Project 2: Guess the Flag

Overview

`Brief`: Make a game using UIKit, and learn about integers, buttons, colors and actions.

`Learn`: @2x and @3x images, asset catalogs, integers, doubles, floats, operators, UIButton, enums, CALayer, UIColor, random numbers, actions, string interpolation, UIAlertController.

- Setting up
- Designing your layout
- Making the basic game work: UIButton and CALayer
- Guess which flag: Random numbers
- From outlets to actions: IBAction and string interpolation
- Wrap up

Setting up

In this project you'll produce a game that shows some random flags to users and asks them to choose which one belongs to a particular country. After the behemoth that was the introductory project, this one will look quite easy in comparison – you've already learned about things like outlets, image views, arrays and Auto Layout, after all.

(PS: If you skipped project 1 thinking it would all be about history or some other tedium, you were wrong. This project will be very hard if you haven't completed project 1!)

However, one of the keys to learning is to use what you've learned several times over in various ways, so that your new knowledge really sinks in. The purpose of this project is to do exactly that: it's not complicated, it's about giving you the chance to use the things you just learned so that you really start to internalize it all.

So, launch Xcode, and choose "Create a new project" from the welcome screen. Choose Single View Application from the list and click Next. For Product Name enter "Project2", then make sure you have Swift selected for language and iPhone for devices. Now click Next again and you'll be asked where you want to save the project – your desktop is fine.
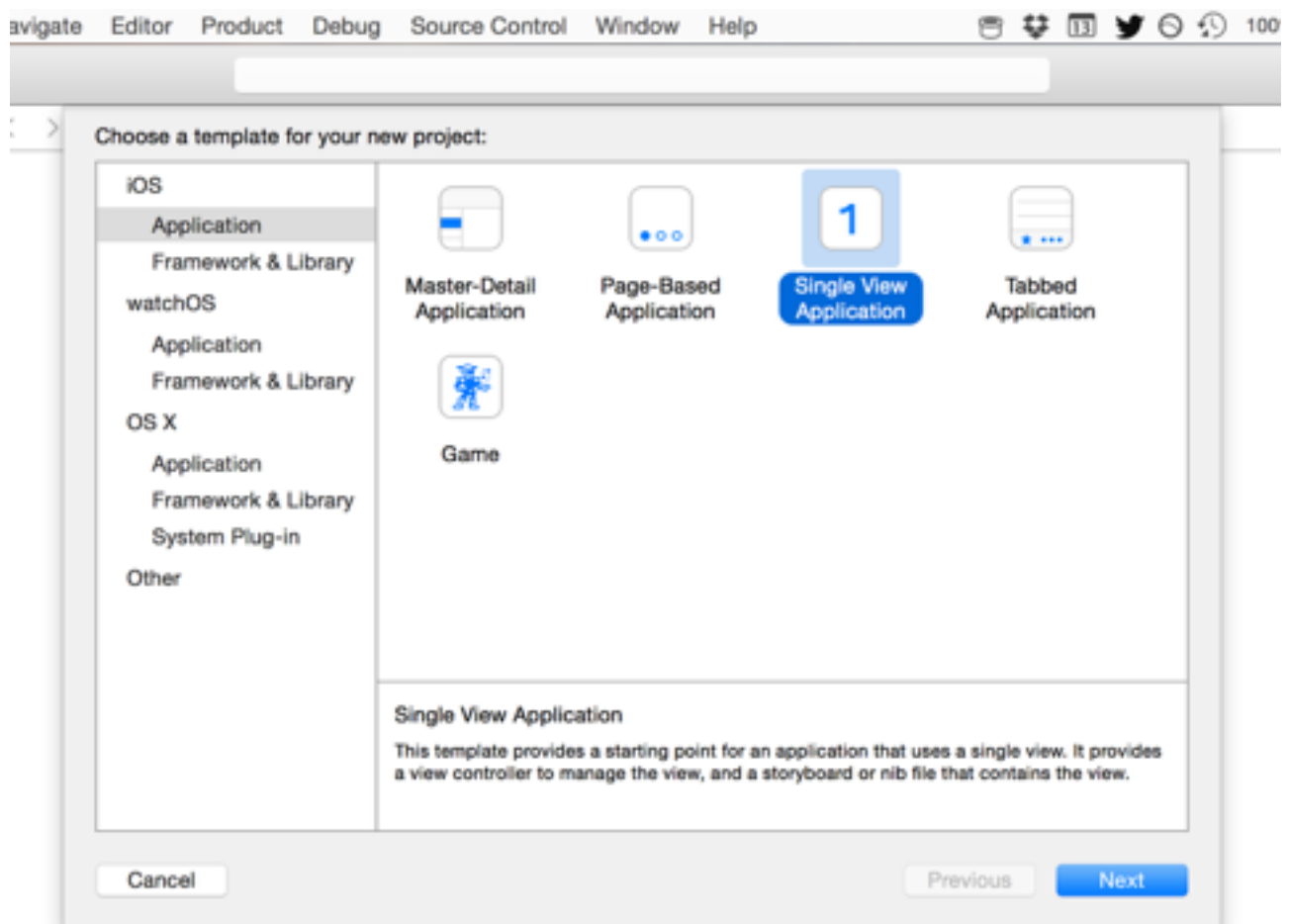
Designing your layout

When working on my own projects, I find designing the user interface the easiest way to begin any project – it's fun, it's immediately clear whether your idea is feasible, and it also forces you to think about user journeys while you work. This project isn't complicated, but still `Interface Builder` is where we're going to begin.

A `Single View Application` project template gives you one `UIViewController`, called `ViewController`, and a storyboard called `Main.storyboard` that contains the layout for our single view controller. Choose that storyboard now to open `Interface Builder`, and you'll see a big, blank space ready for your genius to begin.
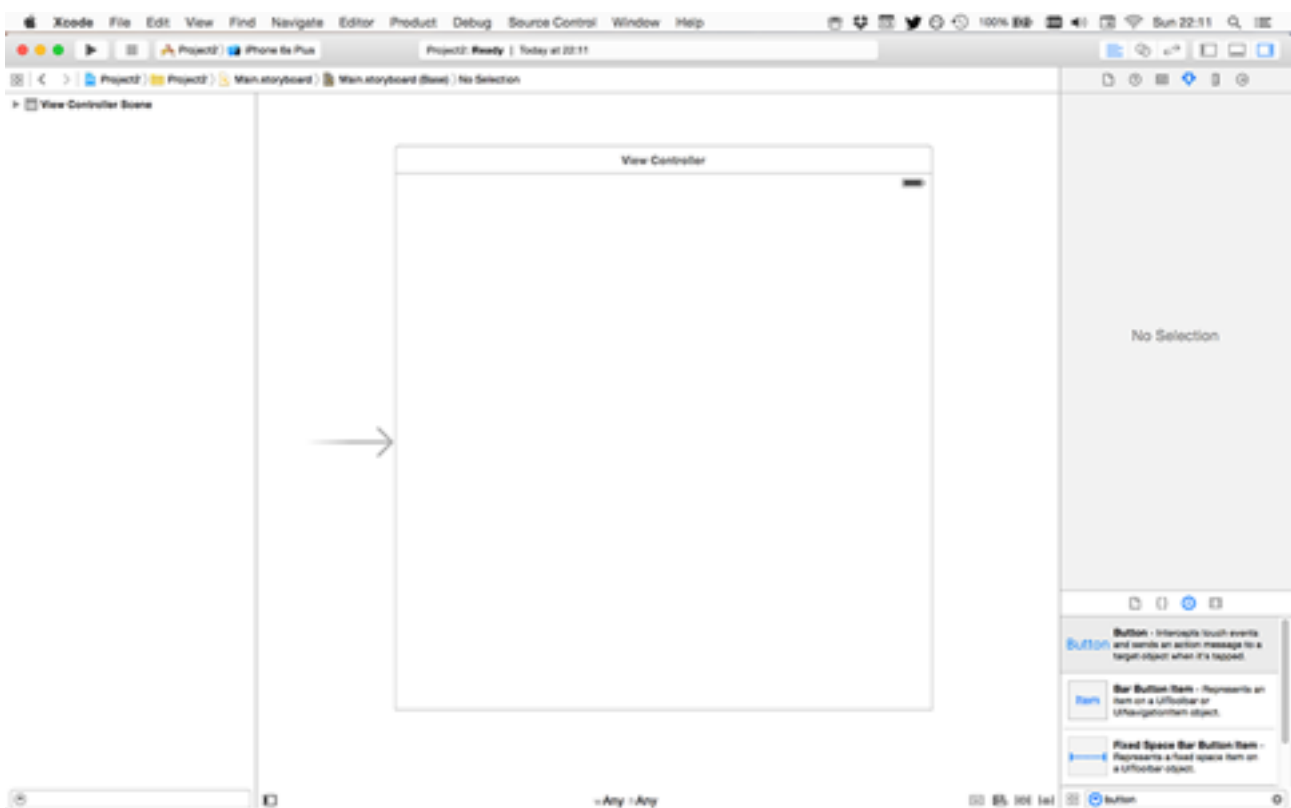
In our game, we're going to show users three flags, with the name of the country to guess shown in the navigation bar at the top. What navigation bar? Well, there isn't

one, or at least not yet. `Single View Application` projects don't come with a navigation controller as standard, but it's trivial to add one: click inside the view controller, then go to the `Editor menu` and choose `Embed In > Navigation Controller`.

With the new navigation controller in place, scroll so you can see our empty view controller again, and draw out three `UIButtons` onto the canvas. This is a new view type, but as you might imagine it's just a button that users can tap. Each of them should be `200` wide by `100` high. You can set these values exactly by using the size inspector in the top-right of the `Xcode` window.
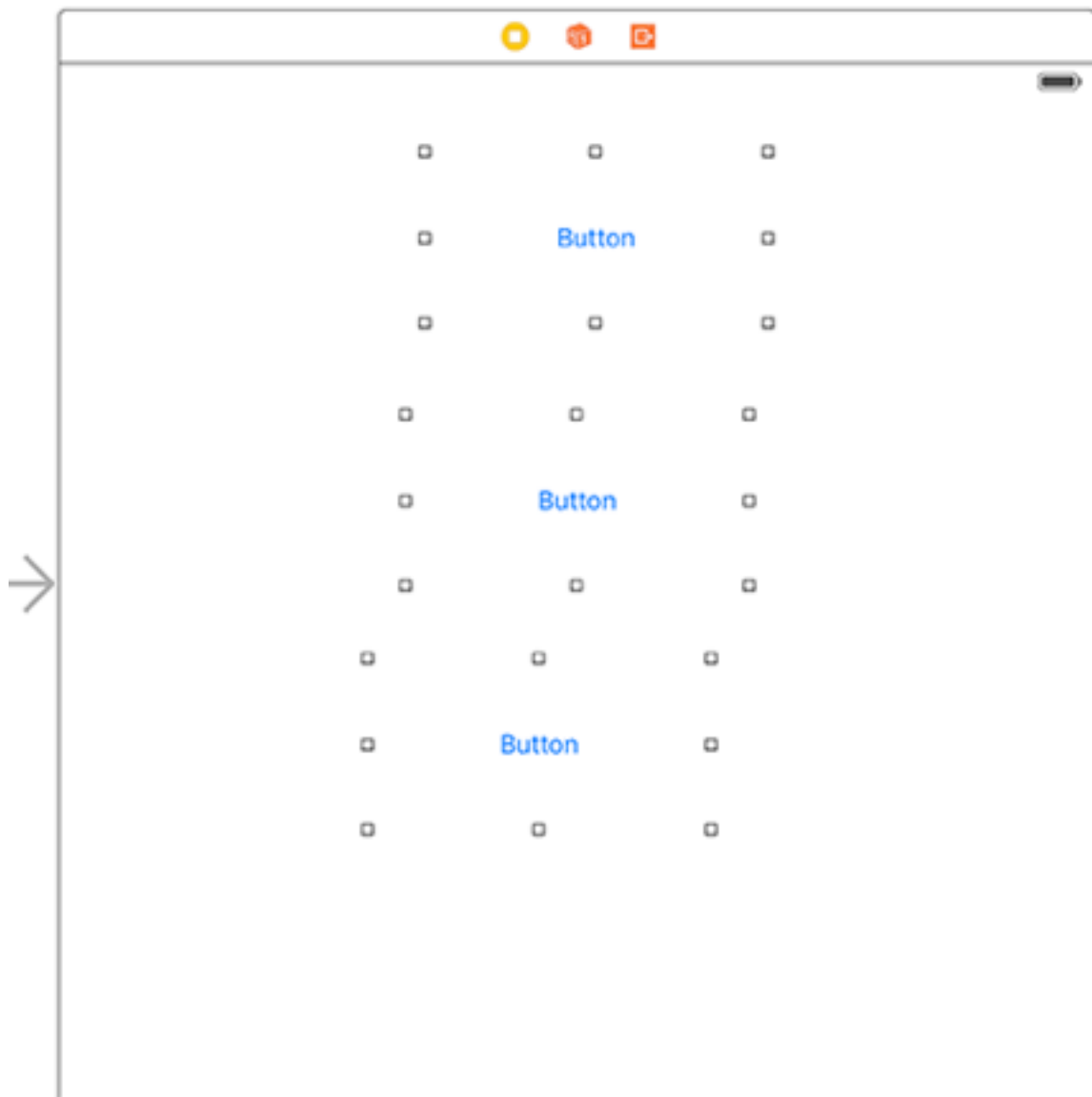
In the "old days" of iOS 6 and earlier, these UIButtons had a white background color and rounded edges so they were visibly tappable, but from iOS 7 onwards buttons have been completely flat with just some text. That's OK, though; we'll make them more interesting soon.

You can jump to the size inspector directly by pressing the keyboard shortcut Alt + Cmd + 5. Don't worry about the X positions, but the Y positions should be 100 for the first flag, 230 for the second, and 360 for the third. This should make them more or less evenly spaced in the view controller.

In the picture below you can see the size inspector, which is the quickest and easiest way to position and size views if you know exactly where you want them.

The next step is to bring in Auto Layout so that we lay down our layout as rules that can be adapted based on whatever device the user has. The rules in this case aren't complicated, but I hope will begin to show you just how clever Auto Layout is.

We're going to create our Auto Layout rules differently from in Project 1. This is not because one way is better than another, instead just so you that you can see the various possibilities and decide which one suits you best.

Select the top button, then Ctrl-drag from

there directly upwards to just outside itself – i.e., onto the white area of the view controller. As you do this, the white area will turn blue to show that it's going to be used for `Auto Layout`.
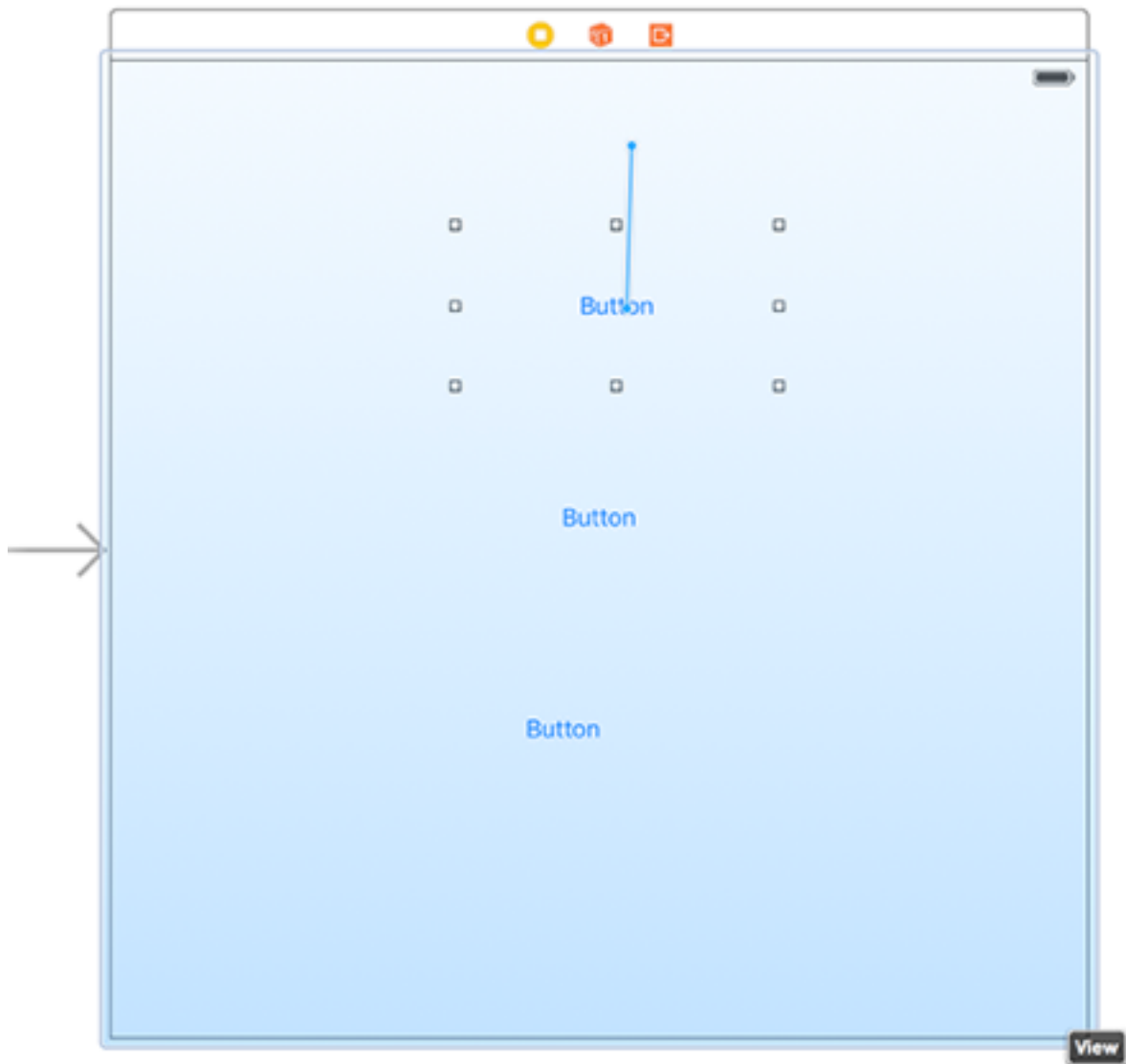
When you let go of the mouse button, you'll be presented with a list of possible constraints to create. In that list are two we care about: "`Top Space to Top Layout Guide`" and "`Center Horizontally in Container`".

You have two options when creating multiple constraints like this: you can either select one then `Ctrl-drag` again and select the other, or you can `hold down`

`shift` before selecting an item in the menu, and you'll be able to select more than one at a time. That is, `Ctrl-drag` from the button straight up to the white space in the view controller, let go of the mouse button and `Ctrl` so the menu appears, then `hold down Shift` and choose "Top Space to Top Layout Guide" and "Center Horizontally in Container".
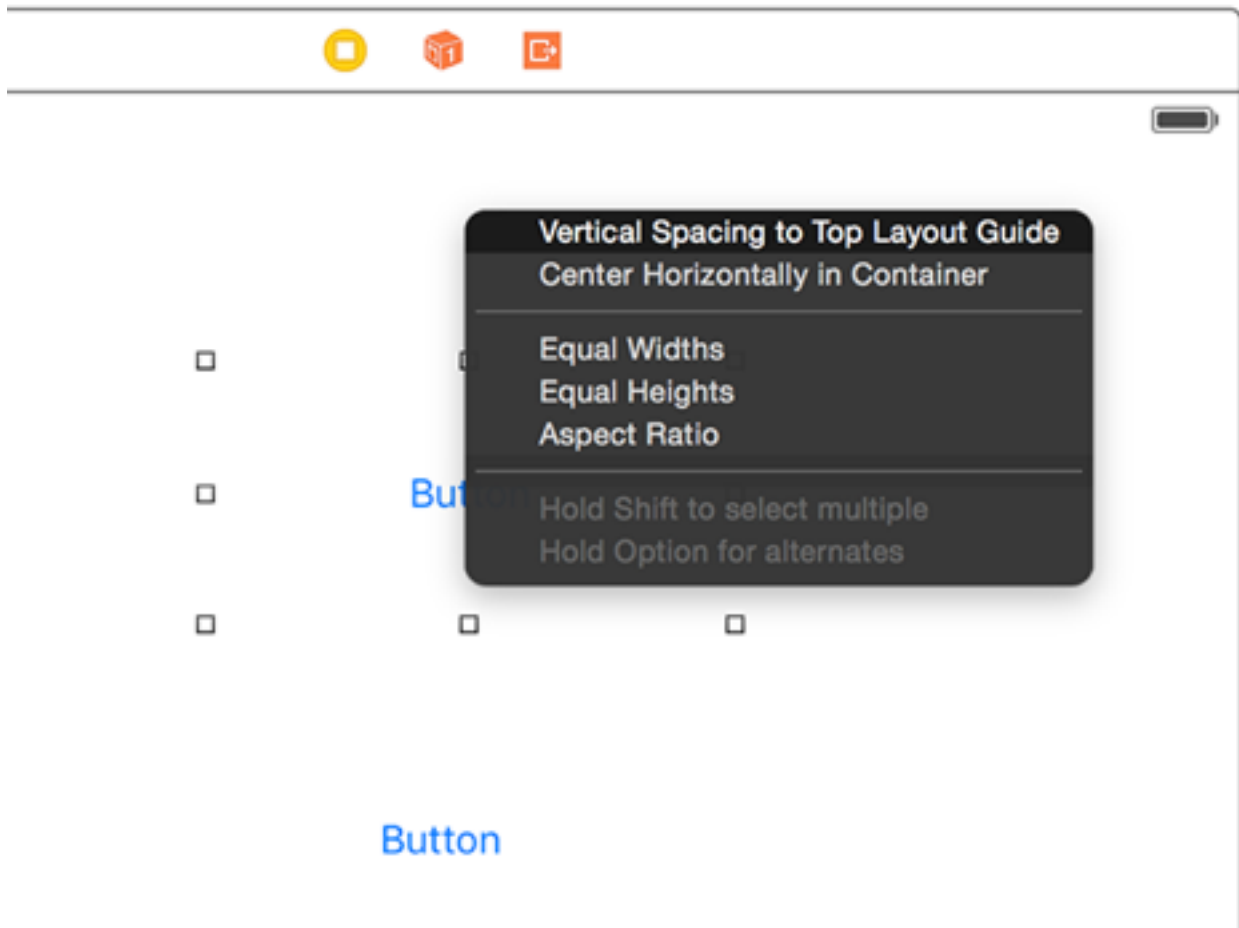
That's the first flag complete, so before we go any further let's bring it to life by adding some example content so you can see how it looks.

In `Project 1`, we added images to a project just by dragging a folder called `Content` into our `Xcode` project. That's perfectly fine and you're welcome to continue doing that for your other projects, but I want to introduce you to another option called `Asset Catalogs`. These are highly optimised ways of importing and using images in `iOS` projects, and are just as easy to use as a content folder.

In your Xcode project, select the file called `Images.xcassets`. This isn't really a file, instead it's our default Xcode asset catalog. Take 36 images files from `Content` folder.

Select all 36 flag pictures from the project files, and drag them into the Xcode window to beneath where it says "AppIcon" in our asset catalog. This will create 12 new entries in the asset catalog, one for each country.
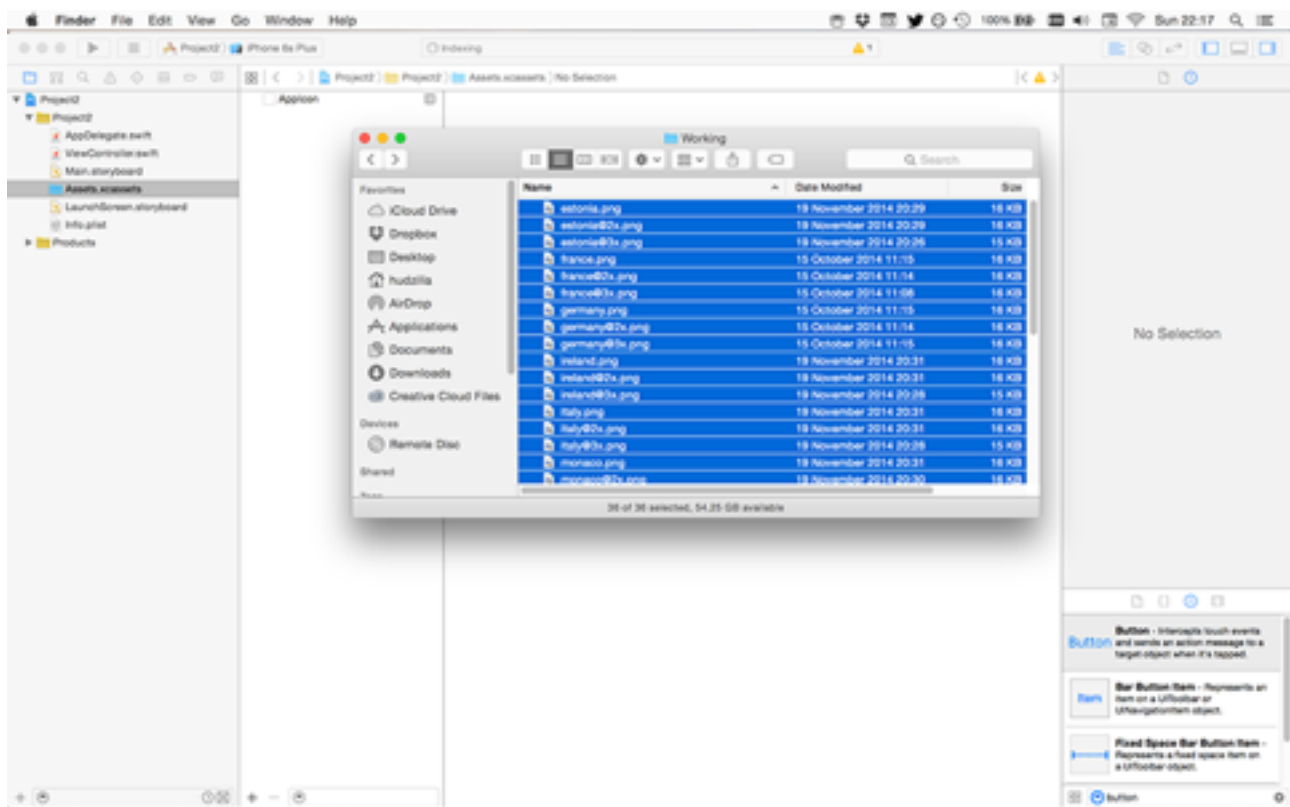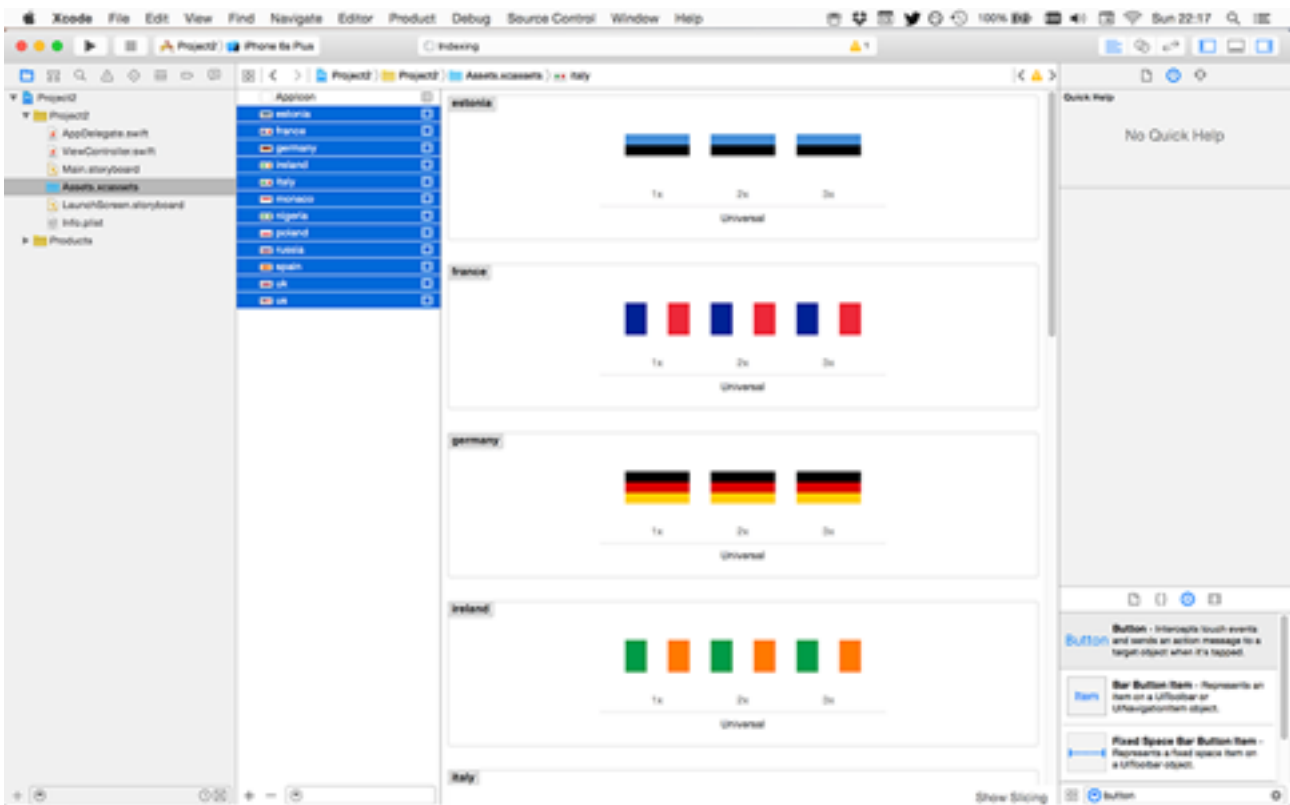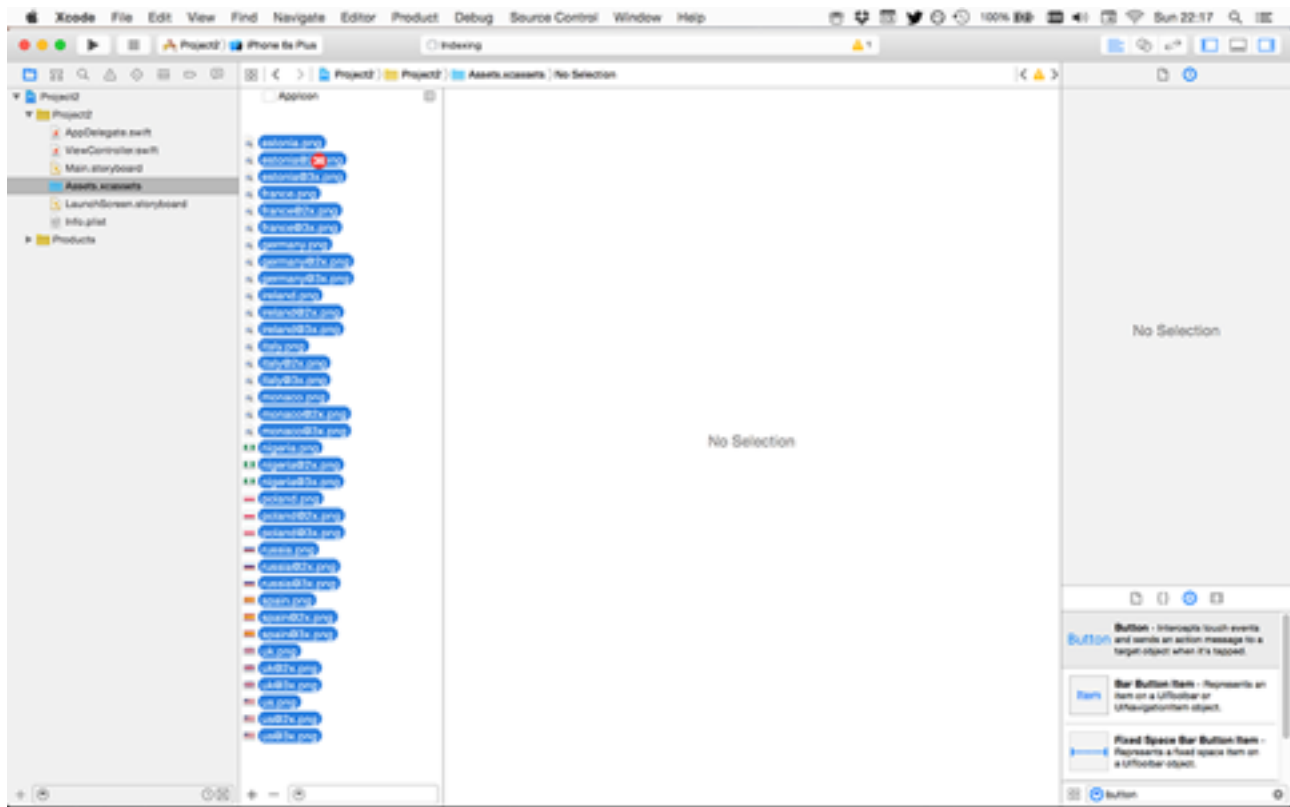
As much as I hate diversions, this one is important: iOS assets come in the sizes 1x, 2x and 3x. A 1x image is just called its regular name, eg `hello.png`, and is

used on all non-retina devices – that's `iPhone`, `iPhone 3G`, `iPhone 3GS`, `iPad`, `iPad 2` and `iPad Mini`.

A 2x image is exactly twice the size of the 1x image, and has @2x before its path extension. For example, `hello@2x.png`. This is used on all retina devices, so that's `iPhone 4`, `iPhone 4s`, `iPhone 5`, `iPhone 5s`, `iPhone 6`, `iPad 3`, `iPad 4`, `iPad Air` and `iPad Air 2`.

A 3x image is exactly three times the size of the 1x image, and has @3x before its path extension. For example, `hello@3x.png`. This is used on retina HD devices, which at the time of writing is just the `iPhone 6 Plus`.

Clearly it would be quite tedious to have to load the correct image for each device you're working with, so `iOS` does two pieces of magic that make our lives much easier. First, images are always referred to by their 1x name: `hello.png`. In fact, for PNG files you don't even need the "`.png`" – you can just write "`hello`". iOS will automatically load 1x, 2x or 3x versions of the image depending on the user's device and what images you have provided.

The second piece of magic `iOS` does is about layout space. In `iOS`, sizes are measured in "`points`", not pixels. Non-retina `iPhones` were 320x480 pixels in size and retina `iPhones` took that up to 640x960 (`iPhone 4` and `4s`) then 640x1136 (`iPhone 5` and `5s`), but rather than make developers have to write their code twice `Apple` again took the pain away: an `iPhone 3GS` (non-retina) is 320x480 points high, and an `iPhone 4` (retina) is also 320x480 points high. So, think of points as a virtual unit of measurement, and the system will automatically map it to physical pixels for you.

All this is important because when we imported the images into our asset catalog, they were automatically placed into 1x, 2x and 3x buckets. This is because I had

named the files correctly: `france.png`, `france@2x.png`, `france@3x.png`, and so on.

Once the images are imported, you can go ahead and use them either in code or in `Interface Builder`, just as you would do if they were loose files inside a content folder. So, go back to your storyboard, choose the first button and select the attributes inspector (`Alt + Cmd + 4`). You'll see it has the title "Button" right now (this is in a text field directly beneath where it says "`Title: Plain`"), so please delete that text. Now click the arrow next to the Image dropdown menu and choose "`us`".

As soon as you set a picture inside the button, our constraints for the button are complete: it has a Y position because we placed a constraint, it has an X position because we're centering it horizontally, and it has a width and a height because it's reading it from the image we assigned. Go ahead and assign the US flag to the other two buttons while you're there.

To complete our `Auto Layout` constraints, we need to assign `Auto Layout` constraints for the middle and bottom buttons. Select the middle button, then `Ctrl-drag` to the first button — not to the view controller. Let go, and you'll see "`Vertical Spacing`" and "`Center X`". Choose both of these. Now choose the third button and `Ctrl-drag` to the second button, and again choose "`Vertical Spacing`" and "`Center X`".

At this point, our `Auto Layout` is almost complete, but you'll notice that even though we chose to center the flags horizontally, they all seem to be stuck where they were placed. The reason for this is that you need to tell `Interface Builder` to update all the frames of your buttons to match the `Auto Layout` rules you just created.

This is easy enough to do – from the four buttons at the bottom of `Interface Builder`, click the one immediately to the right of the `Pin` button we used previously. This button is called "`Resolve Auto Layout issues`", and when you click it you'll see a menu full of options. One of those will be grayed out, saying "`All Views in View Controller`", but directly beneath that is the option "`Update Frames`". Click that now, and all three buttons will snap to the real center.

The last step before we're finished with `Interface Builder` for now is to add some outlets for our three flag buttons, so that we can reference them in code. Activate the assistant editor, then `Ctrl-drag` from the first flag to create an outlet called `button1`, then from the second flag to create `button2`, and from the third flag to create `button3`.
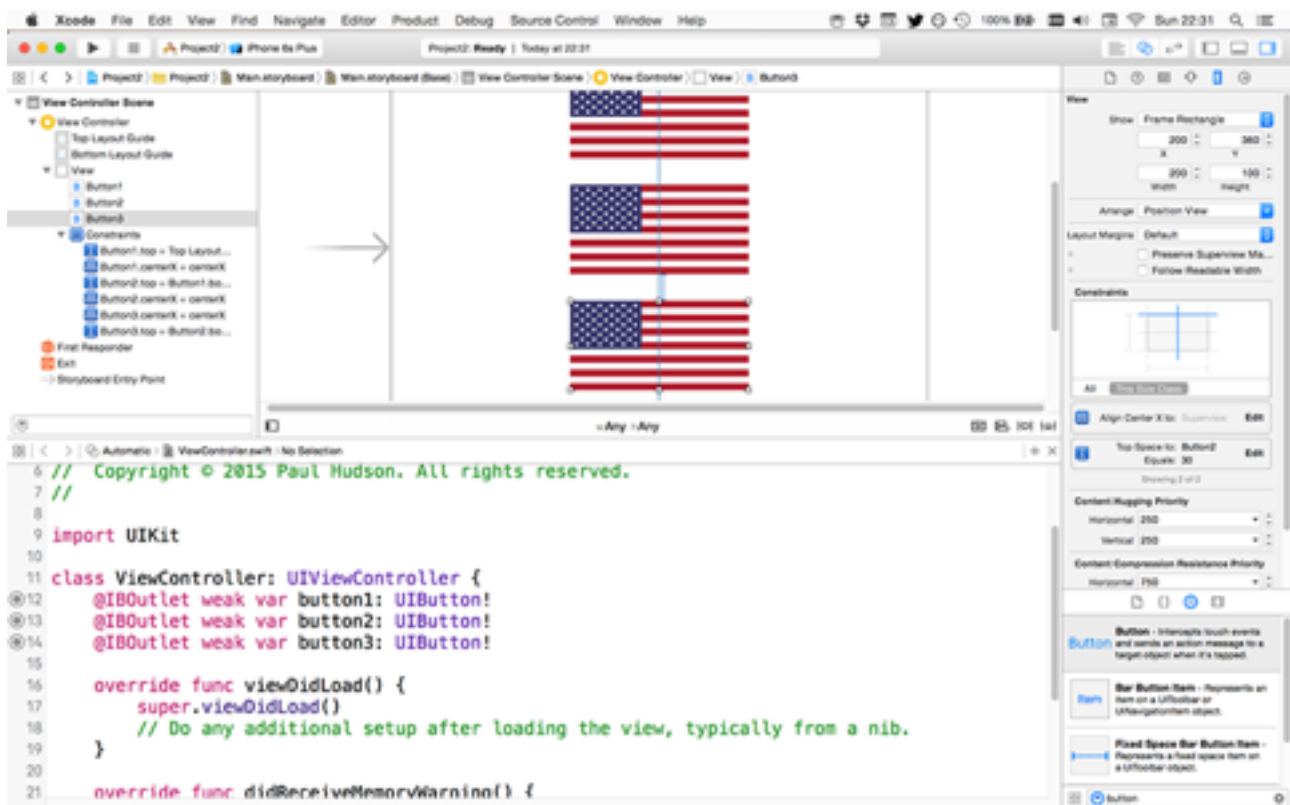
We'll come back to it later on, but for now we're done with `Interface Builder`. Select `ViewController.swift` and go back to the standard editor (that is, press `Cmd + return` turn off the assistant editor) so we can get busy with some coding.

Making the basic game work: UIButton and CALayer

We're going to create an array of strings that will hold all the countries that will be used for our game, and at the same time we're going to create two more properties that will hold the player's current score – it's a game, after all!

Let's start with the new properties. Add these two lines directly beneath the `@IBOutlet` lines you added earlier in `ViewController.swift`:

```
var countries = [String]()
var score = 0
```

The first line is something you saw in `project 1`: it creates a new property called `countries` that will hold a new array of strings. The second one is new, but hopefully it's easy enough to guess: it creates a new property called `score` and gives it the value 0. These two lines ultimately do similar things, but the way they work is different.

- `var a = 0` – tells `Swift` that we want to put the number 0 into `score`. 0 is what's known as an integer (`Int` in Swift), which means it's a whole number. 556 is an integer. `100000001` is an integer. `3.14159` is not, because it's not a whole number. By "`whole number`" I mean a number that has nothing after its decimal point, but for the sake of clarity I should add that integers can be negative too.

- `var a = [String]()` — This tells `Swift` that we want to put an array of strings into a. The syntax looks a little strange because it both declares what type we want, which is `[String]`, and also creates it: those `()` literally make this a method call that creates the `[String]` array.

What you're seeing here is called type inference. Type inference means that `Swift` figures out what data type a variable or constant should be based on what you put into it. This means a) you need to put the right thing into your variables otherwise they'll have a different type from what you expect, b) you can't change your mind later and try to put an integer into an array, and c) you only have to give something an explicit type if `Swift's` inference is wrong.

To get you started, here are some example type inferences:

- `var score = 0` — This makes an `Int` (integer), so it holds whole numbers.
- `var score = 0.0` — This makes a `Double`, which is one of several ways of holding decimal numbers, e.g. `3.14159`.
- `var score = "hello"` — This makes a `String`, so it holds text.
- `var score = ""` — Even though there's no text in the quote marks, this still makes a `String`.
- `var score = ["hello"]` — This makes a `[String]` with one item, so it's an array where every item is a `String`.
- `var score = ["hello", "world"]` — This makes a `[String]` with two items, so it's an array where every item is a `String`.

It's preferable to let `Swift's` type inference do its work whenever possible. However, if you want to be explicit, you can be:

- `var score: Double = 0` – Swift sees the `0` so thinks you want an Int, but we're explicitly forcing it to be a `Double` anyway.
- `var score: Float = 0.0` – Swift sees the `0.0` and thinks you want a `Double`, but we're explicitly forcing it to be a `Float`. I said that `Double` is one of several ways of holding decimal numbers, and `Float` is another. Put simply, `Double` is a high-precision form of `Float`, which means it holds much larger numbers, or alternatively much more precise numbers.

We're going to be putting all this into practice over the next few minutes. First, let's fill our countries array with the flags we have, so add this code inside the `viewDidLoad()` method:

```
countries.append("estonia")
countries.append("france")
countries.append("germany")
countries.append("ireland")
countries.append("italy")
countries.append("monaco")
countries.append("nigeria")
countries.append("poland")
countries.append("russia")
countries.append("spain")
countries.append("uk")
countries.append("us")
```

This is identical to the code you saw in the `project 1`, so there's nothing to learn here. There's a more efficient way of doing this, which is to create it all on one line. To do that, you would write:

```
countries += ["estonia", "france", "germany", "ireland",
"italy", "monaco", "nigeria", "poland", "russia", "spain",
"uk", "us"]
```

This one line of code does two things. First, it creates a new array containing all the countries. Like our existing countries array, this is of type `[String]`. It then uses something new, `+=`. This is called an operator, which means it operates on variables and constants – it does things with them. + is an operator, as are `-, *,` = and more. So, when you say "`5 + 4`" you've got a constant (`5`) an operator (`+`) and another constant (`4`).

In the case of += it combines the + operator (add) and the = operator (assign) to make "`add and assign`". Translated, this means "`add the thing on the right to the thing on the left`", or in the case of our countries line of code it means, "`add the new array of countries on the right to the existing array of countries on the left`".

Now that we have the countries all set up, there's one more line to put just before the end of `viewDidLoad()`:

```
askQuestion()
```

This calls the `askQuestion()` method. Now, this method doesn't actually exist yet, so `Swift` will complain. However, it's going to exist in just a moment. This `askQuestion()` method will be where we choose some flags from the array, put them in the buttons, then prompt wait for the user to select the correct one.
Add this new method underneath `viewDidLoad()`:

```
func askQuestion() {
```

```
    button1.setImage(UIImage(named: countries[0]),
forState: .Normal)
    button2.setImage(UIImage(named: countries[1]),
forState: .Normal)
    button3.setImage(UIImage(named: countries[2]),
forState: .Normal)
}
```

The first line is easy enough: we're declaring a new method called `askQuestion()`, and it takes no parameters. The next three use `UIImage(named:)` and read from an array by position, both of which we used in `project 1`, so that bit isn't new either. However, the rest of those lines is new, and shows off two things:

- `button1.setImage()` – assigns a UIImage to the button. We have the US flag in there right now, but this will change it when `askQuestion()` is called.
- `forState: .Normal` – The `setImage()` method takes a second parameter: which state of the button should be changed? We're specifying `.Normal`, which means "the standard state of the button".

That `.Normal` is hiding two more complexities, both of which you need to understand. First, this is a data type called an "enum", short for enumeration. If you imagine that buttons have three states, normal, highlighted and disabled. We could represent those three states as 0, 1 and 2, but it would be hard to program – was 1 disabled, or was it highlighted?

Enums solve this problem by letting us use meaningful names for things. In place of 0 we can write `.Normal`, and in place of 1 we can write `.Disabled`, and so on.

This makes code easier to write and easier to read, without having any performance impact. Perfect!

The other thing `.Normal` is hiding is that period at the start: why is it `.Normal` and not just `Normal`? Well, we're setting the title of a `UIButton` here, so we need to specify a button state for it. But `.Normal` might apply to any number of other things, so how does `Swift` know we mean a normal button state?

The actual data type `setImage()` expects is called `UIControlState`, and `Swift` is being clever: it knows to expect a `UIControlState` value in there, so when we write `.Normal` it understands that to mean "the Normal value of `UIControlState`". You could, if you wanted, write the line out in full as `UIControlState.Normal`, but that's not common.
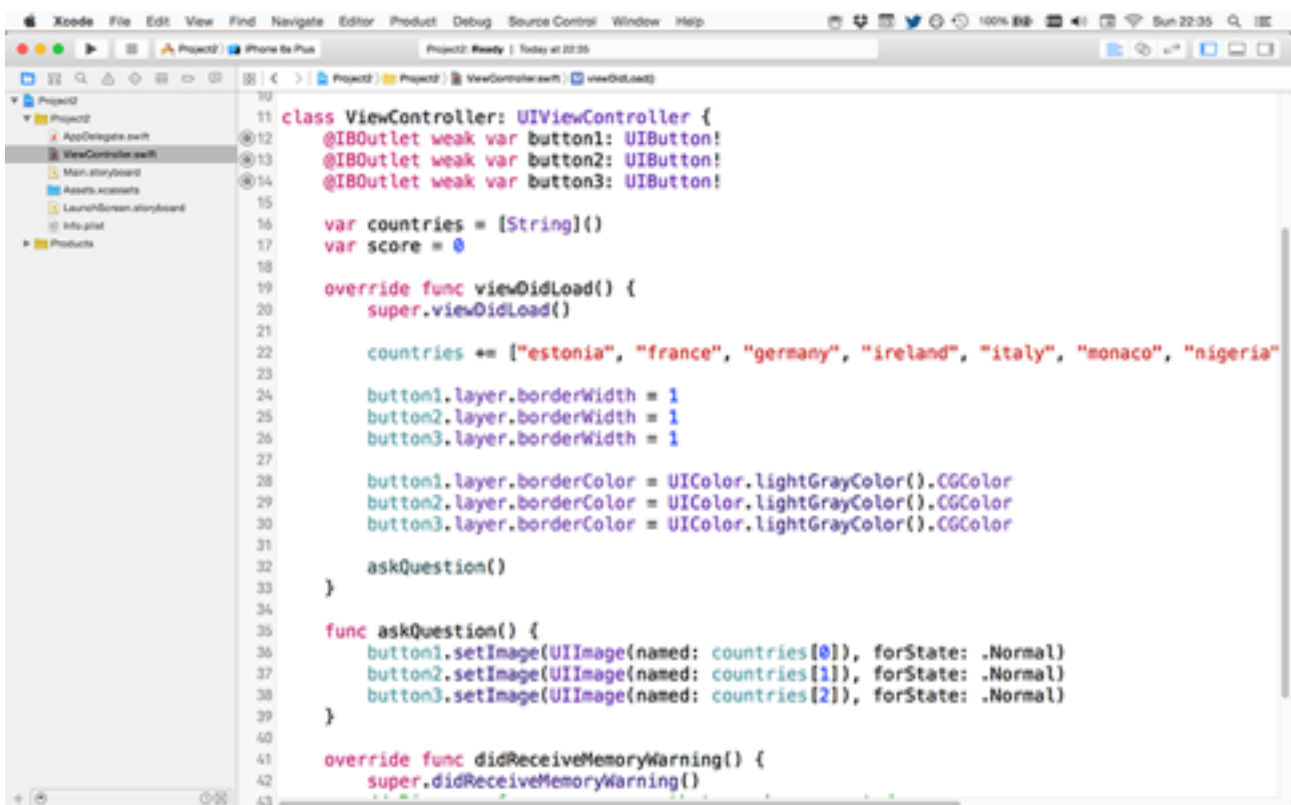
At this point, the game is in a fit state to run, so press `Cmd + R` now to launch the `iPhone Simulator` and give it a try. You'll notice two problems: 1) we're showing

the `Estonian` and `French` flags, both of which have white in them so it's hard to tell whether they are flags or just blocks of color, and 2) the "game" isn't much fun, because it's always the same three flags!

The second problem is going to wait a few minutes, but we can fix the first problem now. One of the many powerful things about views in `iOS` is that they are backed by what's called a `CALayer`, which is a `Core Animation` data type responsible for managing the way your view looks.

Conceptually, `CALayer` sits beneath all your `UIViews` (that's the parent of `UIButton`, `UILabel`, and so on), so it's like an exposed underbelly giving you lots of options for modifying the appearance of views, as long as you don't mind dealing with a little more complexity. We're going to use one of these appearance options now: `borderWidth`.

The `Estonian` flag has a white stripe at the bottom, and because our view controller has a white background that whole stripe is invisible. We can fix that by giving the layer of our buttons a `borderWidth` of 1, which will draw a one point black line around them. Put these three lines in `viewDidLoad()` directly before it calls `askQuestion()`:

```
button1.layer.borderWidth = 1
button2.layer.borderWidth = 1
button3.layer.borderWidth = 1
```

Remember how points and pixels are different things? In this case, our border will be 1 pixel on non-retina devices, 2 pixels on retina devices, and 3 on retina HD devices. Thanks to the automatic point-to-pixel multiplication, this border will visually appear to have more or less the same thickness on all devices.

By default, the border of CALayer is black, but you can change that if you want by using the UIColor data type. I said that CALayer brings with it a little more complexity, and here's where it starts to be visible: CALayer sits at a lower technical level than UIButton, which means it doesn't understand what a UIColor is. UIButton knows what a UIColor is because they are both at the same technical level, but CALayer is below UIButton, so UIColor is a mystery.

Don't despair, though: CALayer has its own way of setting colors called CGColor, which comes from Apple's Core Graphics framework. This, like CALayer, is at a lower level than UIButton, so the two can talk happily – again, as long as you're happy with the extra complexity. Even better, UIColor (which sits above CGColor) is able to convert to and from CGColor easily, which means you don't need to worry about the complexity – hurray!

So, so, so: let's put all that together into some code that changes the border color using UIColor and CGColor together. Put these three just below the three borderWidth lines in viewDidLoad():

```
button1.layer.borderColor = UIColor.lightGrayColor().CGColor
button2.layer.borderColor = UIColor.lightGrayColor().CGColor
button3.layer.borderColor = UIColor.lightGrayColor().CGColor
```

As you can see, UIColor has a method called lightGrayColor that returns (shock!) a UIColor instance that represents a light gray color. But we can't put a UIColor into the borderColor property because it belongs to a CALayer, which doesn't understand what a UIColor is. So, we add .CGColor to the end of the UIColor to have it automagically converted to a CGColor. Perfect.

If lightGrayColor doesn't interest you, you can create your own color like this:

```
UIColor(red: 1.0, green: 0.6, blue: 0.2, alpha: 1.0).CGColor
```

You need to specify four values: red, green, blue and alpha, each of which should range from `0` (none of that color) to `1.0` (all of that color). The code above generates an orange color, then converts it to a `CGColor` so it can be assigned to a `CALayer's borderColor` property.

That's enough with the styling, I think. Time to make this into a real game…

Guess which flag: Random numbers

Our current code chooses the first three items in the countries array, and places them into the three buttons on our view controller. This is fine to begin with, but really we need to choose random countries each time. There are two ways of doing this:

- Pick three random numbers, and use those to read the flags from the array.
- Shuffle up the order of the array, then pick the first three items.

Both approaches are valid, but the former takes a little more work because we need to ensure that all three numbers are different – it'd be even less fun if all three flags were the `French` flag!

The second approach is easy to do, but there's a catch: we're going to use a new `iOS` library called `GameplayKit`. You see, randomness is a complicated thing, and it's easy to write some code that you think randomises an array perfectly when actually it generates a predictable sequence. As a result, we're going to use a new library in `iOS 9` called `GameplayKit` that does all this hard work for us.

Now, you might think, "why would I want to use something called GameplayKit for apps?" But the simple answer is: because it's there, because all devices have it built right in, and because it's available in all your projects, whether games or apps. GameplayKit can do a lot more than just shuffling an array, but we'll get on to that much later.

For now, look at the top of your ViewController.swift file and you'll find a line of code that says import UIKit. Just before that, add this new line:

```
import GameplayKit
```

With that done, we can start using the functionality given to us by GameplayKit. At the start of the askQuestion() method, just before you call the first setImage() method, add this line of code:

```
countries =
GKRandomSource.sharedRandom().arrayByShufflingObjectsInArray(
countries) as! [String]
```

That will automatically randomise the order of the countries in the array, meaning that countries[0], countries[1] and countries[2] will refer to different flags each time the askQuestion() method is called. To try it out, press Cmd + R to run your program a few times to see different flags each time.

The next step is to track which answer should be the correct one, and to do that we're going to create a new property for our view controller called correctAnswer. Put this near the top, just above var score = 0:

```
var correctAnswer = 0
```

This gives us a new integer property that will store whether it's flag `0, 1` or 2 that holds the correct answer.

To actually choose which should be the right answer requires using `GameplayKit` again, because we need to choose a random number for the correct answer. `GameplayKit` has a special method for this called `nextIntWithUpperBound()`, which lets you specify a number as your "`upper bound`" – i.e., the cap for the numbers to generate. `GameplayKit` will then return a number between 0 and one less than your upper bound, so if you want a number that could be 0, 1 or 2 you specify an upper bound of 3.

Putting all this together, to generate a random number between 0 and 2 inclusive you need to put this line just below the three `setImage()` calls in `askQuestion()`:

```
correctAnswer =
GKRandomSource.sharedRandom().nextIntWithUpperBound(3)
```

Now that we have the correct answer, we just need to put its text into the navigation bar. This can be done by using the title property of our view controller, but we need to add one more thing: we don't want to write "`france`" or "`uk`" in the navigation bar, because it looks ugly. We could capitalise the first letter, and that would work great for `France`, `Germany`, and so on, but it would look poor for `Us` and `Uk`, which should be US and UK.

The solution here is simple: uppercase the entire string. This is done using the `uppercaseString` property of any string, so all we need to do is read the string out from the countries array at the position of `correctAnswer`, then uppercase it.

Add this to the end of the `askQuestion()` method, just after `correctAnswer` is set:

```
title = countries[correctAnswer].uppercaseString
```

With that done, you can run the game and it's now almost playable: you'll get three different flags each time, and the flag the player needs to tap on will have its name shown at the top.
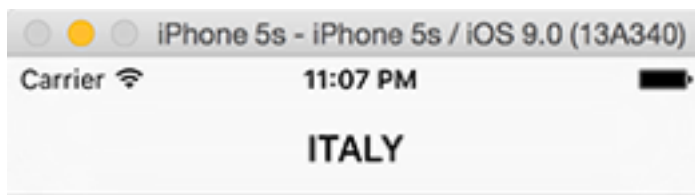
Of course, there's one piece missing: the user can tap on the flag buttons, but it doesn't actually do anything. Let's fix that…
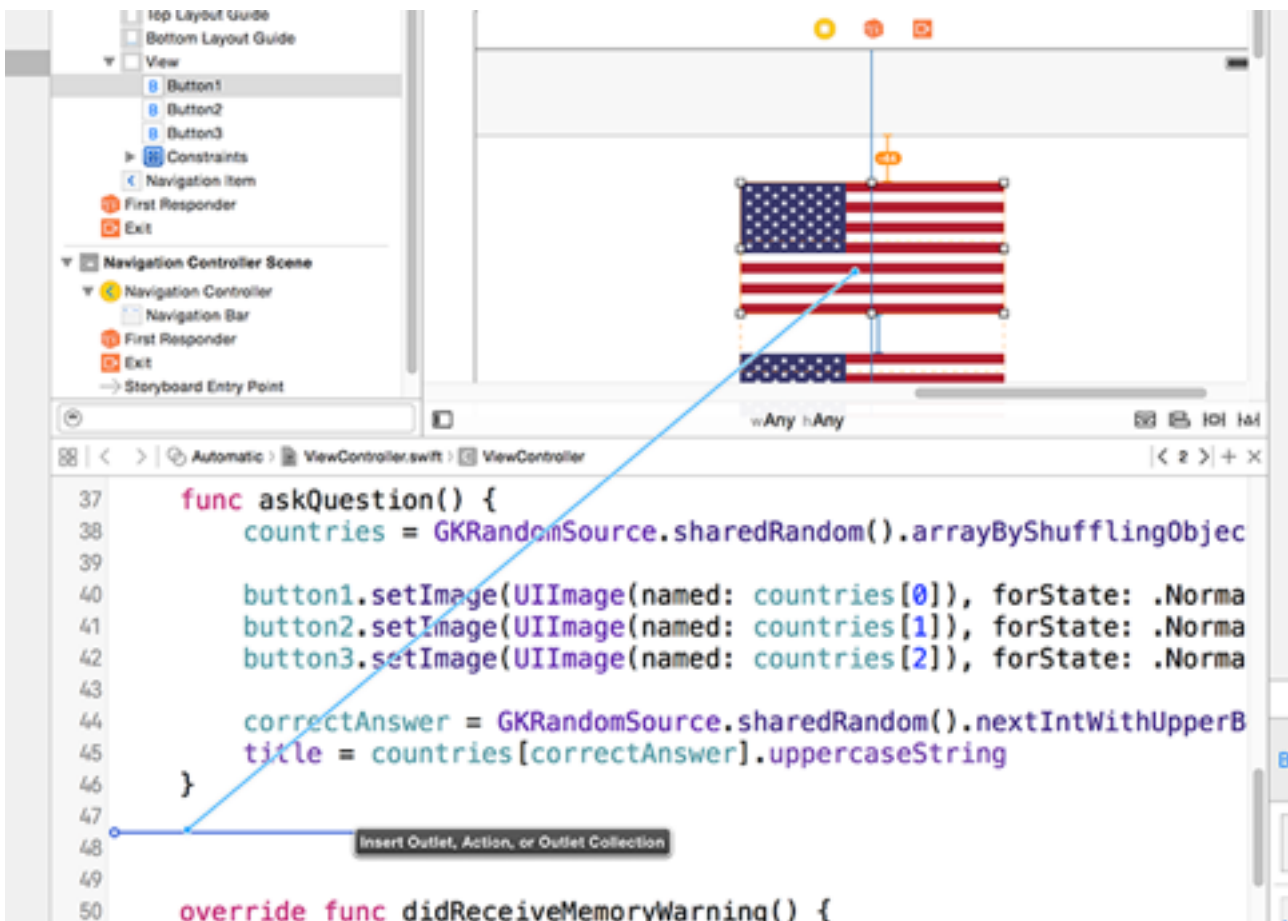
From outlets to actions: IBAction and string interpolation

I said we'd return to `Interface Builder`, and now the time has come: we're going to connect the "tap" action of our `UIButtons` to some code. So, select `Main.storyboard`, then change to the assistant editor so you can see the code alongside the layout.

Warning: please read the following text very carefully. In my haste, I screw this up all the time, and I don't want it to confuse you!

Select the first button, then `Ctrl + drag` from it down to the space in your code immediately after the end of the `askQuestion()` method. If you're doing it correctly, you should see a tooltip saying, "`Insert Outlet, Action, or Outlet Collection`". When you let go, you'll see the same popup you normally see when creating outlets, but here's the catch: don't choose outlet.


```
37    func askQuestion() {
38        countries = GKRandomSource.sharedRandom().arrayByShufflingObjec
39
40        button1.setImage(UIImage(named: countries[0]), forState: .Norma
41        button2.setImage(UIImage(named: countries[1]), forState: .Norma
42        button3.setImage(UIImage(named: countries[2]), forState: .Norma
43
44        correctAnswer = GKRandomSource.sharedRandom().nextIntWithUpperB
45        title = countries[correctAnswer].uppercaseString
46    }
47
48
49
50    override func didReceiveMemoryWarning() {
```

That's right: where it says "Connection: Outlet" at the top of the popup, I want you to change that to be Action. If you choose Outlet here (which I do all too often because I'm in a rush), you'll make problems for yourself!

When you choose Action rather than Outlet, the popup changes a little. You'll still get asked for a name, but now you'll see an Event field, and the Type field has changed from UIButton to AnyObject. Please change Type back to UIButton, then enter buttonTapped for the name, and click Connect.

Here's what Xcode will write for you:

```
@IBAction func buttonTapped(sender: UIButton) {
}
```

…and again, notice the gray circle with a ring around it on the left, signifying this has a connection in Interface Builder.

Before we look at what this is doing, I want you to do make two more connections. This time it's a bit different, because we're connecting the other two flag buttons to the same buttonTapped() method. To do that, select each of the remaining two buttons, then Ctrl-drag onto the buttonTapped() method that was just created. The whole method will turn blue signifying that it's going to be connected, so you can just let go to make it happen. If the method flashes after you let go, it means the connection was made.

So, what do we have? Well, we have a single method called buttonTapped(), which is connected to all three UIButtons. The event used for the attachment is called TouchUpInside, which is the iOS way of saying, "the user touched this button, then released their finger while they were still over it" – i.e., the button was tapped.

Again, Xcode has inserted an attribute to the start of this line so it knows that this is relevant to `Interface Builder`, and this time it's @IBAction. @IBAction is similar to @IBOutlet, but goes the other way: @IBOutlet is a way of connecting code to storyboard layouts, and @IBAction is a way of making storyboard layouts trigger code.

This method takes one parameter, called sender. It's of type `UIButton` because we know that's what will be calling the method. And this is important: all three buttons are calling the same method, so it's important we know which button was tapped so we can judge whether the answer was correct.

But how do we know whether the correct button was tapped? Right now, all the buttons look the same, but behind the scenes all views have a special identifying number that we can set, called its Tag. This can be any number you want, so we're going to give our buttons the numbers 0, 1 and 2. This isn't a coincidence: our code is already set to put flags 0, 1 and 2 into those buttons, so if we give them the same tags we know exactly what flag was tapped.
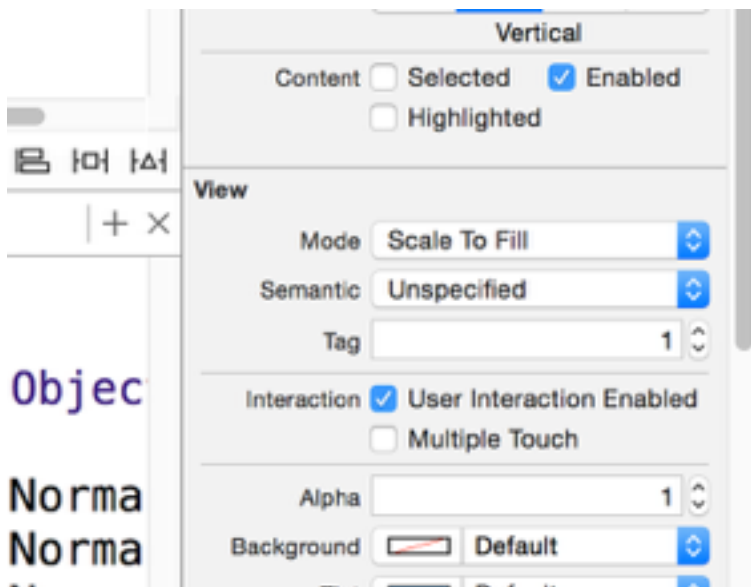
Select the second flag (not the first one!), then look in the attributes inspector (Alt + Cmd + 4) for the input box marked Tag. You'll need to scroll down, because `UIButtons` have lots of properties to work with! Once you find it (it's about two-thirds of the way down, just above the color and alpha properties), make sure it's set to 1.

Now choose the third flag and set its tag to be 2. We don't need to change the tag of the first flag because 0 is the default.

We're done with `Interface Builder` for now, so go back to the standard editor and select `ViewController.swift` — it's time to finish up by filling in the contents of the `buttonTapped()` method.

This method needs to do three things:

1. Check whether the answer was correct.
2. Adjust the player's score up or down.
3. Show a message telling them what their new score is.

The first task is quite simple, because each button has a tag matching its position in the array, and we stored the position of the correct answer in the `correctAnswer` variable. So, the answer is correct if `sender.tag` is equal to `correctAnswer`.

The second task is also simple, because you've already met the += operator that adds to a value. We'll be using that and its counterpart, −=, to add or subtract `score` as needed.

The third task is more complicated, so we're going to come to it in a minute. Suffice to say it introduces a new data type that will show a message window to the user with a title and their current score.

Put this code into the `buttonTapped()` method:

```
var title: String
```

```
if sender.tag == correctAnswer {
    title = "Correct"
    score += 1
} else {
    title = "Wrong"
    score -= 1
}
```

There are two new things here:

1. We're using the == operator. This is the equality operator, and checks if the value on the left matches the value on the right. In this, it will be true if the tag of the button that was tapped equals the correctAnswer variable we saved in askQuestion().
2. We have an else statement. When you write any if condition, you open a brace (curly bracket), write some code, then close the brace, and that code will be executed if the condition evaluates to true. But you can also give Swift some code that will be executed if the condition evaluates to false, and that's the "else" block. Here, we set one title if the answer was correct, and a different title if it was wrong.

Now for the tough bit: we're going to use a new data type called UIAlertController(). This is new in iOS 8, and is used to show an alert with options to the user. To make this work you're going to need to learn three new things, so let's cover them up front before piecing them all together.

The first thing to learn is called string interpolation. This is a Swift feature that lets you put variables and constants directly inside strings, and it will replace them with

their current value when the code is executed. Right now, we have an integer variable called score, so we could put that into a string like this:

```
let mytext = "Your score is \(score)."
```

If the score was 10, that would read "Your score is 10". As you can see, you just write \(, then your variable name, then a closing) and you're done. Swift can do all sorts of string interpolation, but we'll leave it there for now.

The second thing to learn is called a closure. This is a special kind of code block that can be used like a variable — Swift literally takes a copy of the block of code so that it can be called later. Swift also copies anything referenced inside the closure, so you need to be careful how you use them. We're going to be using closures extensively later, but for now we're going to take two shortcuts.

That's all the upfront learning done, so let's take a look at the actual code. Enter this just before the end of the buttonTapped() method:

```
let ac = UIAlertController(title: title, message: "Your score
is \(score).", preferredStyle: .Alert)
ac.addAction(UIAlertAction(title: "Continue",
style: .Default, handler: askQuestion))
presentViewController(ac, animated: true, completion: nil)
```

The title variable was set in our if statement to be either "correct" or "wrong", and you've already learned about string interpolation, so the first new thing there is the .Alert parameter being used for preferredStyle. If you remember using .Normal for UIButton's setImage() method, you should recognise this is as an enum, or enumeration.

In the case of `UIAlertController()`, there are two kinds of style: `.Alert`, which pops up a message box over the center of the screen, and `.ActionSheet`, which slides options up from the bottom. They are similar, but `Apple` recommends you use `.Alert` when telling users about a situation change, and `.ActionSheet` when asking them to choose from a set of options.

The second line uses the `UIAlertAction` data type to add a button to the alert that says "`Continue`", and gives it the style "`Default`". There are three possible styles: `.Default,` `.Cancel`, and `.Destructive`. What these look like depends on `iOS`, but it's important you use them appropriately because they provide subtle hints to users.

The sting in the tail is at the end of that line: `handler: askQuestion`. The handler parameter is looking for a closure, which is some code that it can execute when the button is tapped. You can write custom code in there if you want, but in our case we want the game to continue when the button is tapped, so we pass in `askQuestion` so that `iOS` will call our `askQuestion()` method.

Warning: We must use `askQuestion` and not `askQuestion()`. If you use the former, it means "`here's the name of the method to run`", but if you use the latter it means "`run the askQuestion() method now, and it will tell you the name of the method to run`".

There are many good reasons to use closures, but in the example here just passing in `askQuestion()` is a neat shortcut – although it does break something that we'll need to fix in a moment.

The final line calls `presentViewController()`, which takes three parameters: a view controller to present, whether to animate the presentation, and another closure that should be executed when the presentation animation has finished. We send

our `UIAlertController` for the first parameter, true for the second (animation is always nice!), and take a different shortcut for the closure: we give it nil. This means "`do nothing`", and you'll be using it a lot with closures!

Before the code completes, there's a problem, and Xcode is probably telling you what it is: "`UIAlertAction! is not a subtype of ()`". This is a good example of Swift's terrible error messages, and it's something I'm afraid you'll have to get used to. What it means to say is that using a method for this closure is fine, but Swift wants the method to accept a `UIAlertAction` parameter saying which `UIAlertAction` was tapped.

To make this problem go away, we need to change the way the `askQuestion()` method is defined. So, scroll up and change `askQuestion()` from this:

```
func askQuestion() {
```
…to this:
```
func askQuestion(action: UIAlertAction!) {
```

That will fix the `UIAlertAction` error. However, it will introduce another problem: when the app first runs, we call `askQuestion()` inside `viewDidLoad()`, and we don't pass it a parameter. There are two ways to fix this:

1. When using `askQuestion()` in `viewDidLoad()`, we could send it the parameter nil to mean "`there is no UIAlertAction for this`".
2. We could redefine `askQuestion()` so that the action has a default parameter of nil, meaning that if it isn't specified it automatically becomes nil.

There's no right or wrong answer here, so I'll show you both and you can choose. If you want to go with the first option, change the `askQuestion()` call in `viewDidLoad()` to this:

```
askQuestion(nil)
```

And if you want to go with the second option, change the `askQuestion()` method definition to this:

```
func askQuestion(action: UIAlertAction! = nil) {
```

Now, go ahead and run your program in the simulator, because it's done!

Wrap up

This is another relatively simple project, but it's given you the chance to go over some concepts in a little more detail, while also cramming in a few more concepts alongside. Going over things again in a different way is always helpful to learning, so I hope you don't view this game (or any of the games we'll make in this series!) as a waste of time.

Yes, in this project we revisited Interface Builder, Auto Layout, outlets and other things, but at the same time you've learned about @2x and @3x images, asset catalogs, integers, doubles, floats, operators, UIButton, enums, CALayer, UIColor, random numbers, actions, string interpolation, UIAlertController, and more. And you have a finished game too!

If you feel like working on this app some more, see if you can figure out how to place a `UILabel` onto the view controller and connect it via an outlet, then show the player's score in there rather than in a `UIAlertController`. You'll need to use your label's text property along with string interpolation to make it work. Good luck!