# Project 2
Guess the Flag

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

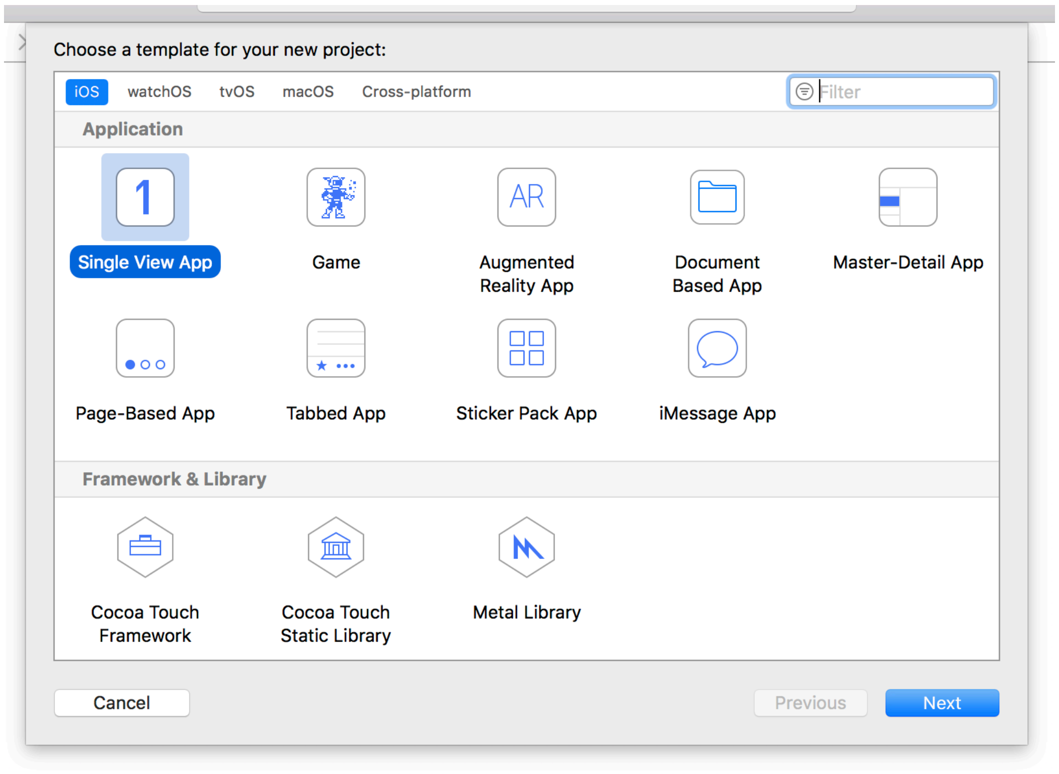
## Setting up

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.

**Warning:** 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 App from the list and click Next. For Product Name enter "Project2", then make sure you have Swift selected for language. Now click Next again and you'll be asked where you want to save the project – your desktop is fine.


Choose a template for your new project:
iOS
watchOS
tvOS
macOS
Cross-platform
Filter
Application
Single View App
Game
Augmented Reality App
Document Based App
Master-Detail App
Page-Based App
Tabbed App
Sticker Pack App
iMessage App
Framework & Library
Cocoa Touch Framework
Cocoa Touch Static Library
Metal Library
Cancel
Previous
Next

## Designing your layout

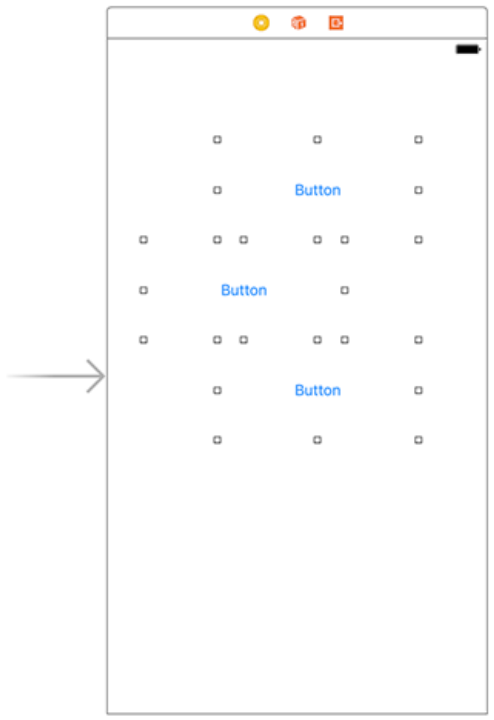
When working on my own projects, I find designing the user interface the easiest way to begin any project – it's fun, it's immediately clear whether your idea is feasible, and it also forces you to think about user journeys while you work. This project isn't complicated, but still Interface Builder is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

Just as in project 1, the Single View App 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. We need to add one, just like we did with the previous project.

We covered a *lot* in project 1, so you've probably forgotten how to do this, but that's OK: Single View App 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.

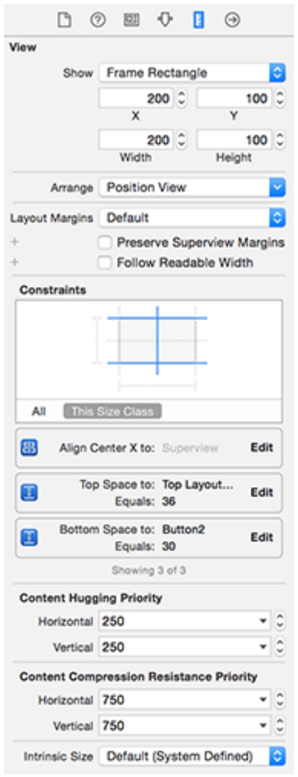
With the new navigation controller in place, scroll so you can see our empty view controller again, and draw out three **`UIButton`**s onto the canvas. This is a new view type, but as you might imagine it's just a button that users can tap. Each of them should be 200 wide by 100 high. You can set these 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 or by going to the View menu and choosing Utilities > Show Size Inspector. 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.


View

Show: Frame Rectangle

X: 200 Y: 100

Width: 200 Height: 100

Arrange: Position View

Layout Margins: Default

Preserve Superview Margins

Follow Readable Width

Constraints

All | This Size Class

Align Center X to: Superview — Edit

Top Space to: Top Layout... Equals: 36 — Edit

Bottom Space to: Button2 Equals: 30 — Edit

Showing 3 of 3

Content Hugging Priority

Horizontal: 250

Vertical: 250

Content Compression Resistance Priority

Horizontal: 750

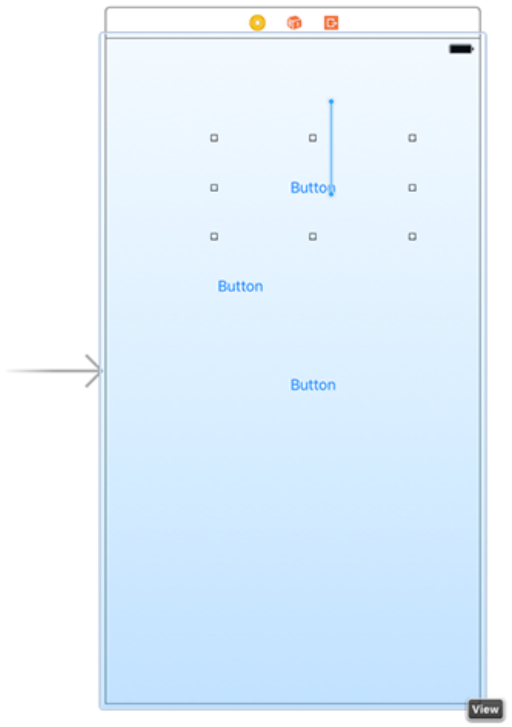
Vertical: 750

Intrinsic Size: Default (System Defined)

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.

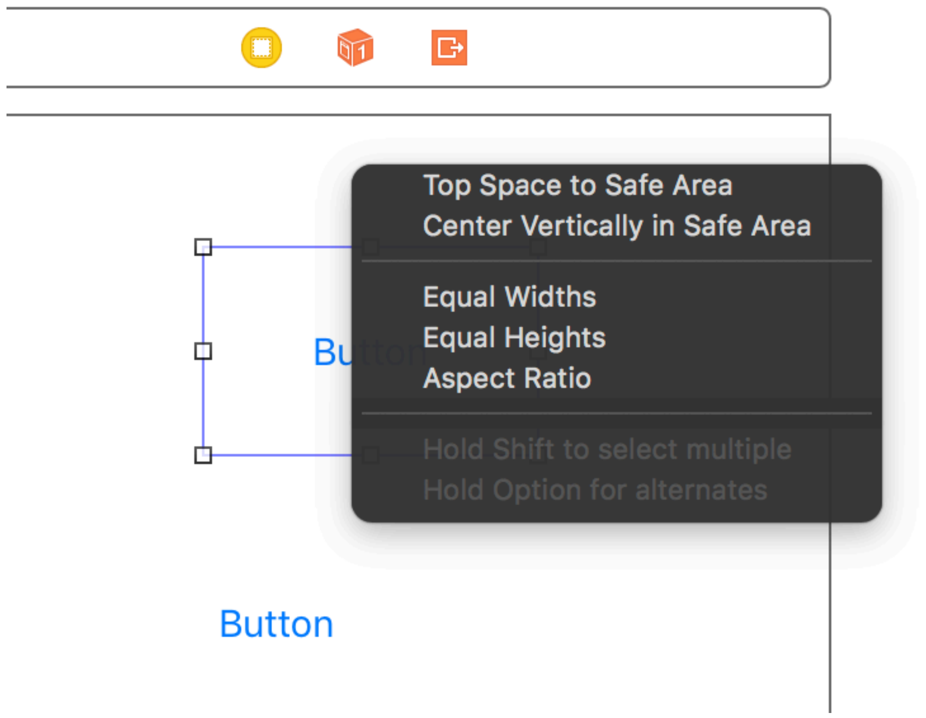
We're going to create our Auto Layout rules differently from in Project 1. This is not because one way is better than another, instead just so that you can see the various possibilities and decide which one suits you best.

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 Safe Area“ and "Center Horizontally in Safe Area.”

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 Safe Area“ and "Center Horizontally in Safe Area.” When that’s done, click Add Constraints to confirm that selection.

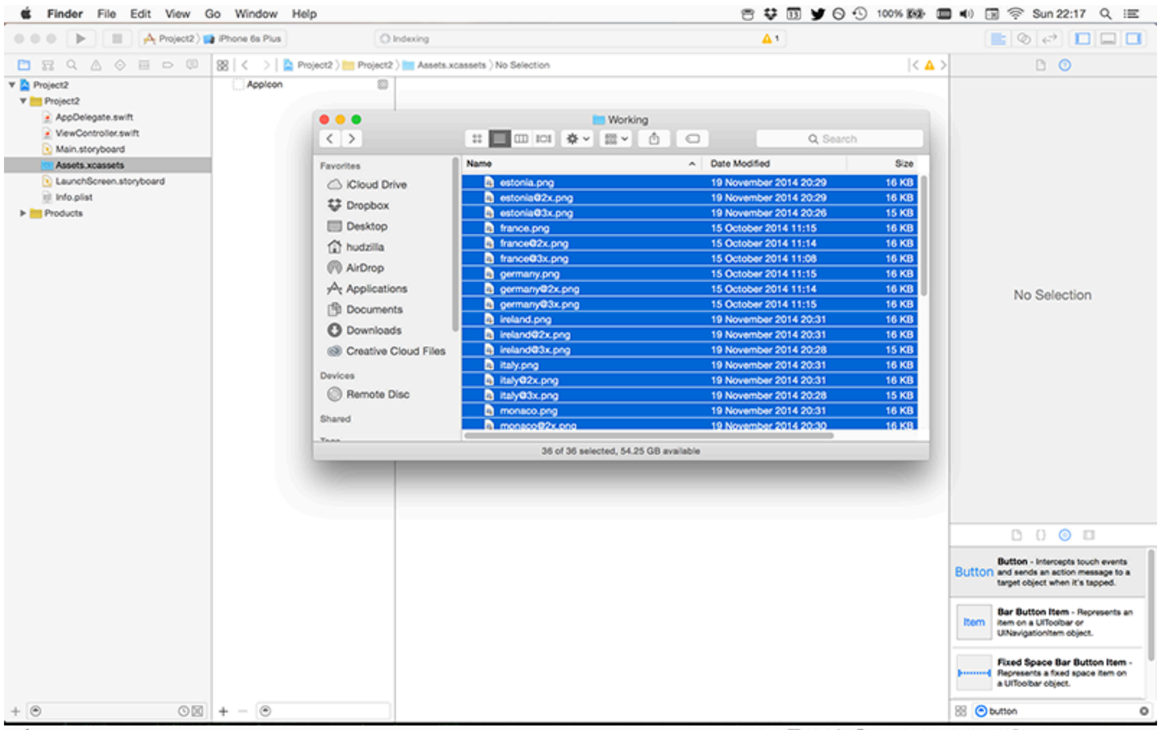

Top Space to Safe Area
Center Vertically in Safe Area
Equal Widths
Equal Heights
Aspect Ratio
Hold Shift to select multiple
Hold Option for alternates
Button
Button

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 optimized ways of importing and using images in iOS projects, and are just as easy to use as a content folder.

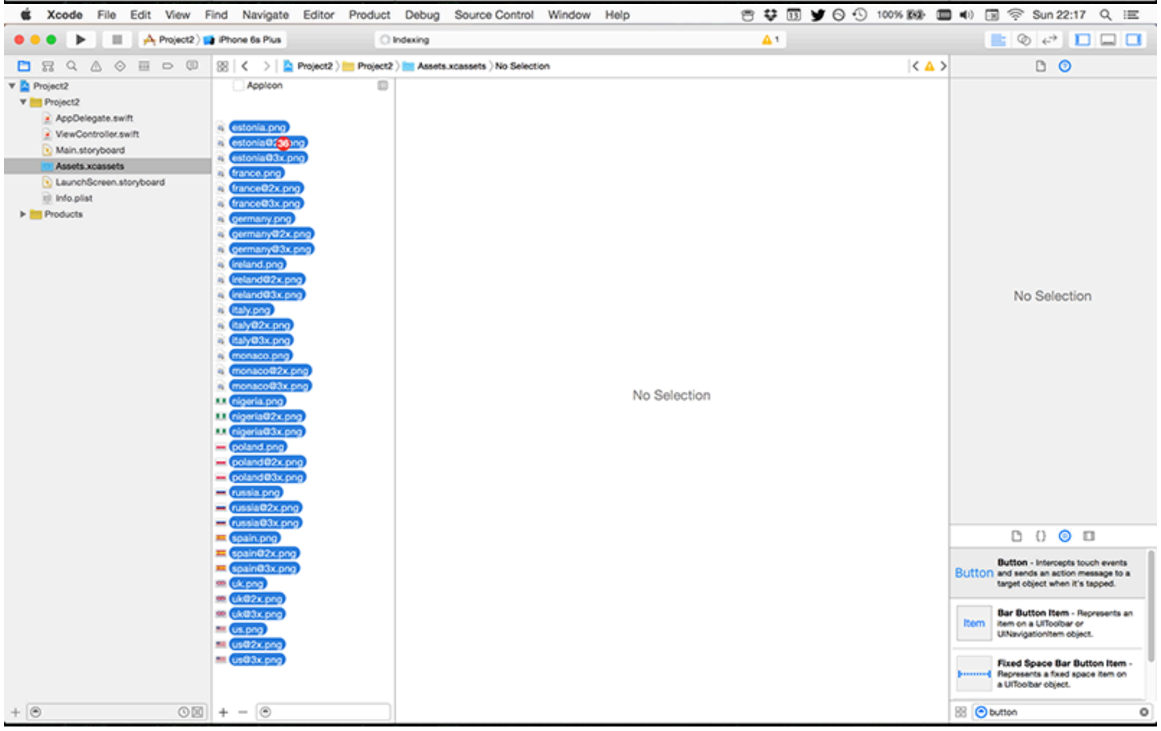
In your Xcode project, select the file called Assets.xcassets. This isn't really a file, instead it's our default Xcode asset catalog. If you haven't already downloaded the files for this project, please do so now from GitHub.

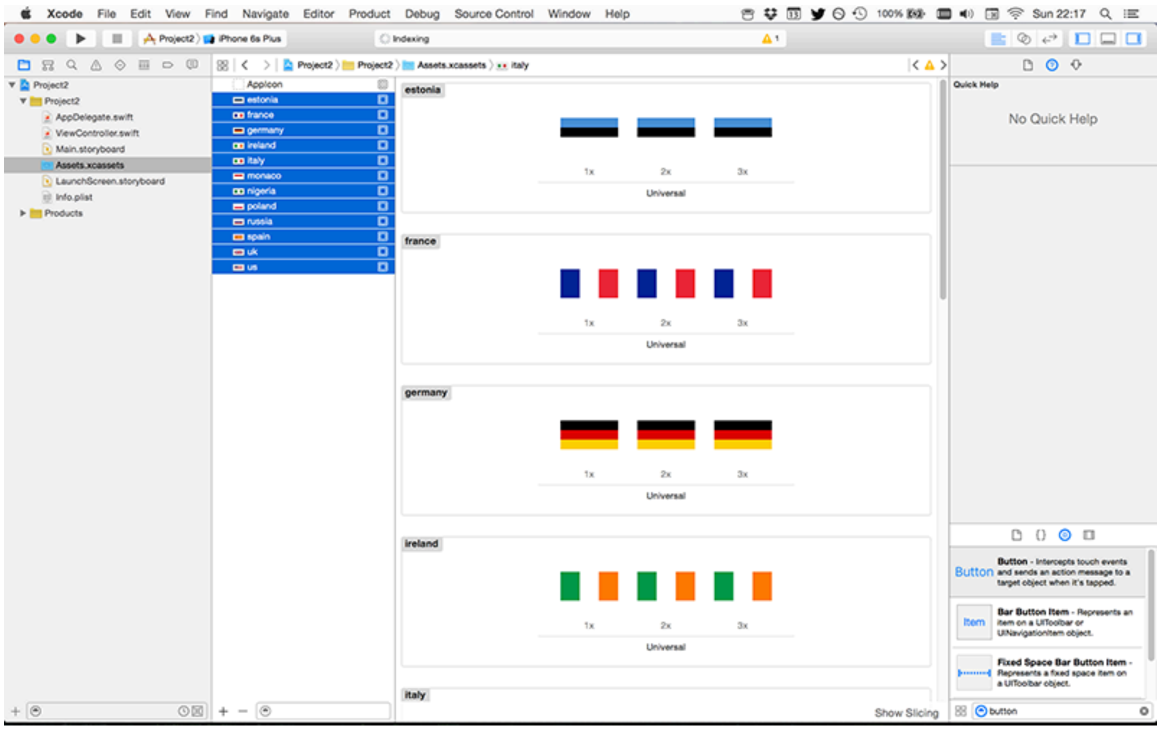
Select all 36 flag pictures from the project files, and drag them into the Xcode window to beneath where it says "AppIcon" in our asset catalog. This will create 12 new entries in the asset catalog, one for each country.



| Name | Date Modified | Size |
|---|---|---|
| estonia.png | 19 November 2014 20:29 | 16 KB |
| estonia@2x.png | 19 November 2014 20:29 | 16 KB |
| estonia@3x.png | 19 November 2014 20:26 | 15 KB |
| france.png | 15 October 2014 11:15 | 16 KB |
| france@2x.png | 15 October 2014 11:14 | 16 KB |
| france@3x.png | 15 October 2014 11:08 | 16 KB |
| germany.png | 15 October 2014 11:15 | 16 KB |
| germany@2x.png | 15 October 2014 11:14 | 16 KB |
| germany@3x.png | 15 October 2014 11:15 | 16 KB |
| ireland.png | 19 November 2014 20:31 | 16 KB |
| ireland@2x.png | 19 November 2014 20:31 | 16 KB |
| ireland@3x.png | 19 November 2014 20:28 | 15 KB |
| italy.png | 19 November 2014 20:31 | 16 KB |
| italy@2x.png | 19 November 2014 20:31 | 16 KB |
| italy@3x.png | 19 November 2014 20:28 | 15 KB |
| monaco.png | 19 November 2014 20:31 | 16 KB |
| monaco@2x.png | 19 November 2014 20:30 | 16 KB |

36 of 36 selected, 54.25 GB available




Xcode File Edit View Find Navigate Editor Product Debug Source Control Window Help
Project2 › iPhone 6s Plus
Indexing
Project2 › Project2 › Assets.xcassets › italy
Project2
AppDelegate.swift
ViewController.swift
Main.storyboard
Assets.xcassets
LaunchScreen.storyboard
Info.plist
Products
AppIcon
estonia
france
germany
ireland
italy
monaco
nigeria
poland
russia
spain
uk
us
1x
2x
3x
Universal
Show Slicing
Quick Help
No Quick Help
Button - Intercepts touch events and sends an action message to a target object when it's tapped.
Bar Button Item - Represents an item on a UIToolbar or UINavigationItem object.
Fixed Space Bar Button Item - Represents a fixed space item on a UIToolbar object.
button

As much as I hate diversions, this one is important: iOS assets come in the sizes 2x and 3x, which are two times and three times the size of the layout you created in Interface Builder. This might seem strange, but it's a little bit of iOS magic that takes away a huge amount of work from developers.

Early iOS devices had non-retina screens. This meant a screen resolution of 320x480 pixels, and you could place things exactly where you wanted them – you asked for 10 pixels in from the left and 10 from the top, and that was what you got.

With iPhone 4, Apple introduced retina screens that had double the number of pixels as previous screens. Rather than make you design all your interfaces twice, Apple automatically switched sizes from pixels to "points" – virtual pixels. On non-retina devices, a width of 10 points became 10 pixels, but on retina devices it became 20 pixels. This meant that everything looked the same size and shape on both devices, with a single layout.

Of course, the whole point of retina screens was that the screen had more pixels, so everything looked sharper – just resizing everything to be larger wasn't enough. So, Apple took things a step further: if you create hello.png that was 200x100 in size, you could also include a file called hello@2x.png that was 400x200 in size – exactly double – and iOS would load the correct one. So, you write hello.png in your code, but iOS knows to look for and load hello@2x.png on retina devices.

More recently, Apple introduced retina HD screens that have a 3x resolution. These follow the same naming convention: hello.png is for non-retina devices, hello@2x.png for retina devices, and hello@3x for retina HD devices. You still just write “hello.png” in your code and user interfaces, and iOS does the rest.

You might think this sounds awfully heavy – why should a non-retina device have to download apps that include @2x and @3x content that it can’t show? Fortunately, the App Store uses a technology called app thinning that automatically delivers only the content each device is capable of showing – it strips out the other assets when the app is being downloaded, so there’s no space wasted.

Cunningly, as of iOS 10 no non-retina devices are supported, so if you’re supporting only iOS 10 or later devices you only need to include the @2x and @3x images. I’ve included the 1x images for this project in case you want to use it on older iOS versions too.

Now, 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. Xcode recognized these names, and arranged all the images correctly.

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".

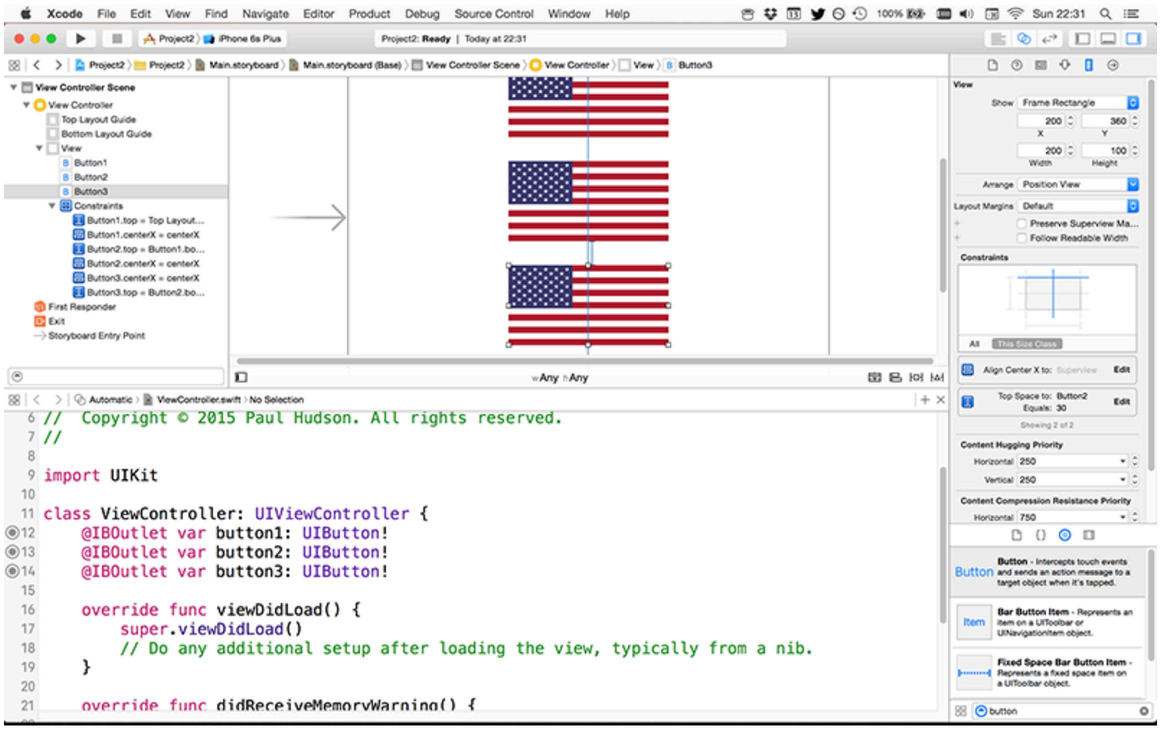
As soon as you set a picture inside the button, our constraints for the button are complete: it has a Y position because we placed a constraint, it has an X position because we're centering it horizontally, and it has a width and a height because it's reading it from the image we assigned. Go ahead and 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 Horizontally." Choose both of these. Now choose the third button and Ctrl-drag to the second button, and again choose "Vertical Spacing" and "Center Horizontally."

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: select all three image views, then press Alt+Cmd+=. If you don't like keyboard shortcuts, go to the Editor menu and choose Resolve Auto Layout Issues > Update Frames. Again, you'll see that option appears twice in the menu, but both do the same thing here so you can select either. This command will update the frames – the positions and sizes – of each image view so that it matches the Auto Layout constraints we set.

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 by pressing Alt+Cmd+Return or by going to View > Assistant Editor > Show Assistant Editor. Now Ctrl-drag from the first flag to your code in order to create an outlet called **`button1`**, then from the second flag to create **`button2`**, and from the third flag to create **`button3`**.



```
//  Copyright © 2015 Paul Hudson. All rights reserved.
//

import UIKit

class ViewController: UIViewController {
    @IBOutlet var button1: UIButton!
    @IBOutlet var button2: UIButton!
    @IBOutlet var button3: UIButton!

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

    override func didReceiveMemoryWarning() {
```

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 to 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 creates a new property called **`score`** that is set to 0.

What you're seeing here is called *type inference*. This 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 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. **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 wait for the user to select the correct one.

Add this new method underneath `viewDidLoad()`:

```
func askQuestion() {
    button1.setImage(UIImage(named: countries[0]), for: .normal)
    button2.setImage(UIImage(named: countries[1]), for: .normal)
    button3.setImage(UIImage(named: countries[2]), for: .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.

- **`for: .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."

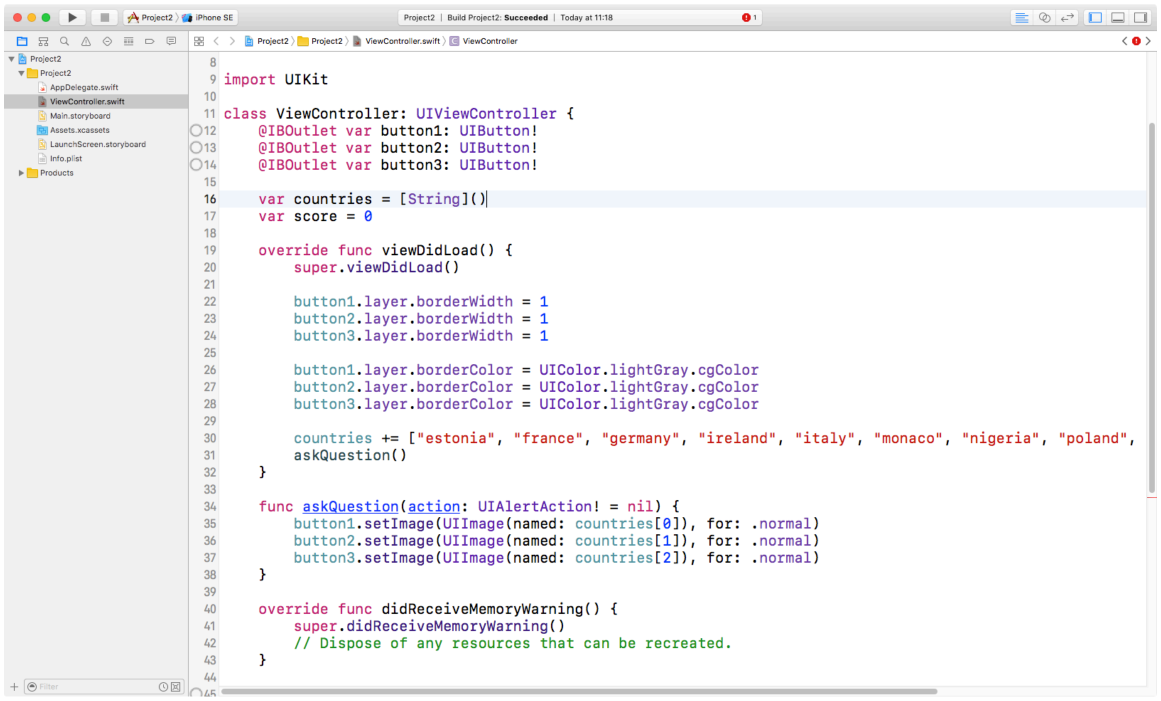
That **`.normal`** is hiding two more complexities, both of which you need to understand. First, this is being used like a data type called an "enum", short for enumeration. If you imagine that buttons have three states, normal, highlighted and disabled. We *could* represent those three states as 0, 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!

**A note for pedants:** I said **`UIControlState`** "is being used *like* a data type called an enum" rather than "*is* an enum*, because this particular example is rather murky behind the scenes. In Objective-C – the language UIKit was written in – it's an enum, but in Swift it gets mapped to a struct that just happens to be* used* like an enum, so if you want to be technically correct it's not a true enum in Swift. At this point in your Swift career there is no difference, but let's face it: "technically correct" is the best kind of correct.

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 image 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 there isn't much point.



```swift

import UIKit

class ViewController: UIViewController {
    @IBOutlet var button1: UIButton!
    @IBOutlet var button2: UIButton!
    @IBOutlet var button3: UIButton!

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

    override func viewDidLoad() {
        super.viewDidLoad()

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

        button1.layer.borderColor = UIColor.lightGray.cgColor
        button2.layer.borderColor = UIColor.lightGray.cgColor
        button3.layer.borderColor = UIColor.lightGray.cgColor

        countries += ["estonia", "france", "germany", "ireland", "italy", "monaco", "nigeria", "poland",
        askQuestion()
    }

    func askQuestion(action: UIAlertAction! = nil) {
        button1.setImage(UIImage(named: countries[0]), for: .normal)
        button2.setImage(UIImage(named: countries[1]), for: .normal)
        button3.setImage(UIImage(named: countries[2]), for: .normal)
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }

```

At this point the game is in a fit state to run, so let's give it a try.

First, select the iPhone 8 simulator by going to the Product menu and choosing Destination > iPhone 8. Now press Cmd+R now to launch the Simulator and give it a try.

You'll immediately 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
2. The "game" isn't much fun, because it's always the same three flags!

The second problem is going to have 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 **`UIView`**s (that's the parent of **`UIButton`**, **`UITableView`**, 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.lightGray.cgColor
button2.layer.borderColor = UIColor.lightGray.cgColor
```

```
button3.layer.borderColor = UIColor.lightGray.cgColor
```

As you can see, **`UIColor`** has a property called **`lightGray`** 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 **`lightGray`** 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:

1. Pick three random numbers, and use those to read the flags from the array.
2. 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 – this game would 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 an iOS framework called GameplayKit. You see, randomness is a complicated thing, and it's easy to write some code that you think randomizes an array perfectly when actually it generates a predictable sequence. As a result, we're going to use an Apple framework 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:

```swift
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:

```swift
countries =
GKRandomSource.sharedRandom().arrayByShufflingObjects(in:
countries) as! [String]
```

That will automatically randomize 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 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 **`nextInt(upperBound:)`**, 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().nextInt(upperBound: 3)
```

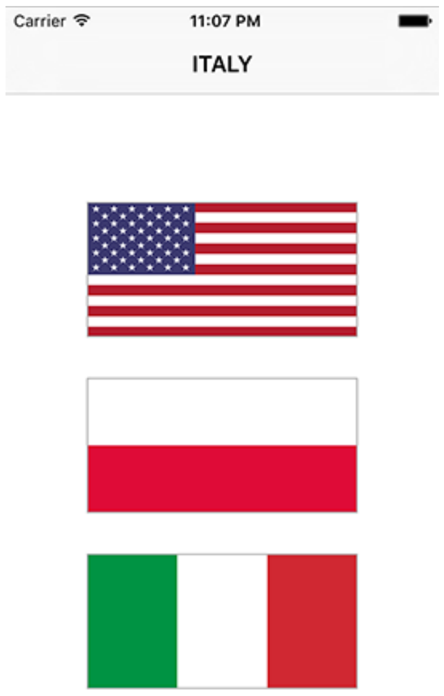
Now that we have the correct answer, we just need to put its text into the navigation bar. This can be done by using the **`title`** property of our view controller, but we need to add one more thing: we don't want to write "france" or "uk" in the navigation bar, because it looks ugly. We could capitalize 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

**`uppercased()`** method 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].uppercased()
```

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.



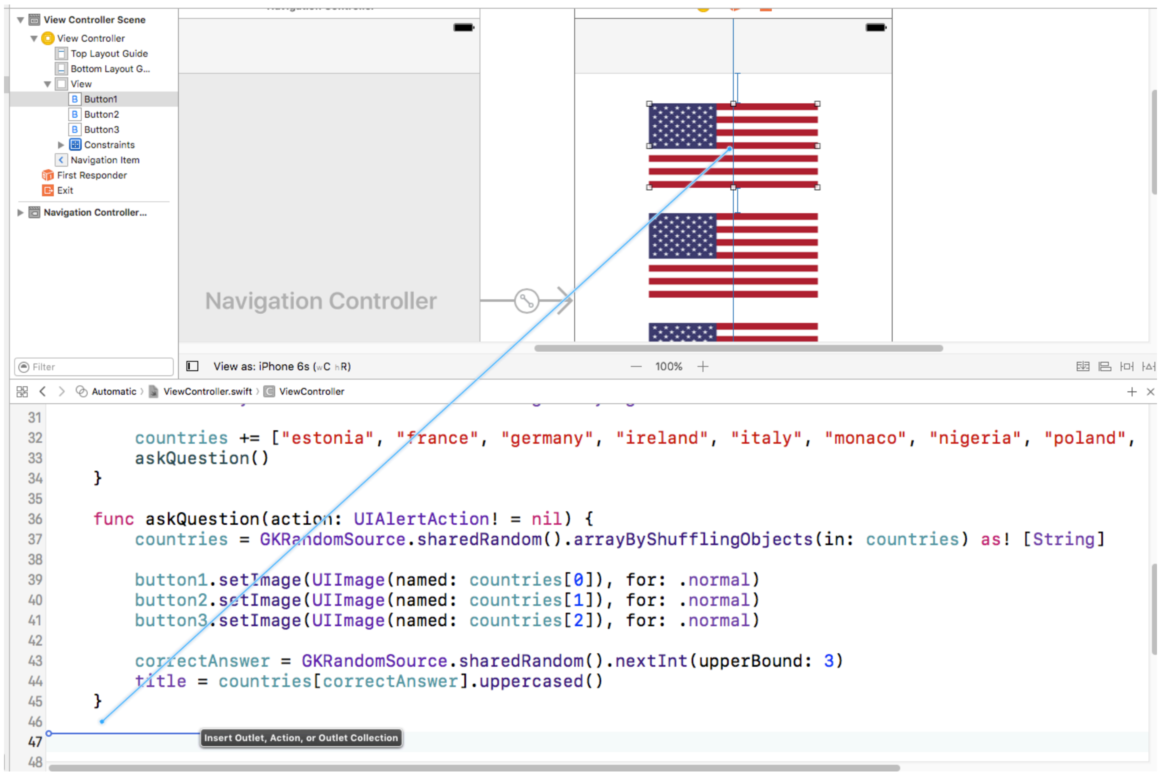
Of course, there's one piece missing: the user can tap on the flag buttons, but they don't actually *do* anything. Let's fix that…

## 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**.



```
        countries += ["estonia", "france", "germany", "ireland", "italy", "monaco", "nigeria", "poland",
        askQuestion()
    }

    func askQuestion(action: UIAlertAction! = nil) {
        countries = GKRandomSource.sharedRandom().arrayByShufflingObjects(in: countries) as! [String]

        button1.setImage(UIImage(named: countries[0]), for: .normal)
        button2.setImage(UIImage(named: countries[1]), for: .normal)
        button3.setImage(UIImage(named: countries[2]), for: .normal)

        correctAnswer = GKRandomSource.sharedRandom().nextInt(upperBound: 3)
        title = countries[correctAnswer].uppercased()
    }
```

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 cause 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 **`Any`**. 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.

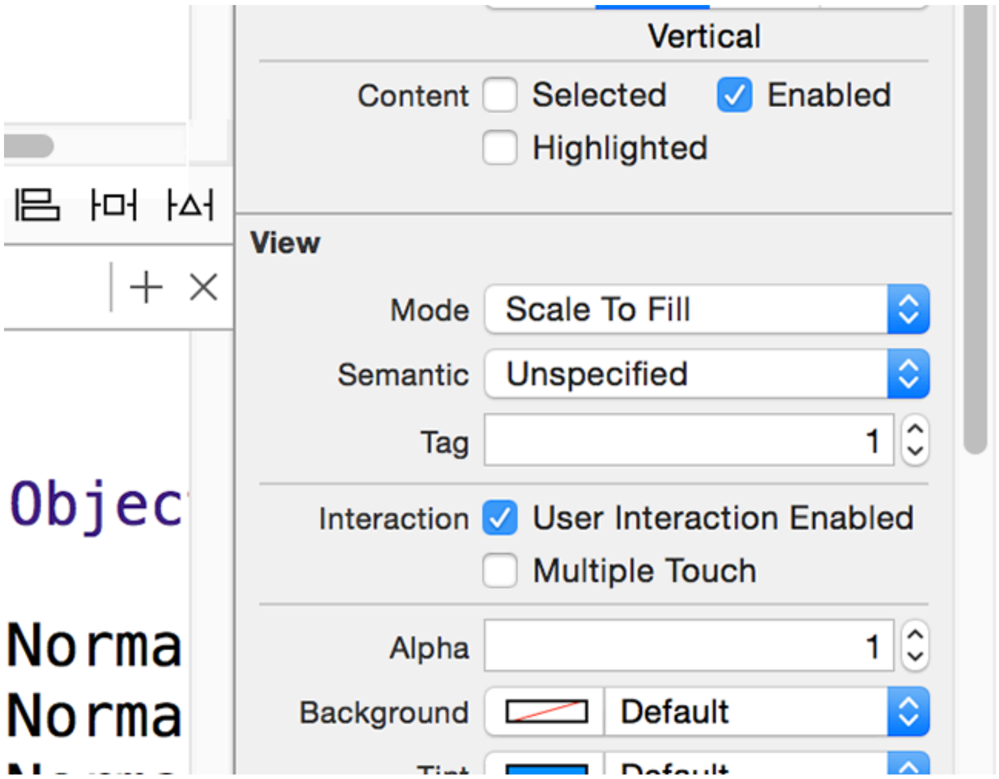
So, what do we have? Well, we have a single method called **`buttonTapped()`**, which is connected to all three **`UIButton`**s. The event used for the attachment is called **`TouchUpInside`**, which is the iOS way of saying, "the user touched this button, then released their finger while they were still 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 might need to scroll down, because `UIButton`s 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.


Vertical
Content: Selected [ ], Enabled [✓], Highlighted [ ]
View
Mode: Scale To Fill
Semantic: Unspecified
Tag: 1
Interaction: User Interaction Enabled [✓], Multiple Touch [ ]
Alpha: 1
Background: Default

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:

```swift
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. The result will be true if the tag of the button that was tapped equals the **`correctAnswer`** variable we saved in **`askQuestion()`**, or false otherwise.
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 used to show an alert with options to the user. To make this work you're going to need to learn two new things, so let's cover them up front before piecing them 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 code, so you need to be careful how you use them. We're going to be using closures extensively later, but for now we’ll 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))
present(ac, animated: true)
```

**That code will produce an error for a moment, but that’s OK.**

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 recognize 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 user interface 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 **`present()`**, which takes two parameters: a view controller to present and whether to animate the presentation. It has an optional third parameter that is another closure that should be executed when the presentation animation has finished, but we don’t need it here. We send our **`UIAlertController`** for the first parameter, and true for the second because animation is always nice.

Before the code completes, there's a problem, and Xcode is probably telling you what it is: “Cannot convert value of type ‘() -> ()’ to expected argument type ‘((UIAlertAction) -> Void)?’.”

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`**.

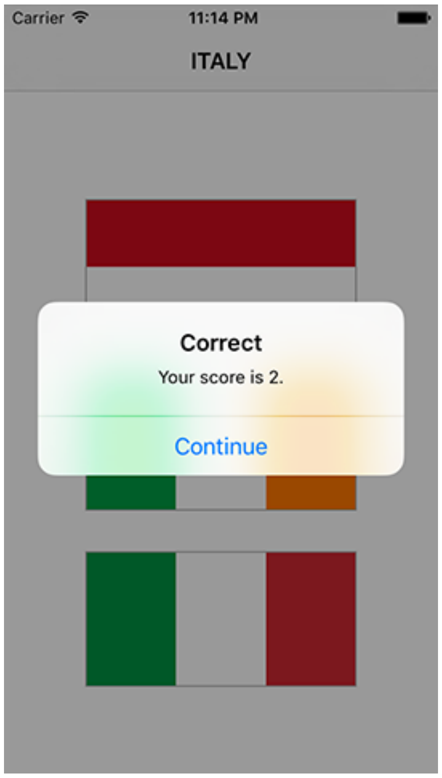
There's no right or wrong answer here, so I'll show you both and you can choose. If you want to go with the first option, change the **`askQuestion()`** call in **`viewDidLoad()`** to this:

```
askQuestion(action: 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 squeezing 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 (`+=` and `-=`), `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!