Fully updated for iOS 7!

iOS Apprentice

SECOND EDITION

Tutorial 1: Getting Started

By Matthijs Hollemans

The iOS Apprentice:
Getting Started

By Matthijs Hollemans

Table of Contents

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

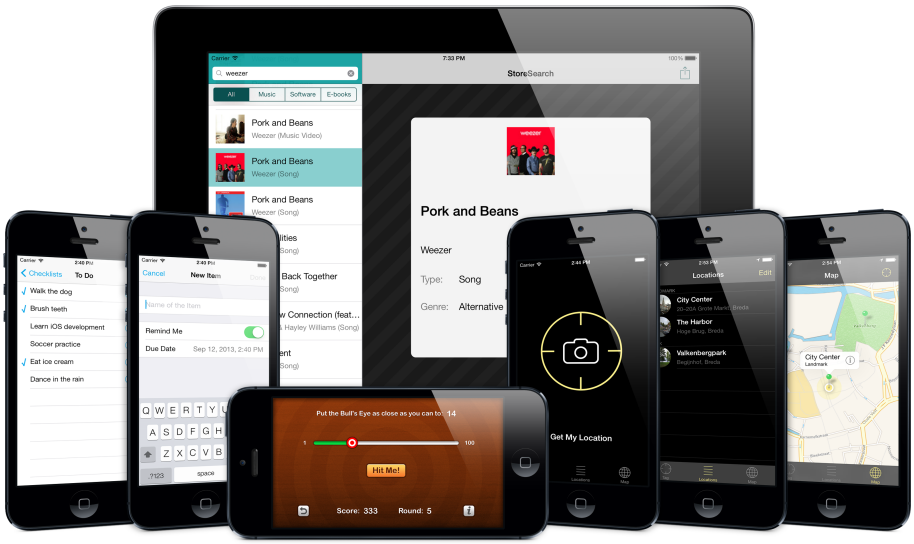
You're about to read the first epic-length tutorial from my series *The iOS Apprentice: iPhone and iPad Programming for Beginners*. This first tutorial is completely standalone and teaches you how to write a complete iPhone game named Bull's Eye. The best part is you can read it here in its entirety for free!

The other tutorials in this series build on what you'll learn here. Each tutorial 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 sell 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 little 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 other parts of the series too.



The apps you'll be making in *The iOS Apprentice* series

Introduction

Everyone likes games, so the first app you're going to make is a game named *Bull's Eye*. Even though the game will be very simple, this first tutorial might still be hard to follow – especially if you've never programmed 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 hang of it. In the subsequent tutorials from this series you'll go over many of these concepts again until they solidify in your mind.

It is important that you not just read the instructions but also actually **follow them**. Open Xcode, type in the source code fragments (or copy-paste them) 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 program 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.

I'm assuming you're a total beginner. You do not need to know anything about programming. Of course, if you do have programming experience, that helps. If you're coming from other programming languages such as PHP or Java, the low-level nature and strange syntax of Objective-C may be overwhelming. Rest assured, most programming languages work in exactly the same way – their similarities are bigger than their differences – and you'll pick up Objective-C in no time, even if it's a little quirky.

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 Objective-C 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 will 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 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!

You will run into problems, guaranteed. Your programs will have strange bugs that will leave you stumped. Trust me, I've been programming for more than 25 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.

Warning: Too many people attempt to write iPhone apps by 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 clearly illustrate what is going on. I'm not just going to give you a bunch of perfect code and tell you how it works. Instead, 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.

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!

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 Objective-C 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.

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

iOS 7 and better only

Things move fast in the world of mobile computing. The iPhone 4S and version 5.0 of iOS are only a few years old but both are quickly becoming obsolete. Even iOS version 6, which was released just in 2012, is starting to look dated now that iOS 7 has become available. And by the time you read this, you may already have your hands on iOS 8.

The tutorials in this series are aimed exclusively at iOS version 7.0 and later. The new look and feel of iOS 7 is such a big departure from previous versions that it just doesn't make sense anymore to keep developing for them. The majority of iPhone, iPod touch, and iPad users are pretty quick to upgrade to

the latest version of iOS, so you don't need to be too worried that you're leaving potential users behind.

Only users with older devices, such as the iPhone 3GS and fourth generation iPod touch, may be stuck with iOS version 6.1 or earlier but this is a tiny portion of the market. The cost of supporting these older OS versions is usually greater than the handful of extra customers it brings you.

It's ultimately up to you to decide whether it's worth supporting older devices and OS versions with your apps, but my recommendation is that you focus your efforts where they matter most. Apple as a company always relentlessly looks towards the future and if you want to play in Apple's backyard, then 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:

- **An iPhone, iPad, or iPod touch.** I'm assuming that you have at least one of these. Even though I mostly talk about the iPhone in this tutorial series, everything I say actually goes for all of these devices. Aside from small hardware differences, they all use iOS and you program them in exactly the same way. You should be able to run any of the apps you will be developing on your iPad or iPod touch without problems.
- **A 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 run at least OS X version 10.8.4 (Mountain Lion). Xcode, the development environment for iOS apps, is a memory-hungry tool so having at least 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!
- **A paid iOS Developer Program account.** This will cost you $99 per year and it allows you to run your apps on your own iPhone, iPad, or iPod touch while you're developing, and to submit finished apps to the App Store. You can download all the development tools for free if you're a paid member, including beta previews of upcoming versions of iOS.

With some workarounds it is possible to develop iOS apps on Windows or a Linux machine, or a regular PC that has OS X 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 you might 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 (10.7 Lion or earlier), then you can purchase an inexpensive upgrade to the latest version of OS X from the online Mac App Store.

Join the program

To sign up for the Developer Program, go to http://developer.apple.com/programs/ios/ and click the **Enroll Now** button.

Tip: Make sure you're on the page for the iOS program. There are also Mac and Safari developer programs and you don't want to sign up for the wrong one!

On the sign-up page you'll need to enter your Apple ID. Your developer program membership will be tied to this account. 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.

There are different types of iOS Developer Programs. You'll probably want to go for the regular iOS Developer Program, either as an Individual or as a Company. There is also an Enterprise program but that's for big companies who will be distributing apps within their own organization only. If you're still in school, the University Program may be worth looking into.

You buy the Developer Program membership from the online Apple Store for your particular country. Once your payment is processed you'll receive an activation code that you use to activate your account.

It may take a few weeks to get signed up as Apple will check your credit card details and if they find anything out of the ordinary (such as a misspelled name) your application may run into delays. So make sure to enter your credit card details correctly or you'll be in for an agonizing wait.

If you're signing up as a Company then you also need to provide a D-U-N-S Number, which is free but may take some time to request. You cannot register as a Company if you have a single-person business such as a sole proprietorship or DBA ("doing business as"). In that case you need to sign up as an Individual.

You will have to renew your membership every year but if you're serious about developing apps then that $99/year will be worth it.

The free account

If you're strapped for cash, you'll be happy to know that it's possible to develop for iOS without paying a dime. There is a free Apple developer account but this restricts you to running your apps in the Simulator. You cannot run the apps on any of your devices and, more importantly, you can't submit to the App Store.

If you just want to get your feet wet with iOS development but you're not sure yet whether you'll like it, then stick to the free account for the time being. You can run

all the apps from this tutorial in the Simulator just fine, but of course that isn't as cool as seeing them on your own iPhone.

To sign up for the free account, go to https://developer.apple.com/programs/register/. You can always upgrade to the paid account later.

Xcode

After you sign up, the first order of business is to download and install Xcode and the iOS SDK (Software Development Kit).

Xcode is the main development tool for iOS. It has a text editor where you'll type in your source code and it has a visual tool 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 (http://itunes.apple.com/app/xcode/id497799835?mt=12). This requires at least OS X Mountain Lion, so if you're still running OS X Lion or even Snow Leopard you'll first have to upgrade to the latest version of OS X (also available from the Mac App Store). Get ready for a big download, as the full Xcode package is about 2 GB.

Tip: 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 5.0 and the iOS 7.0 SDK. 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 5. Likewise for Xcode 4, which was retired in late 2013. So if you're reading some article and you see a picture of Xcode that looks completely different from yours, they're talking about the 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.

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 it, 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. On the other hand, everyday English is not precise enough to use for programming computers. So you will use an intermediary language, Objective-C, 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$pb:
    popl    %eax
    movl    16(%ebp), %ecx
    movl    12(%ebp), %edx
    movl    8(%ebp), %esi
    movl    %esi, -8(%ebp)
    movl    %edx, -12(%ebp)
    movl    %ecx, (%esp)
    movl    %eax, -24(%ebp)
    calll   _objc_retain
    movl    %eax, -16(%ebp)
    .loc    1 161 2 prologue_end
Ltmp98:
    movl    -16(%ebp), %eax
    movl    -24(%ebp), %ecx
    movl    L_OBJC_SELECTOR_REFERENCES_51-L7$pb(%ecx), %edx
    movl    %eax, (%esp)
    movl    %edx, 4(%esp)
    calll   _objc_msgSend_fpret
    fstps   -20(%ebp)
    movss   -20(%ebp), %xmm0
    movss   %xmm0, (%esp)
    calll   _lroundf
```

Actually, what the computer sees is this:

```
00011001010011110100100011001111100101000
00101000100111101011011100111010110100100
01010001110011111010111011000011100011000
10010000011100010100110100111100110011100
```

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:

```
void HandleMidiEvent(char byte1, char byte2, char byte3,
                     int deltaFrames)
```

```
{
  char command = (byte1 & 0xf0);

  if (command == MIDI_NOTE_ON && byte3 != 0)
  {
    PlayNote(byte2 + transpose, velocityCurve[byte3] / 127.0f,
             deltaFrames);
  }
  else if ((command == MIDI_NOTE_OFF)
       ||  (command == MIDI_NOTE_ON && byte3 == 0))
  {
    StopNote(byte2 + transpose, velocityCurve [byte3] / 127.0f,
             deltaFrames);
  }
  else if (command == MIDI_CONTROL_CHANGE)
  {
    if (data2 == 64)
      DamperPedal(data3, deltaFrames);
    else if (data2 == 0x7e || data2 == 0x7b)
      AllNotesOff(deltaFrames);
  }
}
```

That 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. The above snippet is from a sound synthesizer program. It is written the C language, which was invented in the sixties by the same guys who also invented the Unix operating system. Both inventions had a big impact on the world of computing. (Fun fact: the core of iOS is largely based on Unix.)

The language you use to program iOS apps is called **Objective-C** and it is an extension of the C language. Objective-C can do everything that C can but it also adds a lot of useful stuff of its own, most importantly *Object-Oriented Programming* (hence its name). Objective-C was almost extinct until the iPhone brought it back to life and now all the cool kids are using it.

C++ is another language that adds Object-Oriented Programming to C and it is from around the same time as Objective-C. Unlike Objective-C – which tries to be as simple and lean as possible – C++ contains everything but the kitchen sink. 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 Objective-C 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

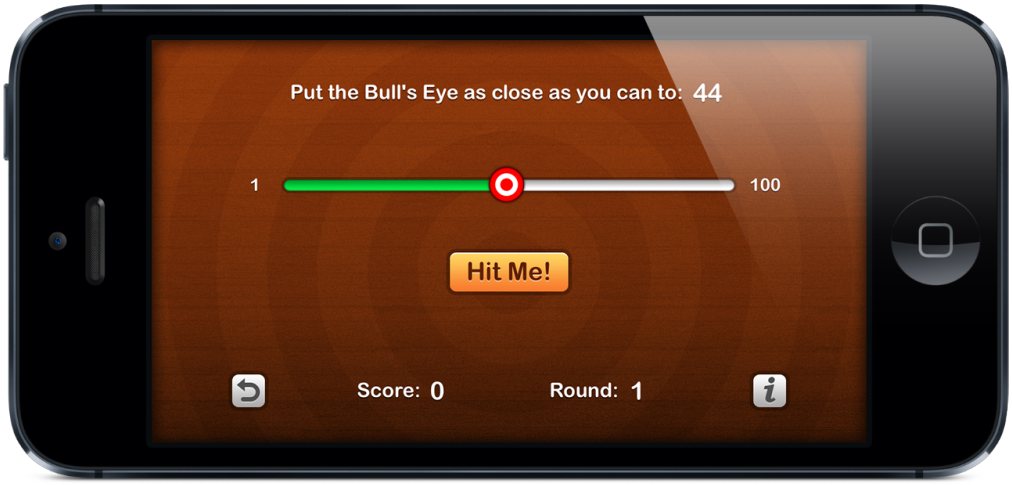
concepts – what is a variable, what is an object, how do you call a method, and so on – are more important than the details. Slowly but surely, all the secrets of the Objective-C language will be revealed to you.

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.

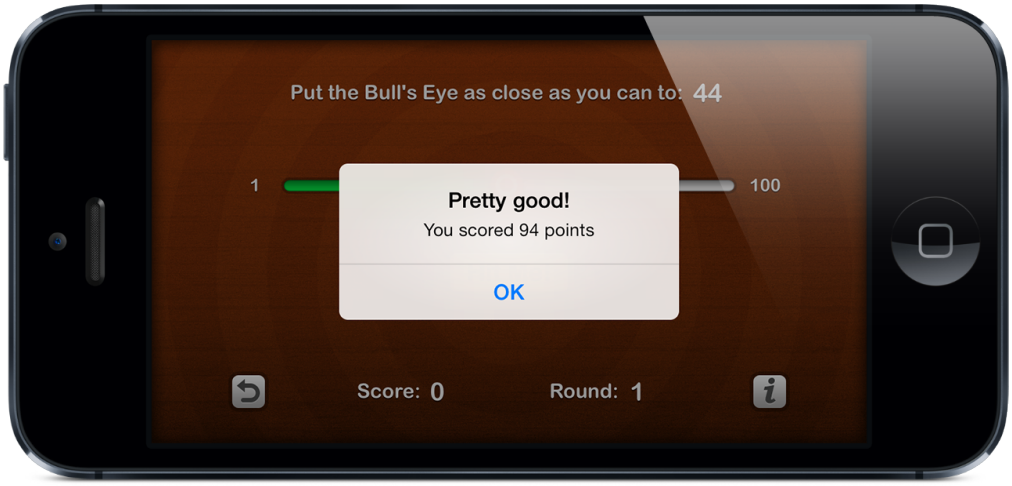
This is what the game will look like when you're finished:



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 44. 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 view*, will tell you what your score is:



An alert view popup shows the score

The closer to the target value you are, the more points you score. After you dismiss the alert view 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 (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!

The 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 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'll be surprised at how many people start programming 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 program it 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:

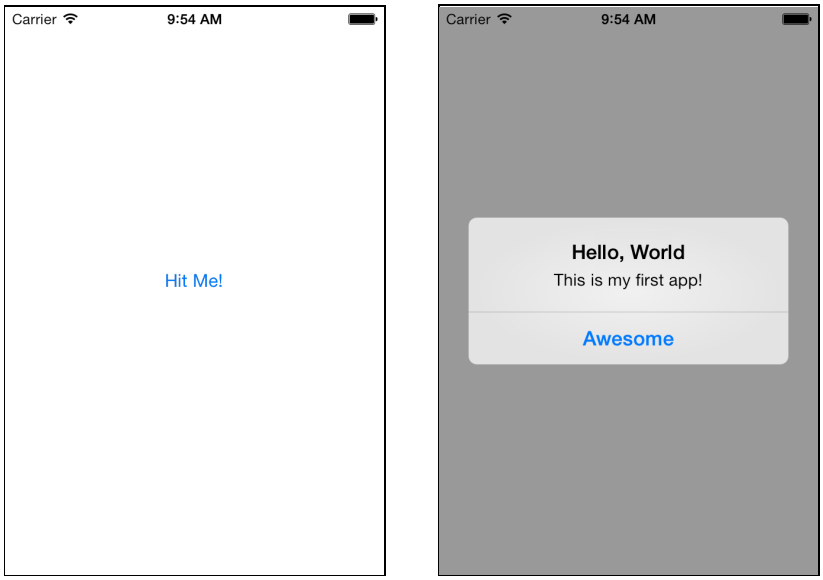
- Put a button on the screen and label it "Hit Me!"
- When the player presses the Hit Me button the app has to show an alert popup to inform the player how well she did. Somehow you have to calculate the score and put that into this alert.
- Put text on the screen, such as the "Score:" and "Round:" labels. Some of this text changes over time, for example the score, which increases when the player finishes a round.
- 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 view.
- 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, that's already quite a few things you need to do.

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:

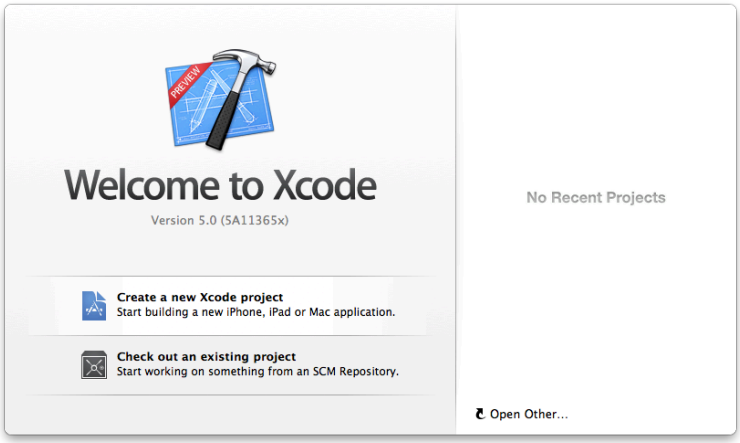


The app contains a single button (left) that shows an alert when pressed (right)

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 5.0 or better. Newer versions of Xcode will also work but anything older than version 5.0 is a no-go. So please update to the latest and greatest if you haven't already.

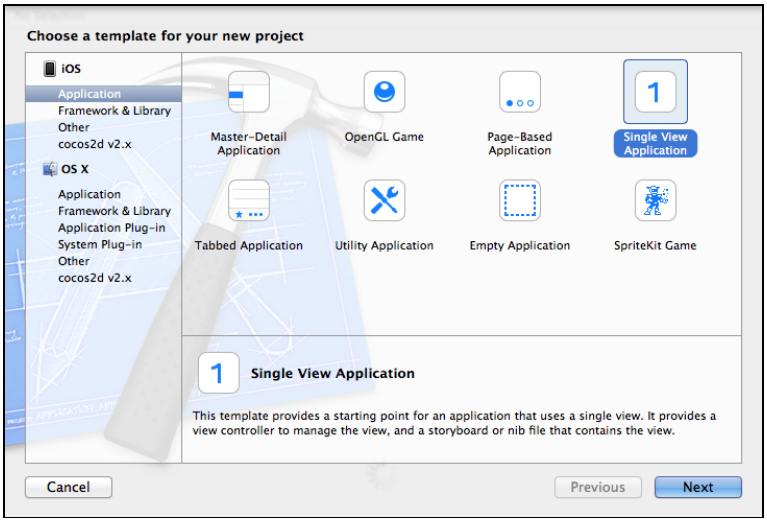
▸ Launch Xcode. If you have trouble locating the Xcode application, you can find it in the folder **/Applications/Xcode** or in your Launchpad. Because I use Xcode all the time, I placed its icon in my dock for easy access.

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:

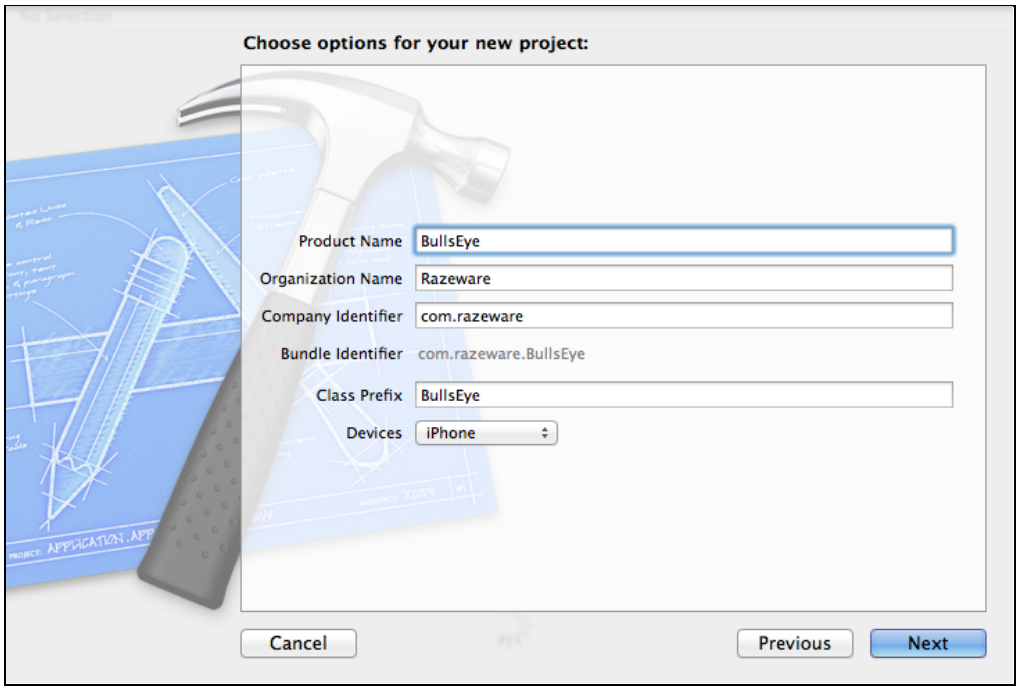


Choosing the template for the new project

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:

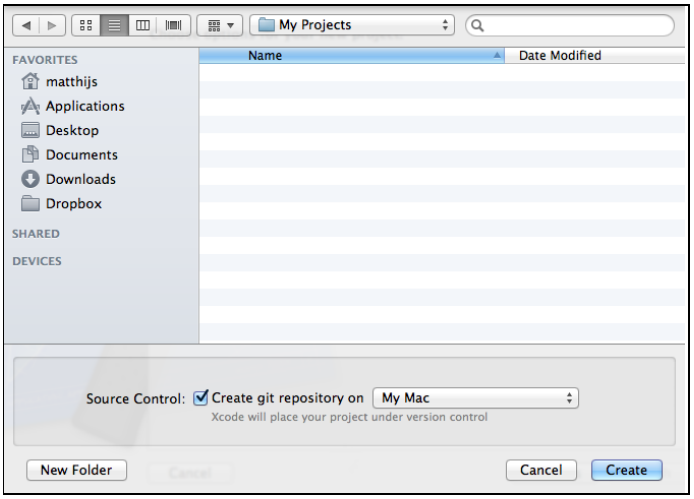


Configuring the new project

› 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.
- Organization Name: Fill in your own name here or the name of your company.
- Company Identifier: Mine says "com.razeware". That is the identifier I use for my apps and 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.
- Class Prefix: **BullsEye**
- Devices: **iPhone**

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



Choosing where to save the project

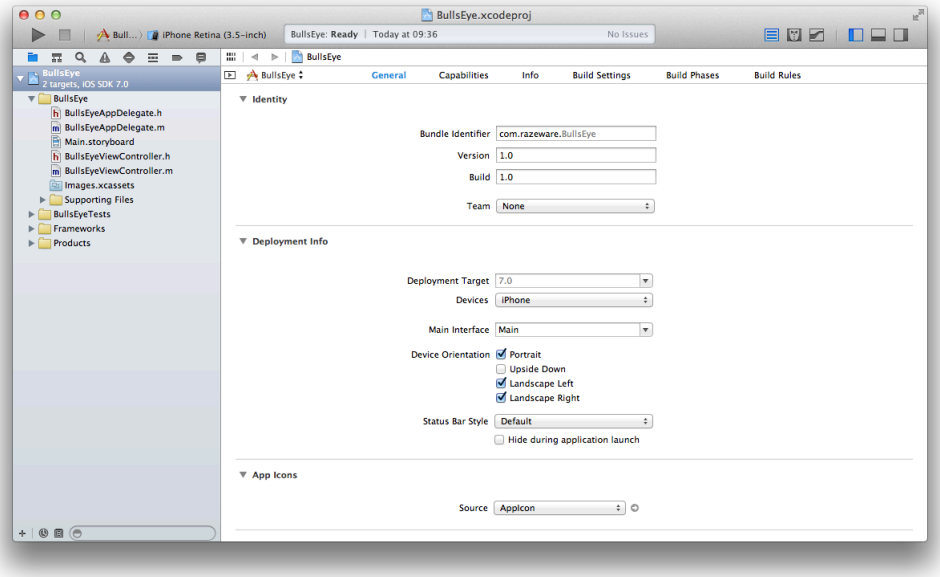
› Choose a location for the project files, for example the Desktop or your Documents folder.

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 of the window 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 and how to use it from Xcode in the next tutorials.

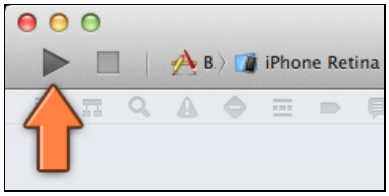
› Press **Create**.

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

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

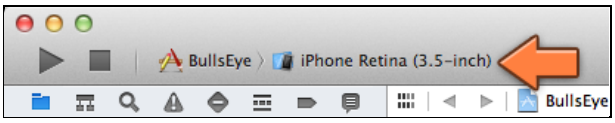


Press Run to launch the app

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" when you press the Run button, then make sure the picker at the top of the window says **BullsEye > iPhone Retina (3.5-inch)**:

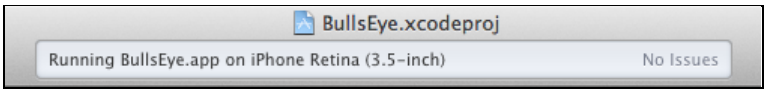


Making Xcode run the app on the Simulator

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. You might be tempted to press the home button on the Simulator, just as you would on your iPhone, 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.app on iPhone Retina (3.5-inch)":



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 that newly built version in the Simulator.

What happens when you press Run?

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

The compiler is the part of Xcode that converts your Objective-C 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 from a single press of the Run button.

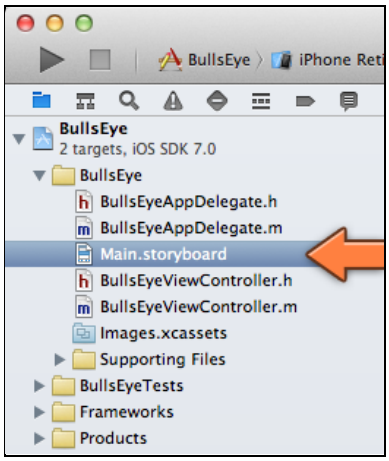
Adding the button

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 that are 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 in new groups to your heart's content. We'll talk more about the different files that your project has later.

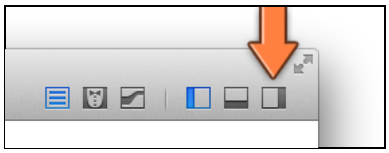
➤ In the **Project navigator**, find the item named **Main.storyboard** and click it once to select it:



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.)

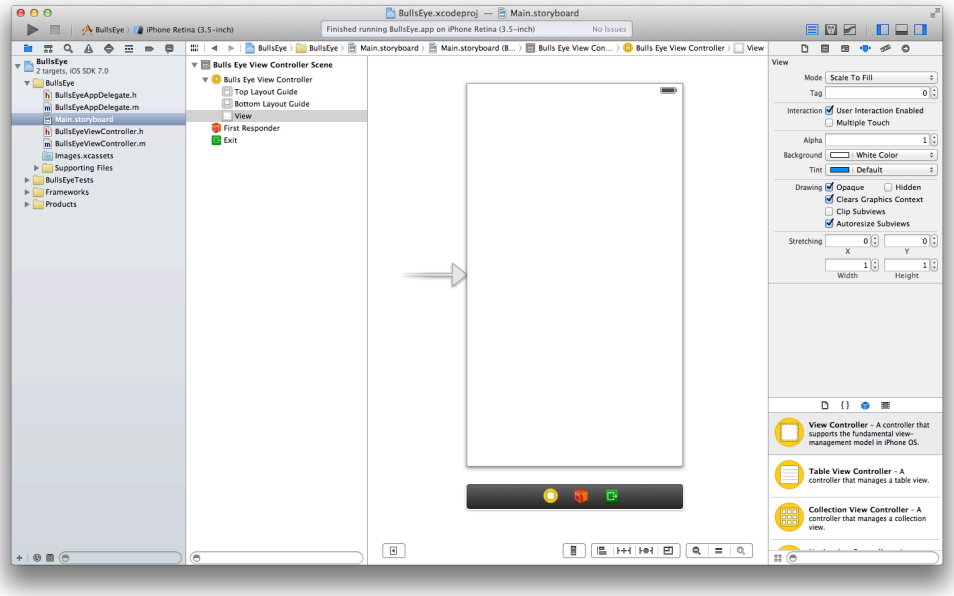
- Click the **Hide or show utilities** button in Xcode's toolbar:



Click this button to show the Utilities pane

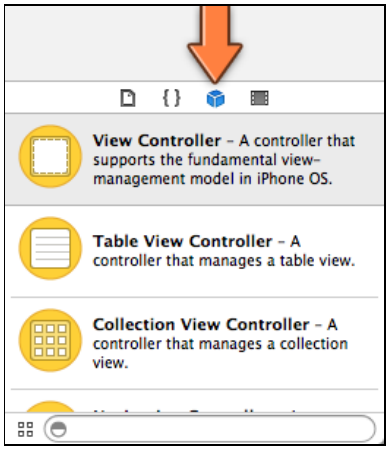
These toolbar buttons change the appearance of Xcode. This one in particular opens a new pane on the right side of the Xcode window.

Your Xcode should now look something like this:



Editing Main.storyboard in Interface Builder

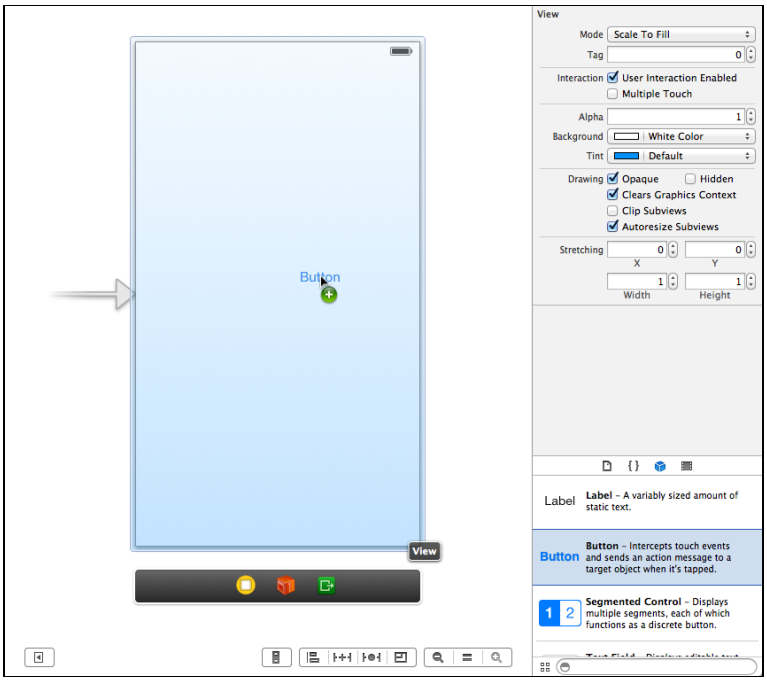
At the bottom of the Utilities pane you will find the **Object Library** (make sure the third button, the one that looks like a box, is selected):



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 white view.



Dragging the button on top of the view

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

Note that in iOS 7 buttons no longer have borders. If you've used previous versions of iOS then you no doubt have seen the standard "round rect button", which had a thin gray border with rounded corners. One of the major changes in iOS 7 is that those borders are now gone, at least for the standard buttons.

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.

Xcode will autosave

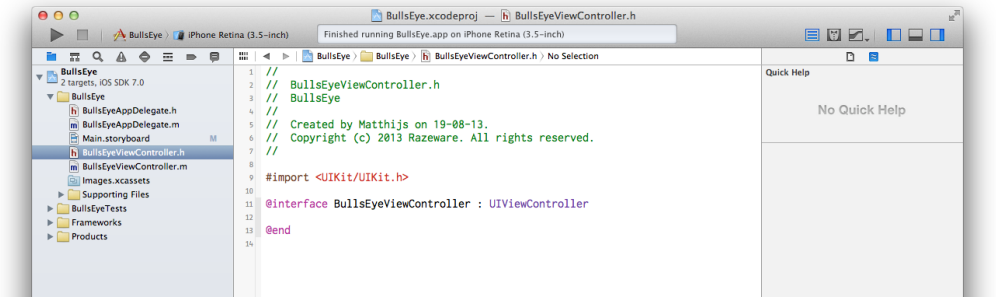
You don't have to save your source code files after you make changes to them because Xcode will automatically save any modified files when you press the Run button. Nevertheless, Xcode isn't the most stable piece of software out there and occasionally it may crash on you before it has had a chance to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

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 view popup. In the finished game the alert view will show the score for the round, but for now we shall limit ourselves to a simple text message ("Hello, World!").

➤ In the **Project navigator**, click on **BullsEyeViewController.h**.

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



The source code editor

➤ Add the following line directly above the line that says `@end`:

```
- (IBAction)showAlert;
```

The source code for **BullsEyeViewController.h** should now look like this:

```
//
//  BullsEyeViewController.h
//  BullsEye
//
//  Created by <you> on <date>.
//  Copyright (c) <year> <you>. All rights reserved.
//

#import <UIKit/UIKit.h>
```

```
@interface BullsEyeViewController : UIViewController

- (IBAction)showAlert;

@end
```

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

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 just added source code to **BullsEyeViewController.h**. If you look at the Project navigator, you will also find a file named **BullsEyeViewController.m**. You probably guessed that these two files have something to do with one another.

These three files together – the storyboard, .h and .m – form the 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. If you tap on a recipe, a new screen opens 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 named `BullsEyeViewController` and the storyboard, .h and .m files all work together to implement it.

Simply put, the storyboard file contains the design of the view controller's user interface, while the .h and .m files contain its functionality – the logic that makes the user interface work, which is written in the Objective-C language. It is customary to name these files after the view controller they represent, in your case `BullsEyeViewController`.

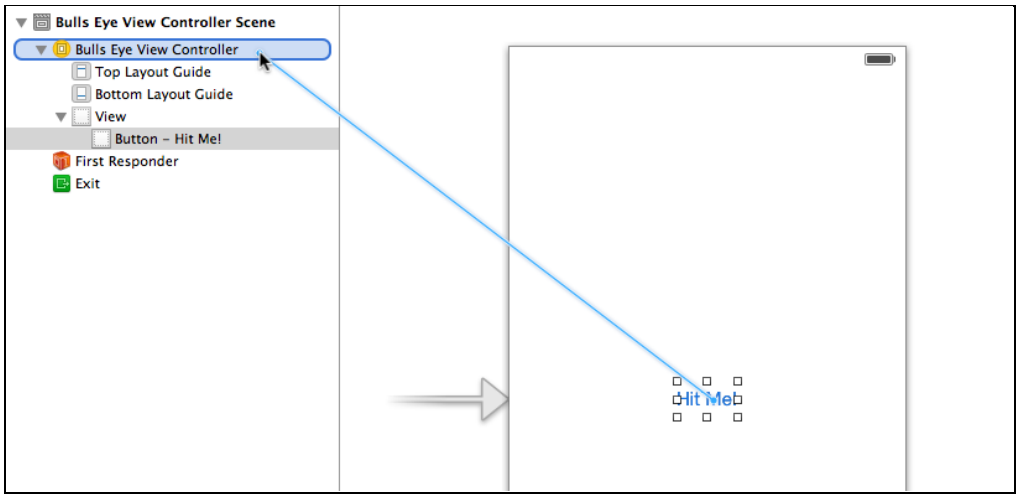
Because you used the Single View Application template, Xcode automatically created the view controller for you and named it after the project. Later you will add a second screen to the game and you will create your own view controller for that.

Making connections

The line of source code you have just added to **BullsEyeViewController.h** lets Interface Builder know that the controller has a "showAlert" action, which presumably will show an alert view popup. You will now connect the button to that action.

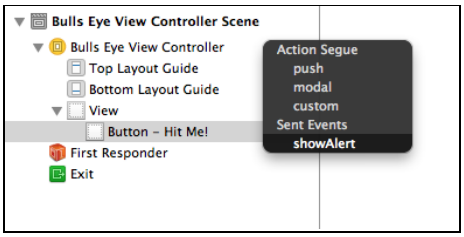
➤ Click **Main.storyboard** to go back into Interface Builder. Click the Hit Me button once to select it.

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

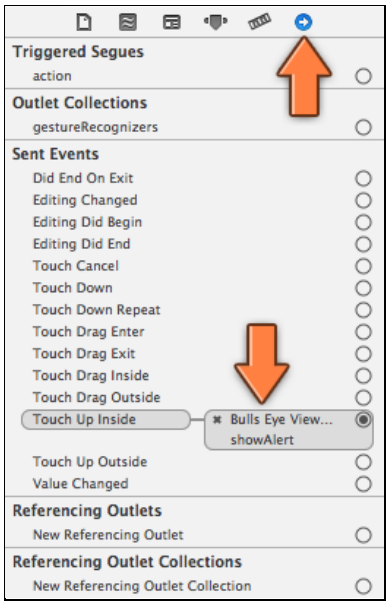
Once you're on Bulls Eye View Controller, let go of the mouse button and a small menu will appear. It contains two sections, "Action Segue" and "Sent Events", with one or more options below each. You're interested in the **showAlert** option below Sent Events. This is the name of the action that you added earlier in the **BullsEyeViewController.h** file.



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 `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 .h 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. You 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. If you wanted to remove this connection, you'd click the little × icon (but don't do that right now!). You can also drag directly from any of these events to Bulls Eye View Controller. Try it out: click on an open circle and the blue line will appear.

Acting on the button

You now have a screen with a button and you have hooked it up to an action named `showAlert` that will be performed when the user taps on the button. However, you haven't yet told the app what that action actually does.

› Select **BullsEyeViewController.m** to edit it.

In case you're wondering what the difference is between the .h and .m files, the .h file tells the computer *what* the view controller does, while the .m file tells the computer *how* it does those things. Confused? Don't worry about it; the next tutorials will explain it all in good time.

There are a few bits of source code in **BullsEyeViewController.m** already but you can ignore that for now.

› Add the following to the bottom of the file, just before the line that says `@end`:

```
- (IBAction)showAlert
{
  UIAlertView *alertView = [[UIAlertView alloc]
    initWithTitle:@"Hello, World"
    message:@"This is my first app!"
    delegate:nil
```

```
      cancelButtonTitle:@"Awesome"
      otherButtonTitles:nil];

    [alertView show];
}
```

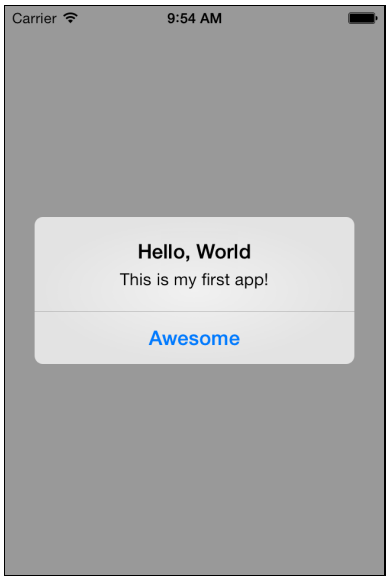
When you added the `– (IBAction)showAlert;` line to the .h file you only said to Interface Builder, "the action showAlert is available," but you did not specify what `showAlert` actually did. This time, however, you provide the actual functionality of this action. The commands between the `{ }` brackets tell the iPhone what to do, and they are performed from top to bottom.

The code in `showAlert` creates an alert view 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.

Note: You're only adding these commands to the .m file; you're not changing BullsEyeViewController.h at this point. The .h file is only for declarations (the so-called "interface"), while the actual commands always go into the .m file (known as the "implementation").

➤ 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.

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.



The alert view in action

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 chapter 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.

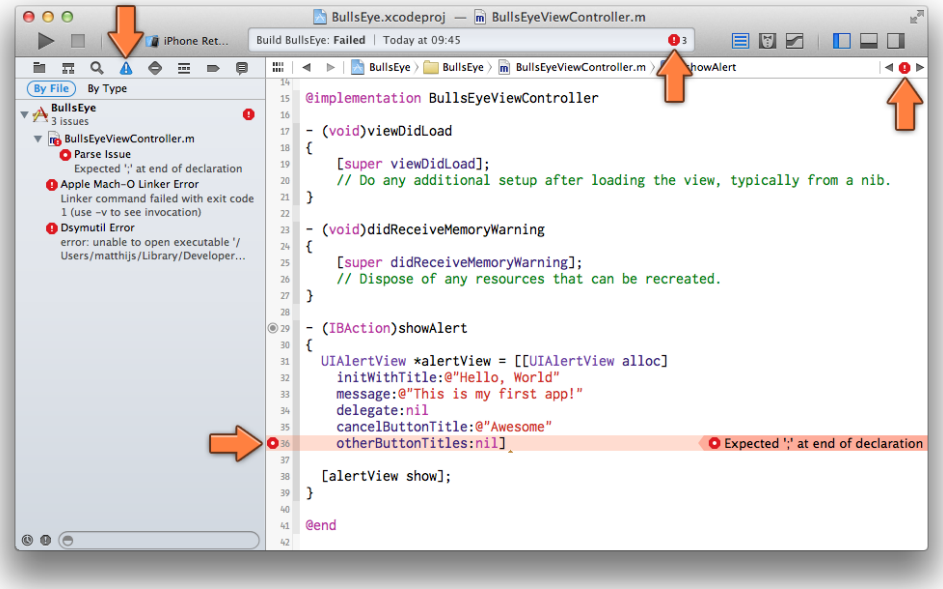
Problems?

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 error messages elsewhere in that file.

Typical mistakes are differences in capitalization. The Objective-C programming language is case-sensitive, which means it sees `AlertView` and `alertView` as two different names. Xcode complains about this with a "<something> undeclared" error.

When Xcode says things like "Parse Issue" or "Expected <something>" then you probably forgot a semicolon somewhere. Forgetting to put semicolons in the right places is another common error. All statements in the Objective-C language must end with a semicolon, just like sentences in English end with a period (or a full stop if you're British).

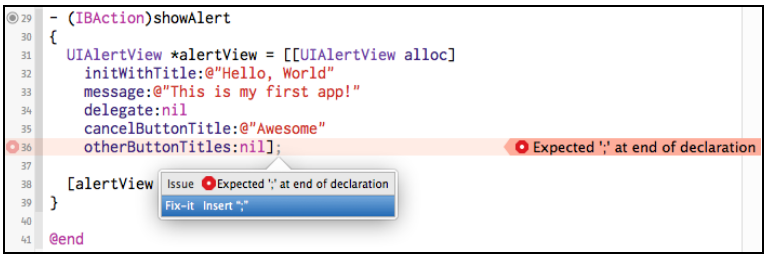
Tiny details like this are very important when you're programming. Even one single misplaced character can prevent the Objective-C compiler from building your app. Fortunately, such mistakes are easy to find.



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**. (You can go back to the project files with the small buttons at the top.) This list shows all the errors and warnings that Xcode has found. Apparently, in the image above, I forgot a semicolon 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 and warnings. 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.

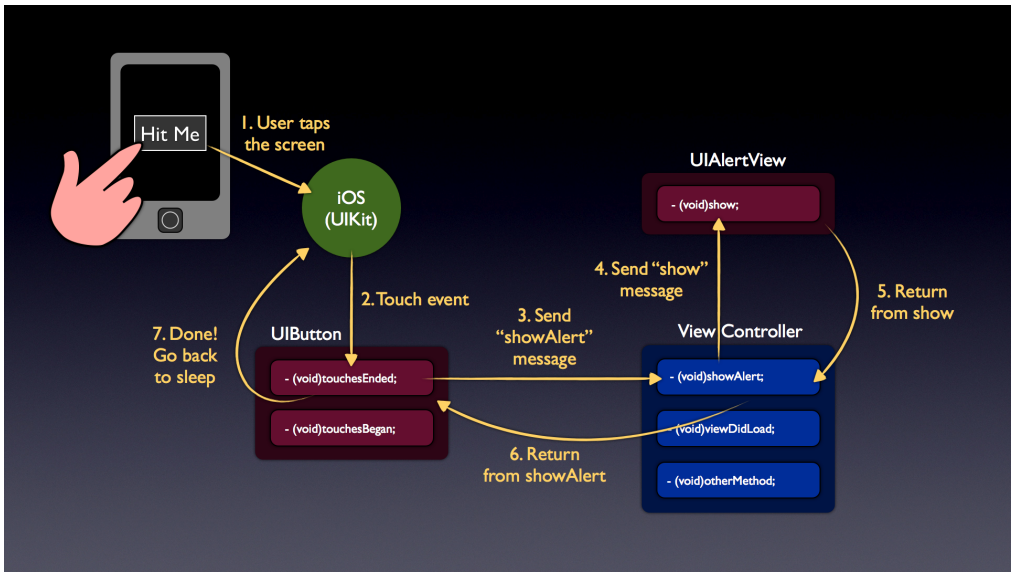
How does an app work?

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 view (a `UIAlertView` 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 that in turn 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.



The general flow of events in an app

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, `show`, to the `UIAlertView` object. Your whole app will be made up of objects that communicate in this fashion.

Maybe you have used PHP or ASP 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 when 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. 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.

Going 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 — to the screen 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

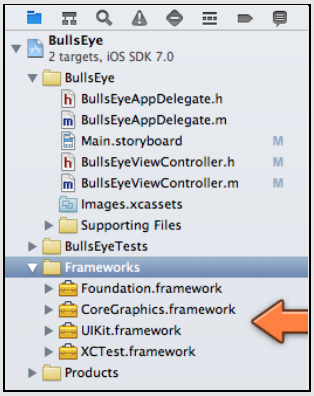
That doesn't quite look like the screenshot I showed you earlier! The difference is that these are the standard UIKit controls. This is what they look like straight out of the box. You've probably seen this look before because it is perfectly suitable for

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. The **Frameworks** group contains the system frameworks that your project uses:



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. Foundation is the framework that provides many of the basic building blocks for writing Objective-C programs (its prefix is NS, as in `NSString`).

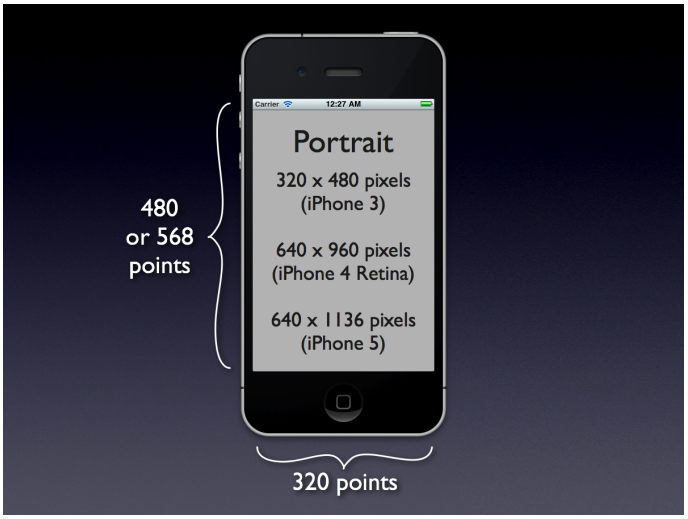
Examples of other frameworks are Core Graphics, for drawing basic shapes such as lines, rectangles, gradients and images on the screen; Core Audio for playing sounds; CFNetwork for doing networking; and many others. The complete set of frameworks for iOS is known collectively as Cocoa Touch.

Portrait vs. landscape

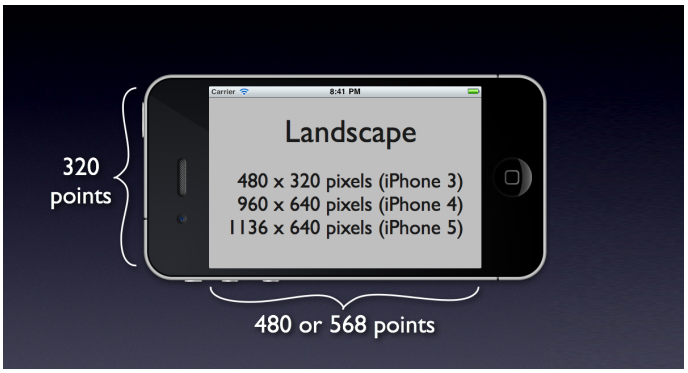
Notice that the dimensions of the app's screen are now slightly different from before: the iPhone is tilted on its side and the screen is wider but less tall. This is called *landscape* mode.

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 in the Mail app with their device flipped over because the wider screen allows for a bigger keyboard and easier typing.

In portrait mode, the iPhone screen consists of 320 points horizontally and 480 points vertically. The screen of the iPhone 5 is larger so that device has a height of 568 points. For landscape these dimensions are switched: 480 or 568 points horizontally and 320 vertically.



Screen dimensions for portrait orientation



Screen dimensions for landscape orientation

So what is a *point*? On the iPhone 3GS and earlier models, as well as the corresponding iPod touch models and the iPad 1, 2 and mini, one point corresponds to one pixel. That's easy. However, on the iPhone 4 and later models, and the new iPad with their high-resolution Retina display, one point actually corresponds to two pixels horizontally and vertically, so *four* pixels in total.

> **Note:** I'm sure you know what a pixel is (http://en.wikipedia.org/wiki/Pixel). 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. Images are produced by changing the color values of these pixels.

On older models, which I'll collectively be referring to as "low-resolution devices", you only get 320×480 pixels but on the iPhone 4 and 4S (or "Retina devices") those dimensions are doubled to 640×960 pixels, which gives you four times as many pixels to work with. On the iPhone 5 and later models, on account of their larger form factor, you get 640×1136 pixels.

The iPad has even more pixels because its screen is a lot bigger: 1024×768 pixels on the iPad 1, 2 and the mini, and a mind-blowing 2048×1536 pixels on the Retina iPad. Insane! But remember that UIKit works with points instead of pixels, so you can always assume that the iPad screen has 1024×768 points regardless of how many actual pixels are in the display.

In older books and blog posts that were written before the Retina display existed (the summer of 2010), you'll often find people referring to pixels when they really should be talking about points. The difference may 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. ;-)

You'll learn how to deal with both low-resolution and Retina graphics later in this tutorial. Thanks to this points-versus-pixels issue, putting high-resolution graphics in your apps is actually pretty straightforward.

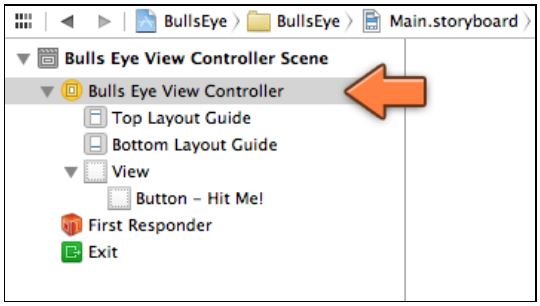
Converting the app to landscape

To convert 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.

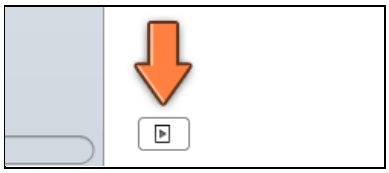
› Open **Main.storyboard** in the Interface Builder. Click on the view controller to select it.

Tip: You can also click in the **Outline pane**. The Outline pane shows the view hierarchy of the storyboard. Here you can see that the Bulls Eye View Controller contains the main view (which is simply named View), which in turn contains one sub-view at the moment, the button. Sometimes it is easier to select the item you want from this list, especially if your screen design is getting crowded.



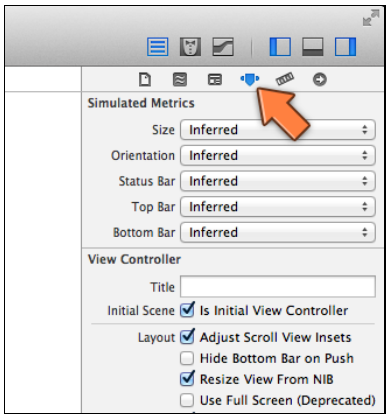
The Outline pane shows the view hierarchy of the storyboard

If this Outline pane is not visible, then click the little arrow icon at the bottom to reveal it:



The arrow button shows or hides the Outline pane

➤ Go to the **Inspector** pane, which is at the other end of the Xcode window, and click on the **Attributes inspector**.



The Attributes inspector for Bulls Eye View Controller

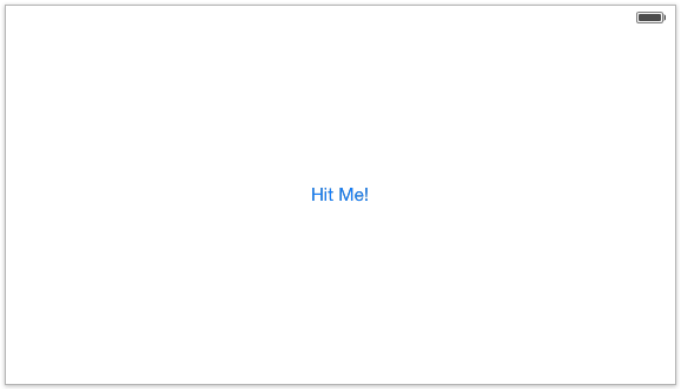
If you haven't already played with this part of Interface Builder: the Inspector area shows various aspects of the item that is currently selected. The Attributes inspector, for example, lets you change the background color of a view or the size of the text on a button. As you become more proficient with Interface Builder, you'll be using all of these inspector panes to configure your views.

To rotate the view controller to landscape, you're looking for the **Orientation** setting in the **Simulated Metrics** section. It is currently set to Inferred, which really means portrait in this case.

➤ Change Orientation to **Landscape**.

This changes the dimensions of the view controller and it probably 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.

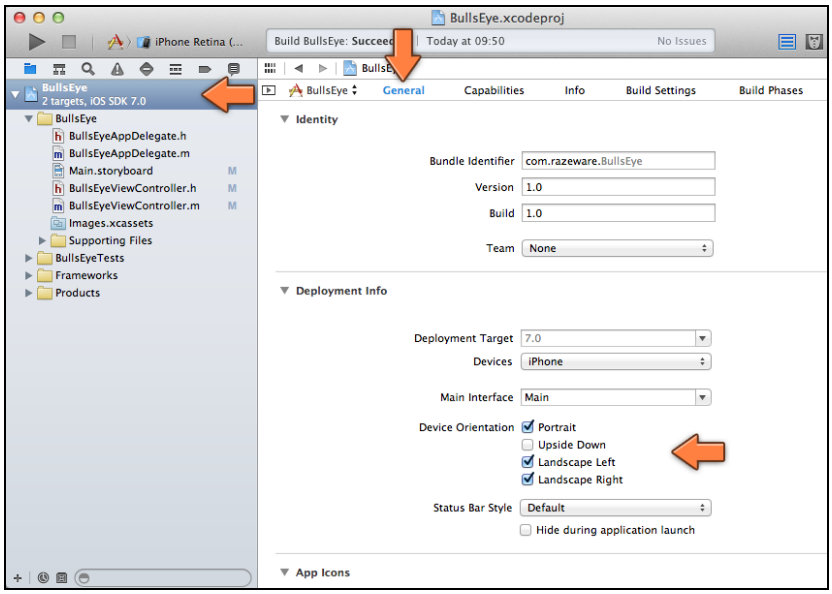
➤ Run the app. The screen does not show up as landscape yet, but the button is no longer in the center either.

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

➤ Choose **Hardware → Rotate Left** or **Rotate Right** from the iOS 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.

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.

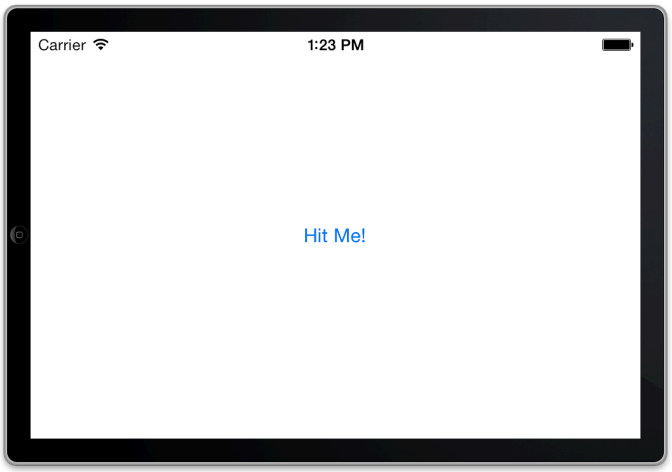
➤ Click the **BullsEye** project icon at the top of the **Project navigator**. The main pane of the Xcode window now reveals a bunch of settings for the project. Make sure that the **General** tab is selected:



The settings for the project

In the section **Deployment Info**, there is an option for **Device Orientation**. Currently the orientations Portrait, Landscape Left and Landscape Right are selected.

› Uncheck the **Portrait** option so that only the two landscape options are left checked. Run the app again and it properly launches in the landscape orientation right from the beginning.



The app in landscape orientation

Objects, messages and methods

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

Objective-C 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 an RSS feed, *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 is `BullsEyeViewController`. The Hit Me button is also an object, as is the alert view. And the texts that you put on the alert view – "Hello, World" and "This is my first app!" – are also objects. The project also has an object named `BullsEyeAppDelegate`, even though you're going to ignore that for this lesson (but feel free to look inside its files if you're curious). These object thingies are everywhere!

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

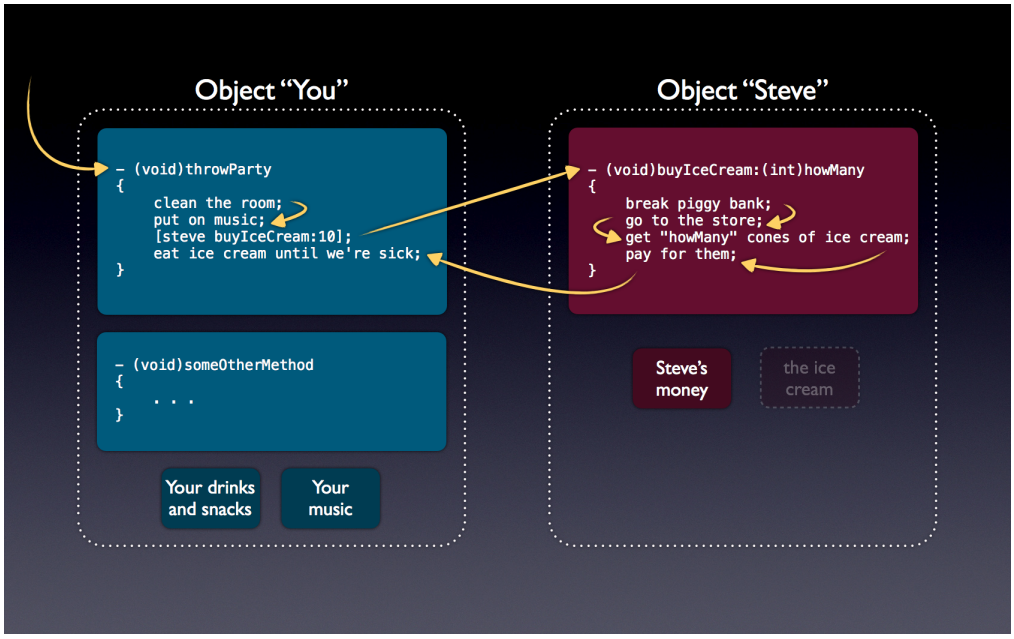
- An example of data is the Hit Me button that you added to the view controller earlier. When you dragged the button into the storyboard, it actually became part of the view controller's data. Data *contains* something. In this case, the view controller contains the button.
- An example of functionality 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 "function", "procedure" or "subroutine", but method is the term in Objective-C.

Your `showAlert` action is an example of a method. If you look through the rest of **BullsEyeViewController.m** 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 doesn't eat it all along the way).

"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.

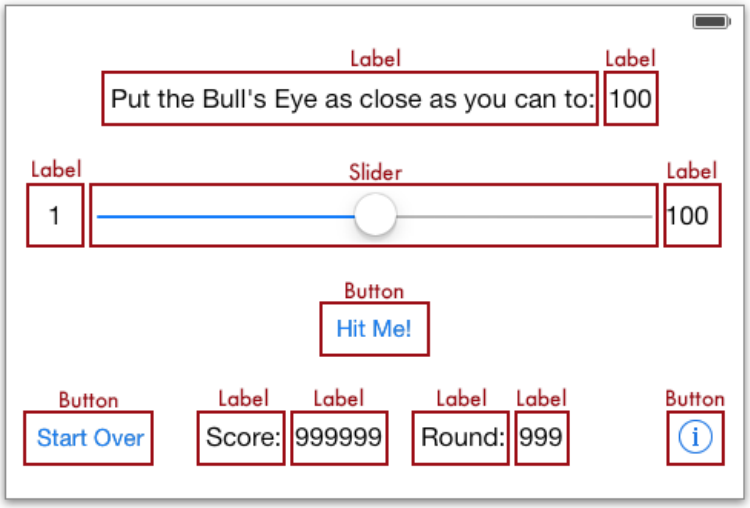
Often the terms "calling a method" or "invoking a method" are used instead. That means the exact same thing as sending a message: the computer jumps to the method you're calling and returns to where it left off when that method is done.

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, 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. 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, "999"). The reason for this is that it 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.

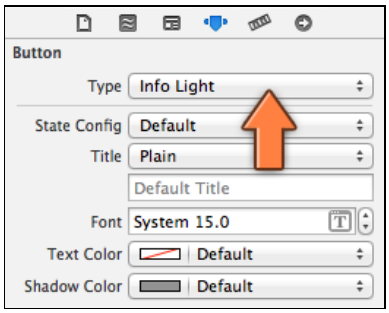
➤ Try to re-create this screen on your own by dragging the various controls from the Object Library. You can see in the screenshot above how big the items should (roughly) be. It's OK if you're a few points off.

3.5-inch vs. 4-inch

Make sure you are editing the screen in 3.5-inch mode. Interface Builder lets you view how your app will look on the iPhone 4S and earlier (the 3.5-inch models) as well as on the larger 4-inch models (iPhone 5, 5s, 5c). You use this tiny button at the bottom of the window to switch between these modes:

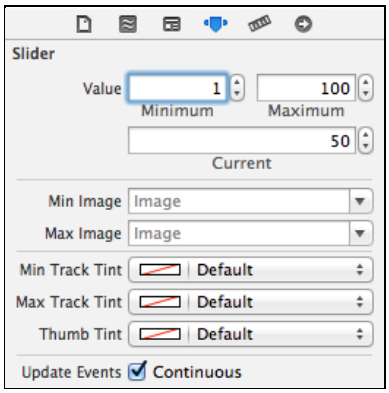
> For now, it's best if you design in 3.5-inch mode and also use the 3.5-inch Simulator to run the app. If you don't then what you see in the storyboard may look different from what you see in the Simulator. Later on you'll learn how to make the app work on the 4-inch phones as well.

Tip: 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

› Set the attributes for 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 view: 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 view), 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 Objective-C 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 view. (If you did disconnect the button, then you should hook it up again first.)

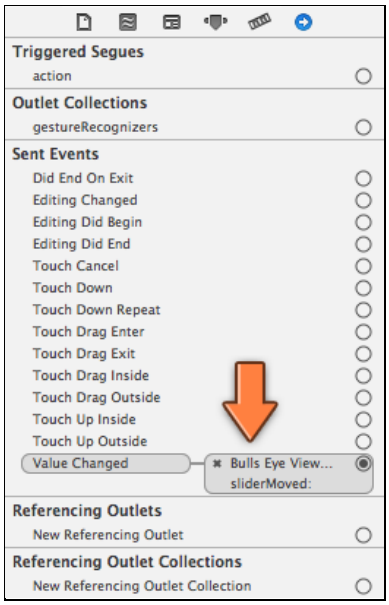
Remember how you added an action to the view controller in order to recognize when the user tapped the button? You can do the same thing for the slider. This action will be performed whenever the user drags the slider's knob. The steps for adding this action are largely the same as what you did before.

▸ First, go to **BullsEyeViewController.h** and add a declaration for the action, on a line between `showAlert` and `@end`:

```
- (IBAction)sliderMoved:(UISlider *)slider;
```

▸ Go back to the storyboard and Ctrl-drag from the slider to Bulls Eye View Controller. Let go of the mouse button and select `sliderMoved:` from the popup. Done!

If you now look at the **Connections inspector**, you can see that the `sliderMoved` action is hooked up to the slider's Value Changed event. This means the action will be called every time the slider's value changes, which happens when the user drags it to the left or right.



The slider is now hooked up to the view controller

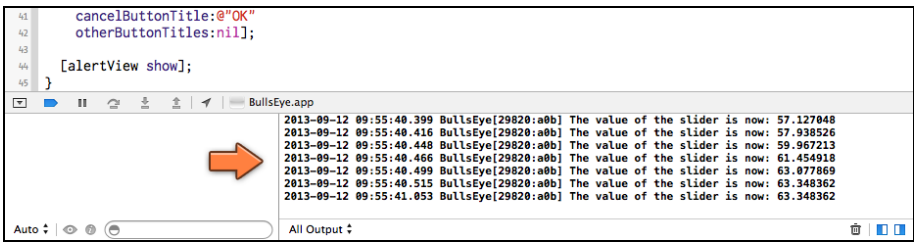
At this point you've only told the slider that there is an action but you haven't actually defined what the action will do.

➤ Go to **BullsEyeViewController.m** and add the following at the bottom, just above the `@end` line:

```
- (IBAction)sliderMoved:(UISlider *)slider
{
  NSLog(@"The value of the slider is now: %f", slider.value);
}
```

➤ 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**, which shows a list of messages:



NSLog messages in the Debug area

If you slide 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.

`NSLog` 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 `NSLog` to make sure my apps are doing the right thing before I add more functionality.

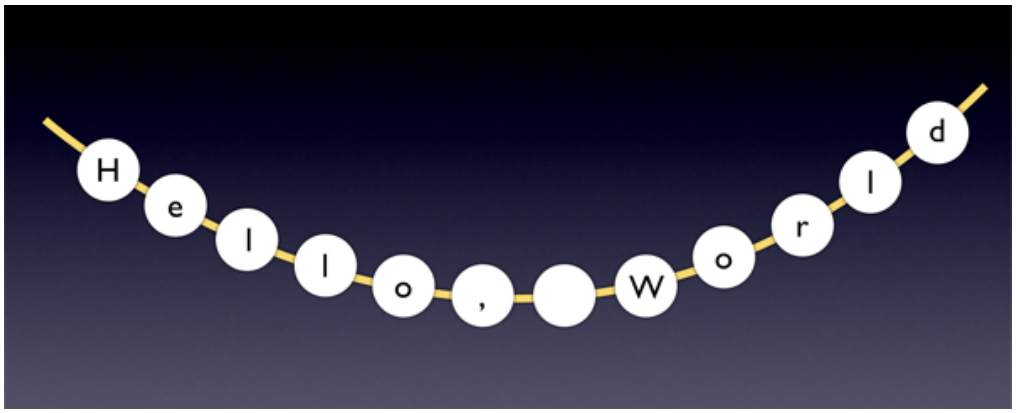
Strings

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!"
@"Awesome"
@"The value of the slider is now: %f"
```

The first three are from the `UIAlertView`, the last one you used with `NSLog` 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 (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 and prefix it with a `@`. The `@` symbol is important! If you've worked with other programming languages before, you did not need to use `@` to make strings, just double or single quotes. Confusingly enough, in Objective-C it is also possible to make strings without the `@`, but that doesn't give you the type of string you want. This often trips up people new to the language. Just remember that a string in Objective-C starts with a `@`.

If you forget the `@`, Xcode will complain with the somewhat cryptic warnings: "Passing argument from incompatible pointer type" and "Semantic Issue: Incompatible pointer types passing 'char []' to parameter of type 'NSString *'". These are only warnings, not fatal errors, so Xcode won't stop you if you go ahead and run the app anyway. However, it will most likely cause your app to crash. Don't forget the `@`.

Putting the `@` inside the string also doesn't work, so `"@this is wrong"`. And you must use double quotes: `'this will not work'`. In Objective-C single quotes can only be used to define a single character, not a whole string of them. The double quotes must be plain double quotes, not typographic "sixes and nines".

To summarize:

```
// This is the proper way to make an Objective-C string:
@"I am a good string"

// These are all wrong:
"I forgot my @ sign"
"@My at-sign is in the wrong place"
'I should have double quotes'
@“My quotes are too fancy”
```

The `NSLog` statement used the string, `@"The value of the slider is now: %f"`. You're probably wondering what the `%f` is for. It is a so-called *format specifier*.

Think of it as a placeholder: `@"The value of the slider is now: X"`, where X will be replaced by the value of the slider. Making a format string and filling in the blanks is a very common way to build up strings in Objective-C.

Introducing variables

Printing information with `NSLog` 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 this information. Let's improve this action method and make it show the value of the slider in the alert view. 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, Objective-C has a building block exactly for this purpose: the *variable*.

➤ Open **BullsEyeViewController.m** and change the `@implementation` line at the top to:

```
@implementation BullsEyeViewController
{
  int _currentValue;
}
```

Note: The Xcode template also put a line `@interface BullsEyeViewController` at the top of the .m file. That is not the line you want to change here! Be sure to make the changes to the line that starts with the word `@implementation`, not `@interface`.

You have now added a variable named `_currentValue` to the view controller. Variables are added inside `{ }` brackets and it is customary to indent these lines with a tab or spaces. (Which one you use is largely a matter of personal preference. I like to use a tab because it's less typing but this book uses two spaces to keep the layout tidy.)

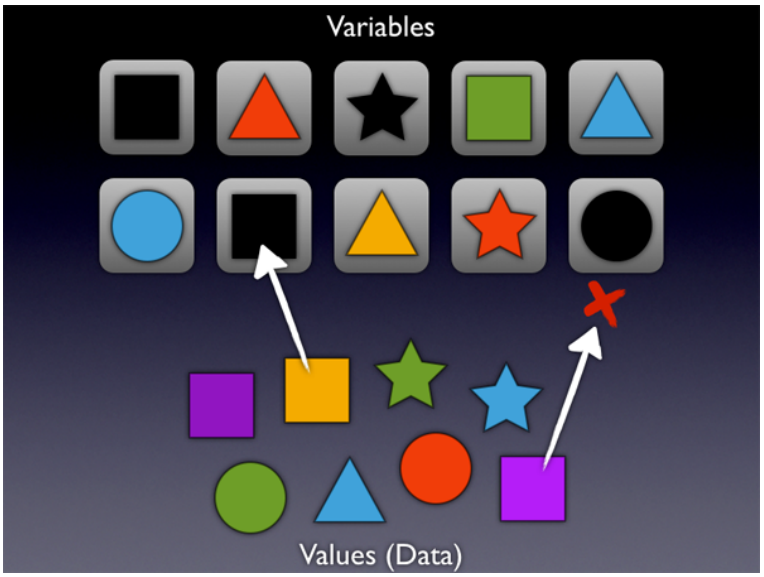
Remember when I said that a view controller, or any object really, could have both data and functionality? The `showAlert` and `sliderMoved` actions are examples of functionality, while the `_currentValue` variable is part of its data.

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 *datatype*. You specified the datatype `int` for the `_currentValue` variable, which means this container can hold whole numbers (also known as "integers") between minus two billion and plus two billion. `int` is one of the most common datatypes 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 datatype 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; 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 datatype of the value and the datatype 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 in the fridge for too long – they'll keep their values 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, `BullsEyeViewController`, 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. In a minute, you'll see variables that live much shorter.

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

➤ Change the contents of the `sliderMoved` action in **BullsEyeViewController.m** to the following:

```
- (IBAction)sliderMoved:(UISlider *)slider
{
  _currentValue = lroundf(slider.value);
}
```

You removed the `NSLog()` 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 of the slider 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 in the Debug pane when you used `NSLog()` to show the slider's value as you moved it.

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 datatype `int`, because that 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 the `_currentValue` variable.

Functions

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). Objective-C is based on the old C programming language and functions are how C programs combine multiple lines of code into single, cohesive units. We do that sort of thing with methods, older programs did that with functions.

You probably won't be writing many functions of your own – most Objective-C programming is done with objects and objects use methods – but they are very similar to methods in principle. The main distinction is that a function doesn't belong to an object while a method does. They also look a little different:

This is how you call a method:

```
[someObject methodName:parameter];
```

This is how you call a function:

```
FunctionName(parameter);
```

Fortunately for us, their C heritage provides our own 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. `NSLog()` 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)showAlert
{
  NSString *message = [NSString stringWithFormat:
    @"The value of the slider is: %d", _currentValue];

  UIAlertView *alertView = [[UIAlertView alloc]
    initWithTitle:@"Hello, World!"
    message:message
    delegate:nil
    cancelButtonTitle:@"OK"
    otherButtonTitles:nil];

  [alertView show];
}
```

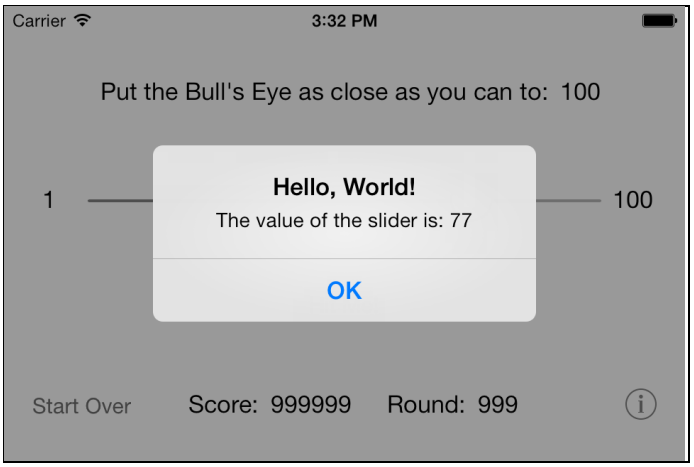
As before, you create and show a `UIAlertView`, except this time its message says: "The value of the slider is: X", where X is replaced by the contents of the `_currentValue` variable (a whole number between 1 and 100).

`NSString` is the iPhone's string object. Remember how I said earlier that objects are everywhere? Well, strings are also objects. You create a new string object named `message`, using the method `stringWithFormat`. This method takes two parameters: a format string and the value to replace in that string.

This should look familiar. It is what you did with `NSLog()` earlier, except that the placeholder is now `%d` instead of `%f`. The difference is that `%d` is used for integer values and `%f` is used for decimals (also known as "floating point" in programmer-speak, hence the f). `_currentValue` stores an integer, so you need to use `%d`.

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: %d"` into `@"The value of the slider is: 34"` and puts that in the `NSString` object named `message`. `NSLog()` 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 view.

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



The alert view shows the value of the slider

Cool. You have used a variable 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 view's message text. If you tap the button again without moving the slider, the alert view will still show the same value. The variable keeps its value until you put a new one into it.

Local variables vs. instance variables

Here is `showAlert` again:

```
- (IBAction)showAlert
{
  NSString *message = [NSString stringWithFormat:
    @"The value of the slider is: %d", _currentValue];

  UIAlertView *alertView = [[UIAlertView alloc]
    initWithTitle:@"Hello, World!"
```

```
      message:message
      delegate:nil
      cancelButtonTitle:@"OK"
      otherButtonTitles:nil];

    [alertView show];
  }
```

Note that `message` is also a variable, as it is a temporary storage container for the newly created string. Its datatype is `NSString` because that is what it stores, a string object. (Maybe you're wondering what is up with the `*`. The asterisk is necessary because `NSString` is an object and in Objective-C, objects have an asterisk in their name.)

By the same token, `alertView` is also a variable (and has been all along!) that stores a `UIAlertView` object.

The difference between `message` and `alertView` on the one hand and `_currentValue` on the other is that the `message` and `alertView` variables only exist very briefly. They are so-called *local variables*, as 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. there are no more statements for it to execute, the computer destroys the `message` and `alertView` variables. Their storage space is no longer needed.

The `_currentValue` variable, however, lives on forever, or at least for as long as the `BullsEyeViewController` does (which is until the user terminates the app). This type of variable is also named an *instance variable* (or *ivar* for short), 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 event to the next.

To make a distinction between the two types of variables, so that it's always clear at a glance how long they will live, the names of instance variables are often prefixed with an underscore. That's why you wrote `_currentValue` instead of just `currentValue`. This is not a requirement, but it is customary among Objective-C programmers.

Confused? Don't worry too much about it at this point. You'll be going over this a few more times before the tutorial is done. Variables will soon become second nature.

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 view 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. We'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, if you don't move the slider, then the `sliderMoved` message is never sent and you never put the slider's value into the `_currentValue` variable. The default value for instance variables in Objective-C is 0, and that is what you are seeing here.

To fix this bug, you're going to do some work inside the `viewDidLoad` method in `BullsEyeViewController`. If you look at the top of **BullsEyeViewController.m**, you should already see something like this:

```
- (void)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.

➤ Change the `viewDidLoad` method to the following:

```
- (void)viewDidLoad
{
  [super viewDidLoad];
  _currentValue = 50;
}
```

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. You simply set `_currentValue` to 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.

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. 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".

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

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 into any source code files that you add to the project. Personally, I don't care much for these comment blocks. Feel free to remove these lines if you don't like them either.

Properties and 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 `viewDidLoad` always assumes it will be 50. You'd have to remember to also fix `viewDidLoad` 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 it is better if you would do the following.

➤ Change `viewDidLoad` to:

```
- (void)viewDidLoad
{
  [super viewDidLoad];
  _currentValue = 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`.

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

➤ Try to run the app.

Xcode says "Build Failed", followed by something like: "Error: Use of undeclared identifier 'slider'". That is 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)sliderMoved:(UISlider *)slider
{
  _currentValue = lroundf(slider.value);
}
```

Here you also use `slider.value` but now `slider` is the *parameter* of the `sliderMoved` action. When you create an action method and hook it up to a UI control in Interface Builder, you can specify that you wish a reference to the control's object to be sent along with the action method. That is convenient when you wish to refer to that object in the method, just as you did here (the object is the `UISlider`), or if you want to use the same action method for more than one UI control.

In this case, when the `sliderMoved` action is performed, `slider` contains a reference to the actual slider object that was moved. The `UISlider` object basically said, "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." That `slider` variable contains this "phone number" but it is only valid for the duration of this particular method. In other words, it is a local variable. You cannot use it anywhere else.

The solution is to store a reference to the slider as an instance variable, just like you did for `_currentValue`. Except that this time, the datatype of the variable is not `int`, but `UISlider`. And you're not using a regular instance variable but a special form called a *property*.

➤ Add the following line to **BullsEyeViewController.h**:

```
@property (nonatomic, weak) IBOutlet UISlider *slider;
```

It doesn't really matter where this line goes, just as long as it is between `@interface` and `@end`. I usually put properties above the action methods.

This tells Interface Builder that you now have an *outlet* named `slider` that can be connected to a `UISlider` object. Just as Interface Builder likes to call methods "actions", it calls these properties outlets.

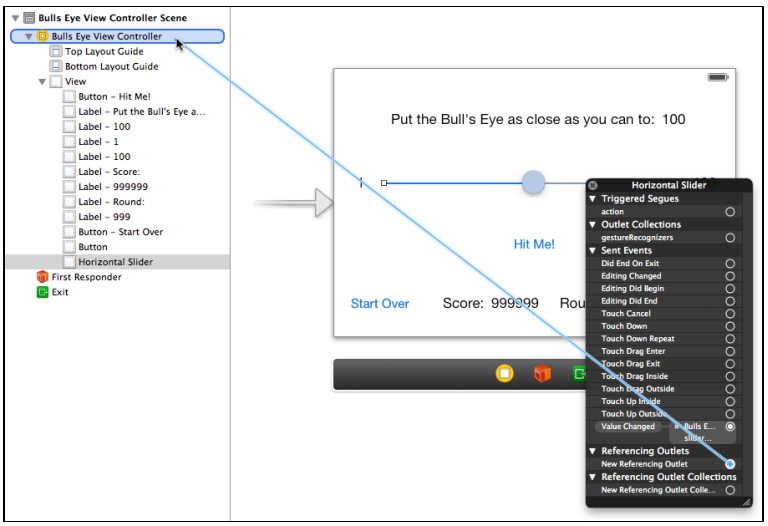
BullsEyeViewController.h should now look something like this:

```
#import <UIKit/UIKit.h>

@interface BullsEyeViewController : UIViewController

@property (nonatomic, weak) IBOutlet UISlider *slider;

- (IBAction)showAlert;
- (IBAction)sliderMoved:(UISlider *)slider;

@end
```

› Open the storyboard. Hold Ctrl and click on the slider. Let go of the mouse button and a menu pops up that shows all the connections for this slider. (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 **Bulls Eye View Controller**:



Connecting the slider to the outlet

➤ In the popup menu that appears, select **slider**.

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

> **Note:** If you've played with Objective-C before, you may have heard that you need to "synthesize" your properties before you can use them. That used to be true but with the latest versions of Xcode, adding the `@synthesize` directive for your properties is no longer necessary. Simply add the `@property` line, connect the outlet in the storyboard, and you're all set.

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` property.

➤ Change `viewDidLoad` to:

```
- (void)viewDidLoad
{
  [super viewDidLoad];
  _currentValue = self.slider.value;
}
```

The only change here is that you added the word `self` in front of `slider`. That is how you refer to a property. 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.

Here, `self` refers to the view controller. The construct `self.slider` refers to the slider property inside the view controller. Finally, `self.slider.value` refers to the slider's value, which also happens to be a property (on the `UISlider` object).

This is known as *dot-notation*, two or more names separated by dots. Whenever you see `something.somethingElse`, properties are being used.

Now 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 view should read 25 now.

> **Properties vs. instance variables**

Properties and instance variables have a lot in common. In fact, when you create a property, it is "backed" by an instance variable. That means your `slider` property stores its value in an instance variable named `_slider` (note the leading underscore). This instance variable was automatically added to the view controller by the Objective-C compiler.

Why? Well, a property needs to store its value somewhere and an instance variable is a good place for that. You can tell the difference between the two because properties are always accessed using `self`.

This uses the property:

```
self.slider.value = 50;
```

This uses the backing instance variable directly:

```
_slider.value = 50;
```

So what is the added benefit of using a property over an instance variable? There are several reasons but mainly instance variables are supposed to be used only by the insides of an object. Other objects aren't intended to see them or use them.

Properties, however, can be accessed by other objects that are outside of your view controller. You'll learn much more about this in the next tutorials because "information hiding" is an important topic in object-oriented programming.

By the way, properties are not just for making outlets. It is customary to use properties for the outlets in your storyboard, but as you will see later, you can also make properties for things that are not in the storyboard.

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 **BullsEyeViewController.m**:

```
  _targetValue = 1 + arc4random_uniform(100);
```

So `viewDidLoad` should now look like:

```
- (void)viewDidLoad
{
  [super viewDidLoad];
  _currentValue = self.slider.value;
  _targetValue = 1 + 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. 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.

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

To make this work, you have to add the variable `_targetValue` to the view controller, otherwise Xcode will complain that it doesn't know anything about this variable.

➤ Add the `_targetValue` variable to the instance variable section at the top of **BullsEyeController.m**:

```
@implementation BullsEyeViewController
{
  int _currentValue;
  int _targetValue;
}

. . .

@end
```

The rest of this file doesn't change, only the highlighted line is added.

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!)

> **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 as you're typing. So sometimes you may see temporary warnings and error messages that go away when you complete the changes that you're making. Don't be too scared of 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`).

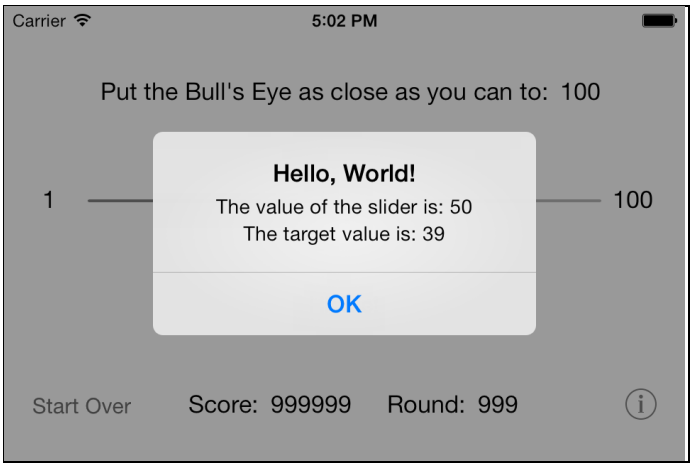
➤ Change `showAlert` to the following:

```
- (IBAction)showAlert
{
  NSString *message = [NSString stringWithFormat:
    @"The value of the slider is: %d\nThe target value is: %d",
    _currentValue, _targetValue];

  // This part did not change:
  UIAlertView *alertView = [[UIAlertView alloc]
    initWithTitle:@"Hello, World!"
    message:message
    delegate:nil
    cancelButtonTitle:@"OK"
    otherButtonTitles:nil];

  [alertView show];
}
```

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 first `%d` is replaced by `_currentValue`, the second `%d` by `_targetValue`.

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 view shows the target value on a new line

Rounds

If you pressed the Hit Me button a few times, you will have noticed 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 only once, 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 the start of a new round we have to do this and that," 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 below `viewDidLoad` in **BullsEyeViewController.m**:

```
- (void)startNewRound
{
  _targetValue = 1 + arc4random_uniform(100);
  _currentValue = 50;
  self.slider.value = _currentValue;
}
```

▸ And change `viewDidLoad` to:

```
- (void)viewDidLoad
{
  [super viewDidLoad];
  [self startNewRound];
}
```

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 also call this method after the player pressed the button, from within `showAlert`.

➤ Make the following change to `showAlert`:

```
- (IBAction)showAlert
{
  . . .

  [alertView show];

  [self startNewRound];
}
```

This is doing something you haven't seen before: `[self startNewRound]`. Earlier you have used `self` to refer to one of the view controller's properties, `self.slider`. Now you're using `self` to call one of the view controller's methods.

So far 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, using the syntax `[object methodName]`. You already did this on `NSString` when you called `[NSString stringWithFormat]` and on `UIAlertView` when you said `[alertView show]`. Anytime you see something between `[ ]` brackets, it's a method call.

When you use `self`, you are sending a message from one method in the object to another method in that same object. Think of this as telling yourself to do something: "Oh wait, I have to buy some ice cream first."

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).

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:

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  _targetValue = 1 + arc4random_uniform(100);
  _currentValue = 50;
  self.slider.value = _currentValue;
}

- (IBAction)showAlert
{
  . . .

  [alertView show];

  _targetValue = 1 + arc4random_uniform(100);
  _currentValue = 50;
  self.slider.value = _currentValue;
}
```

Can you see what is going on here? You have duplicated the same functionality 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 this bit of code is supposed to be doing. Can you tell at a glance what this does?

```
_targetValue = 1 + arc4random_uniform(100);
_currentValue = 50;
self.slider.value = _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:

```
[self 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. □

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, otherwise the player won't know what to aim for and that would make the game impossibly hard to win...

When you made the storyboard, you already added a label for the target value (top-right corner). Now 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 a reference to the label so you can send it messages

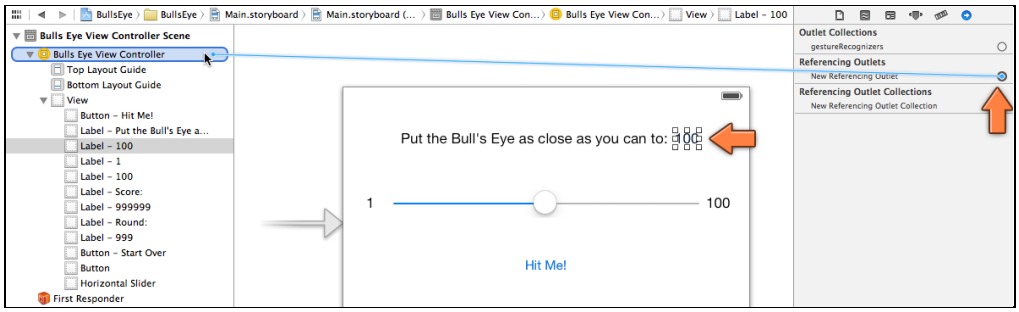
2. Give the label new text to display

You already did something similar with the slider. Recall that you created a `@property` so you could reference the slider anywhere from within the view controller, and that you could ask the slider for its value through `self.slider.value`. You'll do the same thing for the label.

➤ In **BullsEyeViewController.h**, add the following line below the other property:

```
@property (nonatomic, weak) IBOutlet UILabel *targetLabel;
```

➤ 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 **Bulls Eye View Controller**. Select **targetLabel** from the popup, and the connection is made.



Connecting the target value label to its outlet

You should be familiar with this process now. Over the past couple of pages, I've shown you several ways to make connections from various user interface objects to the Bulls Eye View Controller: Ctrl-dragging from the object, Ctrl-clicking on the object to get a context-sensitive popup menu, and now from the Connections inspector. I trust you'll be able to pull this off in the future.

➤ Now on to the good stuff. Add the following method below `startNewRound` in **BullsEyeViewController.m**:

```
- (void)updateLabels
{
  self.targetLabel.text = [NSString stringWithFormat:@"%d",
                                                 _targetValue];
}
```

You put this logic in 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:

```
self.targetLabel.text = _targetValue;
```

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

`self.targetLabel` accesses the `targetLabel` property of the view controller, which is a `UILabel` object. The `UILabel` object has a `text` property, which is an `NSString` object. You can only put string values into an `NSString` 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 `stringWithFormat` does. You've done that a few times before, and now you know why.

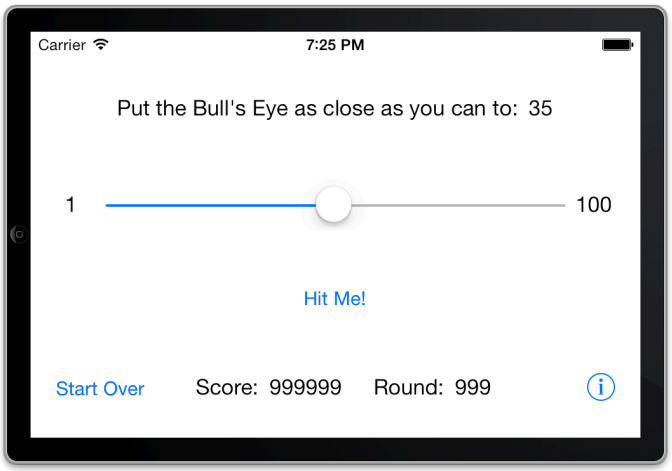
`updateLabels` is a regular method – it is not attached to any UI controls as an action – so it won't do anything until you actually call it. A logical place would be after each call to `startNewRound`, because that is where you calculate the new target value.

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

➤ Change `viewDidLoad` and `showAlert` to:

```
- (void)viewDidLoad
{
  [super viewDidLoad];
  [self startNewRound];
  [self updateLabels];
}

- (IBAction)showAlert
{
  . . .

  [self startNewRound];
  [self updateLabels];
}
```

➤ 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 - Outlet Properties** 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? 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 hook it up 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 them up to your buttons.

This is the simple form of an action method:

```
- (IBAction)doSomething;
```

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

```
- (IBAction)buttonTapped:(UIButton *)button;
```

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

```
- (void)someOtherMethod;
```

★ ★ ★

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

This is only the first tutorial of my tutorial series, *The iOS Apprentice: iPhone and iPad Programming for Beginners*. The full series consists of several more of these huge tutorials and in each you will develop a complete app from scratch. Each new app will be a little more advanced than the one before and 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 Objective-C 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. I will do my best to prepare you for your journey into the world of iPhone and iPad development.

Some highlights of what the *iOS Apprentice* series will teach you:

- How to program in Objective-C, even if you've never programmed before or if Objective-C scares you.

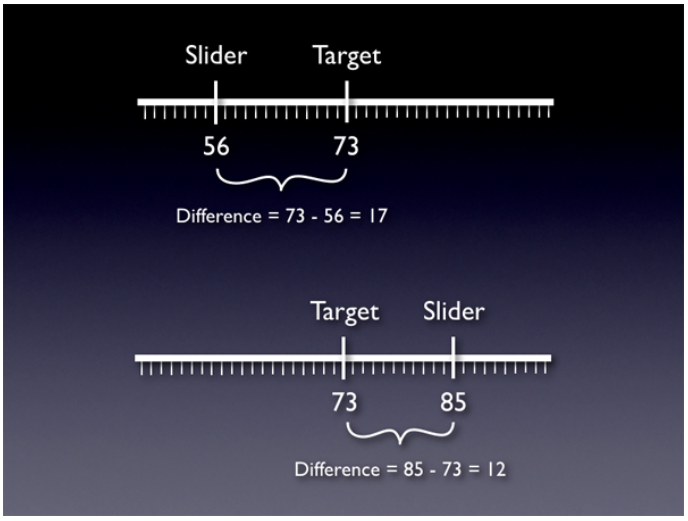
- How to think like a programmer. You are more than a code monkey who just punches source code into an editor. As a programmer you'll have to think through difficult computational problems and find creative solutions. Once you possess this valuable skill, you can program anything!
- The iOS SDK is huge 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.
- There is more to making apps than just programming. We'll discuss user interface design as well as graphics techniques to make apps look better. You already get a taste of that in this first tutorial when you give the game a makeover and add support for the Retina display.
- We will take full advantage of the latest iOS features such as Automatic Reference Counting (ARC) and 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 head on over to http://www.raywenderlich.com/store/ios-apprentice to get the other parts from the *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`. However, that gives a negative value if the slider is to the right of the target because now `_currentValue` is greater than `_targetValue`.

It would be unfair to subtract points from the score, so you need to turn that negative value into a positive value. 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.

Exercise: How would you frame this problem if I asked you to solve it in natural language? Don't worry about how to express this problem to the computer 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 target value.
>
> However, if the target value is greater than the slider value,
> then the difference is the 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 the difference is 60 - 40 = 20. On the screen the slider is to the right of the target

value. However, if the slider is to the left of the target, then it has a smaller value, say 10 for the slider and 30 for the target. 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 have to come up with a few algorithms of your own. Some are simple such as the one above; others can be pretty hard that 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 can 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:

```
int difference;
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 condition between the `( )` parentheses. 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 create the local variable `difference` to hold the result. This will either be a positive whole number or zero, so an `int` will do:

```
int difference;
```

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;
```

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. In other words, 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;
}
```

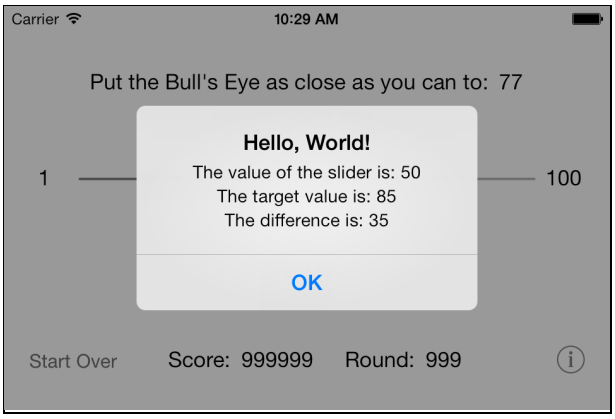
At this point you've already determined that one value is not greater than the other, nor is it smaller, leaving you only one conclusion to draw: the numbers must be equal.

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

```
- (IBAction)showAlert
{
  int difference;
  if (_currentValue > _targetValue) {
    difference = _currentValue - _targetValue;
  } else if (_targetValue > _currentValue) {
    difference = _targetValue - _currentValue;
  } else {
    difference = 0;
  }

  NSString *message = [NSString stringWithFormat:
    @"The value of the slider is: %d\nThe target value is: %d\nThe
difference is: %d",
    _currentValue, _targetValue, difference];

  . . .
}
```

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

➤ Run it and see for yourself.



The alert view 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:

> Subtract the target value from the slider's value.
>
> 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:

```
int 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. Then 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, which is 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.

> **Same symbol, different meaning**
>
> The `*` means multiply, but only in the context of a mathematical expression. When you've seen the asterisk symbol before, for example with `UISlider * slider`, it meant that you were dealing with an object. Sometimes programming languages use the same symbol for different purposes, depending on the context. (There are only so many symbols to go around.)

You could also have written this algorithm as follows:

```
int 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 any math student if you don't believe me.

▸ Give these new algorithms a try. You should replace the old stuff from `showAlert`, like this:

```
- (IBAction)showAlert
{
  int difference = _currentValue - _targetValue;
```

```
  if (difference < 0) {
    difference = difference * -1;
  }

  NSString *message = . . .;
}
```

When you run this new version of the app, it should work exactly the same way as before because the result of the computation does not change, only the technique you used.

The third alternative is to use 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:

```
int difference = abs(_targetValue - _currentValue);
```

➤ `showAlert` now becomes even simpler. Change it to:

```
- (IBAction)showAlert
{
  int difference = abs(_targetValue - _currentValue);

  NSString *message = . . .;
}
```

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.

Now that you have the difference, calculating the player's score for this round is easy.

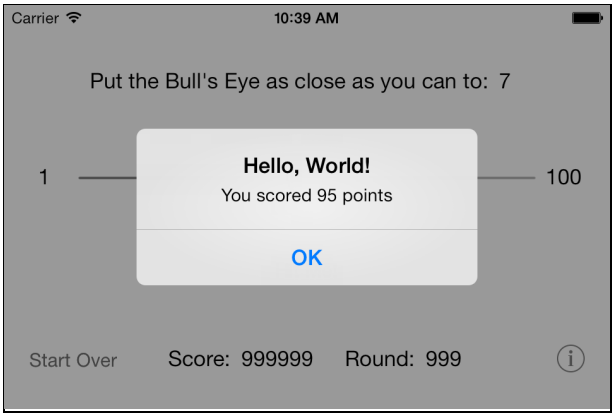
➤ Change `showAlert` to:

```
- (IBAction)showAlert
{
  int difference = abs(_targetValue - _currentValue);
  int points = 100 - difference;
```

```
  NSString *message = [NSString stringWithFormat:@"You scored %d
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 view 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. □

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 this total score around for a long time, you will put it in an instance variable.

▸ Add a new `_score` instance variable to **BullsEyeViewController.m**:

```
@implementation BullsEyeViewController
{
  int _currentValue;
  int _targetValue;
  int _score;
}
```

Now `showAlert` can be amended to update this `_score` variable.

➤ Make the following changes:

```
- (IBAction)showAlert
{
  int difference = abs(_targetValue - _currentValue);
  int points = 100 - difference;
  _score += points;

  NSString *message = [NSString stringWithFormat:@"You scored %d
points", points];

  . . .
}
```

Nothing too shocking here. You just added the 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.

I would like to point out one more time the difference between local variables and instance variables (or "ivars"). 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.

In `showAlert`, there are four local variables and you use three instance variables:

```
int difference = abs(_targetValue - _currentValue);
int points = 100 - difference;
_score += points;
NSString *message = . . .;
UIAlertView *alertView = . . .;
```

Exercise: Point out which variables are the local variables and which ones are the instance variables in the `showAlert` method. □

Local variables are easy to recognize, because the first time they are used inside a method their name is preceded with a datatype, in this case `int`, `NSString` and `UIAlertView`:

```
int difference = . . .;
int points = . . .;
```

```
NSString *message = . . .;
UIAlertView *alertView = . . .;
```

This syntax creates a new variable. Because the variable is created inside the method, it is restricted to that method only and does not exist outside of it. As soon as the method is done, the local variables cease to exist. The next time that method is invoked, these local variables are created anew.

The instance variables, on the other hand, are defined outside of any method at the top of the file:

```
@implementation BullsEyeViewController
{
  int _currentValue;
  int _targetValue;
  int _score;
}
```

As a result, you can use them from any method, without the need to declare them again.

If you were to do this:

```
- (IBAction)showAlert
{
  int difference = abs(_targetValue - _currentValue);
  int points = 100 - difference;
  int _score = _score + points;

  . . .
}
```

then things wouldn't work as you'd expect them to. Because you now put `int` 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 this is something you want to avoid.

To make the distinction between instance variables and local variables extra clear, it is a convention among Objective-C programmers to begin the names of instance variables with an underscore. That's why you keep track of the total score in a variable named `_score`, so that it does not get confused for a local variable.

The underscore has no particular meaning in the Objective-C language; it is just a convention. Some programmers use other prefixes, such as "m" (for member) or "f" (for field) for the same purpose.

> **Tip:** Xcode will give a warning when you attempt to use the name of an instance variable for a local variable. It will say something to the effect of, "Local declaration of '_score' hides instance variable". It's not recommended to have a local variable that is named the same as an instance variable, because the local one takes precedence over the instance variable and effectively "hides" it. If you're unaware of this overlap, you could be sneaking bugs into your app that will be very hard to find. It is always a good idea to pay attention to the warnings of Xcode. They may save you from doing things you did not really intend to. Small mistakes are easy to make!

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 a property 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 the .h:

```
@property (nonatomic, weak) IBOutlet UILabel *scoreLabel;
```

And you went into the storyboard file and Ctrl-dragged from Bulls Eye View Controller to the label (the one that says 999999) to connect the label to the `scoreLabel` property.

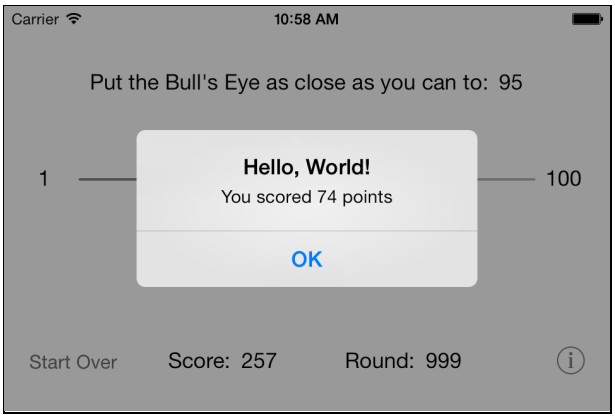
Great, that gives you a `scoreLabel` property that you can use to put text into the label. Now where in the code shall you do that? In `updateLabels`, of course.

➤ Back in **BullsEyeViewController.**m, change `updateLabels` to the following:

```
- (void)updateLabels
{
  self.targetLabel.text = [NSString stringWithFormat:@"%d",
                                                 _targetValue];
  self.scoreLabel.text = [NSString stringWithFormat:@"%d",
                                                    _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 instance variables section (the block in between the `{ }` brackets following `@implementation`):

```
int _round;
```

And also a property for the label in **BullsEyeViewController.h**:

```
@property (nonatomic, weak) IBOutlet UILabel *roundLabel;
```

As before, you should have connected the label to this property 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 property 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 property 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:

```
- (void)updateLabels
{
  self.targetLabel.text = [NSString stringWithFormat:@"%d",
                                                     _targetValue];
  self.scoreLabel.text = [NSString stringWithFormat:@"%d",
                                                          _score];
  self.roundLabel.text = [NSString stringWithFormat:@"%d",
                                                          _round];
}
```

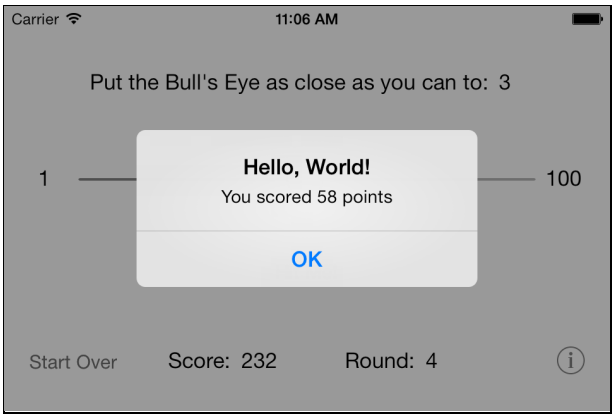
Did you also figure out where to increment the `_round` variable? I'd say the `startNewRound` method is a pretty good place. After all, you call this method whenever you start a new round. It makes sense to increment the round counter there.

➤ Change `startNewRound` to:

```
- (void)startNewRound
{
  _round += 1;
  _targetValue = 1 + arc4random_uniform(100);
  _currentValue = 50;
  self.slider.value = _currentValue;
}
```

Note that instance variables have a default value of 0. That is very handy. When the app starts up, `_round` is guaranteed to be 0. When you call `startNewRound` for the very first time, you add 1 to this default 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 view 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 `UIAlertView`. You already make the message text dynamically and now you will do something similar for the title text.

▸ Here is the changed method in its entirety:

```
- (IBAction)showAlert
{
  int difference = abs(_targetValue - _currentValue);
```

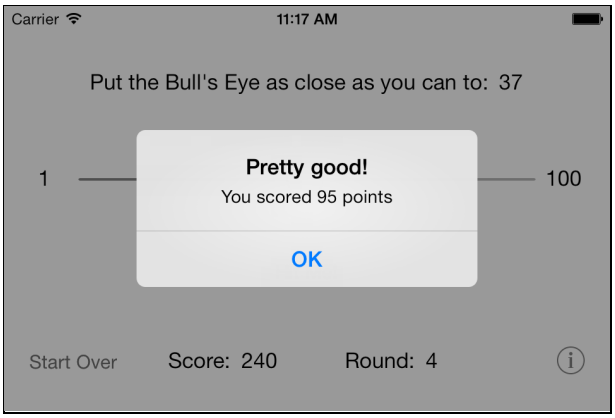
```
  int points = 100 - difference;
  _score += points;

  NSString *title;
  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...";
  }

  NSString *message = [NSString stringWithFormat:
                              @"You scored %d points", points];

  UIAlertView *alertView = [[UIAlertView alloc]
    initWithTitle:title
    message:message
    delegate:nil
    cancelButtonTitle:@"OK"
    otherButtonTitles:nil];

  [alertView show];

  [self startNewRound];
  [self updateLabels];
}
```

You create a new local string variable named `title`, which will contain the text that goes at the top of the alert view. Initially, this `title` variable 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!" And 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 `UIAlertView` object, you now give it this `title` variable 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 view with the new title

Exercise: Give the player an additional 100 bonus points when he 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)showAlert
{
  int difference = abs(_targetValue - _currentValue);
  int points = 100 - difference;

  NSString *title;
  if (difference == 0) {
    title = @"Perfect!";
    points += 100;
  } else if (difference < 5) {
    title = @"You almost had it!";
    if (difference == 1) {
      points += 50;
    }
  } else if (difference < 10) {
    title = @"Pretty good!";
  } else {
    title = @"Not even close...";
  }

  _score += points;

   . . .
}
```

You should notice a few things. In the first `if`, you'll see a new statement between its curly brackets:

```
  if (difference == 0) {
    title = @"Perfect!";
    points += 100;
  } . . .
```

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:

```
  } else if (difference < 5) {
    title = @"You almost had it!";
    if (difference == 1) {
      points += 50;
    }
  } . . .
```

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 this if-statement. After all, if the difference is more than zero but less than five, it could also be one (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.

Finally, the line `_score += points;` has moved below the `if`s. 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.

Waiting for the alert to go away

There is something else that bothers me; you may have noticed it too. As soon as you tap the Hit Me button and the alert view shows 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. This is a little confusing.

It would be better to wait with starting the new round until *after* the player has dismissed the alert view. Only then is the current round truly over. Maybe you're wondering why this isn't already happening? After all, in the `showAlert` method you only call `startNewRound` after you've shown the alert view:

```
- (IBAction)showAlert2
{
  . . .

  UIAlertView *alertView = [[UIAlertView alloc]
    initWithTitle:. . .];

  [alertView show];  // here you make the alert visible

  [self startNewRound];  // here you start the new round
  [self updateLabels];
}
```

Here's the thing: `show` doesn't hold up the execution of this method until the alert view is dismissed (that's how alerts on other platforms tend to work, but not on iOS). Instead, `show` puts the alert on the screen and immediately returns. The rest of the `showAlert` method is then executed right away.

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 view will be done. But it will be well after the `showAlert` method has finished.

So if you can't wait in `showAlert` until the alert view 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 you do in iOS involves waiting for specific events to occur (buttons being tapped, sliders being moved, and so on). This is no different.

You can ask the alert view to send you a message when it is being closed. In the mean time, you simply do nothing. When the user finally taps the OK button on the alert view, the alert will remove itself from the screen and sends you that message. That's your cue to take it from there.

This is also known as the *listener pattern* or *observer pattern*. You make the Bull's Eye view controller listen to events coming from the alert view. In proper iOS terminology such listeners are named *delegates* and that's the term we'll be using from now on.

› Go to the **BullsEyeViewController.h** file and change the `@interface` line to the following:

```
@interface BullsEyeViewController : UIViewController
                                        <UIAlertViewDelegate>
```

(This all goes on one line.)

You're just adding `<UIAlertViewDelegate>` to the existing `@interface` line. This tells the app that your view controller is now a delegate of `UIAlertView`.

➤ Change the bottom bit of `showAlert` to:

```
- (IBAction)showAlert
{
  . . .

  UIAlertView *alertView = [[UIAlertView alloc]
    initWithTitle:title
    message:message
    delegate:self
    cancelButtonTitle:@"OK"
    otherButtonTitles:nil];

  [alertView show];
}
```

The only change is that where it used to say `delegate:nil`, it now says `delegate:self`. This tells the alert view that the `BullsEyeViewController` is now its delegate. You have also removed the calls to `[self startNewRound]` and `[self updateLabels]` because that is the code you want to execute when the alert closes.

➤ Add the following to **BullsEyeViewController.m**. Put it somewhere down the bottom, before the `@end` line:

```
- (void)alertView:(UIAlertView *)alertView
                didDismissWithButtonIndex:(NSInteger)buttonIndex
{
  [self startNewRound];
  [self updateLabels];
}
```

This is the delegate method that is called by the alert view when the user closes it. When you get this message it basically says: "Hi, delegate. The alert view so-and-so was dismissed because the user pressed such-and-such button." Your alert view only has one button but for an alert with multiple buttons you'd look at the `buttonIndex` parameter to figure out which button was pressed. In response to the alert view closing, you start the new round.

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

Delegates in three steps

Delegates are used everywhere in the iOS SDK, so it's good to remember that it always takes three steps to become someone's delegate.

1) You declare yourself capable of being a delegate. Here, you did that by including `<UIAlertViewDelegate>` in your `@interface` line. This tells the

compiler that the view controller can actually handle the notification messages that are sent to it.

2) You let the object in question, in this case the `UIAlertView`, know that you wish to become its delegate. If you forget to tell the alert view that it has a delegate, then it will never send you any notifications.

3) Implement the delegate methods. It makes no point to become a delegate if you're not responding to the messages you're being sent! Often, delegate methods are optional, so you don't need to implement all of them. For example, `UIAlertViewDelegate` actually declares seven different methods but for this app you only care about `alertView:didDismissWithButtonIndex:`.

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. You would use this button if you were playing against another person. The first player does ten rounds, then the score is reset and the second player does ten rounds. The one 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, connect the Start Over button to an action.

➤ Add the following to the **BullsEyeController.h** file, below the other actions:

```
- (IBAction)startOver;
```

➤ Open the storyboard and Ctrl-drag from the **Start Over** button to **Bulls Eye View Controller**. 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.

➤ Add the following to **BullsEyeController.m** below `sliderMoved`:

```
- (IBAction)startOver
{
  [self startNewGame];
  [self updateLabels];
}
```

That looks quite reasonable. If the Start Over button is pressed, you call the `startNewGame` method to start a new game (see, if you choose method names that make sense, then reading source code really isn't that hard).

You haven't programmed `startNewGame` yet, but presumably this method will reset the score and round number and start a new round as well. Because this changes the contents of your instance variables you also call `updateLabels` to update the text of the corresponding score, round and target labels.

➤ Add the `startNewGame` method just below `startNewRound`:

```
- (void)startNewGame
{
  _score = 0;
  _round = 0;
  [self startNewRound];
}
```

Notice that you set `_round` to 0 here. You do that because incrementing `_round` is the first thing `startNewRound` does. If you were to set `_round` to 1 in `startNewGame`, 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 1 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.)

Finally, just to make things consistent, in `viewDidLoad` you should replace the call to `startNewRound` by `startNewGame`. It won't really make any difference to how the app works but it makes the intention of the source code clearer.

➤ Change `viewDidLoad` to:

```
- (void)viewDidLoad
{
  [super viewDidLoad];
  [self startNewGame];
  [self updateLabels];
}
```

➤ Run the app and play a few rounds. If you now press Start Over, the game puts you back at square one.

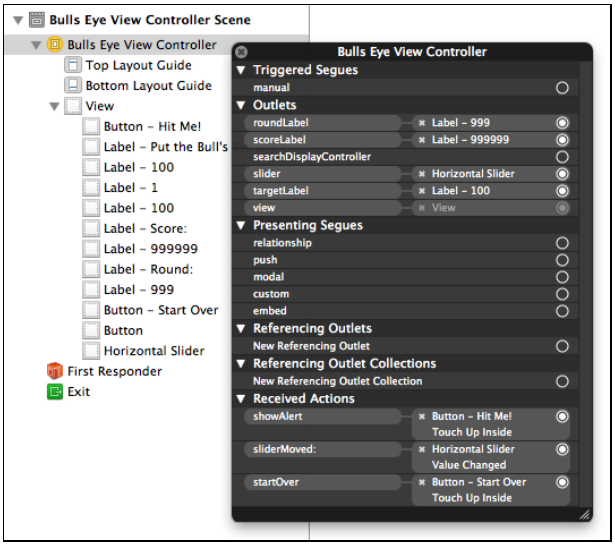
Where to put new methods?

Maybe you're wondering why I asked you to sometimes put new methods below or above another one. Does it matter where you place the methods inside the .m file? The short answer is: No, it does not matter. Anywhere will

do, as long as the methods are placed between the `@implementation` and `@end` directives.

This used to be an issue with older versions of the Objective-C compiler. In the past, you were not able to call a method if the compiler didn't know about it yet. Like humans, the compiler used to read from top-to-bottom. The way to fix this problem was to place the methods in the proper order, or to use so-called forward declarations. Fortunately, that is all ancient history now – although you may still run into this issue when you're defining your own C-style functions.

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

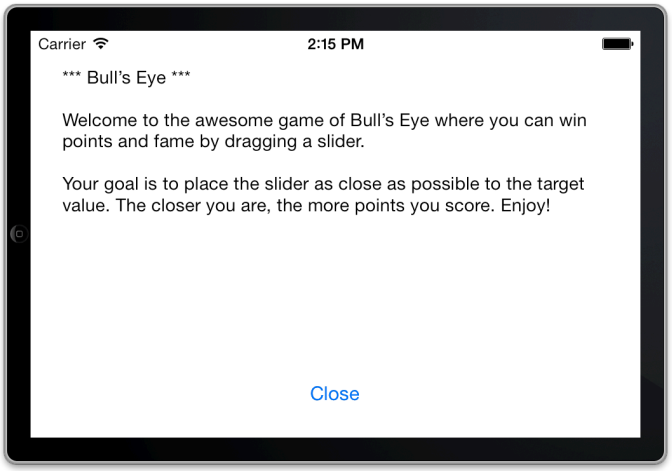


All the connections from Bulls Eye View Controller to the other objects

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 bored 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:



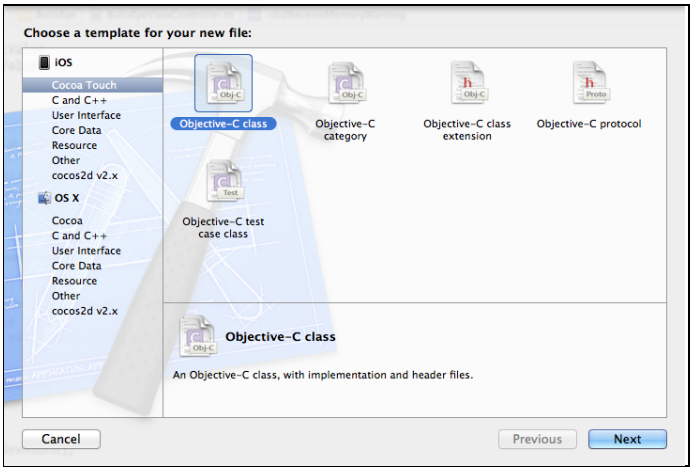
The new About screen

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 on the main game screen.

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.

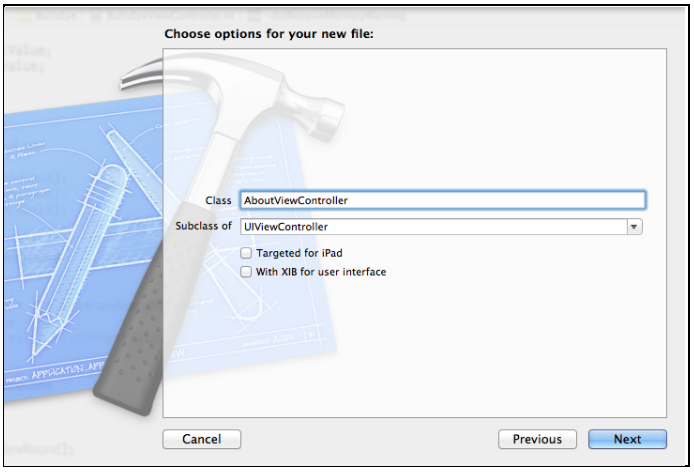
I have pointed it out a few times already: each screen in your app will have its own view controller. If you think "screen", think "view controller". Xcode automatically created the `BullsEyeViewController` for you but the view controller for the About screen you'll have to make yourself.

▸ Go to Xcode's **File** menu and choose **New** → **File...** In the window that pops up, choose the **Objective-C class** template (if you don't see it then make sure **Cocoa Touch** is selected on the left):



Choosing the file template for Objective-C class

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

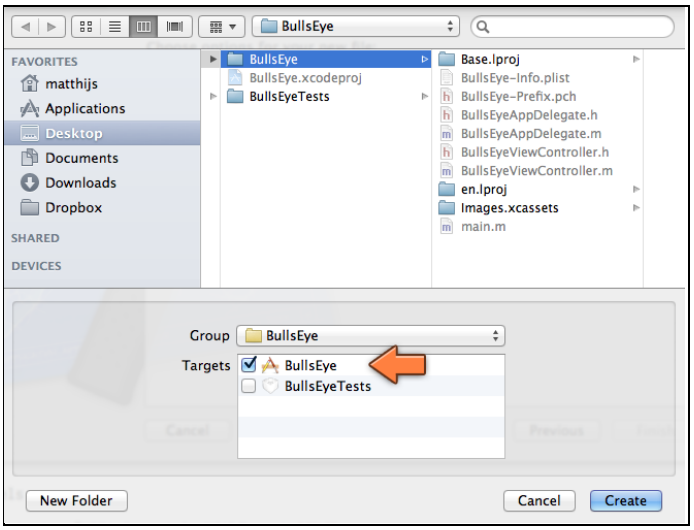


The options for the new file

Choose the following:

- Class: **AboutViewController**
- Subclass of: **UIViewController**
- Targeted for iPad: Leave this box unchecked.
- With XIB for user interface: Leave this box unchecked.

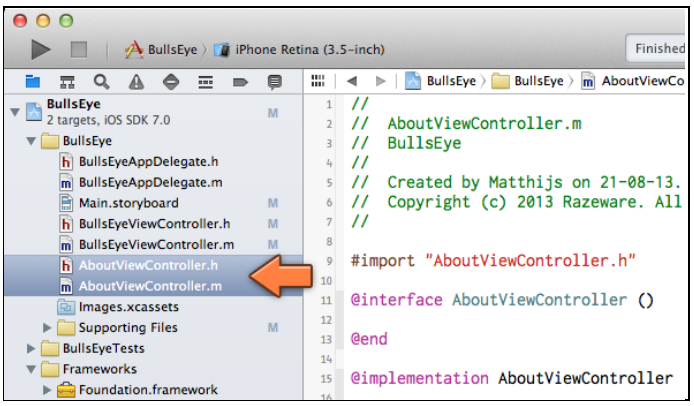
Click **Next**. Xcode will ask you where to save this new view controller:



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**. Click **Create**.

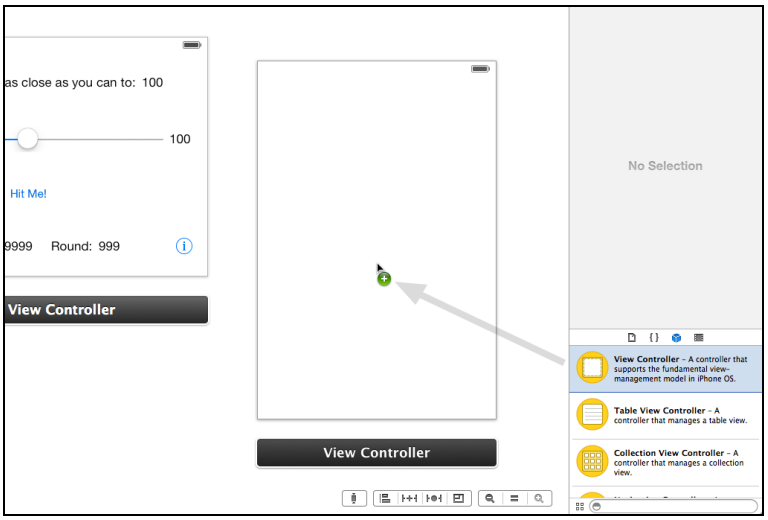
Xcode will make two files and adds them to your project. As you might have guessed, the two files are **AboutViewController.h** and **AboutViewController.m**.



The new files 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 view for this new screen yet, so you will have to add this first.

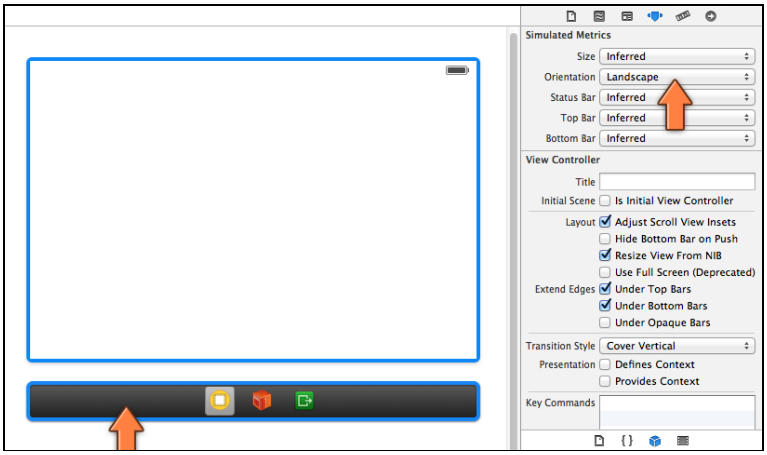
› From the Object Library, choose **View Controller** and drag it into the canvas, to the right of the Bulls Eye View Controller.



Dragging a new View Controller from the Object Library

This new view controller is totally blank and in portrait orientation.

› Click on the black bar below the view controller to select it and go to the **Attributes inspector**. Under **Simulated Metrics**, set **Orientation** to **Landscape**.

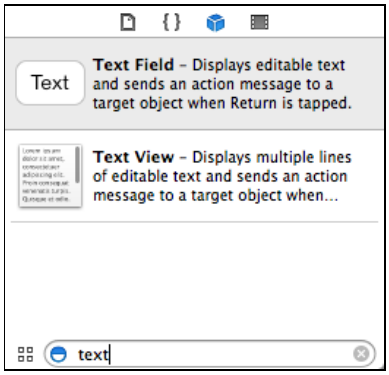


Changing the orientation of the view to landscape

› Drag a new **Button** into the screen and give it the title **Close**. Put it somewhere in the bottom center of the view.

› Drag a **Text View** into the view and make it cover most of the space above the button.

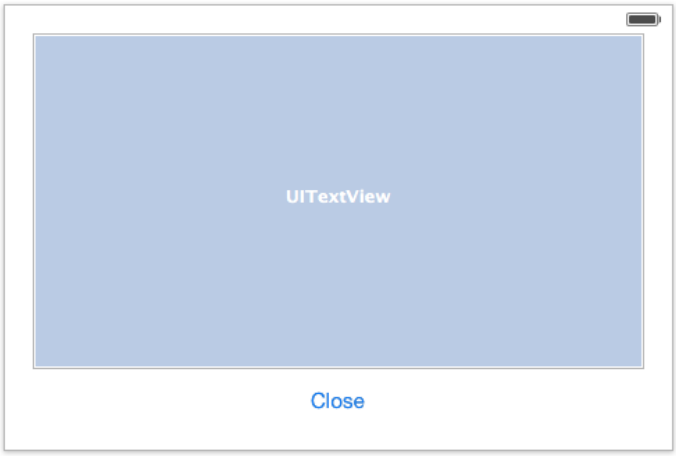
You can find these components in the Object Library. If you don't feel like scrolling, you can filter the components by typing in the field at the bottom:



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 background view, it should look something like this:



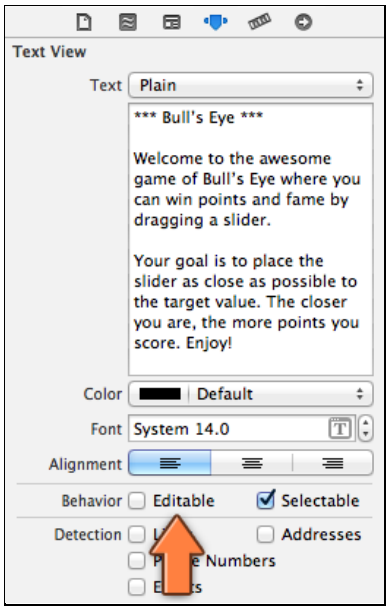
The About screen in the storyboard

➤ 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.

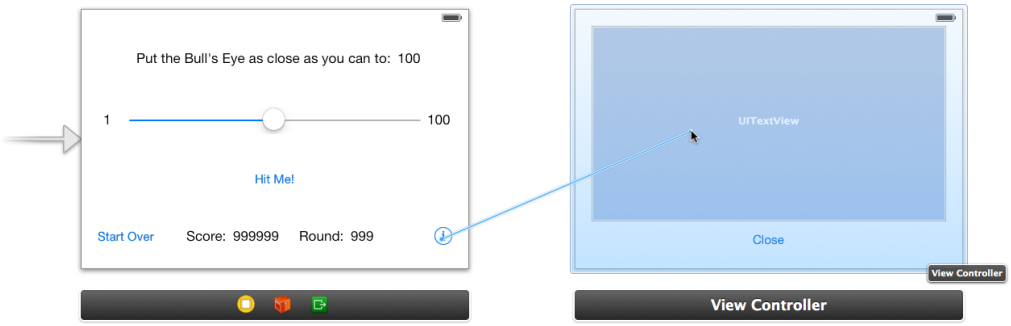
➤ Make sure to uncheck the **Editable** box, otherwise the user can actually type into the text view. For this game it should be set to read-only.



The Attributes inspector for the text view

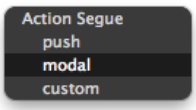
That's the design of the screen finished for now. So how do you open this new About screen when the user presses the (i) button? Storyboards have a neat trick for this: *segues* (pronounced "seg-way" like the silly scooters). A segue is a transition from one screen to another and they are really easy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

› Click the **(i) button** in the **Bulls Eye 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

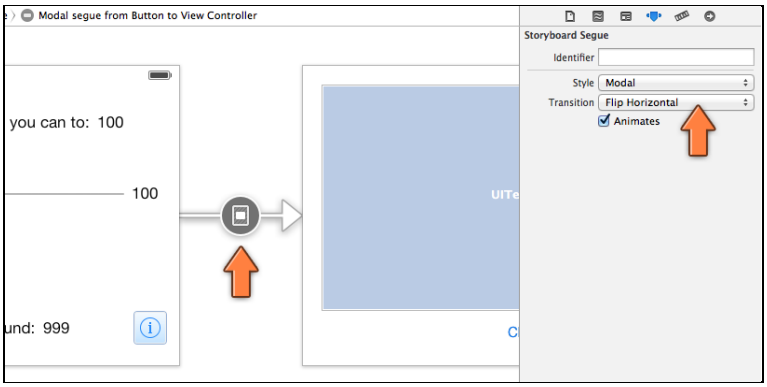
› Let go of the mouse button and a popup appears with several options. Choose **modal**.



Choosing the type of segue to create

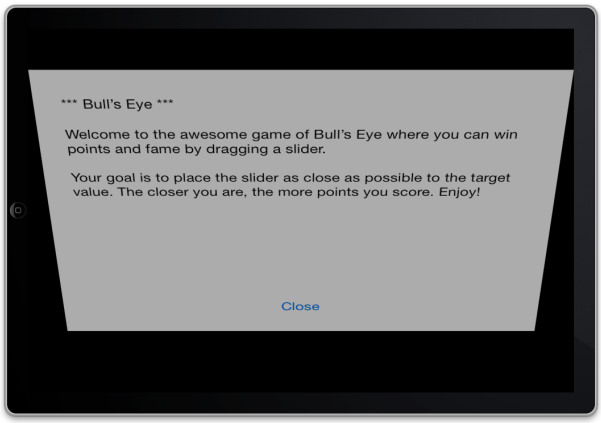
Now an arrow will appear between the two screens. This arrow represents the segue.

➤ 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 `BullsEyeViewController`, because the Close button is part of the About screen, not the main game screen.

➤ Open **AboutViewController.h** and add the following line, just above the `@end` line:

```
- (IBAction)close;
```

➤ Go to **AboutViewController.m** to add the body of this method (anywhere between `@implementation` and `@end` will do):

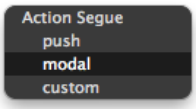
```
- (IBAction)close
{
  [self.presentingViewController
              dismissViewControllerAnimated:YES completion:nil];
}
```

This line tells UIKit to close the About screen with an animation. If you would have said `dismissViewControllerAnimated:NO`, 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.

You should recognize `self.presentingViewController` as a property. It refers to `BullsEyeViewController`. You didn't have to do anything to set this property. UIKit took care of this automatically when it presented the new screen during the segue. UIKit does a lot of stuff for you behind the scenes; you just have to know how to tap into it.

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 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:



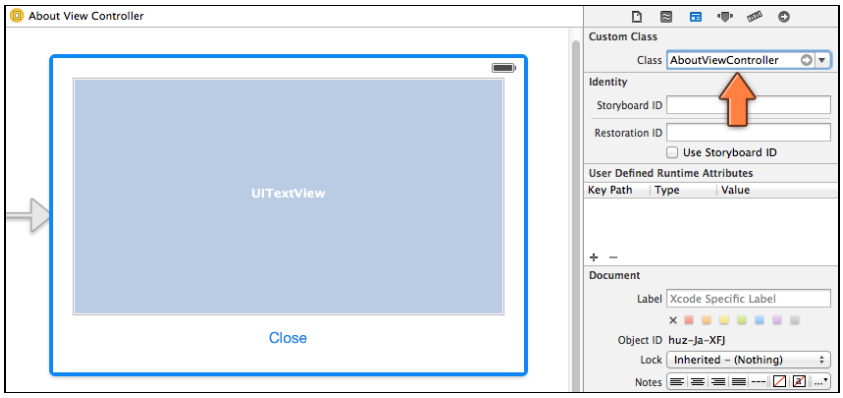
The "close" action is not listed in the popup

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

The problem is that this new view controller in the storyboard does not know that it is supposed to be the `AboutViewController`. You first added the .h and .m source files for `AboutViewController` and then dragged this new view controller into the storyboard, but you haven't told the storyboard that the design for this this new view controller, in fact, belongs to `AboutViewController`. (That's why in the outline pane it says View Controller and not About View Controller.)

› Fortunately, this is easily remedied. Select the **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 fat blue border.)

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

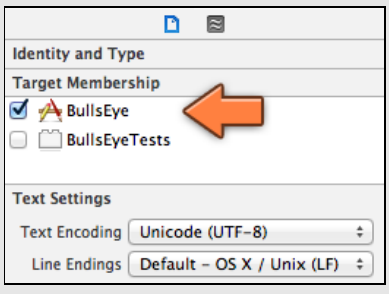
› Ctrl-drag from the **Close** button to **About View Controller**. This should be old hat by now. The popup menu now does have an option for the **close** action. Connect the button to the that action.

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

Do you get this warning message in the Debug output pane?

```
Unknown class AboutViewController in Interface Builder file.
```

And does the app crash when you tap the Close button? In that case, something went wrong when you added the `AboutViewController` source files to the project. This is easy enough to fix.



> ➤ In the Project navigator, select **AboutViewController.m** and open the **File inspector**. This is the first of the inspector tabs. Under **Target Membership**, make sure **BullsEye** is selected.

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.

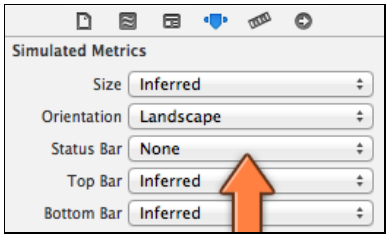
You can find the project files for the app up to this point under **06 - About Screen** in the tutorial's Source Code folder.

Making it look good

So far, the app has kept the iPhone's status bar on top. That's a good idea for most apps but games require a more immersive experience and the status bar detracts from that. So let's get rid of it. It is quite easy once you know how.

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

➤ Open **Main.storyboard** and select the **Bulls Eye View Controller**. Go to the **Attributes inspector** and under **Simulated Metrics** set **Status Bar** to **None**. This removes the status bar from the view. The status bar should automatically disappear from the About View Controller as well.



Remove the status bar from the view controller

Just removing the status bar from the simulated metrics is not enough. If you run the app now, the status bar is still present. There is a reason why this section is labeled *Simulated* Metrics, as the status bar isn't really part of the view. 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.

To remove the status bar during runtime, you also have to make a small change to the code.

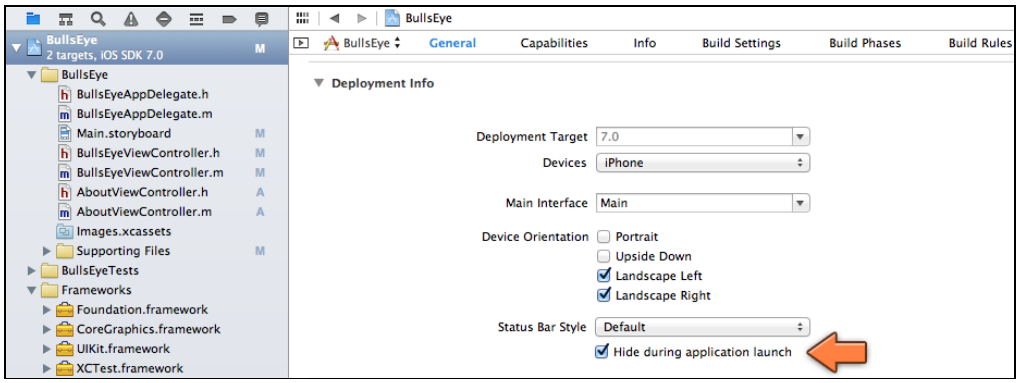
➤ Go to **BullsEyeViewController.m** and add the following method:

```
- (BOOL)prefersStatusBarHidden
{
  return YES;
}
```

If this particular method is present in your view controller, UIKit will query it just before it presents the view controller on the screen. UIKit asks the view controller: "Do you want to hide the status bar?" If your view controller says "yes", then the status bar will not be visible.

▸ Also add this method to **AboutViewController.m**. If you forget to do this, then the status bar will reappear again on the About screen.

The final step to remove the status bar is to make a change to the app's configuration. Go to the **Project Settings** screen and under **Deployment Info**, **Status Bar Style**, check the option **Hide during application launch**.



Hiding the status bar when the app launches

You need to hide the status bar while the app is launching, because otherwise iOS doesn't know you want to do this until `BullsEyeViewController` is active. 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.

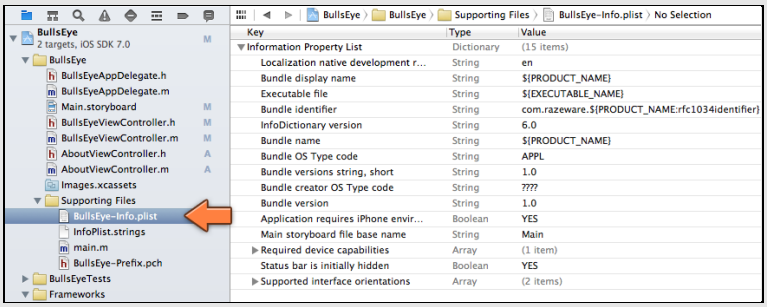
Info.plist

Most of the options from the Target Settings screen 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 Xcode 5 this is hardly necessary anymore. You can make most of the changes

directly from the Target Settings screen. However, it's good to know that Info.plist exists and what it looks like.

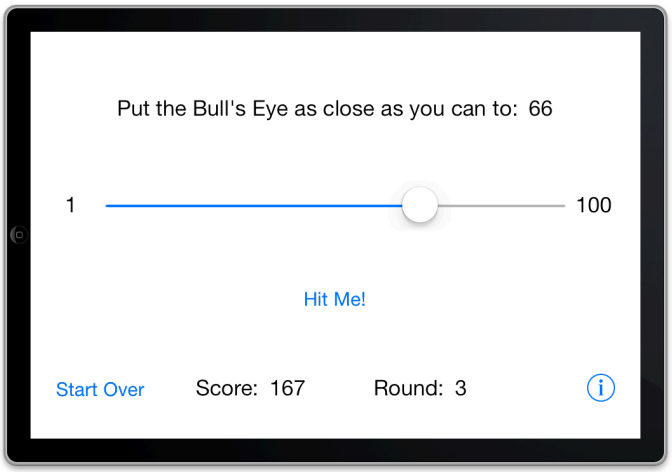
› Go to the **Project navigator**. Under the group **Supporting Files** you will find a file named **BullsEye-Info.plist**. Click BullsEye-Info.plist to open it and Xcode's main edit pane will show you its contents.



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.

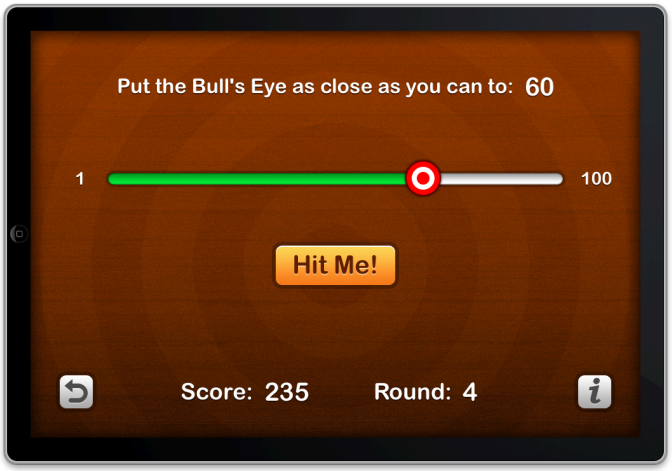
Spicing up the graphics

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



Yawn...

to this:



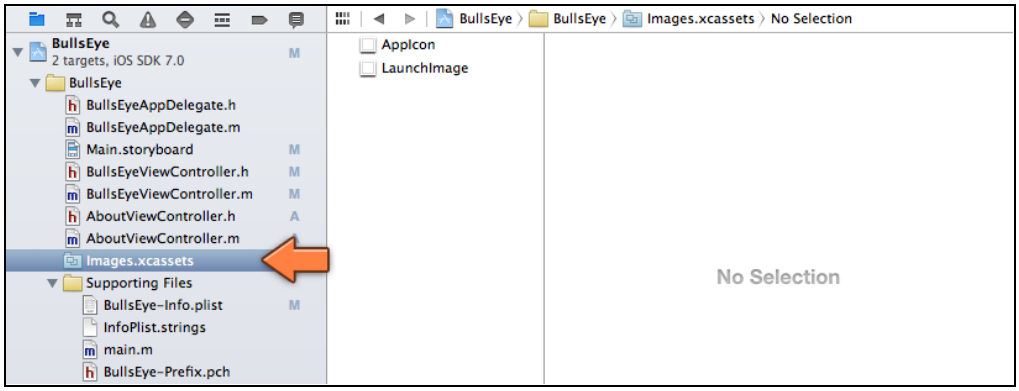
Cool ☺

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.

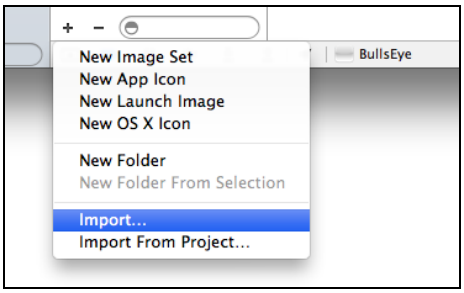
The Resources folder that comes with this tutorial contains a subfolder named Images. You will first import these images into the BullsEye project.

› In the **Project navigator**, find **Images.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's icon and launch images, both of which you'll add soon.



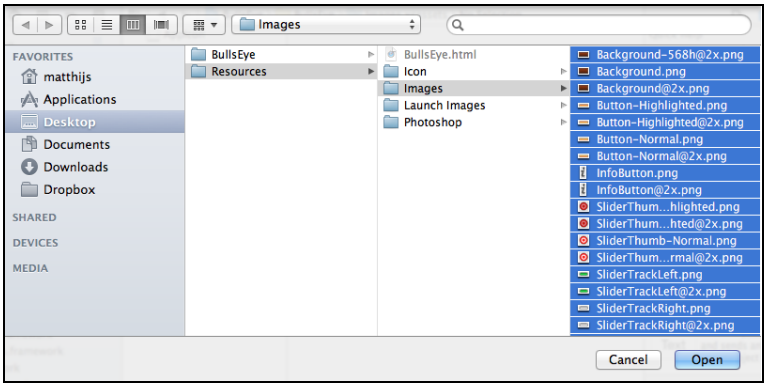
The asset catalog is initially empty

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



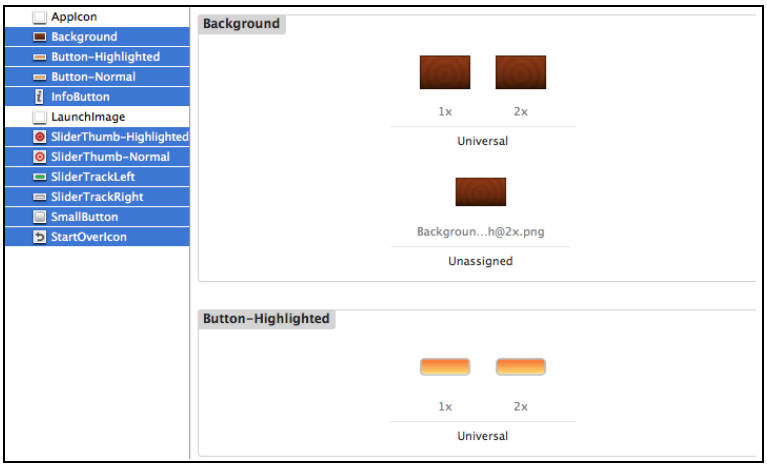
Choose Import to put existing images into the asset catalog

Now 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.

Retina displays

Notice that all of the images in the asset catalog have two variants: 1x and 2x. This is necessary because your apps may need to support two types of displays: low-resolution and Retina.

If you obtained your iPhone recently you'll probably have an iPhone 5 or 4S with a Retina display. I love mine, it's a very cool gadget and the sharpness of its display is amazing. That's because it packs a lot of pixels in a very small space. It becomes almost impossible to make out where one pixel ends and the next one begins, that's how miniscule they are. Compared to this Retina display, the older iPhone models look very "blocky".

Considering that Retina screens have been around for a few years now, and that all older iPhone models with low-resolution screens (such as the 3GS) cannot run iOS 7, it would be great if your apps only needed to support Retina screens. There is just one "but": the iPad 2 and iPad mini. These devices do run iOS 7 but they do not have a Retina screen. If you are making an iPad app – or a universal app that supports both the iPhone and the iPad – then you still need to provide 1x images to accommodate the iPad 2 and iPad mini.

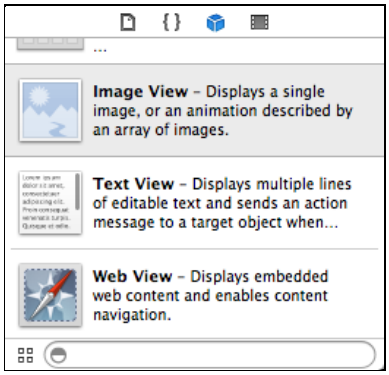
Fortunately, the asset catalog makes this easy. Simply drop the low-resolution image into the 1x slot and the Retina image into the 2x slot. As the names imply, the Retina image is twice as big as the low-res image. So how did the asset catalog know which were the Retina images when you imported them just now? There is a special naming convention for image files: if the filename ends in **@2x**, then that is considered the Retina version.

Bull's Eye is not a universal app, so strictly speaking the 1x images are not necessary. You can still run it on the iPad, even if the app is not universal. The iPad can run all iPhone apps in a special emulation mode. In that case, the app also uses the 2x images.

So if the app you're making is only for the iPhone and for iOS 7 or higher, then you can leave out the 1x images. In all other cases, you will still need to keep supporting low-resolution displays for a while. The reason the tutorials from *The iOS Apprentice* supply 1x images anyway is that this looks better in Interface Builder, which – for historical reasons – shows the interface of the app in low-resolution mode.

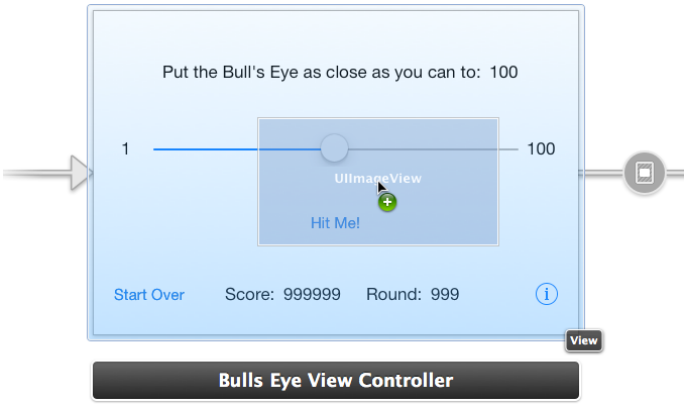
Let's begin by changing the drab white background into something more fancy.

› 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

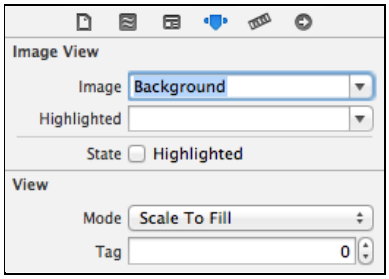
› Drag the image view on top of the existing user interface. It doesn't really matter where you put it, as long as it's inside the Bull's Eye View Controller.



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 480 and Height to 320. This will make the image view cover the entire screen.

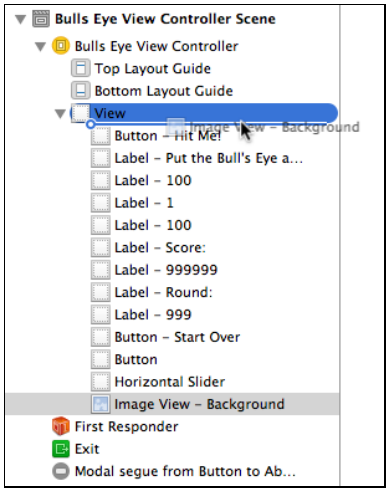
› Go to the **Attributes inspector** for the image view. At the top there is an option named **Image**. Click the downward arrow and choose **Background** from the list. This will put the image named Background.png into the image view.



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). In that case, pick up the image view in the outline pane and drag it to the top to accomplish the same thing.



Put the Image View at the top to make it appear behind the other views

› 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.

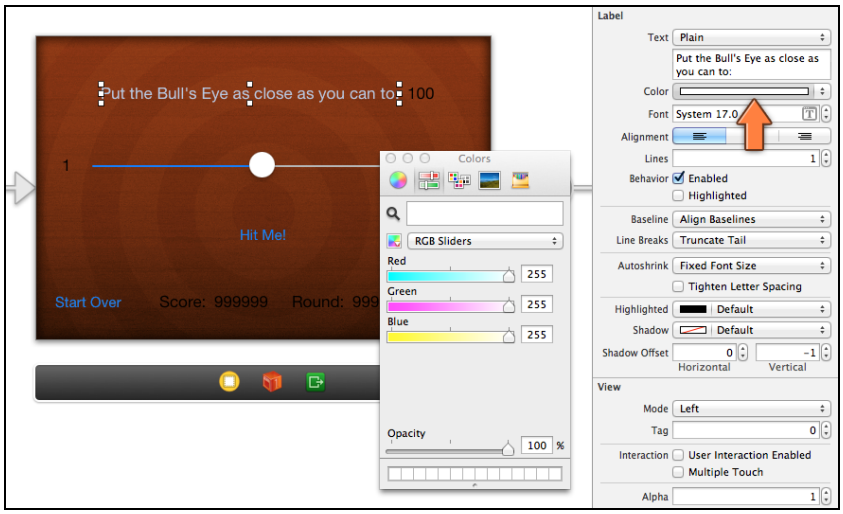
The game with the new background image

Note: When you build the app, you may get a warning message: "Ambiguous Content: The image set "Background" has an unassigned image." Don't worry about that for now. You will fix that soon.

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. There are several types of ways to select colors. (If all you see is a grayscale 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%.

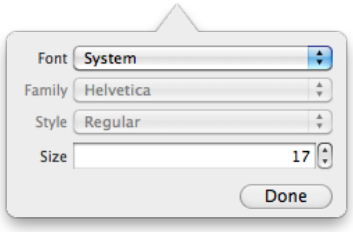
› Click on the **Shadow** item from the Attributes inspector. This lets you add a subtle shadow to the label. By default this color is transparent (also known as "Clear Color") so you won't see the shadow. Using the Color Picker, choose a pure black color that is half transparent, Red: 0, Green: 0, Blue: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

Note: Sometimes when you change the Color or Shadow properties, the background color of the view also changes. I think 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 under the label.

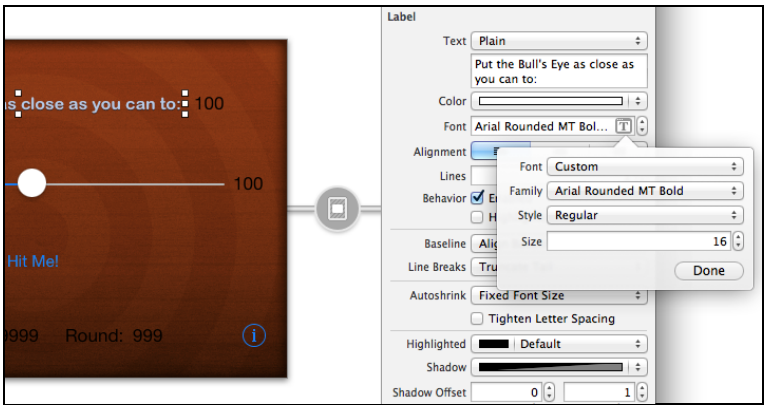
The shadow you've chosen is very subtle. If you're not sure that it's actually visible, then toggle the vertical offset between 1 and 0 a few times. Look closely and you should be able to see the difference. As I said, it's very subtle.

› 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 currently is Helvetica Neue.



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. The label will now become slightly smaller so that it fits snugly around the text.

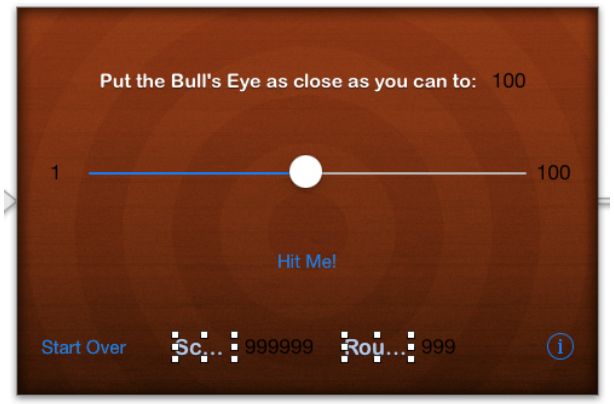
You don't have to set these properties for the other labels one by one; that would be a big chore. You can speed up the process by selecting multiple labels and then applying these changes to that entire selection.

› 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.

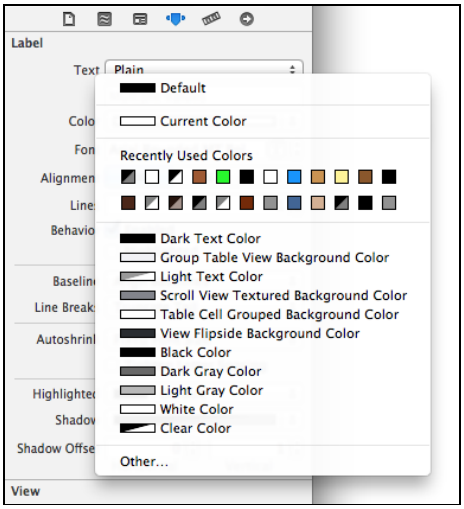
As you can see, in my storyboard the text no longer fits into the Score and Round labels:



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:

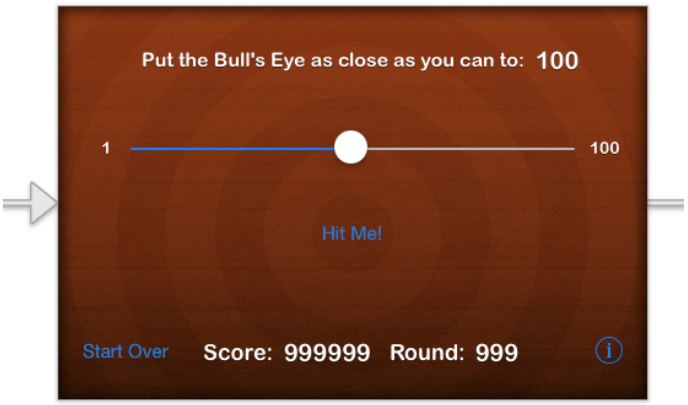


Quick access to recently used colors and several handy presets

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 just as much as it is mine.

The buttons

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

› Select the **Hit Me** button and open 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.

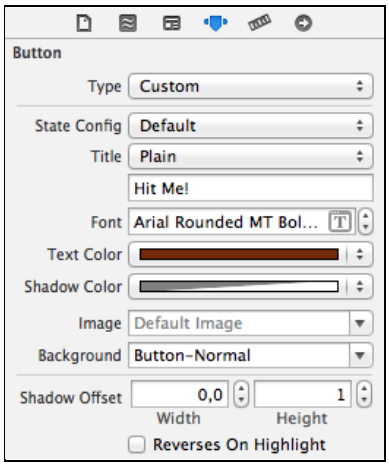
› Set the **Text Color** to red: 96, green: 30, blue: 0, opacity: 100%. This is a dark brown color.

› Set the **Shadow Color** to pure white, 50% opacity and the shadow offset as before to 0 horizontal and 1 vertical.

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:



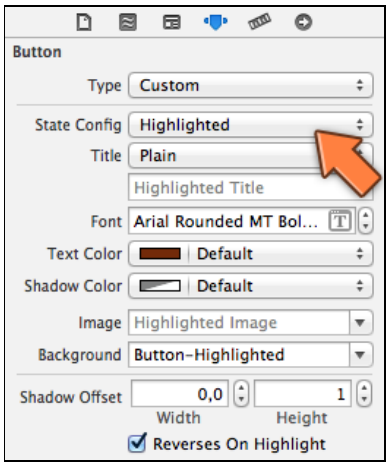
The attributes for the Hit Me button in the default state

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 it is an important visual clue to the user.

› With the button still selected, click the **State Config** setting and pick **Highlighted** from the menu. In the **Background** field, select **Button-Highlighted**.

› Make sure the highlighted **Text Color** is the same color as before (R 96, G 30, B 0, or simply pick it from the Recently Used Colors menu). Change the **Shadow Color** to half-transparent white again. Also 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 that you will replace its title text by an icon.

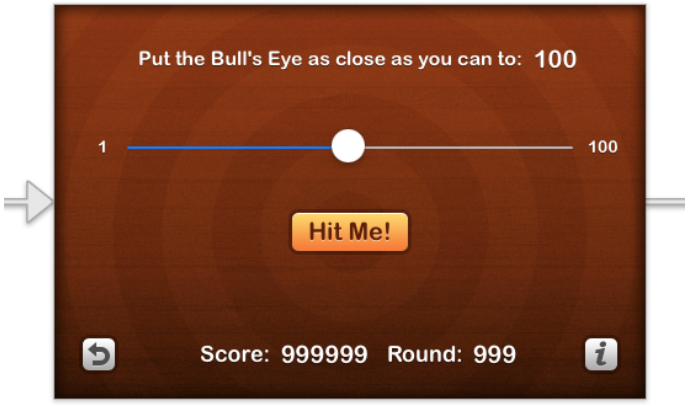
➤ Select the **Start Over** button and change the following attributes:

- Set Type to Custom.
- Clear out the Title (remove the text "Start Over").
- 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.

Note that 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:forState:` 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.

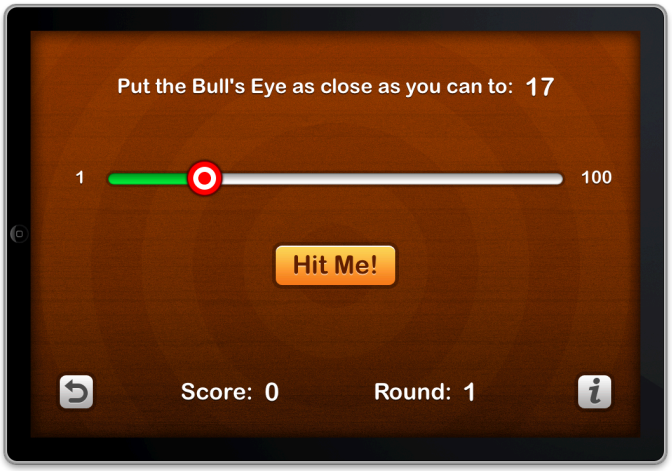
➤ Go to **BullsEyeViewController.m**, and add the following to `viewDidLoad`:

```
UIImage *thumbImageNormal = [UIImage
                               imageNamed:@"SliderThumb-Normal"];
[self.slider setThumbImage:thumbImageNormal
                  forState:UIControlStateNormal];

UIImage *thumbImageHighlighted = [UIImage
                           imageNamed:@"SliderThumb-Highlighted"];
[self.slider setThumbImage:thumbImageHighlighted
                  forState:UIControlStateHighlighted];

UIImage *trackLeftImage =
  [[UIImage imageNamed:@"SliderTrackLeft"]
    resizableImageWithCapInsets:UIEdgeInsetsMake(0, 14, 0, 14)];

[self.slider setMinimumTrackImage:trackLeftImage
                           forState:UIControlStateNormal];

UIImage *trackRightImage =
  [[UIImage imageNamed:@"SliderTrackRight"]
```

```
    resizableImageWithCapInsets:UIEdgeInsetsMake(0, 14, 0, 14)];

[self.slider setMaximumTrackImage:trackRightImage
                         forState:UIControlStateNormal];
```

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

Note that on disk the images are named **SliderThumb-Normal.png** and so on, but that you leave out the **.png** extension when you create the `UIImage` object. If you look inside the asset catalog, you will also see that the images no longer have a file extension. You use the name that is listed in the asset catalog to refer to the image. (However, it would still have worked if you did specify the .png extension. UIKit is smart enough to figure out that you meant to use the image from the catalog.)

Using a web view for HTML content

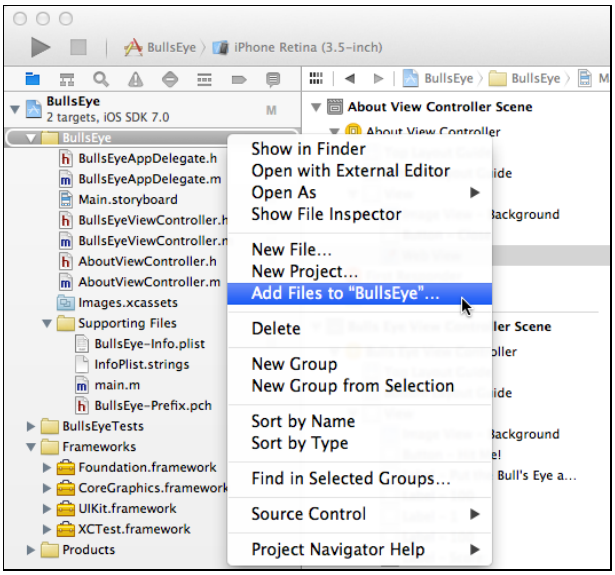
The About screen could still use some work.

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).

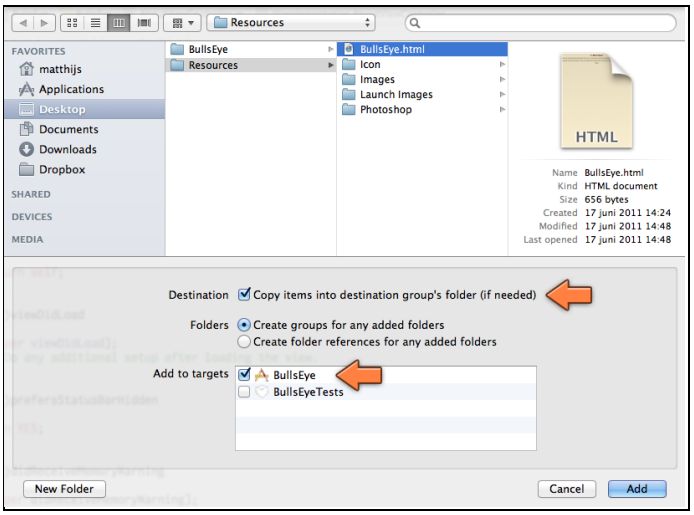
A web view, as its name implies, can show web pages. All you have to do is give it a URL to a web site. However, for this app you will make it display a static HTML page from the application bundle, so it won't actually have to go onto the web and download anything. The web view object is named `UIWebView`.

› 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 into destination group's folder (if needed)** is selected and that under **Add to targets**, there is a checkmark in front of **BullsEye** (you can leave BullsEyeTests unchecked). Press **Add** to add the HTML file to the project.

▸ In **AboutViewController.h**, add an outlet property for the web view. You do this between the `@interface` and `@end` lines:

```
@property (nonatomic, weak) IBOutlet UIWebView *webView;
```

▸ 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** 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.m**, change `viewDidLoad` to:

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  NSString *htmlFile = [[NSBundle mainBundle]
                    pathForResource:@"BullsEye" ofType:@"html"];

  NSData *htmlData = [NSData dataWithContentsOfFile:htmlFile];

  NSURL *baseURL = [NSURL fileURLWithPath:[
                              [NSBundle mainBundle] bundlePath]];

  [self.webView loadData:htmlData MIMEType:@"text/html"
        textEncodingName:@"UTF-8" baseURL:baseURL];
}
```

This loads the local HTML file into the web view. This bit of 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 an `NSData` object, and finally asks the web view to show the contents of this data object.

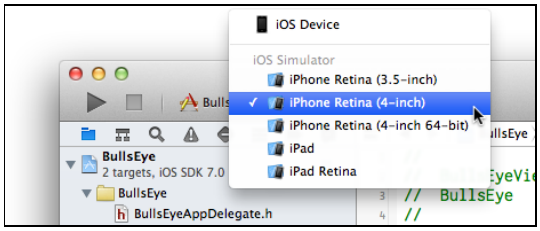
▸ Run the app and press the info button. The About screen should appear with a description of the gameplay rules, this time in the form of an HTML document:

The About screen in all its glory

Supporting 4-inch screens

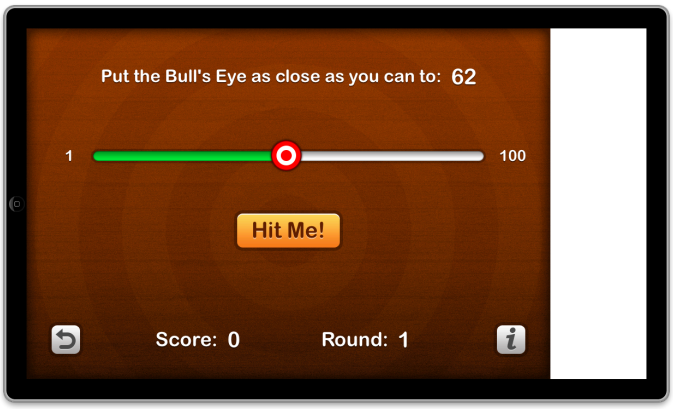
So far you have designed the app for the 3.5-inch screen of the iPhone 4 and 4S, but what about the newer models such as the iPhone 5s and 5c? They have a slightly taller screen, also known as the "4-inch screen". Where the 3.5-inch Retina phones are 960 pixels tall, the 4.inch models have 1136 pixels. That's a difference of 176 pixels that your apps have to fill up somehow. In this section you'll learn how to do that for Bull's Eye.

› Run the app on the 4-inch Simulator. You can switch between Simulators using the scheme selector at the top of the Xcode window:



Using the scheme selector to switch to the 4-inch Simulator

The app will work just fine, but it does have an ugly white bar at the right-hand portion of the screen. This is the extra screen space that an iPhone 5 has.



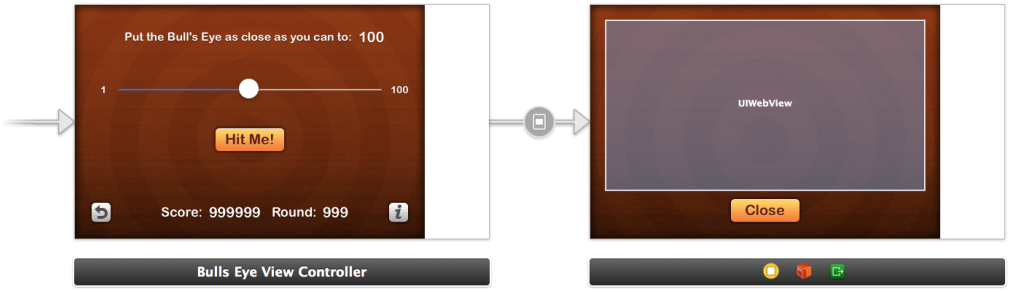
On the 4-inch Simulator, the app doesn't fill up the entire screen

Obviously, this won't do. You need to do something that extra space. Fortunately, UIKit comes with a technology called *Auto Layout* that is designed to make this easy.

› Open **Main.storyboard** and click on the tiny **Form Factor** button at the bottom to switch to 4-inch mode.

Switching between the 3.5-inch and 4-inch form factors in the storyboard

The storyboard now looks like this, sporting the same white strip of nothingness (you may need to rearrange it a little to make everything fit):

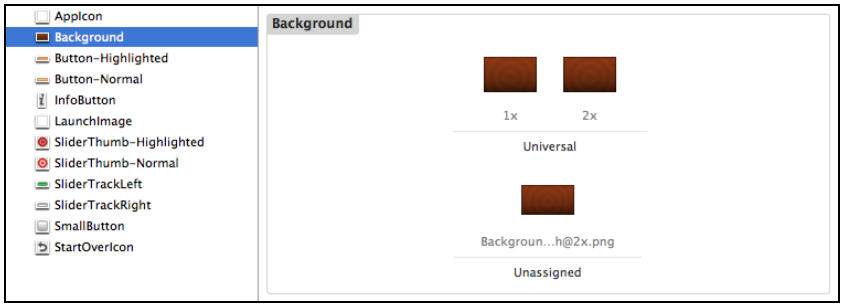


The storyboard in 4-inch mode

› First, let's fix the background image on the main screen. Grab its handles and resize it so that it fills the entire view. It will now be 568 points wide (remember, on a Retina screen one point counts as two pixels, so 1136 pixels / 2 = 568 points).

This is not enough to fix the problem. The image now looks stretched out on a 4-inch screen – it doesn't have enough pixels, so UIKit stretches it out – and it gets cut off on a 3.5-inch screen. You can see that by toggling the Form Factor button.

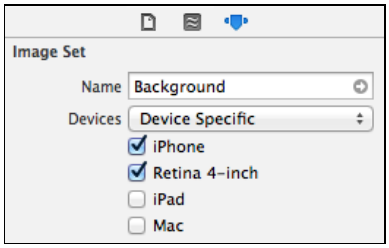
What you need is a second background image that was designed for 640×1136 pixels. As it turns out, you have already added such an image to the asset catalog (the image named **Background-568h@2x.png** in the Images folder from this tutorial's resources). It currently looks like this in the asset catalog (click on **Images.xcassets** to view):



The background image in the asset catalog

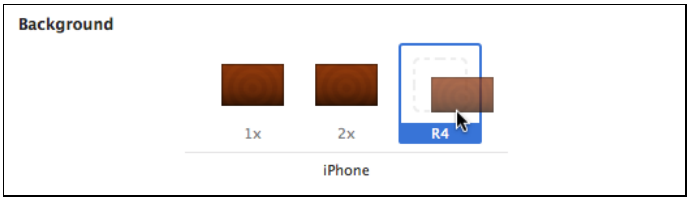
The Background set contains a 1x image (**Background.png)**, a 2x image (**Background@2x.png**) and an image that is listed as "Unassigned". Xcode has been giving a warning message about this unassigned image ever since you added these images to the project.

› Select the **Background** image set and open the **Attributes inspector**. Under **Devices**, change from **Universal** to **Device Specific**. De-select **iPad** and check **Retina 4-inch**.



The attributes for the background image in the asset catalog

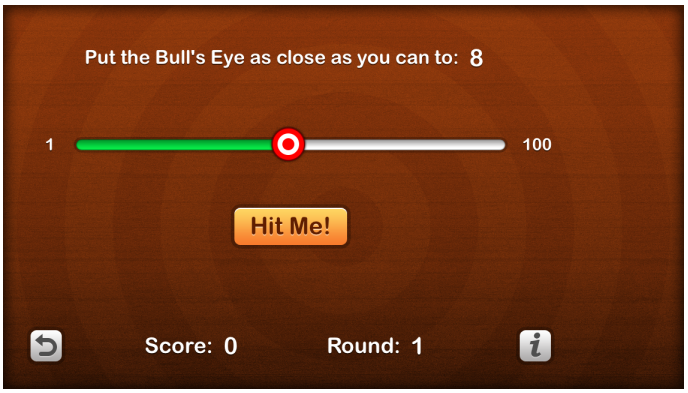
Now drag from the Unassigned image to the new **R4** slot under iPhone:



Dragging the image into the empty R4 slot

I'm not sure what "R4" stands for, but Retina 4-inch sounds reasonable. That is the image you want to display on 4-inch devices.

› Run the app on the 4-inch Simulator. Now it does use the proper background image. You can tell because the circles really are circles, not stretched-out ovals. Compare it to the PNG file from the Images folder if you want to be sure.



The 4-inch Simulator uses the proper background image

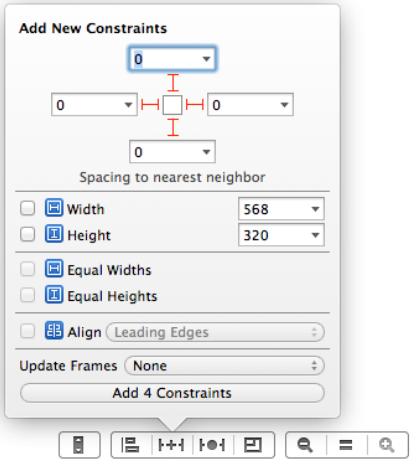
› Now try the app on the 3.5-inch Simulator (again, switch using the scheme selector at the top of the Xcode window). Whoops, that still doesn't look good.

Funnily enough, the app really does use the proper image now, but the image view is still 568 points wide. Because the image view is wider than the visible screen, the image gets stretched out.

This is where Auto Layout comes to the rescue.

› Open the storyboard. (Note that the storyboard doesn't automatically show the R4 image in 4-inch mode, which is a bit confusing. I consider this a bug in Interface Builder.)

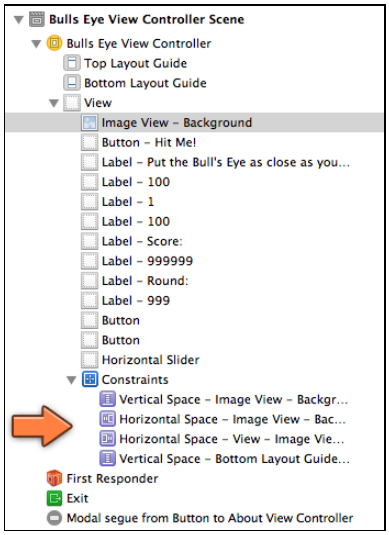
› Select the **image view** in the **Bulls Eye View Controller** and click the small **Pin** button at the bottom of the window:



The Pin menu for adding Auto Layout constraints

This menu lets you "pin" a view to its neighboring views. For the background image view, you want it to always have the same width and height as the view controller itself. The way to do this with Auto Layout is to pin the four sides of the image view to the four sides of the main view. In other words, the top of the image view will always be in the same spot as the top of the main view. Likewise for the bottom and the left and right sides. If the main view changes size, then the image view will follow.

› In the **Pin menu**, click the four T-bars at the top so they become solid red and make sure that all the spacing fields are 0. Then click **Add 4 Constraints**. You won't see much change in the view itself, but in the Outline pane there is now a new item called **Constraints**:



The new Auto Layout constraints appear in the Outline pane

There should be four constraints listed here, two Horizontal Space constraints and two Vertical Space constraints. 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.

There are different types of constraints. The Horizontal and Vertical Space constraints that you're using here make sure that there always is a certain amount of space between the two views. In this case, you set the values of these constraints to 0, which means the views always sit side-by-side, with no extra space between them. Here, that means the edges of the image view always line up with the edges of the window.

› Now run the app again on both the 3.5-inch and the 4-inch Simulator. In both cases, the background should look exactly like their respective images from the resource's folder: **Background@2x.png** for 3.5-inch and **Background-568h@2x.png** for 4-inch.

Let's repeat this for the About screen:

› In the storyboard, resize the **image view** so that it becomes 568 points wide and fills up the entire view.

› Use the **Pin menu** to add the four constraints.

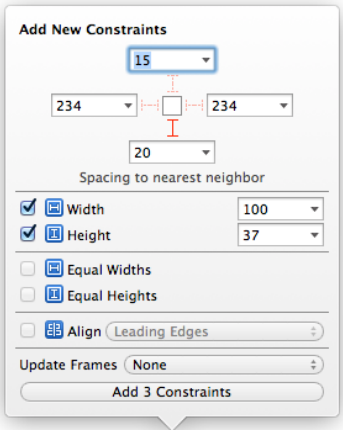
› Run the app to verify that it works. It's a bit harder to see the background image here because the Web View is in the way. (You can temporarily hide it by checking the **Hidden** option inside the **Attributes inspector**.)

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.

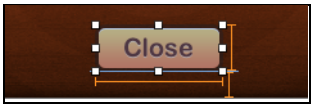
The blue lines are guides that help you position your UI elements

› From the **Pin menu**, select only the T-bar for spacing to the bottom (at 20 points). Also select Width (100) and Height (37). This means you want the Close button to sit at a distance of 20 points from the bottom of the screen, and give it a certain width and height. Click **Add 3 Constraints** to finish.



Pinning the Close button to the bottom

Interface Builder now draws three orange T-bars around the button:



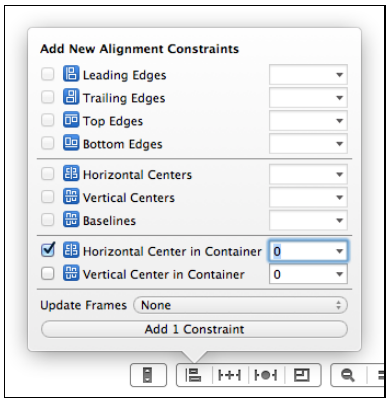
The Close button has orange constraints

The T-bars represent the constraints, of which the button now has three: a Width constraint that says, "This button is always 100 points wide", a Height constraint that says, "This button is always 37 points tall", and a Vertical Space constraint that keeps the button at 20 points distance from the bottom of the screen.

There is one problem: the T-bars are supposed to be blue, not orange. Orange 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. Here, the Close button does have constraints for its size – both width and height – but for its position only in the vertical direction (the Y-coordinate). You also need to add a constraint for its X-coordinate.

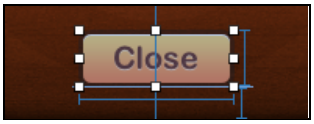
For this app, it will look best if the Close button is always centered horizontally.

➤ Click the **Close** button to select it. From the **Align** menu (the one next to the Pin menu), choose **Horizontal Center in Container** and click **Add 1 Constraint**.



Creating a horizontal centering constraint

Now the constraints all turn blue, meaning that everything is OK:



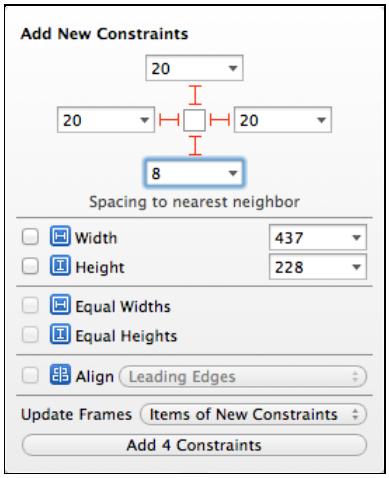
The constraints on the Close button are valid

➤ Run the app. The Close button should always be centered now, regardless of whether you're on the 3.5-inch or the 4-inch Simulator.

> What happens if you don't add any constraints to your views? In that case, Xcode will automatically add constraints for you 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 larger 4-inch screen. If you want that to happen, then it's up to you to add the constraints. As soon as you add one constraint to a view, 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.

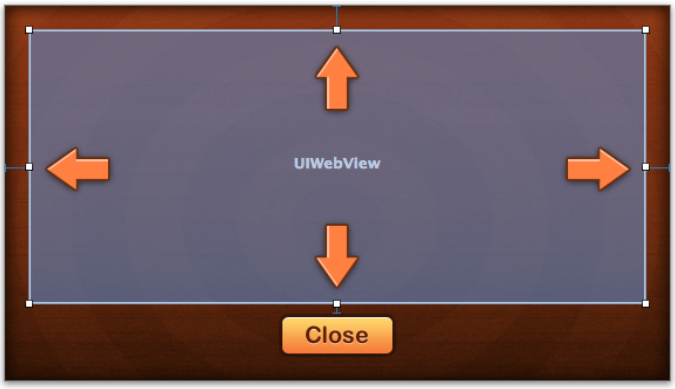
› Select the **web view** and open the **Pin** menu. Select all four T-bars and set them to 20 points, except the bottom one which is 8 points:



Creating the constraints for the web view

› This time, for **Update Frames** select **Items of New Constraints** and then click **Add 4 Constraints**. This option adds the constraints as before and it also fixes the size and position of the web view accordingly. That saves you some dragging.

There are now four constraints on the web view:



The four constraints on the web view

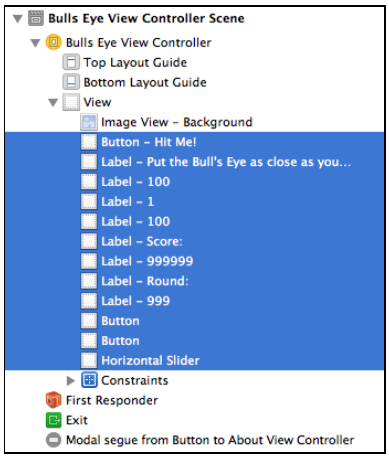
Three of these connect the web view to the main view, so that it always resizes along with it, and one connects it to the Close button. This is enough to determine the size and position of the web view in any scenario.

➤ Run the app on the different Simulators and verify that the Web View always looks good.

The About screen on the 4-inch Simulator

Back to the main game screen, which still needs some work to look good on the larger screen size. You will do this by placing all the labels, buttons and the slider into a new "container" view, and then center that container view in the screen.

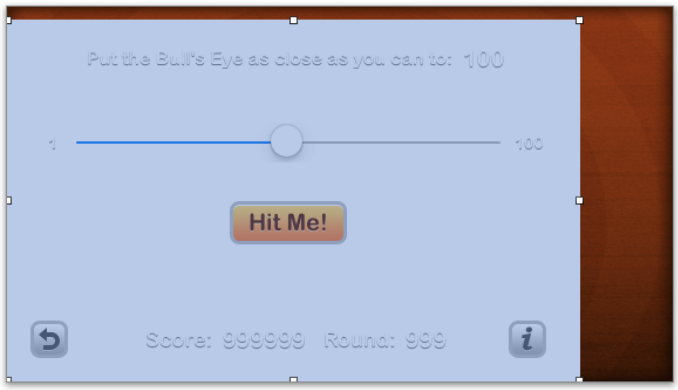
➤ Select all the labels, buttons, and the slider. You can hold down ⌘ and click them individually but an easier method is to go to the **Outline pane**, click on the first view (the Hit Me button), hold down Shift and click on the last view (in my case the slider):



Selecting the views from the Outline pane

You should have selected everything but the 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 a little easier to add the constraints.

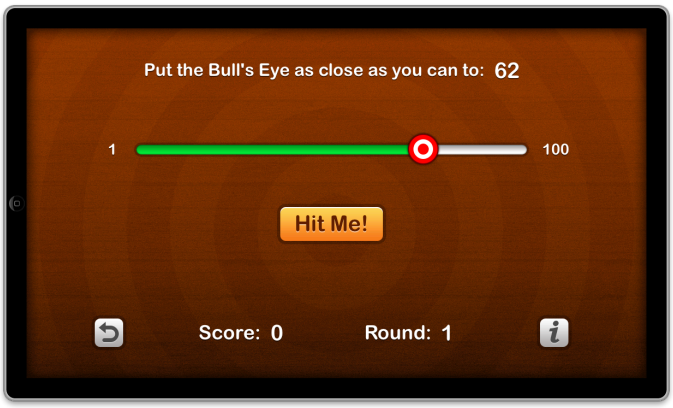
➤ Select the **container view** and open the **Pin menu**. Put checkboxes in front of **Width** and **Height** in order to make constraints for them. Click the button to add the constraints.

➤ With the container view still selected, open the **Align menu**. Check the **Horizontal Center in Container** and **Vertical Center in Container** options. For **Update Frames**, select **Items of New Constraints**; then click **Add 2 Constraints** to finish.

➤ Finally, change the **Background** color of the view to **Clear Color**.

You now have a layout that works correctly on both the 3.5-inch and 4-inch screens!

➤ Try it out:



The game running on the 4-inch Simulator

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. My advice is to ignore Auto Layout when you don't need it and let Xcode add the constraints for you, and only use it when all other options aren't good enough. You'll learn more about Auto Layout, and the alternatives, in the other parts of *The iOS Apprentice*.

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.

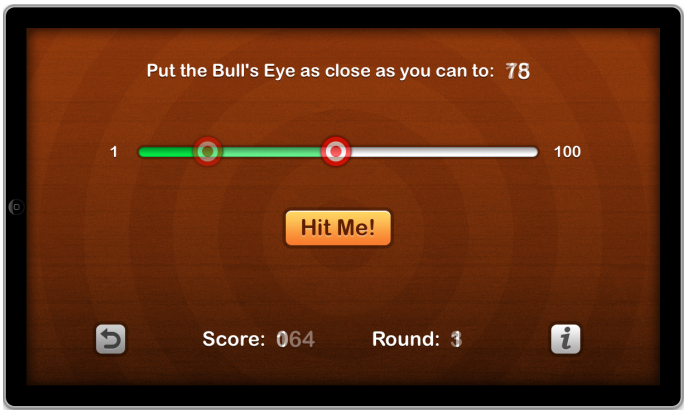
➤ In **BullsEyeViewController.m**, change the `startOver` method to:

```
- (IBAction)startOver
{
  CATransition *transition = [CATransition animation];
  transition.type = kCATransitionFade;
  transition.duration = 1;
  transition.timingFunction = [CAMediaTimingFunction
                functionWithName:kCAMediaTimingFunctionEaseOut];

  [self startNewGame];
  [self updateLabels];

  [self.view.layer addAnimation: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 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.

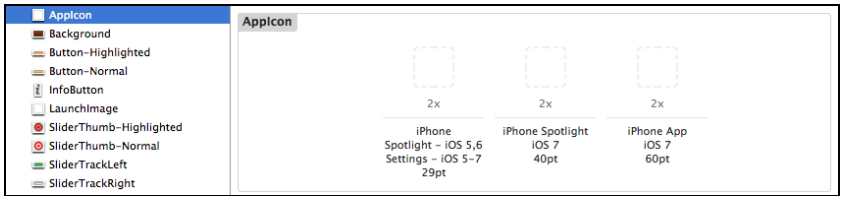


The screen crossfades between the old and new states

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!

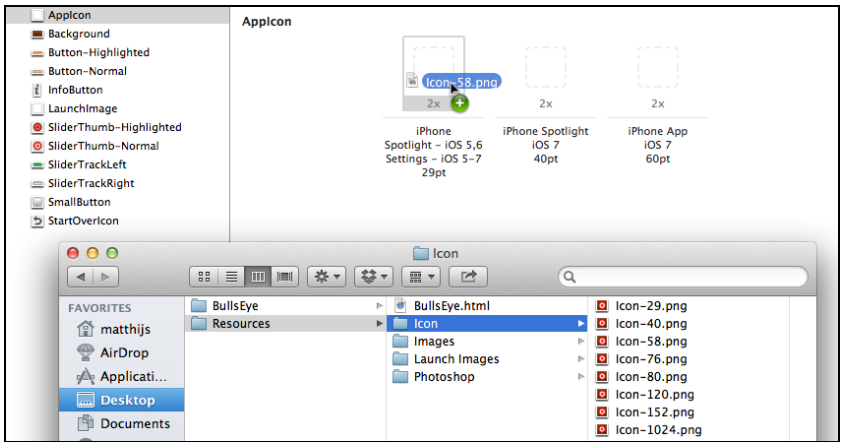
▸ Open the asset catalog (**Images.xcassets**) and select **AppIcon**:



The AppIcon group in the asset catalog

This currently has three items for the different types of icons the app needs.

▸ In Finder, open the **Icon** folder from this tutorial's resources. Drag the **Icon-58.png** file into the first slot, **iPhone Spotlight & Settings 29pt**:



Dragging the icon into the asset catalog

You may be wondering why you're dragging the Icon-58.png file and not Icon-29.png into the slot for 29pt. Remember that this slot says 2x, which means it is for Retina devices and on Retina screens one point counts as two pixels.

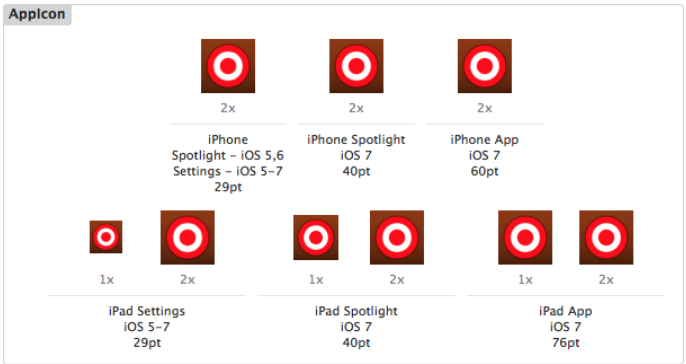
› Drag the **Icon-80.png** file into the middle slot, **iPhone Spotlight 40pt**.

› Drag the **Icon-120.png** file into the slot for **iPhone App 60pt**.

That's only three icons. So what are the other icon files for? They 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** also check the item **iPad iOS 7.0 and Later Sizes**. This adds six new slots to the AppIcon set.

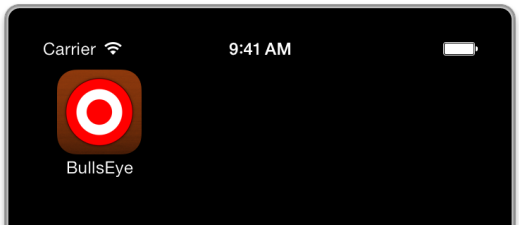
› Drag the icons into the proper slots. Notice that the iPad icons need to be supplied in 1x as well as 2x sizes. You may need to do some mental arithmetic here to figure out which icon goes into which slot!



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

Launch image

Starting up an app usually takes a short while, so an app can specify a placeholder image that is shown while the app is being loaded. Without this launch image, the iPhone will show a black screen instead.

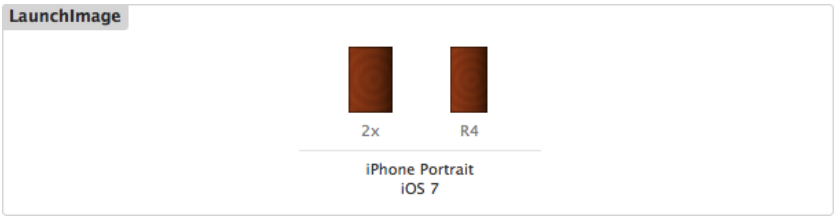
That's exactly what happens when you launch this app. The black screen appears only for a second or two, but you can make the transition between tapping the app icon and actually using the app more seamless by using a launch image.

A lot of developers abuse this feature to show a splash screen with a logo, but it is 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 image will give the illusion the app is loading faster than it actually is.

For Bull's Eye you will just show the wood texture background. Adding these launch images is really easy.

➤ Open the **asset catalog** and go to the **LaunchImage** set. This contains two slots: iPhone Portrait 2x and R4. Even though your app is in landscape, you still need to drop your launch images here.

➤ From the **Launch Images** folder in this tutorial's resources, drag the **Default@2x.png** image into the 2x slot and the **Default-568h@2x.png** image into the R4 slot.



The launch images in the asset catalog

These two images are identical to the background image but turned sideways (in order to compensate for the landscape orientation that the app runs in).

Run the app and notice that the transition into the app looks a lot smoother. It's little details like these that count.

Display name

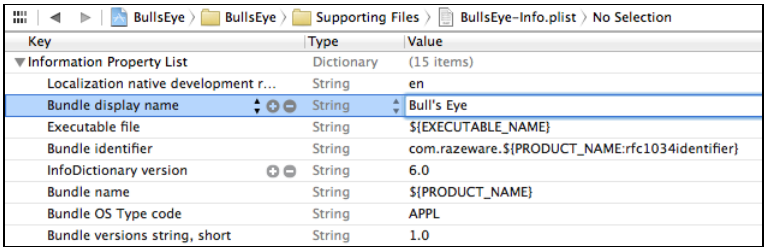
One last thing. You named the project **BullsEye** and that is the name that shows up under the icon. However, I'd prefer to spell it "**Bull's Eye**". There is only limited space under the icon and for apps with longer names you have to get creative to make the name fit. For this game, however, there is enough room to add the space and the apostrophe.

This is a setting that you need to make in the app's Info.plist file.

› From the **Project navigator**, under **Supporting Files**, select **BullsEye-Info.plist**.

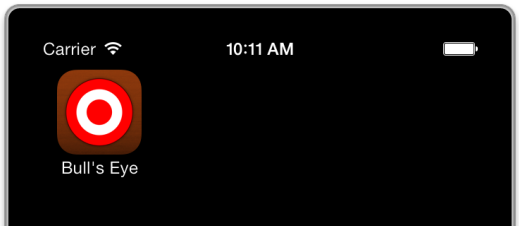
There is a row near the top of the list named **Bundle display name** that currently has the special value "${PRODUCT_NAME}". This means Xcode will automatically put the project name, BullsEye, in this field when it adds the Info.plist to the application bundle. You can simply replace this with the name you want.

› Double-click to edit the **Bundle display name** field and type **Bull's Eye**.



Changing the display name of the app

› 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

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. Developing your apps on the Simulator works fine, but eventually you'll also want to run your creations on your own iPhone. Not only so you can test them properly but also because you will want to show the fruits of your labor to other people. There's hardly a thing more exciting than running an app that *you* made on your own phone.

> Get with the program
>
> **Note:** You cannot run apps on your iPhone unless you have a paid iOS Developer Program account. Without this account, your apps will never leave the Simulator. While it is possible to do a lot of development work on the Simulator, some things it 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! You also need to be a member of the Developer Program if you want to put your apps on the iTunes App Store.

In order to allow Xcode to put an app on your iPhone, the app must be *digitally signed* with your **Development Certificate**. Apps that you want to submit to the App Store must be signed with another certificate, the **Distribution 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. These certificates are part of your Developer Program account.

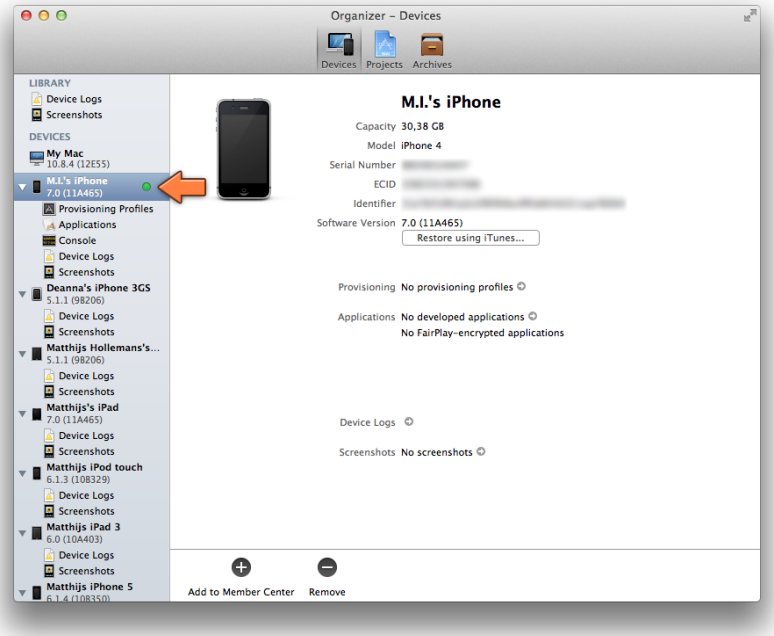
In addition to a valid certificate, you will also need a so-called **provisioning profile** for each app you make. Xcode uses this profile to sign the app for use on your device. The specifics don't really matter, just know that you need a provisioning profile or the app won't go on your device.

There is a limit to how many devices you can use with your Developer Program account. Currently you can register up to 100 devices. This may sound like plenty – you probably don't have a hundred iPhones lying around – but this includes devices of beta testers and anyone else you want to distribute your app to outside of the App Store.

Making the certificates and provisioning profiles used to be frustrating and error-prone. Fortunately, those days are over: Xcode 5 makes it really easy.

➤ Connect your iPhone to your Mac using the USB cable. Then from the Xcode menu bar select **Window** ➤ **Organizer** to open Xcode's Organizer window.

Mine looks like this:



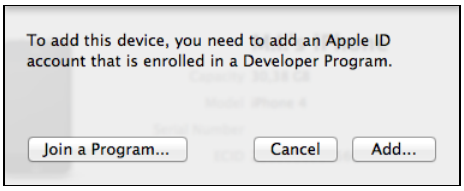
The Xcode organizer

On the left is a list of devices that can be used for development. I have several devices listed. The one with the green light next to its name is currently connected to my Mac.

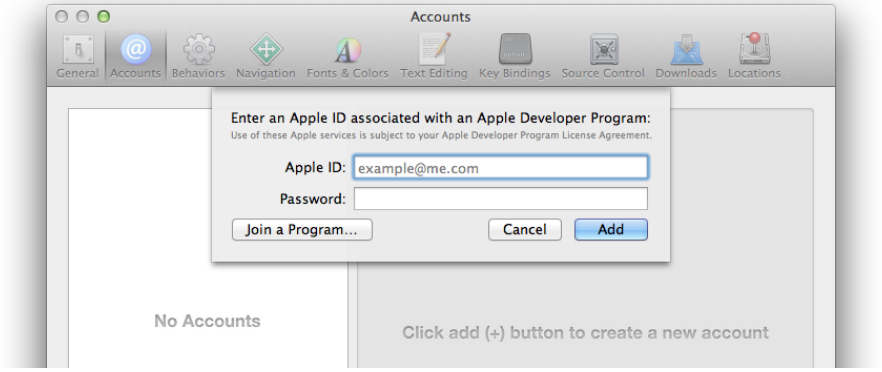
➤ Click on your device name to select it. If this is the first time you're using the device with Xcode, the light is gray and the Organizer window presents a button labeled **Use for Development**. You need to click this otherwise you won't be able to use your iPhone from within Xcode.

(If this is not the first time you're using the device and the light is already green, then you may need to click the **Add to Member Center** button to register the device if you haven't done so already.)

After you press **Use for Development**, Xcode will ask for your Developer Program account:



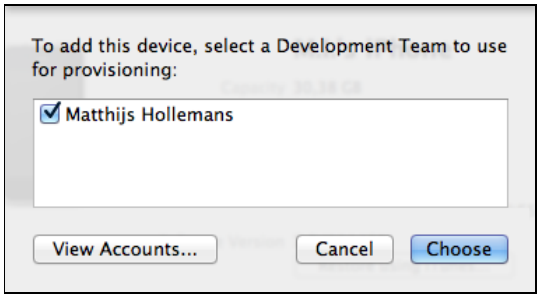
➤ Click **Add...** This opens the **Accounts** pane in the Xcode Preferences window, asking for an Apple ID:



Adding a new Developer Program account to Xcode

› Type the username and password for your Developer Program. This is the same login that you use for the iOS Dev Center website.

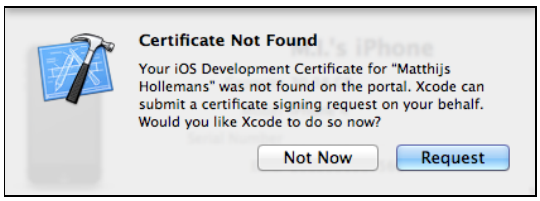
When you're done, close the Accounts window and return to the **Organizer**. It now asks you to choose the development team, which is the account you just added:



Choosing the development team

› Select the name of your account and click **Choose**. Xcode will now contact the Apple servers to fetch the certificates and provisioning profiles it needs.

If this is the first time you're setting this up, you'll get the following message because you do not have a Development Certificate yet:



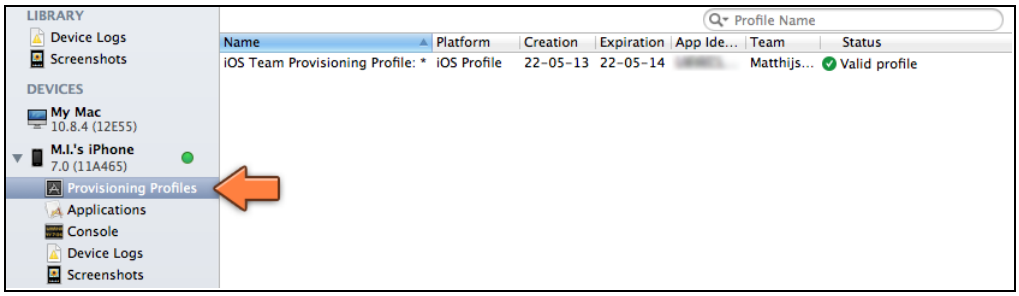
Xcode could not find a Development Certificate

› Click **Request** and wait a few seconds until the light next to your device name turns green. Great! That wasn't so hard, was it? ☺

While you're waiting, Xcode automatically registers your device with your Developer Program account, creates a new Development Certificate, and downloads and installs the so-called Team Provisioning Profile on your device. I suppose they call it

the "team" profile because it will work on all the devices that you have registered with the Developer Program.

You can see the profiles that are installed on your device by clicking the **Provisioning Profiles** item under the device name:

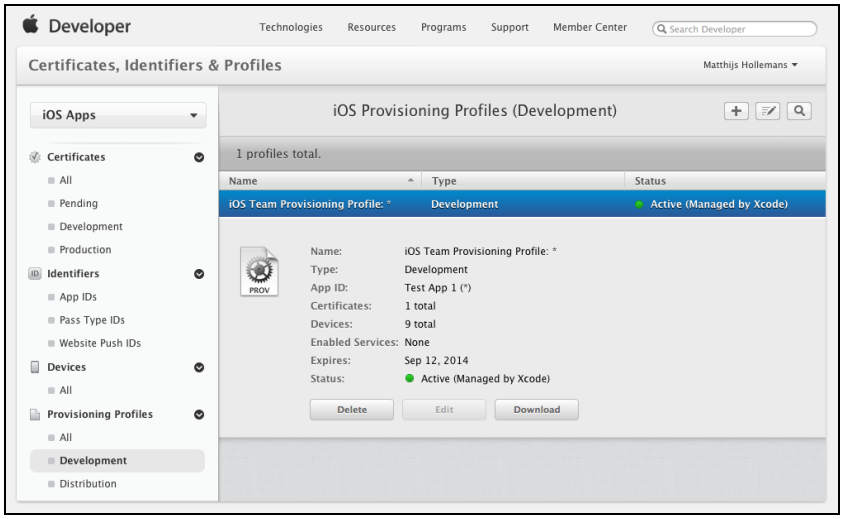


The provisioning profiles on your device

The Team Provisioning Profile has a so-called *wildcard App ID*, which means you can use it for any application you are developing (as long as they don't require any special features such as push notifications). This means you won't have to repeat this procedure for any of the other apps you will be developing in this series. Xcode knows about the profile now and it will automatically use this profile to sign your apps.

You can also login to the iOS Dev Center website to see what the provisioning profile looks like there. Needless to say, you only have access to this portal if you're in the paid iOS Developer Program.

› Go to http://developer.apple.com/devcenter/ios and log in. Under **iOS Developer Program**, choose **Certificates, Identifiers & Profiles**.



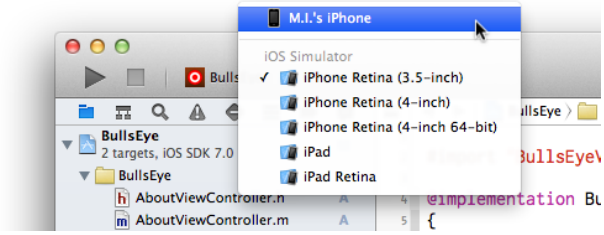
The provisioning website

It's easiest to simply let Xcode handle all the certificate and provisioning profile stuff but you can also manage these things yourself from the iOS Dev Center. You

can download the handy **App Distribution Guide** that explains how all of this works in detail.

This concludes the setup. You have added your Developer Program account to Xcode, obtained a Development Certificate, registered your device, and installed a provisioning profile. You're ready to run the app on your phone!

➤ Go back to Xcode's main window and click on the scheme selector in the toolbar to change where you will run the app. It currently says "iPhone Retina (4-inch)" but 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

➤ Press **Run** to launch the app.

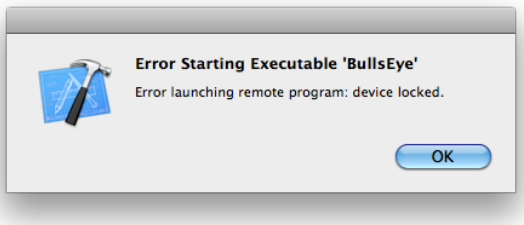
At this point you may get a popup with the question "codesign wants to sign using key … in your keychain". Answer with **Always Allow**. This is Xcode trying to use the new Development Certificate you just created but you need to give it permission first.

Does the app work? Awesome! If not, read on...

There are a few things that can go wrong when you try to run the app on your device, especially if you've never done this before, so don't panic if you run into problems.

Make sure your iPhone is connected to your Mac. There must be a green light next to the device's name in the Organizer window.

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

Or you might get a message in the Xcode Debug output pane:

```
error: failed to launch 'BullsEye' -- device locked
```

Simply unlock your phone (type in the 4-digit passcode) and press Run again.

Code Sign error: a valid provisioning profile matching the application's Identifier 'com.yourname.BullsEye' could not be found. Xcode does not have a valid provisioning profile to sign the app with. The installation of the Team Provisioning Profile has apparently failed.

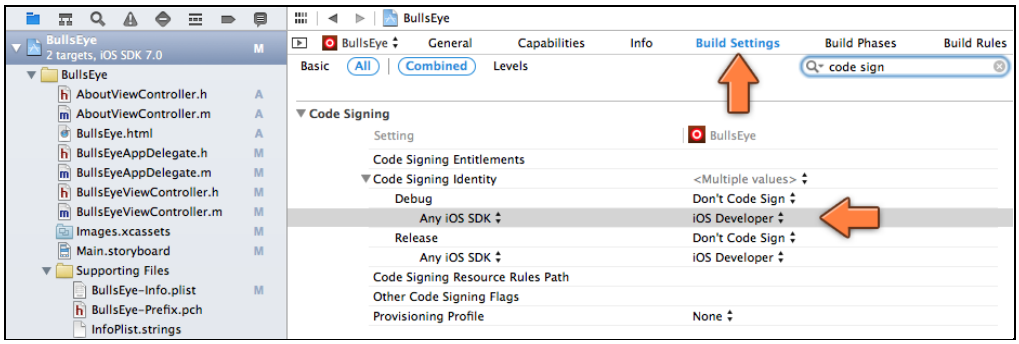
No valid provisioning profile on the device. This really shouldn't happen because Xcode will automatically install the profile onto the device before it runs the app, but you never know.

Xcode could not find a valid private-key/certificate pair for this profile in your keychain. Your development certificate isn't properly installed. This can happen when you moved your certificates to a new computer but forgot to move the corresponding private keys as well.

The solution to the above three problems is to restart Xcode and try again. That's always good advice when Xcode does not do what you want. If that doesn't help, open the **Preferences** window and go to the **Accounts** tab. Select your account and click the **View Details** button. In the dialog that appears, click the refresh button (the arrow in the bottom-left corner). Xcode will now fetch your data from the iOS Dev Center again.

Alternatively, open the Dev Center website in your browser (tip: it works best on Safari) and download the Provisioning Profile to your computer, then drag it into the Organizer window onto the device name in the left column. Click on the Provisioning Profiles item under the name of the device and verify that the profile is installed on that device.

If you want to know how Xcode chooses which profile to sign your app with, then click on your project name and switch to the **Build Settings** tab. There are a lot of settings in this list, so filter them by typing **code sign** in the search box. The screen will look something like this:



The Code Signing settings

Under **Code Signing Identity** it says **Debug, Any iOS SDK: iPhone 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.

Xcode is actually pretty smart about automatically picking the right provisioning profile for you, but now at least you know where to look.

The end... or the beginning?

This has been a very long lesson and 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 Objective-C code. It is perfectly fine if you don't, as long as you're enjoying yourself and you sort of get the hang of 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 will find the complete source code for the Bull's Eye app. I removed the cruft from Xcode's standard template and added plenty of comments. If you're still unclear about some of what you did, it might be a good idea to look at this cleaned up, commented source code.

If you're interested in how I made the graphics, then take a peek at the **Bullseye.psd** and **Icon.psd** Photoshop files in the Resources folder. The wood background texture comes from http://subtlepatterns.com and was made by Atle Mo.

But there's more!

Thank you for reading the first tutorial of my series, *The iOS Apprentice: iPhone and iPad Programming for Beginners*. I hope this first tutorial gave you some taste of what is to come in the rest of the series, which is available from www.raywenderlich.com.

The full series has several 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.

Currently available are:

- **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 Objective-C 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 Objective-C 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, Objective-C 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 tutorials from *The iOS Apprentice* series at http://www.raywenderlich.com/store/ios-apprentice.

They're worth it if you want to become a great iOS developer!

About the author

Matthijs Hollemans is an independent designer and developer who loves to create awesome software for the iPad and iPhone. He also enjoys teaching others to do the same, which is why he wrote *The iOS Apprentice series* of eBooks and a whole stack of tutorials for raywenderlich.com.

Matthijs lives in the Netherlands with his partner Deanna and their very naughty cat, Sophie. In his spare time, Matthijs is learning to play jazz piano (it's hard!) and likes to go barefoot running when the sun is out. Check out his blog at www.hollance.com.

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 http://www.raywenderlich.com/forums/.

Thanks for reading!

Revision history

v1.5 (Sept 2013) – Second edition. Completely updated for iOS 7 and Xcode 5.

v1.4 (Aug 2012) – Updated for iOS 6. Removed explanation of `viewDidUnload`, as its use is no longer recommended by Apple.

v1.3 (June 2012) – Updated for Xcode 4.3. Added PDF version.

v1.2 (Dec 2011) – Added tutorial 4, StoreSearch

v1.1 (Sept 2011) – Updated for iOS 5

v1.0 (July 2011) – First version (iOS 4)