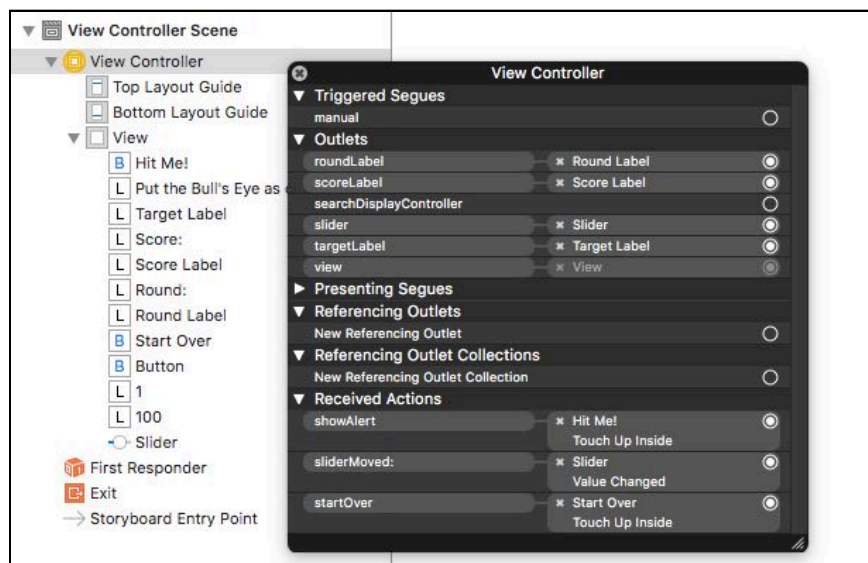
➤ Open the storyboard and Ctrl-drag from the **Start Over** button to **ViewController**. Let go of the mouse button and pick **startOver** from the popup.

That connects the button's Touch Up Inside event to the action you have just defined.

► Run the app and play a few rounds. Press Start Over and the game puts you back at square one.

Tip: If you're losing track of what button or label is connected to what method, you can click on **View Controller** in the storyboard to see all the connections that you have made so far.

You can either right-click on View Controller to get a popup, or simply view the connections in the **Connections inspector**. This shows all the connections that have been made to the view controller.

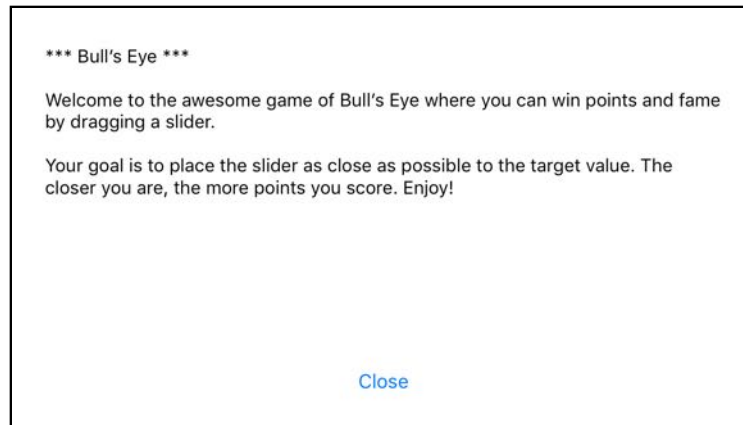


All the connections from View Controller to the other objects

You can find the project files for the current version of the app under **05 - Polish** in the tutorial's Source Code folder.

Adding the About screen

I hope you're not fed up with this app yet, as there is one more feature that I wish to add to it, an "about" screen that shows some information about the game:



The new About screen

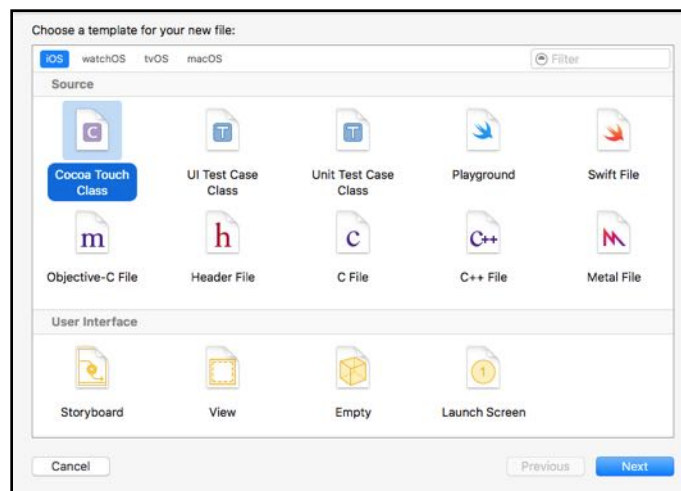
This new screen contains a so-called *text view* with the gameplay rules and a button that lets the player close the screen. You get to the About screen by tapping the **(i)** button in the game.

Most apps have more than one screen, even very simple games, so this is as good a time as any to learn how to add additional screens to your apps.

I have pointed it out a few times already: each screen in your app will have its own view controller. If you think “screen”, think “view controller”.

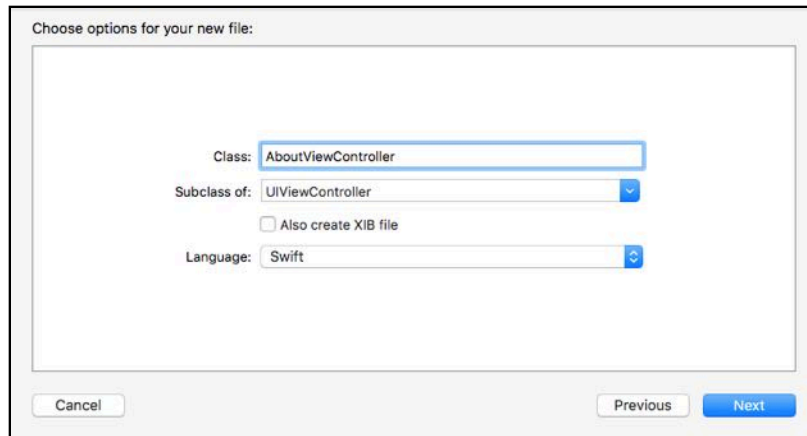
Xcode automatically created the main ViewController object for you but the view controller for the About screen you’ll have to make yourself.

► Go to Xcode’s **File** menu and choose **New → File...** In the window that pops up, choose the **Cocoa Touch Class** template (if you don’t see it then make sure **iOS** is selected at the top):



Choosing the file template for Cocoa Touch Class

Click **Next**. Xcode gives you some options to fill out:

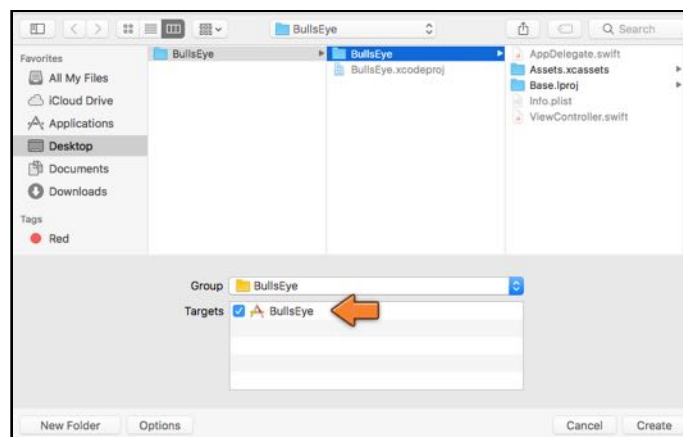


The options for the new file

Choose the following:

- Class: **AboutViewController**
- Subclass of: **UIViewController**
- Also create XIB file: Leave this box unchecked.
- Language: **Swift**

Click **Next**. Xcode will ask you where to save this new view controller:



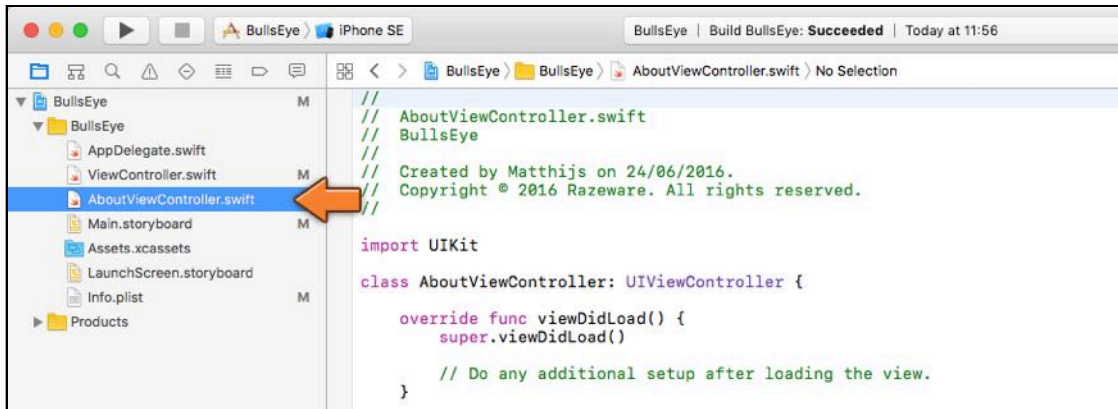
Saving the new file

➤ Choose the **BullsEye** folder (this folder should already be selected).

Also make sure **Group** says **BullsEye** and that there is a checkmark in front of BullsEye in the list of **Targets**. (If you don't see this panel, click the Options button.)

➤ Click **Create** to finish.

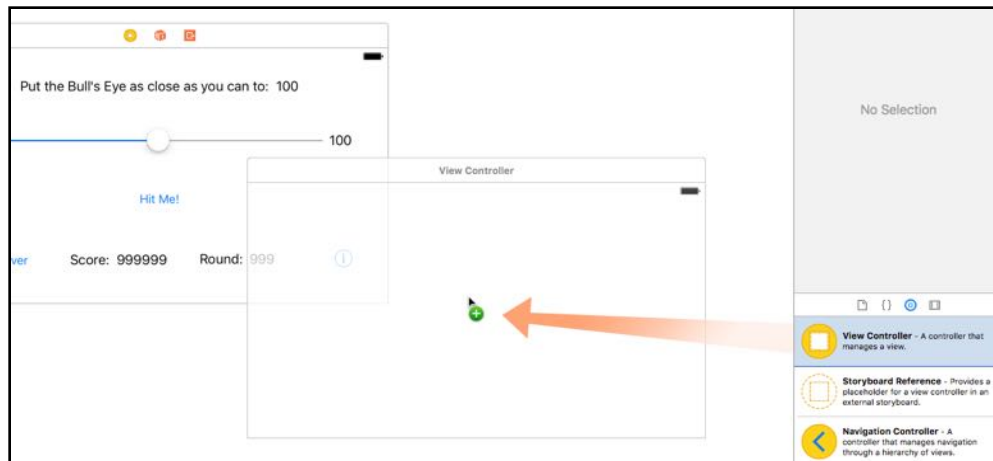
Xcode will make a new file and add it to your project. As you might have guessed, the new file is **AboutViewController.swift**.



The new file in the Project navigator

To design this new view controller, you need to pay a visit to Interface Builder.

- Open **Main.storyboard**. There is no scene representing the About view controller yet, so you'll have to add this first.
- From the **Object Library**, choose **View Controller** and drag it into the canvas, to the right of the main View Controller.



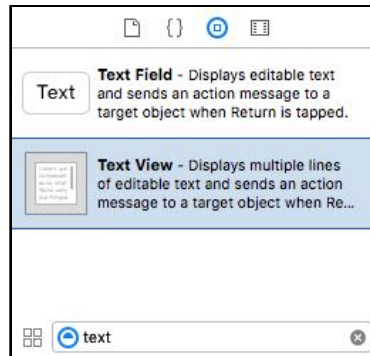
Dragging a new View Controller from the Object Library

This new view controller is totally blank. You may need to rearrange the storyboard so that the two view controllers don't overlap. Interface Builder isn't always very neat with where it puts things.

- Drag a new **Button** into the screen and give it the title **Close**. Put it somewhere in the bottom center of the view (use the blue guidelines to help position it).
- Drag a **Text View** into the view and make it cover most of the space above the

button.

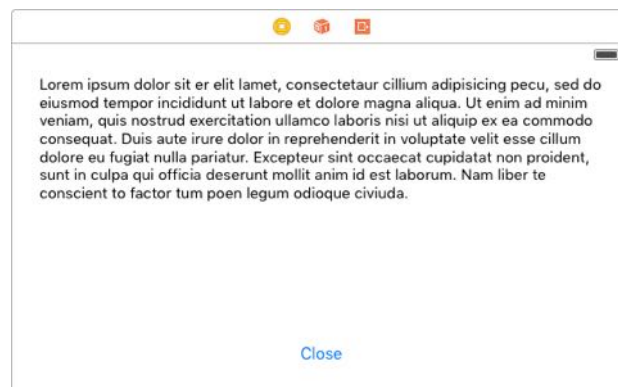
You can find these components in the Object Library. If you don't feel like scrolling, you can filter the components by typing in the field at the bottom:



Searching for text components

Note that there is also a Text Field, which is a single-line text component. You're looking for Text View, which can contain multiple lines of text.

After dragging both the text view and the button into the canvas, it should look something like this:



The About screen in the storyboard

► Double-click on the text view to make its contents editable. By default, the Text View contains a whole bunch of fake Latin placeholder text (also known as "Lorem Ipsum").

Copy-paste this new text into it:

*** Bull's Eye ***

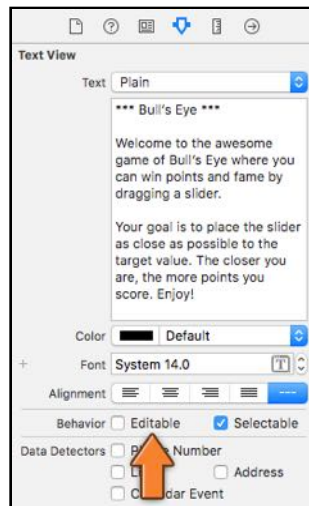
Welcome to the awesome game of Bull's Eye where you can win points and fame by dragging a slider.

Your goal is to place the slider as close as possible to the target

value. The closer you are, the more points you score. Enjoy!

You can also paste that text into the Attributes inspector for the text view if you find that easier.

► Make sure to uncheck the **Editable** setting, otherwise the user can actually type into the text view. For this game it should be set to read-only.

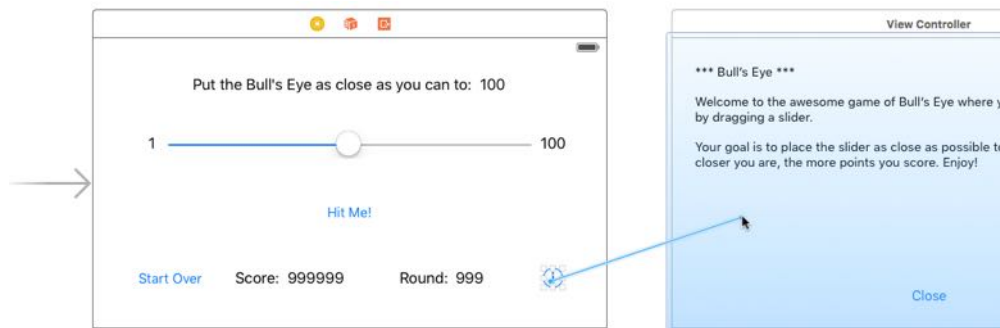


The Attributes inspector for the text view

That's the design of the screen finished for now.

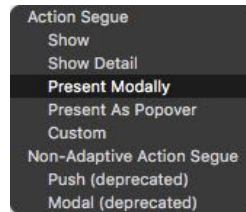
So how do you open this new About screen when the user presses the (i) button? Storyboards have a neat trick for this: *segues* (pronounced "seg-way" like the silly scooters). A segue is a transition from one screen to another and they are really easy to add.

► Click the **(i)** button in the **View Controller** to select it. Then hold down Ctrl and drag over to the **About** screen.



Ctrl-drag from one view controller to another to make a segue

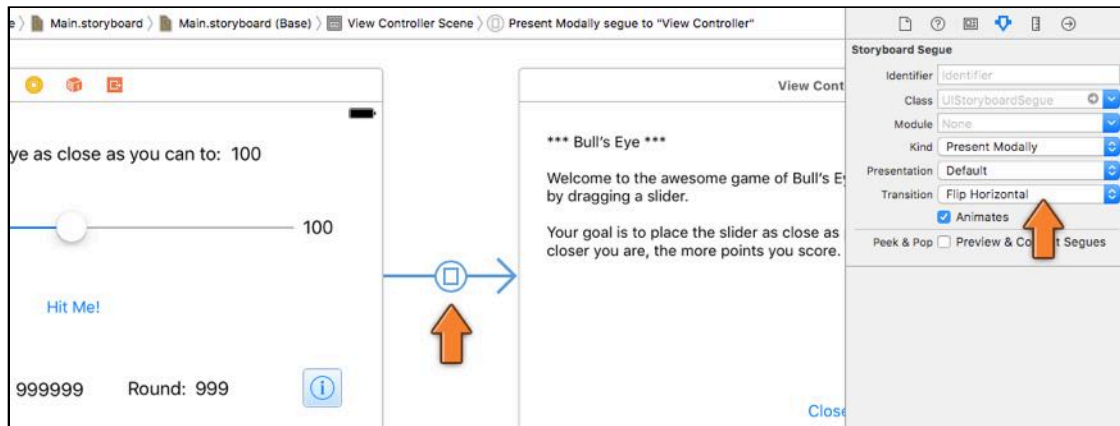
► Let go of the mouse button and a popup appears with several options. Choose

Present Modally.

Choosing the type of segue to create

Now an arrow will appear between the two screens. This arrow represents the segue.

► Click the arrow to select it. Segues also have attributes. In the **Attributes inspector**, choose **Transition, Flip Horizontal**. That is the animation that UIKit will use to move between these screens.



Changing the attributes for the segue

► Now you can run the app. Press the **(i)** button to see the new screen.



The About screen appears with a flip animation

The About screen should appear with a neat animation. Good, that seems to work. However, there is an obvious shortcoming here: tapping the Close button seems to

have no effect. Once the user enters the About screen she can never leave... that doesn't sound like good user interface design to me.

The problem with segues is that they only go one way. To close this screen, you have to hook up some code to the Close button. As a budding iOS developer you already know how to do that: use an action method!

This time you will add the action method to `AboutViewController` instead of `ViewController`, because the Close button is part of the About screen, not the main game screen.

► Open **AboutViewController.swift** and replace its contents with the following:

```
import UIKit

class AboutViewController: UIViewController {

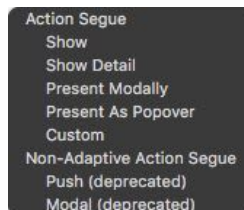
    @IBAction func close() {
        dismiss(animated: true, completion: nil)
    }
}
```

This code inside the `close()` action method tells UIKit to close the About screen with an animation.

If you would have said `dismiss(animated: false, ...)`, then there would be no page flip and the main screen would instantly reappear. From a user experience perspective, it's often better to show transitions from one screen to another with a subtle animation.

That leaves you with one final step, hooking up the Close button's Touch Up Inside event to this new `close` action.

► Open the storyboard and Ctrl-drag from the **Close** button to the About scene's View Controller. Hmm, strange, the **close** action should be listed in this popup, but it isn't. Instead, this is the same popup you saw when you made the segue:



The "close" action is not listed in the popup

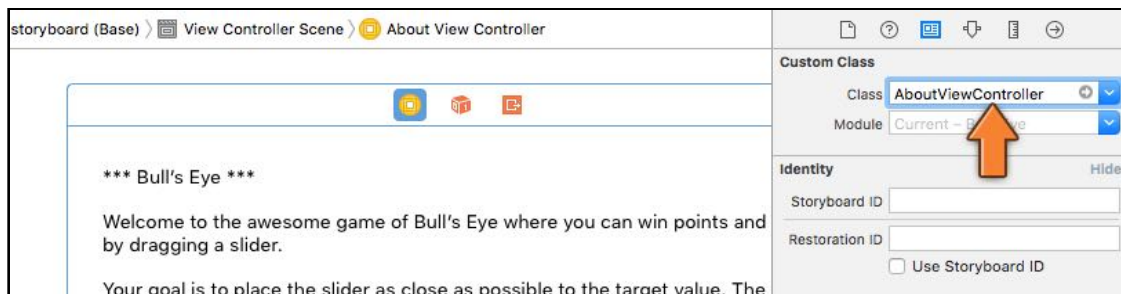
Exercise: Bonus points if you can spot the error. It's a very common – and frustrating! – mistake. ■

The problem is that this scene in the storyboard does not know yet that it is supposed to represent the `AboutViewController`.

You first added the `AboutViewController.swift` source file and then dragged a new view controller into the storyboard, but you haven't told the storyboard that the design for this new view controller, in fact, belongs to `AboutViewController`. (That's why in the outline pane it just says `View Controller` and not `About View Controller`.)

► Fortunately, this is easily remedied. In Interface Builder, select the **About** scene's **View Controller** and go to the **Identity inspector** (that's the button to the left of the Attributes inspector).

► Under **Custom Class**, type **AboutViewController**.



The Identity inspector for the About screen

Xcode should auto-complete this for you once you've typed the first few characters. If it doesn't, then double-check that you really have selected the `View Controller` and not one of the views inside it. (The view controller should also have a blue border to indicate it is selected.)

Now you should be able to connect the `Close` button to the action method.

► Ctrl-drag from the **Close** button to **About View Controller** in the outline pane. This should be old hat by now. The popup menu now does have an option for the **close** action (under **Sent Events**). Connect the button to that action.

► Run the app again. You should now be able to return from the `About` screen.

Congrats! This completes the game. All the functionality is there and – as far as I can tell – there are no bugs to spoil the fun.

But you have to admit the game still doesn't look very good. If you were to put this on the App Store in its current form, I'm not sure many people would be excited to download it. Fortunately, iOS makes it easy for you to create good-looking apps, so let's give `Bull's Eye` a makeover.

You can find the project files for the app up to this point under **06 - About Screen** in the tutorial's `Source Code` folder.

Making it look good

Apps in landscape mode do not display the iPhone status bar, unless you tell them to. That's great for our app. Games require a more immersive experience and the status bar detracts from that.

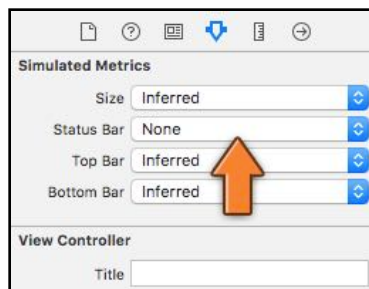
On iOS 7 and before, the status bar did not automatically disappear in landscape, and earlier editions of this tutorial included lengthy instructions on how to remove the status bar from the app.

Even though that is not required anymore, there are still a few things you can do to improve the way Bull's Eye handles the status bar.

First, you will remove the status bar from the storyboard.

► Open **Main.storyboard** and select the **View Controller**. Go to the **Attributes inspector** and under **Simulated Metrics** set **Status Bar** to **None**.

This removes the status bar from the storyboard (you should see the battery icon disappear from the top-right corner of both scenes).



Remove the status bar from the view controller

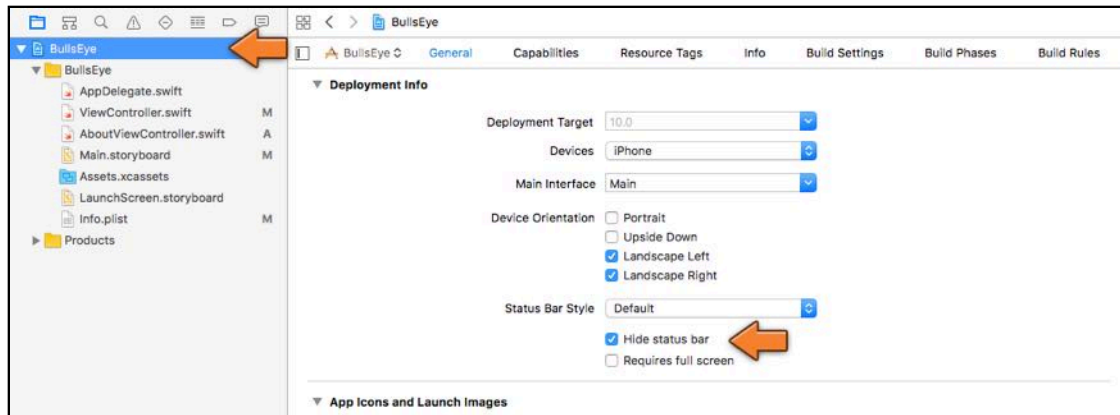
This setting has no influence on what happens when the app runs. That's why this section is labeled *Simulated Metrics*. Interface Builder merely pretends there is a status bar as a visual design aid, so you can see how your screen design looks with the status bar on top.

Try enabling some of the other simulated options and then run the app; you'll see that it won't make a difference.

The final step to get rid of the status bar forever is to make a change to the app's configuration.

► Go to the **Project Settings** screen and scroll down to **Deployment Info**. Under **Status Bar Style**, check the option **Hide status bar**.

This will also hide the status bar during application launch.



Hiding the status bar when the app launches

It's a good idea to hide the status bar while the app is launching. It takes a few seconds for the operating system to load the app into memory and start it up, and during that time the status bar remains visible, unless you hide it using this option.

It's only a small detail but the difference between a mediocre app and a great app is that great apps do all the small details right.

► That's it. Run the app and you'll see that the status bar is history.



Info.plist

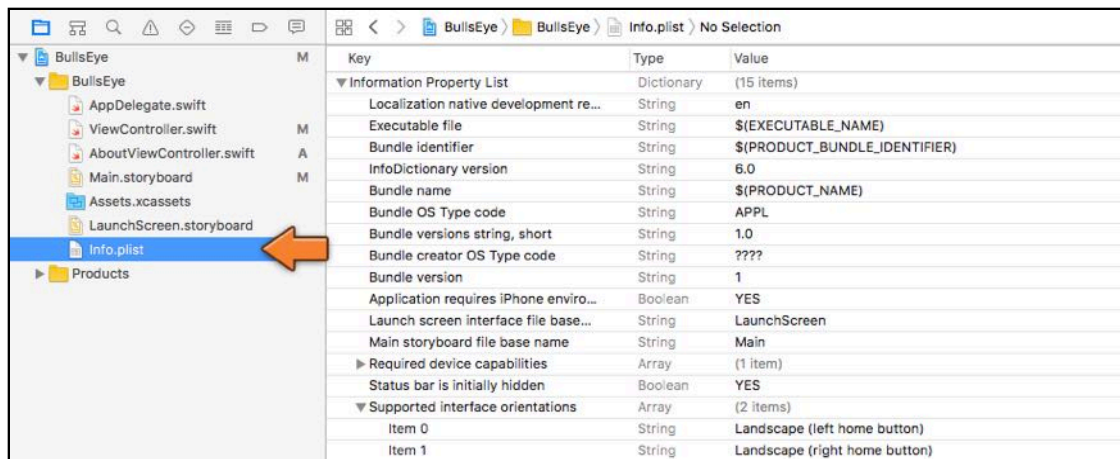
Most of the options from the Project Settings screen, such as the supported device orientations and whether the status bar is visible during launch, get stored in your app's Info.plist file.

Info.plist is a configuration file inside the application bundle that tells iOS how the app will behave. It also describes certain characteristics of the app that don't really fit anywhere else, such as its version number.

With previous versions of Xcode you often had to edit Info.plist by hand, but with Xcode 8 this is hardly necessary anymore. You can make most of the changes directly from the Project Settings screen.

However, it's good to know that Info.plist exists and what it looks like.

► Go to the **Project navigator** and select the file named **Info.plist** to take a peek at its contents.



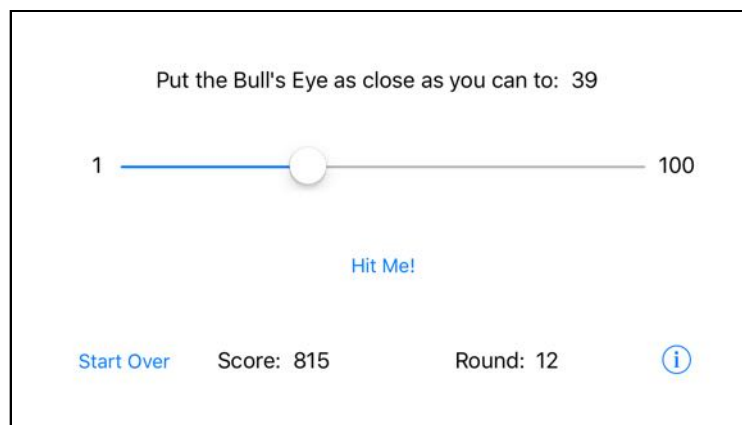
The Info.plist file is just a list of configuration options and their values. Most of these may not make sense to you, but that's OK – they don't always make sense to me either.

Notice the option **Status bar is initially hidden**. It has the value YES. This is the option that you just changed.



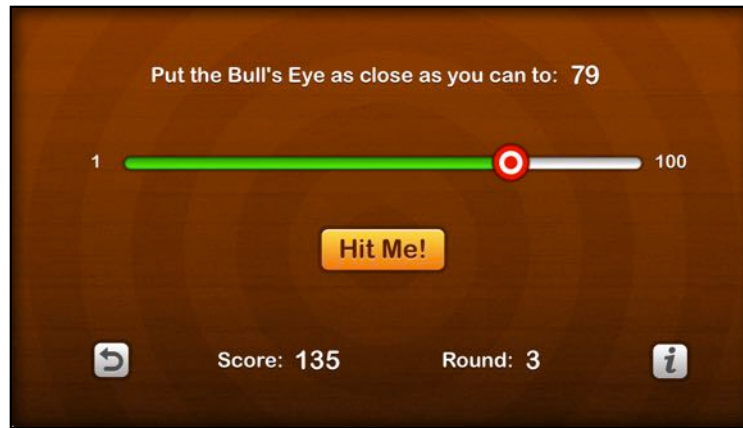
Spicing up the graphics

Getting rid of the status bar is only the first step. We want to go from this:



Yawn...

to something that's more like this:



Cool :-)

The actual controls don't change. You'll simply use images to smarten up their look, and you will also adjust the colors and typefaces.

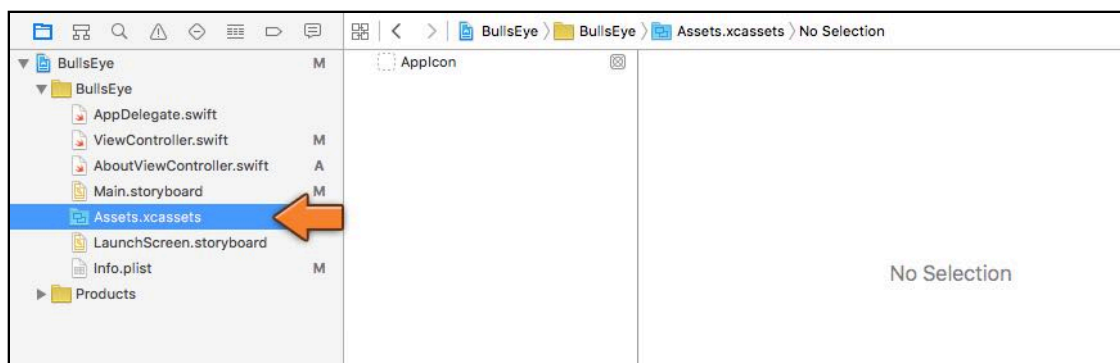
You can put an image in the background, on the buttons, and even on the slider, to customize their appearance. Images should be in PNG format.

If you are artistically challenged, then don't worry, I have provided a set of images for you. But if you do have mad Photoshop skillz, then by all means go ahead and design your own.

The Resources folder that comes with this tutorial contains a subfolder named Images. You will first import these images into the Xcode project.

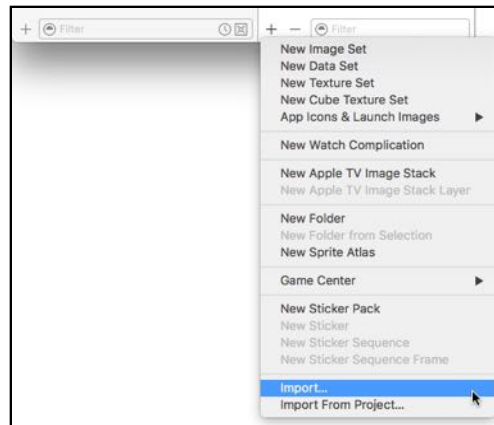
► In the **Project navigator**, find **Assets.xcassets** and click on it.

This is the so-called asset catalog for the app and it contains all the app's images. Right now, it is empty. Its only contents are placeholders for the app icon, which you'll add soon.



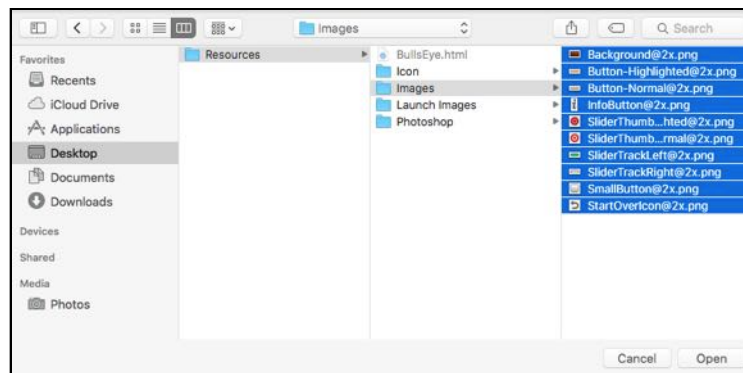
The asset catalog is initially empty

► At the bottom of the pane there is a **+** button. Click it and then select the option **Import...**



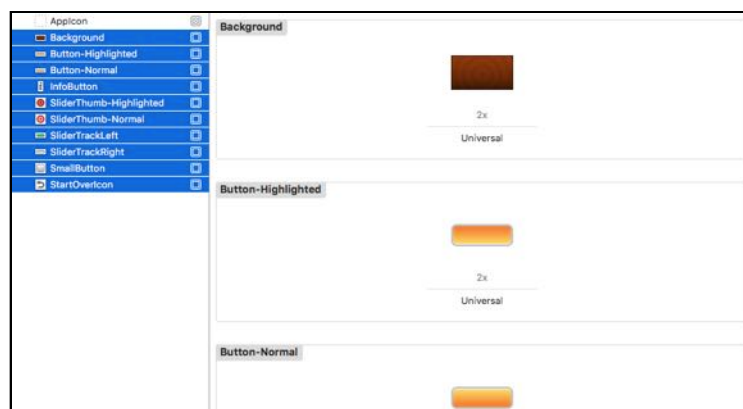
Choose Import to put existing images into the asset catalog

Xcode shows a file picker. Select the **Images** folder from this tutorial's resources and press **⌘+A** to select all the files inside this folder.



Choosing the images to import

Click **Open** and Xcode copies all the image files from that folder into the asset catalog:



The images are now inside the asset catalog

If Xcode added a folder named “Images” instead of the individual image files, then try again and this time make sure that you select the files inside the Images folder rather than the folder itself before you click Open.



1x, 2x, and 3x displays

Currently each image set in the asset catalog has only one slot for a “2x” image, but you can also specify 1x and 3x images. Having multiple versions of the same image in varying sizes allows your apps to support the wide variety of iPhone and iPad displays in existence.

1x is for low-resolution screens, the ones with the big, chunky pixels. There are no low-resolution devices in existence that can actually run iOS 10 – they are too old to bother with – so you’re not likely to come across many 1x images anymore. 1x is only a concern if you’re working on an app that still needs to support iOS 9 or even iOS 8.

2x is for high-resolution Retina screens. This covers most modern iPhones, iPod touches, and iPads. Retina images are twice as big as the low-res images, hence the 2x. The images you imported just now are 2x images.

3x is for the super high-resolution Retina HD screen of the iPhone 6s Plus and 7 Plus. If you want your app to have extra sharp images on these top-of-the-line iPhone models, then you can drop them into the “3x” slot in the asset catalog.

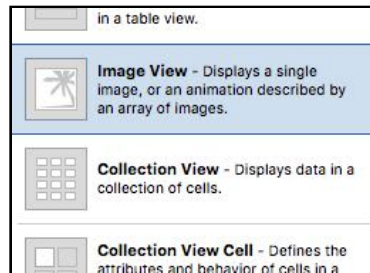
There is a special naming convention for image files. If the filename ends in **@2x** or **@3x** then that’s considered the Retina or Retina HD version. Low-resolution 1x images have no special name (you don’t have to write @1x).



Putting up the wallpaper

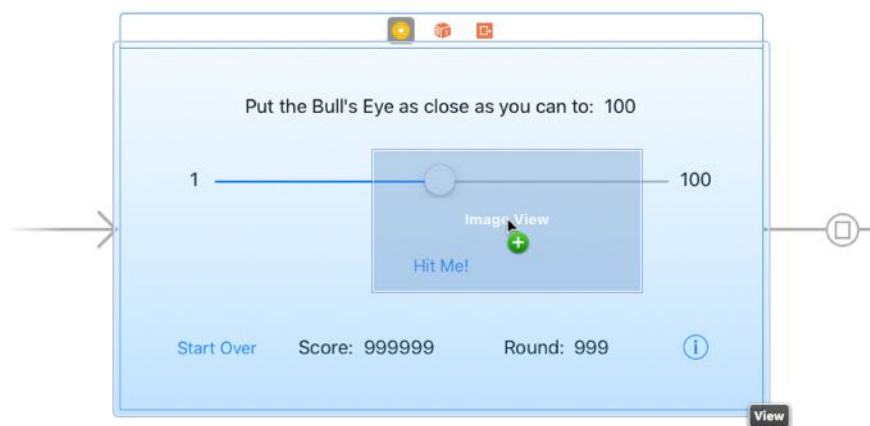
Let’s begin by changing the drab white background into something more fancy.

➤ Open **Main.storyboard**. Go into the **Object Library** and locate an **Image View**. (Tip: if you type “image” into the search box at the bottom of the Object Library, it will quickly filter out all the other views.)



The Image View control in the Object Library

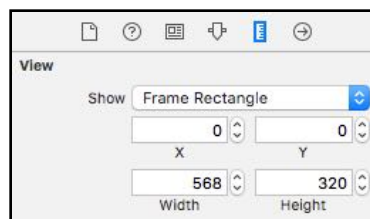
► Drag the image view on top of the existing user interface. It doesn't really matter where you put it, as long as it's inside the Bull's Eye View Controller.



Dragging the Image View into the view controller

► With the image view still selected, go to the **Size inspector** (that's the one next to the Attributes inspector) and set X and Y to 0, Width to 568 and Height to 320.

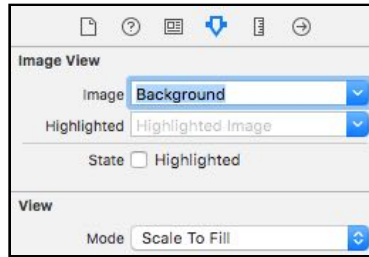
This will make the image view cover the entire screen.



The Size inspector settings for the Image View

► Go to the **Attributes inspector** for the image view. At the top there is an option named **Image**. Click the downward arrow and choose **Background** from the list.

This will put the image from the asset catalog's "Background" group into the image view.



Setting the background image on the Image View

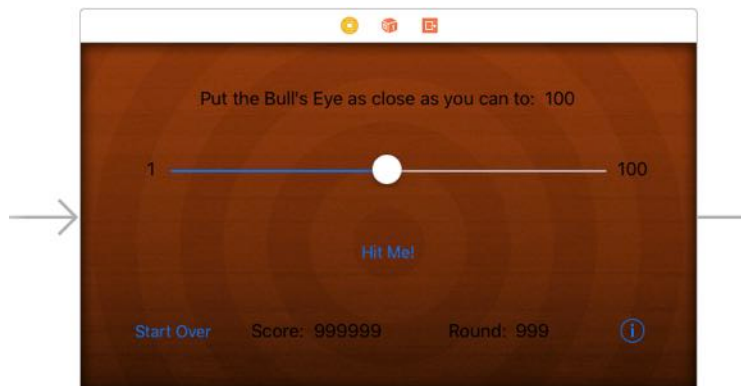
There is only one problem: the image now obscures all the other controls. There is an easy fix for that; you have to move the image view behind the other views.

► In the **Editor** menu in Xcode's menu bar at the top of the screen, choose **Arrange → Send to Back**.

Sometimes Xcode gives you a hard time with this (it still has a few bugs). If so, try de-selecting the Image View and then selecting it again. Now the Send to Back menu item should be available.

Alternatively, pick up the image view in the outline pane and drag it to the top, just below View, to accomplish the same thing.

Your interface should now look something like this:



The game with the new background image

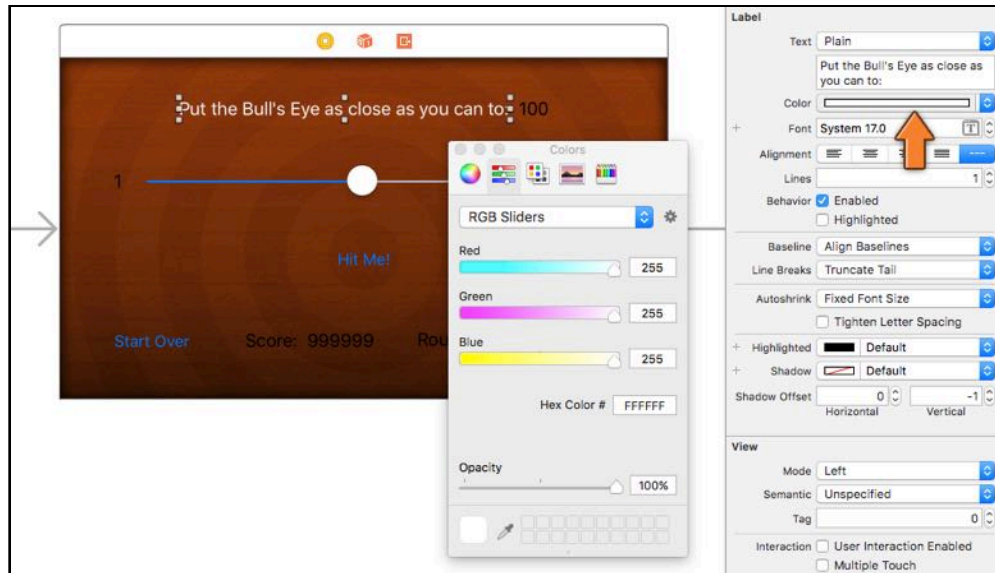
► Do the same thing for the **About View Controller**. Add an Image View and give it the same "Background" image.

That takes care of the background. Run the app and marvel at the new graphics.

Changing the labels

Because the background image is quite dark, the black labels have become hard to read. Fortunately, Interface Builder lets you change their color, and while you're at it you might change the font as well.

- Still in the storyboard, select the label at the top, open the **Attributes inspector** and click on the **Color** item.



Setting the text color on the label

This opens the Color Picker. It has several ways to select colors. I prefer the sliders (second tab). If all you see is a gray scale slider, then select RGB Sliders from the select box at the top.

- Pick a pure white color, Red: 255, Green: 255, Blue: 255, Opacity: 100%.
- Click on the **Shadow** item from the Attributes inspector. This lets you add a subtle shadow to the label. By default this color is transparent (also known as "Clear Color") so you won't see the shadow. Using the Color Picker, choose a pure black color that is half transparent, Red: 0, Green: 0, Blue: 0, Opacity: 50%.

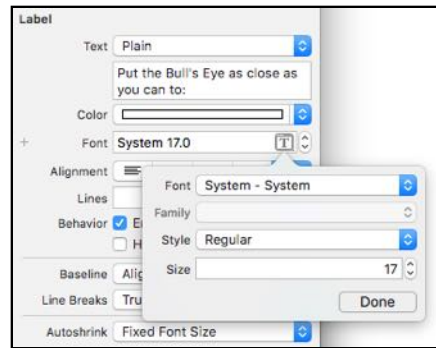
Note: Sometimes when you change the Color or Shadow attributes, the background color of the view also changes. This is a bug in Xcode. Put it back to Clear Color when that happens.

- Change the **Shadow Offset** to Horizontal: 0, Vertical: 1. This puts the shadow below the label.

The shadow you've chosen is very subtle. If you're not sure that it's actually visible, then toggle the vertical offset between 1 and 0 a few times. Look closely and you should be able to see the difference. As I said, it's very subtle.

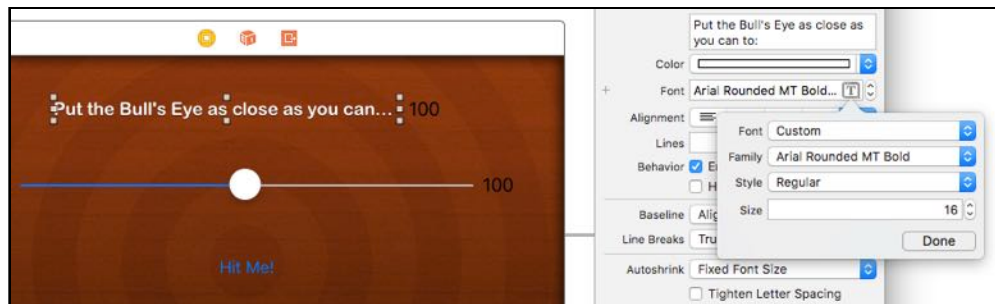
- Click on the **[T]** icon of the **Font** attribute. This opens the Font Picker.

By default the System font is selected. That uses whatever is the standard font for the user's device, which on iOS 10 is San Francisco. It's a nice font but we want something more exciting for this game.



Font picker with the System font

- Choose **Font: Custom**. That enables the Family field. Choose **Family: Arial Rounded MT Bold**. Set the Size to 16.



Setting the label's font

- The label also has an attribute **Autoshrink**. Make sure this is set to **Fixed Font Size**.

If enabled, Autoshrink will dynamically change the size of the font if the text is larger than will fit into the label. That is useful in certain apps, but not in this one. Instead, you'll change the size of the label to fit the text rather than the other way around.

- With the label selected, press **⌘=** on your keyboard, or choose **Size to Fit Content** from the **Editor** menu.

(If the Size to Fit Content menu item is disabled, then de-select the label and select it again. Sometimes Xcode gets confused about what is selected. Poor thing.)

The label will now become slightly larger or smaller so that it fits snugly around the text. If the text got cut off when you changed the font, now it will completely show again.

You don't have to set these properties for the other labels one by one; that would be a big chore. You can speed up the process by selecting multiple labels and then applying these changes to that entire selection.

- Click on the **Score:** label to select it. Hold **⌘** and click on the **Round:** label. Now

both labels will be selected. Repeat what you did above for these labels:

- Set Color to pure white, 100% opaque.
- Set Shadow to pure black, 50% opaque.
- Set Shadow Offset to 0 horizontal, 1 vertical.
- Set Font to Arial Rounded MT Bold, size 16.
- Make sure Autoshrink is set to Fixed Font Size.

As you can see, in my storyboard the text no longer fits into the Score and Round labels:

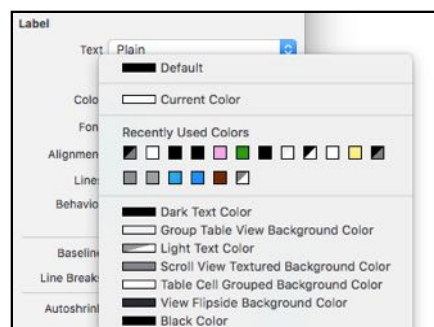


The font is too large to fit all the text in the Score and Round labels

You can either make the labels larger by dragging their handles to resize them manually, or you can use the **Size to Fit Content** option (⌘=). I prefer the latter because it's less work.

Tip: Xcode is smart enough to remember the colors you have used recently. Instead of going into the Color Picker all the time, you can simply choose a color from the Recently Used Colors menu.

Click the tiny arrows and the menu will pop up:

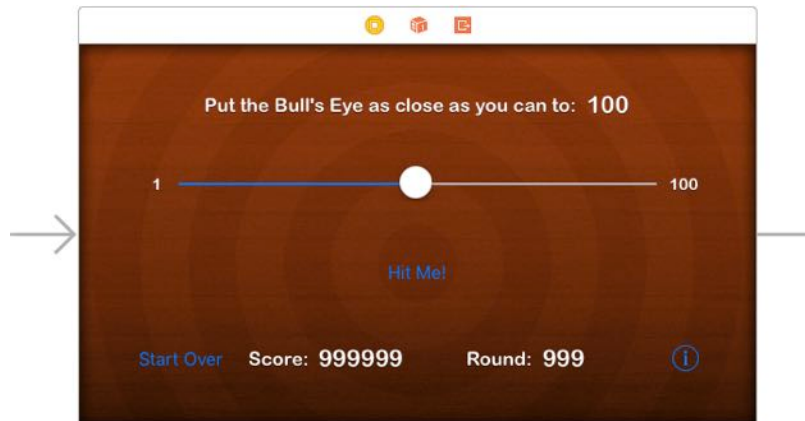


Quick access to recently used colors and several handy presets

Exercise: You still have a few labels to go. Repeat what you just did for the other labels. They should all become white, have the same shadow and have the same font. However, the two labels on either side of the slider (1 and 100) will have font size 14, while the other labels (the ones that will hold the target value, the score and the round number) will have font size 20 so they stand out more. ■

Because you've changed the sizes of some of the labels, your carefully constructed layout may have been messed up a bit. You may want to clean it up a little.

At this point, my screen looks like this:



What the storyboard looks like after styling the labels

All right, it's starting to look like something now. By the way, feel free to experiment with the fonts and colors. If you want to make it look completely different, then go right ahead. It's your app!

The buttons

Changing the look of the buttons works very much the same way.

- Select the **Hit Me** button. In the **Size inspector** set its Width to 100 and its Height to 37.
- Center the position of the button on the inner circle of the background image.
- Go to the **Attributes inspector**. Change **Type** from System to **Custom**.

A "system" button just has a label and no border. By making it a custom button, you can style it any way you wish.

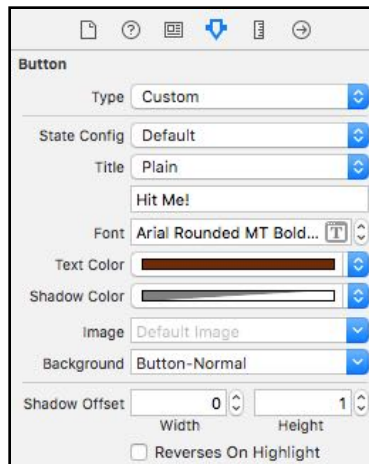
- Press the arrow on the **Background** field and choose **Button-Normal** from the list.
- Set the **Font** to **Arial Rounded MT Bold**, size 20.
- Set the **Text Color** to red: 96, green: 30, blue: 0, opacity: 100%. This is a dark brown color.
- Set the **Shadow Color** to pure white, 50% opacity. The shadow offset should be Width 0, Height 1 (for some reason they don't call it horizontal and vertical here).

Blending in

Setting the opacity to anything less than 100% will make the color slightly transparent (with opacity of 0% being fully transparent). Partial transparency makes the color blend in with the background and makes it appear softer.

Try setting the shadow color to 100% opaque pure white and notice the difference.

This finishes the setup for the Hit Me button in its “default” state:

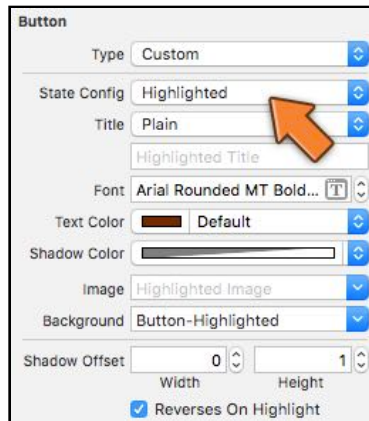


The attributes for the Hit Me button in the default state

Buttons can have more than one state. When you tap a button and hold it down, it should appear “pressed down” to let you know that the button will be activated when you lift your finger. This is known as the *highlighted* state and is an important visual clue to the user.

- With the button still selected, click the **State Config** setting and pick **Highlighted** from the menu. Now the attributes in this section reflect the highlighted state of the button.
- In the **Background** field, select **Button-Highlighted**.
- Make sure the highlighted **Text Color** is the same color as before (red 96, green 30, blue 0, or simply pick it from the Recently Used Colors menu). Change the **Shadow Color** to half-transparent white again.
- Check the **Reverses On Highlight** option. This will give the appearance of the label being pressed down when the user taps the button.

You could change the other properties too, but don’t get too carried away. The highlight effect should not be too jarring.



The attributes for the highlighted Hit Me button

To test the highlighted look of the button in Interface Builder you can toggle the **Highlighted** box in the **Control** section, but make sure to turn it off again or the button will initially appear highlighted when the screen is shown.

That's it for the Hit Me button. Styling the Start Over button is very similar, except you will replace its title text by an icon.

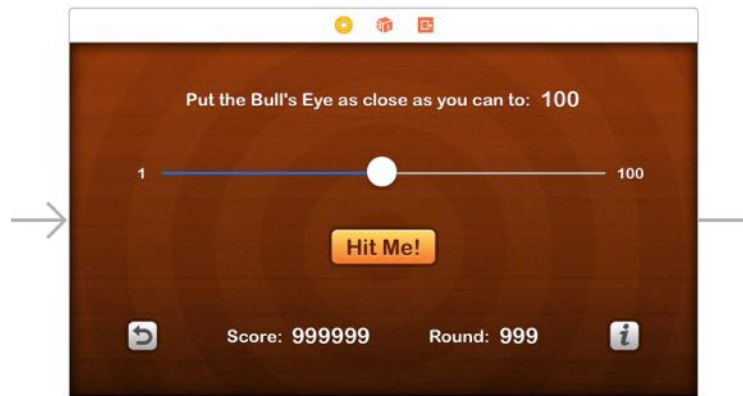
► Select the **Start Over** button and change the following attributes:

- Set Type to Custom.
- Remove the text "Start Over" from the button.
- For Image choose **StartOverIcon**
- For Background choose **SmallButton**
- Set Width and Height to 32.

You won't set a highlighted state on this button but let UIKit take care of this. If you don't specify a different image for the highlighted state, UIKit will automatically darken the button to indicate that it is pressed.

► Make the same changes to the **(i)** button, but this time choose **InfoButton** for the image.

The user interface is almost done. Only the slider is left to do...



Almost done!

The slider

Unfortunately, you can only customize the slider a little bit in Interface Builder. For the more advanced customization that this game needs – putting your own images on the thumb and the track – you have to resort to writing source code.

Everything you have done so far in Interface Builder you could also have done in code. Setting the color on a button, for example, can be done by sending the `setTitleColor()` message to the button.

However, I find that doing visual design work is much easier and quicker in a visual editor such as Interface Builder than writing the equivalent source code. But for the slider you have no choice.

► Go to **ViewController.swift**, and add the following to `viewDidLoad()`:

```
let thumbImageNormal = UIImage(named: "SliderThumb-Normal")!
slider.setThumbImage(thumbImageNormal, for: .normal)

let thumbImageHighlighted = UIImage(named: "SliderThumb-Highlighted")!
slider.setThumbImage(thumbImageHighlighted, for: .highlighted)

let insets = UIEdgeInsets(top: 0, left: 14, bottom: 0, right: 14)

let trackLeftImage = UIImage(named: "SliderTrackLeft")!
let trackLeftResizable =
    trackLeftImage.resizableImage(withCapInsets: insets)
slider.setMinimumTrackImage(trackLeftResizable, for: .normal)

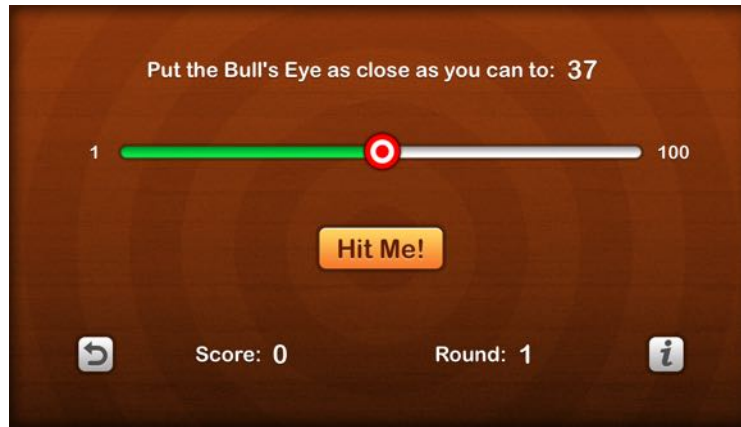
let trackRightImage = UIImage(named: "SliderTrackRight")!
let trackRightResizable =
    trackRightImage.resizableImage(withCapInsets: insets)
slider.setMaximumTrackImage(trackRightResizable, for: .normal)
```

This sets four images on the slider: two for the thumb and two for the track.

The thumb works like a button so it gets an image for the normal, un-pressed state and one for the highlighted state.

The slider uses different images for the track on the left of the thumb (green) and the track to the right of the thumb (gray).

► Run the app. You have to admit it looks pretty good now!



The game with the customized slider graphics

To .png or not to .png

If you recall, the images that you imported into the asset catalog had filenames like **SliderThumb-Normal@2x.png** and so on.

When you create a UIImage object, you don't use the original filename but the name that is listed in the asset catalog, **SliderThumb-Normal**.

That means you can leave off the **@2x** bit and the **.png** file extension.

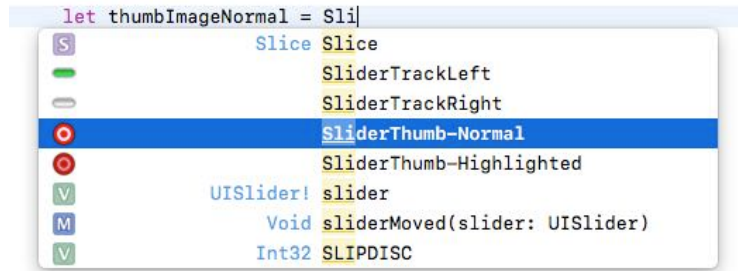
Tip: Xcode 8 has a handy new feature that makes it really easy to add images into your code. Instead of writing,

```
let thumbImageNormal = UIImage(named: "SliderThumb-Normal")
```

you can now type:

```
let thumbImageNormal = Sli
```

and then Xcode's autocomplete will show a list of suggestions to complete the text `Sli`, including any images whose names start with those letters.



Xcode autocomplete also shows images

Pick **SliderThumb-Normal** from the list and your code will look like this:

```
let thumbImageNormal = UIImage(SliderThumb-Normal)
slider.setThumbImage(thumbImageNormal, for: .normal)

let thumbImageHighlighted = UIImage(SliderThumb-Highlighted)
slider.setThumbImage(thumbImageHighlighted, for: .highlighted)

let insets = UIEdgeInsets(top: 0, left: 14, bottom: 0, right: 14)

let trackLeftImage = UIImage(SliderTrackLeft)
let trackLeftResizable = trackLeftImage.resizableImage(withCapInsets: insets)
slider.setMinimumTrackImage(trackLeftResizable, for: .normal)

let trackRightImage = UIImage(SliderTrackRight)
let trackRightResizable = trackRightImage.resizableImage(withCapInsets: insets)
slider.setMaximumTrackImage(trackRightResizable, for: .normal)
```

The images are now part of your source code

Give it a try! I really like that it shows a tiny thumbnail of the image right in your code.

Using a web view for HTML content

The About screen could still use some work.

Exercise: Change the Close button on the About screen to look like the Hit Me button. You should be able to do this by yourself now. Piece of cake! Refer back to the instructions for the Hit Me button if you get stuck. ■

► Now select the **text view** and press the **Delete** key on your keyboard. Yep, you're throwing it away.

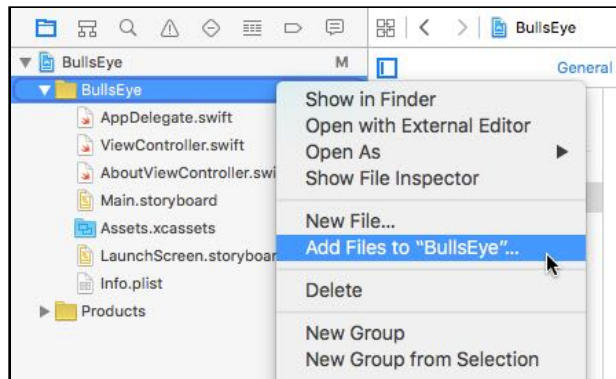
► Put a **Web View** in its place (as always, you can find this view in the Object Library).

A web view, as its name implies, can show web pages. All you have to do is give it a URL to a web site. The web view object is named `UIWebView`.

For this app you will make it display a static HTML page from the application bundle, so it won't actually have to go onto the web and download anything.

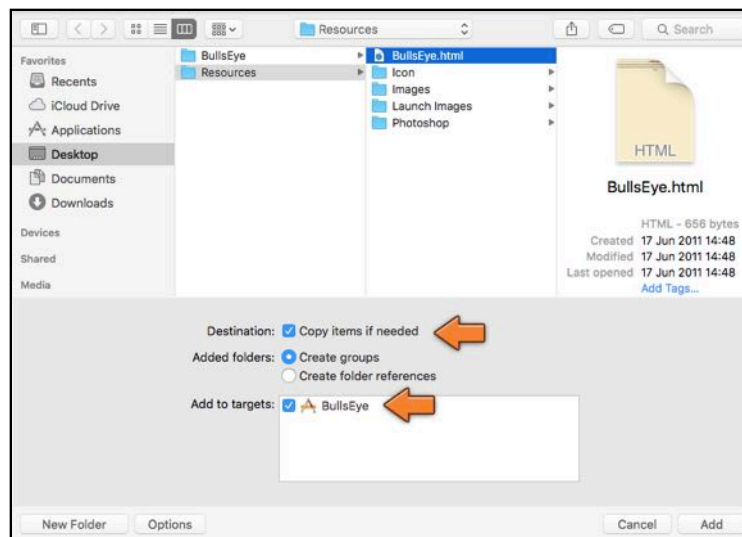
► Go to the **Project navigator** and right-click on the **BullsEye** group (the yellow

folder). From the menu, choose **Add Files to “BullsEye”...**



Using the right-click menu to add existing files to the project

► In the file picker, select the **BullsEye.html** file from the Resources folder. This is an HTML5 document that contains the gameplay instructions.



Choosing the file to add

Make sure that **Copy items if needed** is selected and that under **Add to targets**, there is a checkmark in front of **BullsEye**. (If you don't see these options, click the Options button at the bottom.)

► Press **Add** to add the HTML file to the project.

► In **AboutViewController.swift**, add an outlet for the web view:

```
class AboutViewController: UIViewController {
    @IBOutlet weak var webView: UIWebView!
    . . .
}
```


► In the storyboard file, connect the `UIWebView` element to this new outlet. The easiest way to do this is to Ctrl-drag from **About View Controller** (in the Outline pane) to the **Web View**.

(If you do it the other way around, from the Web View to About View Controller, then you'll connect the wrong thing and the web view will stay empty when you run the app.)

► In **AboutViewController.swift**, add the `viewDidLoad()` method:

```
override func viewDidLoad() {  
    super.viewDidLoad()  
  
    if let url = Bundle.main.url(forResource: "BullsEye",  
                                withExtension: "html") {  
        if let htmlData = try? Data(contentsOf: url) {  
            let baseUrl = URL(fileURLWithPath: Bundle.main.bundlePath)  
            webView.load(htmlData, mimeType: "text/html",  
                        textEncodingName: "UTF-8", baseUrl: baseUrl)  
        }  
    }  
}
```

This loads the local HTML file into the web view.

The source code may look scary but what goes on is not really that complicated: first it finds the **BullsEye.html** file in the application bundle, then loads it into a `Data` object, and finally it asks the web view to show the contents of this data object.

► Run the app and press the info button. The About screen should appear with a description of the gameplay rules, this time in the form of an HTML document:



The About screen in all its glory

Supporting 3.5-inch screens

So far you have designed the app for the 4-inch screen of the iPhone 5, 5c, 5s, and iPhone SE – still some of the most popular iPhone models in use today.

Previous versions of this tutorial showed how to make Bull’s Eye work on the iPhone 4S. This older model has a slightly smaller screen, measuring only 3.5 inches.

Both types of screen are equally wide (320 points), but where the 4-inch Retina phones are 568 points tall, the 3.5-inch models have only 480 points. That’s a difference of 88 points that your apps have to compensate for somehow.

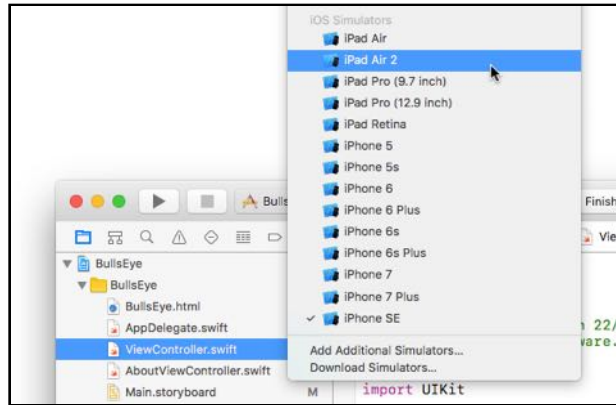
However, this point is now moot... iOS 10 does not support those older 3.5-inch devices anymore. Notice the absence of the iPhone 4S in the list of Simulators at the top of the Xcode window – you can no longer run your app on the iPhone 4S, not even a simulated one.

That said, it’s still useful to learn how to make the app work properly on these smaller screens, for a few reasons:

1. When you run Bull’s Eye on the iPad, it uses the 3.5-inch dimensions. The iPad can run all iPhone apps in a special emulation mode, but needs to use the 3.5-inch form factor because of screen size limitations.
2. If you get a job as an iOS developer you may need to support older versions of iOS. Older devices such as the iPhone 4S can still run iOS 9 or iOS 8.
3. It gives us a good excuse to learn about *Auto Layout*, a core UIKit technology that makes it easy to support many different screen sizes in your apps, including the larger screens of the iPhone 6 and 7, and iPad.

So even though iOS 10 doesn’t officially support the iPhone 4S’s smaller screen anymore, let’s make it work with Bull’s Eye anyway.

► To see what the app looks like on a 3.5-inch screen, run the app on the iPad simulator (I used **iPad Air 2**). You can switch between Simulators using the selector at the top of the Xcode window:



Using the scheme selector to switch to the iPad Simulator

As you may have expected, a portion of the screen gets cut off:



On the iPad Simulator, the app doesn't fill up the entire screen

Obviously, this won't do. Quite a few people use their iPads to play iPhone games, and the last thing you want is half your game's screen to get cut off!

Tip: You can press **⌘1** to **⌘5** to scale the iPad Simulator if it doesn't fit on your screen, or use the **Window** → **Scale** menu item. That iPad is a monster!

Universal apps

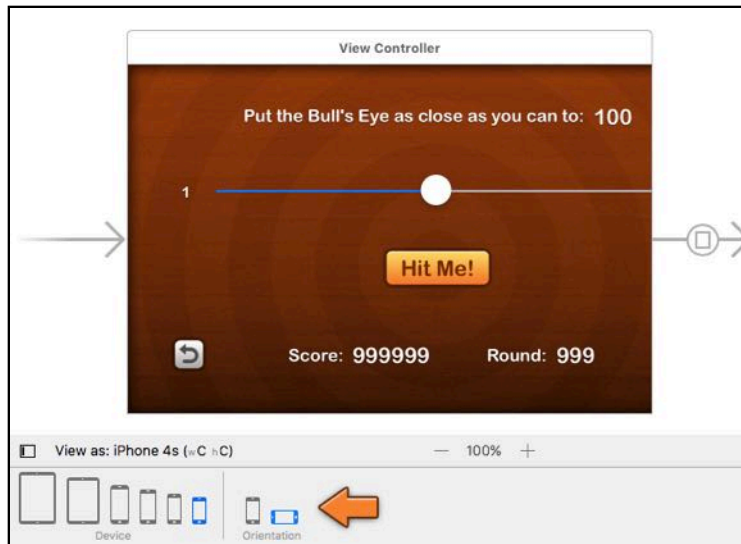
Many apps are *universal*, meaning they support both the iPhone and the iPad. When run on the iPad, universal apps properly take advantage of the iPad's larger screen.

Bull's Eye is not a universal app. For those kinds of apps the iPad acts as if it were an iPhone but shows everything twice as big. With the exception of the 12.9-inch iPad Pro, iPads only have enough pixels to emulate a 3.5-inch phone.

It would be better to make Bull's Eye a true universal app but that's a bit too involved for this tutorial. You'll learn all about the iPad and universal apps in tutorial 4, StoreSearch.

Interface Builder has handy tools to help you make the game fit on the 3.5-inch screen.

► Go to **Main.storyboard**. Open the **View as:** panel at the bottom and choose the smallest device. (You may need to change the orientation back to landscape.)



Viewing the storyboard in 3.5-inch

The storyboard should look just like the app in the iPad Simulator, with the right portion cut off. Now you can see how changes on the storyboard affect the smaller iPhone 4S screen.

First, let's fix the background image. The image is 568 points wide but at 3.5-inch we only have room for 480 points, so the image appears off-center.

This is where Auto Layout comes to the rescue.

► In the storyboard, select the **Background image view** on the main **View Controller** and click the small **Align** button at the bottom of the Xcode window:



The Align button

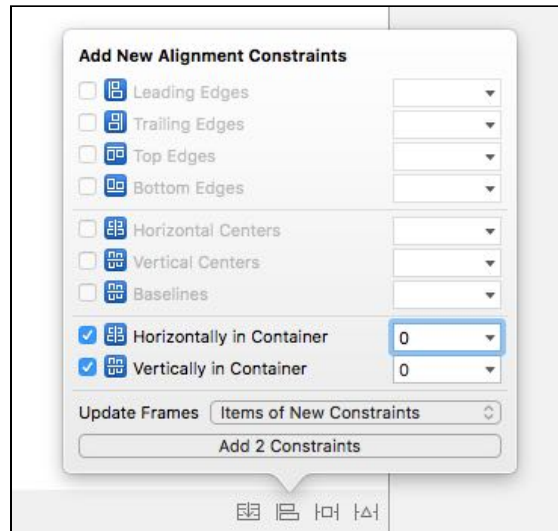
This menu lets you align a view with respect to the other views in the scene.

It looks best if you position the background image so that the rings in the wood are always in the center of the screen. The way to do this with Auto Layout is to create

two alignment constraints, one horizontal and one vertical.

The way you use Auto Layout is by defining relationships between your different views, the so-called *constraints*. When you run the app, UIKit evaluates these constraints and calculates the final layout of the views. This probably sounds a bit abstract, but you'll see soon enough how it works in practice.

► In the **Align** menu, put checkmarks in front of **Horizontally in Container** and **Vertically in Container**:

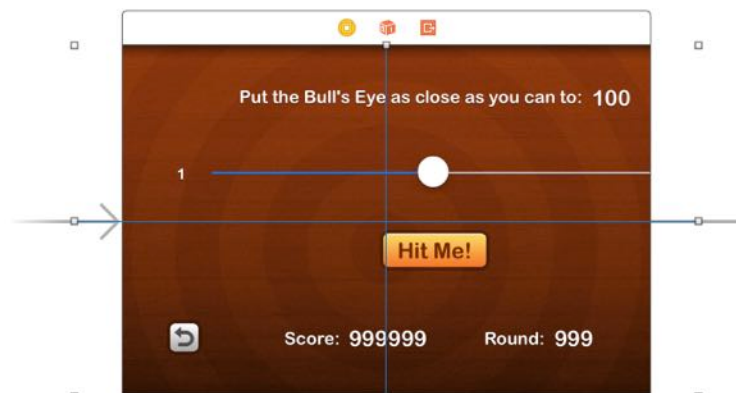


Using the Align menu to center the background image

► Change **Update Frames** to **Items of New Constraints**.

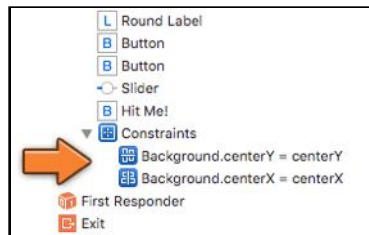
► Press **Add 2 Constraints** to finish. The rings are now properly centered. (Press Undo and Redo a few times to see the difference.)

The new alignment constraints are drawn as blue bars crossing the scene:



The blue bars represent the alignment constraints

In the Outline pane there is also a new item called **Constraints**:



The new Auto Layout constraints appear in the Outline pane

There should be two constraints listed here, one for **Background.centerX** and one for **Background.centerY**.

Note: Depending on exactly where you view these constraints in Xcode, they may also be called "Align Center X" (for horizontal), and "Align Center Y" (for vertical).

► Run the app again on the iPad Simulator and also on the 4-inch iPhone SE Simulator. In both cases, the background should be perfectly centered

If you use the **View as:** panel to switch the storyboard back to the iPhone SE, the background should be perfectly centered there too.

Let's repeat this for the About screen.

► Use the **Align menu** to add the two alignment constraints to the About screen's background image view.

The background image should now be centered. Of course, the Close button and web view are still completely off.

► In the storyboard, drag the **Close** button so that it snaps to the center of the view as well as the bottom guide.

Interface Builder shows a handy guide, the dotted blue line, near the edges of the screen, which is useful for aligning objects by hand. (You may need to move the web view out of the way a bit to make it easier to snap the button.)



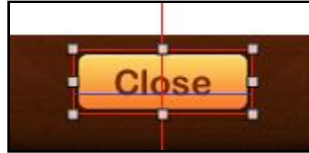
The dotted blue lines are guides that help position your UI elements

Like before, you want to create a centering constraint that keeps the Close button

in the middle of the screen, regardless of how wide the screen is.

➤ Click the **Close** button to select it. From the **Align** menu, choose **Horizontally in Container** and click **Add 1 Constraint**.

Interface Builder now draws a red bar to represent the constraint, and a red box around the button as well.



The Close button has red constraints

That's a problem: the bars are all supposed to be blue, not red. Red indicates that something is wrong with the constraints, usually that there aren't enough of them.

For each view there must always be enough constraints to define both its position and its size. The Close button already knows its size – you typed this into the Size inspector earlier – but for its position there is only a constraint for the X-coordinate (the alignment in the horizontal direction). You also need to add a constraint for the Y-coordinate.

There are different types of constraints. So far you've used alignment constraints but there are also "spacing" constraints that make sure the sides of two views stay glued together with a certain amount of spacing between them. You make those spacing constraints with the Pin menu.

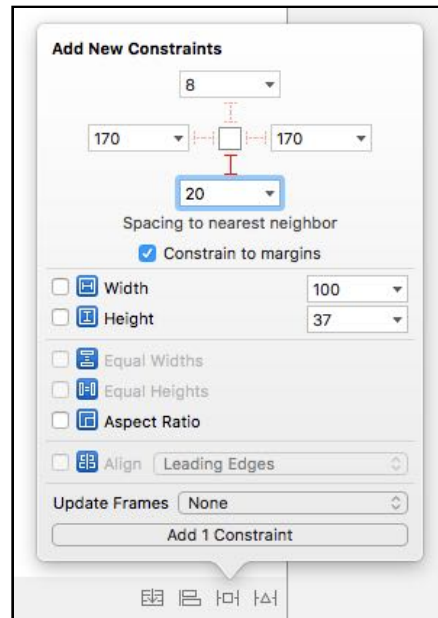
➤ With the **Close** button still selected, click on the **Pin button** at the bottom of the window:



The Pin button for adding spacing constraints

This menu lets you "pin" a view to its neighboring views. For the Close button, you want it to always sit at a distance of 20 points from the bottom of the screen, so that's where you'll pin it.

➤ In the **Pin menu**, in the **Spacing to nearest neighbor** section, there are four bars that represent the four sides of the view that can be pinned. Because you want to pin the bottom of the Close button, select that bar to make it fully red.



The red bars decide the sides that become pinned down

► Make sure the spacing is **20** and then click **Add 1 Constraint** to finish.

Now the constraints all turn blue, meaning that everything is OK:



The constraints on the Close button are valid

If at this point you don't see blue bars but orange ones, then something's still wrong with your Auto Layout constraints:

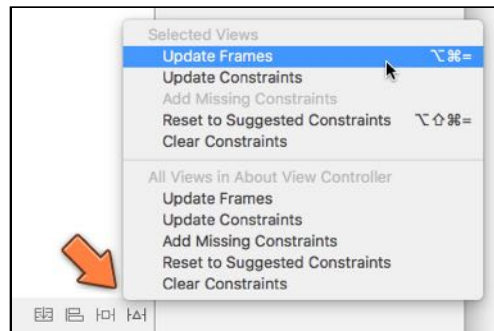


The views are not positioned according to the constraints

This happens when the constraints are valid (otherwise the bars would be red) but the view is not in the right place in the scene. The dashed orange box off to the side is where Auto Layout has calculated the view should be instead, based on the constraints you have given it.

To fix this issue, select the **Close** button again and from the **Resolve Auto Layout**

Issues menu choose **Update Frames**:



Resolving Auto Layout issues

The Close button should now always be perfectly centered, regardless of whether you're on the 3.5-inch or the 4-inch Simulator.

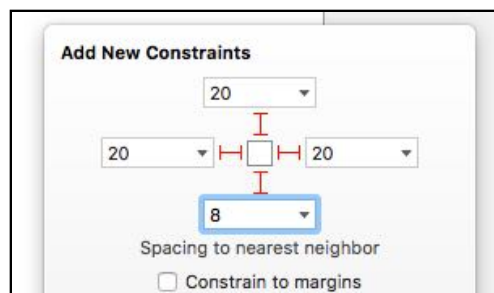
Note: What happens if you don't add any constraints to your views? In that case, Xcode will automatically add constraints when it builds the app. That is why you didn't need to bother with any of this before.

However, these default constraints may not always do what you want. For example, they will not automatically resize your views to accommodate the smaller 3.5-inch screen. If you want that to happen, then it's up to you to add your own constraints. (Auto Layout can't read your mind!)

As soon as you add just one constraint to a view, Xcode will no longer add any other automatic constraints to that view. From then on you're responsible for adding enough other constraints so that UIKit always knows what the position and size of the view will be.

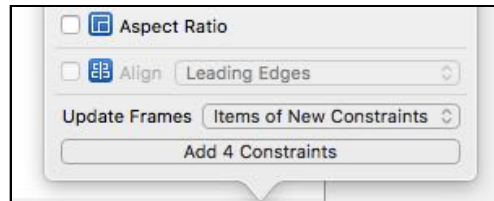
There is one thing left to fix in the About screen and that is the web view.

➤ Select the **Web View** and open the **Pin menu**. First, make sure **Constrain to margins** is unchecked. Then click all four bars so they become solid red and set their spacing to 20 points, except the bottom one which is 8 points:



Creating the constraints for the web view

► For **Update Frames** choose **Items of New Constraints**. This will resize the web view to the proper size, based on the constraints you chose.

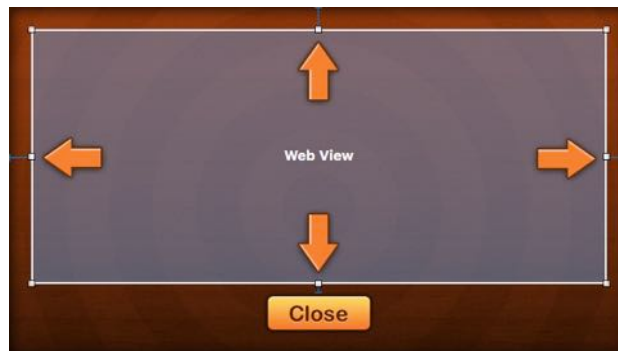


Choosing the correct option for Update Frames

Without this setting, Xcode may display orange bars to complain that the size and position of the web view do not correspond with the constraints you're adding. You can fix that afterwards with the Resolve Auto Layout Issues button, but why bother when Xcode can fix it for you?

► Finish by clicking **Add 4 Constraints**.

There are now four constraints on the web view (the blue bars):



The four constraints on the web view

Three of these pin the web view to the main view, so that it always resizes along with it, and one connects it to the Close button. This is enough to determine the size and position of the web view in any scenario.

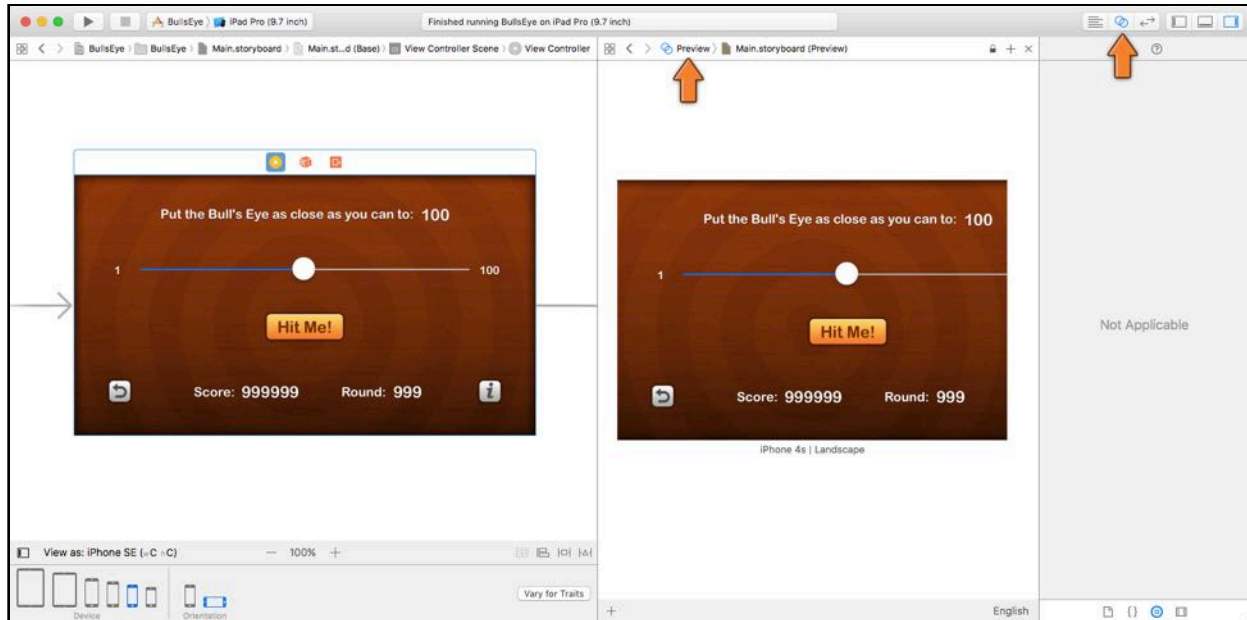
Back to the main game scene, which still needs some work to fit on the smaller screen size.

First, you'll clean up the storyboard by dragging all the controls over to the left so that they fit tidily on the 3.5-inch screen. This is a bit tricky in 3.5-inch mode because some of the buttons and labels are positioned outside the visible area, making it impossible to pick them up and move them around.

Fortunately, Interface Builder has a handy preview pane that can help with this.

► Use the **View as:** panel to switch back to the 4-inch iPhone SE. Now all the labels and buttons are visible again.

- Make sure the main **View Controller** is selected.
- Click the button with the two overlapping circles in the Xcode toolbar to open the **Assistant editor**.
- In the jump bar choose **Preview** and then **Main.storyboard (Preview)**. (You may need to click around a bit before this option becomes visible.)



Enabling the preview assistant in Interface Builder

The screen is now split in two. On the left is the storyboard; on the right is a preview pane that shows how the app will look on different iPhone devices.

If not everything fits on your screen at once, you can make some room by hiding the navigator and utilities panes with the buttons from the toolbar. You can also collapse Interface Builder's Outline pane. (Or buy an extra 30" monitor!)

The preview assistant should currently show a 3.5-inch iPhone 4S in landscape. If not, then do the following:

- Select the preview so that it gets a blue border, and press Delete on your keyboard to remove it.
- Use the small **+** button at the bottom to select **iPhone 4s**. This adds a preview of the 3.5-inch phone, but in portrait.
- Hovering the mouse over the preview makes a rotation icon appear. Click the rotation icon to flip the preview to landscape:



The rotation icon toggles between portrait and landscape

Now the preview pane should look just like the app in the iPad Simulator, with the right portion of the game cut off.

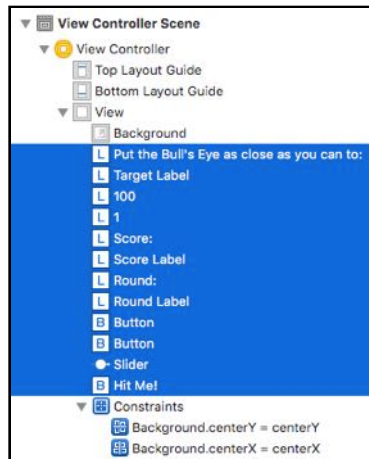
► In the storyboard (the left pane), move the labels, buttons, and slider so that everything looks good in the preview pane on the right. This is where the preview pane comes in real handy!



Everything is rearranged to fit on the smaller 3.5-inch screen

Of course, the game looks a bit lopsided now on 4-inch phones. You will fix that by placing all the labels, buttons and the slider into a new “container” view. Using Auto Layout, you’ll center that container view in the screen, regardless of how big the screen is.

► Select all the labels, buttons, and the slider. You can hold down ⌘ and click them individually but an easier method is to go to the **Outline pane**, click on the first view (for me that is the “Put the Bull’s Eye as close as you can to:” label), then hold down Shift and click on the last view (in my case the Hit Me button):



Selecting the views from the Outline pane

You should have selected everything but the background image view.

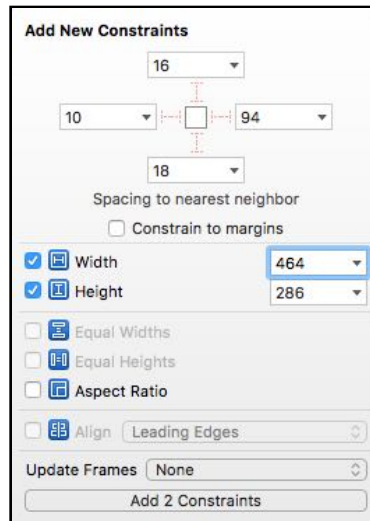
► From Xcode's menu bar, choose **Editor** → **Embed In** → **View**. This places the selected views inside a new container view:



The views are embedded in a new container view

This new view is completely white, which is not what you want eventually, but it does make it easier to add the constraints.

► Select the newly added **container view** and open the **Pin menu**. Put checkboxes in front of **Width** and **Height** in order to make constraints for them. Click **Add 2 Constraints** to finish.



Pinning the width and height of the container view

Interface Builder now draws several bars around the view that represent the Width and Height constraints that you just made, but they are red. Don't panic! It only means there are not enough constraints yet. No problem, you'll add the missing constraints next.

► With the container view still selected, open the **Align** menu. Check the **Horizontally in Container** and **Vertically in Container** options. For **Update Frames**, select **Items of New Constraints**. Click **Add 2 Constraints**.

All the Auto Layout bars should be blue now and the view is perfectly centered.

► Finally, change the **Background** color of the container view to **Clear Color** (in other words, 100% transparent).

You now have a layout that works correctly on both the 3.5-inch and 4-inch iPhones! Try it out:



The game running on 3.5-inch and 4-inch iPhones

Auto Layout may take a while to get used to. Adding constraints in order to position

UI elements is a little less obvious than just dragging them into place.

But this also buys you a lot of power and flexibility, which you need when you're dealing with devices that have different screen sizes.

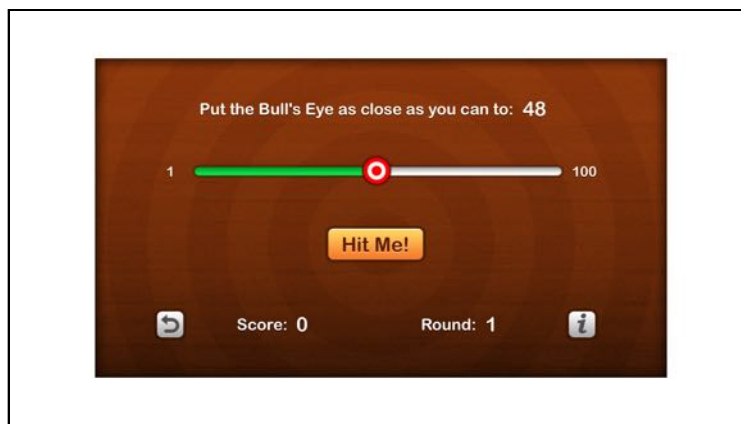
You'll learn more about Auto Layout in the other parts of *The iOS Apprentice*.

Supporting the iPhone 6 and up

Making the game work on smaller devices is one thing, but what about *larger* devices, such as the iPhone 6, 7, and Plus? The regular iPhone 6, 6s, and 7 have a 4.7-inch screen, while the Plus is a whopping 5.5-inches.

► Try it out! You can use **View as:** or the preview pane to look at the storyboard in 4.7-inch and 5.5-inch mode, or you can run the app in the iPhone 6s, iPhone 7, or Plus simulators.

What happened? This is what it looks like on the 7 Plus Simulator:



The game on the iPhone 6s Plus or 7 Plus

(Remember, you can press ⌘1 to ⌘5 to scale the Simulator if it doesn't fit on your screen. An iPhone Plus is almost as big as an iPad!)

Well, I guess it's not too bad – but it's not great either. The background image doesn't quite fit and the app is not taking advantage of all the available space. It would be better if everything were slightly bigger.

There are several ways to tackle this, but we're going to cheat and take the easy way out.

Apps need to opt-in to support the larger screens of the iPhone 6 and up.

If an app does *not* opt-in, the iPhone 6/7/Plus will automatically scale up the app to fill up the extra space. This is done so that older apps are still usable on these larger devices. That's great for us, because scaling up is exactly what we want – and with minimal effort.

Apps that do opt-in for iPhone 6 support must provide a so-called **launch screen**. You've already seen this launch screen in action, probably without knowing it.

Note: Starting up an app usually takes a short while. You can make the transition between tapping the app icon and actually using the app more seamless by using a launch screen. This is a placeholder that is shown while the app is being loaded.

Without this placeholder, the iPhone's screen will simply be blanked out until the app is ready, which isn't very welcoming.

A lot of developers abuse this feature to show a splash screen with a logo, but it's better for the user if you just show a static image of the user interface and not much else. Nobody likes to wait for apps to load and a well-chosen launch screen will give the illusion the app is loading faster than it actually is.

You can use a regular image but you can also use a storyboard file or a XIB file. A XIB, also known as "nib", is like a storyboard except that it can contain the design of only a single screen.

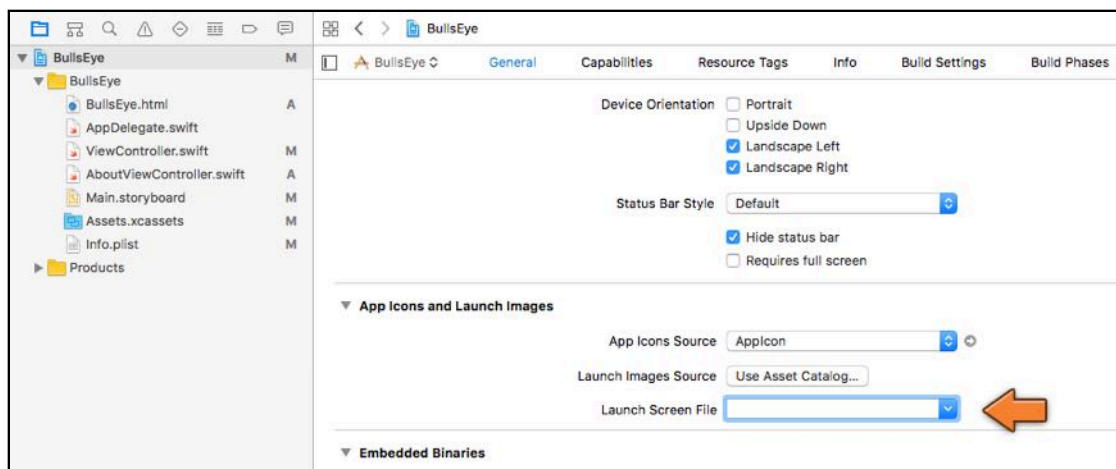
The design for the app's launch screen lives in the file **LaunchScreen.storyboard**. Currently that storyboard contains a completely empty view controller, resulting in a blank launch screen. You've looked at it every time you launched the app, but because it's completely white there wasn't much to see. (For fun, drag a label into this storyboard and see what happens when you run the app.)

To get the automatic scaling on the iPhone 6 and up, you need to remove this storyboard file. In other words, by removing the launch screen you're opting *out* of using the extra pixels from these larger devices. In order not to waste any screen space, UIKit will automatically scale up the game so that it fills the screen.

► In the **Project navigator**, select **LaunchScreen.storyboard** and press the **delete** key to remove it. When Xcode asks for confirmation, choose **Move to Trash**.

That alone is not enough. You also need to tell Xcode that it can no longer use this launch screen file.

► Go to the **Project Settings** screen. In the **App Icons and Launch Images** section, make the box for **Launch Screen File** empty:



The Launch Screen File field must be empty

➤ To completely wipe Xcode's memory of this launch screen file, hold down the **Alt/Option** key and choose **Product** → **Clean Build Folder** from the Xcode menu bar. Confirm by pressing **Clean**.

➤ Run the app. You should no longer see the launch screen. If you do, choose **Simulator** → **Reset Contents and Settings** from the Simulator menu bar to start over with a clean slate.

You might be in for a surprise. This is what the app looks like now on the iPhone 6s and 7 Simulators:



The app is letterboxed on the iPhone 6s and 7

There are two black bars on the sides. What you're seeing here is the app in 3.5-inch mode, but scaled up to the iPhone 6s's larger screen. Weird!

The solution is to add a 4-inch launch image to the project. This is not a XIB or storyboard file, just a static picture of the wood texture background.

➤ In the **Project navigator**, right-click the **BullsEye** group (the one with the yellow icon) and choose **Add Files to "BullsEye"** from the menu.

➤ Navigate to the **Launch Images** folder from this tutorial's resources and select

the **Default-568h@2x.png** file. This image is identical to the background image but turned sideways (launch images must always be in portrait orientation).

Make sure **Copy items if needed** is checked (click the Options button to reveal this option) and press **Add** to add the file to the project. That's all there is to it!

Run the app and notice that the transition into the app looks a lot smoother. It's little details like these that count. And best of all, the app now looks great on the iPhone 6s, 7, and Plus!

Note: Simply scaling up the app for the larger phones works well for *Bull's Eye*, but for most apps you'll want to take advantage of all that extra screen space. iOS has several features that help with this – Auto Layout and Size Classes – and you'll learn all about them in the next tutorials.

Crossfade

I can't conclude this tutorial before mentioning Core Animation. This technology makes it very easy to create really sweet animations in your apps, with just a few lines of code. Adding subtle animations (with emphasis on subtle!) can make your app a delight to use.

You will add a simple crossfade after the Start Over button is pressed, so the transition back to round one won't seem so abrupt.

► In **ViewController.swift**, add the following line at the top, right below the other import:

```
import QuartzCore
```

The Core Animation technology lives in its own framework, QuartzCore. With the import statement you tell the compiler that you want to use the objects from this framework.

► Change the `startOver()` method to:

```
@IBAction func startOver() {
    startNewGame()
    updateLabels()

    let transition = CATransition()
    transition.type = kCATransitionFade
    transition.duration = 1
    transition.timingFunction = CAMediaTimingFunction(name:
                                                         kCAMediaTimingFunctionEaseOut)
    view.layer.add(transition, forKey: nil)
}
```

The calls to `startNewGame()` and `updateLabels()` were there before, but the `CATransition` stuff is new.

I'm not going to go into too much detail here. Suffice it to say you're setting up an animation that crossfades from what is currently on the screen to the changes you're making in `startNewGame()` – reset the slider to center position – and `updateLabels()` – reset the values of the labels.

► Run the app and move the slider so that it is no longer in the center. Press the Start Over button and you should see a subtle crossfade animation.

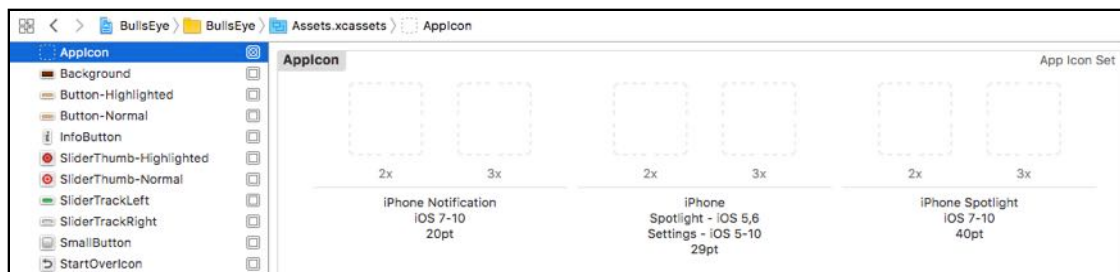


The screen crossfades between the old and new states

The icon

You're almost done with the app but there are still a few loose ends to tie up. You may have noticed that the app has a really boring white icon. That won't do!

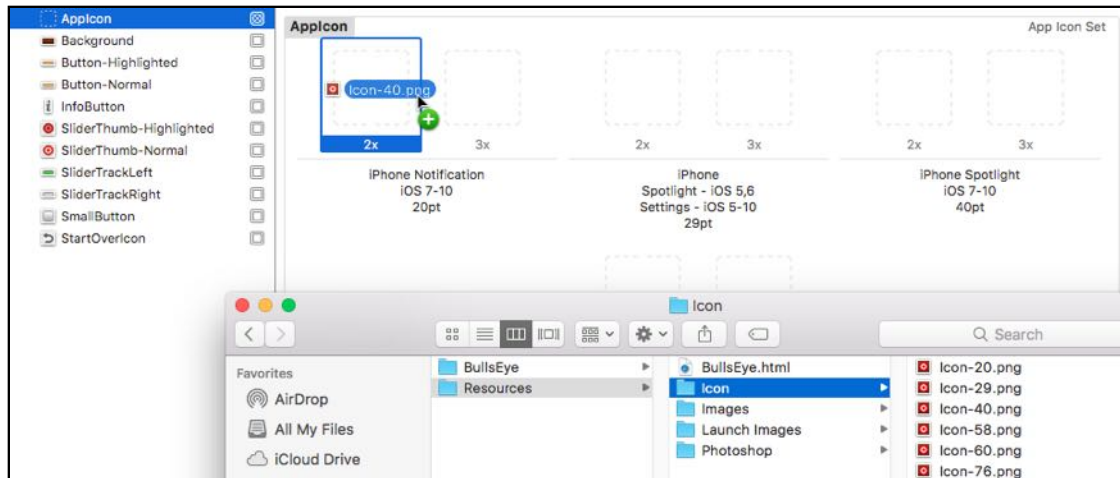
► Open the asset catalog (**Assets.xcassets**) and select **AppIcon**:



The AppIcon group in the asset catalog

This has eight slots for the different types of icons the app needs.

► In Finder, open the **Icon** folder from this tutorial's resources. Drag the **Icon-40.png** file into the first slot, **iPhone Notification 20pt**:



Dragging the icon into the asset catalog

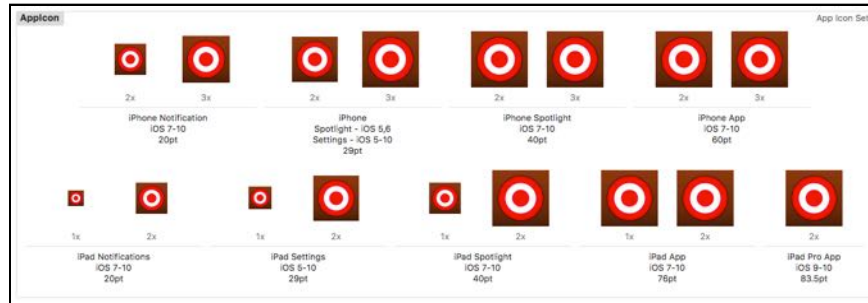
You may be wondering why you're dragging the **Icon-40.png** file and not **Icon-20.png** into the slot for 20pt. Notice that this slot says **2x**, which means it's for Retina devices and on Retina screens one point counts as two pixels.

- Drag the **Icon-60.png** file into the **3x** slot next to it. This is for the iPhone 6s Plus and 7 Plus with its 3x resolution.
- For **iPhone Spotlight & Settings 29pt**, drag the **Icon-58.png** file into the 2x slot and **Icon-87.png** into the 3x slot. (What, you don't know your times table for 29?)
- For **iPhone Spotlight 40pt**, drag the **Icon-80.png** file into the 2x slot and **Icon-120.png** into the 3x slot.
- For **iPhone App 60pt**, drag the **Icon-120.png** file into the 2x slot and **Icon-180.png** into the 3x slot.

That's four icons in two different sizes. Phew!

The other files in the folder are for the iPad. This app does not have an iPad version, but that doesn't prevent iPads from running it. All iPads can run all iPhone apps, but they show up in a smaller frame. To accommodate this, it's nicest if you also supply icons for iPad.

- With **AppIcon** still selected, in the **Attributes** inspector choose **iOS 7.0 and Later** for **iPad**. This adds nine new slots to the AppIcon set.
- Drag the icons into the proper slots. Notice that the iPad icons need to be supplied in 1x as well as 2x sizes (but not 3x). You may need to do some mental arithmetic here to figure out which icon goes into which slot!



The full set of icons for this app, including the iPad icons

The **Icon-1024.png** file is not used by the app. This is for submission to the App Store. As part of the app submission, you are required to upload a 1024×1024 pixel version of the icon.

► Run the app and close it. You'll see that the icon has changed on the Simulator's springboard. If not, remove the app from the Simulator and try again (sometimes the Simulator keeps using the old icon and re-installing the app will fix this).



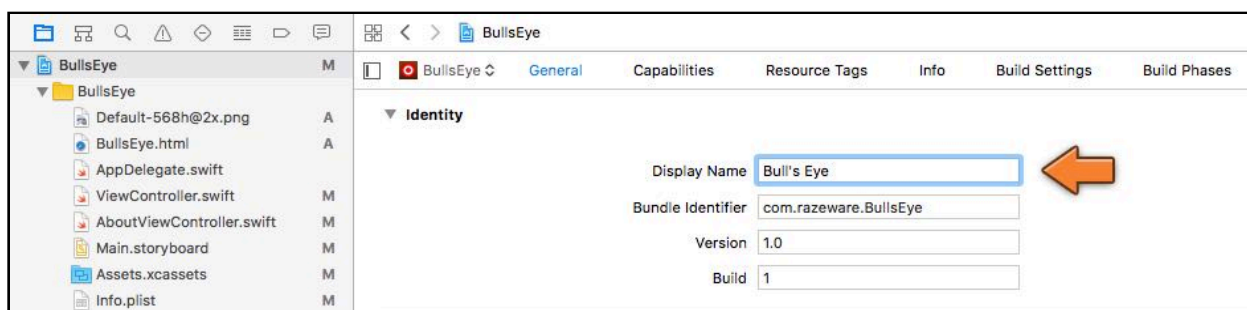
The icon on the Simulator's springboard

Display name

One last thing. You named the project **BullsEye** and that is the name that shows up under the icon. However, I'd prefer to spell it "**Bull's Eye**".

There is only limited space under the icon and for apps with longer names you have to get creative to make the name fit. For this game, however, there is enough room to add the space and the apostrophe.

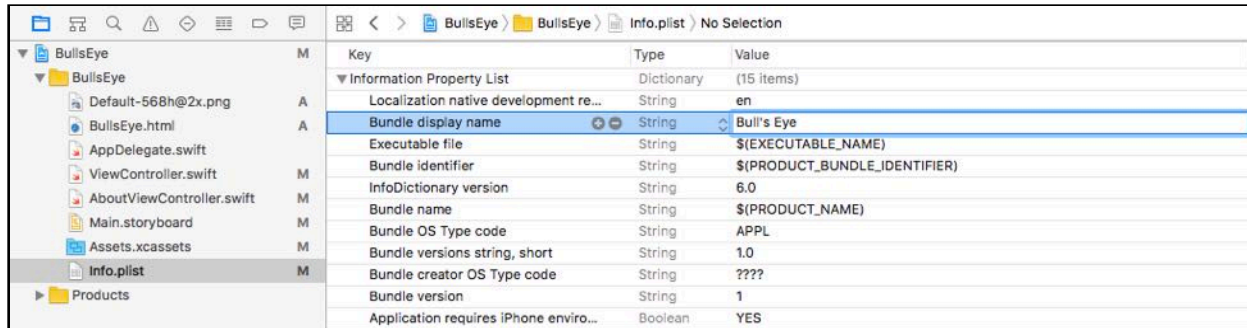
► Go to the **Project Settings** screen. The very first option is **Display Name**. Change this to **Bull's Eye**.



Changing the display name of the app

Like many of the project's settings you can also find the display name in the app's Info.plist file. Let's have a look.

► From the **Project navigator**, select **Info.plist**.



The display name of the app in Info.plist

The row **Bundle display name** contains the new name you've just given the app.

Note: If **Bundle display name** is not present, the app will use the value from the field **Bundle name**. That has the special value "\$(PRODUCT_NAME)", meaning Xcode will automatically put the project name, BullsEye, in this field when it adds the Info.plist to the application bundle. By providing a **Bundle display name** you can override this default name and give the app any name you want.

► Run the app and quit it to see the new name under the icon.



The bundle display name setting changes the name under the icon

Awesome, that completes your very first app!

You can find the project files for the finished app under **07 - Final App** in the tutorial's Source Code folder.

There is also a version named **08 - Final App with Comments** that has a lot of comments to show you what every piece of code does. I also removed anything that was inserted by the Xcode template that isn't actually needed for this game, so that the code is as simple as possible.

Running the game on your device

So far, you've run the app on the Simulator. That's nice and all but probably not why you're learning iOS development. You want to make apps that run on real iPhones and iPads! There's hardly a thing more exciting than running an app that you made on your own phone. And, of course, to show the fruits of your labor to other people!

Don't get me wrong: developing your apps on the Simulator works very well. When developing, I spend most of my time with the Simulator and only test the app on my iPhone every so often.

The Simulator is great, but you do need to run your creations on a real device in order to test them properly. Some things the Simulator simply cannot do. If your app needs the iPhone's accelerometer, for example, you have no choice but to test that functionality on an actual device. Don't sit there and shake your Mac!

Until recently you needed a paid Developer Program account to run apps on your iPhone. Since Xcode 7, however, you can do it for free. All you need is an Apple ID. And Xcode 8 makes it easier than ever before.

- Connect your iPhone, iPod touch, or iPad to your Mac using the USB cable.
- From the Xcode menu bar select **Window** → **Devices** to open the Devices window.

Mine looks like this (I'm using an iPhone 6s):



The Xcode Devices window

On the left is a list of devices that can be used for development.

- Click your device name to select it.

If this is the first time you're using the device with Xcode, the Devices window will say something like, "iPhone is not paired with your computer." To pair the device

with Xcode, you need to unlock the device first (hold the home button). After unlocking, an alert will pop up on the device asking you to trust the computer you're trying to pair with. Tap on **Trust** to continue.

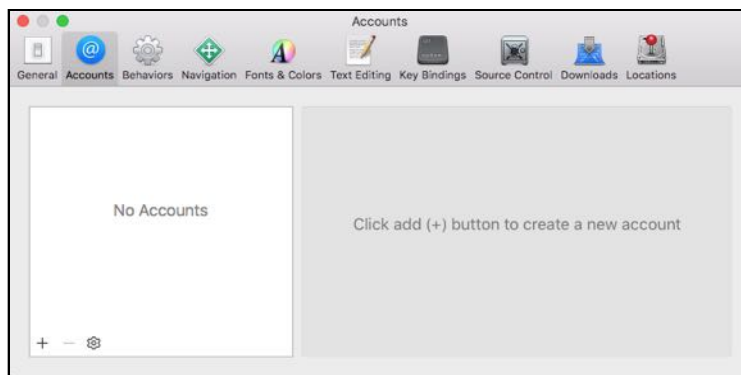
Xcode will now refresh the page and let you use the device for development. Give it a few minutes (see the progress bar in the main Xcode window). If it takes too long, you may need to unplug the device and plug it back in first.

At this point it's possible to get the error message, "An error was encountered while enabling development on this device." You'll need to unplug the device and reboot it. Make sure to restart Xcode before you reconnect the device.

Cool, that is the device sorted.

The next step is setting up your Apple ID with Xcode. It's OK to use the same Apple ID that you're already using with iTunes and your iPhone, but if you run a business you might want to create a new Apple ID to keep these things separate. Of course, if you've already registered for a paid Developer Program account, you should use that Apple ID.

► Open the **Accounts** pane in the Xcode Preferences window:



The Accounts preferences

► Click the **+** button at the bottom and choose **Add Apple ID**.

Xcode will ask for your Apple ID:



Adding your Apple ID to Xcode

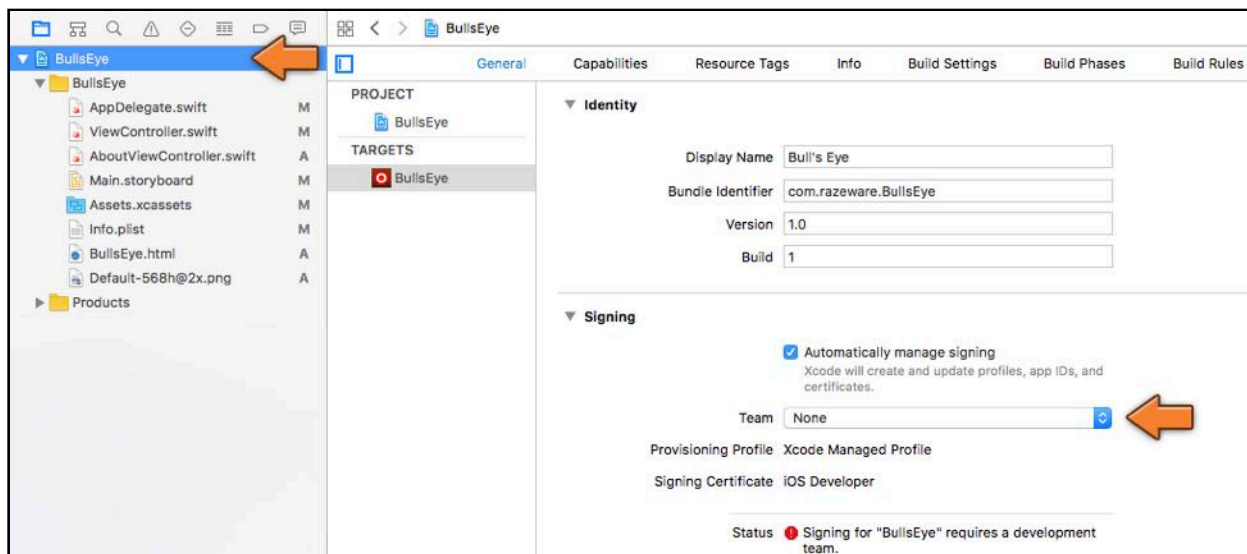
► Type your Apple ID username and password and click **Sign In**.

Xcode verifies your account details and adds them to the accounts window.

Note: It's possible that Xcode is unable to use the Apple ID you provided, for example if it has been used with a Developer Program account in the past that is now expired. The simplest solution is to make a new Apple ID. It's free and only takes a few minutes. <https://appleid.apple.com>

You still need to tell Xcode to use this account when building your app.

► Go to the **Project Settings** screen. In the **General** tab go to the **Signing** section.



The Signing options in the Project Settings screen

In order to allow Xcode to put an app on your iPhone, the app must be *digitally signed* with your **Development Certificate**. A *certificate* is an electronic document that identifies you as an iOS application developer and is valid only for a limited amount of time.

Apps that you want to submit to the App Store must be signed with another certificate, the **Distribution Certificate**. To use the distribution certificate you must be a member of the paid Developer Program but using the development certificate is free.

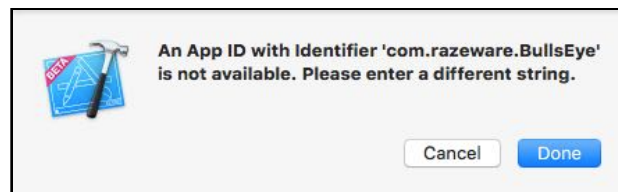
In addition to a valid certificate, you also need a so-called **Provisioning Profile** for each app you make. Xcode uses this profile to sign the app for use on your device. The specifics don't really matter, just know that you need a provisioning profile or the app won't go on your device.

Making the certificates and provisioning profiles used to be frustrating and error-prone. Fortunately, those days are over: Xcode 8 makes it really easy. When the **Automatically manage signing** option is enabled, Xcode will take care of all this business with certificates and provisioning profiles and you don't have to worry about a thing.

► Click on **Team** to select your Apple ID.

Xcode will now automatically register your device with your account, create a new Development Certificate, and download and install the Provisioning Profile on your device. These are all steps you would have had to do by hand in the past but now Xcode 8 takes care of all that.

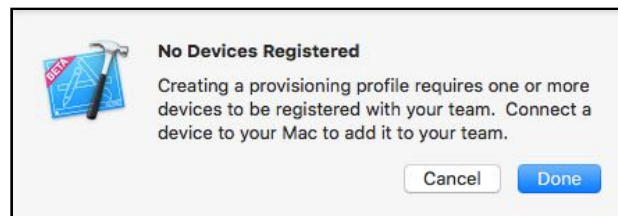
It's possible you get the following error:



The bundle identifier is already in use

The app's Bundle Identifier – or App ID as it's called here – must be unique. If another app is already using that identifier, then you cannot use it anymore. That's why you're supposed to start the Bundle ID with your own domain name. The fix is easy: change the Bundle Identifier field to something else and try again.

It's also possible you get this error:

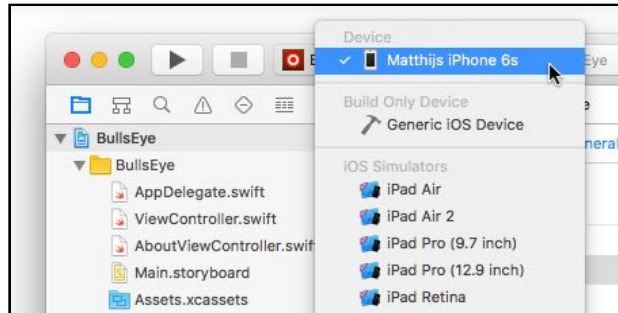


No devices registered

Xcode must know about the device that you're going to run the app on. That's why I asked you to connect your device first. Double-check that your iPhone or iPad is still connected to your Mac and that it is listed in the Devices window.

If all of that checks out, go back to Xcode's main window and click on the box in the toolbar to change where you will run the app. The name of your device should be in that list somewhere.

On my system it looks like this:



Changing where the app will be run

You're all set up and ready to go!

► Press **Run** to launch the app.

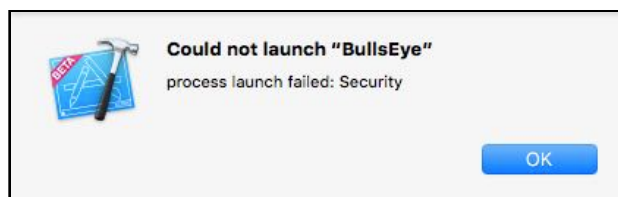
At this point you may get a popup with the question "codesign wants to sign using key ... in your keychain". If so, answer with **Always Allow**. This is Xcode trying to use the new Development Certificate you just created but you need to give it permission first.

Does the app work? Awesome! If not, read on...

There are a few things that can go wrong when you try to put the app on your device, especially if you've never done this before, so don't panic if you run into problems.

The device is not connected. Make sure your iPhone, iPod touch, or iPad is connected to your Mac. The device must be listed in Xcode's Devices window and there should not be a yellow warning icon.

The device does not trust you. You might get this warning:



Quick, call security!

On the device itself there will be a popup with the text, "Untrusted Developer. Your device management settings do not allow using apps from developer ...".

If this happens, open the Settings app on the device and go to General, Profile. Your Apple ID should be listed in that screen. Tap it, followed by the Trust button. Then try running the app again.

The device is locked. If your phone locks itself with a passcode after a few minutes, you might get this warning:

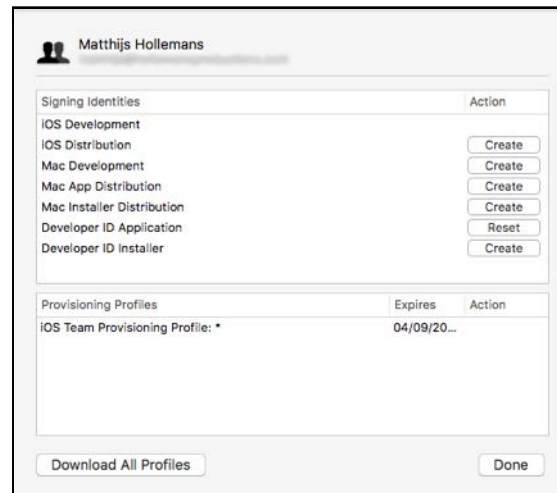


The app won't run if the device is locked

Simply unlock your device (hold the home button or type in the 4-digit passcode) and press Run again.

If you're curious about these certificates and provisioning profiles, then open the **Preferences** window and go to the **Accounts** tab. Select your account and click the **View Details...** button in the bottom-right corner.

This brings up another panel, listing your signing identities (the certificates) and the provisioning profiles:



The account details panel

The "iOS Team Provisioning Profile: *" is the thing that allows you to run the app on your device. (By the way, they call it the "team" profile because often there is more than one developer working on an app and they can all share the profile.)

When you're done, close the Accounts window and go to the Devices window.

You can see the provisioning profiles that are installed on your device by right-clicking the device name and choosing **Show Provisioning Profiles**:

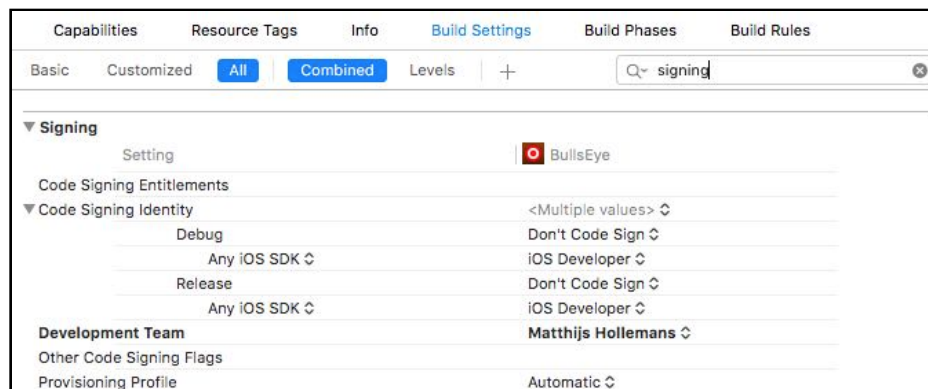


The provisioning profiles on your device

You can have more than one certificate and provisioning profile installed. This is useful if you're on multiple development teams or if you prefer to manage the provisioning profiles for different apps by hand.

To see how Xcode chooses which profile and certificate to sign your app with, go to the Project Settings screen and switch to the **Build Settings** tab. There are a lot of settings in this list, so filter them by typing **signing** in the search box. (Also make sure **All** is selected, not Basic.)

The screen will look something like this:



The Code Signing settings

Under **Code Signing Identity** it says **iOS Developer**. This is the certificate that Xcode uses to sign the app. If you click on that line, you can choose another certificate. Under **Provisioning Profile** you can change the active profile. Most of the time you won't need to change these settings, but at least you know where to find them now.

The end... or the beginning?

This has been a very long lesson – if you're new to programming, you've had to get a lot of new concepts into your head. I hope your brain didn't explode!

At least you should have gotten some insight into what it takes to develop an app.

I don't expect you to understand exactly everything that you did, especially not the parts that involved writing Swift code. It is perfectly fine if you don't, as long as you're enjoying yourself and you sort of get the basic concepts of objects, methods and variables.

If you were able to follow along and do the exercises, you're in good shape!

I encourage you to play around with the code for a bit more. The best way to learn programming is to do it, and that includes making mistakes and messing things up. I hereby grant you full permission to do so! Maybe you can add some cool new features to the game (if you do, let me know).

But for now, pour yourself a drink and put your feet up. You've earned it.

In the Source Code folder for this tutorial you can find the complete source code for the Bull's Eye app, with plenty of added commentary. If you're still unclear about some of what you did, it might be a good idea to look at this cleaned up, fully commented source code.

If you're interested in how I made the graphics, then take a peek at the Photoshop files in the Resources folder. The wood background texture was made by Atle Mo from subtlepatterns.com.

But there's more!

Thank you for reading the first tutorial from my book, *The iOS Apprentice*!

I hope this first tutorial gave you some taste of what is to come in the rest of the book, which is available from www.raywenderlich.com.

The full book has three more epic-length tutorials, each of which explains an app of increasing complexity.

You've seen what it took to build a fairly simple game. In the next tutorials I want to show you how to use features such as table views, navigation controllers, maps and GPS, the photo camera, web services, and much more... All the fundamentals that you need to know to make your own apps.

If you liked working through this free tutorial and you want to learn more about iPhone and iPad programming, then give the next lessons a try. Each new tutorial builds on what you've learned before and by the end of the series you should be able to write your own apps from scratch – with a pretty good idea of what you're doing.

What you'll learn in the rest of the book:

Tutorial 2: Checklists

Now that you've gotten a taste of how everything works, you're going to create a

basic to-do list app.

You'll learn about table views, navigation controllers, delegates, and saving your data. You will also discover the fundamental design patterns that all iOS apps use, and little by little the Swift language should start to make sense to you.

Bonus feature: setting reminders using local notifications.

Tutorial 3: MyLocations

Building on what you've learned in the previous two chapters, this tutorial goes into more depth with both Swift and the iOS frameworks.

You'll be making an app that uses the Core Location framework to obtain GPS coordinates for the user's whereabouts, Map Kit to show the user's favorite locations on a map, the iPhone's camera and photo library to attach photos to these locations, and Core Data to store everything in a database.

That's a lot of stuff! After this lesson, Swift and you will get along just fine and I'd be surprised if you won't be able to write a few apps of your own already.

Tutorial 4: StoreSearch

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

You will learn about view controller containment – or how to embed one view controller inside another – and how to show a completely different UI in landscape. We'll talk about animation, scroll views, downloading images, supporting multiple languages, and porting the app to the iPad.

Finally, I'll explain how to use Ad Hoc distribution for beta testing and how to submit your apps to the App Store. There is hardly a stone left unturned at the end of this monster tutorial!

You can get the full version of *The iOS Apprentice: Beginning iOS Development with Swift, Fifth Edition* at www.raywenderlich.com/store/ios-apprentice.

It's worth it if you want to become a great iOS developer!

Get in touch

Feel free to send Matthijs an email if you have any questions or comments about these tutorials (mail@hollance.com). And of course you're welcome to visit the forums for some good conversation at forums.raywenderlich.com.

Thanks for reading!