Ray Wenderlich para mim 27 de jul

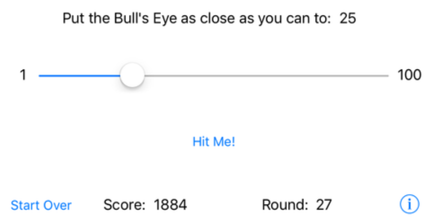
José, welcome back to the raywenderlich.com **iOS Apprentice Email Course**!

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

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

So our next step is to add the rest of the controls – the slider and the text labels – and turn this app into a real game!

Our goal is for the app will look like this:



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

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

But the layout of all these controls is for another day. Today, we'll be taking care of two important prerequisites:

- **Portrait vs. landscape:** Switch your app to landscape mode.
- **Objects, data and methods:** Get a quick primer on the basics of object oriented programming.

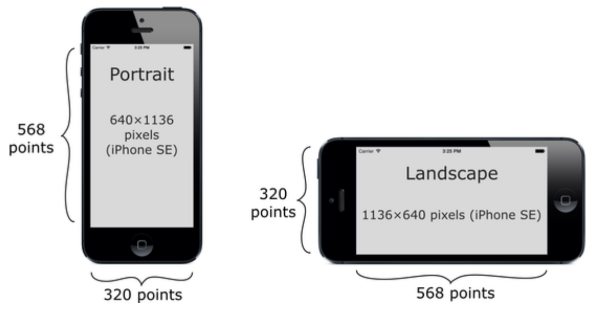
Portrait vs. landscape

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

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

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

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



So what is a **point**?

On older devices – up to the iPhone 3GS and corresponding iPod touch models, as well as the first iPads – one point corresponds to one pixel. As a result, these low-resolution devices don't look very sharp because of their big, chunky pixels.

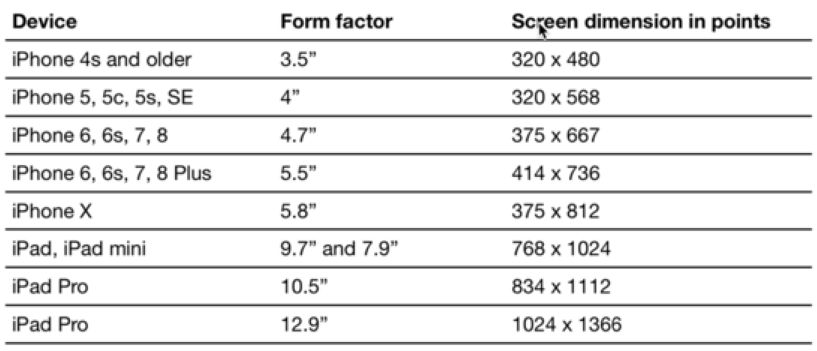
I'm sure you know what a pixel is? In case you don't, it's the smallest element that a screen is made up of. (That's how the word originated, a shortened form of pictures, PICS or PIX + ELement = PIXEL.) The display of your iPhone is a big matrix of pixels that each can have their own color, just like a TV screen. Changing the color values of these pixels produces a visible image on the display. The more pixels, the better the image looks.

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

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

It's not only the number of pixels that differs between the various iPhone and iPad models. Over the years they have received different form factors, from the small 3.5-inch screen in the beginning all the way up to 12.9-inches on the iPad Pro model.

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



Device	Form factor	Screen dimension in points
iPhone 4s and older	3.5"	320 x 480
iPhone 5, 5c, 5s, SE	4"	320 x 568
iPhone 6, 6s, 7, 8	4.7"	375 x 667
iPhone 6, 6s, 7, 8 Plus	5.5"	414 x 736
iPhone X	5.8"	375 x 812
iPad, iPad mini	9.7" and 7.9"	768 x 1024
iPad Pro	10.5"	834 x 1112
iPad Pro	12.9"	1024 x 1366

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

UIKit and other frameworks
iOS offers a lot of building blocks in the form of frameworks or "kits". The UIKit framework provides the user interface controls such as buttons, labels and navigation bars. It manages the view controllers and generally takes care of anything else that deals with your app's user interface. (That is what UI stands for: User Interface.)
If you had to write all that stuff from scratch, you'd be busy for a long 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 put in.
Any object you see whose name starts with UI, such as `UIButton`, comes from UIKit. When you're writing iOS apps, UIKit is the framework you'll spend most of your time with, but there are others as well.
Examples of other frameworks are Foundation, which provides many of the basic building blocks for building apps; Core Graphics for drawing basic shapes such as lines, gradients and images on the screen; AVFoundation for playing sound and video; and many others.
The complete set of frameworks for iOS is known collectively as Cocoa Touch.

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

Developers work in points, designers work in pixels.

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

For the time being, you'll work with just the iPhone SE screen size of 320×568 points – just to keep things simple. Later on you'll also make the game fit on the other iPhone screens.

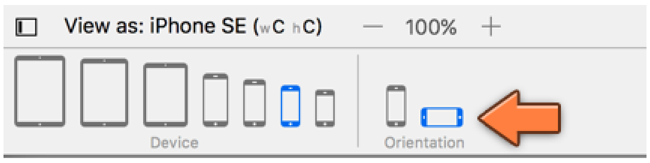
Convert the app to landscape

Open up the Bull's Eye project where you left it off last time (or download the starter project from the corresponding forum discussion thread), and take a look at Xcode.

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

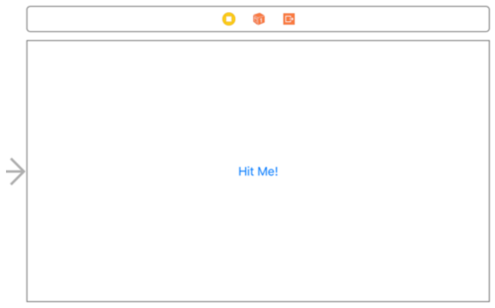
1. Make the view in **Main.storyboard** landscape instead of portrait.
2. Change the **Supported Device Orientations** setting of the app.

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



This changes the dimensions of the view controller. It also puts the button off-center.

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



That takes care of the view layout.

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

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

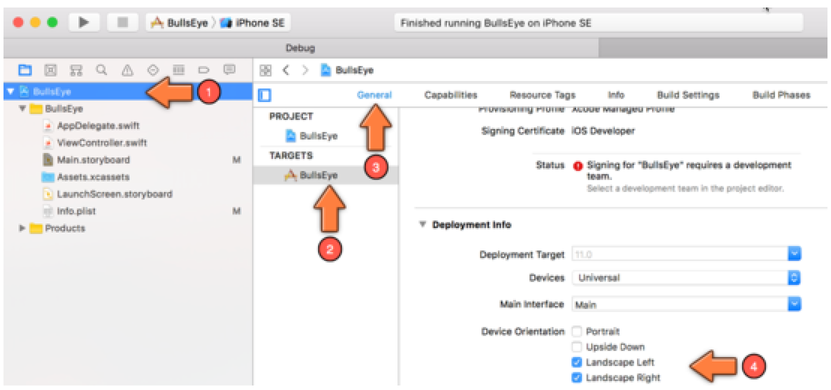
Now, everything will look as it should.

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

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

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

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



Provisioning Profile Xcode Managed Profile
Signing Certificate iOS Developer
Status Signing for "BullsEye" requires a development team.
Select a development team in the project editor.

Deployment Info
Deployment Target 11.0
Devices Universal
Main Interface Main
Device Orientation
☐ Portrait
☐ Upside Down
☑ Landscape Left
☑ Landscape Right

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

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

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

Objects, data and methods

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

Swift is a so-called "object-oriented" programming language, which means that most of the stuff you do involves objects of some kind. I already mentioned a few times that an app consists of objects that send messages to each other.

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

Objects

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

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

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

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

The project also has an object named `AppDelegate` - you're going to ignore that for the moment, but feel free to look at its source if you're curious. These object thingies are everywhere!

Data and methods

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

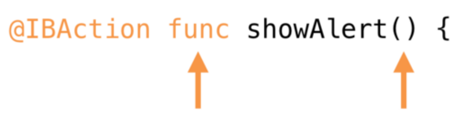
- An example of data is the Hit Me button that you added to the view controller earlier. When you dragged the button into the storyboard, it actually became part of the view controller's data. Data **contains** something. In this case, the view controller contains the button.

- An example of functionality is the `showAlert` action that you added to respond to taps on the 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 tapped on it and will trigger an action in response.

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

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

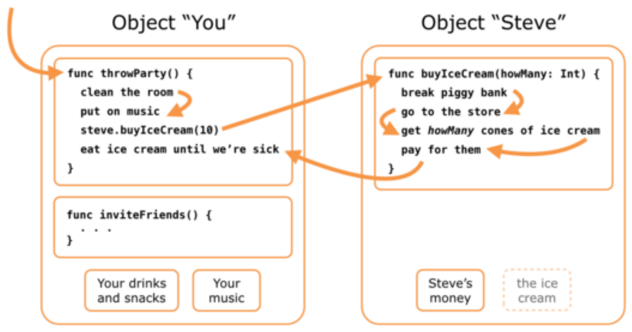


```
@IBAction func showAlert() {
```

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

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

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



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

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

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

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

Messages

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

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

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

Objects can look at each other's data (to some extent anyway, just like Steve may not approve if you peek inside his wallet) and can ask other objects to perform their methods. That's how you get your app to do things. (But not all data from an object can be inspected by other objects and/or code - this is an area known as access control and you'll learn about this later.)

Phew - that was a lot of theory! Time for a break.

In my next email, you'll add the rest of the game's controls, and you'll write some more Swift code! Talk to you soon,

- Ray