The iOS Apprentice 1

Getting Started

By Matthijs Hollemans

Version 1.4

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

You're about to read the first epic-length tutorial from my ebook 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 ebook 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.

Some of the apps we'll be making in The iOS Apprentice series

Introduction

Everyone likes games, so the first app we'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 we'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 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 ebook 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 for yourself how the other parts of the SDK work and teach yourself the rest.

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 20 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. 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.

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). We 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 — 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 we'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 we 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 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 you can do without the training wheels.

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

iOS 5 and better only

Things move fast in the world of mobile computing. The iPhone 3G and version 3 of iPhone OS are only a few years old but both are quickly becoming obsolete. Even iOS version 4, which was released just in 2010, is starting to look dated now that iOS 5 has become available. And by the time you read this, you may already have your hands on iOS 6.

The tutorials in this ebook series are aimed exclusively at iOS version 5.0 and later. The reason for this is the introduction of Automatic Reference Counting (ARC), a new feature in iOS 5 that largely changes the way you write apps. Before ARC, you had to be really careful to follow a certain set of rules for performing "manual memory management". The slightest mistake would cause your app to mysteriously crash or at some point run out of available memory.

While it's still possible to make your app crash in mysterious ways, ARC automates all of the memory management tasks for you. This immensely simplifies the development process and makes apps much more robust. But it also requires us to do away with the old way of working. It just makes no sense to teach you programming techniques that were only needed on previous versions of the system now that we have something much better.

Because we're currently in a transitioning period from manual memory management to ARC, there may be times when you'll need deal with old code that does not use Automatic Reference Counting. There are many older books and blog posts about iOS development that still use the old techniques. You may also want to use a third-party library that hasn't yet been updated to use ARC. I will explain how to make such old source code work on iOS 5 and up, but most of the time we'll be looking forward, not backward.

The obvious downside is that the techniques you will learn here won't work on devices that still run iOS 4 or earlier. Unless you want to do manual memory management, your apps will work on iOS 5 or better only. Will that be a problem? I don't think so.

The majority of iPhone, iPod touch and iPad users have already upgraded to iOS 5 and any stragglers will soon follow. Only users with older devices, such as the iPhone 3G and second generation iPod touch, may be stuck with iPhone OS 3 (or even OS 2) 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. Some developers have already switched to iOS 6 exclusively but in my opinion it's a little too early to leave customers with older devices that won't run iOS 6 behind, such as the first iPad. That's why the tutorials from the iOS Apprentice series still support iOS 5, but don't let that stop you from taking advantage of iOS 6 in your own apps!

What you need

It's a lot of fun to develop for the iPhone and iPad but like most hobbies 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 we 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. It needs to run at least OS X version 10.6 (Snow Leopard) or version 10.7 (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 buy a machine that has an older version of OS X (10.5 Leopard or earlier), then you can purchase an inexpensive upgrade to Snow Leopard and Lion from the Apple Store or a reseller.

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 will 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.

You will have to renew your membership every year but if you're serious about developing apps then that $99 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 but not 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 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 OS X Lion, so if you're still running Snow Leopard you'll first have to upgrade to Lion (also available from the Mac App Store). Alternatively, you can download Xcode for Snow Leopard from the iOS Dev Center website [http://developer.apple.com/devcenter/ios/]. Regardless of where you get your Xcode from, 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 originally wrote this ebook with Xcode version 4.2 and the iOS 5.0 SDK. For the latest revision, I used Xcode 4.5 on Mountain Lion with the iOS 6.0 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 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.

Because not everyone who is reading this will have switched to the latest-and-greatest already, the tutorials from the iOS Apprentice will continue to work on Xcode 4.2 and iOS 5, and I will point out major differences between the versions where necessary.

For a long time Xcode 3 was the official development environment for iOS apps but it has now been superseded by Xcode 4. Some people still use Xcode 3 as version 4 is a radical departure from the old way of working. However, Xcode 4 is the future so that's what we will be using. I mention this because many older books and blog posts talk about Xcode 3. 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 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, we don't have to write any ones and zeros ourselves. Everyday English is not precise enough to use for programming computers, so we will use use an intermediary language, Objective-C, that is a little bit like English so it's reasonably easy 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:

```
00011001010011110100100011001111100101 0
00101000100111101011011100111010110100 1
01010001110011111101011101100001110001 10
10010000011100010100110100111100110011 1
```

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 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. (The core of iOS is largely based on Unix.)

The language we use to program iOS apps is called Objective-C and it is basically 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.

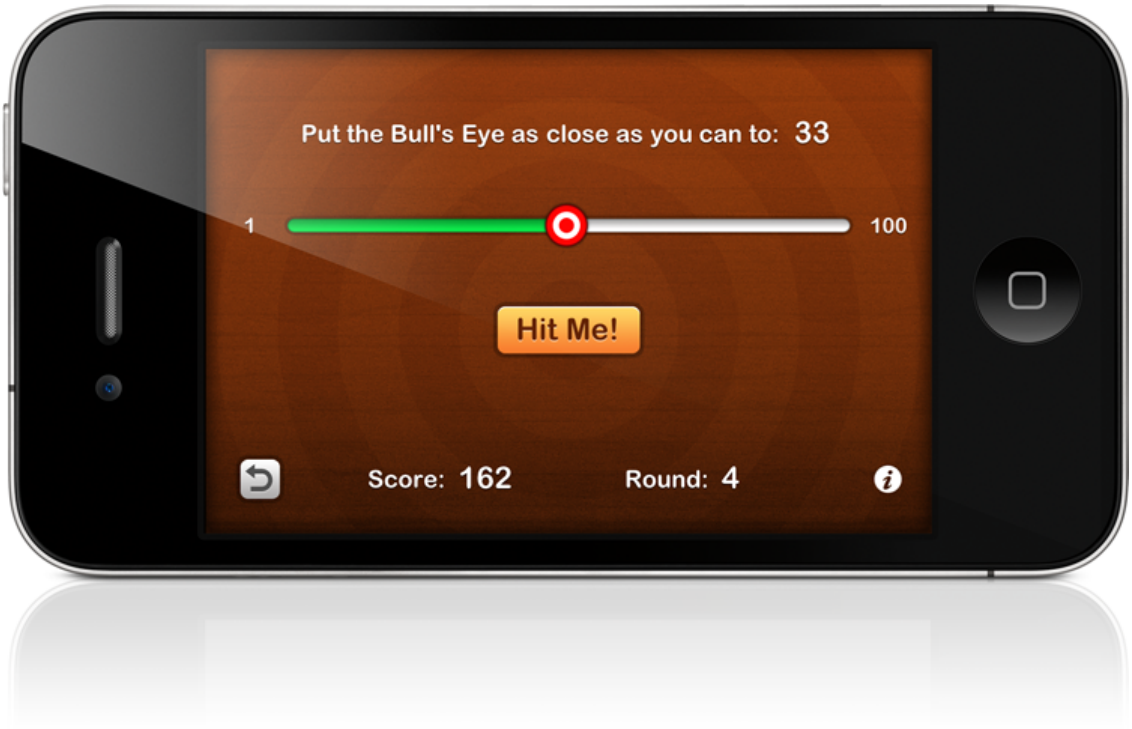
I could have started this ebook series with a in-depth treatise on Objective-C but you'd probably fall asleep halfway. So instead I will explain the language as we go along, very briefly at first but more in-depth later. In the beginning, the general concepts — what is a variable, what is an object, 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 we're going to create a game called Bull's Eye. This is what the game will look like when we're finished:

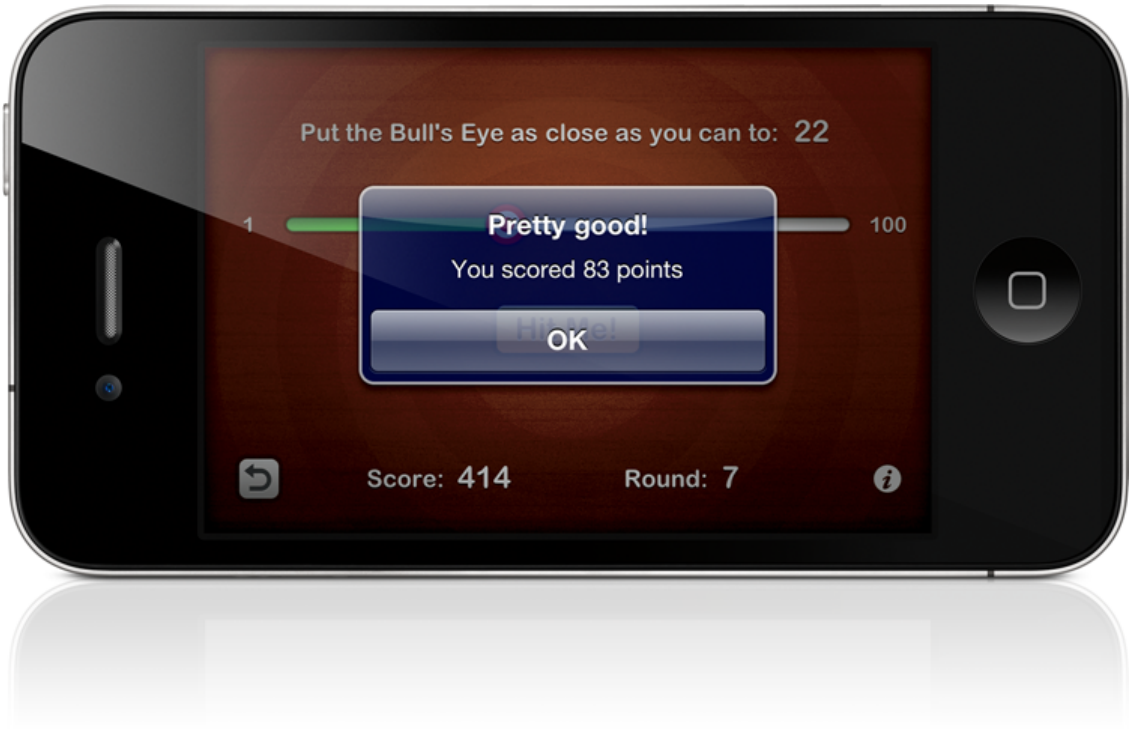
The finished Bull's Eye game



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 33. 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 we 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 he did. Somehow we 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, such as 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. We 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 we 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 we are going to do for now. Once we have this working, we 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, we'll be working with Xcode 4.2 or better. Newer versions of Xcode will also work but anything older than version 4.2 is a no-go. So please update to the latest and greatest if you haven't already.

There are some pretty big differences between iOS 5 and its predecessors as far as programming is concerned, so rather than teaching you how to code for iOS the *old way*, I'm going to focus on iOS 5 and better only. Don't worry, you'll still be able to write apps that also work on older versions of iOS, but to keep things simple in the first couple of tutorials we'll focus on iOS 5+ exclusively.

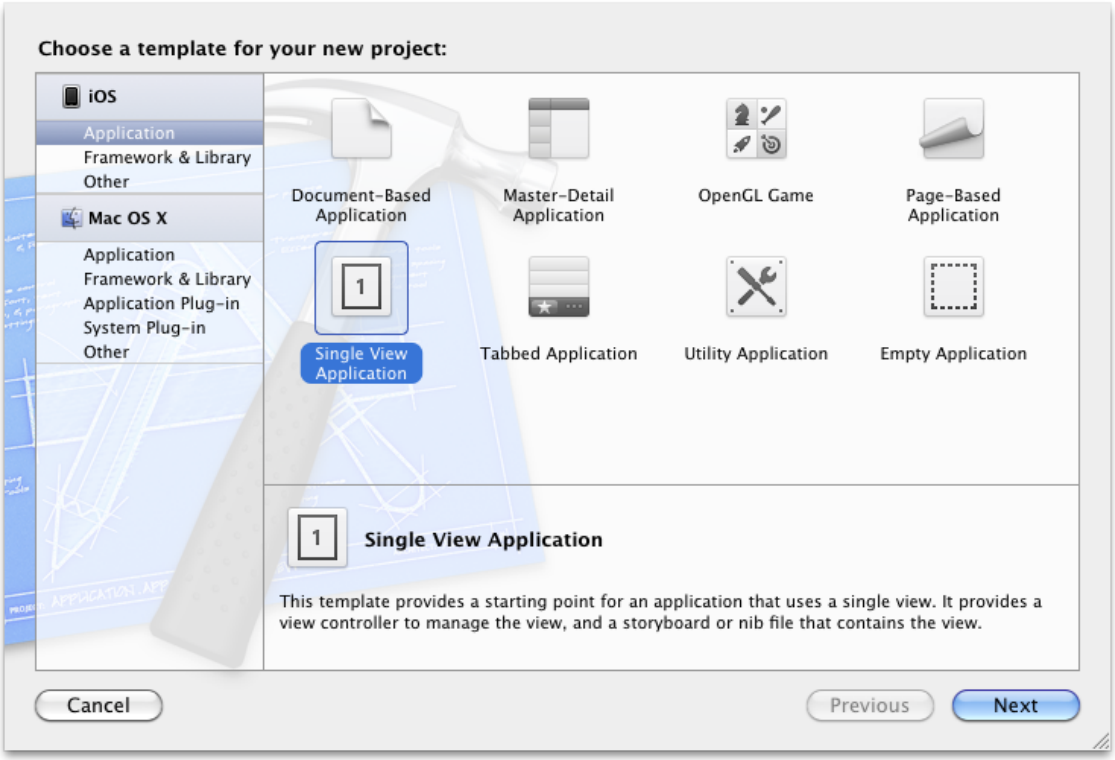
» Launch Xcode. If you have trouble locating the Xcode application, you can find it in the folder /Developer/Applications/Xcode. If you installed Xcode from the Mac App Store, it will be in your Launchpad. Because I use Xcode all the time, I placed its icon in my dock for easy access.

Xcode shows the “Welcome to Xcode” window when it starts:

Xcode bids you welcome

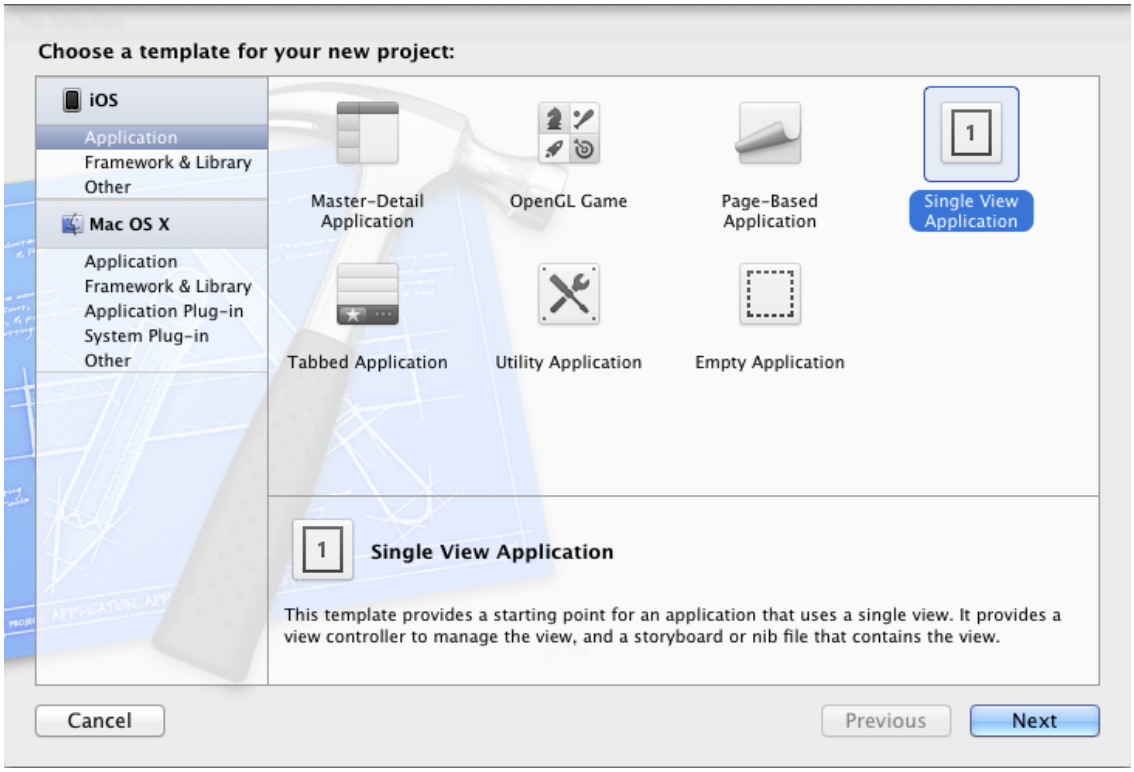
Choose “Create a new Xcode project”. The main Xcode window appears with an assistant that lets you choose a template:

Choosing the template for the new project



On Xcode 4.3 and better, the window looks slightly different:

Choosing the template for the new project in Xcode 4.3

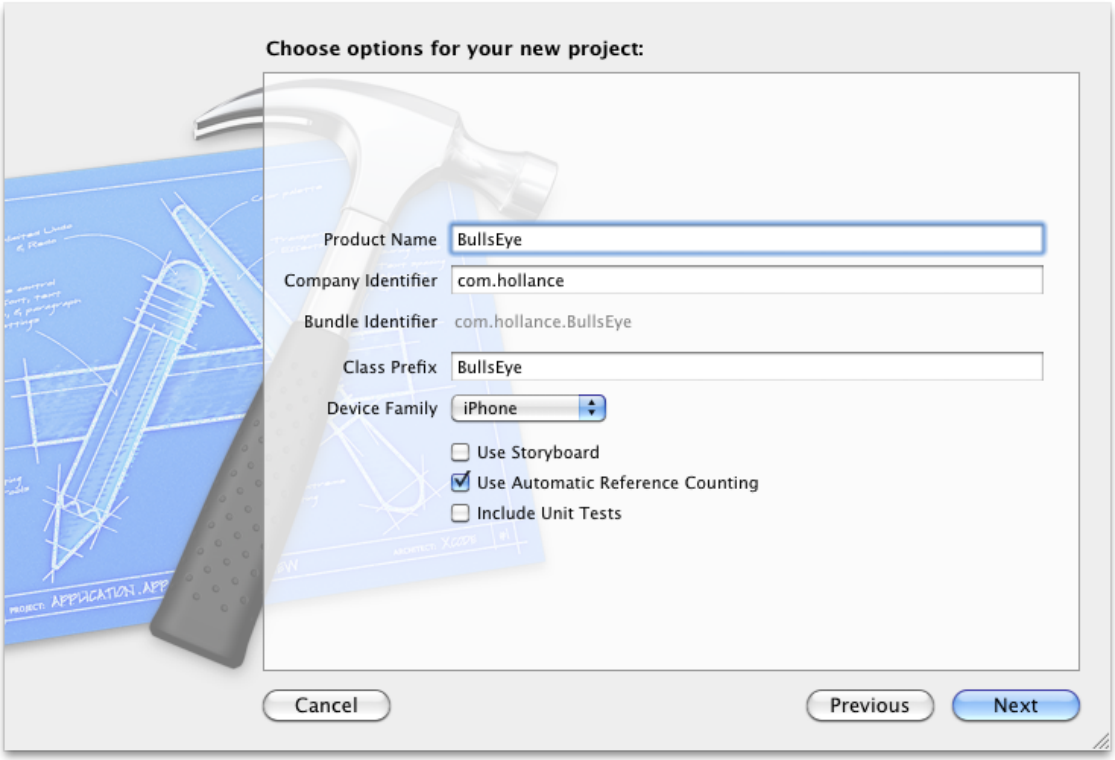


There are templates for a variety of application styles. Xcode will make a pre-configured project for you based on the template you choose. The new project will already include many of the source files you need. These templates are handy because they can save you a lot of typing. They are ready-made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

Note: One of the differences between Xcode 4.2 and 4.3 is that these templates have changed. In this tutorial I'll include instructions for both sets of templates, so you can use either version of Xcode. There's a good chance that the templates will change again with an upcoming version of Xcode.

» Select the "Single View Application" and press Next.

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

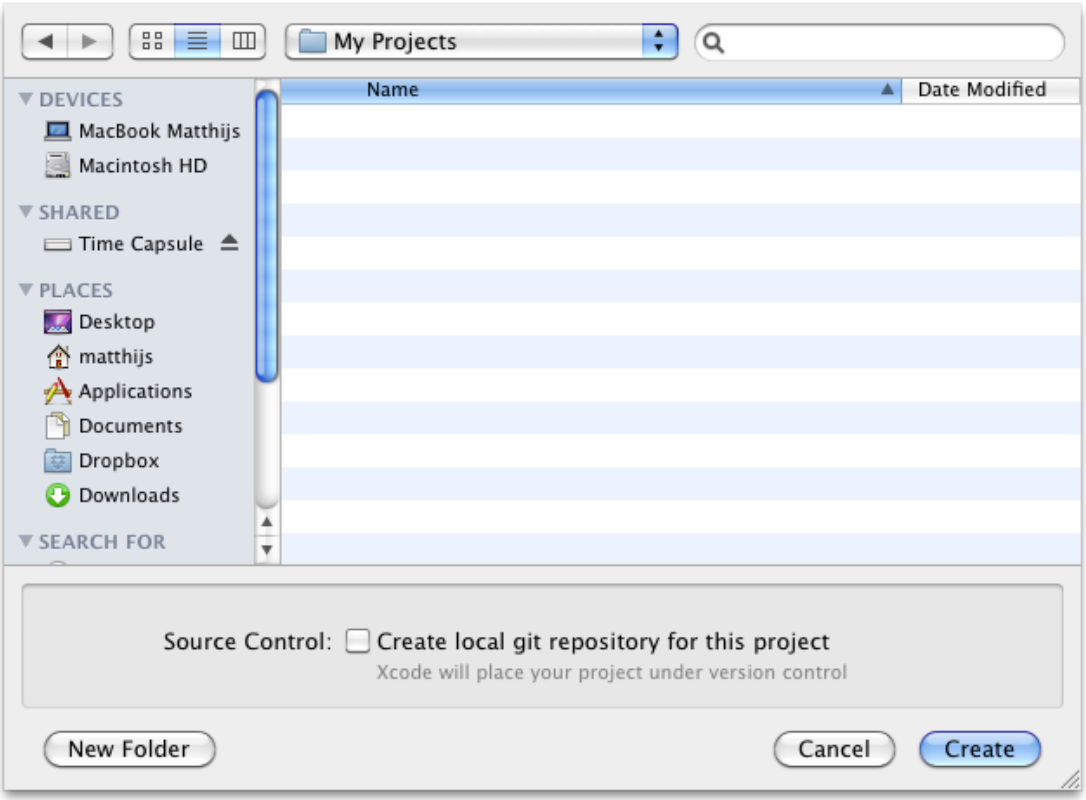


» Fill out these options as follows:

- Product Name: BullsEye. If you want to use proper English, you can name the project Bull's Eye instead of BullsEye, but I like to avoid spaces and other special characters in my project names.
- Company Identifier: Mine says "com.hollance". That is the identifier I use for my apps and as is customary, it is my domain name written the other way around. If you already have registered for the paid iOS Developer Program, Xcode will fill this in for you. If not, choose 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
- Device Family: iPhone
- Use Storyboard: Uncheck this box. We will not be using Storyboards in this lesson.
- Use Automatic Reference Counting: Keep this box checked.
- Include Unit Tests: Leave this box unchecked.

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.

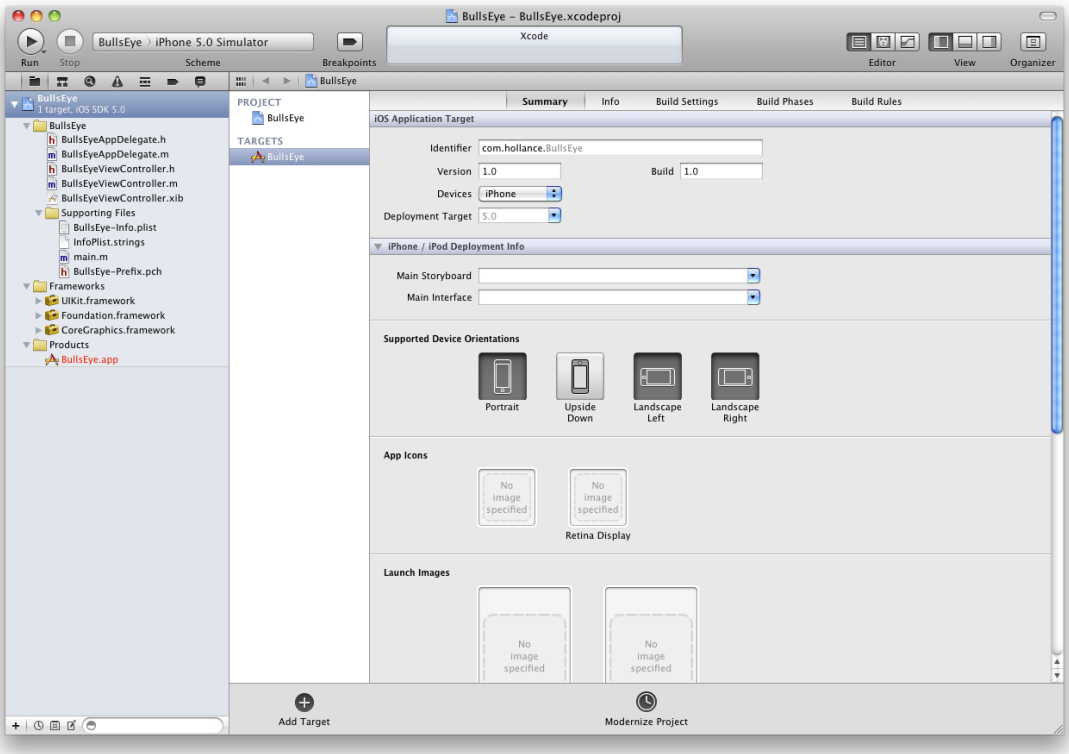
Xcode will automatically make a new folder for the project using the Product Name that you entered in the previous step (in our case BullsEye), so you don't need to make a new folder yourself.

At the bottom of the window there is a checkbox that says "Create local git repository for this project". You can ignore this for now. We'll talk about the Git version control system and how to use it from Xcode in a later tutorial.

» Press Create.

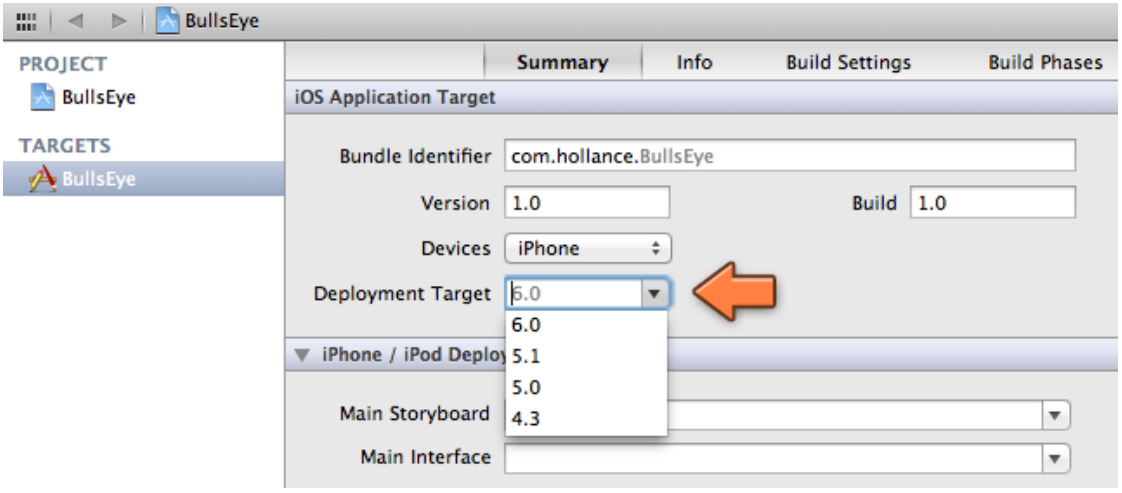
Xcode will now create a new project named BullsEye, based on the Single View Application template, in the folder you specified. When it is done, the screen looks like this:

The main Xcode window at the start of our project



If you're using Xcode 4.5 and the iOS 6 SDK, then the Deployment Target field says "6.0" (or a higher number). Because we want this app to run on iOS 5 as well, click this box and choose "5.0":

Changing the deployment target



» Press the Run button in the top-left corner:

Press Run to launch the app

Xcode will rattle for a bit and then it launches our 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 our 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 select box labeled Scheme at the top-left of the window says “BullsEye | iPhone 5.0 Simulator” (or a later version). 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 we stick with the Simulator.

Set the app to run on the simulator

Next to the Run button is the Stop button. 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 on iPhone Simulator”:

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, xib 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, the application bundle is copied to the Simulator or the iPhone and the app is launched. All from a single press of the Run button.

Adding the button

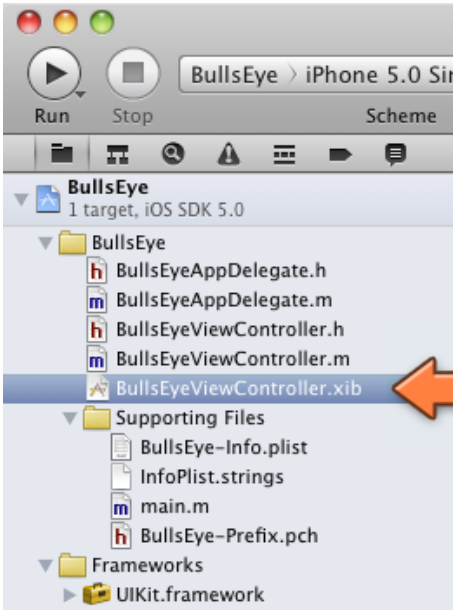
I'm sure you're as little impressed as I am with an app that just displays a dull gray screen, so let's add a button to it.

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 "BullsEyeViewController.xib" 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 compo-

nents such as buttons into the app. (OK, bad analogy, but Interface Builder *is* a super tool in my opinion.)

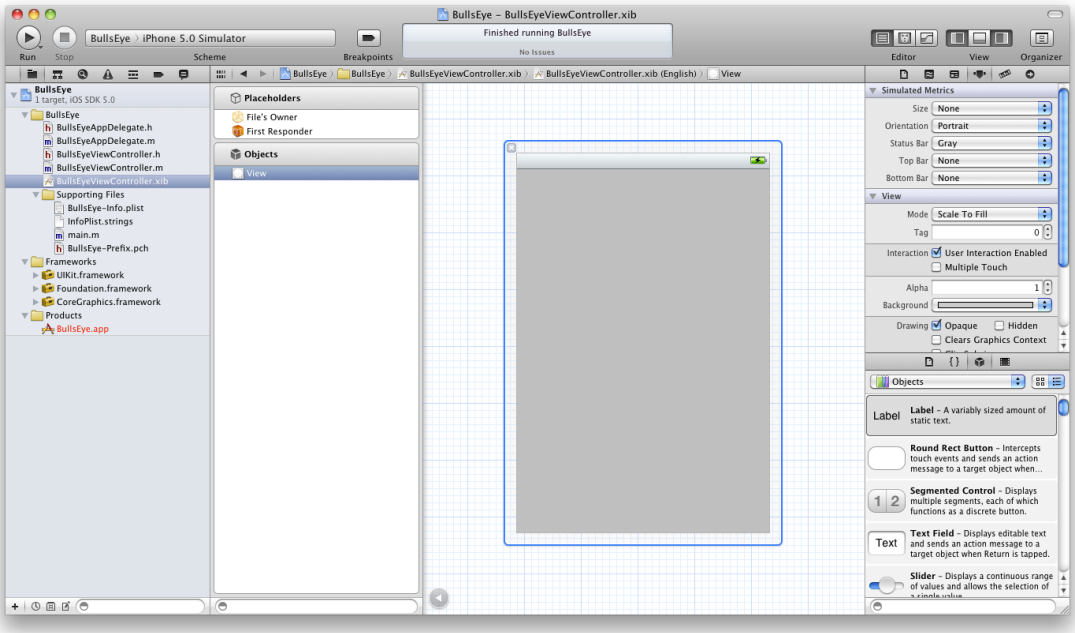
» Click the “Hide or show utilities” button in Xcode’s toolbar:

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.

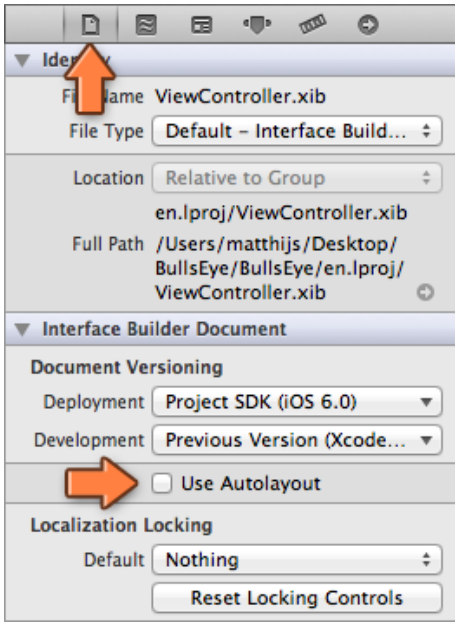
Your Xcode should now look something like this:

Editing BullsEyeViewController.xib



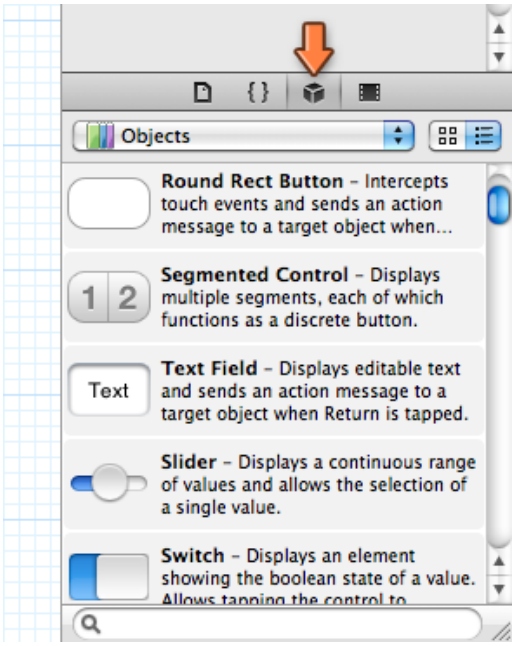
Important for users of Xcode 4.5 or later: iOS 6 introduced a new feature named Auto Layout. This is an advanced tool for making flexible user interface designs. We’re not going to use the Auto Layout feature in this tutorial, so you first need to disable it. You do this from the Utilities pane on the right. From the row of small icons at the top, click the first one to open the so-called File Inspector. Then simply uncheck the “Use Autolayout” option and you’re done:

Disabling Auto Layout (on Xcode 4.5)



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):

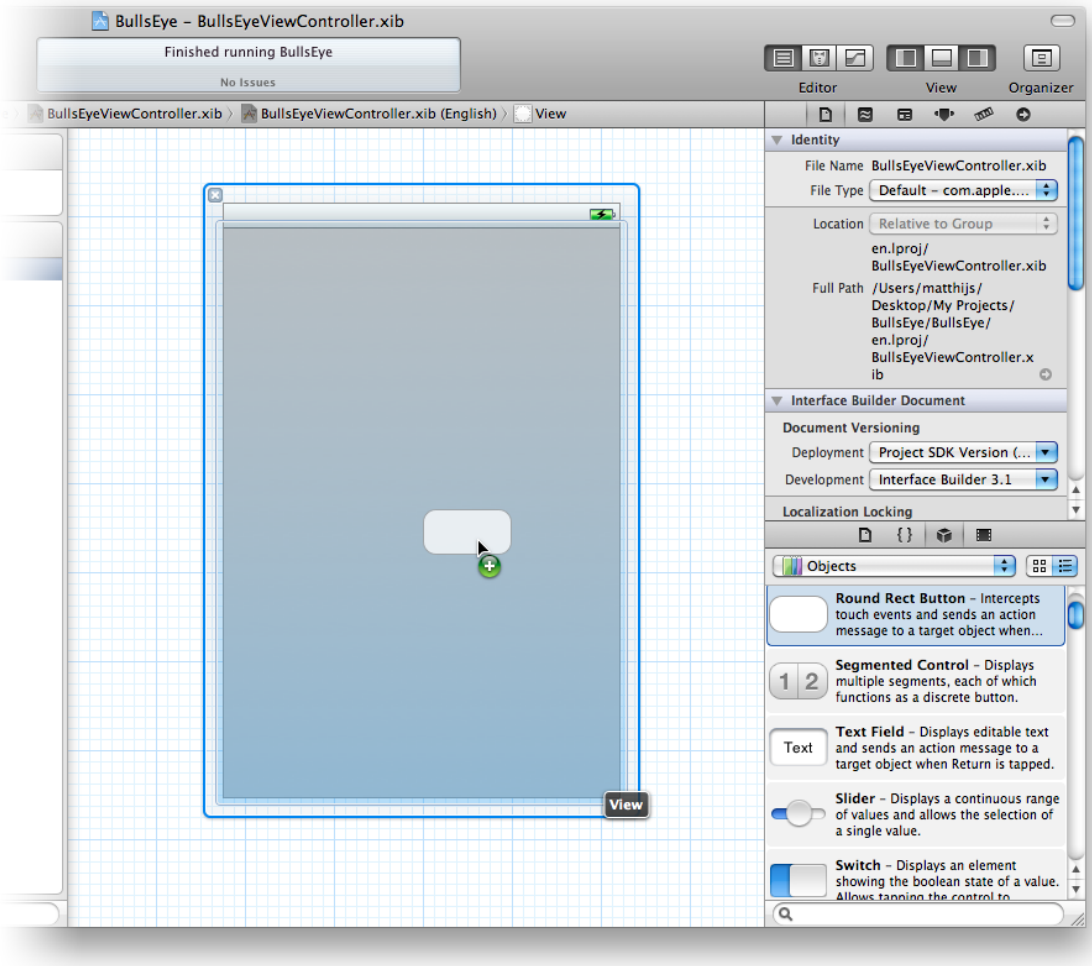
The Object Library



In the screenshot above, the first item in the Object Library's list is "Round Rect Button".

» Click on Round Rect Button and drag it into the working area, on top of the gray view.

Dragging the button on top of the view



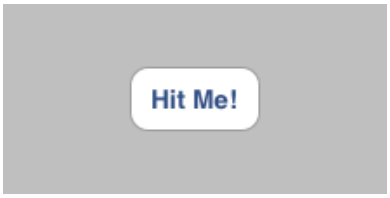
That's how easy it is to add new buttons, just drag & drop. That goes for all other user interface elements too. We'll be doing a lot of this, so take some time to get familiar with the process.

» Drag-and-drop a few other controls, such as labels, sliders, 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 now has a title



When you're done playing with Interface Builder, press Xcode's Run button. The app should now appear in the Simulator, complete with a nice 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 files after you make changes to them because Xcode will automatically save any modified files when you press the Run button. Nevertheless, as I write this, Xcode 4 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 Cmd+S on a regular basis to save my files.

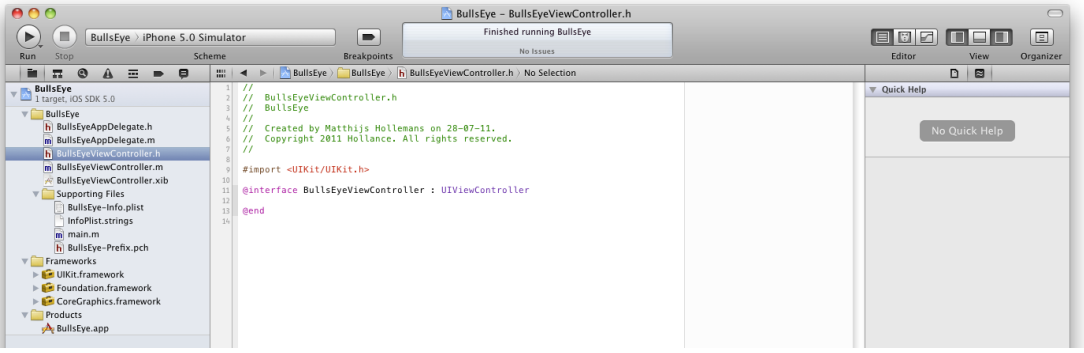
The source code editor

A button that doesn't do anything when it is 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 our 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

```
//
//  BullsEyeViewController.h
//  BullsEye
//
//  Created by <you> on <date>.
//  Copyright <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

We've edited the BullsEyeViewController.xib file to build the user interface of our app. It's only a button on a gray background, but a user interface nonetheless. We also just added source code to BullsEyeViewController.h. You probably guessed that these two files have something to do with one another. There is also a BullsEyeViewController.m.

These three files together — the .xib, .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 a detail view of a single item. So 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 our app has only one screen (the gray one with the button on top) and thus only needs one view controller. That view controller is named `BullsEyeViewController` and the .xib, .h and .m files all work together to implement it.

Simply put, the xib 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 our case `BullsEyeViewController`.

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

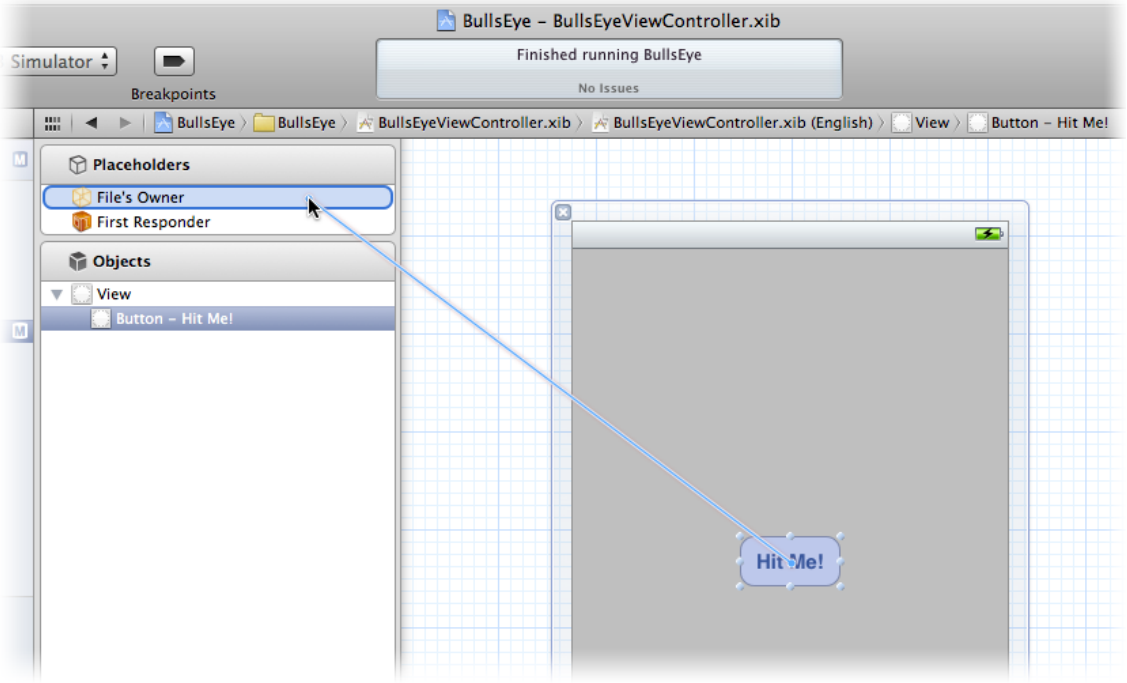
Making connections

The line of source code we 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. We will now connect our button to that action.

» Click BullsEyeViewController.xib to go back into Interface Builder. Click the Hit Me button once to select it.

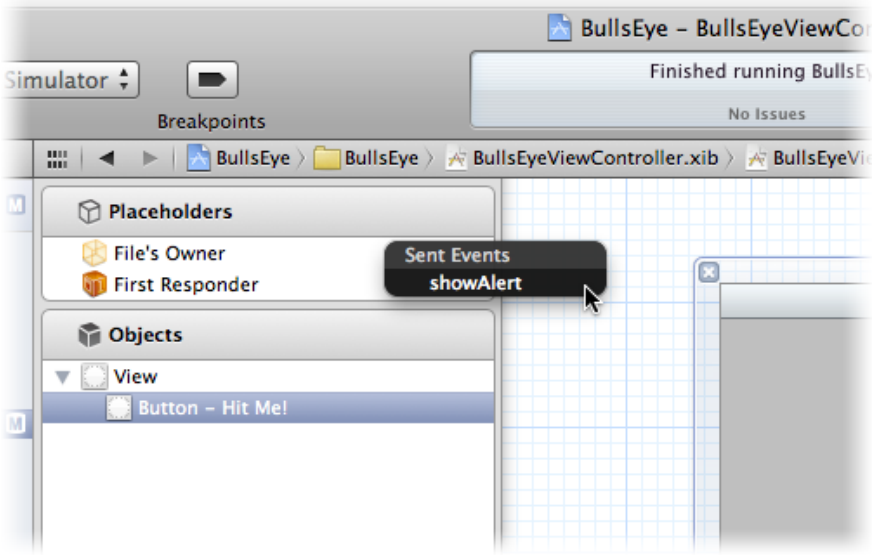
With the Hit Me button selected, hold down the Ctrl key, click on the button and drag up to the File's Owner item in the Placeholders section. You should see a blue line going from the button up to File's Owner. (Instead of holding down Ctrl, you can also right-click and drag, but don't let go of the mouse button before you start dragging.)

Ctrl-drag from the button to File's Owner



Once you're on File's Owner, let go of the mouse button and a small menu will appear. It is titled "Sent Events" and contains one option, `showAlert`. This is the name of the action that we added earlier in the BullsEyeViewController.h file.

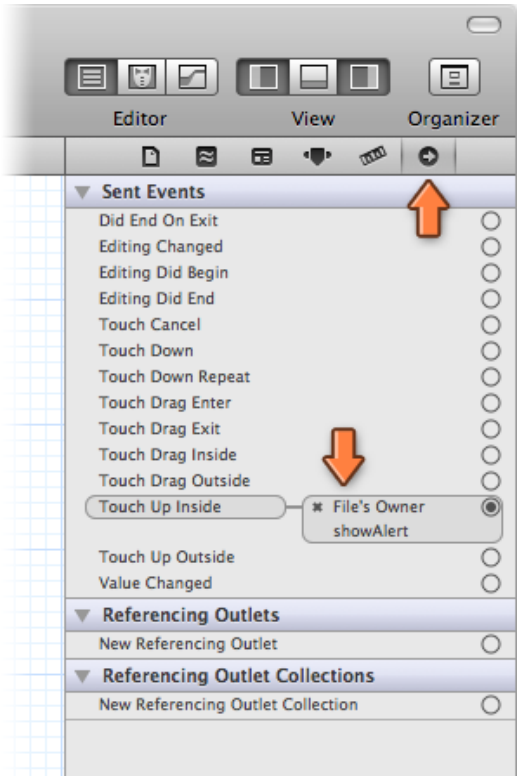
The Sent Events menu from File's Owner



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 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 Connections 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 X icon (but don't do that right now!). You can also drag directly from any of these events to File's Owner. Click on an open circle and the blue line will appear.

File's Owner

A xib, the user interface of a view controller, is *owned* by that view controller. This means the view controller will load that xib file, put all of its objects on the screen, and make the connections. The view controller will dispose of the xib file when the screen closes. File's Owner, therefore, refers to the view controller object.

In our app we have a view controller named `BullsEyeViewController` that owns the xib file of the same name. The `showAlert` action also belongs to `BullsEyeViewController`. Because actions are logic, not user interface, they are not defined in the xib file. In order for the button to hook up with `showAlert`, it must have a way to refer to the view controller's actions.

File's Owner is a convenient way to accomplish that. By dragging from the button to File's Owner, you tell Interface Builder that you want to connect the button to an action from the view controller.

If you're ever wondering who the File's Owner is for a xib, then select File's Owner and open the Identity Inspector (the third button at the top of the inspector area). The Custom Class field will show you the owner's name. If you do that for our nib, you'll see that it says `BullsEyeViewController`.

Acting on the button

We now have a screen with a button and we have hooked it up to an action `showAlert` that will be performed when the user taps on the button. However, we haven't yet told the app what the 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 is a whole bunch of source code in BullsEyeViewController.m already but we can ignore that for now.

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

BullsEyeViewController.m

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

  [alertView show];
}
```

When we added the `(IBAction)showAlert;` line to the .h file we only said to Interface Builder, "the action `showAlert` is available," but we did not specify what `showAlert` actually did. This time, however, we 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.

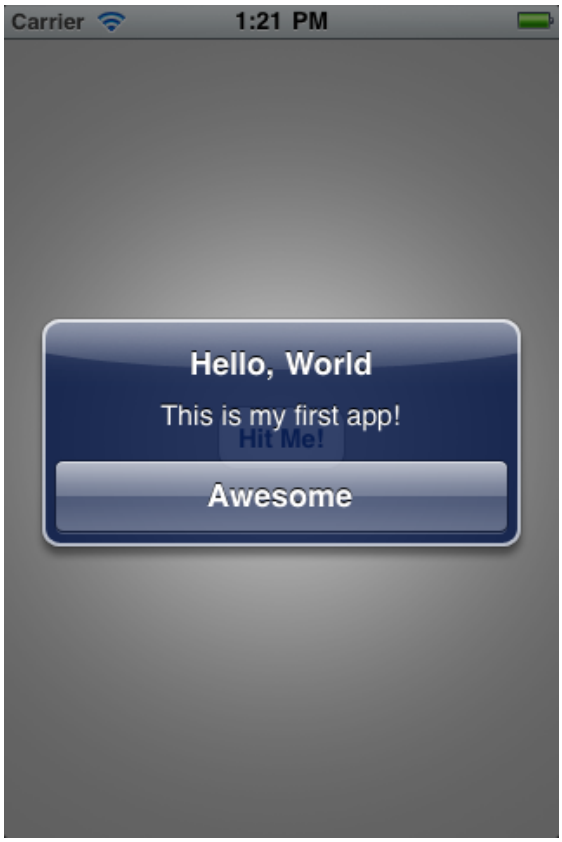
The code in `showAlert` creates an alert view with a title ("Hello, World"), a message ("This is my first app!"), and a single button labeled "Awesome". If you're not sure about the distinction between the title and the message: both show text, but the title is slightly bigger and in a bold typeface.

Note: We're only adding this to the .m file; we'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 we 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



We can strike off the first two items from our to-do list: 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! We're only just getting started…

You can find the project files for the app we've made thus far in the "01 - One Button App" folder inside the Source Code folder that comes with this tutorial.

Problems?

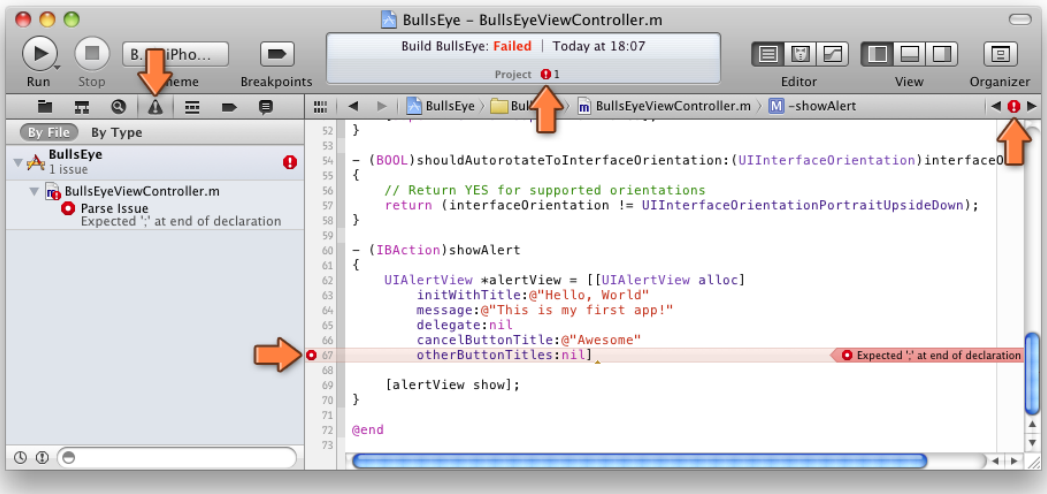
If Xcode gives you a "Build Failed" error message after you press Run, then make sure you typed in everything correctly. Even the smallest mistake will totally confuse Xcode. It can be quite overwhelming to make sense out of the error messages because a small typo at the top of the source code can produce 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.

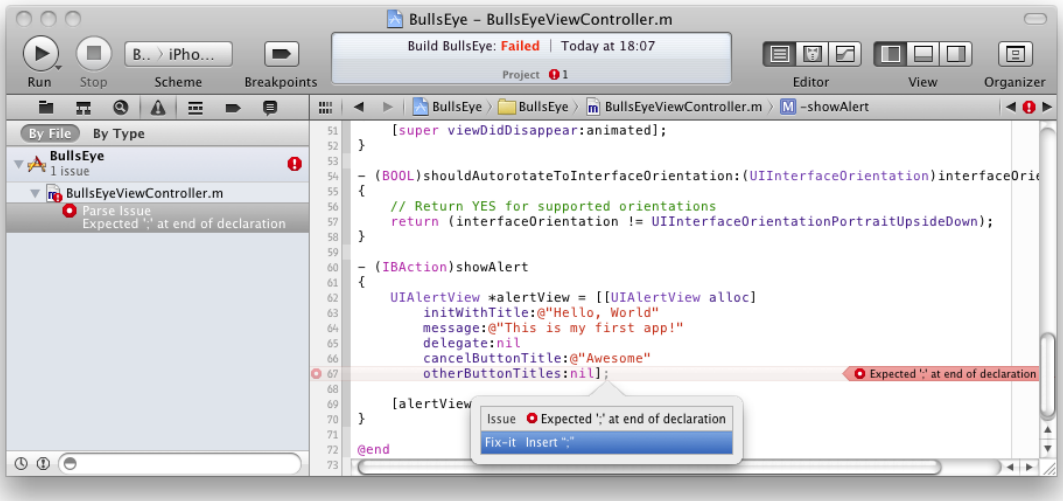
Xcode makes sure you can't miss errors



When Xcode detects an error it switches the pane on the left, where our 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 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.

How does an app work?

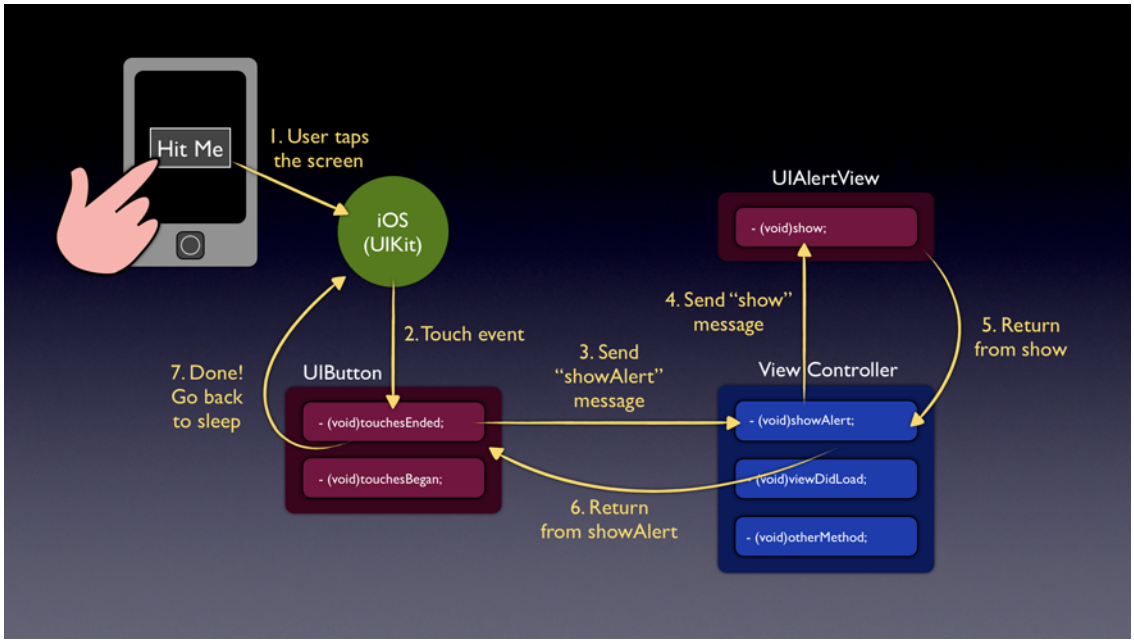
It will be good at this point to get some sense of what goes on behind the scenes of an app. An app is essentially made up of objects that can send messages to each other. Many of the objects in your app are provided by iOS, for example the button (a `UIButton` object) and the alert view (`UIAlertView`). 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 our app, for example, that `UIButton` object sends a message to our 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 our app, the button's Touch Up Inside event is connected to the view controller's `showAlert` action. When the button recognizes it has been tapped it sends the `showAlert` message to our view controller. Inside `showAlert`, the view controller sends the `show`

message 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.

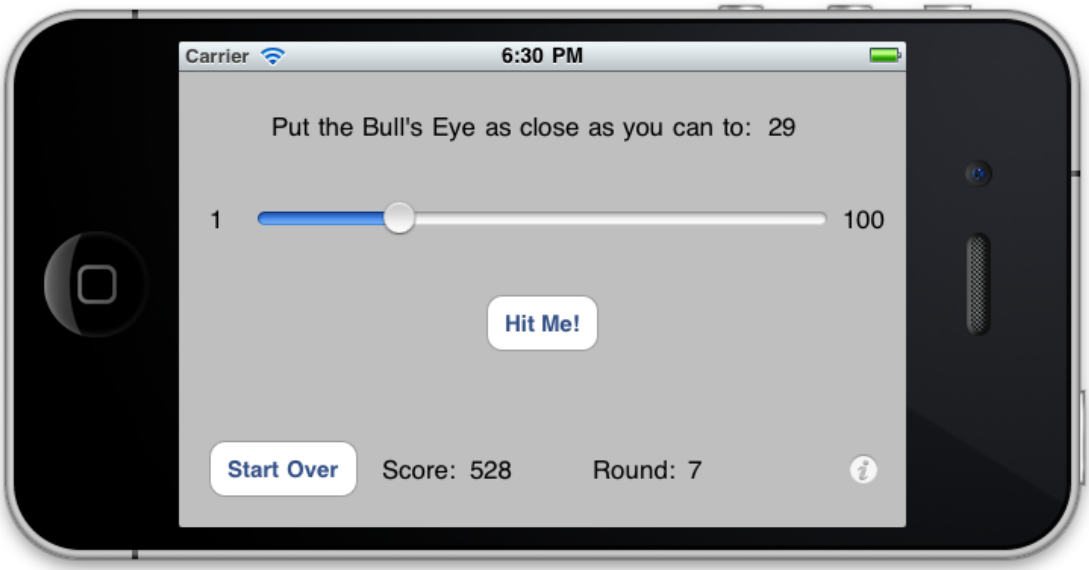
Going down the to-do list

Now that we have accomplished the first task of putting a button on the screen and making it show an alert, we'll simply go down the list and tick off the other items. We don't have to do this in any particular order, although some things make sense to do before others. For example, we cannot read the position of the slider if we 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 we're done, the app will look like this:

The Game screen with standard UIKit controls



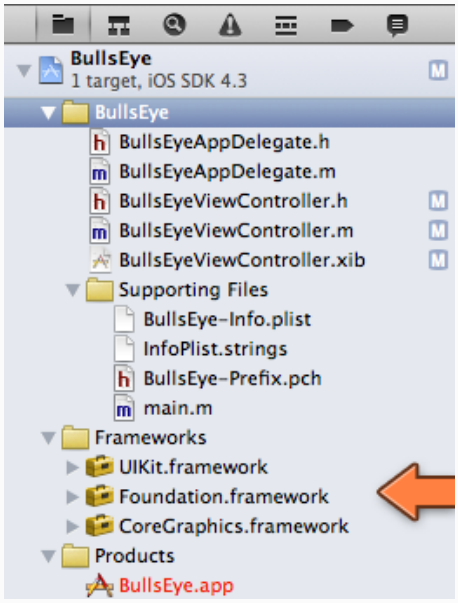
That doesn't quite look like the screenshot I showed you earlier! The difference is that these are the standard UIKit controls. This is what they look like straight out of the box. You've probably seen this look before because it is perfectly suitable for regular apps. But because the default look is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

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 in our project



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

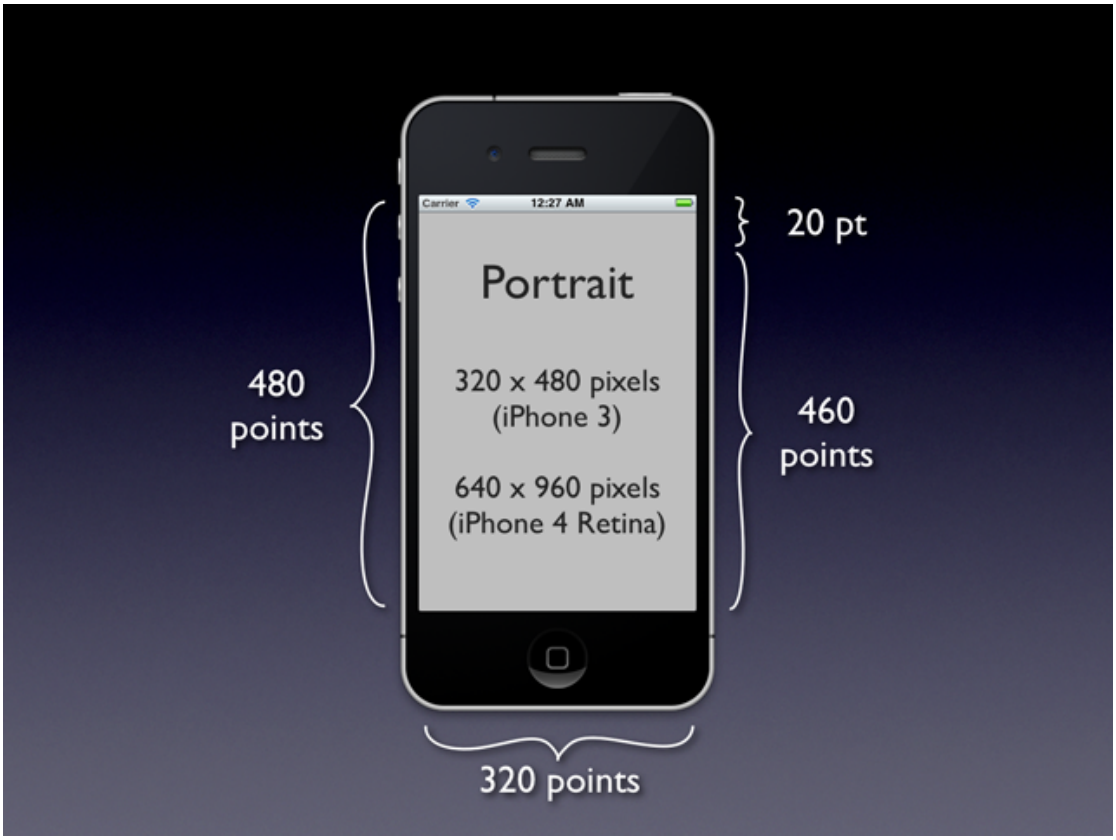
Notice that the dimensions of the screen are now slightly different from before: the iPhone is tilted on its side and the screen is wider but less tall. This is called *landscape* mode.

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

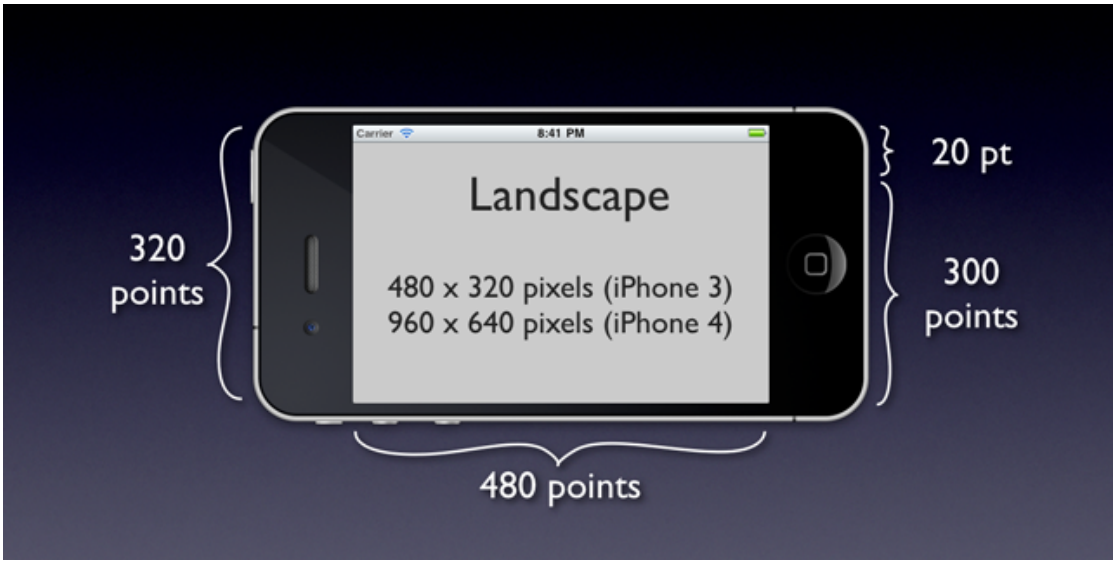
to write emails in the Mail app with their device flipped over because the wider screen allows for a bigger keyboard and easier typing.

In portrait mode, the iPhone screen consists of 320 points horizontally and 480 points vertically. For landscape these dimensions are switched: 480 points horizontally and 320 vertically. In both orientations, you lose 20 vertical points for the status bar.

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 and 2, one point corresponds to one pixel. That's easy. (I'm sure you know what a pixel [http://en.wikipedia.org/wiki/Pixel] is. In case you don't, it's the smallest element that a screen is made up of. The display of your iPhone is a big matrix of pixels that each can have their own color. Images are produced by changing the color values of these pixels.)

However, on the iPhone 4 and new iPad with their high-resolution Retina display, one point actually corresponds to two pixels horizontally and vertically, so *four* pixels in total. On older models, which I'll collectively be referring to as "iPhone 3" or "low-resolution devices", you only get 320x480 pixels but on the iPhone 4 (or "Retina devices") those dimensions are doubled to 640x960 pixels, which gives you four times as many pixels to work with.

The iPad has even more pixels because its screen is a lot bigger: 1024x768 pixels on the iPad 1 and 2, and a mind-blowing 2048x1536 pixels on the new Retina iPad. Insane!

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. ;-)

We'll get to Retina graphics later in this tutorial. Thanks to this points-vs-pixels issue, putting high-resolution graphics in your apps is actually pretty straightforward.

Converting the app to landscape

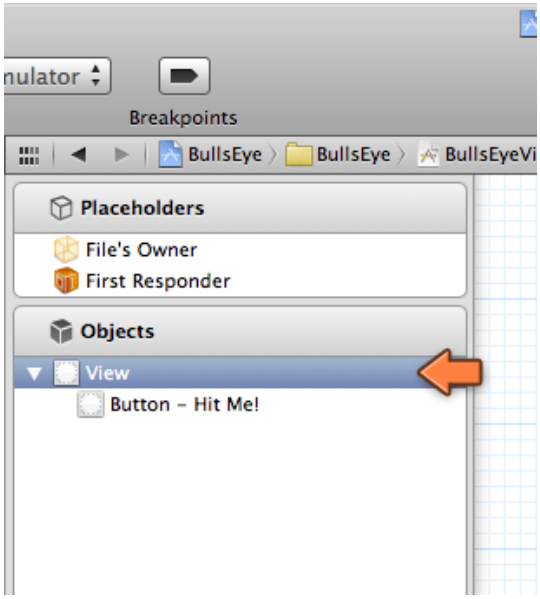
To convert our app from portrait into landscape, we have to do three things:

1. Make the view from BullsEyeViewController.xib landscape instead of portrait.
2. Change one line of code in BullsEyeViewController.m that will allow the view controller to *autorotate* to landscape mode.
3. Change the "Supported Device Orientations" settings of our app.

» Open BullsEyeViewController.xib in the Interface Builder. Click on the gray background to select it.

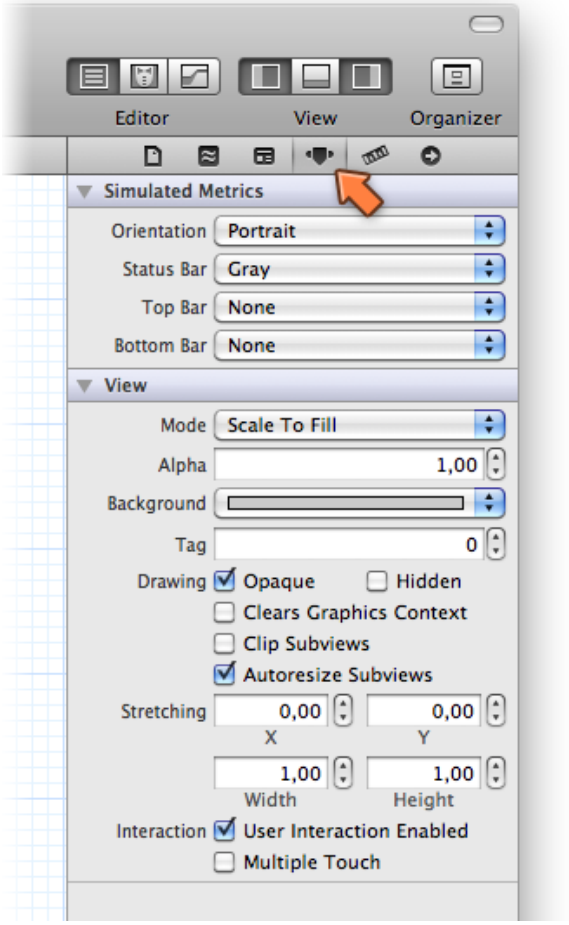
Tip: You can also click in the Objects pane. The Objects pane shows the view hierarchy of the xib. Here you can see that the main view (which is simply named View) currently contains one sub-view, the button. Sometimes it is easier to select the item you want from this list, especially if your screen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

The Objects pane shows the view hierarchy of our xib



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

The Attributes Inspector



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

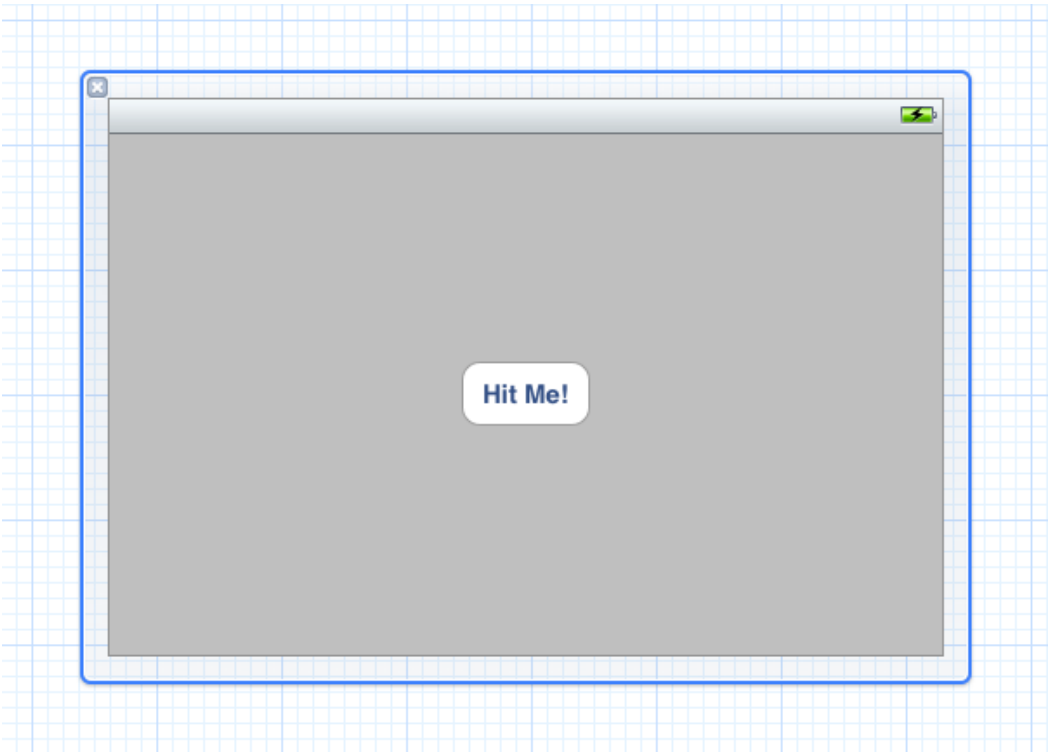
We're looking for the Orientation setting in the Simulated Metrics section. It is currently set to Portrait.

» Change Orientation to Landscape.

This changes the dimensions of the view 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. Now we have to change the view controller itself.

» Open BullsEyeViewController.m and find the bit of code that says:

BullsEyeViewController.m

```
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)↵
    interfaceOrientation
{
  // Return YES for supported orientations
  return (interfaceOrientation != UIInterfaceOrientationPortraitUpsideDown);
}
```

This was automatically put into the view controller by Xcode when it created the project from the Single View Application template. (If you're using Xcode 4.3, it does not include the green text.) This particular piece of functionality is queried by UIKit when the user rotates his iPhone. UIKit essentially asks the view controller: "Do you want to rotate to this new orientation?"

If your view controller says "yes", then the screen will rotate with the device from portrait to landscape or the other way around. If the view controller says "no" to that new orientation, the view stays the same and the user will have to turn his head to properly

read what is on the screen. UIKit doesn't assume that all apps always want to rotate, so it nicely asks the view controller for permission.

» Our app should allow landscape only, so change the code to the following:

BullsEyeViewController.m

```
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)↵
    interfaceOrientation
{
  return UIInterfaceOrientationIsLandscape(interfaceOrientation);
}
```

Even though you may not speak much Objective-C yet, you should be able to get the gist of what is going on here. This piece of source code returns "yes" if the new interface orientation is landscape and "no" if it isn't, effectively always forcing the view controller into landscape mode.

When the app starts up, UIKit asks the view controller, "Do you want to run in portrait orientation?" because that is probably how you're holding the iPhone. Thanks to our modification, `BullsEyeViewController` says: "Nope." Then UIKit asks again, this time for landscape orientation. Our view controller confirms and the screen is flipped to landscape. Even if the user rotates his device back to portrait later, the app stays in landscape.

» Run the app now and the screen shows up as landscape.

(If it doesn't, choose Hardware → Rotate Left or Rotate Right from the iOS Simulator's menubar at the top of the screen, or hold Cmd and press the left or right arrow keys on your keyboard. This will flip the simulator around.)

The app in landscape orientation

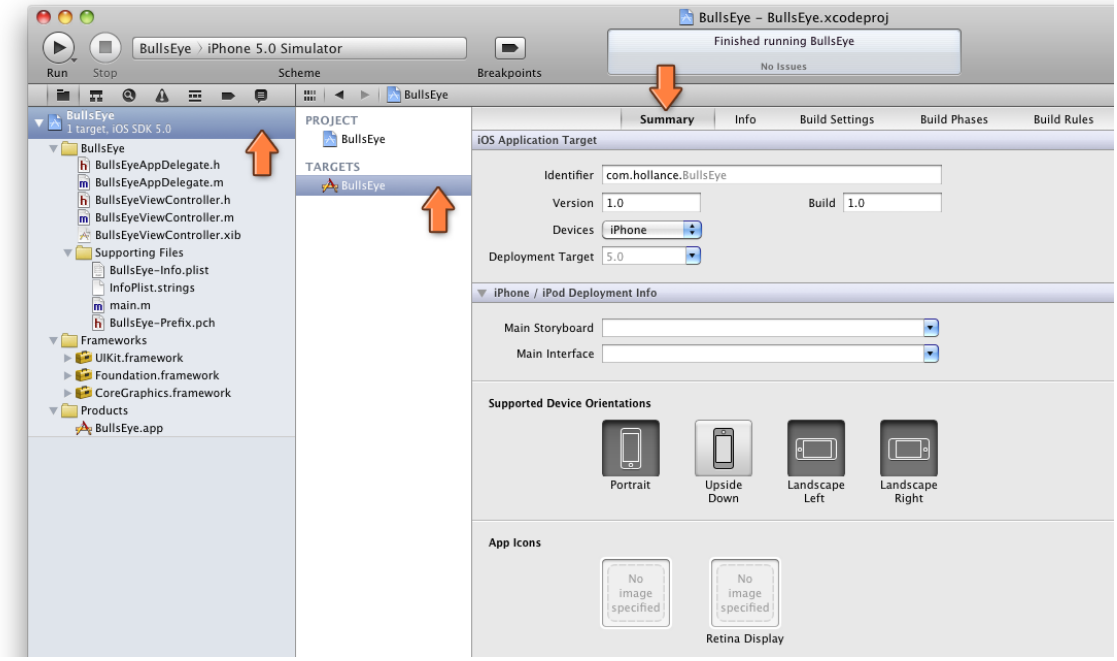
We really should do one more thing. If you look closely when the app starts up, it first assumes the portrait orientation (the position of the status bar makes this obvious) and then abruptly switches to landscape as our view controller is loaded. This happens because until `BullsEyeViewController` is active, iOS doesn't know we want our app to be in landscape.

It takes a few seconds for the operating system to load the app into memory and start it up, and during that time the screen appears in portrait. 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. Fortunately, this is an easy thing to fix.

We can set a configuration option that lets iOS know right from the beginning that the app should be launched in landscape orientation, even before it has had a chance to ask our view controller.

» 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. Under Targets click the BullsEye target and switch to the Summary tab:

The settings for the project



In the section iPhone / iPod Deployment Info, there is an option for Supported Device Orientations. In the image above the orientations Portrait, Landscape Left and Landscape Right are selected.

» Click Portrait to de-select it. If you now Run the app again, it properly launches in the landscape orientation right from the beginning.

Objects, messages and methods

Time for some programming theory.

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 our small starter app already contains several different objects. The one we have spent the most time with is `BullsEyeViewController`. The Hit Me button is also an object, as is the alert view. Our project also has an object named `BullsEyeAppDelegate`, even though we're going to ignore that for this lesson (but feel free to look inside its files if you're curious). And the texts that we put on the alert view — "Hello, World" and "This is my first app!" — are also objects. These object thingies are everywhere!

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

- An example of data is the Hit Me button that we added to the view controller's screen. When we dragged the button into the xib, 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 we added to respond to that button. Functionality *does* something.

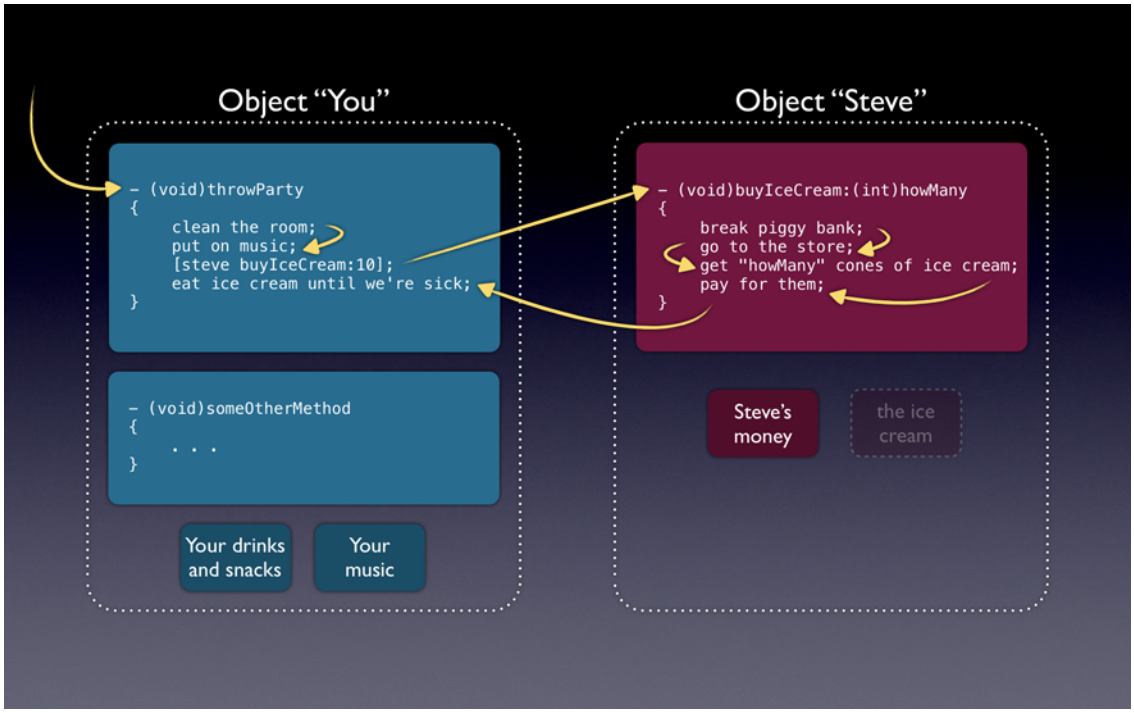
The button itself also has data and functionality. Examples of button data are the text and color of its label, its position on the screen, its width and height, and so on. The button also has functionality: it can recognize that the user taps on it and will trigger an action in response.

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.

Our `showAlert` action is an example of a method. Another method we've already seen is `shouldAutorotateToInterfaceOrientation`. 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; they were placed there by the Xcode template for our convenience. These specific methods are often used by view controllers, so it's likely that we 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") wants 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.

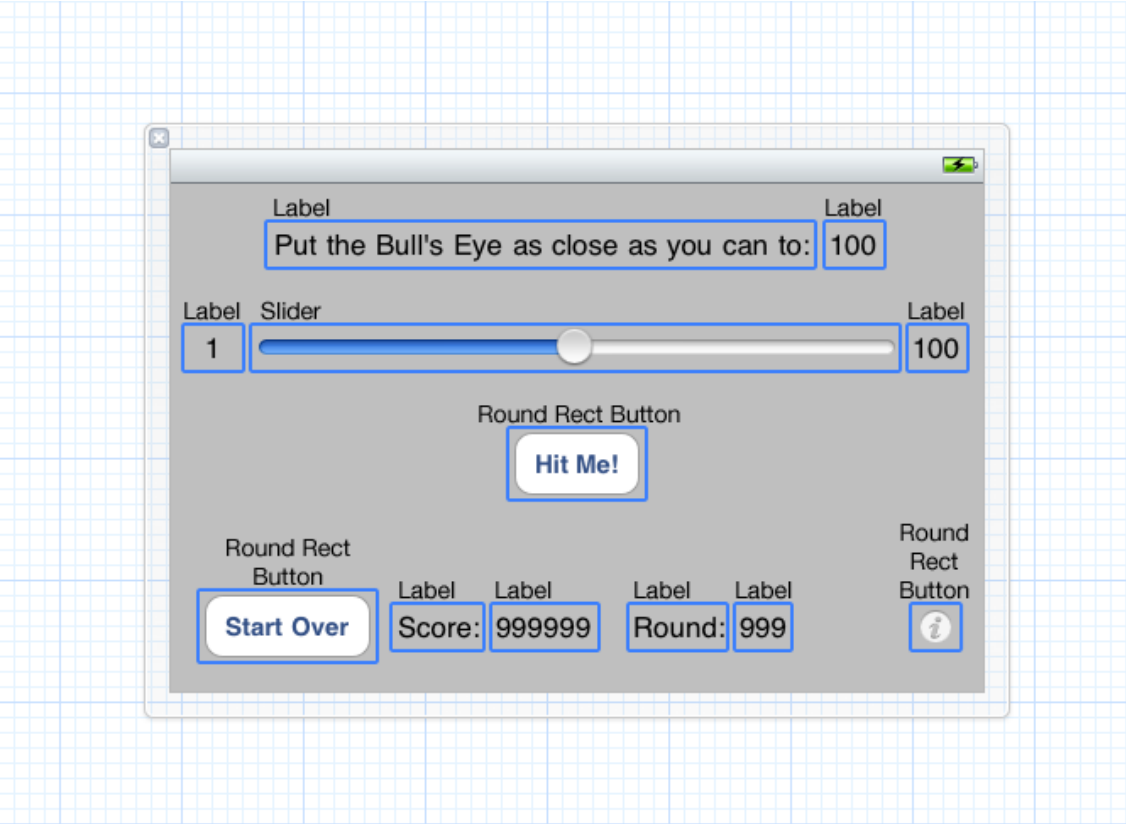
Often the terms "calling a method" or "invoking a method" are used instead. That means the exact same thing as sending a message: the computer jumps to the method you're calling and returns to where it left off when that method is done.

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

Our screen already has a button, but we 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

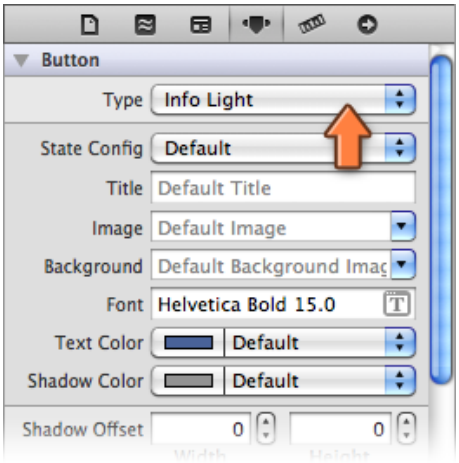


As you can see, I put placeholder values into some of the labels (for example, “999”). The reason for this is that it makes it easier to see how the labels will fit on the screen when they’re actually used. The score label could potentially hold a large value, so we’d better make sure the label has room 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.

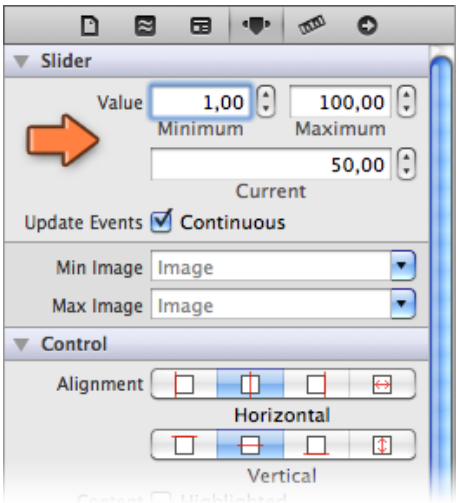
Tip: The (i) button is actually a regular Round Rect button, but its Type is set to Info Light in the Attributes Inspector:

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. (Note that slider values can have numbers behind the decimal point; on my system the decimal point is actually a comma because I'm in Europe. We want to use integral values with the slider, so just type the whole number and Interface Builder will take care of the decimal point for you. It's also possible that your version of Xcode doesn't show decimal points.)

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.

We can tick a few more items off our to-do list, all without any programming! That is

going to change really soon, because we will have to write Objective-C code to actually make the controls do anything.

Xib or nib

Xib files are also known as “nib” files. Say what? Technically speaking, a xib file is compiled into a nib file that is put into your application bundle. The term nib mostly stuck for historical reasons. You can just consider the terms “xib file” and “nib file” to be equivalent. The preferred term seems to be nib, so that is what I will be using from now on. (This won’t be the last time computer terminology is confusing, ambiguous or inconsistent. The world of programming is full of colorful slang.)

The slider

The next item on our 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 Buider, you did not accidentally disconnect the button from the `showAlert` action, we 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 we added an action to the view controller in order to recognize when the user tapped the button? We 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 we did before.

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

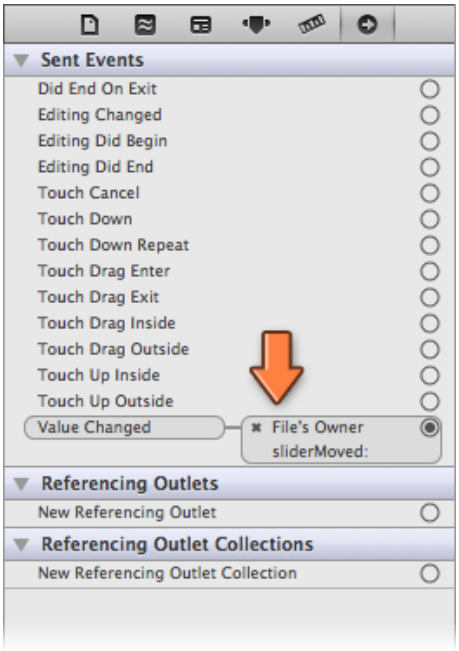
BullsEyeViewController.h

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

» Go back to the nib (same thing as xib, remember?) and Ctrl-drag from the slider to File’s Owner. 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 we've only told the slider that there is an action but we 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:

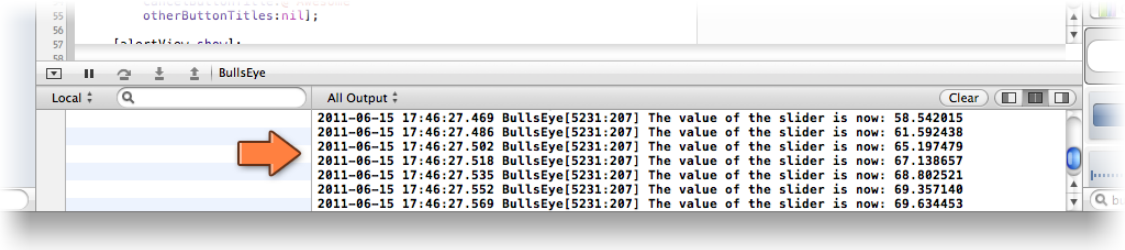
BullsEyeViewController.m

```
- (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. We used it to verify that we properly hooked up the action to the slider and that we 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

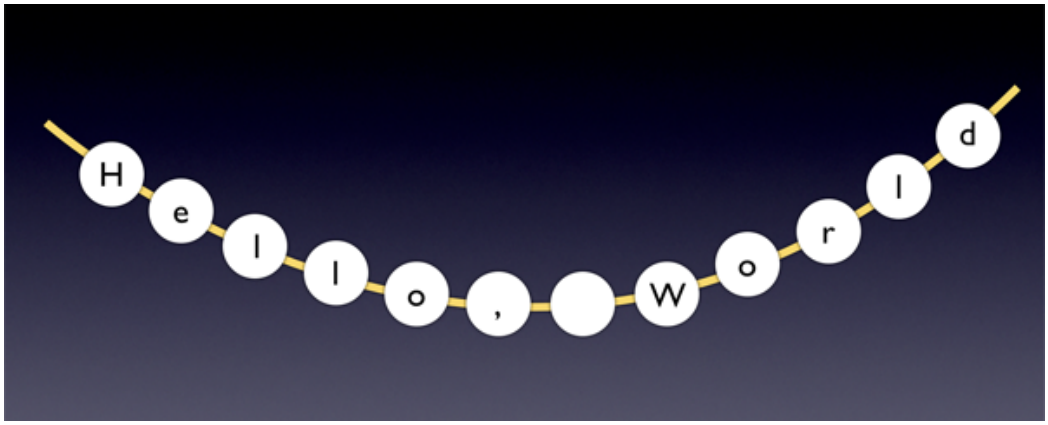
To put text in your app, you use something called a "string". The strings we have used so far are:

```
@"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 we 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 sending '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 our action method and make it show the value of the slider in the alert view. So how do we get the slider's value into `showAlert`?

When we read the slider's value in `sliderMoved`, that piece of data disappears when the action method ends. It would be handy if we 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:

BullsEyeViewController.m

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

Note: Xcode 4.3 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`.

We 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 4 spaces. (Which one you use is largely a matter of personal preference. I like to use a tab because it's less typing.)

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

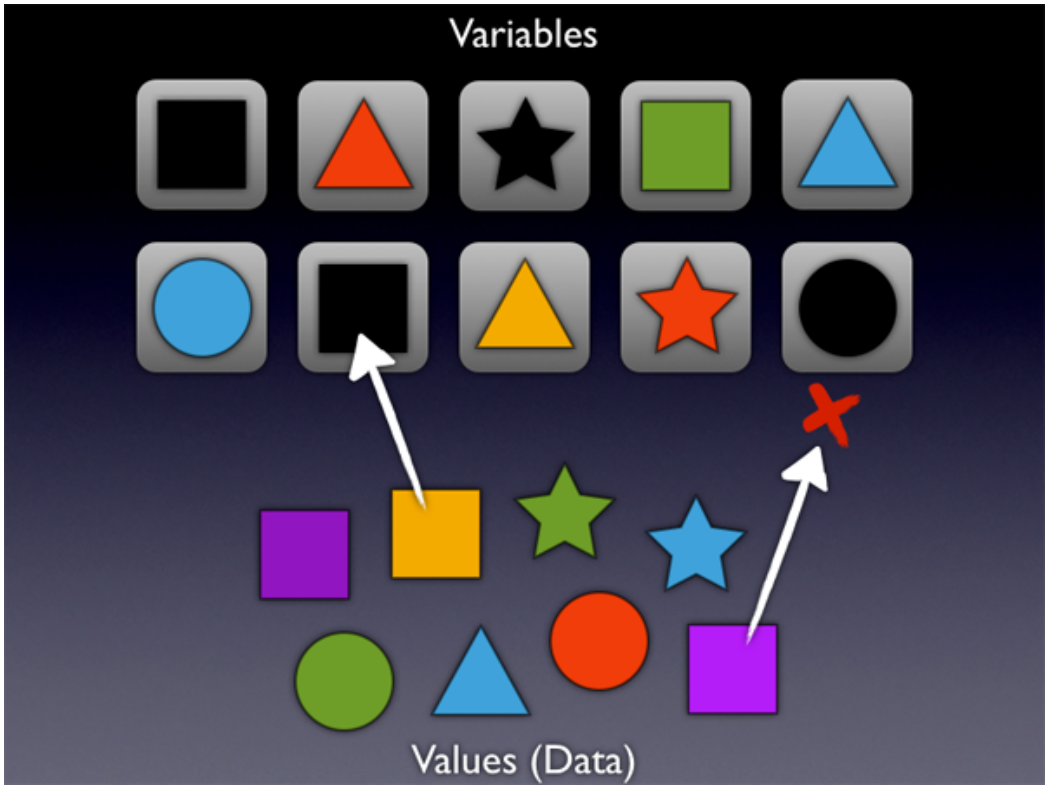
A variable allows the app to remember things. Think of a variable as a temporary storage container for a single piece of data. There are containers of all sorts and sizes, just as data comes in all kinds of shapes and sizes.

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, we 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*. We specified the datatype `int` for our `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 we put a new value into that variable or until we 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 as long as its owner, `BullsEyeViewController`. Their fates are intertwined. Our view controller, and thus `currentValue`, are there for the duration of the app. They don't get destroyed until the app quits. In a minute, we'll see variables that live much shorter.

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

» Change the contents of the `sliderMoved` action to the following:

BullsEyeViewController.m

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

We 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. `currentValue` is the name of the variable we 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. We 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 we used `NSLog()` to show the slider's value as you moved it.

However, our game would be really hard if we 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 if we used whole numbers only. That is why `currentValue` has datatype `int`, because that stores *integers*, a fancy term for whole numbers.

We use the function `lroundf()` to round the decimal number to the nearest whole number and we 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:
SomeFunction(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 we'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:

BullsEyeViewController.m

```
- (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, we 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).

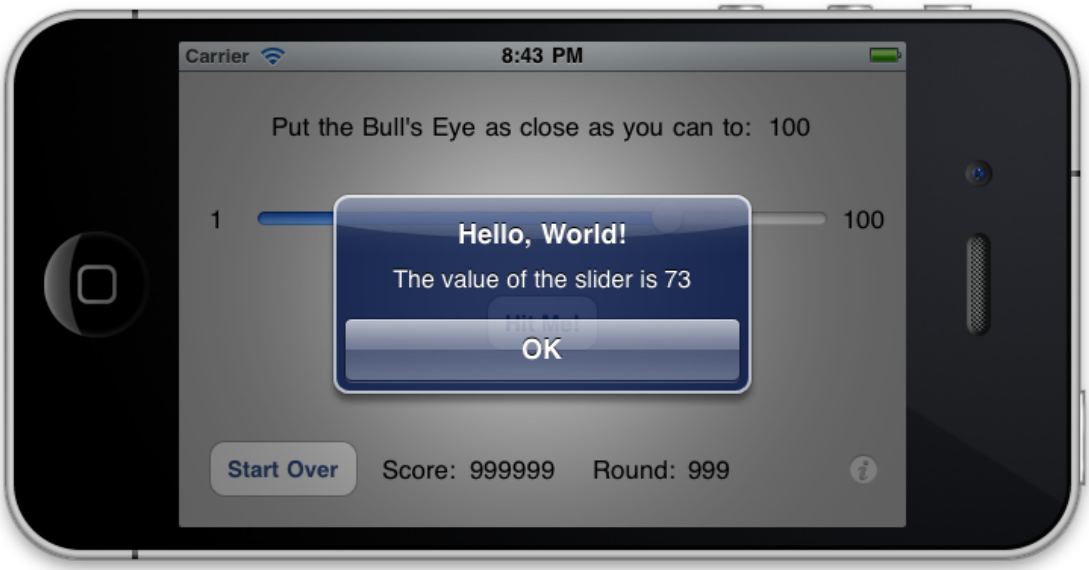
`NSString` is the iPhone's string object. Remember I said earlier that objects are everywhere? Well, strings are also objects. We create a new string object named `message`, using the method `stringWithFormat`. This method takes two parameters: a format string and the value to replace in that string.

This should look familiar. It is what we 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 we 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 put that in the `NSString` object named `message`. `NSLog()` did something similar, except that it printed the result to the Debug pane. Here, however, we 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. We've 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 our 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 we 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 an `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 named *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, `BullsEyeViewController` 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. You use instance variables if you want to keep a certain value around, from one event to the next.

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

Our first bug

There is a small problem with our 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 we would expect the value to be 50. Our first 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 we don't move the slider. Of course, if we don't move the slider, then the `sliderMoved` message is never sent and we 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 we are seeing here.

To fix this bug, we're going to do implement the `viewDidLoad` method in `BullsEyeViewController`. If you look at BullsEyeViewController.m, you should already see something like this:

BullsEyeViewController.m

```
- (void)viewDidLoad
{
  [super viewDidLoad];
  // Do any additional setup after loading the view, typically from a nib.
}
```

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

» Change the `viewDidLoad` method to the following:

BullsEyeViewController.m

```
- (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 nib file. At this point, the view controller isn't visible yet, so it's a good place to set instance variables to their proper initial values. In our case, we'll 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, 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

We managed to store the value of the slider into a variable and show it on the alert. That's good but we can still improve on it a little. What if you decide to set the initial value of the slider in the nib to something other than 50, say 1 or 100? Then `currentValue` would be wrong again because `viewDidLoad` always assumes it must 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 we would do the following.

» Change `viewDidLoad` to:

BullsEyeViewController.m

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

The idea is that we take whatever value we set on the slider in the nib (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:

BullsEyeViewController.m

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

Here we also do `slider.value` but now `slider` refers to 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 object in question to be sent along with the action method. That is convenient when you wish to refer to that object in the method, just as we did, 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. We cannot use it anywhere else.

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

» Add the following line to BullsEyeViewController.h:

BullsEyeViewController.h

```
@property (nonatomic, strong) 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.

BullsEyeViewController.h should now look something like this:

BullsEyeViewController.h

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

@interface BullsEyeViewController : UIViewController

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

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

@end
```

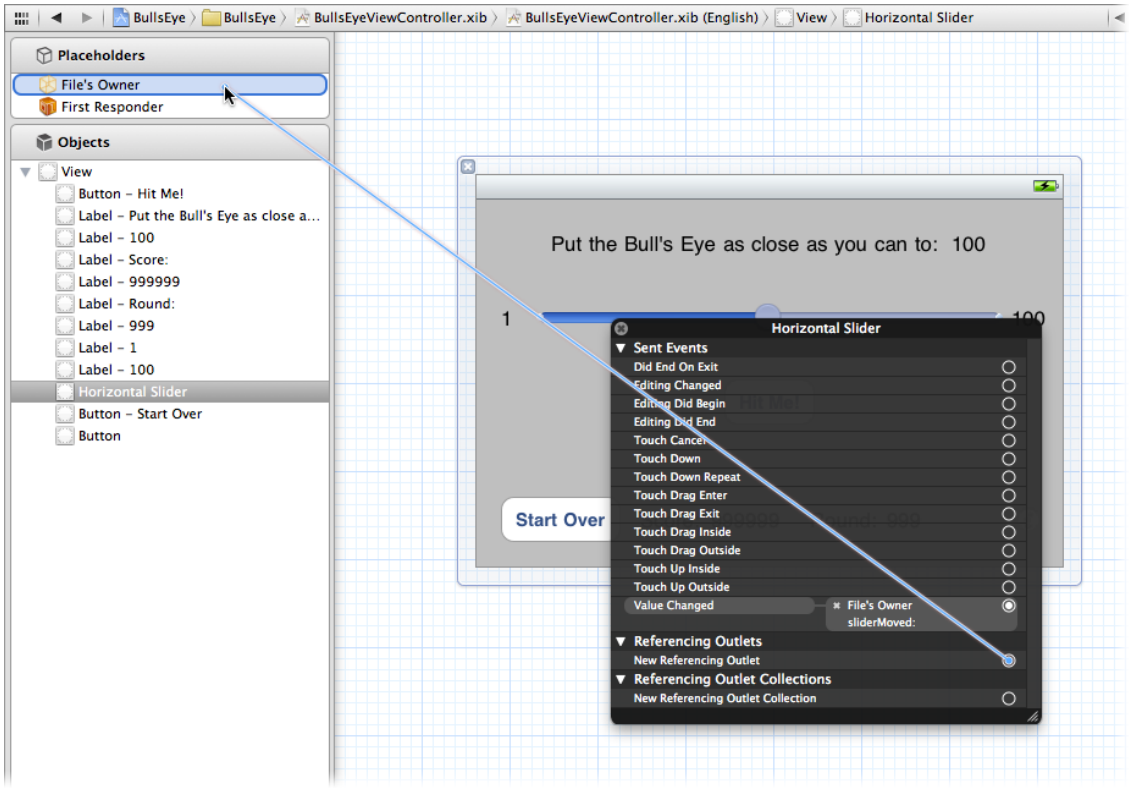
This tells Interface Builder that we now have an "outlet" named slider that can be connected to a `UISlider` object. Just as Interface Builder likes to call methods "actions", it calls these properties outlets.

» Open the nib. 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 File's Owner:

Connect the slider to the outlet



» In the popup menu that appears, select "slider".

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

To complete this job, we need to add some code to BullsEyeViewController.m.

» Directly below the `@implementation` line and the instance variable section, add:

BullsEyeViewController.m

```
@synthesize slider;
```

The top of BullsEyeViewController.m should now look like this:

BullsEyeViewController.m

```
#import "BullsEyeViewController.h"

@implementation BullsEyeViewController {
  int currentValue;
}

@synthesize slider;
```

Or if you use Xcode 4.3 or newer, it will look like this:

BullsEyeViewController.m

```
#import "BullsEyeViewController.h"

@interface BullsEyeViewController ()

@end

@implementation BullsEyeViewController {
  int currentValue;
}

@synthesize slider;
```

These three steps are necessary for just about any property you add to the view controller if that property refers to a view in the nib:

1. add `@property` to the .h file,
2. connect the outlet in Interface Builder,
3. `@synthesize` in the .m file.

(You can also make properties for things that are not in the nib and the rules are slightly different for those. But more about that later.)

The `@synthesize` line automatically adds some code to the view controller in order to be able to use the property. If you forget `@synthesize`, Xcode will complain and the app will most likely crash.

Tip: If you're using Xcode 4.4 or better, then you no longer need to add `@synthesize` statements. The designers of Objective-C decided that the compiler could figure out for itself that the properties need to be synthesized. So from now on when I say, "synthesize this property", you can simply ignore that and your app will still work. How cool is that! Unfortunately, users of Xcode 4.2 and 4.3 will still have to type in the `@synthesize` statements, so that's yet another reason to upgrade to the latest version of Xcode.

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

» Change `viewDidLoad` to:

BullsEyeViewController.m

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

The only change here is that we 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).

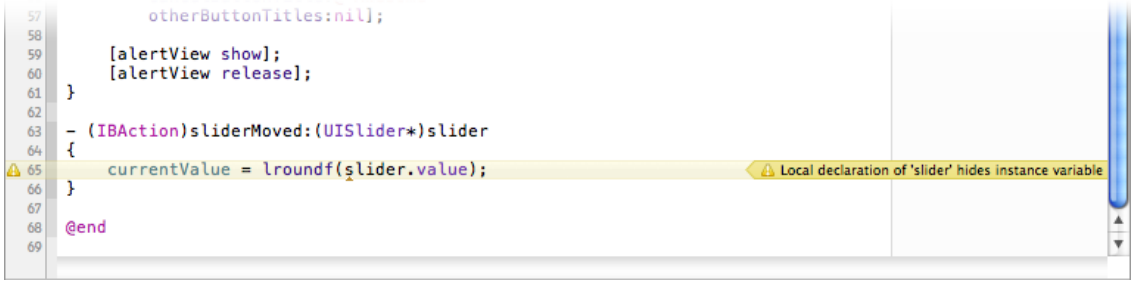
This is known as *dot-notation*, two or more names separated by dots. Whenever you see `something.somethingElse`, properties are being used.

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, 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.

By the way, Xcode might now be complaining about the `sliderMoved` method. For me, it says: "Warning: local declaration of 'slider' hides instance variable".

Warning message in Xcode



We get that warning because adding a property named `slider` also implicitly makes an instance variable named `slider`. We did not have to put that new instance variable into the view controller ourselves, that is what the `@synthesize` statement did for us. But it's certainly there!

Xcode thinks it's not very smart 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. That is why Xcode shows the warning. It's good to take Xcode's warnings seriously, so we should fix this before we continue.

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 errors. Fix the warning before you continue. 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.

A simple solution is to rename the local variable `slider` to something else.

» Change `sliderMoved` to the following:

BullsEyeViewController.m

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

The warning will now go away.

» Change the corresponding line in BullsEyeViewController.h as well so that it says:

BullsEyeViewController.h

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

I used the new name `sender`, which is really the standard term for the parameter of an action method. Other names you see people use are `aSlider`, `theSlider` or `_slider`. It doesn't really matter what you name the parameter, as long as it doesn't conflict with the names of any instance variables or properties. As with many other things in programming, the actual way you do things is often a matter of personal preference.

Properties vs instance variables

Properties and instance variables have a lot in common. In fact, when you use `@synthesize` to create the property, it is "backed" by an ivar. That means our `slider` property stores its value in an instance variable, also named `slider`, that 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 over using a property versus an instance variable? There are several reasons, but mainly ivars 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's what we did with Interface Builder: we hooked up the slider object to the view controller's `slider` property. However, you can't connect things from within Interface Builder to the view controller's private parts, the instance variables. Much more about this in later tutorials because it's an important topic in object-oriented programming.

By the way, if you look at older books or tutorials (from before 2010), you will notice that people back then did explicitly create the backing instance variable for their properties. So if they had a `UISlider` property named `slider`, they would also add a `slider` instance variable:

```
@interface BullsEyeViewController : UIViewController {
  UISlider *slider;   // not really necessary
}

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

@end
```

Those duplicate declarations used to be necessary, but they no longer are. So you can save yourself some typing and leave it to `@synthesize` to take care of making the ivar. Also note that the ivar in this example is added to the `@interface` section of the view controller. As of iOS 5 we can add instance variables to the `@implementation` section instead, which is a better place for them.

Generating the random number

We still have quite a ways to go before our game is playable, so let's get on with the next item on our 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 we can employ a so-called *pseudo-random generator* to spit out numbers that at least appear that way. We'll use my favorite one, the `arc4random()` function.

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

» Add the following line to `viewDidLoad`:

```
targetValue = 1 + (arc4random() % 100);
```

So `viewDidLoad` should now look like:

BullsEyeViewController.m

```
- (void)viewDidLoad
{
  [super viewDidLoad];
  currentValue = self.slider.value;
  targetValue = 1 + (arc4random() % 100);
}
```

What did we do here? First, we're using a new variable, `targetValue`. We haven't actually defined this variable yet, so we'll have to do that in a minute. (If we 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 we're using the variable properly everywhere.)

We're also calling the function `arc4random()` to deliver an arbitrary integer (whole number) between 0 and about 4 billion. Our slider doesn't go quite that far so we want to limit the random number to be between 1 and 100. To accomplish that, we first use the modulo operator (%) to restrict the result of `arc4random()` to a number between 0 and 99. Then we add 1 to this number to get it in the range 1 - 100.

If you're unsure of the modulo operator: it returns the remainder of a division. You may have seen this in maths class. For example 13 % 4 = 1, because 13 divided by 4 is 3.25, which is 3 with a leftover of 0.25. The modulus of 13 and 4 is therefore 1 because what remains is one-fourth. However, 12 % 4 is 0 because there is no remainder.

The remainder always lies between 0 and the denominator (exclusive), so if we do `arc4random()% 100` we get a random number between 0 and 99 (inclusive). I hope this calculation didn't freak you out too much. It's often said you need to be good at maths to be a good programmer, but really the math in most apps isn't more complicated than this kind of arithmetic.

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

» Add the `targetValue` variable to the instance variable section:

BullsEyeViewController.m

```
@implementation BullsEyeViewController {
  int currentValue;
  int targetValue;
}

. . .

@end
```

The rest of this file doesn't change, only the line `int targetValue;` 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!)

I hope the reason is clear why we make `targetValue` an instance variable. We want to calculate the random number in one place (`viewDidLoad`) and then remember it until the user taps the button (`showAlert`).

» Change `showAlert` to the following:

BullsEyeViewController.m

```
- (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];
}
```

We'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 we 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!

Rounds

If you pressed the Hit Me button a few times, you'll have noticed that the random number never changes. I'm afraid the game won't be much fun that way. This happens because we 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 our game.

When the game starts, the player has a score of 0 and the round number is 1. We set the slider halfway (to value 50) and calculate a random number. Then we wait for the player to press the Hit Me button. As soon as he does, the round ends. We calculate the points for this round and add them to the total score. Then we increment the round number and start the next round. We 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 above `viewDidLoad`:

BullsEyeViewController.m

```
- (void)startNewRound
{
```

```
  targetValue = 1 + (arc4random() % 100);

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

» And change `viewDidLoad` to:

BullsEyeViewController.m

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

It's not very different from what we did before, except that we moved the logic for setting up a new round into its own method, `startNewRound`. The advantage of doing this is that we can also call this method after the player pressed the button, from `showAlert`.

» Make the following changes to `showAlert`:

BullsEyeViewController.m

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

  [alertView show];

  [self startNewRound];
}
```

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

So far the methods from the view controller have been invoked for us 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.

It is also possible to call methods by hand, using the syntax `[object methodName]`. We already did this on `NSString` when we called `[NSString stringWithFormat]` and on `UIAlertView` when we 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 we didn’t, the code for `viewDidLoad` and `showAlert` would look like this:

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

  targetValue = 1 + (arc4random() % 100);
  currentValue = 50;
  self.slider.value = currentValue;
}

- (IBAction)showAlert
{
  . . .

  [alertView show];

  targetValue = 1 + (arc4random() % 100);
  currentValue = 50;
  self.slider.value = currentValue;
}
```

Can you see what is going on here? We 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 we decide to make a change to this logic (as we will shortly)? Then we 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() % 100);
currentValue = 50;
self.slider.value = currentValue;
```

You probably have to reason your way through it: "We're calculating a new random number and then reset 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 we did before (we 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 in the label

We figured out how to calculate the random number and how to store it in an instance variable, `targetValue`, so that we can access it later. Now we 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 we made the nib, we 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, we need to accomplish two things:

1. Create a reference to the label so we can send it messages

2. Give the label new text to display

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

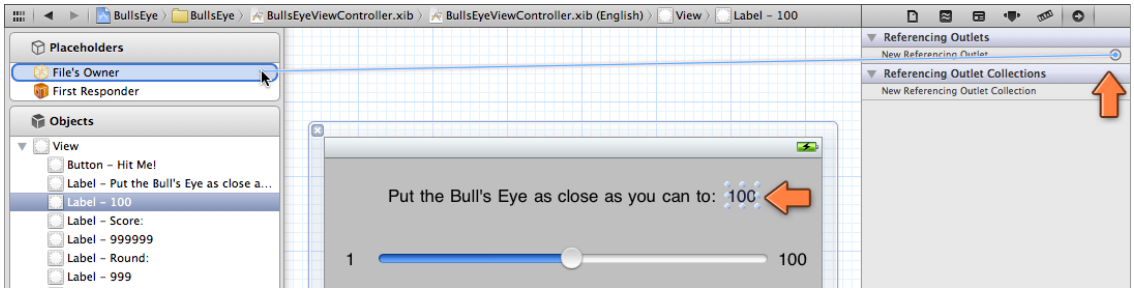
» In BullsEyeViewController.h, add the following line below the other property:

BullsEyeViewController.h

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

» In BullsEyeViewController.xib, click to select the label. Go to the Connections Inspector and drag from "New Referencing Outlet" to File's Owner. 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 File's Owner: 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.

» Add the following in BullsEyeViewController.m, below the other `@synthesize` line:

BullsEyeViewController.m

```
@synthesize targetLabel;
```

This is the housekeeping you have to do when you create an outlet property for something in your nib: create the property, connect it in Interface Builder, and synthesize it.

» Now on to the good stuff. Add the following method above `startNewRound`:

BullsEyeViewController.m

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

I put this logic in its own method because it's something we 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 updating a single label, but later on we will add code to update the other labels (total score, round number) as well.

The code inside `updateLabels` should have no surprises for you, although you may wonder why we 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. We can only put string values into an `NSString` but the code above 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 we have to convert the `int` into a string, and that is what `stringWithFormat` does. We'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 we actually call it. A logical place would be after each call to `startNewRound`, because that is where we calculate the new target value.

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

» Change `viewDidLoad` and `showAlert` to:

BullsEyeViewController.m

```
- (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.

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;

// 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 ebook series, *The iOS Apprentice: iPhone and iPad Programming for Beginners*. The full series consists of several more of these huge tutorials and in each we 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 ebooks 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 we give the game a makeover and add support for the iPhone 4's Retina display.
- We will take full advantage of new iOS 5 features such as Automatic Reference Counting (ARC) and Storyboards. There is no point in teaching you the *old* way of iOS development. iOS 5 adds improved development techniques and we'll use these to our benefit.
- Not just a bunch of dry theory but hands-on practical advice! I'll explain how everything works along the way while we're making real apps, with lots of pictures that clearly illustrate what is going on.

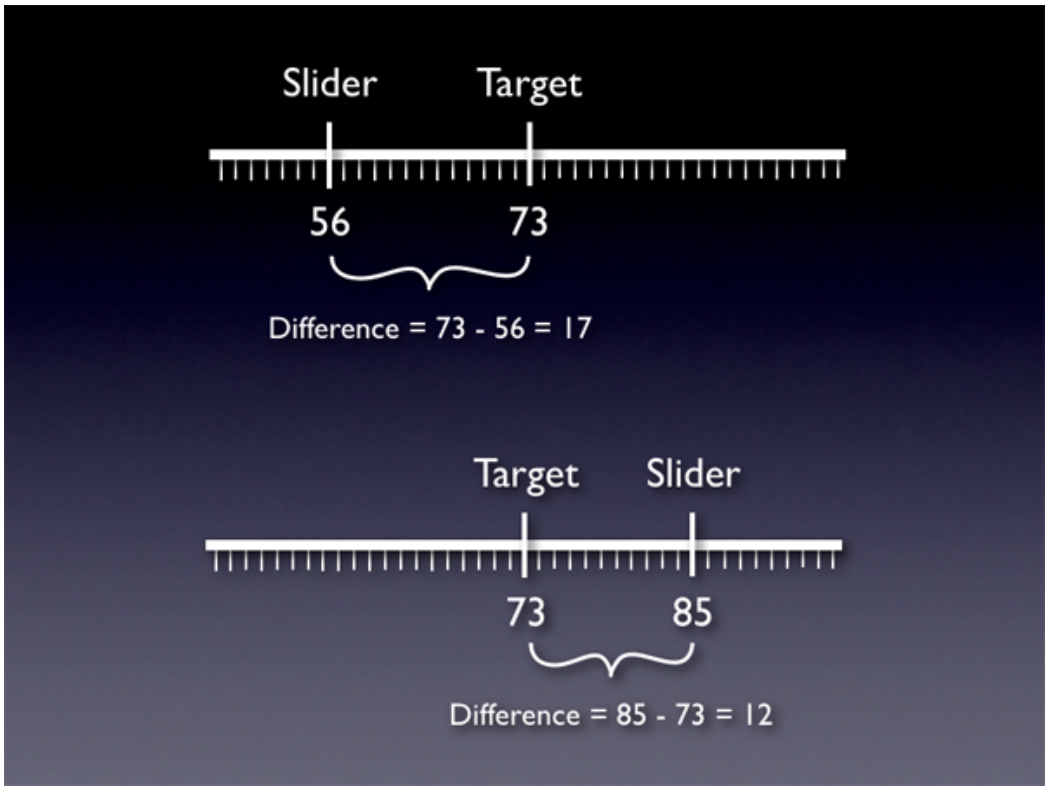
If this sounds like your idea of a fun time, then head on over to raywenderlich.com [http://www.raywenderlich.com/ios-apprentice] to get the other ebooks from the *iOS Apprentice* series.

Calculating the score

Now that we have both the target value (the random number) and a way to read the slider's position, we 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, we 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`).

We don't want to subtract points from the score, so we need to turn that negative value into a positive value. Doing the subtraction the other way around (`currentValue` minus `targetValue`) won't solve things because now 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 would come 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 we 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 would be 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 we'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 we can use in our own programs, so that saves us 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 simple such as the one above, some 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 we 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 our 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 our little algorithm we 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 we compare the `currentValue` against the `targetValue`. First we 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;
```

We 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 to me 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 meaning.

Back to our if-statement. If `currentValue` is equal to or less than `targetValue`, the condition is untrue (or *false* in computer-speak) and we will skip the code block until we reach 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 following line will only be executed when `targetValue` is the greater of the two values:

```
  difference = targetValue - currentValue;
```

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

There is only one situation we 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 we've already determined that one value is not greater than the other, nor is it smaller, leaving us only one conclusion to draw: the numbers must be equal.

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

BullsEyeViewController.m

```
- (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 we can see that it works, I've added the `difference` value to the alert view message as well.

» Run it and see for yourself.

Alternative ways to calculate the difference

I mentioned earlier there are other ways to calculate the difference between `currentValue` and `targetValue` as a positive number. The above algorithm works 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 I was thinking of 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 we're no longer avoiding the negative number, as computers can work fine with negative numbers, but we simply turn it into a positive number if we find one.

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. We first subtract the two variables and put the result into the `difference` variable. Then we use an `if` to determine whether `difference` is negative, i.e. less than zero. If it is, we 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 we put 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 we've seen the asterisk symbol before, for example with `UISlider *slider`, it meant that we 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.)

We could also have written this algorithm as follows:

```
int difference = currentValue - targetValue;
if (difference < 0) {
  difference = -difference;
}
```

Instead of multiplying by -1, we 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:

BullsEyeViewController.m

```
- (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 our computation does not change, only the technique we used.

The third alternative is to use a function. You've already seen functions a few times before: we used `lroundf()` for rounding off the slider's decimals and `arc4random()` when we made random numbers. To make sure a number is always positive, we 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 we need here and the

standard library contains a convenient function for it, which allows us to reduce our entire problem to a single line:

```
int difference = abs(targetValue - currentValue);
```

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

BullsEyeViewController.m

```
- (IBAction)showAlert
{
  int difference = abs(targetValue - currentValue);

  NSString *message = . . .
}
```

It really doesn't matter whether we 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 we have the difference, calculating the player's score for this round is easy.

» Change `showAlert` to:

BullsEyeViewController.m

```
- (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.

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

We want to show the player's total score on the screen. After every round, we should add the newly scored points to the total and then update the score label. Because we want to keep this total score around for a long time, we will put it in an instance variable.

» Add a new `score` instance variable:

BullsEyeViewController.m

```
@implementation BullsEyeViewController {
  int currentValue;
  int targetValue;
  int score;
}

. . .

@end
```

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

» Make the following changes:

BullsEyeViewController.m

```
- (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. We just added the line:

```
score += points;
```

This will add the points that the user scored in this round to the total score. Note that we 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 (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`, we have four local variables and we 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;
```

```
}

. . .

@end
```

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 we put `int` in front of `score`, we have made a new local variable that is only valid inside this method. In other words, we wouldn't add `points` to the *instance variable* `score` but to a new *local variable* that also happens to be named `score`. The ivar `score` never gets changed, even though it has the same name. Obviously this is something you want to avoid.

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 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.

Underscores in ivar names

If you read other people's code, sometimes you will see ivars whose names begin with a leading underscore. So in our app they would be named `_score`, `_currentValue` and `_targetValue`. Some programmers like to do this so they can easily spot whether a variable is an instance variable (it starts with an underscore) or a local variable (it doesn't).

The underscore has no particular meaning in the Objective-C language, it is just a convention. I've seen others use an "m" (for member) or an "f" (for field) prefix for the same purpose. My personal preference is not to use a prefix (I think it looks ugly), but by all means do so if you think it will be handy.

Showing the score on the screen

We're going to do exactly the same thing that we 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:

BullsEyeViewController.h

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

You went into the nib file and dragged from File's Owner to the label to connect the label to the `scoreLabel` property.

You added the following to the .m file directly below the other synthesize statements:

BullsEyeViewController.m

```
@synthesize scoreLabel;
```

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

» Change `updateLabels` to the following:

BullsEyeViewController.m

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

Nothing new here. We 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.

Speaking of rounds, we also have to increment the round number each time we start 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 we had to add another ivar, then you were right. You should have added the following line to the instance variables section (the block in between the { } brackets following `@implementation`):

BullsEyeViewController.m

```
  int round;
```

And also a property for the label:

BullsEyeViewController.h

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

As before, you should have connected the label to this property in Interface Builder and you synthesized this property in the .m file.

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 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:

BullsEyeViewController.m

```
- (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, we call this method whenever we start a new round. It makes sense to increment the round counter there.

» Change `startNewRound` to:

BullsEyeViewController.m

```
- (void)startNewRound
{
  round += 1;

  targetValue = 1 + (arc4random() % 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 we call `startNewRound` for the very first time, we add 1 to this default value and 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.

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

We could leave it at this. The gameplay rules have been implemented and the logic doesn't seem to have any big flaws. As far as I can tell, there are no bugs. But I can still see some room for improvement. Obviously, the game is not very pretty yet and we will get to work on that soon. In the mean time, there are a few smaller tweaks we can make.

Unless you already changed it, the title of the alert view still says "Hello, World". We could give it the name of the game, "Bull's Eye", but I have a better idea. What if we 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 we create the `UIAlertView`. We already make the message text dynamically and now we will do something similar for the title text.

» Here is the changed method in its entirety:

BullsEyeViewController.m

```
- (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];
}
```

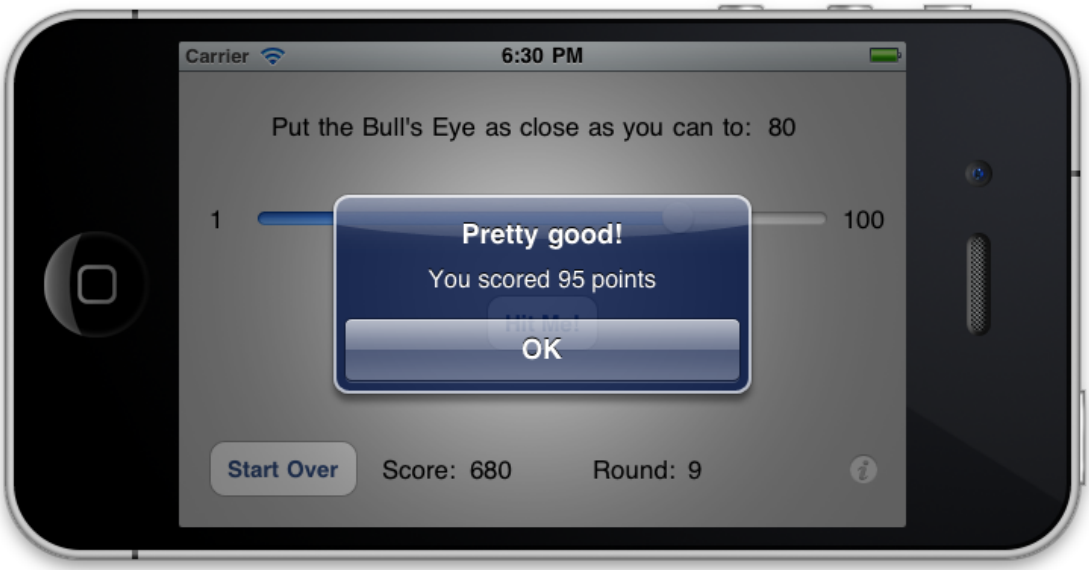
We create a new local string variable named `title`, which will contain the text that goes at the top of the the alert view. Initially, this `title` variable doesn't have any value.

To decide which title text to use, we look at the difference between the slider position and the target. If it equals 0, then the player was spot-on and we put the text "Perfect!" into `title`. If the difference is less than 5, we use the text "You almost had it!". A difference less than 10 is "Pretty good!". However, if the difference is 10 or greater, then we 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 we create the `UIAlertView` object, we 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:

BullsEyeViewController.m

```
- (IBAction)showAlert
{
  int difference = abs(targetValue - currentValue);
  int points = 100 - difference;

  NSString *title;
  if (difference == 0) {
    title = @"Perfect!";
    points += 100;
  } else if (difference < 5) {
    if (difference == 1) {
      points += 50;
    }
    title = @"You almost had it!";
  } 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 that I added a new statement between its curly brackets:

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

When the difference is equal to zero, we 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) {
    if (difference == 1) {
      points += 50;
    }
    title = @"You almost had it!";
  }
```

There is now an `if` inside another `if`. Nothing wrong with that! If the difference is more than zero but less than five, it could also be one (but not necessarily all the time). Therefore, we perform an additional check to see if the difference truly was 1, and if so, add 50 extra points.

Finally, I moved the line `score += points;` below the `if`s. This is necessary because we might update the `points` variable inside those if-statements and we 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 we're still watching the results of the last round. I find this 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 we only call `startNewRound` after we've shown the alert view:

BullsEyeViewController.m

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

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

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

  [self startNewRound];
  [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 us 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 we can't wait in `showAlert` until the alert view is dismissed, then how do we wait for it to close? The answer is simple: events! As you've seen, a lot of the programming we do in iOS involves waiting for specific events to occur (buttons being tapped, sliders being moved, and so on). This is no different.

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

This is also known as the *listener pattern* or *observer pattern*. Our view controller listens 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:

BullsEyeViewController.h

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

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

» Change the bottom bit of `showAlert` to:

BullsEyeViewController.m

```
- (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 view controller is now its delegate. We have also removed the calls to `[self startNewRound]` and `[self updateLabels]` because that is the code we want to execute when the alert closes.

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

BullsEyeViewController.m

```
- (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." Our 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, we start the new round.

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

Delegates in two steps

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

1) You declare yourself capable of being a delegate. We did that by including `<UIAlertViewDelegate>` in our `@interface` line. That 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.

Starting over

No, we'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 could use this button if you're 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. ■

First things first. Let's connect the Start Over button to an action.

» Add the following to the .h file, below the other actions:

BullsEyeViewController.h

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

» Open the nib and Ctrl-drag from the Start Over button to File's Owner. Let go of the mouse button and pick "startOver" from the popup. That connects the button's "Touch Up Inside" event to the action we have just defined.

» Add the following to .m below `showAlert` or `sliderMoved`:

BullsEyeViewController.m

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

That looks quite reasonable. If the Start Over button is pressed, we 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). We 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 our ivars we call `updateLabels` to update the text of the corresponding score, round and target labels.

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

BullsEyeViewController.m

```
- (void)startNewGame
{
  score = 0;
  round = 0;
  [self startNewRound];
}
```

Notice that we set `round` to 0 here. We do that because incrementing `round` is the first thing `startNewRound` does. If we 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 we start 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:

BullsEyeViewController.m

```
- (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.

Above or below?

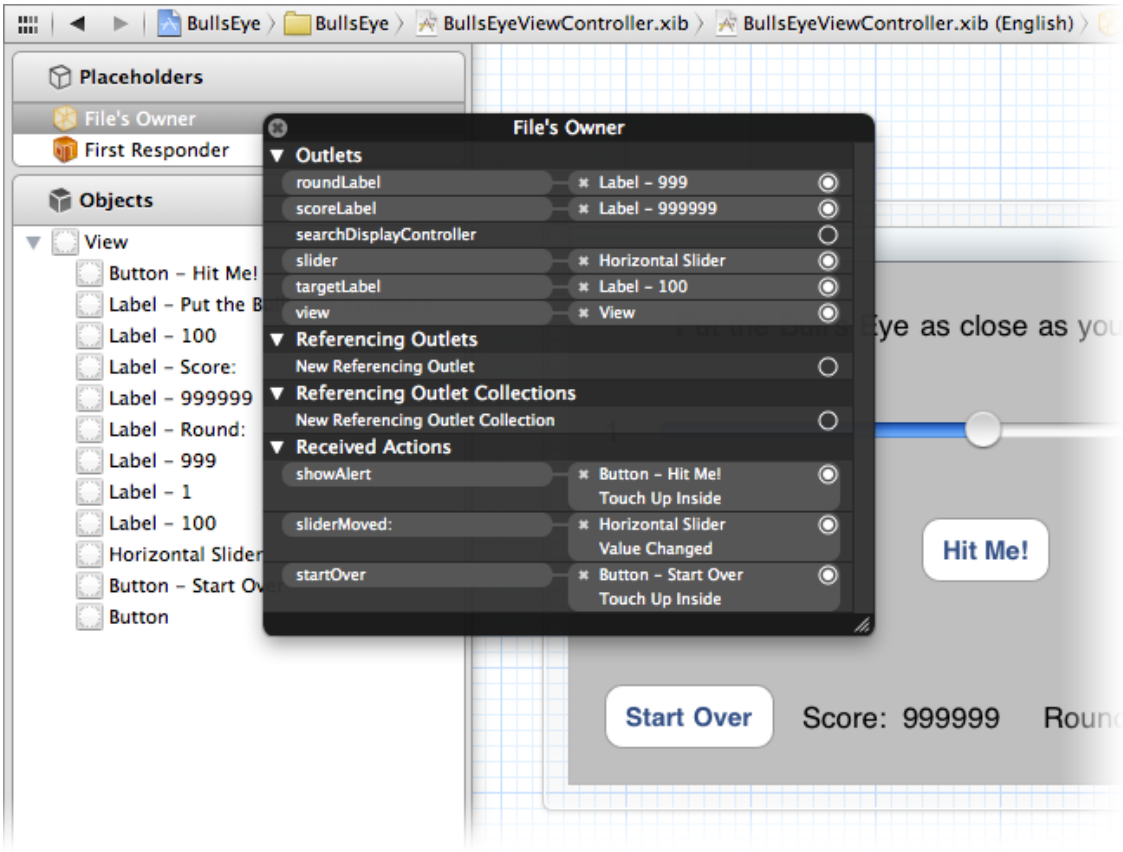
In case you are wondering why I ask you to sometimes put a method below or above another one, here's a short explanation: you cannot call a method if the compiler doesn't know about it yet. If `startNewGame` was defined below `viewDidLoad`, then Xcode would give an error when `viewDidLoad` tries to call `startNewGame`. Like humans, the compiler reads from top-to-bottom.

The simplest solution is to put `startNewGame` above `viewDidLoad`, so that when the compiler gets to translating the `viewDidLoad` method it will already have seen `startNewGame`. There are other ways to achieve this (so-called forward declarations) and we'll use these in later tutorials.

Update: One of the great new features of the compiler that ships with Xcode 4.3 is that this actually isn't necessary anymore. You can now place your methods anywhere in the .m file, as long as they are in the `@implementation` section. That's one more reason to upgrade to Lion and get the latest Xcode. :-)

Tip: If you're losing track of what button or label is connected to what method, you can click on File's Owner to see all the connections that you have made so far. You can either right-click File's Owner to get a popup, or simply view the connections in the Connections Inspector. This shows all the connections that have been made to File's Owner, which as you'll remember is `BullsEyeViewController`.

All the connections from File's Owner to the other objects

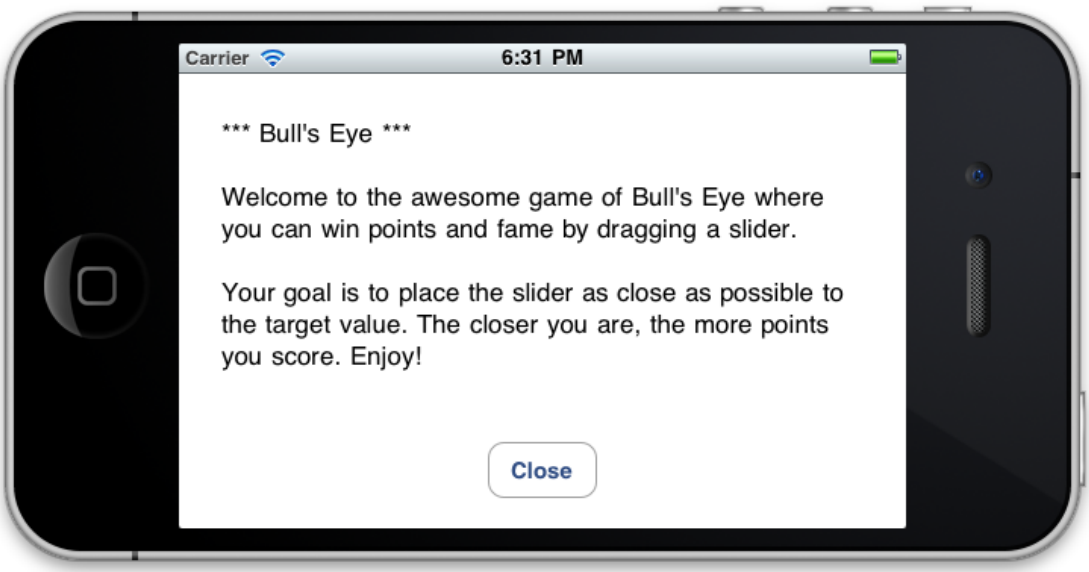


You can find the project files for the current version of the app under "05 - Polish" in the tutorial's Source Code folder.

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



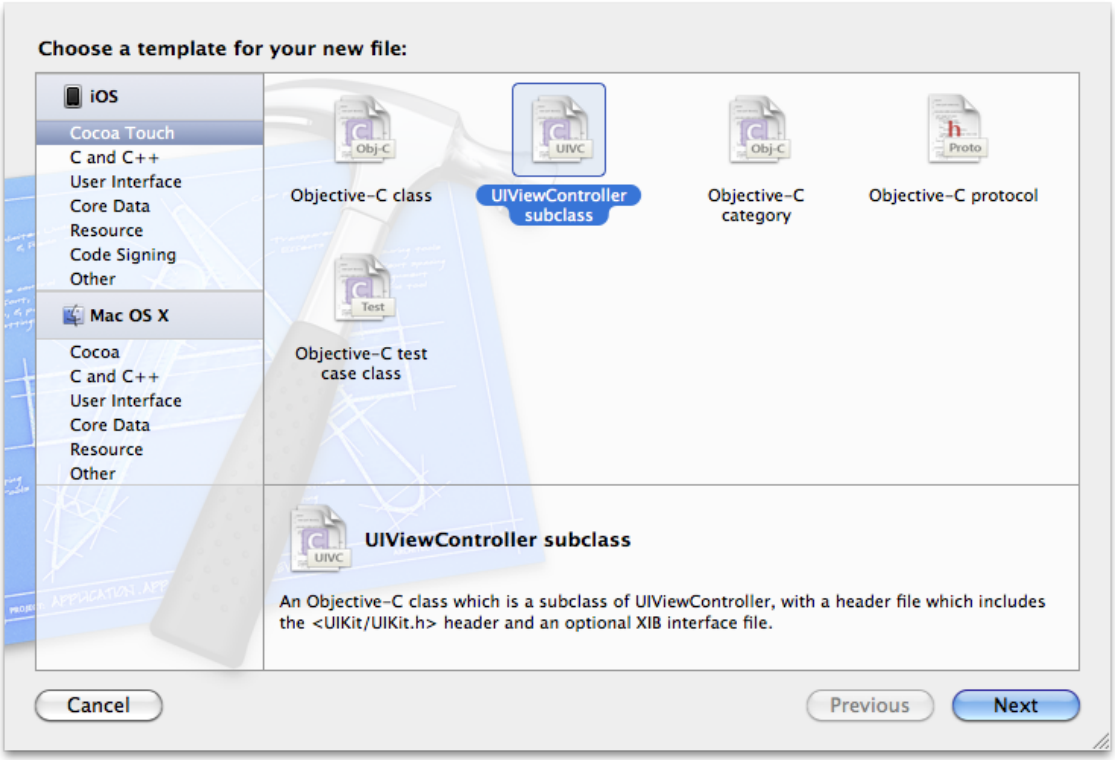
This new screen contains a so-called *text view* with the gameplay rules and a button that lets the player close the screen. You get to the About screen by tapping the (i) button in the bottom-right corner of the main game screen.

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 us, but the view controller for the About screen we'll have to make ourselves.

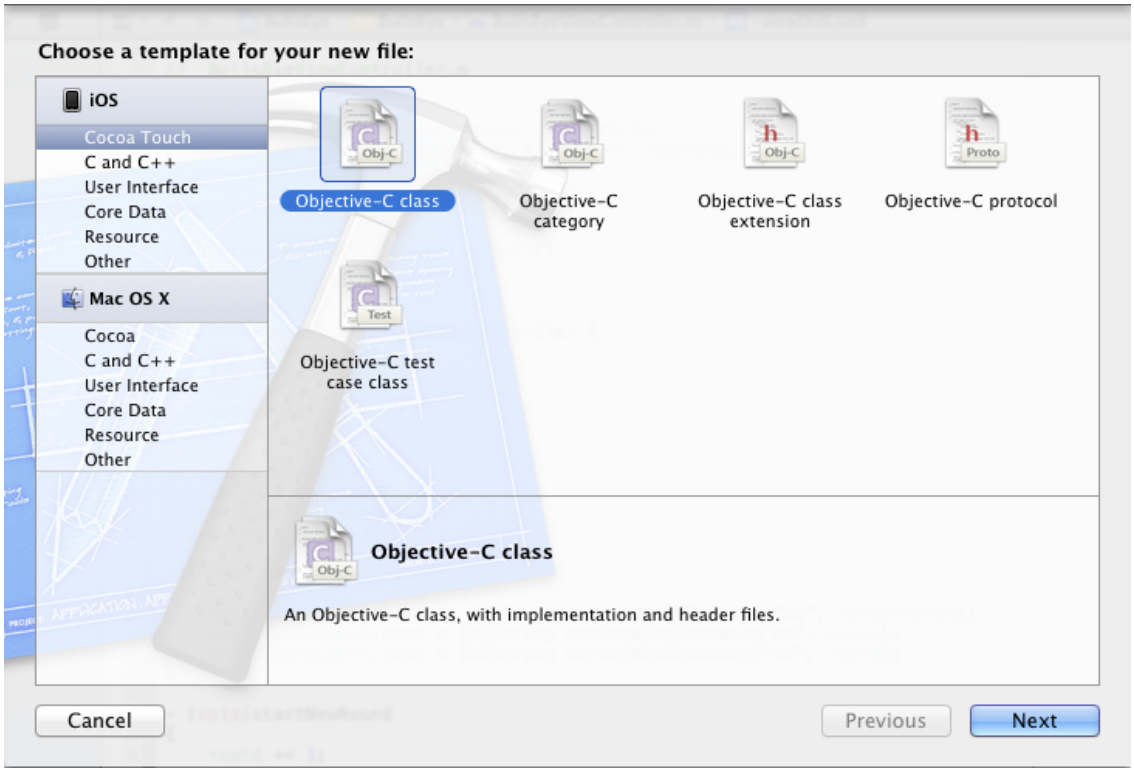
» Go to Xcode's File menu and choose New File… In the window that pops up, choose "UIViewController subclass":

Choose the file template for UIViewController subclass



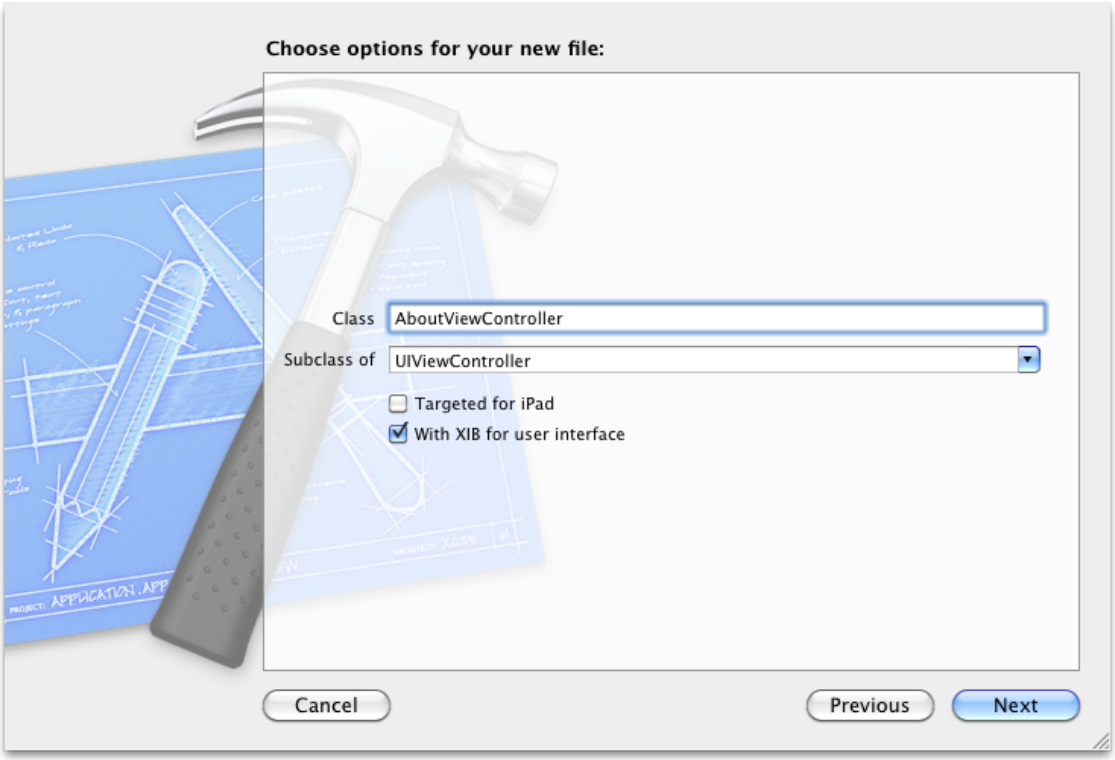
This is where Xcode 4.3 is a bit different. If you're on Xcode 4.3, choose the "Objective-C class" template instead:

Choose the file template for Objective-C class in Xcode 4.3



Click Next. Xcode will give you some options:

The options for the new file

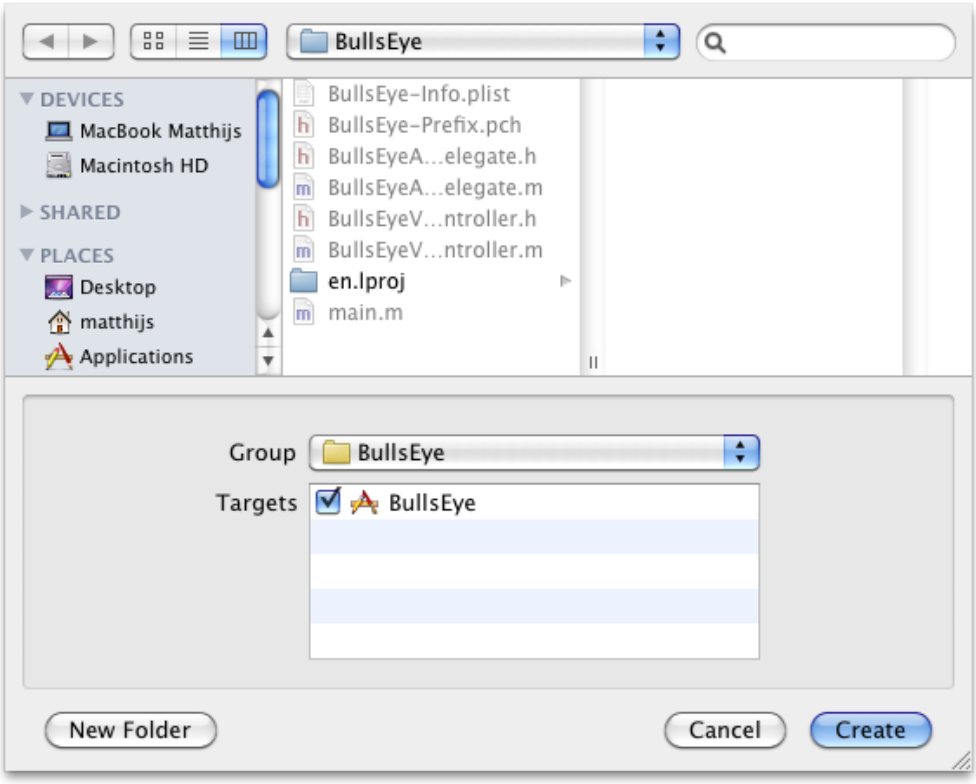


Choose the following:

- Class: `AboutViewController`
- Subclass of: `UIViewController` (in Xcode 4.3 you'll have to type this yourself)
- Targeted for iPad: Leave this box unchecked.
- With XIB for user interface: Check this box.

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

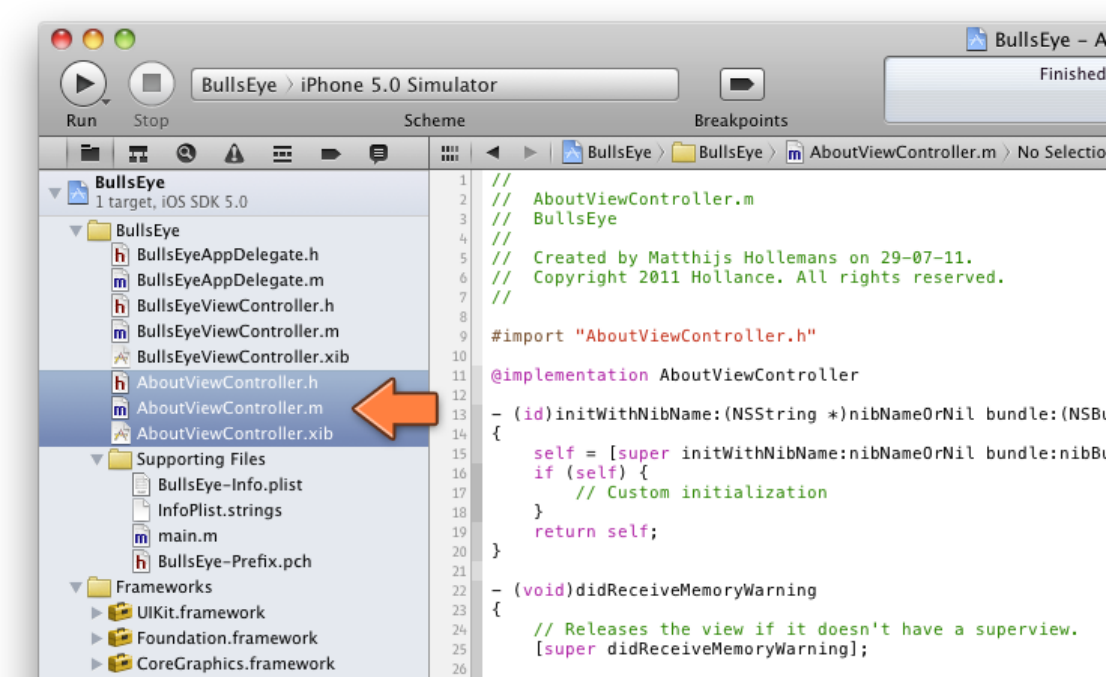
Saving the new file



» Choose the BullsEye folder (this folder should already be selected). Also make sure Group says “BullsEye”. Click Create.

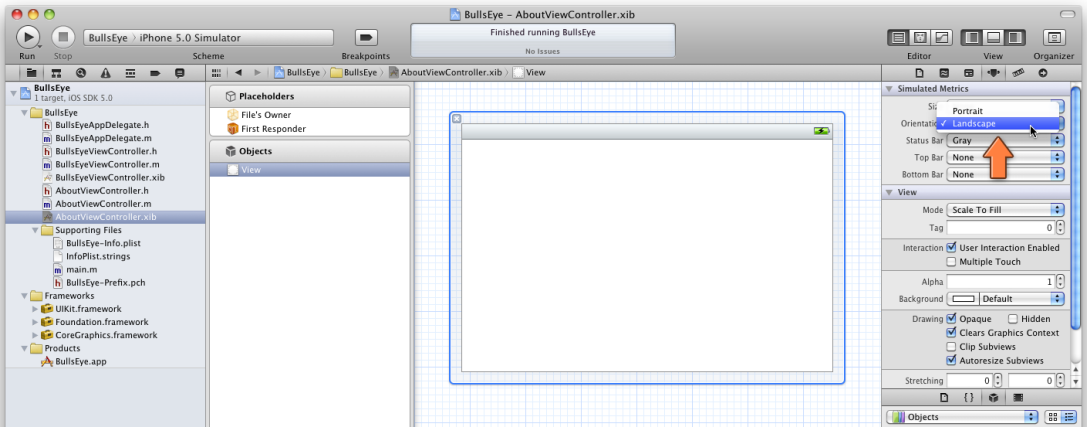
Xcode will make three files and add them to your project. As you might have guessed, the three files are a .xib for the user interface and a .h / .m pair for the source code.

The new files in the Project Navigator



» Let's start with the AboutViewController.xib. Click to open it. Click on the white view to select it and go to the Attributes Inspector. Under Simulated Metrics, set Orientation to Landscape.

Changing the orientation of the view to landscape



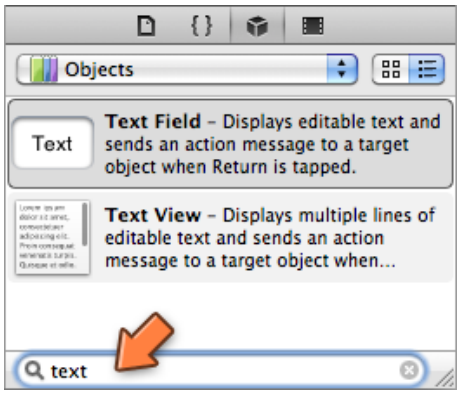
Important: If you're using Xcode 4.5 or better, go into the File Inspector for the nib file and disable the "Use Autolayout" option. It's not such a big deal if you leave Auto Layout enabled for this screen, but you don't really need it and it will make the app incompatible with iOS 5. So it's best to turn this feature off.

» Drag a new Round Rect 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 this component in the Object Library. If you don't feel like scrolling, you can filter the components by typing in the field at the bottom:

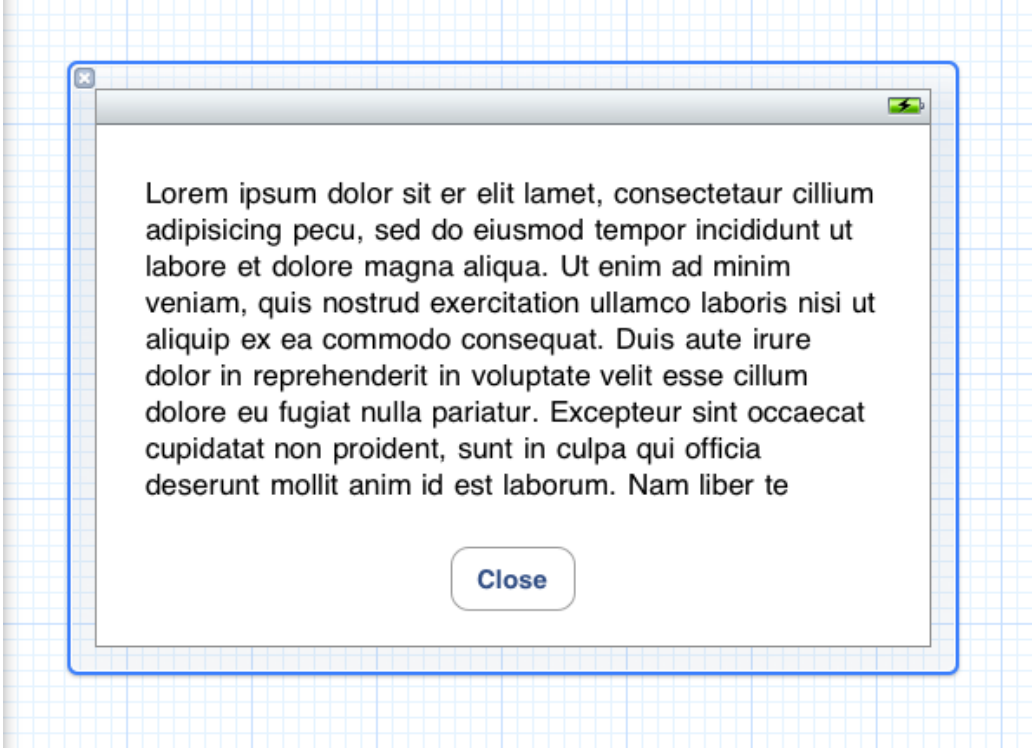
Searching for any text components



Note that there is also a Text Field, which is a single-line text component. We'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 with placeholder text



By default, the Text View contains a whole bunch of fake Latin placeholder text (also known as “Lorem Ipsum”).

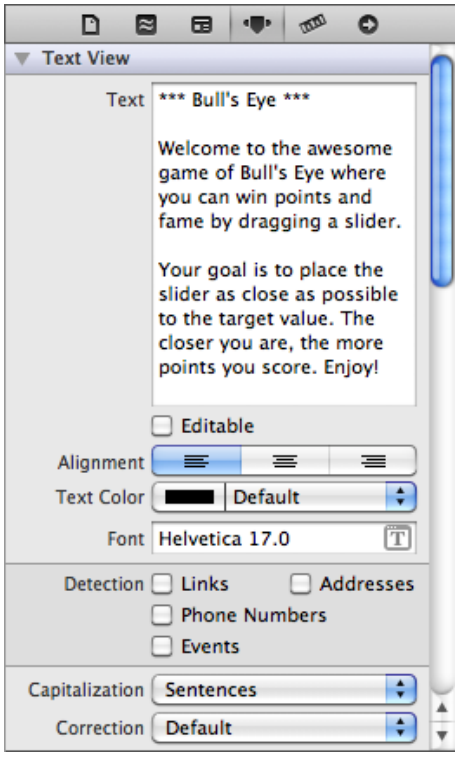
» Double-click on the text view to make its contents editable and copy-paste this new text into it:

```
*** 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 we would like it to be read-only.

The Attributes Inspector for the text view



That’s the nib for now.

Next, we'll add some code to the main screen that will open this new About screen when the user presses the (i) button.

» Go to BullsEyeViewController.h and add the following action:

BullsEyeViewController.h

```
- (IBAction)showInfo;
```

You probably know where I'm headed with this. Indeed, open BullsEyeViewController.xib and connect the Info button to this action. The Info button is a regular `UIButton`, just like the two other buttons we've used. It only looks special.

» Select the Info button, Ctrl-drag to File's Owner, and pick the "showInfo" action.

» In BullsEyeViewController.m, below `startOver`, add:

BullsEyeViewController.m

```
- (IBAction)showInfo
{
  AboutViewController *controller = [[AboutViewController alloc] ↵
      initWithNibName:@"AboutViewController" bundle:nil];
  controller.modalTransitionStyle = UIModalTransitionStyleFlipHorizontal;
  [self presentViewController:controller animated:YES completion:nil];
}
```

There's a bit of new stuff here. `AboutViewController` is the name of the view controller that we just added to the project. Here we create a new `AboutViewController` object and store it in the local variable `controller`:

```
  AboutViewController *controller = [[AboutViewController alloc] ↵
      initWithNibName:@"AboutViewController" bundle:nil];
```

Then we set the new view controller's *transition style*. This is the animation that is used to go from the current screen to the new one. We will make the page flip. This animation is really designed for apps in portrait mode where the screen flips from right to left, but we can also use it in landscape orientation where it flips from bottom to top:

```
  controller.modalTransitionStyle = UIModalTransitionStyleFlipHorizontal;
```

Recognize the `something.somethingElse` notation? That means `modalTransitionStyle` is a property on `AboutViewController`. All view controllers have this property.

Now that we have created the view controller object and configured its properties (well, at least one of them), we can show it on the screen:

```
  [self presentViewController:controller animated:YES completion:nil];
```

The `presentViewController` method opens a new screen on top of the current one. This line tells the `BullsEyeViewController` to open the About screen with an animation. If you would have said `animated:NO`, then there would be no page flip and the new screen would instantly appear. From a user experience perspective, it's often better to show transitions from one screen to another with a subtle animation.

» If you now try to run the app, Xcode will give an error message: "Use of undeclared identifier 'AboutViewController"'. I point this out because a lot of beginning programmers forget the following step. When you see this error, you forgot to tell the compiler about an object that it needs.

» At the very top of BullsEyeViewController.m, add the following line, directly below the existing `#import` line:

BullsEyeViewController.m

```
#import "AboutViewController.h"
```

Without this `#import` statement, `BullsEyeViewController` doesn't know anything about `AboutViewController`. So we have to import the description of `AboutViewController` from AboutViewController.h into `BullsEyeViewController`'s source file.

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

Whoops, that doesn't look quite right. Did you see what happened? The iPhone switched to portrait mode as soon as the About screen became visible.

Our game runs in landscape orientation, so the About screen should be in landscape orientation too. You don't want to force users to turn their devices (or their heads) just so they can read what's on the newly opened screen. Not many users will appreciate it when you force this kind of physical exercise on them!

We did set the view to landscape orientation in Interface Builder, but that is not enough. To fix this, we will also have to edit AboutViewController.m.

Open AboutViewController.m. You'll see that Xcode already put a bunch of methods in here. One of them is `shouldAutorotateToInterfaceOrientation`. Recall that we

also changed this method in BullsEyeViewController.m way back at the beginning of this tutorial.

» Replace `shouldAutorotateToInterfaceOrientation` with:

AboutViewController.m

```
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)↵
    interfaceOrientation
{
  return UIInterfaceOrientationIsLandscape(interfaceOrientation);
}
```

This tells UIKit that `AboutViewController` will only show itself in landscape orientation.

» Run the app and verify that it works.

There is another obvious shortcoming of the `AboutViewController`: pressing the Close button doesn't work yet. Once the user enters the About screen he can never leave. That also doesn't sound like good user interface design to me, so let's hook up the Close button to an action.

This time we 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:

AboutViewController.h

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

» Go into AboutViewController.xib and Ctrl-drag from the button to File's Owner. This should be old hat by now. Connect the button to the "close" action.

Note that File's Owner for this nib is `AboutViewController`. When we edited BullsEyeViewController.xib, File's Owner was the `BullsEyeViewController`. Now that we're editing the nib for the About screen, the owner of this file is the `AboutViewController`. Therefore when you drag from the Close button to File's Owner you will only see the actions that are declared in `AboutViewController`, not those in `BullsEyeViewController`.

» That leaves us with one final step. Add the following in AboutViewController.m, just above the `@end` line at the bottom:

AboutViewController.m

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

This is a common pattern for opening and closing screens. The "presenting" view controller, in this case `BullsEyeViewController`, creates an instance of the new view controller, `AboutViewController`, and makes it visible with a call to the `presentViewController` method.

When the user is done with the new screen, that view controller can close itself by sending the `dismissViewControllerAnimated` message back to the screen that presented it.

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

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

This completes our 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, it 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 us 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.

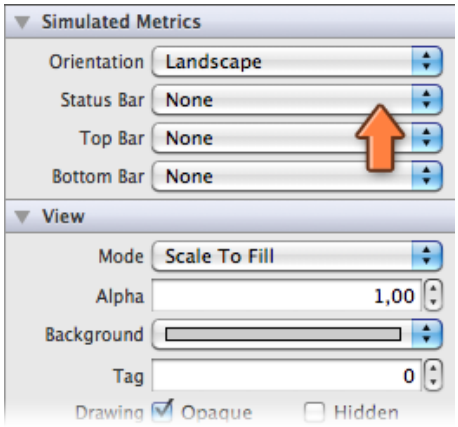
Making it look good

So far, our 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, we will remove the status bar from the nibs.

» Open BullsEyeViewController.xib and select the gray background view, which is the top-level container that holds all the other controls. Go to the Attributes Inspector and under Simulated Metrics set Status Bar to None. This removes the status bar from the view.

Remove the status bar from the view



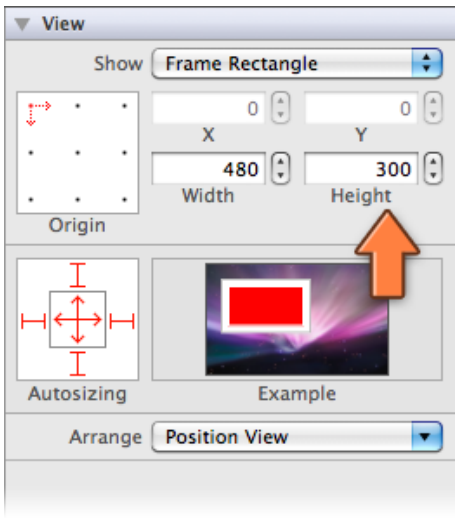
There is a reason why this section is labeled *Simulated* Metrics, as the status bar isn't really part of the view. Interface Builder merely pretends there is a status bar as a visual design aid, so you can see how your screen design looks with the the status bar on top.

There are other common UI elements that you can simulate, such as a navigation bar on top of the view and a tab bar at the bottom. That will give you a good feel for how your screen will look in these situations, plus it properly resizes the view to account for these additional UI elements that surround it.

In landscape orientation, the iPhone's screen is 320 points high. The status bar takes up 20 of these points, which leaves 300 points for the rest of the screen. Now that we have removed the status bar, the view is still only 300 points high, so we'll need to resize it to use the full 320 points that are available to us.

» With the top-level view still selected, open the Size Inspector. Type 320 in the Height field and press Enter. The view will now resize to the proper height.

Changing the size of the view



» Repeat this same procedure for the `AboutViewController`: remove the simulated status bar and resize the view to 320 points vertically.

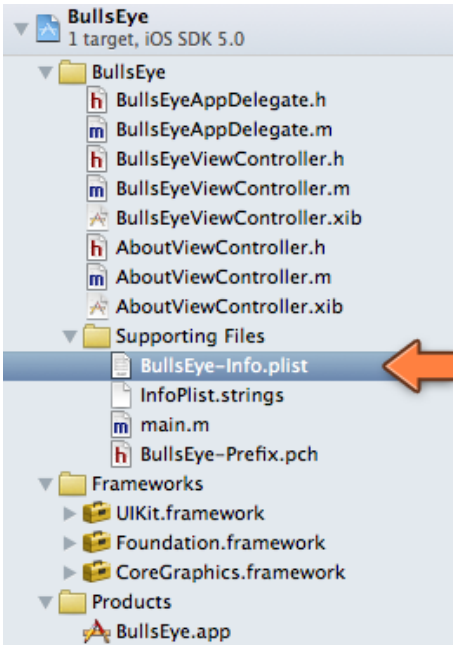
The final step to remove the status bar is to edit our app's Info.plist file. Info.plist is a configuration file inside the application bundle that tells iOS how the app will behave. It also describes certain characteristics of the app that don't really fit anywhere else, such as its version number.

We've already made changes to Info.plist, be it indirectly, when we removed Portrait as one of the Supported Device Orientations way back in the beginning. This told iOS that the app should always be launched in landscape orientation.

There is no button in Xcode that we can simply click to remove the status bar but fortunately editing the Info.plist file is very straightforward.

» Go to the Project Navigator. You will find a file named BullsEye-Info.plist under the group "Supporting Files". Click BullsEye-Info.plist to select it and Xcode's main edit pane will show you its contents.

The BullsEye-Info.plist file

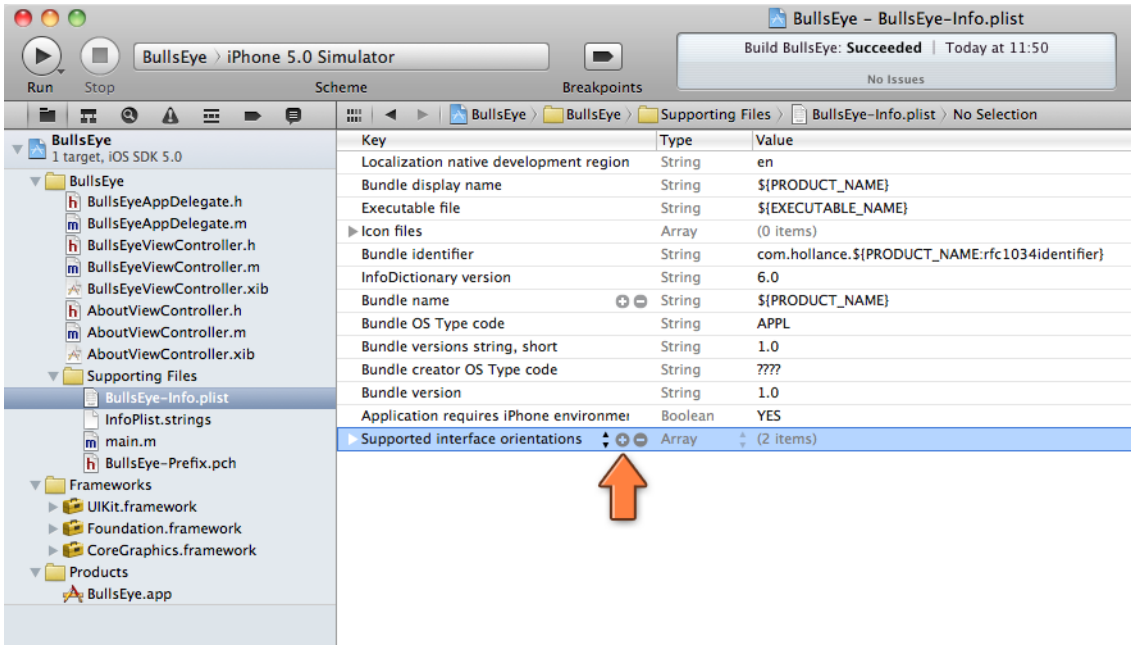


The Info.plist file is just a list of configuration options and their values. Most of these may not make sense to you, but that's OK (they don't always make sense to me either).

We are going to add a setting for the status bar.

» Click on the last row to select it. In the screenshot below that row is named "Supported interface orientations" but it doesn't really matter what it says:

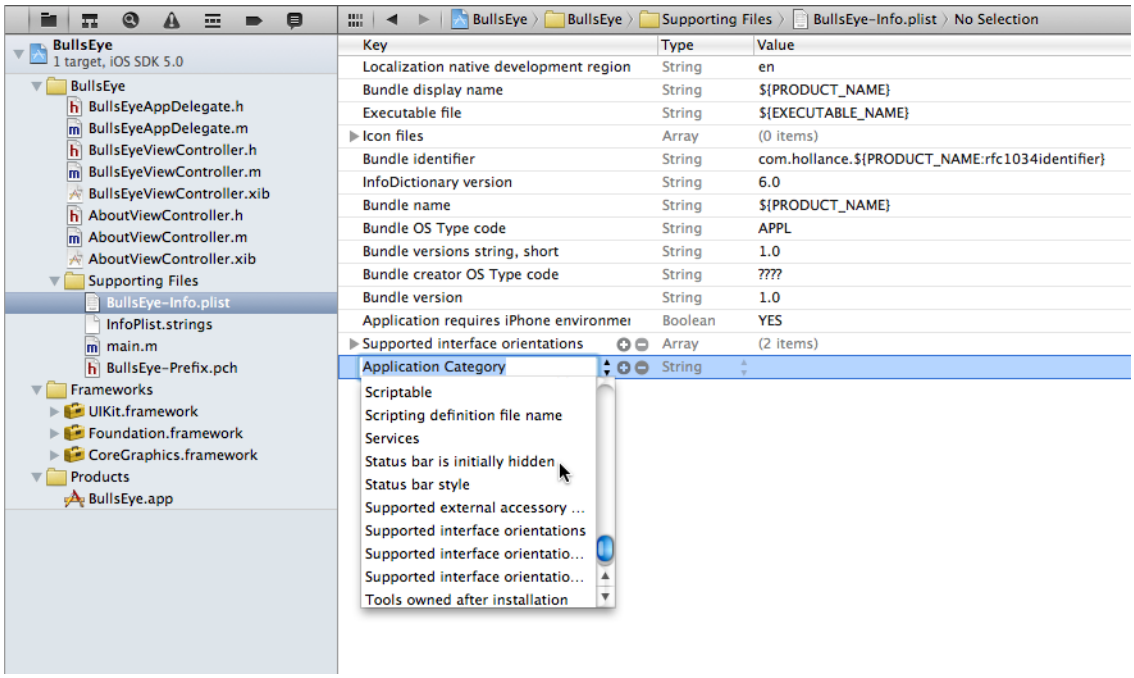
Adding a row to BullsEye-Info.plist



Two small icons will appear, a plus sign and a minus sign.

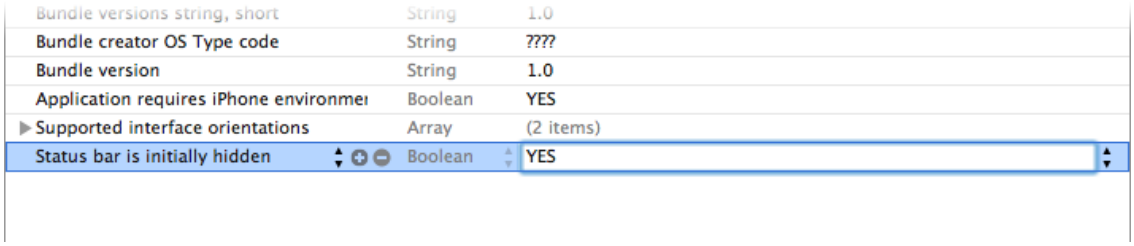
» Click the plus. This will add a new row. A popup also appears that lets you choose the name for the setting. We are looking for the one that says “Status bar is initially hidden”. Scroll down until you see it and then click to select it:

Adding a row for a new setting



» Click in the Value column for this new row. Change it to `YES` (in all capitals). If Xcode won’t let you edit this value, try double-clicking it first.

Changing the value of the setting to YES

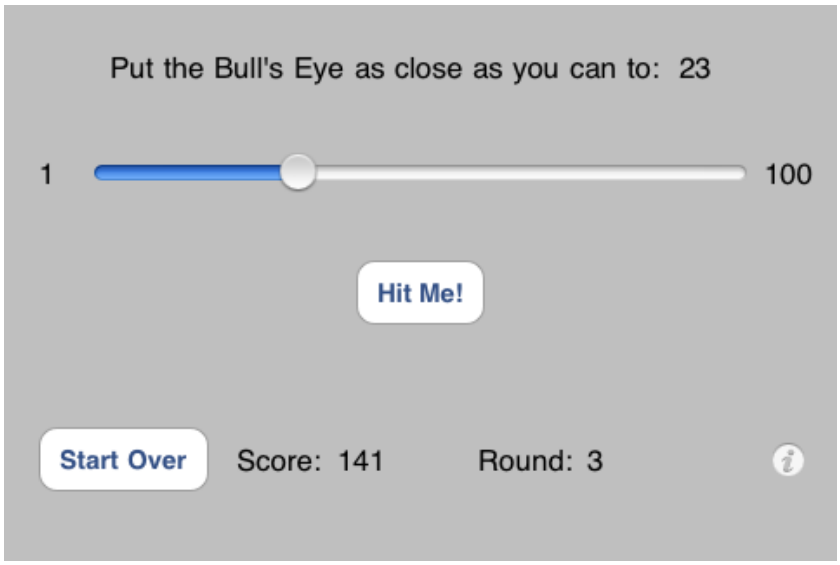


That’s it. Run the app and you’ll see that the status bar is history.

Spicing up the graphics

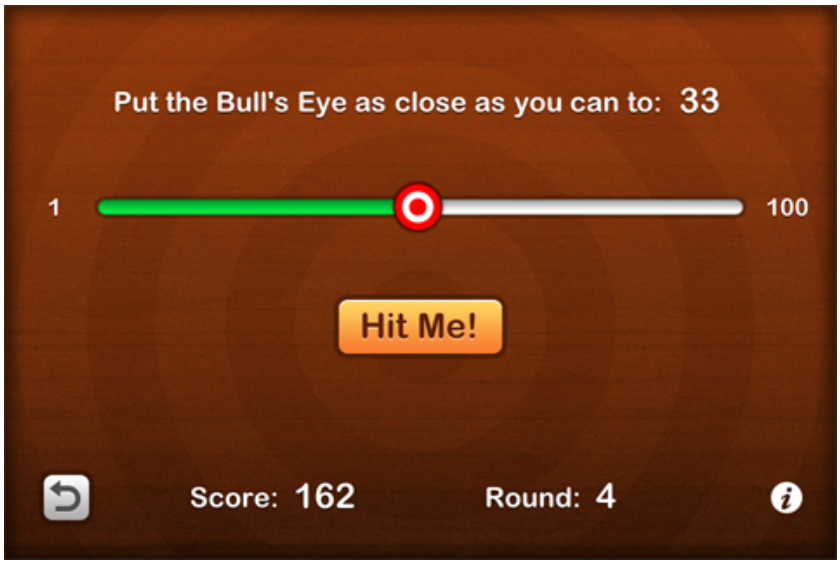
Getting rid of the status bar is only the first step. We want to go from this:

Boring!



to this:

Cool!



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

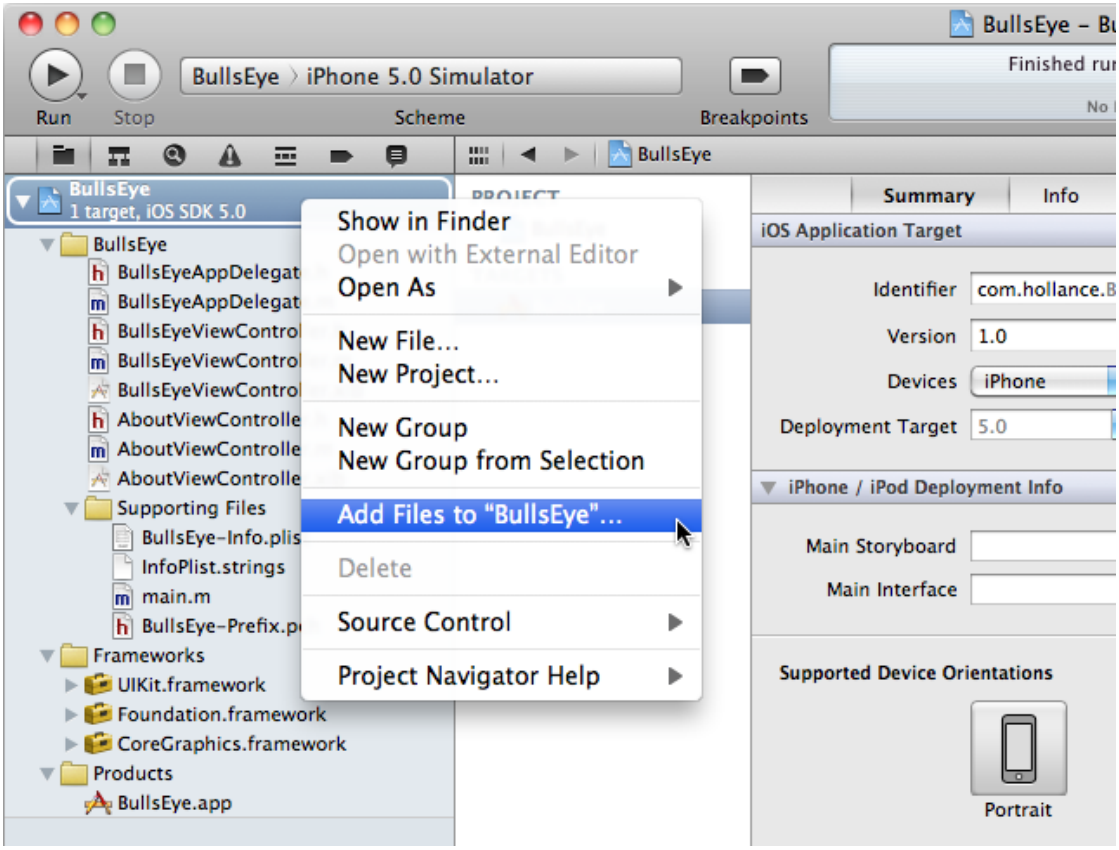
We can put an image in the background, on the buttons, and even on the slider, to customize their appearance. Images should be in PNG format. If you are artistically challenged, then don't worry, I have provided a set of images for you. But if you do have mad Photoshop skillz, then by all means go ahead and design your own.

The Resources folder that comes with this tutorial contains a subfolder named Images.

We will import these images into the BullsEye project.

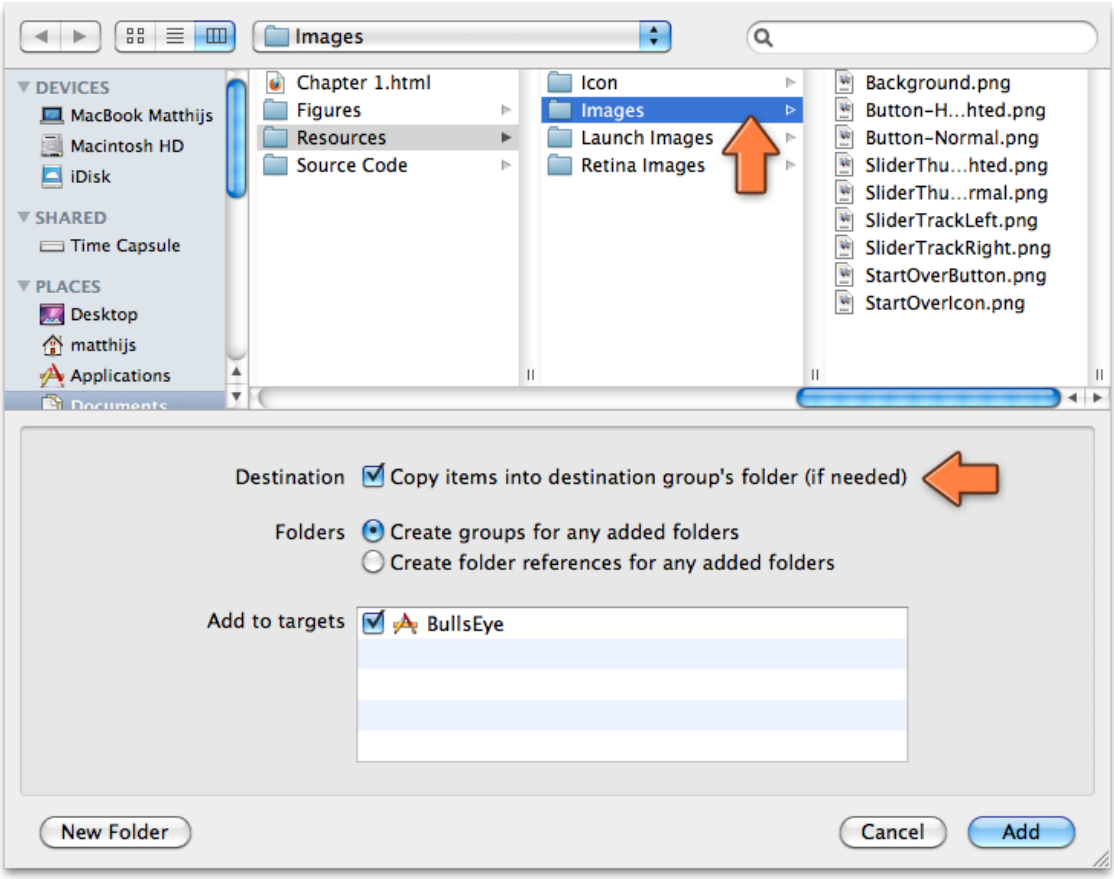
» Right-click the project name in the Project Navigator and choose Add Files to “Bulls-Eye”.

Adding files to the project



Xcode will ask you for the files you wish to add:

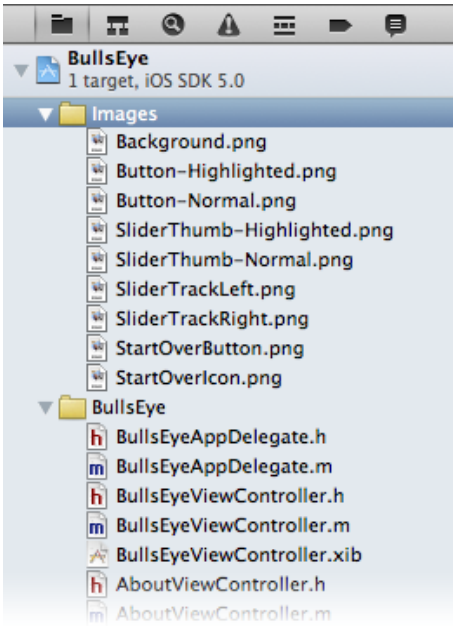
Adding files from the Images folder



» Select the Images folder itself (not its contents). Make sure you check “Copy items into destination’s group folder (if needed)” and “Create groups for any added folders”.

This will copy the entire Images folder into the project directory. There is now a new group named Images in the Project Navigator:

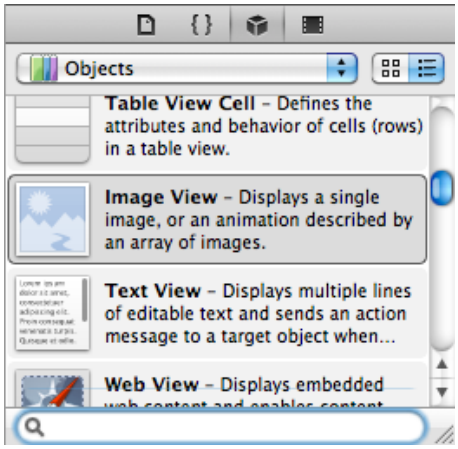
The image files have been added to a group of their own



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

» Open BullsEyeViewController.xib. 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

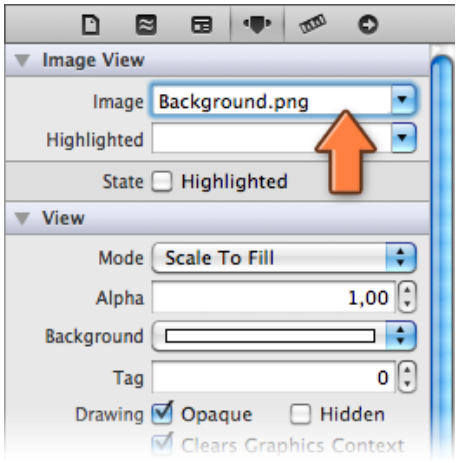


» Drag the image view on top of our existing user interface. It doesn't really matter where you put it, as long as it's inside the view controller's screen.

» With the image view still selected, go to the Size 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 blue arrow and choose Background.png from the list. This will put the image named Background.png into the image view.

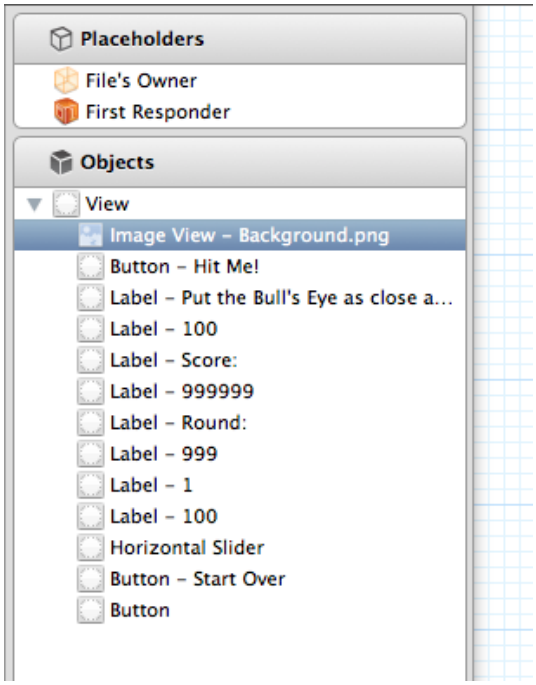
Setting the background image on the Image View



There is only one problem: the image now obscures all the other controls. There is an easy fix for that, we 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 Arrangement → 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 Objects pane and drag it to the top, that will 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 `AboutViewController`, again with the Background.png file.

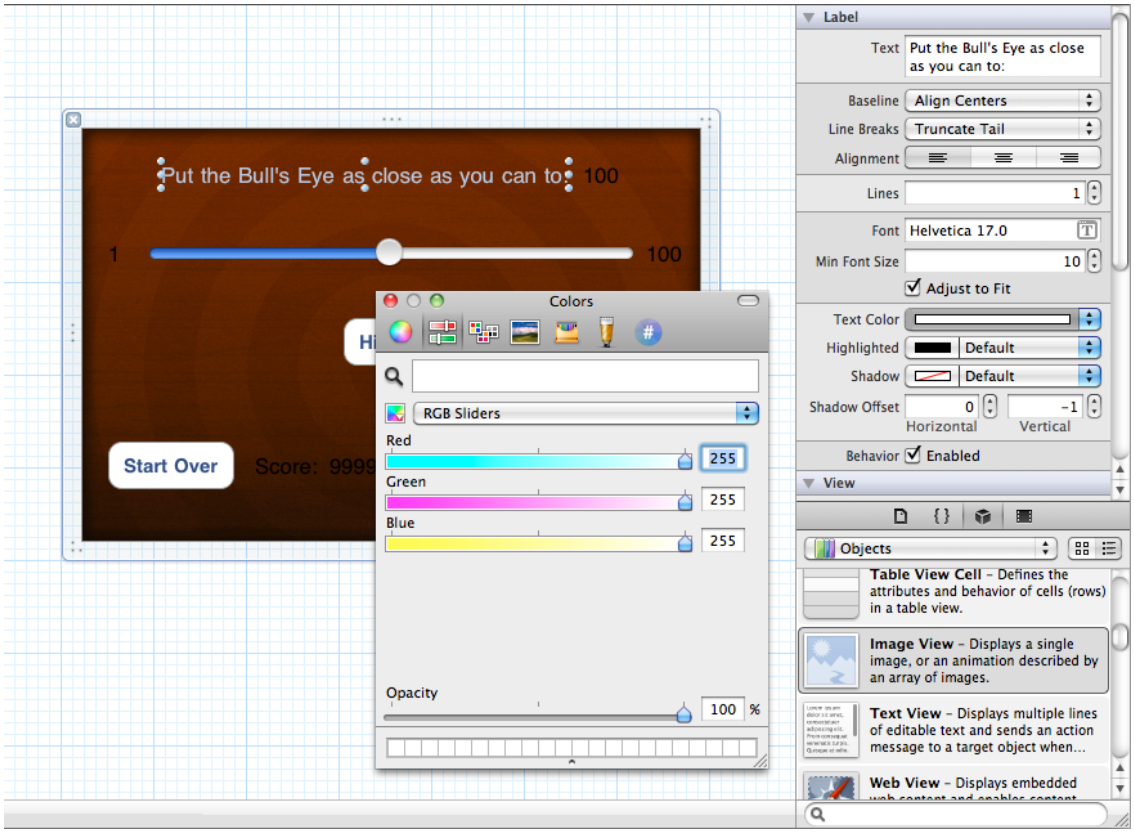
That takes care of the background. Run the app and marvel at our new graphics.

Changing the labels

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

» Select the label at the top, open the Attributes Inspector and click on the Text Color item.

Setting the text color on the label



This opens the Color Picker. There are several types of ways to select colors here. (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%.

» With the Color Picker still open, click on the Shadow item from the Attributes Inspector. This lets us 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. Choose a pure black color that is half transparent, Red: 0, Green: 0, Blue: 0, Opacity: 50%.

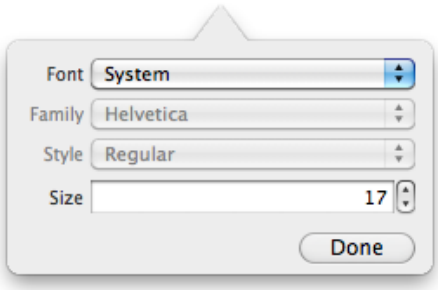
Note: Sometimes when you change the Text Color or Shadow Color properties, the background color of the view also changes. I think this is a bug in Xcode. Put it back to Clear Color when that happens.

» Change the Shadow Offset to Horizontal: 0, Vertical: 1. This puts the shadow under the label.

The shadow we’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.

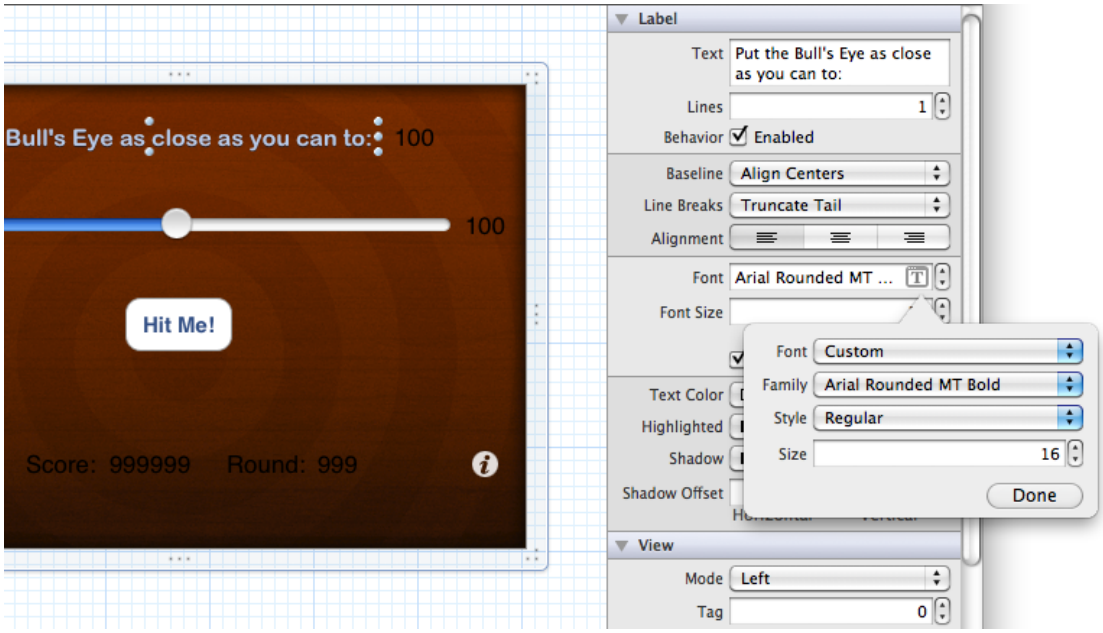
» Click on the [T] icon of the Font attribute. This opens the Font Picker. By default the System font is selected. That uses whatever is the standard font for the user's device, which currently is either Helvetica or Helvetica Neue.

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". Uncheck this.

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 ours. Instead, we'll change the size of the label to fit our text rather than the other way around.

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

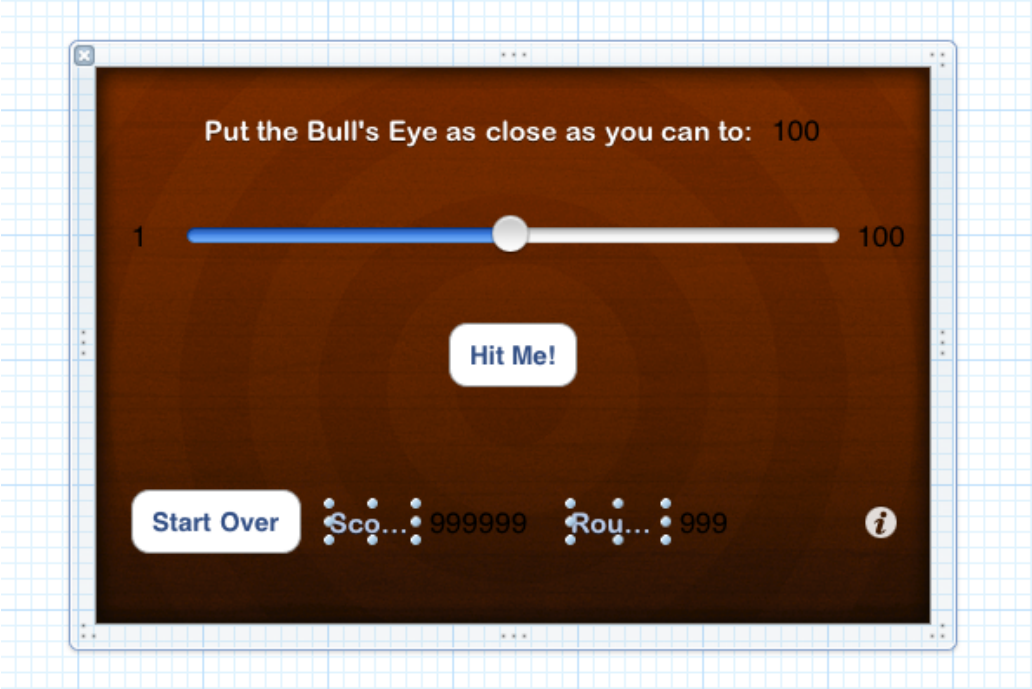
We don’t have to set these properties for the other labels one by one, that would be a big chore. We can speed up the process by selecting multiple labels and then applying these changes to that entire selection.

» Click on the Score: label to select it. Hold Cmd and click on the Round: label. Now both labels will be selected. Repeat what we did above for these labels:

- Set Text Color to pure white, 100% opaque.
- Set Shadow Color to pure black, 50% opaque.
- Set Shadow Offset to 0 horizontal, 1 vertical.
- Set Font to Arial Rounded MT Bold, size 16.
- Uncheck Autoshrink.

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

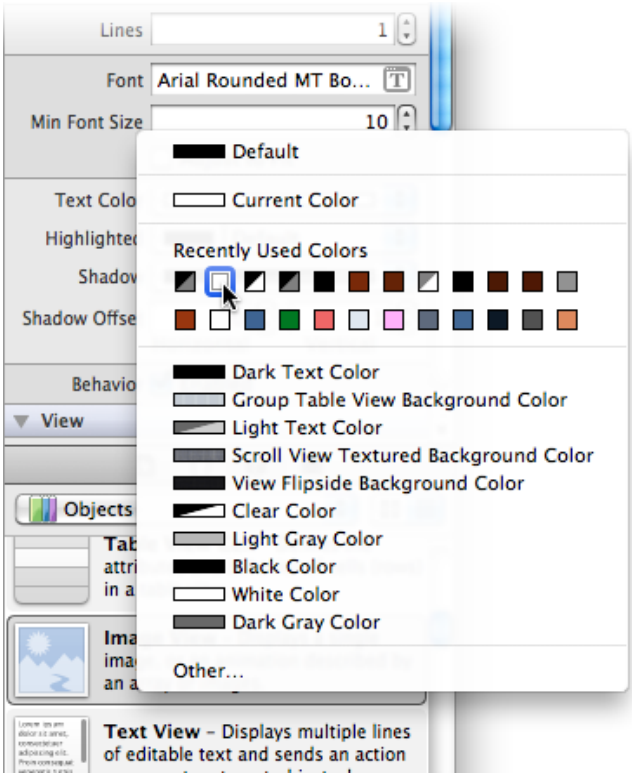
The font is too large to fit all the text in the Score and Round labels



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 (Cmd =). 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 blue arrow and the menu will pop up:

Quick access to recently used colors and several handy presets



Exercise: We still have a few labels to go. Repeat what we 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 we've changed the sizes of some of the labels, your carefully constructed layout may have been messed up a bit. You can rearrange the labels by hand, but Xcode also has some handy layout tools that you can use.

» For example, make a selection of the two labels at the top and choose Editor → Alignment → Align Horizontal Center in Container. This will properly center these two labels in the view.

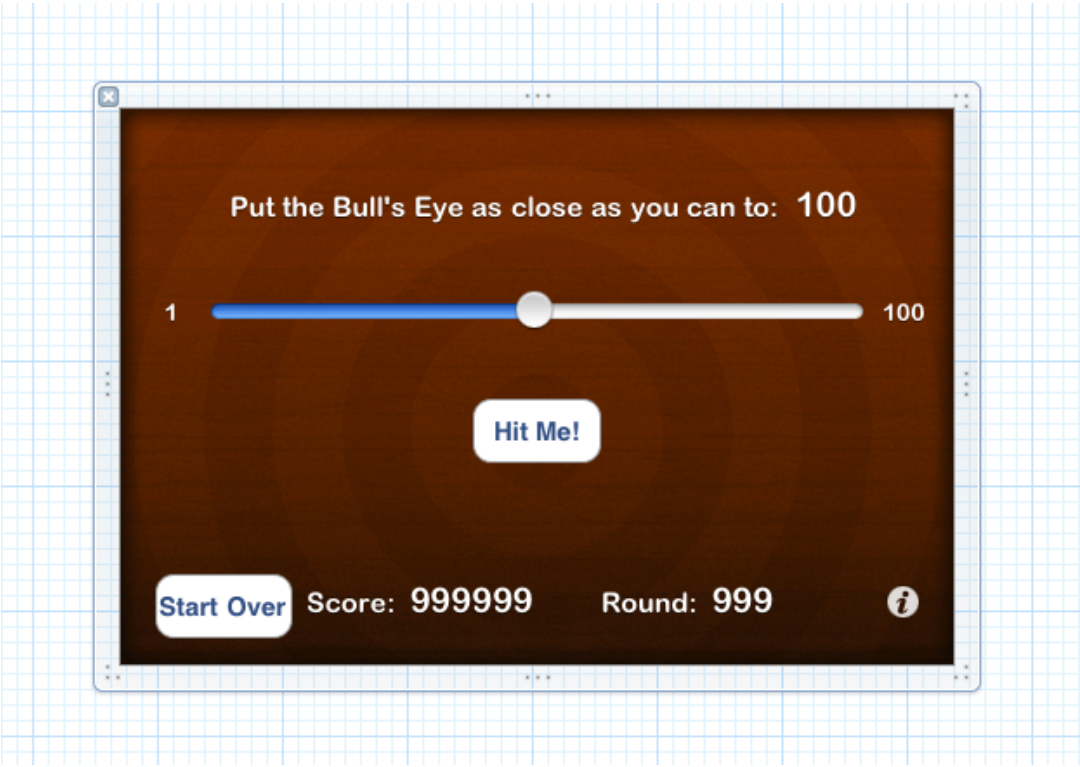
Fonts

Interface Builder shows the fonts that are available on your Mac, not the ones that are actually on your iPhone. The iPhone comes with a fair number of fonts but fewer than your Mac. If you use a font that is not on the iPhone, then your app won't look like what you expect when you run it on the device.

You can see a list of available iPhone fonts at iosfonts.com [http://iosfonts.com] — note that you will have to view this site on your actual iPhone. There are also a bunch of free apps on the App Store that let you browse the fonts on your device. If you find the standard fonts lacking, you can bundle your own TTF font files with the app.

At this point, my screen looks like this:

What the nib 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 like something 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. I centered the position of the button on the inner circle of the background image.

» Go back to the Attributes Inspector. Set Type to Custom. The shape of the button now disappears. Press the arrow on the Background field and choose Button-Normal.png.

» 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.

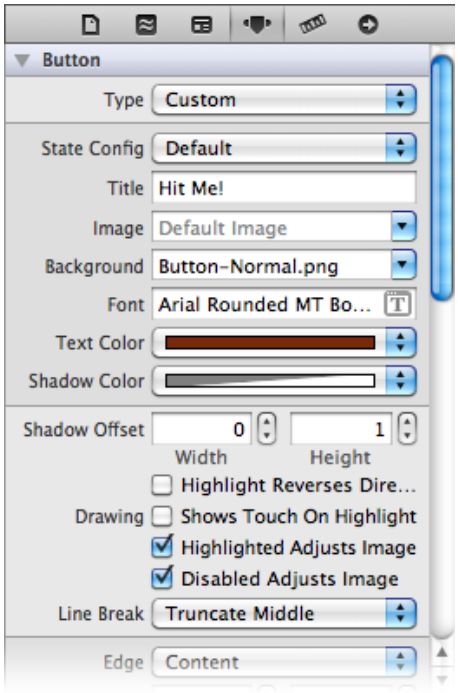
» Set the shadow color to pure white, 50% opacity and the shadow offset as before to 0 horizontal and 1 vertical.

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 for its default state



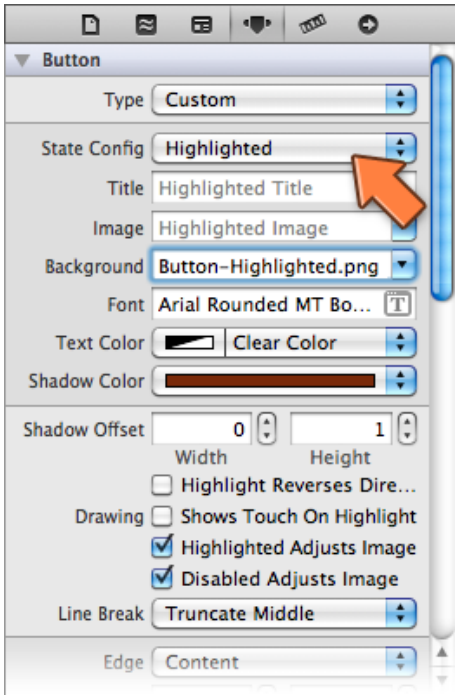
Buttons can have more than one state. When you tap a button and hold it down, it should appear "pressed down" to let you know that the button will be activated when you lift your finger. This is known as the "highlighted" state and it is an important visual clue to the user. The plain Round Rect Button turns the background blue when you tap it but now that we've set the button's type to Custom, we should handle this ourselves.

» Click the State Config setting and pick Highlighted from the menu. In the Background field, select "Button-Highlighted.png". (Note: if we don't specify a different image for the highlighted state, UIKit will automatically darken the button to indicate this new state.)

» By default, the highlighted Text Color is white. Change that to Clear Color (i.e. fully transparent). Change the Shadow Color to the dark brown color we used earlier (simply pick it from the Recently Used Colors menu). 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 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.

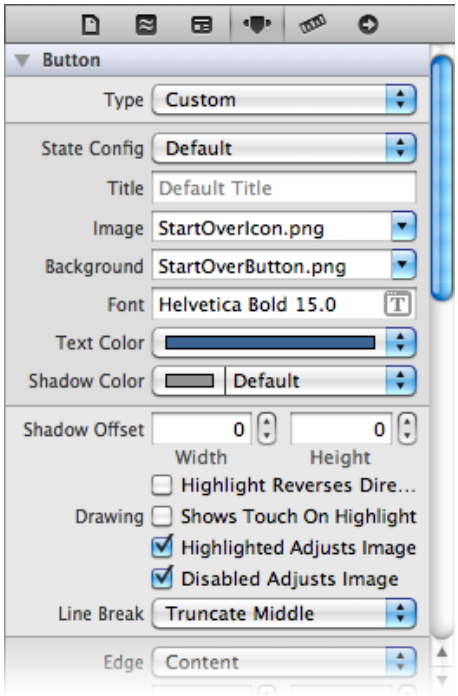
That’s it for the Hit Me button. Styling the Start Over button is very similar, except that we will replace its title text by an icon.

» Select the Start Over button and change the following attributes:

- Set Width and Height to 32.
- Set Type to Custom.
- Clear out the Title.
- For Image choose StartOverIcon.png
- For Background choose StartOverButton.png

We won’t set a highlighted state on this button but let UIKit take care of this automatically.

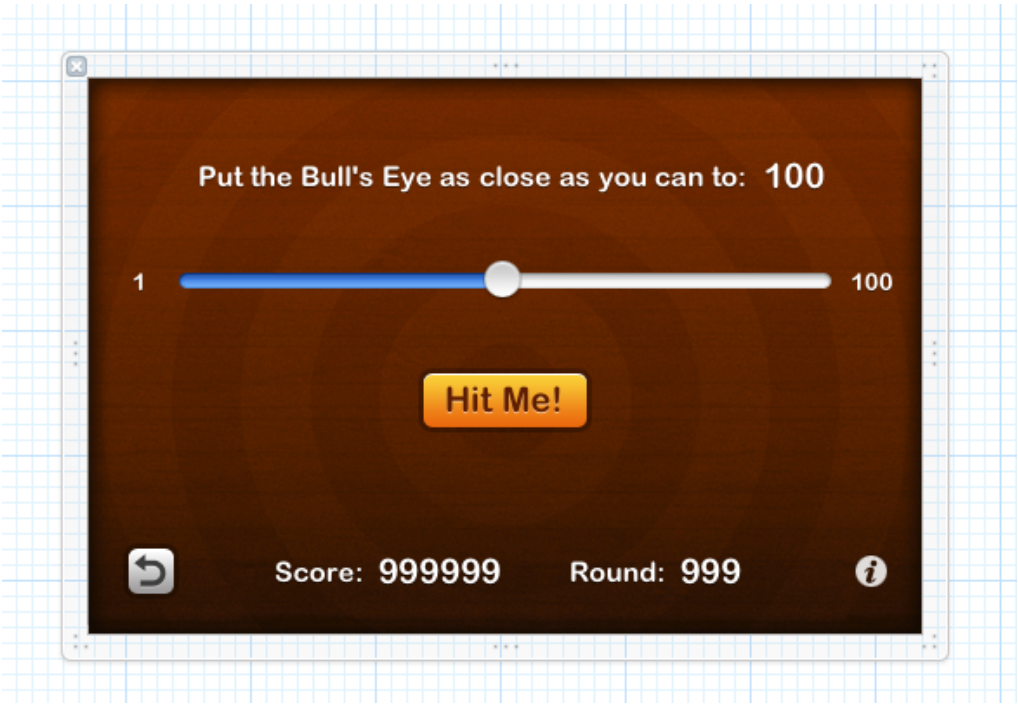
The styling of the Start over button



We don’t have to do anything on the Info button as it looks good already.

The user interface is almost done. Only the slider is left to do:

Almost done!



The slider

Unfortunately, we can only customize the slider a little bit in Interface Builder. For the more advanced customization that we want to do, putting our own images on the thumb and the track, we have to resort to writing source code.

Note that everything we have done so far in Interface Builder, we 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 we have no choice.

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

BullsEyeViewController.m

```
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"] ↵
    stretchableImageWithLeftCapWidth:14 topCapHeight:0];
[self.slider setMinimumTrackImage:trackLeftImage forState:↵
    UIControlStateNormal];

UIImage *trackRightImage = [[UIImage imageNamed:@"SliderTrackRight"] ↵
    stretchableImageWithLeftCapWidth:14 topCapHeight:0];
[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, unpressed 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).

Note that the images are named “SliderThumb-Normal.png” and so on, but that we leave out the .png extension when we create the `UIImage` objects. That’s the preferred way of specifying image filenames for iOS 4 and above and it has everything to do with supporting the iPhone 4’s Retina display. More about that in a second. However, it would also have worked if we did specify the .png extension.

» Run the app. You have to admit it looks pretty good now!

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. ■

» Now select the text view and press the Delete key on your keyboard. Yep, we'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, we will make it display an HTML page from our application bundle, so it won't actually have to go onto the web and download anything.

» Go to the Project Navigator and choose Add Files. Add the BullsEye.html file from the Resources folder to the project. This is an HTML5 document that contains the gameplay instructions.

» In AboutViewController.h, add an outlet property for the web view:

AboutViewController.h

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

» In the nib file, connect the `UIWebView` element to this new outlet. The easiest way to do this is to Ctrl-click on File's Owner and drag to the Web View. (If you do it the other way around, from the Web View to File's Owner, then you'll connect the wrong thing and the web view will stay empty when you run the app.)

» In AboutViewController.m, synthesize the property by adding this line just below the `@implementation` line:

AboutViewController.m

```
@synthesize webView;
```

» Finally, in the `viewDidLoad` method, add:

AboutViewController.m

```
- (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 we find the BullsEye.html file in our application bundle, then we load it into an `NSData` object, and finally we ask 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

Retina graphics

If you obtained your iPhone recently you'll probably have an iPhone 4S. 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 begins, that's how miniscule they are. Compared to the iPhone 4's Retina display, the older iPhone models look very "blocky".

If you were to run our app on an iPhone 4 or 4S, the UI would look a bit blocky as well because all the images get scaled up. If you've ever worked with bitmap graphics before,

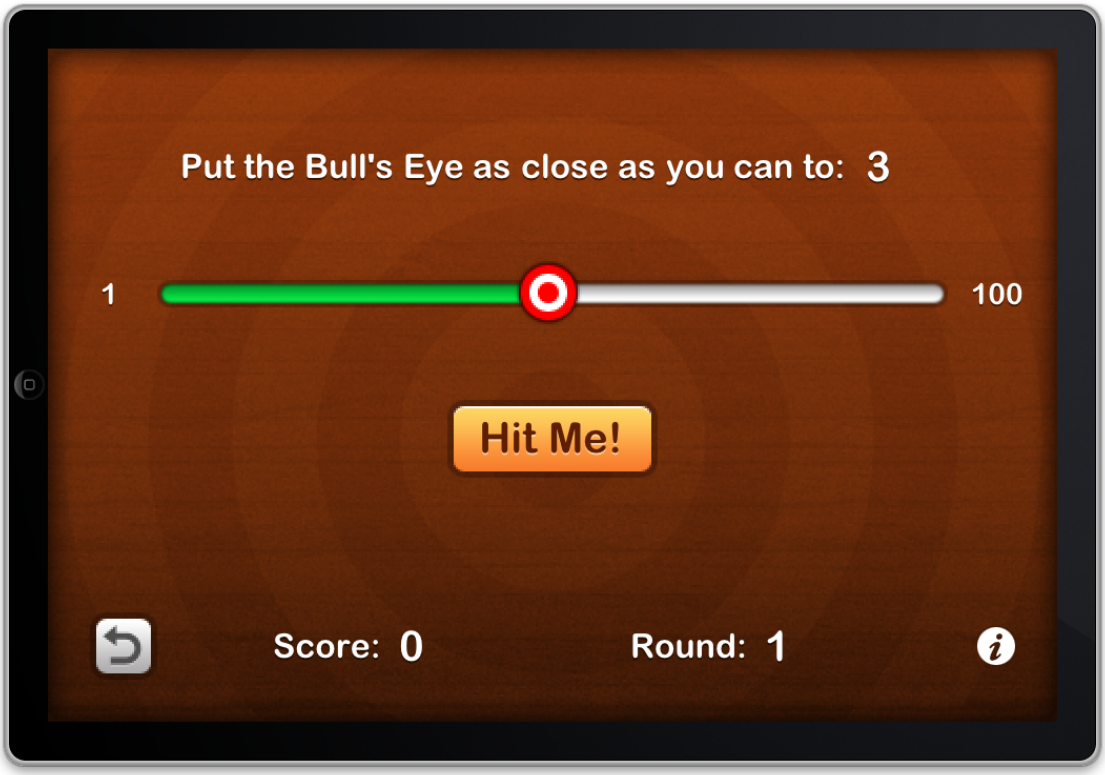
you know that scaling down is usually okay but scaling up will make the pictures look much worse.

The Simulator can actually run in Retina mode, so you can see for yourself. First, press the Stop button to stop the app. Switching Simulator modes while the app is running will make it crash, so it's better to terminate the app first.

» Click on the iOS Simulator to activate its window. From the Hardware menu (in the menubar at the top of the screen), choose Device → iPhone (Retina). The Simulator's window will now double in size and almost completely fill your screen. Yup, that's how many pixels the iPhone 4 has!

If you run the app now you'll see that the text labels are very sharp. iOS will automatically draw these at the higher resolution of the Retina display. The images look a little blocky, though. That is because they have been scaled up by doubling their pixels. Click on the image below to zoom in so you can see it clearly, especially on the slider thumb and the buttons.

Blocky images on the Retina display



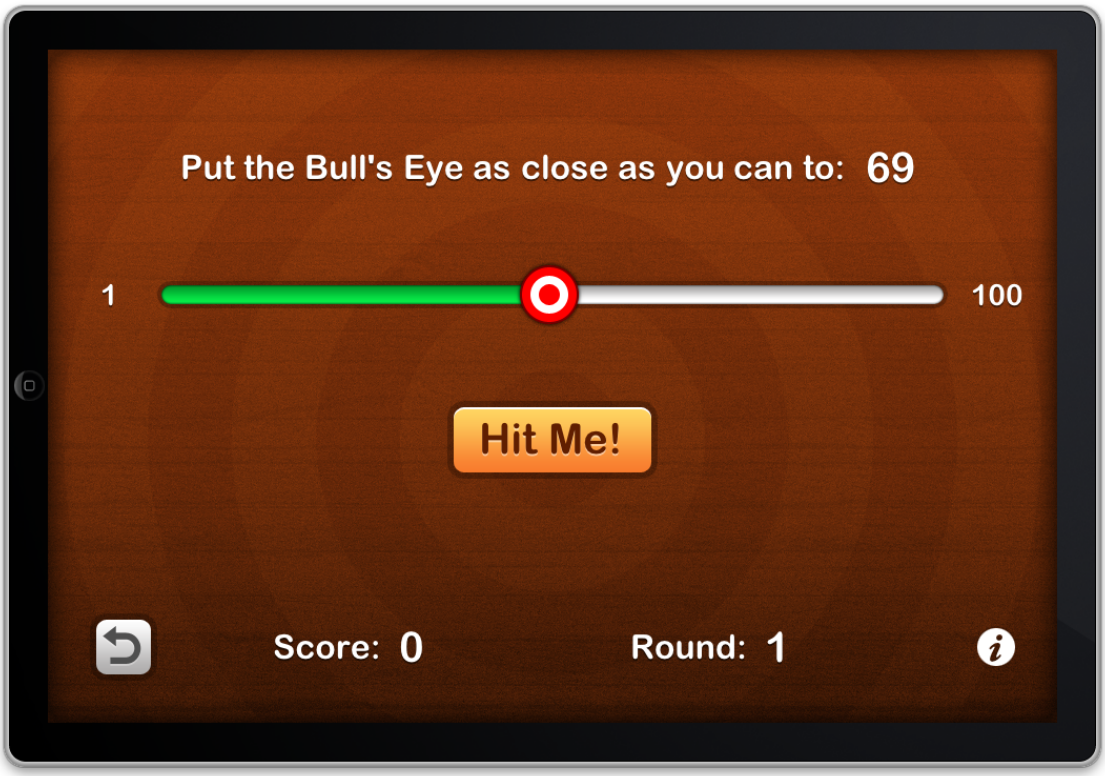
We can do better than that. To make the app look good on Retina, you have to provide a second set of images that are twice as big as the regular images.

» Go to the Project Navigator and right-click to choose “Add Files to BullsEye”. Select the Retina Images folder from this tutorial’s Resources folder. Make sure “Copy items into destination’s group” is selected. This will add a bunch of new images to the project.

Note that for each image that we already had there is now one named @2x.png. This is a special naming convention for Retina devices. If your app runs on a Retina device, it will first attempt to load the @2x version of the image if that exists. If not, then the low-resolution version is used.

» Run the app again. Voila, instant Retina graphics. That’s all it takes to make your app look good on the newer devices. All you have to do is provide images that are twice as big and give them an @2x.png extension.

High-resolution images on the Retina display



Note: Even though *you* may only have an iPhone 4 to develop on, you still need to provide graphics for the older models. That is probably why the Simulator starts out in the low-resolution mode, so you don’t forget to include low-resolution graphics as well!

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.

We 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:

BullsEyeViewController.m

```
- (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` block is new. I'm not going to go into details here. Suffice to say we're setting up an animation that crossfades from what is currently on the screen to the changes we're making in `startNewGame` (reset the slider to center position) and `updateLabels` (reset the values of the labels).

» At the very top of the file, add the line:

BullsEyeViewController.m

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

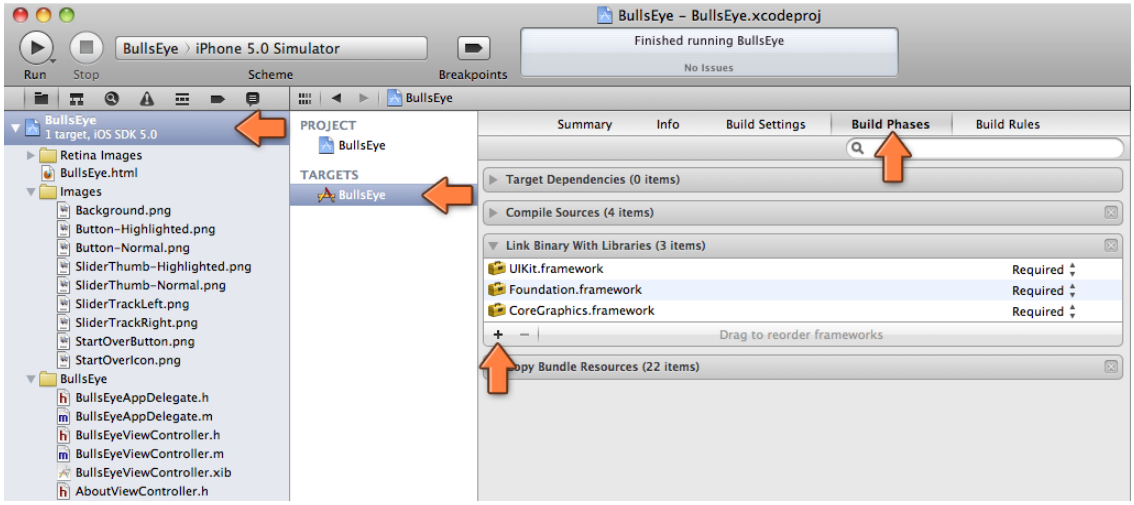
Core Animation is part of the QuartzCore framework. So far we've been using the UIKit and Foundation frameworks, which are a standard part of your projects. QuartzCore, however, must be added separately.

If you try to run the app now, Xcode will give several error messages, such as "Apple Mach-O Linker Error: _OBJC_CLASS_$_CATransition, referenced from: …" and so on. Whenever you see errors like these, you forgot to link in the proper framework.

» In the Project Navigator, click on the project name to open the project settings editor. Then select the BullsEye target and choose the Build Phases tab. Expand the Link Binary

With Libraries section and click the + sign at the bottom:

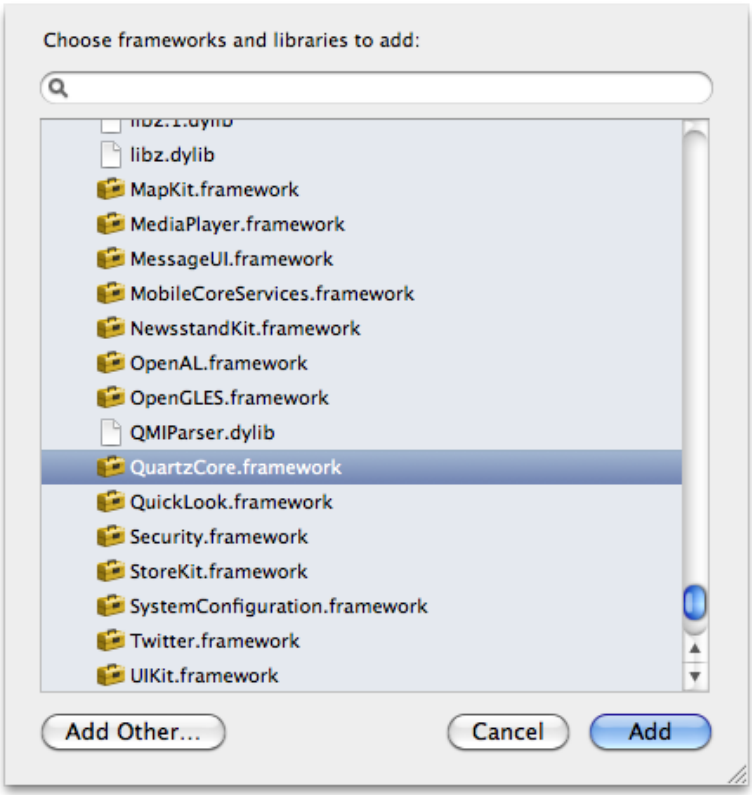
The Link Binary With Libraries section in the Build Phases tab



This shows a list that lets you select one or more frameworks to add.

» Scroll to the bottom and select QuartzCore.framework. Click Add to complete.

Adding the QuartzCore framework



Now all the functionality from the QuartzCore framework is available to our app.

» 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 icon

We'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!

» Copy the Icon folder from this tutorial's Resources into your project (the same way we've been adding files before).

This adds nine images to the project:

Icon.png	The icon for iPhone 3	57x57 pixels
Icon@2x.png	The icon for iPhone 4 (Retina)	114x114 pixels
Icon-72.png	The icon for iPad	72x72 pixels
Icon-72@2x.png	The icon for Retina iPad	144x144 pixels
Icon-Small.png	The icon for Settings and Spotlight on iPhone 3, and Settings on iPad	29x29 pixels
Icon-Small@2x.png	The icon for Settings and Spotlight on iPhone 4 (Retina), and Settings on Retina iPad	58x58 pixels
Icon-Small-50.png	The icon for Spotlight on iPad	50x50 pixels
Icon-Small-50@2x.png	The icon for Spotlight on Retina iPad	100x100 pixels
iTunesArtwork	The icon for iTunes	1024x1024 pixels

If you now run the app and then 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).

iTunes Artwork

Note that "iTunesArtwork" does not contain the .png extension. This file isn't strictly necessary and it is only used for Ad Hoc distribution. When you submit your app to the App Store, you will have to upload this image file separately. Personally, I always like to include the iTunesArtwork file in my application bundle but if you're looking to shave some bytes off your app's file size, you can leave it out.

We should add the icons to the Info.plist as well.

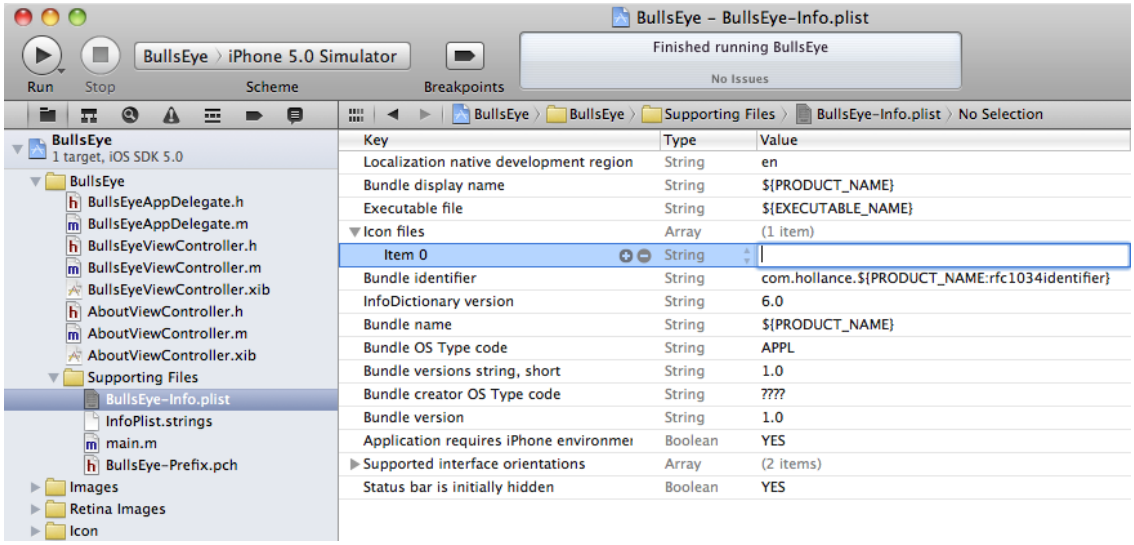
» Click on BullsEye-Info.plist to open it.

There should already be an "Icon files" entry in this file. If there isn't, then you can add it by right-clicking and selecting the "Add Row" option. Be sure to add an entry for "Icon files", not "Icon file".

The "Icon files" row is of type Array, which means that it contains not a single item but a list of items.

» Click the triangle icon in front of the row to expand it. Press the + button to add a new row. This should create a new row named "Item 0" inside the Icon files section.

The Icon files entry in BullsEye-Info.plist



» Enter "Icon.png", without the quotes, and press Enter.

» Click + to add a new row. Give it the value "Icon@2x.png". Repeat this process six more times for Icon-72.png, Icon-72@2x.png, Icon-Small.png, Icon-Small@2x.png, Icon-Small-50.png, and Icon-Small-50@2x.png respectively. (iTunesArtwork is not added

to this list.)

When you're done, the plist should look like this:

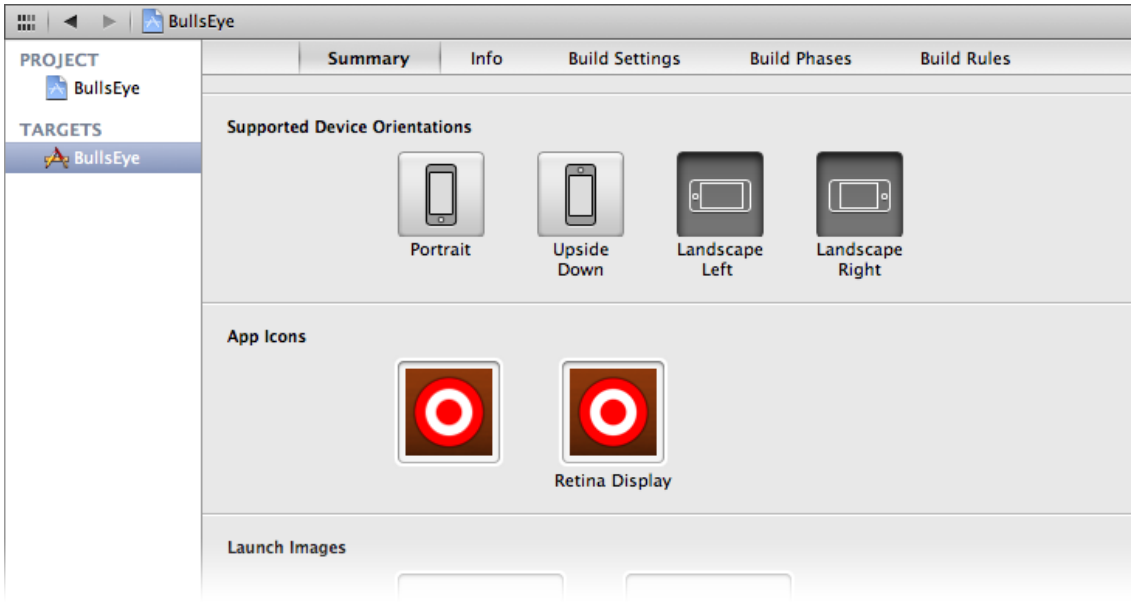
The icons in BullsEye-Info.plist

Icon files	Array	(8 items)
Item 0	String	Icon.png
Item 1	String	Icon@2x.png
Item 2	String	Icon-72.png
Item 3	String	Icon-72@2x.png
Item 4	String	Icon-Small.png
Item 5	String	Icon-Small@2x.png
Item 6	String	Icon-Small-50.png
Item 7	String	Icon-Small-50@2x.png

» Save the plist (press Cmd+S).

If you now go to the Target Summary screen (under Project) you will see the App Icons section is filled in:

The icons are now recognized by the app



Launch image

You may have seen that the Target Summary screen contains a section for Launch Images (below App Icons). 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 we launch our app. The black screen appears only for a second or two, but we can make the transition between tapping the app icon and actually using the app more seamless by using a launch image.

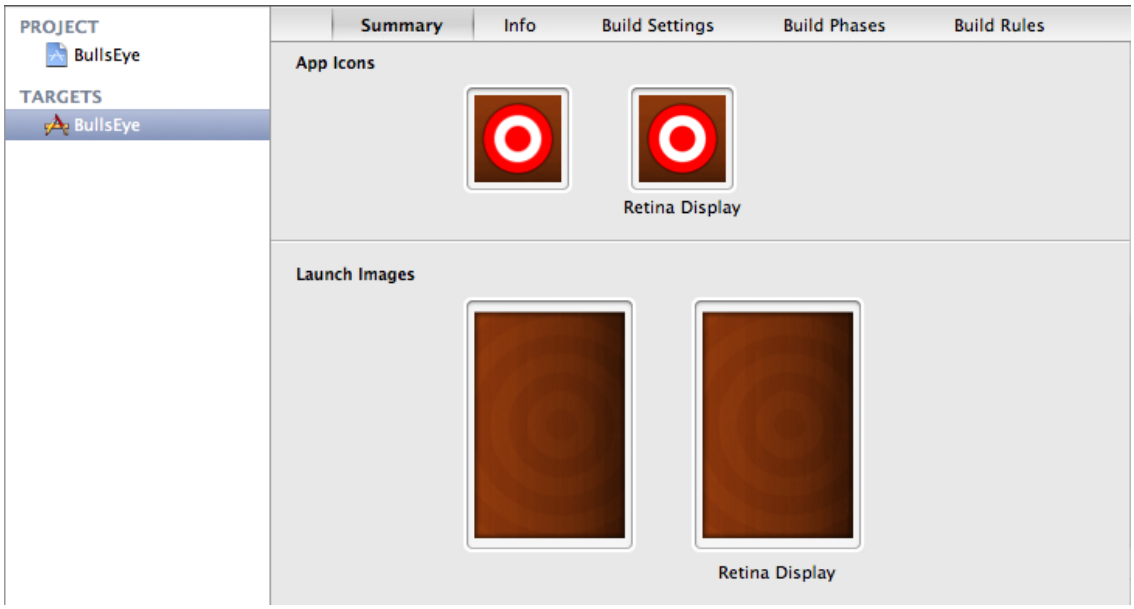
A lot of developers abuse this feature to show a splash screen with a logo, but it is better for the user if you just show a static image of the user interface and not much else. Nobody likes to wait for apps to load and a well-chosen launch image will give the illusion the app is loading faster than it actually is.

We'll just show the wood texture background. Adding these launch images is really easy.

» Select the project, right click, choose Add Files to BullsEye. Add the images from the Launch Images folder to the project, Default.png and Default@2x.png.

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

The launch images that are shown when the app starts



Now the Target Summary screen will show these images in the Launch Images section. When you start the app, the transition into the app will look a lot smoother. It's little details like these that count.

Display name

One last thing. We 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

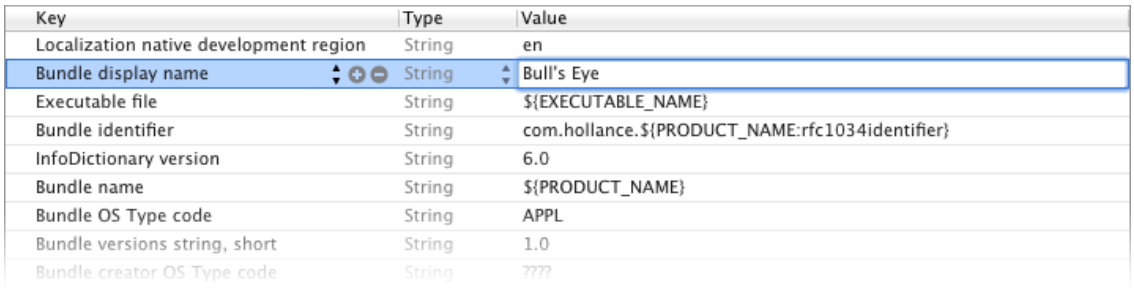
the icon and for apps with longer names you have to get creative to make the name fit. In our case, however, we have enough room to add the space and the apostrophe.

» Once again, select BullsEye-Info.plist.

There is a setting 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. We can simply replace this with the name we want.

» Double-click to edit the “Bundle display name” field and type “Bull’s Eye” (without the quotes).

Changing the display name of the app



You can find the project files for the finished app under “07 - Final App” in the tutorial’s Source Code folder. There is also a version named “08 - Final App with Comments” that has a lot of comments to show you what every piece of code does. I also removed anything that was inserted by the Xcode template that isn’t actually needed for this game, so that the code is as simple as possible.

Running the game on your device

So far, we'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 computer! Of course, you also need to be a member of the Developer Program if you want to put your apps on the iTunes App Store.

Making your Development Certificate

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.

You need to obtain these Development and Distribution Certificates from the iOS Provisioning Portal [https://developer.apple.com/ios/manage/overview/index.action]. Needless to say, you only have access to this portal if you're in the paid iOS Developer Program. For now, just make the Development Certificate — you will need that for testing your app — and do the Distribution Certificate later when you're ready to submit your app to the App Store.

The Provisioning Portal has a set of pretty good guides and videos to help you get started. On the home page you'll find a section named "How-To's" (in the right sidebar). I suggest you watch the "Obtaining your Certificate" video and follow the instructions to the letter.

This whole process may seem overwhelming, but it's really quite simple. All you have to do right now is make a Development Certificate, nothing else. To do so, you use the Keychain Access utility on your Mac to create a Certificate Signing Request (CSR) and send it to the portal. The portal then issues the certificate to you. You download that

certificate from the portal and install it on your Mac. Done! It only takes a few minutes. The video explains these steps in detail.

Setting up the Provisioning Profile

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.

It is a bit of a hassle to create a new provisioning profile for each app that you're making, especially if you're learning and trying out new things. Fortunately, it is possible to make a special profile, named a *wildcard profile*, that can be used by more than one app. Even better, Xcode has an "Automatic Device Provisioning" feature that automatically creates a wildcard provisioning profile for you and makes this available to all your devices.

Connect your iPhone to your Mac using the USB cable. Then open Xcode's Organizer window. You can do this with the button at the top-right corner of the main window:

The button that opens the Organizer window

This opens the Organizer window. Mine looks like this:

The Organizer performs different tasks. The tab that is currently showing is Devices.

Of great interest to you should be the Documentation tab. Here you can find the full documentation of the iOS SDK — all the frameworks, objects, methods and functions that you need to write iOS apps. The other tabs let you manage your source code and your projects.

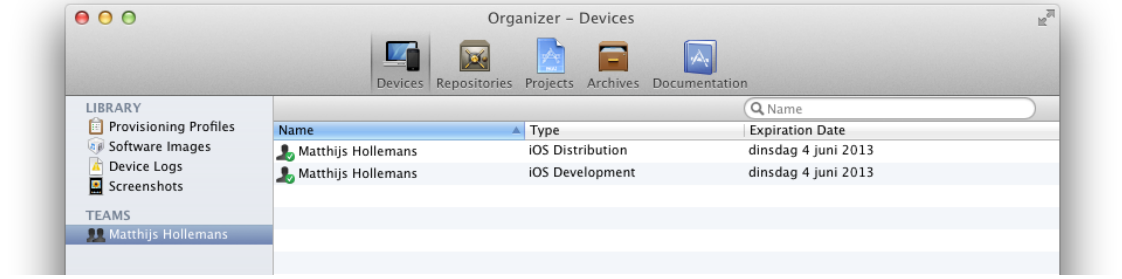
Right now, we're going to stick to the Devices tab. On the left you'll see a section named Library that contains information about your Developer Program account and the provisioning profiles that you have created. Below Library is a list of devices that can be used for development. I have four devices listed. The one with the green light next to its name is currently connected to my Mac.

In Xcode 4.2, the Developer Profile section of the Organizer shows information about your iOS Developer Program account. If your certificates are installed correctly, they show up here:

The Developer Profile section shows your iOS Developer Program certificates

In Xcode 4.3 this looks a bit different. You will now find your certificates under Teams:

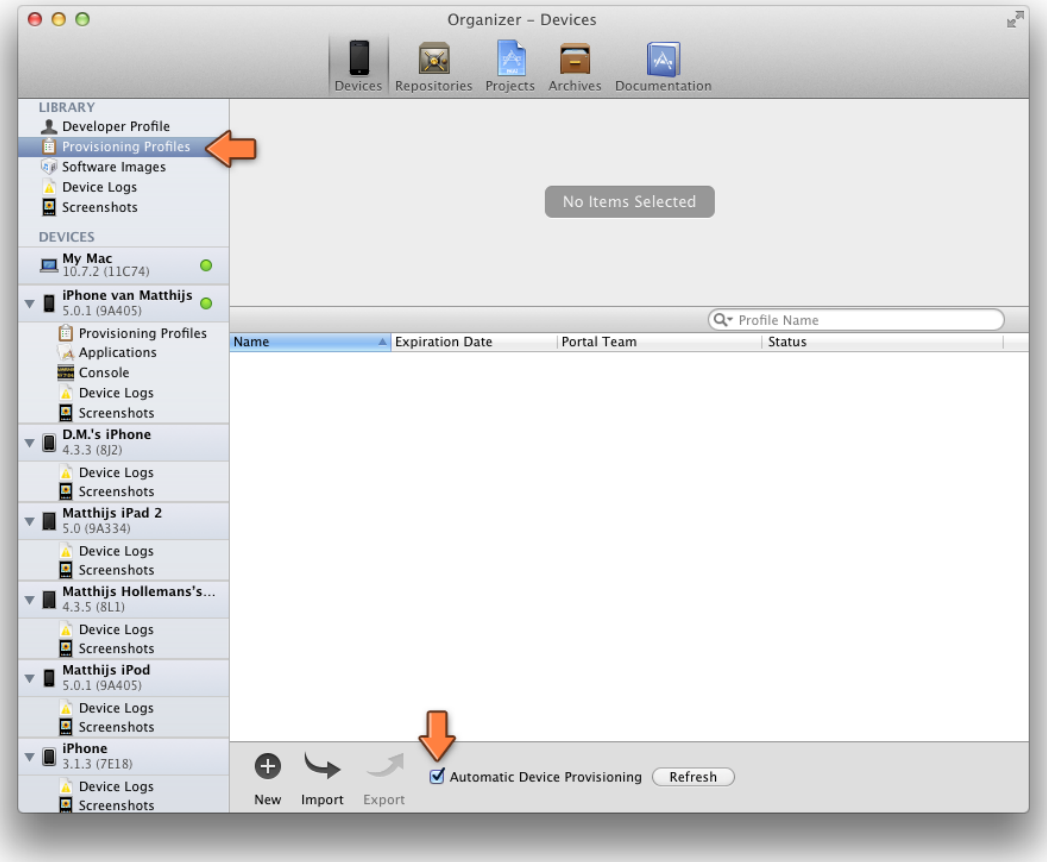
The Teams section shows your iOS Developer Program certificates



Click on Provisioning Profiles. If you haven't installed any provisioning profiles yet (and I'm assuming you haven't), this brings up a screen that is mostly empty.

» If you're using Xcode 4.2, there is a checkbox at the bottom labeled "Automatic Device Provisioning". Make sure this is selected. In Xcode 4.3 you no longer have to do this.

The empty Provisioning Profiles list



» Click on the name of your device. If this is the first time you're using the device with Xcode, 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.

After you press "Use for Development", Xcode will ask you for your iOS Provisioning Portal login:

Login to the iOS Provisioning Portal

Type the username and password for your Developer Program. This is the same login that you use for http://developer.apple.com/devcenter/ios. Xcode will ask for this password every time you do something that requires a connection to the portal, so you might want to check “Remember Password in Keychain” to save yourself some typing in the future.

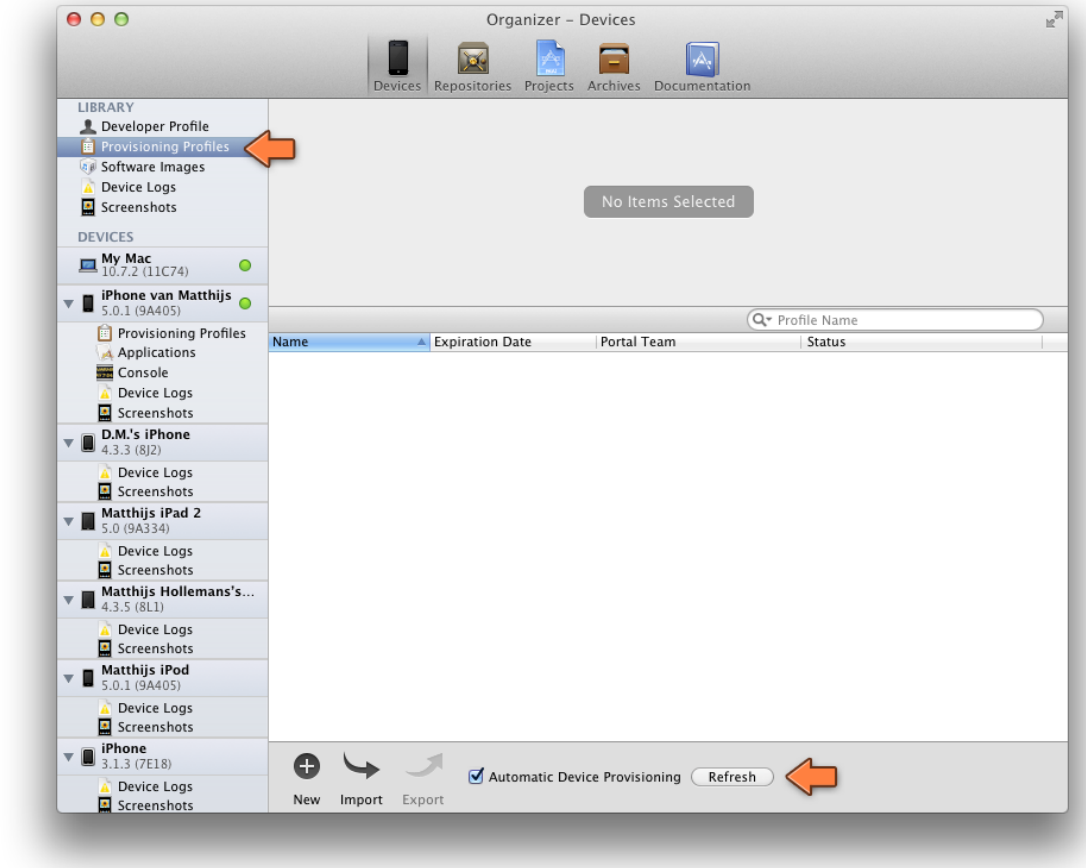
Xcode will now automatically register your device with your Developer Program account, and install the Team Provisioning Profile on your device. If you have Xcode 4.3, you may also need to click the “Add to Portal” button at the bottom of the screen to register the device.

> If the version of iOS on your device is newer than the version of the SDK you’re using then Xcode may need to collect data on the newer OS from your phone first. You should allow it to do so, it will only take a few minutes.

If you’ve used your device with Xcode before, then there is no “Use for Development” button. You can still make Xcode do the automatic provisioning. Go to the “Provisioning Profiles” section, make sure Automatic Device Provisioning is enabled, and click the Refresh button next to it.

Note: On Xcode 4.3 there no longer is a checkbox for Automatic Device Provisioning, just a Refresh button. If you’ve already added the device to the portal then you can simply click Refresh here. To add the device to the portal, select the name of the device in the sidebar and then click the Add to Portal button at the bottom of the screen.

Click Refresh to perform Automatic Device Provisioning



Xcode will ask you for your iOS Provisioning Portal login. Press Log in and… wait. After a few seconds, Xcode will update the screen:

Xcode has automatically created a new profile for you named the Team Provisioning Profile. I suppose they call it the “team” profile because it will work on all the devices that you have registered with the Developer Program. If you haven’t added your device ID to the Provisioning Portal yet, then Xcode should automatically have done this for you.

Verify that your device is listed behind Devices in the Team Provisioning Profile section at the top. If not, you may have to login to http://developer.apple.com/devcenter/ios, go to the Provisioning Portal, and add your device manually.

The Team Provisioning Profile has a 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 that won’t work with wildcard IDs such as push notifications). This means we won’t have to repeat this procedure for any of the other apps we will be developing in this series. Xcode knows about the profile now and it will automatically use this profile to sign your apps.

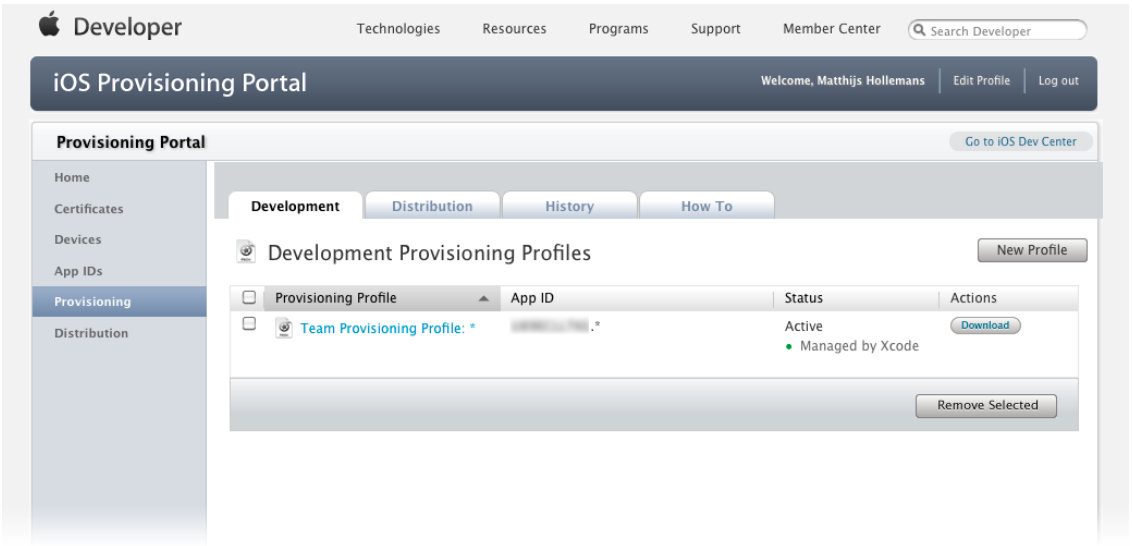
Xcode was also kind enough to install the provisioning profile on the connected device.

» Click on the name of your device in the left pane and verify that the profile is listed:

Verify that the Team Provisioning Profile was added to your device

You can also login to the Provisioning Portal using your web browser to see what the provisioning profile looks like there:

The Team Provisioning Profile in the web-based Provisioning Portal

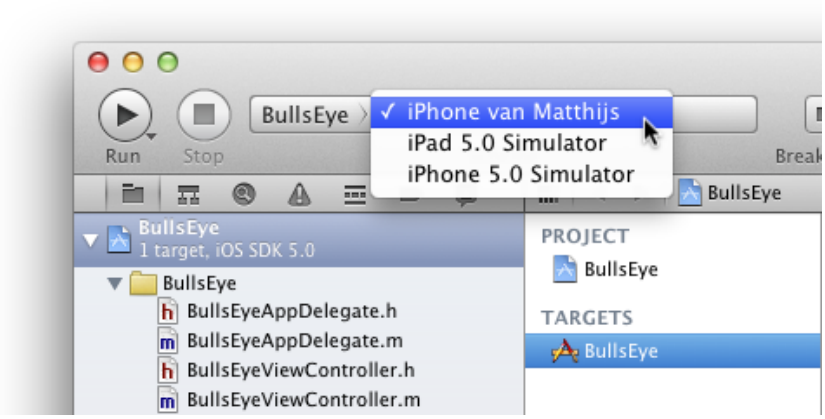


There is currently one item in this list (I actually have quite a few profiles in there but for privacy reasons I cut them out). You can see that this profile says: “Managed by Xcode” because we used the automatic provisioning feature from the Xcode Organizer. Click on the profile name to see more details.

Running the app on your device

» Go back to Xcode's main window and click on the Scheme box in the toolbar to change where we will run the app. It currently says "iPhone Simulator" 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



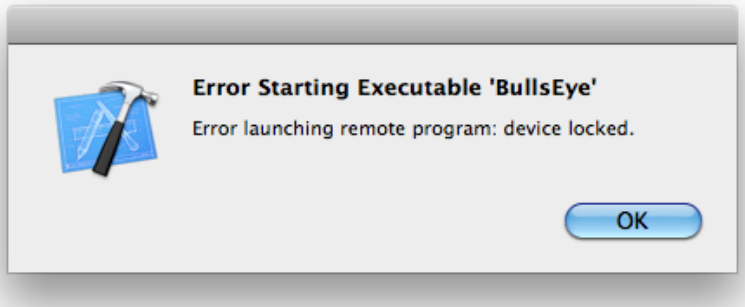
» Press Run to launch the app. Does it work? Awesome! If not, read on…

There are a few things that can go wrong here, 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 after a few minutes, you might get this warning:

The app won't run if the device is locked



Or a message in the Xcode Output pane:

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

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

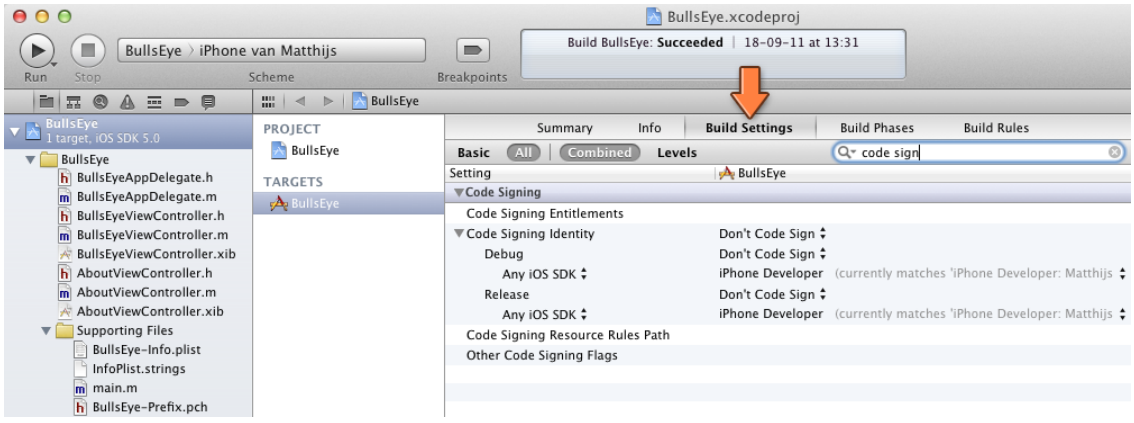
No valid provisioning profile on the device. If Xcode complains that no valid provisioning profile could be found on your iPhone, then go into the Organizer and simply drag the Team Provisioning Profile onto the device in the left column. This shouldn't really happen because Xcode will automatically install the profile onto the device before it runs the app, but you never know. Click on the Provisioning Profiles item under the name of the device and verify that the profile is installed on that device.

Code Sign error: a valid provisioning profile matching the application's Identifier 'com.yourname.BullsEye' could not be found. This means Xcode does not have a valid provisioning profile to sign the app with. The installation of the Team Provisioning Profile has apparently failed. Make sure you have installed your Developer Program certificates and follow the above steps again.

Xcode could not find a valid private-key/certificate pair for this profile in your keychain. You haven't properly installed your development certificate yet or you are trying to use a provisioning profile that belongs to someone else's Developer Program.

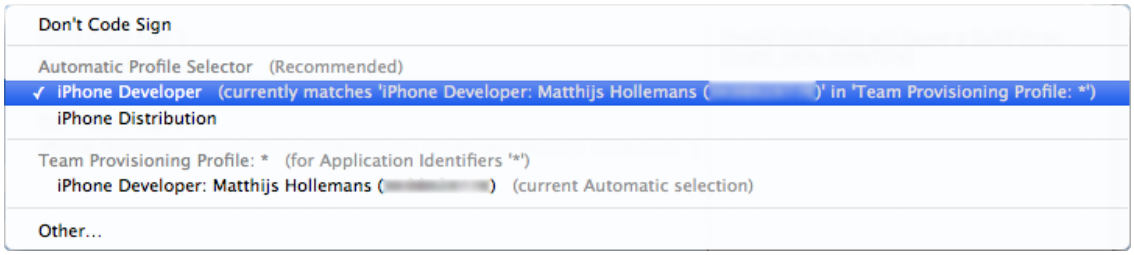
If you want to know how Xcode chooses which profile to sign your app with, then click on your project name, select the target, and switch to the Build Settings tab. There are a lot of settings in this list, so filter them by typing "code sign" in the search box. The screen will look something like this:

The Code Signing settings



Under Code Signing Identity it says Debug, Any iOS SDK: "iPhone Developer (currently matches and so on)". This is the provisioning profile that Xcode uses to sign the app. If you click on that line, you can choose another profile:

Choosing the Code Signing Identity



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

That's it!

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 we 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 [mailto:mail@hollance.com]).

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 we did, it might be a good idea to look at this cleaned up, commented source code. I have provided two versions, the one with the standard UIKit shapes (in the folder Source-Plain) and the good-looking one (in Source-Pretty).

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 subtlepatterns.com [http://subtlepatterns.com] and was made by Atle Mo.

But there's more!

Thank you for reading the first tutorial of my ebook 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 raywenderlich.com [http://www.raywenderlich.com/ios-apprentice].

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, we're going to create a basic to-do list app using Xcode's new Storyboards feature. 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. We'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 we'll do in this final tutorial of the series. We'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 from raywenderlich.com/ios-apprentice [http://www.raywenderlich.com/ios-apprentice]. They're worth it if you want to become a great iOS developer!

About the author

Matthijs Hollemans is an independent iPad and iPhone developer and designer from the Netherlands. He writes about the technical and non-technical aspects of developing iOS apps on his blog, www.hollance.com [http://www.hollance.com]. He also writes tutorials for raywenderlich.com [http://www.raywenderlich.com].

Feel free to send Matthijs an email [mailto:mail@hollance.com] if you have any questions or comments about these tutorials. And of course you're welcome to visit the forums [http://www.raywenderlich.com/forums/viewforum.php?f=9] for some good conversation.

Thanks for reading!

Revision history

- v1.4 (14 Aug 2012) - Updated for iOS 6. Removed explanation of `viewDidUnload`, as its use is no longer recommended by Apple.
- v1.3 (6 June 2012) - Updated for Xcode 4.3. Added PDF version.
- v1.2 (18 Dec 2011) - Added tutorial 4, StoreSearch
- v1.1 (18 Sept 2011) - Updated for iOS 5
- v1.0 (5 July 2011) - First version (iOS 4)