

HACKING WITH SWIFT



Practical iOS 10

COMPLETE TUTORIAL COURSE

Learn to develop apps
for iOS 10 by building
real-world projects

Paul Hudson

Contents

Preface	4
Welcome	
What's new in Swift 3?	
Happy Days	21
Setting up	
Building the user interface	
Permissions! Permissions everywhere!	
Importing into the collection view	
Recording, transcribing, and playing audio	
Searching using Spotlight	
Wrap up	
TimeShare	78
Setting up	
Building the user interface	
Working with MSMessagesAppViewController	
Selecting possible dates	
Sending and receiving messages	
Making our message look attractive	
Wrap up	
[On Hold]	122
Ride Sharing	
Polyglot	124
Setting up	
Creating a basic language app	
Building a today extension	
Animating views with UIViewPropertyAnimator	
Wrap up	
Alarmadillo	155
Setting up	
Building the user interface	

Creating model classes
Listing groups
Listing alarms
Editing alarms
Loading and saving data
Adding local notifications
Responding to user input
Wrap up

Flip 224

Setting up
Building the basic game
Creating a model of our game
Detecting legal moves
Capturing pieces
Capturing pieces
Monte Carlo strategy
Making an AI player
Wrap up

Dead Storm Rising 271

Setting up
SpriteKit tile maps
Moving around the map
Selecting game items
Moving and attacking
Controlling the game
Wrap up

Techniques 338

Animations
Units
Core Data
Raw photography
Warp Geometry

Preface

Welcome

iOS 10 is another major milestone for Apple, and introduces a variety of changes that offer you new opportunities for development. Things like speech transcription, Siri integration, iMessage apps, and more are headline features showing how Apple is opening up more of its platform to developers. But there are thousands of smaller changes: raw photo imports, SpriteKit warp geometry and tile maps, in-app Spotlight search, wide color support, as well as massive Core Data simplifications.

On top of all that, Xcode 8 shipped with Swift 3, which brings with it the biggest set of language changes so far. A lot of work has gone into making Swift 3 the last version with major breakage. That's not to say that Swift 4 won't change some things (spoiler: it will), just that Apple have gone to great lengths to try to make all the biggest breakages now, in Swift 3. That's good in the long term because it hopefully means a period of stability, but in the short term it does mean that all non-trivial Swift projects will break with Swift 3 because almost everything has changed.

The combination of a whole new iOS and what is almost a whole new Swift made this book challenging to write in places, but with enough time, determination, and coffee all things are possible!

About this book

This book is being written in 48-hour installments, so you'll be getting updates every two days until the final version is ready. Although I do my best to remove typos, some will inevitably get through as I spend more time focusing on writing new chapters than proof-reading existing ones.

It's also very likely that I'll be going back through existing work to refactor code, simplify explanations, and add features that were a bit flaky in the beta I was using. Every time I send out an email, I'll make sure to let you know what's changed and why so you can re-read any important parts.

Thank you for your patience! If you spot any mistakes or have suggestions for ways to refactor the code, do let me know by Twitter (@twostraws) or email (paul@hackingwithswift.com).

This book is explicitly *not* designed to teach you Swift. You should already have completed a large part of Hacking with Swift and/or Pro Swift, or have equivalent experience using Swift.

These projects are *not* easy, because they are not *designed* to be easy – I've written them to show you the power of what iOS 10 can do, but they are also full, standalone projects in their own right. So if you've come to this book with little Swift experience I suggest you start with Hacking with Swift first, then perhaps follow up with Pro Swift too.

Warnings

- This book is still in development, so expect things to change in the future. I will always tell you what changed and why in the emails you receive.
- I'm still in the process of taking screenshots. They will come!
- In these early stages, I'll be releasing only PDF and ePub. Once the book is finished, I'll be sending out Mobi and HTML too.
- Some things only seem to work on iOS 10 devices, and sometimes you'll get nothing, an error, or an outright crash in the simulator. Hopefully this will improve in coming betas, but for now I strongly suggest you install iOS 10 on a spare device in order to follow along in confidence.

Dedication

This book is dedicated to my 90-year-old grandmother. She doesn't quite understand what I do beyond "computers", but she does FaceTime me on her iPad to provide encouragement and support while I write. What she lacks in knowledge of programming she more than makes up for with her love and compassion, plus she makes mean potato farls.

Copyright

Swift, the Swift logo, Xcode, Instruments, Cocoa Touch, Touch ID, AirDrop, iBeacon, iPhone, iPad, Safari, App Store, Mac and OS X are trademarks of Apple Inc., registered in the U.S. and other countries.

Practical iOS 10, Pro Swift, and Hacking with Swift are copyright Paul Hudson. All rights

reserved. No part of this book or corresponding materials (such as text, images, or source code) may be reproduced or distributed by any means without prior written permission of the copyright owner.

What's new in Swift 3?

It is a truth universally acknowledged that every Swift version brings with it some improvements, some deprecations, and some new features.

Swift 1.2 brought us the `as!` typecast, let us create constants without assigning them, multiple optional unwraps with `if/let`, and the `Set` struct. Swift 2.0 brought us `do/catch`, `guard`, `defer`, `#available`, and more. Swift 2.2 deprecated C-style `for` loops, deprecated `++` and `--`, deprecated stringified selectors, deprecated `var` parameters for functions, and also added tuple comparison and compile-time Swift version checking with `#if swift()`.

All those changes were big, with Swift 2.0 probably being the largest. At least until now, because Swift 3.0 takes the language evolution to a whole new level with the most wide-ranging changes to date. Your code *will* break; that's not a maybe or a might – it *will* break. There are Xcode Fix-its in place that do most if not all of the work for you, but you should really take the time to figure out what's changed and why so you can write solid, idiomatic Swift 3 going forward.

In this book we'll be exclusively using Swift 3, and that alone will help you get into the new mindset. However, it does take time: I myself still write new methods the Swift 2.2 way, then either catch myself and refactor into Swift 3 or someone else kindly points it out. It takes time to adjust, and it's going to take a couple of years to migrate existing Swift code to the new style.

Reminder of Swift 2.2

Swift 2.2 was a relatively small release that introduced some of the first new changes agreed by the Swift Evolution community. Most of the changes it made were deprecations, but these have subsequently been removed from Swift 3.0 so it's worth going over them briefly:

1. C-style `for` loops were deprecated. That's where you wrote loops like `for var i = 1; i <= 10; i += 1`, which isn't very Swift. In their place you should use ranges for simple increments, `stride()` for complex increments, or regular fast enumeration.
2. `++` to add 1 to a variable and `--` to subtract 1 were both deprecated, so instead you need to use `+=` and `-=`.
3. `var` parameters for functions were deprecated, largely because they didn't offer much benefit and were easily confused with `inout`.

4. Stringified selectors were deprecated, and replaced with the new `#selector` syntax. This is checked at compile time, which is much safer.
5. Compile-time Swift version checking was introduced, using `#if swift()`. This wasn't useful at first, but it's much more useful with Swift 3.0 because it lets you write Swift 2.2 and 3.0 code side by side if you need to.
6. You can compare tuples up to arity 6 using `==`, which means they can have up to six elements.
7. The debug identifiers `__LINE__`, `__FUNCTION__` and so on have been renamed to `#line`, `#function`.
8. Many keywords, like `for` and `repeat`, can now be used as parameter labels for methods.

Items 1 to 4 were deprecated in Swift 2.2 and formally removed in Swift 3.0. Item 8 is interesting, though, because even though it didn't do much for Swift 2.2 it became pretty integral to Swift 3.

All function parameters have labels unless you request otherwise

Let's turn our eyes to Swift 3 now, and focus first on one of its most disruptive and influential changes: all function parameters require labels, including the first one, unless you explicitly opt out using `_`.

In Swift 2.x, methods and functions didn't require a label for their first parameter, which means the meaning of the first parameter was often encoded directly into the method name, so you would see methods named like this:

```
NSTimer.scheduledTimerWithTimeInterval(...)  
UIFont.preferredFontForTextStyle(UIFontTextStyleSubheadline)  
names.indexOf("Taylor")
```

Because Swift 3.0 forces a label on all parameters, those method names automatically get split up into two parts: the first part remains as the method name, but the second part becomes the name of the first label. So:

```
NSTimer.scheduledTimerWithTimeInterval(...)  
NSTimer.scheduledTimer(...)  
  
UIFont.preferredFontForTextStyle(UIFontTextStyleSubheadline)  
UIFont.preferredFont(forTextStyle: UIFontTextStyleSubheadline)  
  
names.indexOf("Taylor")  
names.index(of: "Taylor")
```

This has a direct impact on many of the callback methods from UIKit and other frameworks that must follow a specific name but must also be used without labels. Methods like these:

```
override func viewDidAppear(animated: Bool)  
override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int  
override func didMoveToView(view: SKView)  
override func traitCollectionDidChange(previousTraitCollection: UITraitCollection?)  
func textFieldShouldReturn(textField: UITextField) -> Bool
```

...have all changed so that their first parameter does not require a label, like this:

```
override func viewDidAppear(_ animated: Bool)  
override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int  
override func didMoveToView(_ view: SKView)  
override func traitCollectionDidChange(_ previousTraitCollection: UITraitCollection?)  
func textFieldShouldReturn(_ textField: UITextField) -> Bool
```

It's not a massive problem, but it does add a small amount of clutter to your Swift 3.0 code.

Omit needless words

Once you start splitting up method names to ensure there's always a first parameter label, one even bigger becomes possible: you can start to remove words that can be inferred by context. For example, when you use `UIColor.blueColor()`, did you really expect to get a blue rhino back? Or a blue pair of socks? No: the second "color" part isn't needed because the context is clear, and so Swift 3.0 renames it to `UIColor.blue()`.

This change has automatically been applied to thousands of method names across all of Cocoa, as you can see with these before and after examples:

```
let min = numbers.minElement()  
let min = numbers.min()  
  
attributedString.appendAttributedString(anotherString)  
attributedString.append(anotherString)  
  
names.insert("Jane", atIndex: 0)  
names.insert("Jane", at: 0)  
  
UIDevice.currentDevice()  
UIDevice.current()
```

This makes your code a *lot* shorter, because repetition has almost disappeared. This is particularly strongly felt with strings, where repetition was rampant – here are some more before and afters:

```
" Hello
```

```

".stringByTrimmingCharactersInSet(.whitespaceAndNewlineCharacterSet()
)
"Hello ".trimmingCharacters(in: .whitespacesAndNewlines)

"Taylor".containsString("ayl")
"Taylor".contains("ayl")

"1,2,3,4,5".componentsSeparatedByString(",")
"1,2,3,4,5".components(separatedBy: ",")

myPath.stringByAppendingPathComponent("file.txt")
myPath.appendingPathComponent("file.txt")

"Hello, world".stringByReplacingOccurrencesOfString("Hello",
withString: "Goodbye")
"Hello, world".replacingOccurrences(of: "Hello", with: "Goodbye")

"Hello, world".substringFromIndex(7)
"Hello, world".substring(from: 7)

"Hello, world".capitalizedString
"Hello, world".capitalized

```

Foundation has been reworked

The Foundation framework – responsible for many central types such as **NSDate**, **NSData**, **NSIndexPath**, **NSURL**, and more, has been dramatically reworked for Swift 3. This reworking comes in three flavors:

1. Many types have dropped their “NS” prefix, so **NSUserDefaults** is now just **UserDefaults**, and **NSFileManager** is now just **FileManager**.

2. Some key types – **Data**, **Date**, **URL**, **UUID**, and more – have had all-new Swift wrappers written that make them structs rather than classes.
3. There are new improvements to make certain types more Swifty, such as new initializers and inline mutation where previously it was impossible.

Of those, point 2 is the one that's most likely to surprise you, because a huge range of types have gone from being classes to structs. Yes, **NSDate** has become **Date** because of point 1, but it's the change in usage that matters – you can now declare dates as variable and mutate them inline, which wasn't possible before. As part of this change, **Date**, **Data**, **DateComponents**, **IndexPath**, **IndexSet**, **Notification**, **URL**, **URLComponents**, and **UUID**, among others, have all become value types in Swift 3, which makes your code that little bit easier to reason about.

Going back to point 1, a lot of things have remained classes but just dropped “NS” from their name: **NSDateFormatter**, **NSNotificationCenter**, **UserDefaults**, **FileManager**, **OperationQueue**, **Timer**, and more. This is a bit surprising at first, but doesn't take long to adjust to.

Swift 3's converter system ensures that all imported APIs use the new types, whether that's just dropping the **NS** prefix or converting it to use the new value types.

UpperCamelCase has been replaced with lowerCamelCase for enums and properties

Above and beyond being syntactically correct, developers also have a set of conventions that attach meaning to their code. Even though they have no semantic meaning to the compiler, these conventions are followed almost everywhere and so have meaning to other developers.

For example, if I write **MyObject** you assume I mean a class because it starts with a capital letter, but if I write **myObject** you assume I mean a variable, because it starts with a lowercase letter. These two styles are called UpperCamelCase and lowerCamelCase, because we always add capital letters at the start of second and subsequent words.

Prior to Swift 3, this rule was enforced nearly all the time, but there were some exceptions. Specifically, any properties or parameter labels that included acronyms were written using uppercase: **myColor.CGColor**, **myColor.CIColor**, **NSURLRequest(URL: someURL)**, and so on. This has been changed in Swift 3 so that the convention is applied uniformly, which can

be a bit jarring at first: `myColor.cgColor`, `myColor.ciColor`, and `NSURLRequest(url: someURL)`.

Elsewhere, enum cases have been changed from UpperCamelCase to lowerCamelCase, which is an extension of the same convention: an enum's name is UpperCamelCase just like a class, whereas the values of the enum are lowerCamelCase just like variables. So:

```
UIInterfaceOrientationMask.Portrait // old  
UIInterfaceOrientationMask.portrait // new
```

```
NSTextAlignment.Left // old  
NSTextAlignment.left // new
```

```
SKBlendMode.Replace // old  
SKBlendMode.replace // new
```

You get the idea. However, this tiny change brings something much bigger because Swift's optionals are actually just an enum under the hood, like this:

```
enum Optional {  
    case None  
    case Some(Wrapped)  
}
```

This means if you use `.Some` to work with optionals, you'll need to switch to `.some` instead. Of course, you could always take this opportunity to ditch `.some` entirely – these two pieces of code are identical:

```
for case let .some(datum) in data {  
    print(datum)
```

```
}
```

```
for case let datum? in data {  
    print(datum)  
}
```

Swifty importing of C functions

Swift 3 introduces attributes for C functions that allow library authors to specify new and beautiful ways their code should be imported into Swift. For example, all those functions that start with "CGContext" now get mapped to properties methods on a **CGContext** object, which makes for much more idiomatic Swift. Yes, this means the hideous wart that is **CGContextSetFillColorWithColor()** has finally been excised.

To demonstrate this, here's an example in Swift 2.2:

```
let ctx = UIGraphicsGetCurrentContext()  
  
let rectangle = CGRectMake(x: 0, y: 0, width: 512, height: 512)  
CGContextSetFillColorWithColor(ctx, UIColor.redColor().CGColor)  
CGContextSetStrokeColorWithColor(ctx, UIColor.blackColor().CGColor)  
CGContextSetLineWidth(ctx, 10)  
CGContextAddRect(ctx, rectangle)  
CGContextDrawPath(ctx, .FillStroke)  
  
UIGraphicsEndImageContext()
```

In Swift 3 the **CGContext** can be treated as an object that you can call methods on rather than repeating "CGContext" again and again. So, we can rewrite that code like this:

```
if let ctx = UIGraphicsGetCurrentContext() {  
    let rectangle = CGRect(x: 0, y: 0, width: 512, height: 512)  
    ctx.setFillColor(UIColor.red().cgColor)  
    ctx.setStrokeColor(UIColor.black().cgColor)  
    ctx.setLineWidth(10)  
    ctx.addRect(rectangle)  
    ctx.drawPath(using: .fillStroke)  
  
    UIGraphicsEndImageContext()  
}
```

Note: in both Swift 2.2 and Swift 3.0 **UIGraphicsGetCurrentContext()** returns an optional **CGContext**, but because Swift 3 uses method calls we need to safely unwrap before it's used.

This mapping of C functions exists elsewhere, for example you can now read the **numberOfPages** property of a **CGPDFDocument**, and **CGAffineTransform** has been souped up quite dramatically. Here are some examples showing old and new:

CGAffineTransformIdentity

CGAffineTransform.identity

CGAffineTransformMakeScale(2, 2)

CGAffineTransform(scaleX: 2, y: 2)

CGAffineTransformMakeTranslation(128, 128)

CGAffineTransform(translationX: 128, y: 128)

CGAffineTransformMakeRotation(CGFloat(M_PI))

CGAffineTransform(rotationAngle: CGFloat.pi)

Grand Central Dispatch (GCD) has also seen a massive rewrite, and is much more pleasing as a result. For example, old code used global functions like `dispatch_async()` and `dispatch_get_main_queue()` to push work to the main thread:

```
dispatch_async(dispatch_get_main_queue()) {
    // do stuff
}
```

That's been rewritten for Swift 3 so that the code is now this:

```
DispatchQueue.main.async {
    // do stuff
}
```

Again, it's shorter to write, but note how it's beautifully namespaced – no more pollution of the global namespace.

New `#keyPath` construct for key-value observing

KVO has a long history on Apple's platforms, and allow us to be notified when a particular value changes even if there are no available delegate callbacks for it. These key paths were previously written as strings, like this:

```
webView.addObserver(self, forKeyPath: "estimatedProgress",
options: .New, context: nil)
```

That's problematic for all the same reasons as stringified selectors, which were deprecated and replaced with `#selector` in Swift 2.2.

Well, in Swift 3 the same change has been made for key paths: there's a new **#keyPath** directive that means you now no longer use strings, and instead use properties of classes directly. This allows for compile-time checking rather than runtime crashes, but it also enables things like code completion because Xcode knows what you're doing.

So, the above code rewritten in Swift 3 looks like this:

```
webView.addObserver(self, forKeyPath:  
#keyPath(WKWebView.estimatedProgress), options: .new, context: nil)
```

Note: you observe the property on the *class*, not on the object. This means if you write code like the below you'll get a crash:

```
webView.addObserver(self, forKeyPath:  
#keyPath(webView.estimatedProgress), options: .new, context: nil)
```

There are other places where something like **#keyPath** would be very useful, not least specifying sort descriptors and predicates for Core Data, but at this time they still use plain strings. Maybe in iOS 11?

Verbs and nouns

This is the part where some people will start to drift off in confusion, which is a shame because it's important.

Here's are some quotes from the Swift API guidelines:

- "When the operation is naturally described by a verb, use the verb's imperative for the mutating method and apply the "ed" or "ing" suffix to name its nonmutating counterpart"
- "Prefer to name the nonmutating variant using the verb's past participle"
- "When adding "ed" is not grammatical because the verb has a direct object, name the nonmutating variant using the verb's present participle"

- “When the operation is naturally described by a noun, use the noun for the nonmutating method and apply the “form” prefix to name its mutating counterpart”

Got that? It's no surprise that Swift's rules are expressed using linguistic terminology – it is after all a language! – but this at least gives me a chance to feel smug that I did a second degree in English. What it means is that many methods are changing names in subtle and sometimes confusing ways.

Let's start with a couple of simple examples:

```
myArray.enumerate()  
myArray.enumerated()
```

```
myArray.reverse()  
myArray.reversed()
```

Each time Swift 3 modifies the method by adding a "d" to the end: this is a value that's being returned.

These rules are mostly innocent enough, but it causes confusion when it comes to array sorting. Swift 2.2 used `sort()` to return a sorted array, and `sortInPlace()` to sort an array in place. In Swift 3.0, `sort()` is renamed to `sorted()` (following the examples above), and `sortInPlace()` is renamed to `sort()`.

Many closure parameters now have default values

This is a small change but a welcome one nonetheless: if you ever got tired of writing `completion: nil` for methods that expect a closure, you might now find this is now no longer required in Swift 3.

When combined with the previous changes, this makes a huge difference to the amount of code you need to write. For example, this is how you dismissed a view controller in Swift 2.2:

```
dismissViewControllerAnimated(true, completion: nil)
```

And in Swift you now just write this:

```
dismiss(animated: true)
```

Project 1

Happy Days

Setting up

In this project we're going to build Happy Days: an app that stores photos the user has selected, along with voice recordings recounting what's in the picture.

To add some spice to the project, we'll be using the new iOS 10 Speech framework to transcribe the user's voice recordings, then store that transcription in Core Spotlight. iOS 10 also gives us the ability to search our Spotlight index inside the app using the new [CSSearchQuery](#) class, so we'll be drawing on that too.

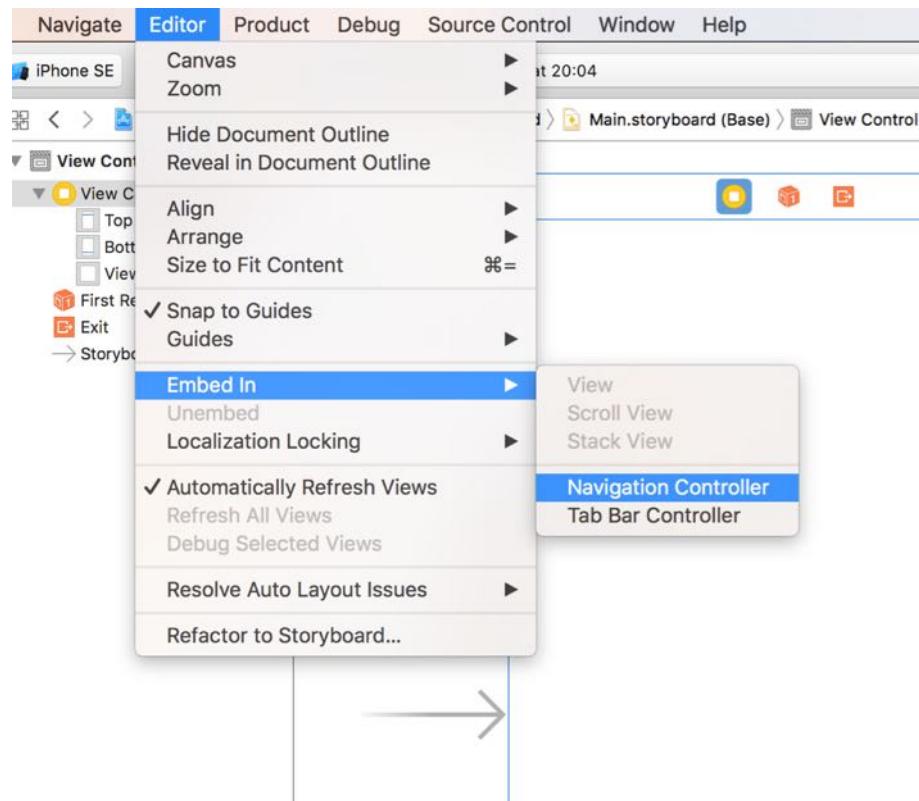
Launch Xcode 8, and create a new project using the Single View Application template. Give it the name HappyDays, then set the device to Universal. All set for some Swift 3? Let's do it!

Building the user interface

Happy Days is going to display the user's photos inside a collection view controller, which will itself be inside a navigation controller. It will also need a second view controller for asking permissions when the app first runs: we need to access the user's photo library and microphone, and also request permission to transcribe their speech – Apple are predictably cautious about apps transcribing speech without user permission!

The Single View Application template gave us a regular view controller, so we can use that for our permissions view controller. You could, if you wanted, do without this and just request permissions on demand, but because we're asking for three at once I found it was a more pleasant experience to ask for them when the app first launches rather than confuse the user with requests later on.

Let's start with the permissions first run screen, because it's easy enough. Select the current view controller, and embed it inside a navigation controller by going to Editor > Embed In > Navigation Controller. You'll see a simulated navigation bar appear in the view controller – double-click in the center of that to edit the title, and write "Welcome".

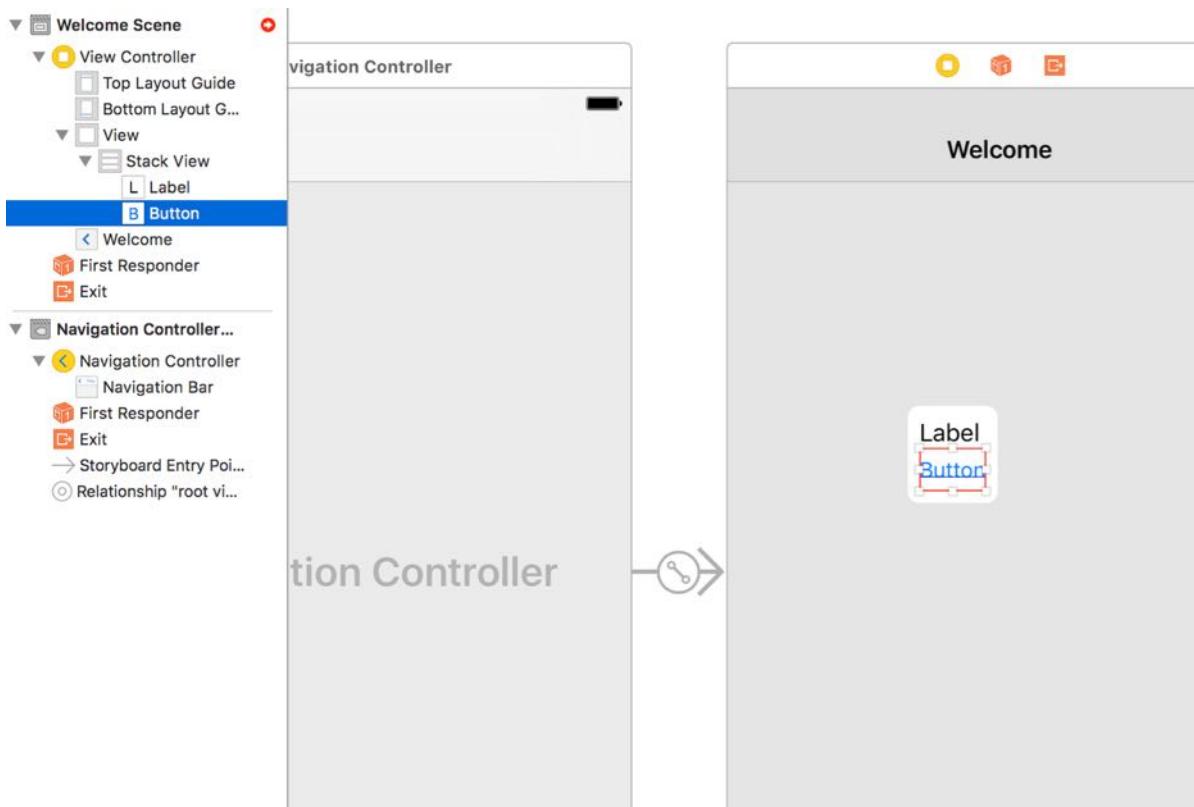


To embed a view controller inside a navigation controller, go to the Editor and menu and look

under Embed In.

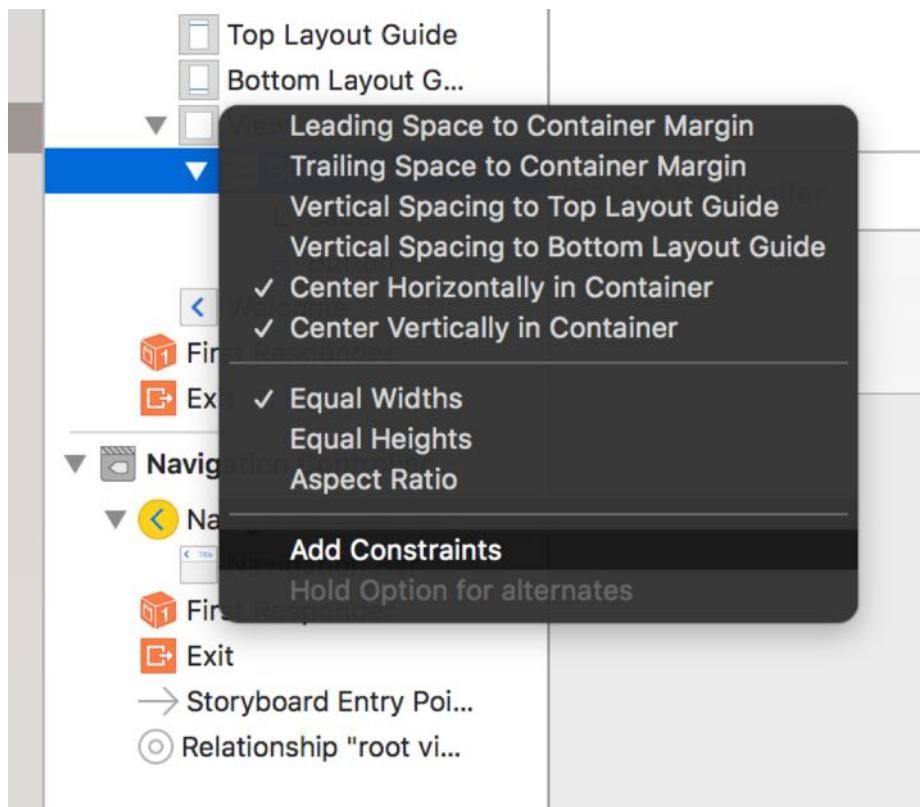
Next, drop a Vertical Stack View object into the view controller anywhere inside – we'll be using this to host a label describing what's going on, as well as a button let users proceed. Having these two inside a stack view allows us to center the stack inside the parent view, even when the text resizes.

Drag a new label into the stack view, and you'll see the stack view shrink so that it wraps itself around the label. Please also drag a button into the stack view, below the label – again you'll see the stack view resize so that it fits them both neatly.



Place both the label and the button inside a vertical stack view, which will shrink down to fit them exactly.

We want to position the stack view so that it fits neatly onto the screen. To do that, Ctrl-drag from the stack view to the parent view to create constraints, and select Center Horizontally in Container, Center Vertically in Container. Finally, then Equal Widths. Having the stack view fill the view horizontally doesn't look great, though, so we're going to make a tiny change to that last constraint to pull it in from the edges.



Add constraints so that the stack view is centered within its parent, but make it match its width so that it stays on the screen.

Make sure the stack view is selected, then go to the size inspector (`Alt+Cmd+5`) so you can see the constraints we just created. Look for the one marked “Equal width to: Superview” and click the Edit button next to it. In the popup that appears, enter `-40` for the Constant value so that the stack view is 40 points slimmer than its parent.

That's all our layout done, but before we update the frames we need to make a couple more changes. First, select the label:

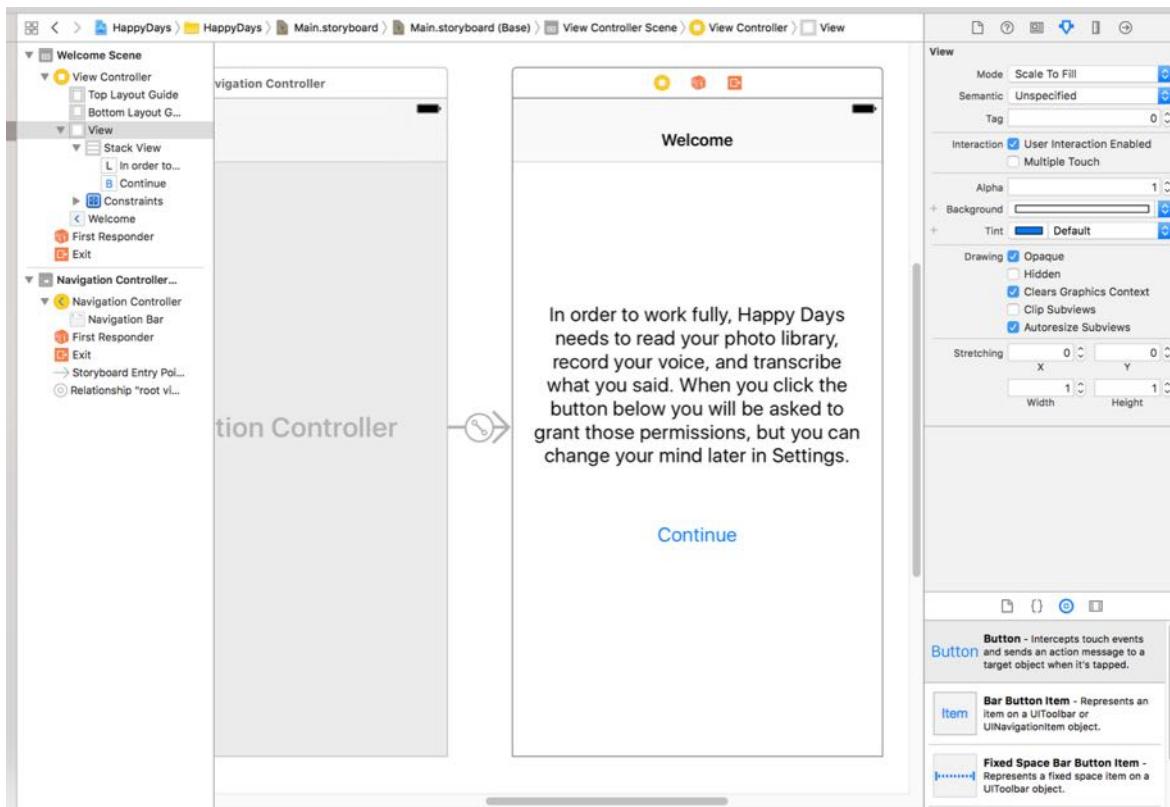
- Set its font size to 20.
- Set its text to be center aligned.
- Set its Lines value to 0.

Now give it this text: “In order to work fully, Happy Days needs to read your photo library, record your voice, and transcribe what you said. When you click the button below you will be asked to grant those permissions, but you can change your mind later in Settings.”

Second: select the button, then set its font size to 25 and its text to “Continue”.

By default stack views place their arranged views directly adjacent to each other, which doesn't look that great. To fix that, change Spacing to be 50 and you'll see the label and button jump apart.

That's all our layout done, so click in the whitespace above the label to select the main view. Now go to the Editor menu and choose Resolve Auto Layout Issues > Update Frames – make sure you choose the one under “All Views in View Controller”.



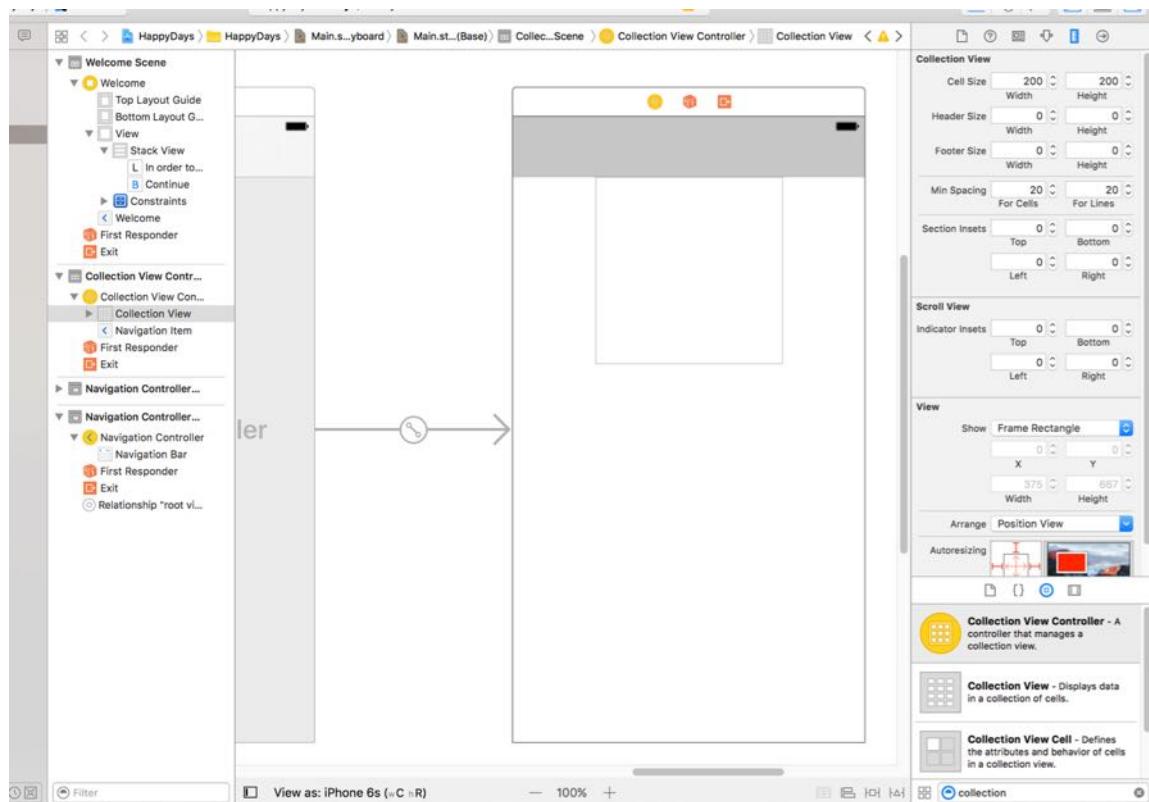
The finished layout: one label, one button, and one stack view, all neatly centered on the screen.

Creating the main view controller

We're done with the first run screen for the time being – we'll come back to it soon enough to hook it up to code. Now let's turn to the main view controller: the part that lets the user interact with their memories.

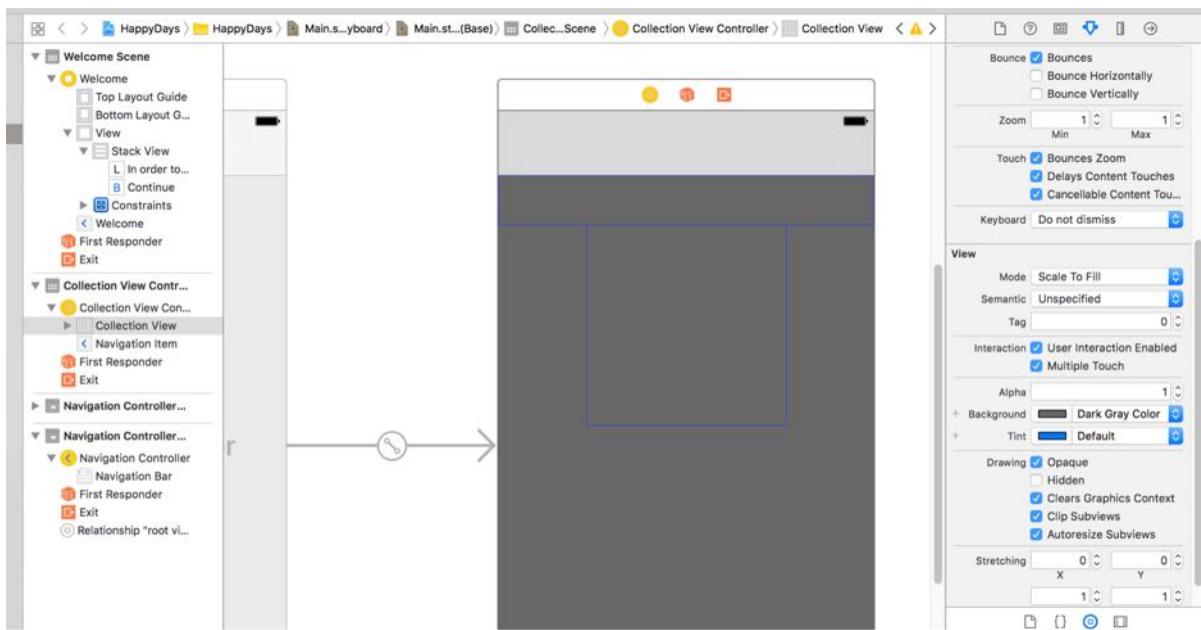
Start by dragging a new collection view controller onto your storyboard – that's the one in a yellow circle, not the plain collection view. Again, please embed it in a navigation controller: click the yellow circle directly above the new view, then choose Editor > Embed In > Navigation Controller.

We get one collection view cell prototype by default, but it's pretty small. To make it larger, click the large white space in the view to select the collection view, then go to the size inspector and change Cell Size from 50x50 to 200x200. While you're there, change Min Spacing to 20 and 20 for the “For Cells” and “For Lines” fields. We'll be using that to show photos.



Most of the action in our app will happen in this collection view controller.

Go back to the attributes inspector for the collection view, then check the box marked Section Header – we'll be using *that* to hold our search box. But first: give the collection view a dark gray background color, so that the images we load are prominent.



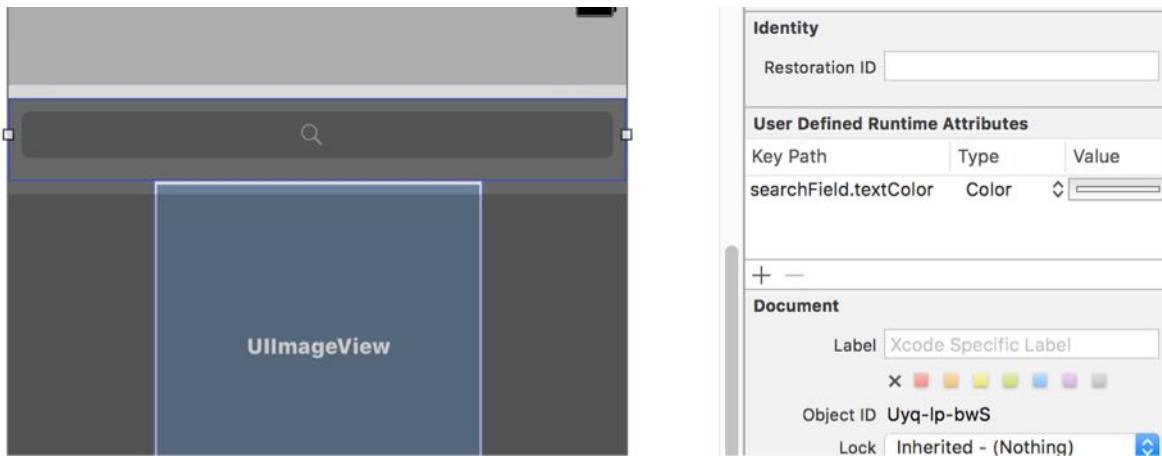
Giving the collection view a dark gray background color will help the photos we import stand out more clearly.

Right now we have one 200x200 collection view cell prototype taking up most of the screen. This is going to be used to hold the photos in our collection view, so please add an image view to it now. Xcode will make the image view a completely different size and shape to the cell – the easiest way to fix that is by going to the attributes inspector and hand-typing the values 0 for X and Y, and 200 for Width and Height. Now select the new image view in the document outline, then go to Editor > Resolve Auto Layout Issues > Add Missing Constraints. Technically the image view was already selected, but Xcode occasionally gets confused when you've been typing into one of the inspectors!

As for the collection view's section header, please drag a search bar object into there. These things are a fixed size, but you might need to ensure it has its X and Y position set to 0. I found that changing the Search Style to Minimal gave the best look, but you're welcome to experiment. If you always want to go with the minimal style, you should know that it defaults to black text, which looks pretty poor on a dark gray background. There's no color picker to change that in Xcode, so instead we're going to modify a key path using the identity inspector – something you don't really have to do very much, thankfully.

Make sure the search bar is selected, then press Alt+Cmd+3 to activate the identity inspector. Under “User Defined Runtime Attributes” click the + button, then double-click where it says “keyPath” in the newly inserted row. Replace “keyPath” with “searchField.textColor”, then change Type from “Boolean” to “Color”. Doing that will change

the Value field from a checkbox to a color well, so click that now and change it to white. It might seem like a bit of a hack, but that's the only way to do it without poking around in code!



Xcode doesn't have a way to control the text color inside a UISearchBar, so we need to use a key path instead.

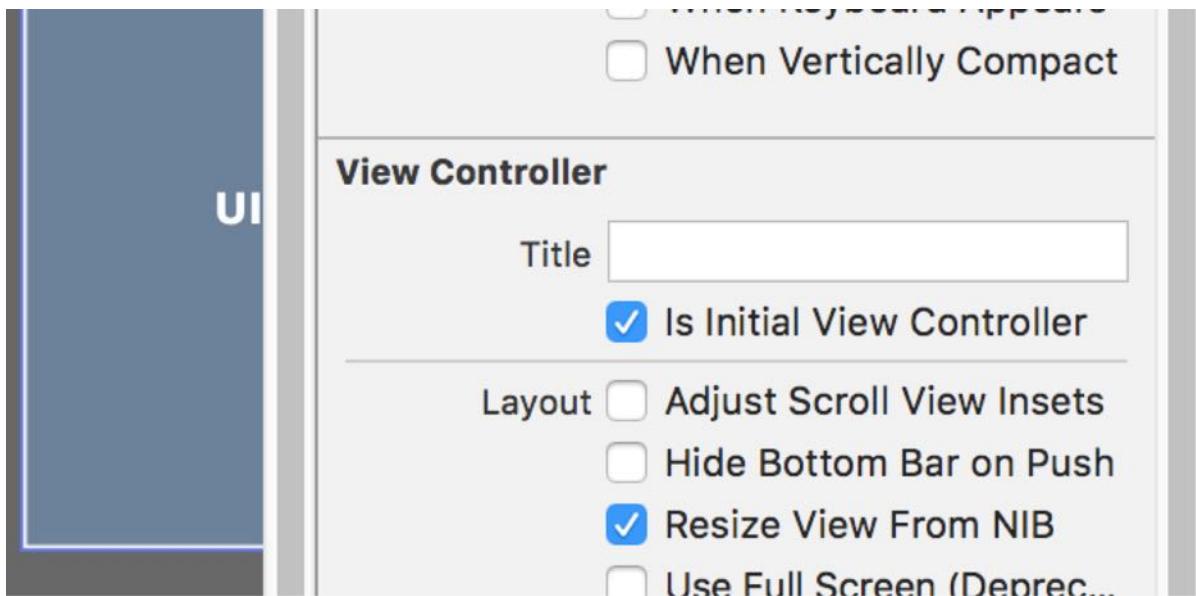
Finally, double-click the simulated navigation bar and give it the title Happy Days.

Connecting the interface to code

Although the drag-and-drop parts of our user interface are done, but we still need to make a few changes so that everything sits together neatly:

1. Ensure the collection view's navigation controller is the initial view controller.
2. Set a delegate for the search bar so we can act on the user's typing.
3. Configure re-use identifiers for the collection view cell and section header.
4. Give the first run screen a storyboard identifier so we can create it on demand.
5. Create a class for the collection view controller.
6. Create a class for the collection view cell that will hold images so that we can reference the image view in code.
7. Set up outlets and actions.

Let's start with number 1: making the collection view's navigation controller the initial view controller for the app. That's trivial: click the navigation controller, select the attributes inspector, then check the box marked Is Initial View Controller. One down!



To make the collection view's navigation controller be shown when the app starts, just select it then check the box marked Is Initial View Controller.

Number 2: set a delegate for the search bar so that we can act on the user's typing. To make this happen, Ctrl-drag from the search bar to Collection View Controller in the document outline, then select Delegate from the popup that appears. Boom: two down. Who said coding was hard?

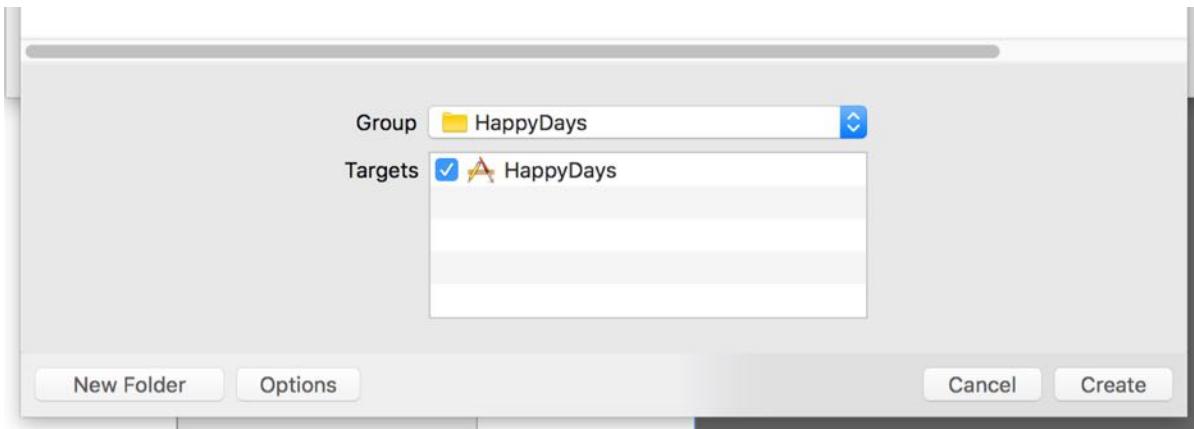
Number 3: configure re-use identifiers for the collection view cell and section header. Because both our cell and header have child views that occupy their full space, you should use the document to select each item. So, select Collection Reusable View then enter "Header" for the Identifier value in the attributes inspector. Then select Collection View Cell and give it the identifier "Memory".

Number 4: give the first run screen a storyboard identifier so we can create it on demand. To do this, select the navigation controller that wraps the first run screen, then choose the identity inspector. Look for the Storyboard ID field in there, and give it the value FirstRun.

Number 5: create a class for the collection view controller. This is the first non-trivial task, but even then we can re-use the existing ViewController.swift file given to us by the template.

To create a class for the collection view controller, choose File > New > File, then select iOS > Source > Cocoa Touch Class and click Next. Make the new class inherit from **UIViewController**, give it the name MemoriesViewController, then click Next again. In the

final screen, make sure Group is set to Happy Days with the yellow folder icon next to it, then click Create.



Xcode makes it unhelpfully easy to create your files in the wrong place, so please make sure you see a yellow folder next to the name Happy Days

Note: if you're paying attention, you should have read that last paragraph and thought "why are we subclassing **MemoriesViewController** from **UIViewController** when it's clearly a **UICollectionViewController**?" The answer is simple: Xcode's default code for **UICollectionViewController** subclasses is suboptimal, and we can do better.

Now let's connect that the new class to our user interface. Because we created a regular **UIViewController** subclass, we need to make one tiny tweak in `MemoriesViewController.swift` before we can go into IB. Open that file, and change this line:

```
class MemoriesViewController: UIViewController {
```

To this:

```
class MemoriesViewController: UICollectionViewController {
```

Now open `Main.storyboard` in Interface Builder, and select the Collection View Controller. We need to tell IB that this view controller is an instance of the **MemoriesViewController** we just created, so go to the identity inspector and change Class to "MemoriesViewController".

That's task number 5 done!

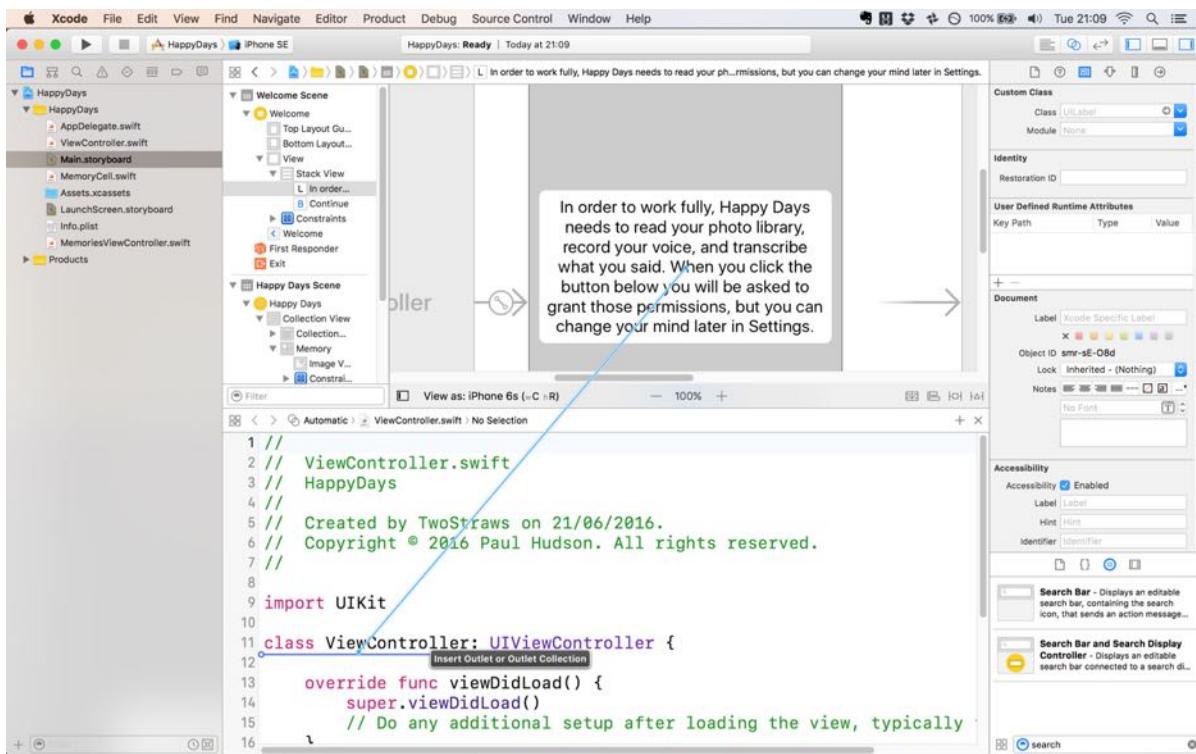
On to task number 6: create a class for the collection view cell that will hold images so that we can reference the image view in code. This isn't strictly needed because the cell is so trivial we could just use a tag, but later on you'll want to expand this so adding a full class is a smart move.

Go to File > New > File, then choose iOS > Source > Cocoa Touch Class and press Next. Make it a subclass of **UICollectionViewCell** then name it "MemoryCell" and click Next. As before, make sure you use Happy Days with the yellow folder icon for your group, then click Create.

That creates the class, so now it's just a matter of connecting the cell prototype to the class in IB. To do that, select "Memory" in the document outline, then use the identity inspector to change the Class value to "MemoryCell".

And now the final task, number 7: set up outlets and actions. We'll do the first run screen first, so select the first run controller in the document outline, then change to the assistant editor so we can see its code while working in IB. This works sometimes and other times either selects the wrong thing or selects nothing at all - in theory you should have IB in the top pane, and ViewController.swift in the bottom pane. If you have something else (or nothing!) in the bottom pane, click where it says Automatic and choose Manual > Happy Days > Happy Days > ViewController.swift.

I'd like you to create an outlet for the label called **helpLabel**, and an action for the button called **requestPermissions**.



Use the assistant editor create outlets by Ctrl-dragging from IB straight into your code

While running through this project a couple of times for testing, Xcode would occasionally refuse to create an outlet because it somehow couldn't find **ViewController** – if this happens to you, select your project icon in the project navigator, then choose HappyDays under the Targets list. Select the Build Phases tab, then open Compile Sources – you should see ViewController.swift in there. Select it, then click - to stop it from being built, then click + and re-add it. For whatever reason, that was the only thing I found that would clear the glitch, but hopefully it doesn't happen to you!

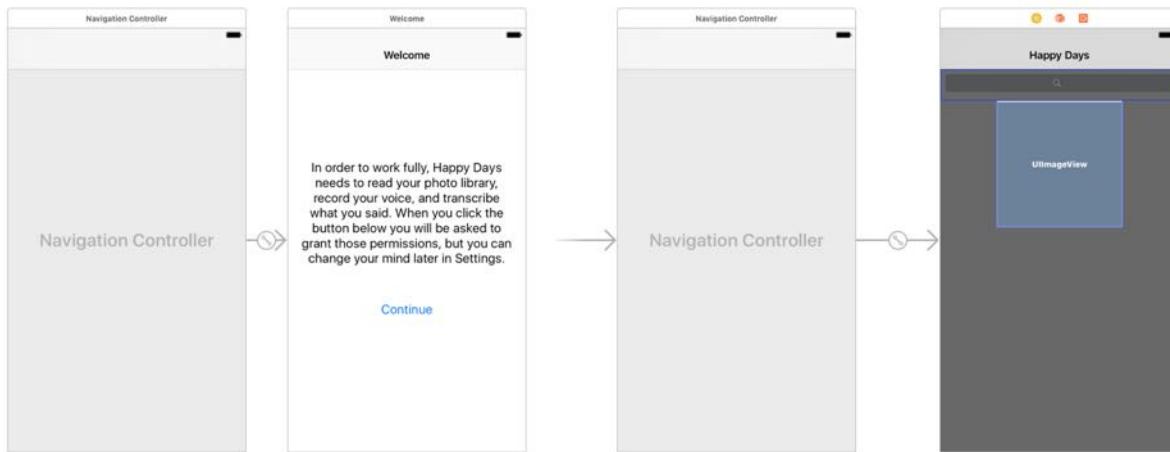
Anyway, back to Main.storyboard: you should have an outlet for the label, and an action for the button like this:

```
@IBAction func requestPermissions(_ sender: AnyObject) {  
}
```

The other outlet we're going to create is for the image view inside **MemoryCell**. Find the "Memory" collection view cell prototype in your IB document outline, then use manual

selection in the bottom pane of the assistant editor to open `MemoryCell.swift`. Now Ctrl-drag from the image view to `MemoryCell.swift` to create a property called **imageView**.

Finally, we're done with Interface Builder, so switch back to the standard editor view and open `ViewController.swift` for editing.



The finished user interface has four controllers, two of which are navigation controllers.

Permissions! Permissions everywhere!

We need three different kinds of permissions to make this app work. I'm going to code the app so that it requires all three to work, but you're welcome to make it more fine-grained if you want. If any of the permissions fail, I'm going to rewrite the contents of **helpLabel** with a meaningful message so that the user can go to Settings and grant the permissions by hand if they change their mind.

The permissions system in iOS is all asynchronous, which means that you make a request then receive a callback when the user has made a choice. These callbacks can happen on any thread, so you should always push work to the main thread before taking any action – particularly if you're manipulating the user interface. We're going to use these callbacks to create a smooth chain of permission requests: rather than bombard the user with three requests at once, we're going to request one, wait for a callback, request another, wait for a callback, then request the last one.

Each of these permission requests are done on a different framework: AVFoundation handles the microphone, Photos handles accessing user photos, and Speech handles transcription. So, please add these three imports near the top of ViewController.swift:

```
import AVFoundation  
import Photos  
import Speech
```

With that done, we can start the authorization process – Photos authorization is as good a place as any, so we'll start there.

To get access to their user's photo library, we need to call **requestAuthorization()** on the **PHPhotoLibrary** class. You need to give it a closure that will be called when the user has either granted or denied permission to access the photo library, and you'll be given a parameter to work with that represents the user's permission status. If that's set to **.authorized** it means we're good to continue, otherwise we'll show a message in the help label and stop.

Add this method to ViewController.swift:

```
func requestPhotosPermissions() {
    PHPPhotoLibrary.requestAuthorization { [unowned self] authStatus
        in
            DispatchQueue.main.async {
                if authStatus == .authorized {
                    self.requestRecordPermissions()
                } else {
                    self.helpLabel.text = "Photos permission was
declined; please enable it in settings then tap Continue again."
                }
            }
        }
}
```

You'll get an error because we haven't written **self.requestRecordPermissions()** just yet - we'll do that in just a moment.

First, though: notice that I'm pushing all the work to the main thread using **DispatchQueue.main.async**. This might be your first exposure to Grand Central Dispatch in Swift 3, but I hope you can appreciate how marvelously simple it is compared to GCD in Swift 2.2!

Next: the **requestRecordPermissions()** method. This is almost identical to requesting access to the photo library, except now we call **requestRecordPermission()** on the shared **AVAudioSession** instance. This will again call a closure, but the parameter is simpler this time: we'll get a boolean that's true if we've been given access to the microphone.

As before, we immediately push our work to the main thread to avoid problems, then either call another method to continue the permissions requests, or set the help label to something meaningful. Here's the code:

```
func requestRecordPermissions() {
    AVAudioSession.sharedInstance().requestRecordPermission()
```

```
{ [unowned self] allowed in
    DispatchQueue.main.async {
        if allowed {
            self.requestTranscribePermissions()
        } else {
            self.helpLabel.text = "Recording permission was
declined; please enable it in settings then tap Continue again."
        }
    }
}
```

The last permissions method we need to write is **requestTranscribePermissions()**, which is almost identical to requesting permissions to Photos – the only difference is that you call the method on the new **SFSpeechRecognizer** class rather than **PHPhotoLibrary**. Here's the code:

```
func requestTranscribePermissions() {
    SFSpeechRecognizer.requestAuthorization { [unowned self]
authStatus in
    DispatchQueue.main.async {
        if authStatus == .authorized {
            self.authorizationComplete()
        } else {
            self.helpLabel.text = "Transcription permission was
declined; please enable it in settings then tap Continue again."
        }
    }
}
```

That's the last of the permissions work. We need to make two further additions, though. First, add this method to handle dismissing the first run controller when permissions are fully granted:

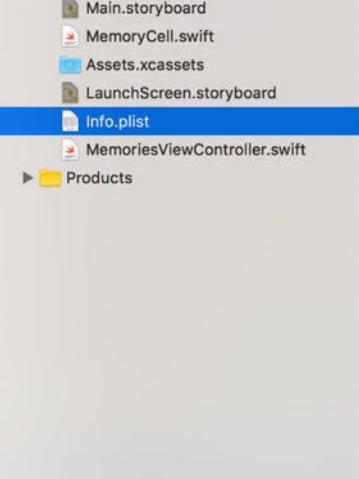
```
func authorizationComplete() {  
    dismiss(animated: true)  
}
```

Second, we need to kick off the whole permissions flow by adding a call to **requestPhotosPermissions()** inside the **requestPermissions()** method we created earlier, like this:

```
@IBAction func requestPermissions(_ sender: AnyObject) {  
    requestPhotosPermissions()  
}
```

That's all the code for permissions: we request photo library access, then request microphone access, then request transcription access. However, even though the code is complete and correct, it won't work quite yet because we need to tell the user *why* we're requesting the permissions.

This is done in the Info.plist file: we need to add three keys there to describe why we're asking for each of the permissions. Please add the keys **NSPhotoLibraryUsageDescription**, **NSMicrophoneUsageDescription**, and **NSSpeechRecognitionUsageDescription**. Give each of them some meaningful text - something like "We use this to let you import photos", "We use this to record your narration", and "We use this to transcribe your narration" respectively.



The screenshot shows the Xcode project navigator on the left with files like Main.storyboard, MemoryCell.swift, Assets.xcassets, LaunchScreen.storyboard, Info.plist, and MemoriesViewController.swift. The Info.plist file is selected, and its contents are displayed in a table on the right.

Bundle identifier	String	<code>\$(PRODUCT_BUNDLE_IDENTIFIER)</code>
InfoDictionary version	String	6.0
Bundle name	String	<code>\$(PRODUCT_NAME)</code>
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
Bundle version	String	1
Application requires iPhone enviro...	Boolean	YES
Launch screen interface file base...	String	LaunchScreen
Main storyboard file base name	String	Main
Privacy - Photo Library Usage Des...	String	We use this to let you import photos
Privacy - Microphone Usage Desc...	String	We use this to record your narration
NSSpeechRecognitionUsageDescr...	String	We use this to transcribe your narration
► Required device capabilities	Array	(1 item)
► Supported interface orientations	Array	(3 items)
► Supported interface orientations (i...	Array	(4 items)

You need to add all three keys to your Info.plist file in order for permissions to be granted.

We're done with ViewController.swift – it's sole purpose is to guide the user through all the permissions requests, so all that's left to do is call it at the right time. We know when that "right time" is inside the `viewDidAppear()` method of **MemoriesViewController**: we can check what permissions we have, and show **ViewController** if any are lacking.

Open `MemoriesViewController.swift`, and add these three just above the existing `import UIKit` line:

```
import AVFoundation
import Photos
import Speech
```

I'm going to encapsulate the permissions check in its own method, `checkPermissions()`, because it's possible you'll want to check this in other places later on. All it's going to do is go through **PHPhotoLibrary**, **AVAudioSession**, and **SFSpeechRecognizer** to make sure each of them have permissions granted, then combine them all in to a single boolean, `authorized`, which can be checked. If that's false, it means we're missing at least one permission, so we'll show the "FirstRun" view controller, which is the navigation controller that contains the permissions view.

Here's the `checkPermissions()` method – please put this inside `MemoriesViewController.swift`:

```
func checkPermissions() {  
    // check status for all three permissions  
  
    let photosAuthorized = PHPhotoLibrary.authorizationStatus()  
== .authorized  
  
    let recordingAuthorized =  
AVAudioSession.sharedInstance().recordPermission() == .granted  
  
    let transcribeAuthorized =  
SFSpeechRecognizer.authorizationStatus() == .authorized  
  
    // make a single boolean out of all three  
  
    let authorized = photosAuthorized && recordingAuthorized &&  
transcribeAuthorized  
  
    // if we're missing one, show the first run screen  
    if authorized == false {  
  
        if let vc =  
storyboard?.instantiateViewController(withIdentifier: "FirstRun") {  
  
            navigationController?.present(vc, animated: true)  
        }  
    }  
}
```

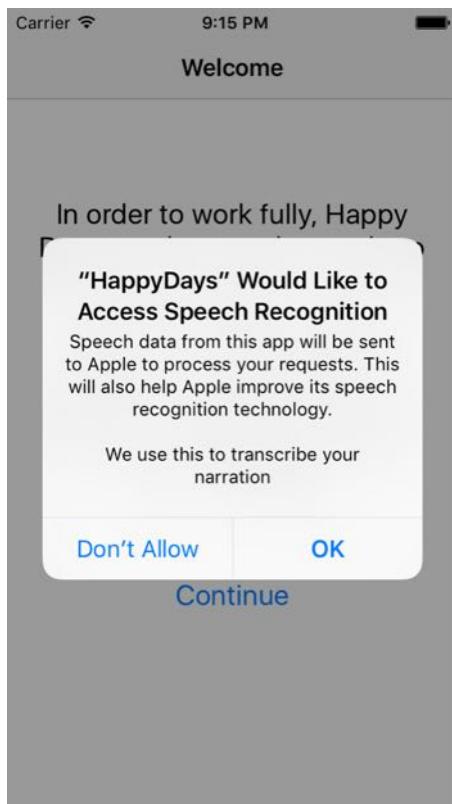
With that method written, all we need to do now is call it inside **viewDidAppear()**. Add this now:

```
override func viewDidAppear(_ animated: Bool) {  
    super.viewDidAppear(animated)  
  
    checkPermissions()  
}
```

Done!

At this point we finally have some code that can be run, so press Cmd+R to build and launch in the simulator. All being well you should see the first run screen appear and ask for various permissions – you may not see the microphone permission appear in the simulator, but it will appear on real devices.

When the final permission is granted, the welcome screen will disappear and you'll see the empty collection view controller. Let's tackle that now...



When the app runs it will request permissions using the text you entered into your Info.plist keys.

Importing into the collection view

The first big task we're going to tackle is to fill our collection view with pictures the user has selected. The final version of the app will have pictures, audio, and text for each memory, so we need some sort of way to tie this together.

We *could* use a custom class for this task: by making it conform to **NSCoding** we could read and write whole arrays of data using something like **NSKeyedArchiver**. However, I think that would prove inefficient: picture data can easily be 10MB depending on the device you have, and there's audio data too – we wouldn't want to have to write all that data out when the user changed one small part.

So instead we're going to go for something much simpler: every memory will have a unique name, and we'll just hang different file extensions off that unique name to provide the various components. For example, if a memory had the name abc123, we would have the following files:

- abc123.jpg: the full-size picture the user imported.
- abc123.thumb: the thumbnail-sized picture
- abc123.m4a: the audio narration they recorded
- abc123.txt: the plain text transcription of what they said.

Of those, only the first two can be guaranteed to exist – the latter two won't exist until the user has added narration and we've transcribed it.

Working with files used to be quite easy in Swift because you could read and write files using path strings. However, in the name of safety, Apple has slowly been forcing us to migrate to URLs for files, which in turn means we need to litter optionals and try/catch code throughout our code. It's safer in the long term, but it might frustrate you until you get the hang of it!

Loading memories

Let's start with the easiest part: loading memories into an array. We're going to create a property that contains the full path to the root name of a memory – that's the memory name without the ".jpg" or ".m4a". To keep things uniform, this will be an array of URLs, so please add this property to **MemoriesViewController**:

```
var memories = [URL]()
```

We'll fill that array in a dedicated method, **loadMemories()**, so that we can call it later on – for example, when we add a new memory from the photo library.

This method needs to do several things:

1. Remove any existing memories. We'll be calling it multiple times, so we don't want duplicates.
2. Pull out a list of all the files stored in our app's documents directory. This needs to be work with URLs, rather than path strings.
3. Loop over every file that was found, and, if it was a thumbnail, add it to our memories array. We only count thumbnails to avoid duplicating items - we don't want to add the .jpg and .thumb for each memory.

That last point is actually quite complicated thanks to the rather graceless way Swift handles URLs, but first the easy bit – finding our app's documents directory can be done using this helper function:

```
func getDocumentsDirectory() -> URL {  
    let paths =  
        FileManager.default.urlsForDirectory(.documentDirectory,  
        inDomains: .userDomainMask)  
  
    let documentsDirectory = paths[0]  
  
    return documentsDirectory  
}
```

Now for the **loadMemories()** method itself. I've added comments directly inline with the code, because it's harder than you might think. Add this method to **MemoriesViewController**:

```
func loadMemories() {
```

```

memories.removeAll()

    guard let files = try?
FileManager.default.contentsOfDirectory(at: getDocumentsDirectory(),
includingPropertiesForKeys: nil, options: []) else { return }

for file in files {
    guard let filename = file.lastPathComponent else { continue }

    if filename.hasSuffix(".thumb") {
        let noExtension = filename.replacingOccurrences(of:
".thumb", with: "")

        if let memoryPath = try?
getDocumentsDirectory().appendingPathComponent(noExtension) {
            memories.append(memoryPath)
        }
    }
}

collectionView?.reloadSections(IndexSet(integer: 1))
}

```

That small chunk of code has two uses of **guard let** and two of **try?** – all because of the inherent optionality in URLs. We need to call **contentsOfDirectory()** with **try?** because it might fail if we have missing permissions, and we need to use **try?** again when calling **AppendingPathComponent()** because that might fail if we tried to add an invalid path component. I also had to use a **guard let** when unwrapping **LastPathComponent**, because that might not exist. Now, obviously we know our code is safe, but Swift doesn't!

One bright side of all this complexity is the **guard** statement: it's great for checking and unwrapping optionals, particularly when combined with **try?**.

You might have wondered why I've used **reloadSections()** to reload the second section in the collection view – they count from 0, remember. We'll be placing the search box in a section all by itself, with all the pictures in section 1. Using this approach means we can call **reloadSections()** on section 1 to have the photos reload without touching the search box, which is how we'll be doing filtering. Without this split, reloading the photos would cause the search box to reload, causing the user's search to stop.

Finally, add a call to that method in **viewDidLoad()**:

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    loadMemories()  
}
```

Adding new memories using **UIImagePickerController**

I used **UIImagePickerController** in several Hacking with Swift projects, so hopefully you're already familiar with it by now.

First things first: make **MemoriesViewController** conform to these two protocols:

```
UIImagePickerControllerDelegate, UINavigationControllerDelegate,
```

We're going to trigger the image picker using a button in the navigation bar, so please add this to **viewDidLoad()**:

```
navigationItem.rightBarButtonItem =  
UIBarButtonItem(barButtonSystemItem: .add, target: self, action:  
#selector(addTapped))
```

That will call a method named **addTapped()** (not written yet!), which will show the image picker controller. This method just needs to create an instance of **UIImagePickerController()** then show it. To make the UI a little more attractive on iPad, I suggest setting the **modalPresentationStyle** property to **.formSheet** so that it occupies only part of the screen.

Here's the code – add this to **MemoriesViewController**:

```
func addTapped() {  
    let vc = UIImagePickerController()  
    vc.modalPresentationStyle = .formSheet  
    vc.delegate = self  
    navigationController?.present(vc, animated: true, completion:  
nil)  
}
```

Once the image picker is showing, the user can either tap Cancel to abandon the process, or browse through their images to find the one they want. If they cancel the image picker will dismiss itself automatically, so all we care about is if they selected an image

When an image is selected, we'll get the **didFinishPickingMediaWithInfo** callback, and need to look for the **UIImagePickerControllerOriginalImage** key inside the dictionary we get sent. If there's a picture there we'll create a memory out of it then reload the memories array. Regardless of whether an imagine is found, we need to dismiss the image picker controller to return control back to **MemoriesViewController**.

```
func imagePickerController(_ picker: UIImagePickerController,  
didFinishPickingMediaWithInfo info: [String : AnyObject]) {  
    dismiss(animated: true, completion: nil)  
  
    if let possibleImage = info[UIImagePickerControllerOriginalImage]  
as? UIImage {  
        saveNewMemory(image: possibleImage)  
        loadMemories()  
    }  
}
```

```
    }  
}  


---


```

That code won't compile because we haven't made the **saveNewMemory()** method yet – add this stub to fix this:

```
func saveNewMemory(image: UIImage) {  
}
```

This new method needs to do some more try/catch juggling because we're back to working with URLs again, but I've tried to group it together to avoid too much mess.

Here's what it needs to do:

1. Generate a new, unique name for the memory. We could use **NSUUID** for this, but instead I'm going to use the current time in seconds so that memories are loaded and saved in order.
2. Use that unique name to create filenames for the .jpg and the .thumb.
3. Try to create an absolute URL by combining **getDocumentsDirectory()** with the image name.
4. Convert the **UIImage** it receives into JPEG data that can be saved.
5. Write that data to disk at the URL we already created.
6. Create a thumbnail for the image.

We'll tackle part six in a moment, but here's the code for the first five items:

```
func saveNewMemory(image: UIImage) {  
    // create a unique name for this memory  
    let memoryName = "memory-\\" + (Date().timeIntervalSince1970).description  
  
    // use the unique name to create filenames for the full-size  
    // image and the thumbnail  
    let imageName = memoryName + ".jpg"
```

```

let thumbnailName = memoryName + ".thumb"

do {
    // create a URL where we can write the JPEG to
    let imagePath = try
        getDocumentsDirectory().appendingPathComponent(imageName)

    // convert the UIImage into a JPEG data object
    if let jpegData = UIImageJPEGRepresentation(image, 80) {
        // write that data to the URL we created
        try jpegData.write(to: imagePath, options:
            [.atomicWrite])
    }

    // create thumbnail here
} catch {
    print("Failed to save to disk.")
}

```

There's a comment in there where we'll put thumbnail generation code in just a moment. But first, a recap: the user has shown the image picker, and selected an image. We then saved the image they chose to disk, and called **loadMemories()** again to refresh the collection view.

Creating thumbnails

We could, if we chose, proceed without thumbnails and just use the full-size images everywhere. However, if you try that you'll realize it's pretty painful as soon as you use the full app: modern iPhones capture huge images, and if you try to scroll through them you'll see the user interface stutter quite badly.

The solution: whenever a new memory is added, create a downsized version of it that is used for quick previewing while scrolling around. There are lots of ways to do this, but I'm going to show you the simplest: resize so that one edge, in our case the width, is no more than the size we need. First, add this method to **MemoriesViewController**:

```
func resize(image: UIImage, to width: CGFloat) -> UIImage? {  
    // calculate how much we need to bring the width down to match  
    // our target size  
    let scale = width / image.size.width  
  
    // bring the height down by the same amount so that the aspect  
    // ratio is preserved  
    let height = image.size.height * scale  
  
    // create a new image context we can draw into  
    UIGraphicsBeginImageContextWithOptions(CGSize(width: width,  
                                                height: height), false, 0)  
  
    // draw the original image into the context  
    image.draw(in: CGRect(x: 0, y: 0, width: width, height: height))  
  
    // pull out the resized version  
    let newImage = UIGraphicsGetImageFromCurrentImageContext()  
  
    // end the context so UIKit can clean up  
    UIGraphicsEndImageContext()  
  
    // send it back to the caller  
    return newImage  
}
```

I've added comments inline, but one part you might be unfamiliar with is the call to **UIGraphicsBeginImageContextWithOptions()**. This creates a new canvas for drawing at a size we specify, and you can set parameter 2 to **true** if you want to draw on a transparent background. Parameter 3 is the scale factor you want to draw with, which is used to control retina graphics – you can write 1 here to draw at exact sizes, or 2 to write a double sizes for retina resolution. I've specified 0, which means “use whichever scale the device uses,” which means it will be 200x200 on non-retina devices, 400x400 on retina devices, and 600x600 on retina HD devices.

With that method written, we can replace the **// create thumbnail here** comment with this code:

```
if let thumbnail = resize(image: image, to: 200) {  
    let imagePath = try  
        getDocumentsDirectory().appendingPathComponent(thumbnailName)  
    if let jpegData = UIImageJPEGRepresentation(thumbnail, 80) {  
        try jpegData.write(to: imagePath, options: [.atomicWrite])  
    }  
}
```

That's identical to the previous image writing code, with the addition of calling **resize()** at the beginning.

Displaying memories

At this point we've loaded existing memories from disk, written code to import new photos using the image picker, and added thumbnails for faster scrolling. The next step is surprisingly easy: we just need to fill in the collection view data source methods telling it how many sections there are, and so on.

Remember, we're going to have a search box in its own section at the top so that it doesn't interfere with the rest of the collection view; we'll come back to the implementation of that later but leave space for it now.

Add these two methods to **MemoriesViewController** now:

```
override func numberOfSections(in collectionView: UICollectionView) -> Int {
    return 2
}

override func collectionView(_ collectionView: UICollectionView, numberOfRowsInSection section: Int) -> Int {
    if section == 0 {
        return 0
    } else {
        return memories.count
    }
}
```

So, again: we have two sections in the collection view, and the first one is implemented as a section header so it has 0 rows. The second section is where all our memories will be stored, so it needs to have **memories.count** rows.

The next method to implement is **cellForItemAt**, where we can dequeue one of the **MemoryCell** cells we created earlier and set it up with a thumbnail. Once again, this is made slightly complicated by the optionality of URLs, so to make things easier I wrote some helper methods that strip out one layer of try/catch for times we know it's safe. Add these methods to the **MemoriesViewController** class now:

```
func imageURL(for memory: URL) -> URL {
    return try! memory.appendingPathExtension("jpg")
}

func thumbnailURL(for memory: URL) -> URL {
    return try! memory.appendingPathExtension("thumb")
```

```
}
```

```
func audioURL(for memory: URL) -> URL {
    return try! memory.appendingPathExtension("m4a")
}

func transcriptionURL(for memory: URL) -> URL {
    return try! memory.appendingPathExtension("txt")
}
```

This means we can take a URL we know is valid and append a string we also know is valid, without need to scatter **try!** elsewhere in our code.

With those helpers in place, **cellForItemAt** becomes much simpler: dequeue a cell, pull out the memory it matches, load a **UIImage** for the thumbnail of that memory, then put it into the cell's image view. There's still a little optionality in there because converting a URL to a string might fail (sigh) but that's easily solved with the nil coalescing operator.

Here's the code for the method:

```
override func collectionView(_ collectionView: UICollectionView,
cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {

    let cell =
    collectionView.dequeueReusableCell(withReuseIdentifier: "Memory",
for: indexPath) as! MemoryCell

    let memory = memories[indexPath.row]
    let imageName = thumbnailURL(for: memory).path ?? ""
    let image = UIImage(contentsOfFile: imageName)
    cell.imageView.image = image

    return cell
}
```

```
}
```

There are two final methods to implement before our code is in a usable state: we need to make the search bar header work. The code to create the header is marvelously simple: just add this method:

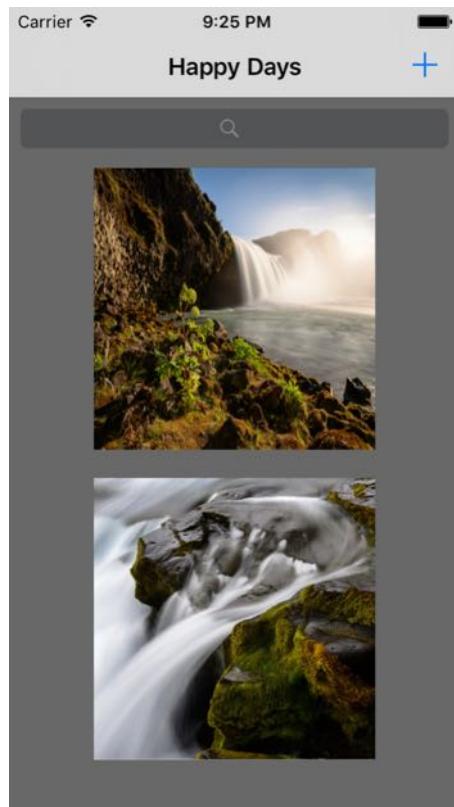
```
override func collectionView(_ collectionView: UICollectionView,  
viewForSupplementaryElementOfKind kind: String, at indexPath:  
IndexPath) -> UICollectionViewReusableView {  
  
    return collectionView.dequeueReusableCellSupplementaryView(ofKind:  
kind, withReuseIdentifier: "Header", for: indexPath)  
}
```

Where it's more *complicated* is handling which sections should have a header. You see, for whatever reason known only to Apple, the above method *must* return a view – you can't return **nil** to mean “no section header here please”. So, Apple's official, documented solution is just to provide a header everywhere, then set the height to zero for the headers you don't want. Yes, really.

First, add **UICollectionViewDelegateFlowLayout** to the list of protocols that **MemoriesViewController** conforms to. Now add this code somewhere in the class:

```
func collectionView(_ collectionView: UICollectionView, layout  
collectionViewLayout: UICollectionViewLayout,  
referenceSizeForHeaderInSection section: Int) -> CGSize {  
  
    if section == 1 {  
  
        return CGSize.zero  
    } else {  
  
        return CGSize(width: 0, height: 50)  
    }  
}
```

That's it – you can go ahead and run the code now and you'll see you can add memories and have them appear in the collection view.



Using a collection view means our pictures will automatically rearrange themselves to make best use of the screen space.

Recording, transcribing, and playing audio

I covered recording audio in project 33 of Hacking with Swift, but this time it's special for a couple of reasons. First, I want to start and stop recording by looking for a long press on a particular photo – the user will be able to press and hold to start speaking, then release to stop. Second, we're going to ask iOS to transcribe the audio into text so that we can search for parts of the narration by typing.

At the center of the process is a long press gesture recognizer that attach to every cell when it's first created. If it's in the **.began** state we start recording, and we'll turn the screen red so the user knows their microphone is on and recording what they say. When it hits the **.ended** state we know they released the long press and can stop recording – that's where the transcription process can kick off.

First things first: adding a gesture recognizer. By default cells don't have gesture recognizers, so we can check for **cell.gestureRecognizers** being nil in **cellForItemAt** in order to attach our recognizers only once. I'm going to hijack this opportunity to add a couple of layout tweaks at the same time: adding a white border to the pictures with gently rounded corners, to give a slightly more attractive feel to the app.

Add this code to **cellForItemAt**, just before **return cell**:

```
if cell.gestureRecognizers == nil {  
    let recognizer = UILongPressGestureRecognizer(target: self,  
action: #selector(memoryLongPress))  
    recognizer.minimumPressDuration = 0.25  
    cell.addGestureRecognizer(recognizer)  
  
    cell.layer.borderColor = UIColor.white().cgColor  
    cell.layer.borderWidth = 3  
    cell.layer.cornerRadius = 10  
}
```

You'll get an error because **memoryLongPress()** hasn't been created yet, however **memoryLongPress()** requires several other methods, so to make sure code compile cleanly please add these empty method stubs now:

```
func memoryLongPress(sender: UILongPressGestureRecognizer) {  
}  
  
func recordMemory() {  
}  
  
func finishRecording(success: Bool) {  
}  
  
func transcribeAudio(memory: URL) {  
}
```

The first one, **memoryLongPress**, is called when a long press has started or ended. The second, **recordMemory()**, is called to perform the microphone recording. The third, **finishRecording()**, is called when recording has finished, and is responsible for linking the recording to the memory. The final one, **transcribeAudio()**, handles transcribing the narration into text and linking that to the memory too. Remember, different parts of a memory – image, thumbnail, audio, and text – are linked just by having the same filename with different extensions, so this process isn't too hard.

Let's start with **memoryLongPress()**. When this is triggered it means the user has started or ended a long press on a collection view cell, so we need to distinguish that state by reading the **state** property of the gesture recognizer. If it's in the began state, we need to figure out which cell was tapped, and store the unique ID that memory away for later, so we know where to attach the finished recording. Once that's done, we call the **recordMemory()** method. If the gesture recognizer is in the **ended** state, we call the **finishRecording()** method with its success parameter set to **true**, but more on that later.

We're going to create a new property to store which memory activated the long press gesture

recognizer – add this to **MemoriesViewController** now:

```
var activeMemory: URL!
```

Figuring out which memory was selected is a two-stage process: first we convert the gesture recognizer's **view** property to a **MemoryCell**, then we can pass that to the **indexPath()** method of our collection view. That returns an optional index path because it's possible we might be trying to query a cell that doesn't exist, so we need to unwrap the result safely.

Here's the full version of **memoryLongPress()** - please put it in place of what you have right now:

```
func memoryLongPress(sender: UILongPressGestureRecognizer) {
    if sender.state == .began {
        let cell = sender.view as! MemoryCell

        if let index = collectionView?.indexPath(for: cell) {
            activeMemory = memories[index.row]
            recordMemory()
        }
    } else if sender.state == .ended {
        finishRecording(success: true)
    }
}
```

As you can see, that squirrels away the memory that was selected, then calls the **recordMemory()** method, which takes us to the next step: recording audio.

Recording photo narration

Recording audio in iOS isn't really that hard, but it does require some set up. Like I said, I covered it back in project 33, but a) you might not have read Hacking with Swift (how dare you! etc), and b) chances are you've forgotten it, because it does have some annoying little details.

Step one: we need to make **MemoriesViewController** conform to the **AVAudioRecorderDelegate** protocol. Easy!

Step two: we need to store a property for the **AVAudioRecorder** we'll be using. To make things slightly less repetitive, I'll also be adding a property to store the URL we record to – we'll record the audio to the same file each time, then move it to the correct name once it finishes successfully. This means if a user starts a re-record that fails for some reason, we don't write over their existing recording.

So, add these two properties now:

```
var audioRecorder: AVAudioRecorder?  
var recordingURL: URL!
```

You should create **recordingURL** in **viewDidLoad()** so that it's set up only once:

```
recordingURL = try?  
getDocumentsDirectory().appendingPathComponent("recording.m4a")
```

Now for step three: the **recordMemory()** method. This needs to do several things:

1. Set the background color to red so the user knows their microphone is recording.
2. Configure the app for both playing and recording audio.
3. Set up a recording session using high-quality AAC recording.
4. Create an **AVAudioRecorder** instance pointing at **recordingURL**.
5. Set our **MemoriesViewController** as the recording delegate, so we know when it's stopped.

Only point 2 is tricky, because you need to provide very specific settings to the recorder. I've inserted comments below matching the numbers above to help make it clear:

```
func recordMemory() {  
    // 1: the easy bit!  
  
    collectionView?.backgroundColor = UIColor(red: 0.5, green: 0,  
blue: 0, alpha: 1)  
  
    // this just saves me writing AVAudioSession.sharedInstance()  
everywhere  
  
    let recordingSession = AVAudioSession.sharedInstance()  
  
    do {  
        // 2. configure the session for recording and playback  
through the speaker  
  
        try  
recordingSession.setCategory(AVAudioSessionCategoryPlayAndRecord,  
with: .defaultToSpeaker)  
  
        try recordingSession.setActive(true)  
  
        // 3. set up a high-quality recording session  
  
        let settings = [  
            AVFormatIDKey: Int(kAudioFormatMPEG4AAC),  
            AVSampleRateKey: 44100,  
            AVNumberOfChannelsKey: 2,  
            AVEncoderAudioQualityKey: AVAudioQuality.high.rawValue  
        ]  
  
        // 4. create the audio recording, and assign ourselves as the  
delegate  
  
        audioRecorder = try AVAudioRecorder(url: recordingURL,  
settings: settings)  
        audioRecorder?.delegate = self
```

```
        audioRecorder?.record()  
    } catch let error {  
        // failed to record!  
        print("Failed to record: \(error)")  
        finishRecording(success: false)  
    }  
}
```

Being the delegate of the **AVAudioRecording** is easy enough because it doesn't have any required methods that must be implemented. However, we really ought to catch when the recording got terminated by the system, e.g. if a phone call came in. To do that, add this method:

```
func audioRecorderDidFinishRecording(_ recorder: AVAudioRecorder,  
successfully flag: Bool) {  
    if !flag {  
        finishRecording(success: false)  
    }  
}
```

All that does is pass the recorder's own "finish recording" method onto our **finishRecording()**, marking that it did not succeed.

Linking audio to a memory

At this point recording is working, but our work isn't complete yet because we don't link the recorded audio back to the memory it's attached to. This is where the **finishRecording()** method comes in: if it gets called with its **success** parameter set to **true** it means the recording is ready to finish cleanly and we should move the recording to the correct filename – i.e., name-of-memory.m4a. This method also needs to put the background color back to dark gray, and kick off the transcription process.

So, in all these are the tasks:

1. Set the background color back to normal.
2. Stop the recording if it isn't already stopped.
3. If the recording was successful, we need to create a file URL out of the active memory URL plus "m4a"
4. If a recording already exists there, we need to delete it because you can't move a file over one that already exists.
5. Move our recorded file (stored at the URL we put in **recordingURL**) into the memory's audio URL.
6. Start the transcription process.

There are a couple of calls that can throw errors, so a large part of the method needs to be wrapped with try/catch, but otherwise I think it's relatively straightforward.

Here's the code, with numbers matching the list above:

```
func finishRecording(success: Bool) {  
    // 1  
    collectionView?.backgroundColor = UIColor.darkGray()  
  
    // 2  
    audioRecorder?.stop()  
  
    if success {  
        do {  
            // 3  
            let memoryAudioURL = try  
activeMemory.appendingPathExtension("m4a")  
            let fm = FileManager.default  
  
            // 4  
            if fm.fileExists(atPath: memoryAudioURL.path!) {  
                // 5  
                fm.removeItem(at: memoryAudioURL)  
            }  
        } catch {  
            print("Error: \(error.localizedDescription)")  
        }  
    }  
}
```

```

        try fm.removeItem(at: memoryAudioURL)
    }

    // 5

    try fm.moveItem(at: recordingURL, to: memoryAudioURL)

    // 6
    transcribeAudio(memory: activeMemory)
} catch let error {
    print("Failure finishing recording: \(error)")
}
}

```

Converting voice to text

Speech transcription is one of my favorite new features in iOS 10, partly because it's so easy to use, and partly because... well, it's *speech transcription* – how awesome is that?

We have an empty stub for **transcribeAudio()**, and we're going to flesh it out now. It gets passed the root path to a memory – i.e., the memory's unique identifier, without ".jpg" or ".m4a" attached. We already have helper methods that convert that into an audio URL and a transcription URL, helpfully named **audioURL()** and **transcriptionURL()**.

The interesting part of this method is where it triggers the speech transcription. This is done using several new classes: **SFSpeechRecognizer** is responsible for managing the whole process, **SFSpeechURLRecognitionRequest** is responsible for loading one file at a specific URL and transcribing it, and **SFSpeechRecognitionResult** is what you get back when the transcription has completed.

There are two important things you need to know when doing transcription:

1. It runs asynchronously, so you give it a closure that will be run when transcription data comes back.
2. Your closure will get called several times, because transcription comes back in chunks as words are matched. We only care about the final, finished transcription, so we'll add code to check the **isFinal** property of the result.

Here's the new version of **transcribeAudio()**, complete with comments:

```
func transcribeAudio(memory: URL) {  
    // get paths to where the audio is, and where the transcription  
    // should be  
  
    let audio = audioURL(for: memory)  
    let transcription = transcriptionURL(for: memory)  
  
    // create a new recognizer and point it at our audio  
    let recognizer = SFSpeechRecognizer()  
    let request = SFSpeechURLRecognitionRequest(url: audio)  
  
    // start recognition!  
    recognizer?.recognitionTask(with: request) { [unowned self]  
        (result, error) in  
        // abort if we didn't get any transcription back  
        guard let result = result else {  
            print("There was an error: \(error!)")  
            return  
        }  
  
        // if we got the final transcription back, we need to write  
        // it to disk  
        if result.isFinal {  
            // pull out the best transcription...  
            let text = result.bestTranscription.formattedString
```

```
// ...and write it to disk at the correct filename for  
this memory.  
  
        do {  
  
            try text.write(to: transcription, atomically: true,  
encoding: String.Encoding.utf8)  
  
        } catch {  
  
            print("Failed to save transcription.")  
  
        }  
  
    }  
  
}
```

Now, when you enter that code yourself, Xcode will give you a warning that the **[unowned self]** isn't need because **self** isn't being used. It's right, but only for now – that will change soon enough, and for the same reason that I created the **text** property!

Side note: I used **result.bestTranscription** above, but **SFSpeechRecognitionResult** also has an array of results called **transcriptions** that stores other possible transcriptions, sorted best first. We could very easily store all of these to make the system smarter – that smells like homework to me!

Playing back our audio

You can run the app now if you want, but there isn't much point – you can't tell if the audio recording and transcription is working, because there's no way to play it back! So, a sensible next step is to add audio playback so that we can be sure the audio is working, and we can also hijack this to double-check the transcription is working too.

First, add a property store an instance of **AVAudioPlayer** that we can use for playback:

```
var audioPlayer: AVAudioPlayer?
```

Now put this at the very start of **recordMemory()**, so that if playback is in flight we stop it before recording begins:

```
audioPlayer?.stop()
```

To trigger audio playback for a cell, we're going to use the **didSelectItemAt** method of our collection view. This is going to use the **audioURL()** and **transcriptionURL()** helper methods again to get absolute URLs for the audio and transcription, which it then feeds into the **FileManager** class to check whether they exist.

If the audio file exists, we create an **AVAudioPlayer** instance out of it, stored in the **audioPlayer** property we just declared, and make it play. If the transcription exists – which it might not at first, given that it takes a second or so to complete – we're going to load it into a string and print it in the debug console.

Here's the code:

```
override func collectionView(_ collectionView: UICollectionView,
didSelectItemAt indexPath: IndexPath) {
    let memory = memories[indexPath.row]
    let fm = FileManager.default

    do {
        let audioName = audioURL(for: memory)
        let transcriptionName = transcriptionURL(for: memory)

        if fm.fileExists(atPath: audioName.path!) {
            audioPlayer = try AVAudioPlayer(contentsOf: audioName)
            audioPlayer?.play()
        }
    }
}
```

```
if fm.fileExists(atPath: transcriptionName.path!) {  
    let contents = try String(contentsOf: transcriptionName)  
    print(contents)  
}  
}  
} catch {  
    print("Error loading audio")  
}  
}
```

It doesn't do anything terribly exciting, but all being well you can run your app, record some audio, and play it back with transcription!

Like I said at the beginning of the book, the simulator is quite flaky right now, and this is one area you can expect problems – if you get crashes or other errors, try on a real device.

Searching using Spotlight

We've had that big search box in our app since the beginning, but it's been lying dormant all this time. Well, I think we're ready to change that, and we're going to change it with *style*.

In project 32 of Hacking with Swift, we used the Core Spotlight framework to store our app's data in the Spotlight search repository. We're going to use that same technique here, but with the added bonus of a new iOS 10 feature that lets us add Spotlight search to our own app. That's what we'll be using for the search bar: when a user types some letters in, we'll query our own Spotlight index to find matches.

Using this approach means that we get a double whammy: our app is compatible with Spotlight, plus our search feature springs to life without a great deal of work from us.

Indexing our content in Spotlight

When a user adds a transcription for one of their photos, we're going to store that transcription in Spotlight so that it's available for them to find even when they aren't in the app. Spotlight is exposed to use using the CoreSpotlight framework, but we'll also need to pull in MobileCoreServices because that's where the important data types are defined.

So, add these two imports to `MemoriesViewController.swift` now:

```
import CoreSpotlight
import MobileCoreServices
```

We already have a pretty clean chain of events: user long presses to start recording, releases their press to stop recording, `finishRecording()` is run to link the audio to the photo, then `transcribeAudio()` performs the transcription and links that too.

It's that last stage we need to hook into for Spotlight indexing: we need to have the transcription back so that we can store it safely away. So, in `transcribeAudio()`, find the line that starts `try text.write` and add this new line directly below:

```
self.indexMemory(memory: memory, text: text)
```

That's going to call a method we'll write in a moment. Note that it uses **self**, which is why we used **[unowned self]** earlier, despite Xcode's complaints. It gets called with the memory we're working with, along with the text of the transcription, and the job of **indexMemory()** is to store that data in Spotlight for easier searching.

In case you haven't read project 32 (impossible!) or have forgotten how Core Spotlight works, here's a brief primer:

1. You create an instance of **CSSearchableItemAttributeSet** containing the attributes you want to save, such as a title and description.
2. You put that attribute set into a **CSSearchableItem**, with an identifier that is unique to the item. You also tell Spotlight when the item is no longer considered valid.
3. You ask Core Spotlight to index that item, providing it with a closure to run when the action succeeds or fails.

And that's it – nothing too hard, I hope. There's a slight hiccup in that creating the attribute requires a tiny bit of typecasting, but otherwise there's nothing too surprising here.

Here's the code:

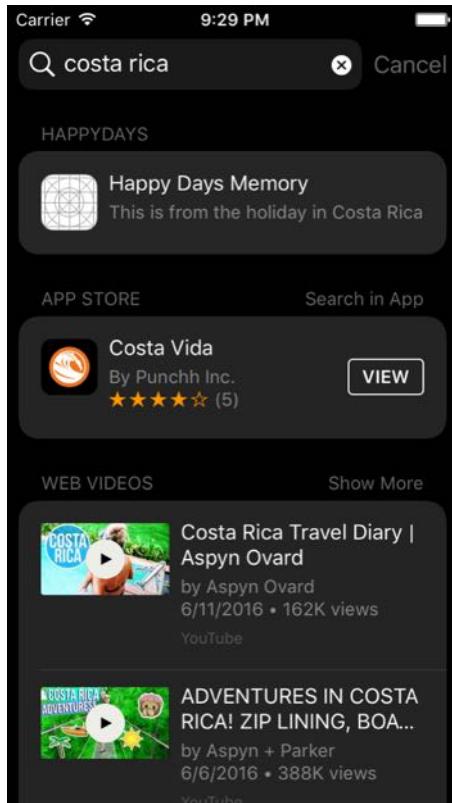
```
func indexMemory(memory: URL, text: String) {  
    // create a basic attribute set  
  
    let attributeSet = CSSearchableItemAttributeSet(itemContentType:  
kUTTypeText as String)  
  
    attributeSet.title = "Happy Days Memory"  
    attributeSet.contentDescription = text  
  
    // wrap it in a searchable item, using the memory's full path as  
    // its unique identifier  
  
    let item = CSSearchableItem(uniqueIdentifier: memory.path!,  
        domainIdentifier: "com.hackingwithswift", attributeSet: attributeSet)  
  
    // make it never expire
```

```
item.expirationDate = Date.distantFuture

// ask Spotlight to index the item
CSSearchableIndex.default().indexSearchableItems([item]) { error
in
    if let error = error {
        print("Indexing error: \(error.localizedDescription)")
    } else {
        print("Search item successfully indexed: \(text)")
    }
}
```

Using the full path to the memory as the Spotlight unique identifier is a simple hack that lets us match Spotlight data directly to our own array of memories – this will be used soon!

To test that it works, all you need to do is run the app, record something, give it a couple of seconds for the transcription to complete, then exit the app and try searching for your words in Spotlight – good job!



It's trivial to index data using Spotlight, but it adds a huge benefit to users who want to find your content wherever they are.

But wait: notice how it just shows your app icon next to the search result in Spotlight? We can do better than that, and it actually just takes one line of code! Add this in **indexMemory()**, just below the **attributeSet.contentDescription** line:

```
attributeSet.thumbnailURL = thumbnailURL(for: memory)
```

Now you'll see the image alongside the narration – much better!

In-app search using **CSSearchBar**

For our final trick, we're going to make the search bar at the top work by reading from our Spotlight index. Filtering the items means keeping an array of items that match whatever filter is currently active, then making the collection view read from that.

So, this means we need to make a pretty invasive change to our code: every time we read from the **memories** array, we need to read from a new array called **filteredMemories** instead. Create that property now:

```
var filteredMemories = [URL] ()
```

Now go to these four places in your code and replace **memories** with **filteredMemories**:

- **numberOfItemsInSection**.
- **cellForItemAt**.
- **didSelectItemAt**.
- **memoryLongPress()**.

By default, the **filteredMemories** array should contain all memories, so add this line of code in the **loadMemories()** method, just above the call to **reloadSections()**:

```
filteredMemories = memories
```

Filtering will be activated whenever the user changes the text in the search bar. We made our **MemoriesViewController** the delegate of the search bar back near the beginning, which means we'll get two methods from iOS: **textDidChange**, which we'll use to start the search, and **searchBarSearchButtonClicked()**.

The first of those is going to call a new method called **filterMemories()**, which will accept the string to search for. The second will just call **resignFirstResponder()** on the search bar, to make the keyboard go away. So, by using another stub for **filterMemories()**, we can get some code out of the way immediately:

```
func searchBar(_ searchBar: UISearchBar, textDidChange searchText: String) {
    filterMemories(text: searchText)
}
```

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
    searchBar.resignFirstResponder()
}

func filterMemories(text: String) {
```

So far, so easy. Now it's time for the new **CSSearchQuery** class, which works entirely asynchronously. We're going to give it two closures to work with: one to call when it finds a matching item (which will append the item to an array), and one to call when the search finishes, at which point we're going to update our UI with the search results.

You might be thinking that **CSSearchQuery** sounds very similar to using Core Data – and you'd be right. It even has the same approach to specifying search criteria: we're going to search for "**contentDescription == \\"*\\"(text)*\"c"**", which means "find things that have a **contentDescription** value equal to any text, followed by our search text, then any text, using case-insensitive matching.

There are a few more things you need to know before I show you the code:

1. **CSSearchQuery** returns **CSSearchableItem** items, so we need to store that data type.
2. We'll be taking advantage of closure capturing to share that array between the "found items" closure and the "search is finished" handler.
3. Your closures can be called on any thread, so as we're manipulating the UI when the search finishes I'll be pushing that work to the main thread.
4. You need to explicitly call **start()** on the search to make it begin.
5. In case a user types really fast, we want to a way to cancel the existing search before starting a new one. To make that happen, we'll be storing the **CSSearchQuery** object as a property in the class, then calling **cancel()** on it before searching.

Here's the code:

```
func filterMemories(text: String) {
```

```

var allItems = [CSSearchableItem]()

searchQuery?.cancel()

let queryString = "contentDescription == \"*\\"(text)*\"c"
searchQuery = CSSearchQuery(queryString: queryString, attributes: nil)

searchQuery?.foundItemsHandler = { items in
    allItems.append(contentsOf: items)
}

searchQuery?.completionHandler = { error in
    DispatchQueue.main.async { [unowned self] in
        self.activateFilter(matches: allItems)
    }
}

searchQuery?.start()
}

```

That won't build just yet because we haven't created the **searchQuery** property, and I also snuck in an **activateFilter()** method that figures out the results.

To fix the first problem, add this property to the **MemoriesViewController** class:

```
var searchQuery: CSSearchQuery?
```

To fix the second, add this stub for **activateFilter()**:

```
func activateFilter(matches: [CSSearchableItem]) {  
}
```

Before we fill in that method, there's one more thing I'd like to do. In the **filterMemories()** method, what happens if the user deletes their search? Right now it will perform an empty search and find no matches, which isn't very useful. To fix that, we need to add some code to the start of **filterMemories()** that checks for the search text being empty. If it *is* empty, we'll reset **filteredMemories** to the complete **memories** array, then reload the collection view. To make it a bit snappier, I've put the collection view reloading in a non-animating block.

Here's the code – put this at the start of **filterMemories()**:

```
guard text.characters.count > 0 else {  
    filteredMemories = memories  
  
    UIView.performWithoutAnimation {  
        collectionView?.reloadSections(IndexSet(integer: 1))  
    }  
  
    return  
}
```

That just leaves one more method and this project is complete: once we have the array of matching items back from Spotlight, we need to create a new **filteredMemories** array based on those results.

This can be done using a single call to **map** on the **matches** array we built from Spotlight. Because we used the full path to the memory as the Spotlight unique identifier, all we have to do is convert that to a fully formed file URL and we're done. As with clearing the collection view on empty searches, I've put the **reloadSections()** call into a non-animating block so that it doesn't fade in and out while the user is typing.

Here's the code for **activateFilter()**:

```
func activateFilter(matches: [CSSearchableItem]) {
    filteredMemories = matches.map { item in
        return URL(fileURLWithPath: item.uniqueIdentifier)
    }

    UIView.performWithoutAnimation {
        collectionView?.reloadSections(IndexSet(integer: 1))
    }
}
```

That's it! I hope you'll agree that Spotlight bears a huge amount of the work, and being able to unify Spotlight search and in-app search is smart, efficient, and powerful.

Wrap up

I hope you're pleased with what you produced: it's an app that works with a variety of device permissions, imports photos, attaches recordings, transcribes those recordings, then puts the whole lot into Spotlight for searching.

Yes, it took quite a lot of work – and a fair amount of Interface Builder time too – but you're already using two fantastic new features from iOS 10, while getting in some practice with **UIStackView**, **UIImagePickerController**, **AVAudioPlayer**, and more.

This is just the first project in this book, and I hope it's whetted your appetite for iOS 10. If you spot any mistakes or typos, or have any suggestions for how I can improve the code – making it simpler is just as useful as making it more efficient – please get in touch.

And of course, watch your inbox because you can expect the next installment in 48 hours!

Thanks for your support! If you enjoyed this first project, don't be afraid to tell your friends about the book – they can buy it from <https://gum.co/ios10>.

Homework

It's time for you to try stretching your legs and have a go at manipulating this project in interesting ways.

First, and most importantly: storing the full path to the item in Spotlight isn't a smart idea, as it could change under our feet and cause problems. Try to modify your code so that it stores only the unique memory name rather than the full path. To make this work, look at the **indexMemory()** method, in particular this line:

```
let item = CSSearchableItem(uniqueIdentifier: memory.path!,  
domainIdentifier: "com.hackingwithswift", attributeSet: attributeSet)
```

What could be used instead of **memory.path!** so that it stores only the partial path?
Whatever you choose, remember to modify **activateFilter()** as well, so that loading works.

Second, given that transcription is notoriously difficult, can you think of a way to store and search the alternate transcriptions from the **SFSpeechRecognitionResult** result? You might need to do a bit of research, but it should help solidify your knowledge of Core Spotlight!

Project 2

TimeShare

Setting up

iOS 10 introduced a wholly new kind of application type in the form of iMessage Apps, which are extensions to the Messages app that provide custom content and functionality. These extensions aren't like the other extension points Apple has opened before, because they *don't* need to be contained in an app in order to ship: they have their own dedicated App Store for iMessage, so while you *can* write an app to go alongside it's not necessary.

All this functionality is powered through a new Messages framework, which integrates into other system frameworks such as StoreKit for in-app purchase and Apple Pay for cash transfers. This gives developers like us a massive, new opportunity to deliver something fun, interesting, or productive for users, so I hope you're keen to try it out.

In this project we're going to build an iMessage App called TimeShare. It will allow one person to create a list of dates that an event could take place on, and send that list to other people in the chat. They will be able to see how many votes have been cast for each date, then add their own votes. In a group chat, this should mean that a group can pick the best date from various options, although if you have friends like mine then you might find that brute force is the only thing that will get them to agree!

Before we start, I have three important warnings:

1. iMessage Apps only run on iOS 10, but they can be viewed on watchOS and macOS. In Apple's perfect world, all messages have a unique URL attached to them that can be viewed using a web browser on macOS, but setting up remote hosting is clearly outside the remit of this tutorial.
2. The iOS 10 simulator includes a special version of the Messages app that lets you send and receive messages between two test participants. This makes debugging significantly easier because it all runs through the simulator, but you need to prepare your brain because you're both the sender and receiver of messages at the same time.
3. The current iOS 10 beta has many warnings no matter what you do, but the Messages framework is particularly talkative. Be prepared to have your Xcode log filled with Apple's own log messages – if you want your own messages to be useful, prefix them with "HELLO" or some obvious string that you can filter effectively.
4. If you want to make this into a real app, make sure you pay close attention to the icon requirements for iMessage Apps because they *aren't* the regular square items you're used to.

That's it!

Enough talk: let's start coding. Fire up Xcode 8, then create a new application. This time, I'd like you to select the Messages Application, name it TimeShare, then save it on your desktop.

Building the user interface

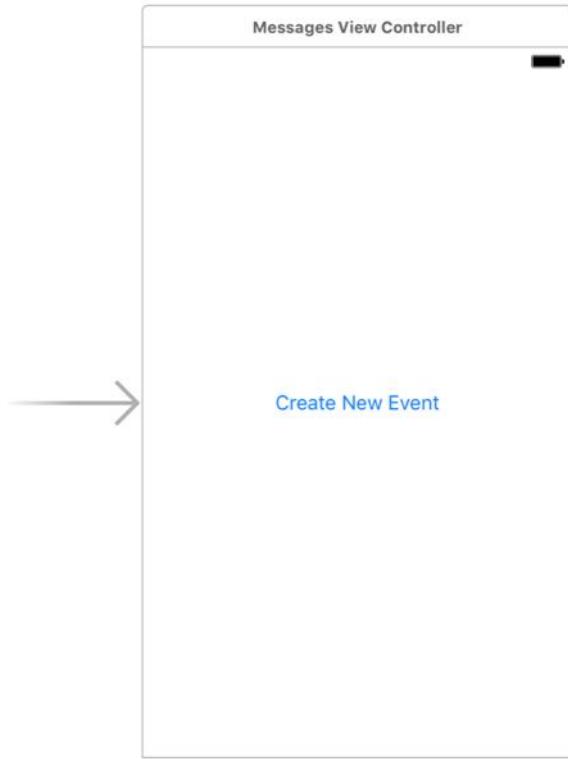
As with project 1, we're going to start by creating the user interface for the extension so that you can see exactly how the application fits together.

What you'll see is that Xcode has created a group called TimeShare, and another group called MessagesExtension. The first group is where you'll place your app if you want to create something separate, but realistically you'll be spending most of your time in the MessagesExtension.

Open MainInterface.storyboard now, and you'll see the template has given you a single view controller and a label in the middle. You'll know that I'm a big fan of using navigation controllers as a smart and simple way to keep functionality organized, but here we're not going to use one because our extension will run inside the Messages app – it has its own navigation bar, and we need to leave that alone.

This app will have three view controllers: one that triggers creating new events, one that lets someone choose which dates to add to the event and cast votes for their favorites, and one that lets others only cast votes.

We'll use the existing view controller to trigger creating new events, because it's what's shown when our extension first appears. So, delete the label, and replace it with a button that has the text "Create New Event". Ctrl-drag from the button to the parent view to give it the constraints Center Horizontally In Container and Center Vertically in Container, then add an explicit height constraint so that it's always 44 points high. Chances are it will be 30 points high by default, so just create the height constraint then edit its constant value.

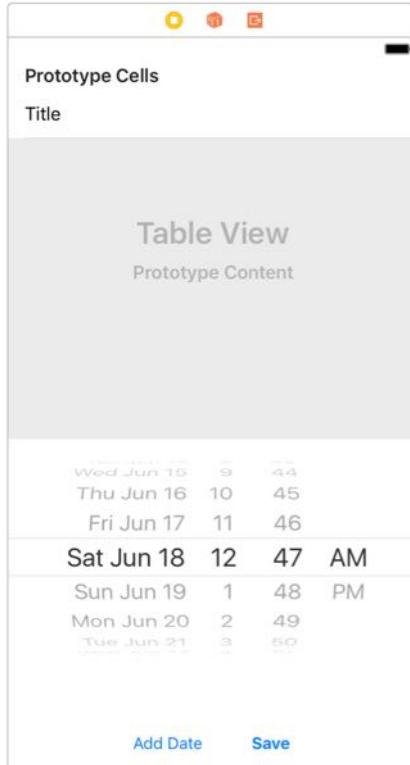


The first view controller for our project is simple: just one button aligned to the center horizontally and vertically.

That's the first view controller done already, but the next one won't be quite so easy!

Creating an event

Drag a new view controller next to the first one, then use the identity inspector to give it the storyboard identifier “CreateEvent”. This is where users can create a new event, which means adding a list of possible dates to a table view, then sending that list to friends.



The second view controller needs to have a table view, a date picker, and some buttons inside a stack view.

I want you to add three things to this view controller: a table view near the top, a date picker below that, then a horizontal stack view near the bottom. Don't worry about sizes for now; we'll get to that in a moment. Next, please add two buttons to the stack view: the first needs the title "Add Date" and the second needs the title "Save". I found it looked better to make the Save button use a bold font to make it clear it's the main action.

We're going to add some Auto Layout constraints in just a moment, but first we need to make a small workaround. In my tests, Auto Layout and **UIDatePicker** didn't play nicely together inside Messages extensions – it might just have been a random glitch, it might be a beta bug that will go away soon, or it might be something a bit more sinister. Either way, I found that embedded the date picker inside a **UIView** made the problem go away, so please select the date picker then go to Editor > Embed In > View.

At this point we have three top-level objects: the table view, a regular view containing the date picker, then a stack view at the bottom. We're going to make the date picker view and the stack view a fixed size, leaving the table view take up the rest of the space.

Note: For the avoidance of doubt: when I say “the date picker view” I mean the view that wraps the date picker, not the date picker itself. When I want you to adjust the date picker, I’ll say “the date picker”. Please be careful!

Start by selecting the stack view, and Ctrl-drag from there to the parent view to give it the constraints Center Horizontally In Container and Vertical Spacing To Bottom Layout Guide. By default the bottom space constraint will be set to whatever number matched where you placed the stack view, but I’d like you to change it to 0. I would also like you to give it a fixed height of 44 points, so the buttons are easy to tap. You should also give the Save button a fixed width of 44 points for the same reason. The combination of these three constraints should place the stack view centered at the bottom of the view, with a fixed height. I suggest you use the attributes inspector to add some spacing to the stack view – a value of about 40 looked good for me, but you’re welcome to experiment.

Next, select the date picker view (*not* the date picker itself, remember!) using the document outline, then Ctrl-drag to itself and choose Height. In my tests, Xcode liked to give this a constant of 256, but you should change it to 216 so that it wraps the date picker tightly. Now Ctrl-drag from the date picker view to its parent view, then add the constraints Leading Space To Container and Trailing Space to Container. Finally, Ctrl-drag from the date picker view down to the stack view, and give it a Vertical Spacing constraint. You might find the Leading, Trailing, and Bottom Space constraints have unusual constants values by default – mine had -20, -20, and 36 by default – but you should just edit them all to 0.

Finally, select the table view then Ctrl-drag from there to the parent view to give it the constraints Leading Space To Container, Trailing Space To Container, and Vertical Spacing to Top Layout Guide. It still needs a constraint for height, but because we’ve fixed the date picker view and stack view in height, we can ask the table view to occupy the rest of the space by aligning its bottom to the date picker view’s top. To do that, Ctrl-drag from the table view to the date picker view, then choose Vertical Spacing. As per usual, all the constraints we just created will have various constants depending on where you placed the table view, so go to the size inspector and change them all to 0.

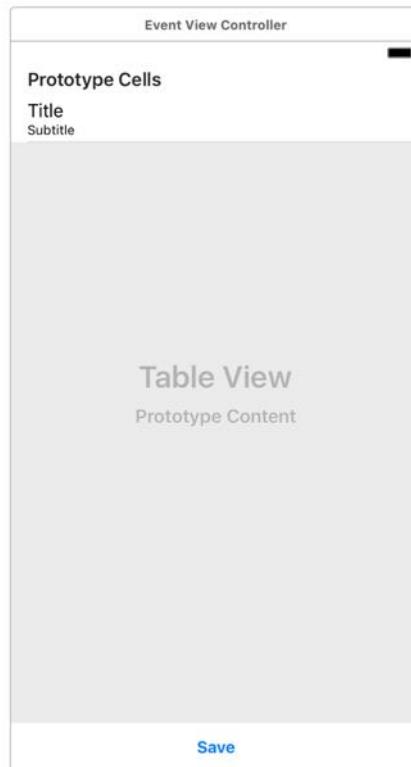
That completes all the layout constraints for this view controller, so select the main view in the document outline then go to Editor > Resolve Auto Layout Issues > Update Frames, making sure to choose the option under “All Views”. Hopefully your UI is looking pretty good at this point, but there might be one small mistake: you might find that the date picker (*not* the date picker view) has a strange frame. To fix it, select the date picker itself, then change its X and Y values to 0, and its width to the same value as the view that contains it. To ensure

the date picker resizes correctly, give it a flexible with autoresizing mask, and pin it to the left and right edges.

We need to do one more thing to this view before we're done, which is to create a prototype cell. This is nothing special: just select the table view, and change its Prototype Cells value to 1. Now select the prototype cell that was created, give it the identifier "Date", and the Style Basic.

Selecting dates

The final view controller we're going to create is effectively a simpler version of the one we just defined because it just needs to have a table view and a single button.



The third view controller is a simplified version of the second: just a table view and a single button.

Drag one last view controller into your storyboard, and give it the storyboard identifier "SelectDates". Drag a table view near the top, and a button near the bottom – that's all we

need for this view.

Give the button the title Save, and again give it a bold font. Give it the same constraints to the stack view we made earlier: exactly 44 points high, centered in the parent view, and with 0 vertical spacing to the bottom layout guide. Again, give the button a 44-point fixed width – Apple consistently recommends making every tappable element at least 44x44 in size, so it's beyond me why IB creates buttons so small by default by default.

For the table, Ctrl-drag to the parent view to give it the constraints Leading Space To Container, Trailing Space To Container, then Vertical Space To Top Layout Guide. Now Ctrl-drag from the table to the button, and select Vertical Spacing. As per usual, use the size inspector to set all the constants to 0, then use Resolve Auto Layout Issues to make the UI update to reflect our changes.

To finish up this view, select the table and give it one cell prototype. Give the cell the identifier “Date” and the style “Subtitle” – this is different from the cell prototype in the other view.

Hooking it all up

That's all our user interface design complete, but before we're done with IB we need to hook up our designs to some code. With three view controllers and two table view cells you might expect that we need to create a variety of classes to back all that up.

Instead, in my quest to keep things simple, we're going to create just one class: the table view cells don't need to do anything special, the initial view controller already has a class to represent it (in `MessagesViewController.swift`), and the two view controllers we created can both be backed by a single new view controller because they are so similar in functionality.

So, go to File > New > File, then select Cocoa Touch Class and click Next. Make it a subclass of **UIViewController**, name it “EventViewController”, then click Next. In the final screen, make sure the Group option is set to “MessagesExtension”, then click Create.

With that new class in place, I want you to set the class property for both of our new view controllers to be EventViewController: two different view controllers will point at the same class. This is possible because they both share the same table view and Save buttons, so there's no point repeating the code in two places.

We need to create a few outlets and actions before we're done. First, Ctrl-drag from one of the tables to the view controller to make it the data source and delegate, then repeat for the other table. Now switch to the assistant editor and create outlets for tableView (for the table view) and datePicker (for the date picker, *not* the date picker view), as well as an action for the button called `saveSelectedDates()`. If you do this for the first view controller, you should be able to connect the second one up to the same outlets and action, because they both share the same class. You also need to create one extra action for the “Add Date” button, called `addDate()`.

The last thing to do is connect the Create New Event button to a new action in `MessagesViewController.swift`, called `createNewEvent()`.

Finally, we’re done with the the user interface, so we can get stuck into some actual coding...

Working with MSMessagesAppViewController

When we created the Messages Application template, Xcode gave us a single view controller in `MessagesViewController.swift`. This class, `MessagesViewController`, inherits from `MSMessagesAppViewController`, which is a `UIViewController` subclass that gets displayed inside the Messages app. Because we inherit from `MSMessagesAppViewController`, we automatically get access to a number of useful properties and methods:

- `activeConversation` represents the conversation that is active between the current user and one or more contact. The conversation is the entire history of chats between those people.
- `presentationStyle` stores how your view controller is currently being displayed. If you're in compact mode you occupy a small space where the keyboard normally sits; if you're in expanded mode you occupy almost all the screen.
- `requestPresentationStyle()` lets you request a change from compact to expanded mode, or vice versa, depending on user input.
- `dismiss()` can be used when your app is finished working, and you want the keyboard to appear.

Let's put some of those into practice now. When our app runs, it will start life in the bottom of the screen, with the user's active conversation above – that's all the text bubbles between them and one or more contacts. This is compact mode, and by default our app will show the Create New Event button we made earlier.



There's our view controller, nestled at the bottom of the **Messages** app where the keyboard normally is.

When that's tapped, we want to switch to expanded mode so the user can start adding dates to a new event. So, add this line of code to the **createNewEvent()** method in **MessagesViewController.swift**:

```
requestPresentationStyle(.expanded)
```

That will cause our app to move from compact to expanded layout, so clearly you don't want to do that unless the user expects it!

When our app moves between presentation styles we'll get informed in a method called **willTransition(to presentationStyle:)**, which will either be passed **.compact** or **.expanded** depending on how the app is changing. Initially the user sees just one button – Create New Event – and we just made tapping that switch the app into expanded mode, which means our **willTransition()** method will get called with **.expanded**, and it's down to us to display the

correct UI.

Now, there's a slight complication: the same instance of our **MessagesViewController** is used in compact and expanded mode, which means we need to display our content directly inside that rather than pushing and popping new view controllers. Obviously we don't want to put all our app's functionality in a single view, so instead we're going to use child view controllers: we're going to create an instance of the correct view controller, then place it directly inside the existing **MessagesViewController** instance.

Embedding one view controller inside another is called view controller containment, and you might never have used it before – even though it's been around since iOS 5, it doesn't really get that much use. Doing view controller containment properly involves multiple steps:

1. Creating the child view controller.
2. Calling **addChildViewController()** on the parent, passing in the child. This ensures important messages, such as rotation, are forwarded from parent to child.
3. Give the child's view a meaningful frame within the parent.
4. Add Auto Layout constraints so the child can adapt to changes in the parent layout.
5. When you're finished, call **didMove(toParentViewController:)** on the child.

In our case, we have one extra step: before we do any of that, we're going to perform a quick sanity check to ensure we have an active conversation to work with. I already said we get a **activeConversation** property to work with, but it's optional – it might not exist. Clearly we don't want to do any sort of messaging if somehow the whole conversation doesn't exist, so I'm going to add a step 0 to the above list: check and unwrap the active conversation.

We can put the list above, including the extra step 0, into code without too much hassle. But before we do that, you'll notice I made this method accept two parameters: a conversation to work with, and an identifier that decides which controller to show. The first parameter is required because our **activeConversation** property isn't always available when we're required to load, so by passing it in we can be sure it exists. The second parameter is required because we have two possible story board view controllers to choose from, and the second parameter decides which is shown.

Time for the code – add this into the **MessagesViewController** class:

```
func displayEventViewController(conversation: MSConversation?,  
identifier: String) {
```

```

// 0: sanity check, is there a conversation?
guard let conversation = conversation else { return }

// 1: create the child view controller
guard let vc =
storyboard?.instantiateViewController(withIdentifier: identifier) as?
EventViewController else { return }

// 2: add the child to the parent so that events are forwarded
addChildViewController(vc)

// 3: give the child a meaningful frame: make it fill our view
vc.view.frame = view.bounds
vc.view.translatesAutoresizingMaskIntoConstraints = false
view.addSubview(vc.view)

// 4: add Auto Layout constraints so the child view continues to
fill the full view
    vc.view.leftAnchor.constraint(equalTo: view.leftAnchor).isActive
= true

    vc.view.rightAnchor.constraint(equalTo:
view.rightAnchor).isActive = true

    vc.view.topAnchor.constraint(equalTo: view.topAnchor).isActive =
true

    vc.view.bottomAnchor.constraint(equalTo:
view.bottomAnchor).isActive = true

// 5: tell the child it has now moved to a new parent view
controller
    vc.didMove(toParentViewController: self)
}

```

You'll get a warning because we're not using the **conversation** constant just yet, but that will

be coming later on. Before that, I want to focus on calling the new **displayEventViewController** from inside **willTransition()** so you can start to see the app working.

It's possible – indeed likely – for the **willTransition()** method to be called multiple times, because the user might move in and out of expanded mode as much as they want to. As a result, we need an extra step: before creating any child view controllers we need to clear out any that already exist, effectively resetting our main controller before we do any fresh work.

You should have an existing stub for **willTransition()** inside `MessagesViewController.swift`, but I'd like you to replace it with this new code:

```
override func willTransition(to presentationStyle: MSMessagesAppPresentationStyle) {
    for child in childViewControllers {
        child.willMove(toParentViewController: nil)
        child.view.removeFromSuperview()
        child.removeFromParentViewController()
    }

    if presentationStyle == .expanded {
        displayEventViewController(conversation: activeConversation,
                                  identifier: "CreateEvent")
    }
}
```

This demonstrates the other half of view controller containment, so let me walk through what it's doing:

1. It runs when we begin transitioning from one presentation style to another, e.g. from compact to expanded.
2. It loops through all the view controllers that belong to the current view controller.
3. For any it finds, it tells the child it's about to have no parent view controller.
4. It then removes the child's view from its parent, and removes the child itself from *its*

parent.

Once that loop is complete, the code checks the value of **presentationStyle** to see whether we're entering expanded mode, and, if so, it calls the new **displayEventViewController()** method and gives it the current active conversation (provided by **MSMessagesAppViewController**) and the storyboard identifier "CreateEvent" to kick off event creation.

You *could* run the code now, but you'll find tapping the Create New Event button will crash. Worse, chances are you'll see no meaningful reason for the crash inside the Xcode log – it will give you absolutely no idea why the code isn't working. Hurray for betas!

Fortunately, I'm here to tell you exactly why it's crashing: we made **EventViewController** the data source and delegate of our table views, but didn't actually write the code for that. So, iOS tries to ask the view controller how many sections should be in the table, but the corresponding method doesn't exist and throws an exception.

Let's fix that now: add **UITableViewDataSource** and **UITableViewDelegate** to the list of protocols that **EventViewController** conforms to, then add these function stubs so the code compiles:

```
func numberOfSections(in tableView: UITableView) -> Int {  
    return 1  
}  
  
func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {  
    return 1  
}  
  
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {  
    let cell = tableView.dequeueReusableCell(withIdentifier: "Date",  
for: indexPath)  
    cell.textLabel?.text = "Date goes here"  
}
```

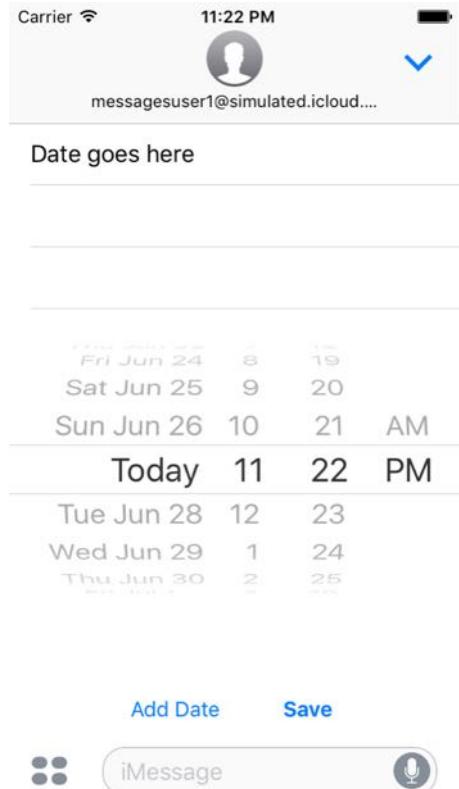
```
    return cell  
}
```

With that change the code ought to run and – bonus! – not crash when you click the Create New Event button. To try it out, press Cmd+R to build and run your code, then choose Messages for the debug target when Xcode prompts you.

When the simulator first loads you'll see two iMessage accounts inside the Messages app, but they are both linked like two ends of a pipe: what one sends the other receives, and vice versa. Select the first of them to enter the regular Messages view, then look for the oval-shaped App Store icon – it might be hidden behind a chevron button to the left of the message text field. Tapping that will open the list of iMessage Apps, and you'll see in the bottom-left corner an icon showing four ovals: tap that to bring up the list of installed apps, then tap “MessagesExtension” (probably shortened to “MessagesE...”) to launch our extension.

And now: wait. Yes, do nothing. Messages will load our extension, and Xcode will detect that load and try to attach its debugger. This can take a couple of seconds, so you should wait for it to complete before continuing. You'll know it's safe to complete when a large chunk of debugging messages appear in your Xcode debugging window – it's ugly, but at least we know when we're attached and ready to go.

After a few seconds have passed, you'll see the Create New Event button inside Messages. Even better, tapping it should cause our CreateEvent view controller to appear, showing one row in the table, the date picker, plus buttons to add and save.



It's not functional yet, but at least we're headed in the right direction.

Selecting possible dates

At this point we've created the user interface for the whole app, and users can get to the point where they can see our event creation view. The next step is to let them create an event by using the date picker.

As a reminder, the user creating the event can add as many dates as they want. They then select which dates they prefer, and send those off to other users to look at and vote on. We'll tackle the first half of that now: we have the CreateEvent view showing, so let's make it work.

An event is made up of several things: the list of dates that are on offer, the list of votes from everyone in the group, plus the list of our votes that we intend to cast. That last one is important, because the user might spend a minute or so on this screen, scrolling around the table view trying to make their mind up - we need to keep track of the days they want separately from the public list of votes so we know which dates they like.

We're going to represent those values as three properties in **EventViewController**, so please add these now:

```
var dates = [Date]()  
var allVotes = [Int]()  
var ourVotes = [Int]()
```

Each of those arrays should have exactly the same number of items, because each element matches an element in the same position in the other arrays. That is, the date at **dates[0]** matches the number of votes in **votes[0]**. I've made **ourVotes** an array of integers just because it makes calculating the final votes array easier. If we had used booleans instead, we'd need to check each value when adding **ourVotes** to **allVotes**.

With the **dates** in place, you can change **numberOfRowsInSection** to return the total number of dates on offer, like this:

```
return dates.count
```

When in this view, the user will slide around the date picker until they find a date and time they like, then tap Add Date. This will happen repeatedly, and each time it does we need to add the date that's currently in the date picker to the **date** array. This happens inside the **addDate()** method we created earlier, but we're going to make it a little smarter:

1. Yes, we add the current date to the **dates** array, but we'll also add 0 to the **allVotes** array because the new date has no votes, and 1 to the **ourVotes** array so the event creator votes for it by default.
2. We'll use the **insertRows()** method to animate a new row sliding into the table.
3. We'll also call the **scrollToRow()** method on the table view so that the new row becomes visible to the user.
4. Just to make absolutely sure the user is aware something happened, we'll also call the **flashScrollIndicators()** method to bring the user's attention to the changed table.

Here's the code, with numbered comments:

```
@IBAction func addDate(_ sender: AnyObject) {  
    // 1: add to the arrays  
    dates.append(datePicker.date)  
    allVotes.append(0)  
    ourVotes.append(1)  
  
    // 2: insert a row in the table using animation  
    let newIndexPath = IndexPath(row: dates.count - 1, section: 0)  
    tableView.insertRows(at: [newIndexPath], with: .automatic)  
  
    // 3: scroll the new row into view  
    tableView.scrollToRow(at: newIndexPath, at: .bottom, animated: true)  
  
    // 4: flash the scroll bars so the user knows something has changed  
    tableView.flashScrollIndicators()  
}
```

Showing user dates

Our code so far lets the user add lots of date rows to the array, and will also animate new rows appearing in the table, but it won't actually show any dates just yet because our current `cellForRowAt` method just writes "Date goes here" again and again.

Let's change that now. We're going to make the table show all the dates in the array, one per row. If a date has any votes cast for it already, we'll show that number below as subtitle text. If you remember, we made the `CreateEvent` table use the `Basic` cell style, and the `SelectDates` table use the `Subtitle` cell style. This is because when you're creating a new event it doesn't have any votes yet, but thanks to the power of optional chaining we don't need to worry whether there's a subtitle text label or not – we can just try to set it, and Swift will silently ignore the call if it doesn't exist.

So, the `cellForRowAt` needs to optionally show a subtitle with a vote count. At the same time, we also need to let users select which dates they prefer, and for that we're going to use the `accessoryType` property to create a checkmark next to rows they have voted for.

There's one more thing this method needs to do, and it's the most important thing: it needs to display the date in the table view cell. We could just print the `Date` object directly, but that looks nasty. Instead we're going to use the `DateFormatter` class to create a localized date string in the user's preferred style.

In total, then, the method needs to do quite a bit:

1. Dequeue a reusable cell with the Date identifier.
2. Pull out the correct date for the cell, then format it as a string.
3. If we voted for this date, give it a checkmark.
4. If other people voted for this date, show the vote count.

You might not have used the `DateFormatter.localizedString()` method before, but it's a little bit of Cocoa magic: you tell it what kind of style you want for date and time (short, long, etc), and it converts a `Date` object into a string that matches the user's region and language settings at the length you specified.

The other thing to watch out for is the optional chaining used to modify `cell.detailTextLabel` – remember, only one of the two table view cell prototypes has a subtitle, so we need to be careful when writing a subtitle.

Here's the new `cellForRowAt` for `EventViewController`:

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    // 1: dequeue a cell
    let cell = tableView.dequeueReusableCell(withIdentifier: "Date",
for: indexPath)

    // pull out the corresponding date and format it neatly
    let date = dates[indexPath.row]
    cell.textLabel?.text = DateFormatter.localizedString(from: date,
dateStyle: .long, timeStyle: .short)

    // add a checkmark if we voted for this date
    if ourVotes[indexPath.row] == 1 {
        cell.accessoryType = .checkmark
    } else {
        cell.accessoryType = .none
    }

    // add a vote count if other people voted for this date
    if allVotes[indexPath.row] > 0 {
        cell.detailTextLabel?.text = "Votes: \
(allVotes[indexPath.row])"
    } else {
        cell.detailTextLabel?.text = ""
    }
}
```

```
    return cell  
}
```

Selecting preferred dates

If you run the app now you'll see that the date picker works as intended: you can choose a date, tap Add Date, choose another one, tap Add Date, and so on, ultimately building a list of dates that the group ought to consider.

Right now whenever a user adds a date we're assuming they are OK with it, but it's possible they are being altruistic and add dates they themselves cannot attend. We also need to cater for other users who receive the message: they need to be able to cast votes for particular dates they prefer.

We've already written the code to *display* a user's selection by looking in the `ourVotes` array, so now we just need to write the code to let them make that selection. The easiest way to make that happen is with the `didSelectRowAt` method for the table view, following some basic logic:

1. Deselect the row that was tapped; we don't want it to stay selected.
2. Find the cell that was tapped and query whether it has a checkmark right now
3. If it does, remove the checkmark, and set the corresponding element in `ourVotes` to 0.
4. If it doesn't have a checkmark, add one, then set the corresponding element in `ourVotes` to 1.

That's it – all we're doing is flipping a few bits, because the real work happens in the `cellForRowAt` method.

Here's the `didSelectRowAt` code – add this to `EventViewController`:

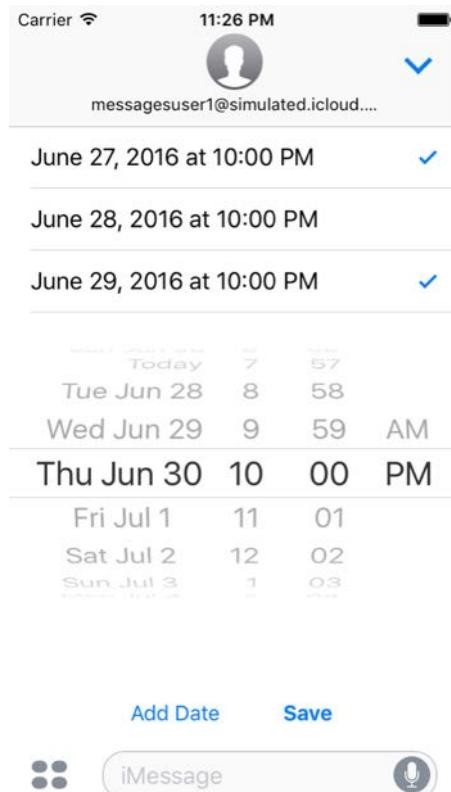
```
func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {  
    tableView.deselectRow(at: indexPath, animated: true)
```

```

        if let cell = tableView.cellForRow(at: indexPath) {
            if cell.accessoryType == .checkmark {
                cell.accessoryType = .none
                ourVotes[indexPath.row] = 0
            } else {
                cell.accessoryType = .checkmark
                ourVotes[indexPath.row] = 1
            }
        }
    }
}

```

If you run the project *now* you'll see that you can add all the dates that the group might want, then deselect any you personally don't like. That's pretty much all we need to proceed to the next step, which is where things get more interesting: sending our message.



You can now create events, select which dates are possible, and highlight those you prefer.

Sending and receiving messages

Our app now has a complete user interface design, plus the ability to create events and select dates. However, the crux of the app – sending and receiving those dates – is still not written, and that's our next task.

In the previous chapter we filled in the **EventViewController** class so that it keeps track of dates, all votes, and our votes, so the immediate next step is to make the Save button work so that it adds our votes to all the votes and returns that new array back to the **MessagesViewController** instance.

There are lots of ways to pass messages from one view controller to another, but the easiest and arguably cleanest is to use delegates. So, please add this property to **EventViewController**:

```
weak var delegate: MessagesViewController!
```

We're going to assign that property as soon as the **EventViewController** is created, so add this next line to the **displayEventViewController()** method of **MessagesViewController**, just before the call to **addChildViewController()**:

```
vc.delegate = self
```

While you're in **MessagesViewController**, I'd like you to add the stub for a new method called **createMessage()** – we'll be calling this from the **EventViewController** when the user saves their changes. Add this method now:

```
func createMessage(with dates: [Date], votes: [Int]) {  
}
```

As you can see, it takes an array of dates and an array of votes, but for now we're just going to leave it empty.

With a delegate set and a stub that we can call in place, it's time to turn to the **saveSelectedDates()** method in **EventViewController**. We added this a while ago as an empty action from Interface Builder, but now it's time to fill it in.

The **EventViewController** class has three properties: **dates**, containing a list of **Date** objects; **allVotes**, containing a list of integers that match how many votes have been cast for each date; and **ourVotes**, containing a list of integers that match how we voted. The delegate's callback method we just wrote a stub for accepts an array of dates and an array of votes, which means it's down to **EventViewController** to merge the **allVotes** and **ourVotes** arrays before passing it on.

Fortunately, this isn't too hard to do: **allVotes** and **ourVotes** contain the same number of items, so because **ourVotes** contains integers all we need to do is add **allVotes[0]** to **ourVotes[0]**, then **allVotes[1]** to **ourVotes[1]**, and so on. That will create an array of total votes, including ours, which can then go back to the delegate.

Here's the new version of **saveSelectedDates()** for **EventViewController**:

```
@IBAction func saveSelectedDates(_ sender: AnyObject) {
    var finalVotes = [Int]()
    for (index, votes) in allVotes.enumerated() {
        finalVotes.append(votes + ourVotes[index])
    }
    delegate.createMessage(with: dates, votes: finalVotes)
}
```

Easy, right? Well, yes, but that's just the beginning...

Encoding data using URLQueryItem

Here's where things get trickier: the `createMessage()` method receives an array of dates and votes, but we can't send that over iMessage. Instead, we need to convert everything – all the data we want to send – into a series of URL items because that's how the message will be transmitted.

To put that into perspective, you need to imagine a URL like the one below:

`http://www.example.com/somepage?name=paul&fizz=buzz`

The parts after the question mark are part of the query string: there's a parameter called "name" with the value "paul", and another parameter called "fizz" with the value "buzz". That's how our data will be transferred over iMessage.

Now, that might seem painful, but it's not quite as bad as you might imagine. Apple provides two helper classes that make it easy – or at least easier – to construct these URLs, and we'll be using them now. The first class is called `URLQueryItem`, and is created from a key-value pair like "name" and "paul". If either the name or value contain characters that don't work well in URLs, `URLQueryItem` will automatically escape them so they *do* work. The second class is called `URLComponents`, and is responsible for taking a whole array of `URLQueryItem` objects, along with any other URL components, and creating a valid `URL` object out of it.

Once we have our dates and votes as a URL, the next step is to convert that into a message that can be sent using iMessage. Apple gives us lots of options here, but there are three things we really care about right now:

1. The class `MSMessage` is responsible for crafting messages: you create it with the URL built using `URLComponents`, then give it layout instructions.
2. Those layout instructions are provided using `MSMessageTemplateLayout`, which lets you describe what metadata is attached to your content.
3. There's a class called `MSSession`, which is responsible for tracking updates to individual messages over time.

When you create an `MSMessage` object, you get to choose whether it should be part of a session or not. If you want it to be interactive – i.e., to change over time as other participants

do things with it – then you need a session. If you just want messages to be “fire and forget”, then don’t use one; that’s pretty much it.

Enough explanation: let’s start looking at exactly what `createMessage()` needs to do:

1. It should return our extension to compact mode, because we’re finished manipulating dates and now want to send things.
2. It will do a quick sanity check to ensure we have an active conversation to work with – it’s optional, remember.
3. It will convert its `dates` and `votes` parameters into `URLQueryItem` objects, then wrap that in a `URLComponents` object. Each item is stored as either “date-[number]” or “vote-[number]” so that it’s identified uniquely, but the order of items is preserved in the array.
4. It will check to see if the currently selected message has an active session. If so, we’ll use it for our changes; if not, we’ll create a new one.
5. It will create a new `MSMessage` using the session from step 4, and give it the URL from the `URLComponents` object from step 3.
6. It will create a default `MSMessageTemplateLayout` layout and assign that to the message.
7. Finally, it will insert the new `MSMessage` into the current conversation.

That last step introduces two things worth discussing before we get into the code. First, you can never send things on behalf of the user. When we insert messages, all we’re doing is adding it to the Messages text entry field ready to send, but we can’t force it out. Second, because these messages are interactive – i.e., they change over time – Messages has a built-in mechanism to collapse message changes in a single conversation.

Here’s the new code for `createMessage()`, with numbered comments matching my list above:

```
func createMessage(with dates: [Date], votes: [Int]) {  
    // 1: return the extension to compact mode  
    requestPresentationStyle(.compact)  
  
    // 2: do a quick sanity check to make sure we have a conversation  
    // to work with  
    guard let conversation = activeConversation else { return }  
}
```

```

// 3: convert all our dates and votes into URLQueryItem objects
var components = URLComponents()
var items = [URLQueryItem]()

for (index, date) in dates.enumerated() {
    let dateItem = URLQueryItem(name: "date-\(index)", value:
String(from: date))
    items.append(dateItem)

    let voteItem = URLQueryItem(name: "vote-\(index)", value:
String(votes[index]))
    items.append(voteItem)
}

components.queryItems = items

// 4: use the existing session or create a new one
let session = conversation.selectedMessage?.session ??
MSSession()

// 5: create a new message from the session and assign it the URL
we created from our dates and votes
let message = MSMessages(session: session)
message.url = components.url

// 6: create a blank, default message layout
let layout = MSMessagesTemplateLayout()
message.layout = layout

// 7: insert it into the conversation
conversation.insert(message) { error in
}

```

```
        if let error = error {
            print(error)
        }
    }
}
```

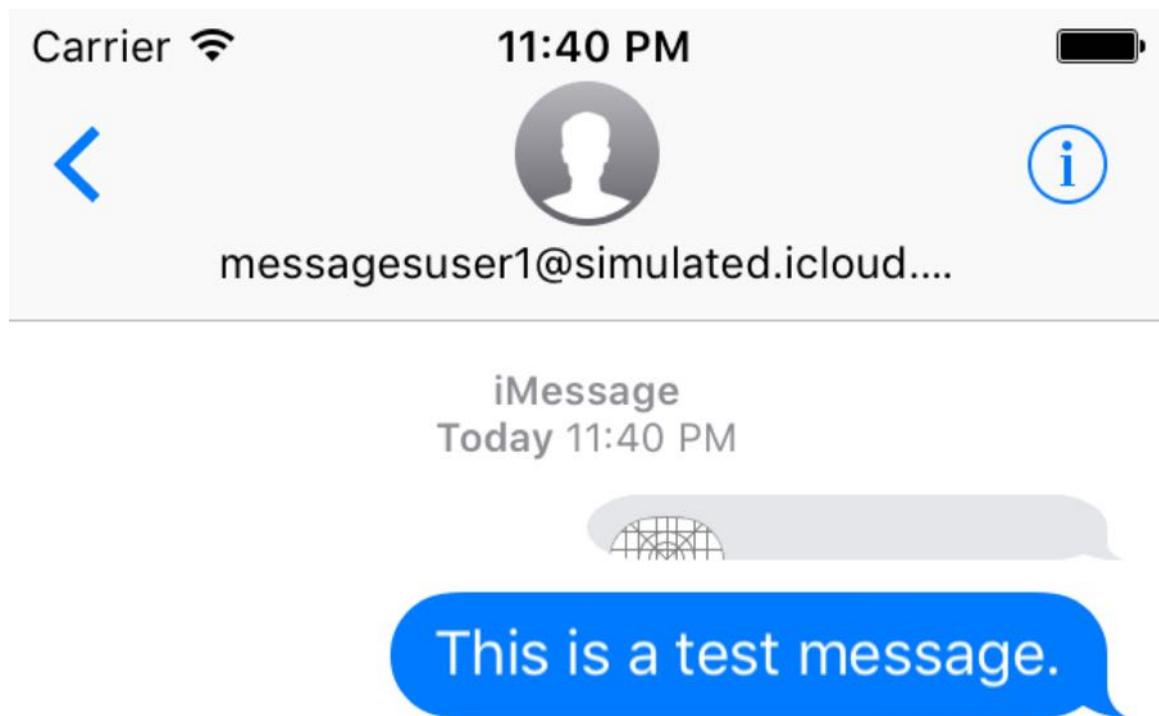
Now, that code won't work just yet because I slipped in a tiny helper method when converting the dates to **URLQueryItem** objects. Everyone reads dates differently – day-month-year, month/day/year, and so on – but when working with groups of people who will have different device settings we need to agree a uniform way of expressing dates so that we can send and receive the same thing.

That's where my helper method comes in: it accepts a date, and returns a string containing that date formatted in a specific way, using the UTC timezone, so that everyone sees the same thing.

Add this code to **MessagesViewController**:

```
func string(from date: Date) -> String {
    let dateFormatter = DateFormatter()
    dateFormatter.timeZone = TimeZone(name: "UTC")
    dateFormatter.dateFormat = "yyyy-MM-dd-HH-mm"
    return dateFormatter.string(from: date)
}
```

You should be able to run the code now and send messages. It won't look good, and you won't be able to read them meaningfully on the other side, but I hope you can see the light at the end of the tunnel.



At this point we're able to send messages with custom attachments, but the attachments are so small they are hard to tap.

(And no, the light at the end of a tunnel is *not* a rapidly approaching train about to flatten you.)

Receiving and decoding messages

Right now our message appears as a small, empty bubble, with our app icon in the corner. In theory you can click on it, but in practice it's almost impossible. So, before we're able to look at reading messages, let's make one tiny change to sending – add this line to the `createMessage()` method, just after the `let layout =` line:

```
layout.caption = "I voted"
```

With that one change, you'll now find it much easier to click on the message bubble that gets

sent. It looks even worse, though, because the caption underlaps the app icon, so you'll probably just see "ted" in the bubble. We'll fix that later: for now, let's get back to reading.



Adding a caption - even with its bad alignment - at least makes the bubble big enough to tap.

Send a message using our extension, then switch to the other person in the Messages simulator and click on the bubble to read the message. What you'll see is a full-screen version of our **MessagesViewController**: the user will see a large empty space, and a button saying "Create New Event". Clearly that's not what we want: the user tapped a message with some dates in, so they ought to be taken to the SelectDates view where they can cast their vote.

The reason we went to the wrong screen is because Messages follows a different flow when users trigger an extension by tapping a bubble. Previously we activated the event creation screen when our **willTransition()** method was called, but that doesn't happen when the user taps on a bubble: rather than launching the app in compact view then transition to expanded view, the app immediately gets activated in expanded view. As a result, **willTransition()** doesn't get called in this instance, and instead we need to use the **willBecomeActive(with conversation:)** method.

Now, cast your mind back a few chapters and you'll remember we created the `displayEventViewController()` method so that it accepts an `MSConversation` as its first parameter. It's now time for that to become important: previously we could just have used `activeConversation` to find which conversation we were in, but when you're in the `willBecomeActive()` method that property hasn't been set yet. Fortunately, `willBecomeActive()` gets passed the active conversation as a parameter, which goes back to why I made `displayEventViewController()` accept a parameter: when we're in `willBecomeActive()` we pass it the parameter that was handed to the method, and elsewhere we can use `activeConversation`.

So, when the user is creating an event, we start life in compact then transition to expanded using `willTransition()`; if they are viewing an event that was created elsewhere, we start life in expanded and `willBecomeActive()` is called. That's where we're going to show the `SelectDates` view, which is just a matter of calling the existing `displayEventViewController()` method and passing in "SelectDates" as the second parameter.

Here's the code for `willBecomeActive()`:

```
override func willBecomeActive(with conversation: MSConversation) {
    if presentationStyle == .expanded {
        displayEventViewController(conversation: conversation,
identifier: "SelectDates")
    }
}
```

With that change the app will now show the correct view when a bubble is tapped, but it still won't show anything useful in there because we haven't loaded the message URL data we carefully encoded earlier.

So, the next step is to create a new method `load()` method in `EventViewController` that accepts an `MSMessage` and pulls out date and vote values from there. We can then call this method inside `displayEventViewController()` so that it will load a received message if one exists, otherwise it will do nothing.

Now, there's a huge amount of optionality here: there might not be a message to work with; if there is a message, there might not be a URL attached to it; if there is a URL there might not be any URL components inside it; and if there are URL components there might not be any URL query items there. So, the first thing our new `load()` method will do is have four **guard** statements to ensure all values are present and correct – if any of them are missing, it does nothing.

Here's the first version, to be put inside **EventViewController**:

```
func load(from message: MSMessages?) {  
    guard let message = message else { return }  
    guard let messageURL = message.url else { return }  
    guard let urlComponents = NSURLComponents(url: messageURL,  
        resolvingAgainstBaseURL: false) else { return }  
    guard let queryItems = urlComponents.queryItems else { return }  
}
```

If all four of those **guard** statements run through OK, then it's safe to loop over the **queryItems** array to look for values that were sent. Because we prefixed each item with either "date-" or "vote-" it's just a matter of reversing the process from earlier to convert **URLQueryItem** objects into dates and integers.

There are two small complexities here. First, you'll remember that we converted our **Date** objects into strings using **NSDateFormatter**. Well, we need to reverse that here, so I created another helper method for **EventViewController** to do just that:

```
func date(from string: String) -> Date {  
    let dateFormatter = DateFormatter()  
    dateFormatter.timeZone = TimeZone(name: "UTC")  
    dateFormatter.dateFormat = "yyyy-MM-dd-HH-mm"  
    return dateFormatter.date(from: string) ?? Date()  
}
```

Note that `dateFormatter.date()` returns an optional `Date` because the parsing might fail. These are all dates we constructed so it should never fail, but rather than force unwrap the optional I just used the nil coalescing operator to provide a default value.

The second complexity is that the `value` property of `URLQueryItem` objects is optional, so we need to provide a default value. This becomes doubly complicated because we need to convert the vote count to an integer, so we need to convert the option `value` string into an integer, which itself is optional because creating an integer from a string gives you an `Int?`. As a result of the double optionality, we need to write code like this:

```
let voteCount = Int(item.value ?? "") ?? 0
```

That means “if the item has a value use it, otherwise use an empty string; then create an integer out of that value, but if that fails use 0.” It’s not pretty, but it would have been a darn sight worse without nil coalescing!

Here’s the full code for the `load()` method:

```
func load(from message: MSMessages?) {
    guard let message = message else { return }
    guard let messageURL = message.url else { return }
    guard let urlComponents = NSURLComponents(url: messageURL,
                                                resolvingAgainstBaseURL: false) else { return }
    guard let queryItems = urlComponents.queryItems else { return }

    for item in queryItems {
        if item.name.hasPrefix("date-") {
            dates.append(date(from: item.value ?? ""))
        } else if item.name.hasPrefix("vote-") {
            let voteCount = Int(item.value ?? "") ?? 0
            allVotes.append(voteCount)
        }
    }
}
```

```
    ourVotes.append(0)
}
}

}
```

That code won't compile at first because it uses **MSMessage** in **EventViewController**. To resolve that, add **import Messages** to the top of the file.

Note that we insert 0 into the **ourVotes** array for every vote that's found so that **allVotes** and **ourVotes** contain the same number of items.

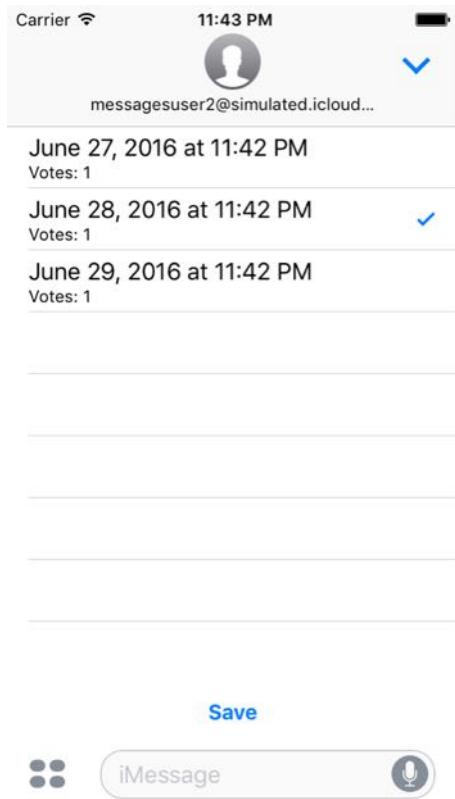
There's one last change to make, and it will clear up a warning in **MessagesViewController** that has been there for some time: we need to tell the **EventViewController** to load dates and votes from the current message. To do that, add this line of code to **MessagesViewController** in its **displayEventViewController()** method, just before the **vc.delegate = self** line:

```
vc.load(from: conversation.selectedMessage)
```

The **conversation.selectedMessage** property is optional because it could be a new message, but that's OK; the **load()** method we wrote will bail out if the message is nil.

That's the final piece of code to make our app fully functional: if you run it now, you should be able to create an event, select your preferences, send it to someone else, and see those votes in the simulator. That person can then cast their own votes and send it back.

Are we done? *Almost*. There's one further change I want to make, but it's entirely optional so I left it to last...



You can now read messages that have been sent, including votes that have been cast, then add your own and send it all back.

Making our message look attractive

Our iMessage App is fully functional at this point, but it has one flaw that will hold it back from being a widespread success: it looks terrifying ugly. Just writing “I voted” in the bubble would be bad enough, but the fact that half that text is obscured by our app icon makes it much worse.

Don’t worry, there’s a solution, but it requires a little effort. When you send any message, Apple lets you provide a picture preview of the contents of the message. You can either attach this directly as an image, or you can use the **mediaFileURL** property to send a link to a movie.

Sending images makes the whole message look a great deal better, but it has some important rules.

First, you should aim for an image that’s no more than 300x300 points in size and rendered at 3x resolution – i.e. 900x900 in total. This is because your message could be viewed on any device in the Apple ecosystem, so by sending it at the highest resolution the receiving device can downscale locally to ensure maximum fidelity.

Second, Apple recommends against adding text to the image, precisely because of the downscaling mentioned above. If you want to provide text to accompany the image you can use the **imageTitle** property to provide text that will be rendered over the image on the local device, thus ensuring it draws crisply no matter what.

So, those are the suggestions, and in theory they are great. But in practice... well, our whole app is about dates, so if we’re going to send an image, it’s pretty much going to have to be text because there’s nothing else useful we could send. Sure, you could make some sort of fancy calendar image showing the most popular choice so far, or perhaps just provide a generic image that represents your app in action, but I’m not sure either of those provide any value. Instead, we’re going to blithely ignore Apple’s suggestions and send an image of the dates that are on offer – it’s actually not too hard!

First, find this line in the **createMessage()** method:

```
let layout = MSMessageTemplateLayout()
```

Now add this line directly below it:

```
layout.image = render(dates: dates)
```

That calls a **render()** method we haven't written yet, but as you can probably guess it needs to accept an array of dates and return a **UIImage** that can go into the **layout.image** property.

Rendering text on iOS is remarkably easy: just create an attributed string with whatever formatting you want, then call its **draw()** method in an active context. Of course, there's a huge amount of work happening behind the scenes to make that happen, but from our point of view it's almost magic!

In order to make the result look a little nicer, we're going to create a string from all the dates in the input array, and calculate the size of it using the system's recommended font size. We'll then add 20 points on each side so it doesn't go edge to edge, and draw it to an image context.

I covered attributed strings in Hacking with Swift project 32, and drawing text to an image context was in Hacking with Swift project 27, however iOS 10 introduces a new way of drawing to images. The new **UIGraphicsImageRenderer** class is able to take advantage of new display technology that yields a wider color range, and Apple are recommending you use it rather than **UIGraphicsBeginImageContextWithOptions()** in new code.

Once you have an image context open, the old and the new methods are effectively identical. However, they are created quite differently. Previously you would have written something like this:

```
UIGraphicsBeginImageContextWithOptions(imageSize, true, 3)
UIColor.white().set()
UIRectFill(CGRect(origin: CGPoint.zero, size: imageSize))
let image = UIGraphicsGetImageFromCurrentImageContext()
UIGraphicsEndImageContext()
```

Using the new iOS 10 class, you pass your drawing instructions as a closure. This means you can set up your drawing instructions once, then re-use them in multiple ways if you choose. For example, **UIGraphicsImageRenderer** is capable of outputting PNG image data directly rather than a **UIImage**.

You also get to create and reuse the *settings* for your rendering, using a separate class called **UIGraphicsImageRendererFormat**. I've already said that we need to render our images at 3x resolution in order for them to look good on all devices, but by default images rendered with **UIGraphicsImageRenderer** are rendered at the device's native scale. To provide a custom scale – or to set background opacity and toggle wide color support – you need to provide some settings inside **UIGraphicsImageRendererFormat** object.

Before I show you the code, there's one added bonus to the new iOS 10 method of drawing: when you extract an image from **UIGraphicsImageRenderer** you will always get a **UIImage** back. It might be empty, but it will definitely exist. This is different from calling the old **UIGraphicsGetImageFromCurrentImageContext()** function, which returned an optional image – hurray for reducing optionality a little!

```
func render(dates: [Date]) -> UIImage {
    // define our 20-point padding
    let inset: CGFloat = 20

    // create the attributes for drawing using Dynamic Type so that
    // we respect the user's font choices
    let attributes = [NSFontAttributeName:
        UIFont.preferredFont(forTextStyle: UIFontTextStyleBody),
        NSForegroundColorAttributeName: UIColor.darkGray()]

    // make a single string out of all the dates
    var stringToRender = ""

    dates.forEach {
        stringToRender += DateFormatter.localizedString(from: $0,
            dateStyle: .long, timeStyle: .short) + "\n"
    }
}
```

```

    // trim the last line break, then create an attributed string by
merging the date string and the attributes

    let trimmed =
stringToRender.trimmingCharacters(in: .whitespacesAndNewlines)

    let attributedString = NSAttributedString(string: trimmed,
attributes: attributes)

    // calculate the size required to draw the attributed string,
then add the inset to all edges

    var imageSize = attributedString.size()
imageSize.width += inset * 2
imageSize.height += inset * 2

    // create an image format that uses @3x scale on an opaque
background

    let format = UIGraphicsImageRendererFormat()
format.opaque = true
format.scale = 3

    // create a renderer at the correct size, using the above format

    let renderer = UIGraphicsImageRenderer(size: imageSize, format:
format)

    // render a series of instructions to `image`

let image = renderer.image { ctx in
    // draw a solid white background
    UIColor.white().set()
    UIRectFill(CGRect(origin: CGPoint.zero, size: imageSize))

    // now render our text on top, using the insets we created
attributedString.draw(at: CGPoint(x: inset, y: inset))

```

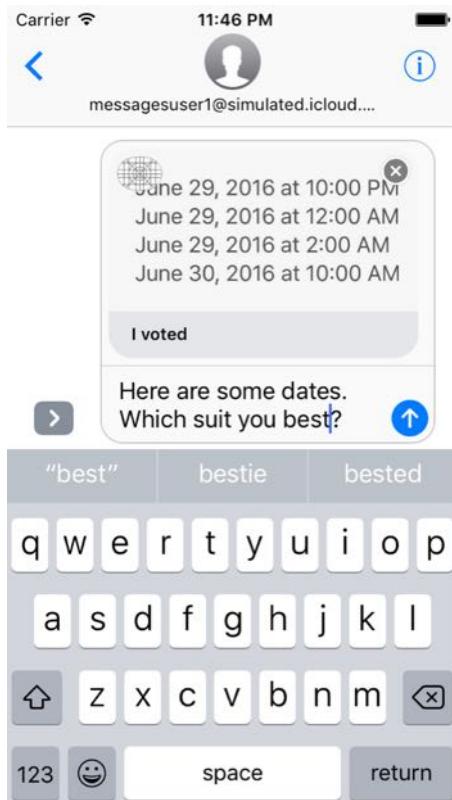
```

        }

    // send the resulting image back
    return image
}

```

If you run the app now, you'll see that not only do our votes get transferred in the message, but a picture of the available options also gets sent – it makes the whole thing look much more polished.



Having a render of the dates as the attachment image isn't ideal, but it does help make the end result look significantly better.

Wrap up

That's another iOS 10 project complete, and I hope you're starting to feel like you're mastering the changes Apple has introduced. Once again we're using some pretty hefty components in order to make the app work: stack views, view controller containment, **URLQueryItem**, **DateFormatter**, and **AttributedString** to name a few, not least the completely new Messages framework and all its classes.

But the end result works, solves a genuine problem, and doesn't look too bad, I think. It's not perfect, and in fact has one glaring problem: if it's used in group chats, it's possible that two people might hit send at exactly the same time, which would cause a clash: one set of votes would effectively overwrite the other.

Apple doesn't have a magic solution for this, and indeed their suggestion is just to run your own server where messages get sent. That way you can handle conflict resolution in code on your server, thus avoiding the problem – or at least pushing it somewhere else. That's a whole other project for a whole other book, and realistically in a whole other coding language, but before you go down that route it's probably worth at least considering whether CloudKit could solve the problem for you.

Homework

This is a complete app by itself, but it could do lots of other things depending on how far you want to take it. Here are some suggestions for adding polish and features:

1. How could you make it so that everyone can vote only once? Right now you can vote again and again if you wish.
2. If you click around, you can get the wrong UI to appear. Can you nail this down so that the CreateEvent screen is only shown for new events?
3. We've made the app use dates, but it could do so much more. Can you make it accept free-text strings, such as "where do you want to go for lunch?"

Project 3

[On Hold]

Ride Sharing

This project, a ride sharing app using SiriKit and Apple Maps, is currently on hold. I spent a huge amount of time working on it and have made some progress, but there are too many bugs in the current iOS 10 beta that are holding it back. I'm happy with *some* bugs, and file reports with Apple when I find them, but with too many the project just becomes unusable.

I'm hopeful that iOS 10 beta 2 will bring significant improvements, and will revisit this project at that point.

Project 4

Polyglot

Setting up

Today Extensions were introduced in iOS 8 as a way to present users with some information from your app along with other system notifications. The theory was sound: give apps a way to deliver their most important information at a system-wide level, and users will lap it up. But in practice, today extensions are used infrequently, and the majority of apps don't bother providing them.

That may all be about to change in iOS 10, because Apple has made two major changes to these extensions. First, they now appear on the lock screen by swiping to the left, where they are much more prominent in their design. Second, any app that creates a today extension will have that extension loaded as part of their 3D Touch quick actions. These quick actions are triggered when users with 3D Touch-enabled devices press hard on an icon, but because this feature was introduced *after* iOS 9 it has seen only partial take up. In iOS 10, Apple has stated that "every app should provide quick actions," so I think it's likely we'll see usage soar.

This combination of today extensions getting far prominence on the lock screen, extensions being displayed in 3D Touch quick actions, and 3D Touch as a whole being pushed hard by Apple, has the potential to create a perfect storm that makes today extensions hugely more popular in iOS 10. As a result, they are a perfect candidate for this book.

In this project you're going to build Polyglot: a lock screen widget aimed at language learners. It will show a list of words in English, which, when tapped, will reveal their equivalent in French. My original plan for this project was just that, but it turns out Apple made lock screen widgets so easy to do that I wanted to add something else: how to add those same widgets to 3D Touch quick actions that can be triggered from your app icon. But it turns out Apple made *that* incredibly easy too, so I decided to go one step further and teach the new animation system introduced in iOS 10.

So, this project doesn't do anything too complicated, but it *does* demonstrate a few interesting new things from iOS 10 all in one, so there's lots to be learned. As an extra bonus for you, I'll be demonstrating a couple of extra techniques along the way – sorry, but I'm a sucker for squeezing in extra learning when I see a chance!

Fire up Xcode 8, and create a new Single View Application for iPhone. Name it Polyglot, and save it on your desktop.

PS: If you were wondering, "Polyglot" comes from Greek words meaning "many tongue", and is used to mean someone who speaks several languages.

Creating a basic language app

We're going to start by doing the easiest part of this project, which is to create a simple app that lets you users browse through words and see their meanings in French.

Let's start with the easy stuff: creating a user interface for our app. Open ViewController.swift, and change the **ViewController** class so that it inherits from **UITableViewController** rather than directly from **UIViewController**.

Now open Main.storyboard, then delete the basic view controller we were given. In its place, drop a **UITableViewController** and embed it inside a navigation controller. Set its class name to **ViewController** so that it connects to the existing view controller we just tweaked. Add one prototype cell to the table if it doesn't have one already, then give it "Right Detail" for style and "Word" for identifier. Select its left-hand label and change its font to be System Bold.

Finally, select the navigation controller, and use the attributes inspector to select the "Is Initial View Controller" checkbox.

That's it! Yes, that's a much simpler interface than our previous projects – I think you all deserve a little break from Interface Builder, at least for the time being.

Loading example words

The first task we're going to complete is to load a list of words out of **User Defaults** and show it in the table view. Right now that will be a fixed list of words, but later on we'll let users customize the list.

First, add this property to the **ViewController** class:

```
var words = [String]()
```

We'll be using that to store all the user's vocabulary. To keep things as simple as possible, each entry will be listed in the format "English::French", e.g. "fox::le renard", and we'll just split that up as needed to display.

The **words** array will be saved to **UserDefaults** for ease of access, but when the app first runs there won't be anything saved so we're going to provide some initial words using a call to **saveInitialValues()**. Here's the code for **viewDidLoad()** and **saveInitialValues()**:

```
override func viewDidLoad() {
    super.viewDidLoad()

    let defaults = UserDefaults.standard()

    if let savedWords = defaults.object(forKey: "Words") as? [String]
    {
        words = savedWords
    } else {
        saveInitialValues(to: defaults)
    }
}

func saveInitialValues(to defaults: UserDefaults) {
    words.append("bear::l'ours")
    words.append("camel::le chameau")
    words.append("cow::la vache")
    words.append("fox::le renard")
    words.append("goat::la chèvre")
    words.append("monkey::le singe")
    words.append("pig::le cochon")
    words.append("rabbit::le lapin")
    words.append("sheep::le mouton")

    defaults.set(words, forKey: "Words")
}
```

There's nothing interesting in there: if we can load a saved words array then do so, otherwise create some example entries then save that.

To make things a little more interesting, let's make the title of our view controller look slightly customized by changing its font. This is done using the **titleTextAttributes** property of the navigation bar – add this to the **viewDidLoad()** method, just after the call to **super.viewDidLoad()**:

```
let titleAttributes = [NSFontAttributeName: UIFont(name: "AmericanTypewriter", size: 22)!]

navigationController?.navigationBar.titleTextAttributes =
titleAttributes

title = "POLYGLOT"
```

Although we now have data in the **words** property, it isn't being displayed just yet. To make it spring into life, we need to override the standard three methods for table views: **numberOfSections()**, **numberOfRowsInSection**, and **cellForRowAt**. The first two of those are trivial, and can be added now:

```
override func numberOfSections(in tableView: UITableView) -> Int {
    return 1
}

override func tableView(_ tableView: UITableView,
 numberOfRowsInSection section: Int) -> Int {
    return words.count
}
```

The third is slightly more complex, but only because we're storing English and French in each string and need to split them up. So, the **cellForRowAt** method will pull out the word at the specified index path, split it into an array by finding the string “::”, then put the first item into

the text label.

Here's the code:

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "Word",
for: indexPath)

    let word = words[indexPath.row]
    let split = word.components(separatedBy: "::")

    cell.textLabel?.text = split[0]

    return cell
}
```

Revealing answers

The app now displays a list of English words in the table, but it's quite useless for learning languages because you can't see the matching French word.

Fortunately, we can pretty much complete the app – at least an MVP – just by adding one more method call, plus a line of code elsewhere. The method in question is the table view's **didSelectRowAt** call, triggered when users tap rows. When that happens we want to reveal the French word, and we can do that just by setting the **detailTextLabel** property of the cell that was tapped. This means repeating some of the same code from **cellForRowAt**: find the word that was tapped and split it in two using “::”, but this time we'll choose the second item in the array because that's the French half. When the word is tapped again, we'll clear the detail text label.

Add this code to **ViewController** now:

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
    tableView.deselectRow(at: indexPath, animated: true)

    if let cell = tableView.cellForRow(at: indexPath) {
        if cell.detailTextLabel?.text == "" {
            let word = words[indexPath.row]
            let split = word.components(separatedBy: ":::")
            cell.detailTextLabel?.text = split[1]
        } else {
            cell.detailTextLabel?.text = ""
        }
    }
}
```

You'll notice I slipped a call to **deselectRow()** in there right at the beginning – this has the effect of deselecting rows when they are tapped, which I think looks quite pleasing.

There's only one small change to make in order for the app to be functional: inside **cellForRowAt** we need to clear the contents of the detail text label. If we don't do this, table cell recycling will mean you'll see incorrect words appearing in the table, which would make the app more of a hindrance than a help!

So, add this line to **cellForRowAt**, just before **return cell**:

```
cell.detailTextLabel?.text = ""
```

Adding and saving words

If you run the app now you'll see lots of English words that, when tapped, show their French equivalents. The app is usable, and even useful to a degree, but let's face it: unless the user is deeply interested in memorizing a handful of French animal names, they're going to want to add their own words, and delete the ones they don't want.

Deleting words is easy enough to do: we need to override the **commit** method of our table view controller, then remove the word that was selected from both the **words** array and the table view controller. Here's the code:

```
override func tableView(_ tableView: UITableView, commit editingStyle: UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
    words.remove(at: indexPath.row)
    tableView.deleteRows(at: [indexPath], with: .automatic)
    saveWords()
}
```

That won't build just yet because I've made it call a **saveWords()** method that we haven't written yet. Just putting the above method in place is enough to let users swipe to the left on a word to reveal a "Delete" button.

I made saving words into a separate **saveWords()** method because we'll be calling it from two places: adding new words and deleting existing ones. It doesn't do anything complicated – at least not yet – so this shouldn't cause you any trouble:

```
func saveWords() {
    let defaults = UserDefaults.standard()
    defaults.set(words, forKey: "Words")
}
```

Now for the more interesting part: adding new words. I've broken this down into three components: creating a bar button item that starts the process, writing an **addNewWord()** method that uses **UIAlertController** to prompt for English and French entry for a word, and

finally an `insertFlashcard()` method that does the actual work of joining the two languages together and updating our data source. Making those last two separate helps to keep the code smaller and easier to understand, as you'll see in a moment.

The first step is trivial: adding a bar button item that calls `addNewWord()`. Put this code into `viewDidLoad()`:

```
navigationItem.leftBarButtonItem =  
    UIBarButtonItem(barButtonSystemItem: .add, target: self, action:  
        #selector(addNewWord))
```

The `addNewWord()` method isn't too hard either, as long as you have some experience adding text fields to `UIAlertController` popups. If not, here's a brief primer:

1. You call `addTextField()` on your alert controller, and provide it a configuration closure that will be run using the new text field as its parameter. We'll be using that to set some placeholder text for the text fields.
2. When you create the `UIAlertAction` to handle the user tapping OK, you can read the `textFields` array of the alert controller to find out what each textfield is set to.
3. The `textFields` array is optional, as is the `text` property of a given text field. To ensure a value is provided, we'll use nil coalescing.

To make it easier to understand, I've added comments to my code so you can follow along if it's your first time. Add this new method to `ViewController` now:

```
func addNewWord() {  
    // create our alert controller  
    let ac = UIAlertController(title: "Add new word", message: nil,  
        preferredStyle: .alert)  
  
    // add two text fields, one for English and one for French  
    ac.addTextField { textField in  
        textField.placeholder = "English"  
    }  
}
```

```

ac.addTextField { (textField) in
    textField.placeholder = "French"
}

// create an "Add Word" button that submits the user's input
let submitAction = UIAlertAction(title: "Add Word",
style: .default) { [unowned self, ac] (action: UIAlertAction!) in
    // pull out the English and French words, or an empty string
    if there was a problem
        let firstWord = ac.textFields?[0].text ?? ""
        let secondWord = ac.textFields?[1].text ?? ""

        // submit the English and French word to the
        insertFlashcard() method
        self.insertFlashcard(first: firstWord, second: secondWord)
    }

// add the submit action, plus a cancel button
ac.addAction(submitAction)
ac.addAction(UIAlertAction(title: "Cancel", style: .cancel))

// present the alert controller to the user
present(ac, animated: true)
}

```

That just leaves the **insertFlashcard()** method. This is going to receive the English and French strings, then insert it into our **words** array and update the table view. However, if something went wrong with the alert controller or if the user didn't enter one of the languages, this method will get one or more empty strings submitted to it – in which case we don't want to do anything.

So, this method will:

1. Ensure it has valid strings for both English and French.
2. Create a new **IndexPath** object based on the number of words in the **words** property.
3. Add the new word to the **words** property by combining the English and French parts.
4. Call **insertRows()** on the table view to update the UI.
5. Call **saveWords()** to write the changed **words** property to disk.

Note that 1 and 2 are in that order for a reason: by constructing the index path based on the number of words *before* the new entry was added, it accounts for the fact that **words.count** is always one higher than the index path will be

Here's the code:

```
func insertFlashcard(first: String, second: String) {  
    guard first.characters.count > 0 && second.characters.count > 0  
    else { return }  
  
    let newIndexPath = IndexPath(row: words.count, section: 0)  
  
    words.append("\\" + (first) + ":" + (second) + ")")  
    tableView.insertRows(at: [newIndexPath], with: .automatic)  
  
    saveWords()  
}
```

If you run the app now you'll see it's fully functional, if a bit boring. It's time to do something more interesting: today extensions!

Building a today extension

The app we have right now is a useful, if basic, language learning tool. To make it more useful, we're going to add a today extension that mimics the same functionality right on the user's lock screen, so they can revise words without having to launch your app. Much of the code to make this happen is identical to what we've already done, but there are a few interesting additions along the way!

To get started, go to File > New > Target in Xcode, then choose iOS > Application Extensions > Today Extension from the window that appears. For "Product Name" enter Widget, but leave the rest of the settings alone – they determine how the extension is bundled inside your main app.

When you click Finish to create the extension, you'll see it appears as a group called "Widget" inside the project navigator pane, complete with its own view controller and storyboard.

Following the theme of this book so far, we're going to start with the user interface, so open MainInterface.storyboard inside the extension group. You'll see a single, small view controller with a dark gray background, and a label saying "Hello World" in the center.

Go ahead and delete the "Hello World" label, then click in the gray area to select the view. The height that you see isn't the size that will be used for the extension, so we're going to make it bigger now so it's easier to work with. To do that, go to the size inspector and change the height value up to 200 – more than enough room for activities!

Now drag a table view out onto the view, and make it fill the space. For me, that meant having X and Y set to 0, width set to 320, and height set to 200. Now go to Editor > Resolve Auto Layout Issues, and choose the "Add Missing Constraints" item under the "Selected View" header, causing the table view to resize to fill its parent view when its size changes.

Add one prototype cell to the table, then give it the style "Right Detail" and identifier "Word", just like we did before. However, this time I want to make the widget inherit a little of the design of the parent app, so select the left-hand label in the cell and change its font to be American Typewriter at size 17. Now select the right-hand label and change its font to something else – be creative! I thought System Italic 17 looked pretty good. I suggest you set the text color of the right-hand label to be black rather than the default gray, otherwise it's hard to read on the lock screen.

To finish up, let's make some outlets. Connect the table view's data source and delegate outlets to the Today View Controller in the document outline. Finally, switch to the assistant editor then create an outlet for the table view called **tableView**.

Getting the extension up and running

We're going to tackle the easiest part of the extension first, which is simply replicating what we have in the existing app, albeit with a couple of tweaks. You might think that's quite a boring place to start, but trust me: it has a speed bump smack in the middle of it that you need to be aware of!

First, add the same **words** property to the **TodayViewController** class:

```
var words = [String]()
```

Now add this code to **viewDidLoad()** to load the user's saved words from user defaults:

```
let defaults = UserDefaults.standard()

if let savedWords = defaults.object(forKey: "Words") as? [String] {
    words = savedWords
}
```

Next, we made our view controller the delegate and data source of a table view, so we need to conform to those two protocols now. Modify the definition of the **TodayViewController** class to this:

```
class TodayViewController: UIViewController, NCWidgetProviding,
UITableViewControllerDataSource, UITableViewDelegate {
```

Now we need to add the four of the table view methods from the previous view controller into

the today view controller. Specifically, copy the `numberOfSections()`, `numberOfRowsInSection`, `cellForRowAt`, and `didSelectRowAt` methods, and paste them into the `TodayViewController` class. You'll need to remove the "override" keyword from each of them, because `TodayViewController` is a direct subclass of `UIViewController` rather than going via `UITableViewController`.

So far, so boring. To run the extension, look up to the right of the Play and Stop buttons for where it says "Polyglot", and click that. From the menu that appears, choose Widget > iPhone SE, then click Play. You'll be asked to choose an app to host the extension, so please choose Today and click Run.

When the simulator launches, you'll be taken to the new widget area, and you should see the Polyglot widget there waiting for you. If not – it's a bit glitchy! – try pressing Cmd+R in Xcode again and repeating the process.

The speed bump with extensions

As you'll notice – when you finally get the simulator to play ball – the extension is completely empty: there are no words in there. I made you copy and paste code directly from the original app to avoid problems, and yet it hasn't worked. What gives?

The problem our code is failing is a fairly fundamental one: the app we built is, in the eyes of the app, executed quite separately from the extension. This means the two aren't able to exchange data by reading and writing the standard user defaults – they both have their own user defaults settings, rather than sharing one pool. If we changed our code around so that we wrote to files, we've had the same problem: both the app and the extension would write to separate file sandboxes.

The fix for this problem is to build our app so that the main app and the extension are both able to read and write from shared data storage. Apple makes this possible using a technology called App Groups. This registers a shared grouping with Apple, then writes that grouping directly into your built product, with the end result that multiple apps and extensions can share the same data sandbox on devices.

To get started, select your project in the project navigator – that's the thing at the top of the left-hand pane, with a blue icon next to it. Under Targets make sure "Polyglot" is selected, then go to the Capabilities tab and flick the App Groups option to On. Click the small +

button in there and enter group.com.hackingwithswift.polyglot as the name, changing the “com.hackingwithswift” part to whatever you used for your app’s bundle identifier.

This process creates a new app group with that name, and adds the Polyglot app to that group. You now need to select “Widget” under the Targets list, then enable App Groups for that too, selecting the same “group.com.hackingwithswift.polyglot” group.

What we’ve done is tell Apple we want to have the ability for our extension and app to share data. We’re not actually sharing anything yet, but we do now have the *ability* to do it.

In order to save and load the same data, we need to change the code used to access the **UserDefaults** class. Right now we’re using code like this:

```
let defaults = UserDefaults.standard()
if let savedWords = defaults.object(forKey: "Words") as? [String] {
    words = savedWords
}
```

That uses the **standard()** user defaults, which is *not* shared as part of an app group. Every time we access **UserDefaults.standard()** we need to replace it with a new **UserDefaults** constructor: **UserDefaults(suiteName:)**. That lets us specify the identifier of an app group, so the line means “load the shared user defaults” rather than “load my own sandboxed user defaults.”

To make this work, we need to make one change inside `TodayViewController.swift`, and two more inside `ViewController.swift`. Each time the change is the same: replace the **standard()** user defaults with ones from the app group we just created, each time wrapped in if/let because the **suiteName** initializer is failable.

First, open `TodayViewController.swift` and change the **UserDefaults** code to this:

```
if let defaults = UserDefaults(suiteName:
"group.com.hackingwithswift.polyglot") {
    if let savedWords = defaults.object(forKey: "Words") as? [String]
{
```

```
    words = savedWords  
}  
}
```

Now open ViewController.swift and find the **saveWords()** method. Replace it with this instead:

```
func saveWords() {  
    if let defaults = UserDefaults(suiteName:  
        "group.com.hackingwithswift.polyglot") {  
        defaults.set(words, forKey: "Words")  
    }  
}
```

The third and final change is inside **viewDidLoad()** for ViewController.swift, which needs to be this:

```
if let defaults = UserDefaults(suiteName:  
    "group.com.hackingwithswift.polyglot") {  
    if let savedWords = defaults.object(forKey: "Words") as? [String]  
    {  
        words = savedWords  
    } else {  
        saveInitialValues(to: defaults)  
    }  
}
```

At this point, we need to do one more thing, which is to reset the simulator by going to Simulator > Reset Contents and Settings. This ought not to be needed, but in my experience the simulator tends to get very confused when app capabilities are changed, and this is the

only way to be sure.

When the reset completes, launch the regular (non-extension!) app from Xcode so that our default values get installed, then run the Widget again. All being well – fingers, toes, legs, and tentacles crossed – you should see the extension spring to life with the user’s list of words. Good job!

Note: It can take the debugger some time to attach to today extensions. If you find nothing appears in the simulator, you might need to either restart the simulator then re-run the app to register new defaults, or just wait a few seconds more. If you’re able to run on a real device, you’ll find it has fewer bugs.

Adding some polish

Our extension is pretty basic right now, but with a few changes we can make it work better. First, if you’re running on a real device you’ll notice that the default gray selection style of our table view cells looks pretty poor in today extensions. By default the cells have a translucent white color that lets a little of the background image through, and the gray selected color looks pretty ugly in comparison.

You can change the selection style by adding a custom value to the cell’s **selectedBackgroundView** property, and the easy way to do that is just by adding two lines of code to the **cellForRowAt** method, just before **return cell**, like this:

```
cell.selectedBackgroundView = UIView()  
cell.selectedBackgroundView!.backgroundColor = UIColor(white: 1,  
alpha: 0.20)
```

Even though that creates new views for every cell, the code above doesn’t waste any resources because no table view cell recycling happens – today extensions scroll up and down as one, rather than having individual table views scrolling around. This new selected background view is far more subtle, and also lets a little of the background image through, so it makes the whole extension a little more attractive.

The second change we’re going to make is to increase the height of our extension’s table

view cells just a little. Right now you'll see it contains two full rows, and about half of a third. Right now, the height of widgets is locked at 110 points, which is what's causing this problem. Sadly this can't be changed – although I filed a bug report with Apple asking this to be rectified! – so the easiest thing to do is increase the row height from 44 to 55 so that two cells take up exactly 110 points. To do that, go to MainInterface.storyboard, select the table view, then go to the size inspector and change Row Height to 55.

The third thing we're going to do is to use a new feature of iOS 10 that lets users toggle between expanded and compact widgets. By default, all widgets are shown in compact mode, meaning that they have a height of 110 points. But if you tell iOS you're able to handle expanded mode, it will automatically add a "Show More" button to your widget so that it can expand. To do that, set the **widgetLargestAvailableDisplayMode** property of your extension context to be **.expanded** in **viewDidLoad()**, like this:

```
extensionContext?.widgetLargestAvailableDisplayMode = .expanded
```

You can run the widget now if you want – the "Show More" button should be there (again, give the debugger a few seconds to attach!), but it won't do anything. To make it work, we need to implement another method that's new in iOS 10,

widgetActiveDisplayModeDidChange(). This tells us when the user has pressed the "Show More" button and wants to see more information from our widget, and it's where we can adjust the size of our view controller to show more detail. Once pressed, this button automatically becomes "Show Less", which triggers the same method when tapped.

To change the size of your view controller when it's running as a today extension, you just need to set your **preferredContentSize** property – iOS will handle the rest, including animation. You don't need to worry about the width value for this, because that will automatically be set to the correct width depending on the screen. So, add this new method to **TodayViewController** now:

```
func widgetActiveDisplayModeDidChange(_ activeDisplayMode:  
NCWidgetDisplayMode, withMaximumSize maxSize: CGSize) {  
  
    if activeDisplayMode == .compact {  
  
        preferredContentSize = CGSize(width: 0, height: 110)  
    } else {
```

```
preferredContentSize = CGSize(width: 0, height: 440)  
}  
}
```

That will show either two cells or eight depending on whether the widget is in compact or expanded mode.

3D Touch quick actions

One of the really welcome new features in iOS 10 is the ability to add today extensions directly to the quick action menu for 3D Touch devices.

This is indeed a really welcome feature, but it's also surprisingly complicated to implement – you need to edit a few plist files, import a new framework, then write a couple of hundred lines of code.

Actually, just kidding: it already works. Yes, just the action of creating a today extension automatically registers it as a 3D Touch quick action for your app, so if you have a 3D Touch-capable device around just press hard on the Polyglot icon on the home screen and you'll see our today extension appear. Plus, it's fully interactive: you can tap on any of the English words to see the French translation, with no further work required.

One small niggle: continuing the running theme of the last few minutes, I've filed a bug report with Apple regarding today extensions in quick actions. If you bring up the menu and tap a word, then close the menu and bring it up again, you'll see the French translation appears for a split second before going away again. Obviously this rather undermines the point of our application, and also looks pretty ugly – again, I'm hoping Apple will be able to address the bug soon.

Animating views with UIViewControllerAnimated

So far you've how to take advantage of the new lock screen widgets in iOS 10, and you may also have tried app groups and navigation bar **titleTextAttributes** for the first time, but we're not done just yet. Instead, I want to demonstrate the new iOS 10 animation framework that lets us create interactive, reversible animations without too much hassle.

To finish off this project, we're going to add a test mode to the app so that users will see a French word and need to think of what the matching English word might be. When they want to guess, they'll tap a button and the correct answer will be shown in green. I had briefly considered **SFSpeechRecognizer** for this task – which would let you speak into your microphone to record answers – but I think that's best left as homework, because it's just repeating the same work we did in the Happy Days project.

In Xcode, go to File > New > File, then choose Cocoa Touch Class and click Next. Name it TestViewController, then make it a subclass and click Next again. In the final step, make sure the Group box is set to Polyglot with a yellow folder next to it, then click Create.

Open Main.storyboard so we can create a simple user interface for this test screen. Drag a new view controller onto the storyboard, then add a vertical stack view to it. Set its spacing to be 40, then add two labels to it. Give the first label the text "How do you say this in English?" and set its text alignment to center. Give the second label the text "LE CHAMEAU" as an example, set its alignment to center, then give it the font American Typewriter at 40 point.

Next, add constraints from the stack view so that it's centered horizontally and vertically in its parent, then use Editor > Resolve Auto Layout Issues > Update Frames to make the UI jump into position.

The last thing to do is create some outlets. Start by selecting the view controller, then going to the identity inspector and changing its class to be TestViewController. While you're there, set its storyboard identifier to be "Test". Now activate the assistant editor, and create outlets for the stack view and the "LE CHAMEAU" label named **stackView** and **prompt** respectively.

We're done with IB now, but before we dive into TestViewController.swift we need to add some code to the original view controller so that the user can activate test mode.

First, add this method to the **ViewController** class:

```
func startTest() {  
    guard let vc =  
        storyboard?.instantiateViewController(withIdentifier: "Test") as?  
        TestViewController else { return }  
  
    vc.words = words  
  
    navigationController?.pushViewController(vc, animated: true)  
}
```

That creates a new instance of our **TestViewController** class, and sets its **words** property to be our user's words. That won't build just yet because we haven't created a **words** property in **TestViewController**, but that will come soon enough.

Now, add these two lines of code to **viewDidLoad()**:

```
navigationItem.rightBarButtonItem =  
    UIBarButtonItem(barButtonSystemItem: .play, target: self, action:  
    #selector(startTest))  
  
navigationItem.backBarButtonItem = UIBarButtonItem(title: "End Test",  
    style: .plain, target: nil, action: nil)
```

The first line will trigger the **startTest()** method we just added, and the second is a tiny UI fix. The **startTest()** method pushes the test view controller onto the existing stack of the navigation controller, which means by default the back button will look like “< POLYGLOT”. That's a bit ugly, but by setting the **backBarButtonItem** property we can provide a custom string saying “End Test” instead. Much nicer!

Creating a simple test

Now that we have a basic user interface, the next step is to present questions and let users

see the answer. We're going to create a **words** array that will store all the words in the test, but at the same time we're going to create two more properties:

- **questionCounter** will track which question is currently being asked. We'll shuffle the array once when the view loads, then load whichever item is pointed to by **questionCounter**.
- **showingQuestion** will be either true or false, depending on whether we're currently showing the question (true) or the answer (false).

Add these properties to **TestViewController** now:

```
var words: [String]!
var questionCounter = 0
var showingQuestion = true
```

In the **viewDidLoad()** method we're going to shuffle the array, add a button for the user to tap to advance the test, then give our view controller a title that can appear in the navigation bar. Add these three lines of code to **viewDidLoad()** now:

```
navigationItem.rightBarButtonItem =
UIBarButtonItem(barButtonSystemItem: .fastForward, target: self,
action: #selector(nextTapped))

words = GKRandomSource.sharedRandom().arrayByShufflingObjects(in:
words) as! [String]

title = "TEST"
```

You'll need to import GameplayKit in order for thee shuffling line to work, so add this near the top of the file:

```
import GameplayKit
```

Note: iOS 10 introduced a shorter **shuffled()** method to do array shuffling, but a) it only works on **NSArray**, and b) still doesn't use generics, and so still requires the **as! [String]**

typecast.

To make the test screen work in a basic form, we're going to add four new methods:

1. **viewDidAppear()** will be used to ask the initial question, which means it will wait until the navigation controller's push animation has finished fully before starting.
2. **nextTapped()** is called by the bar button item we added a moment ago, and will either show the answer or prepare to show the next question depending on the value of **showingQuestion**. Showing the answer will set the prompt text color to green.
3. **askQuestion()** will advance **questionCounter** one place then set up the question prompt with the current French word.
4. **prepareForNextQuestion()** will reset the UI so that the prompt text is black, then call **askQuestion()**.

So, to recap: when the view is first shown we will call **askQuestion()** straight away. After that, tapping the fast forward bar button item will do one of two things: if we're showing the question, it will show the answer and set its color to be green; otherwise it will call **prepareForNextQuestion()** to set the color back to black then call **askQuestion()**. This might seem like a strange approach, but it will make sense in just a few minutes!

First, add these four methods to **TestViewController**:

```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)

    // the view was just shown - start asking a question now
    askQuestion()

}

func nextTapped() {
    // move between showing question and answer
    showingQuestion = !showingQuestion

    if showingQuestion {
```

```

        // we should be showing the question - reset!
        prepareForNextQuestion()

    } else {

        // we should be showing the answer - show it now, and set the
        color to be green

        prompt.text = words[questionCounter].components(separatedBy:
"::") [0]

        prompt.textColor = UIColor(red: 0, green: 0.7, blue: 0,
alpha: 1)

    }

}

func askQuestion() {

    // move the question counter one place
    questionCounter += 1

    if questionCounter == words.count {

        // wrap it back to 0 if we've gone beyond the size of the
        array

        questionCounter = 0

    }

    // pull out the French word at the current question position
    prompt.text = words[questionCounter].components(separatedBy:
"::") [1]

}

func prepareForNextQuestion() {

    // reset the prompt back to black
    prompt.textColor = UIColor.black()

    // proceed with the next question
    askQuestion()
}

```

```
}
```

Adding animations

If you run the app now you'll see that the test mode works, although it doesn't look great – not only do you briefly see "LE CHAMEAU" when the view first appears, but also the lack of animation makes it look quite dull.

We're going to fix both of these problems now using animation. When the view first loads, we're going to hide the stack view by setting its alpha to 0 and scaling it down to 80% of its regular size. In `addQuestion()` we're going to create a spring animation to set the alpha 1 and remove the scale transformation. Finally, in `prepareForNextQuestion()` we're going to make the stack view disappear again, before calling `askQuestion()` again. Best of all, we're going to do this with the new `UIViewControllerAnimated` class, which lets us have much more control over animation – you can pause animations, reverse them, scrub through them, change their timing part way through, and much more. We don't need those advanced features here, so I'll save them for a separate technique project!

To start with, we want to hide the stack view when the test view first loads. That's done by adding these two lines of code to the end of `viewDidLoad()`:

```
stackView.transform = CGAffineTransform(scaleX: 0.8, y: 0.8)  
stackView.alpha = 0
```

The next step is to make the stack view bounce each question in, which is done using a spring animation. This used to be done using the method `UIView.animate()` and providing parameters for `withDuration`, `delay`, `usingSpringWithDamping`, `initialSpringVelocity`, `options`, `animations`, and `completion` – in other words it was far harder than it needed to be. Using the new `UIViewControllerAnimated` class you can create a spring animation by providing a duration and damping ratio - i.e., how "springy" the spring should be – and have iOS do the rest for you.

To use `UIViewControllerAnimated`, you create the object with a duration and any timing

information (springiness, in our case), then give it animations to run. When you’re ready, you call **startAnimation()** on your property animator object, and the animation will run. This opens up a whole new way of doing animations: you can create them up front, then run them whenever and wherever you need later on.

For now, let’s just stick with creating and running a spring animation to make the question appear. Add this code to the end of the **askQuestion()** method:

```
let animation = UIViewPropertyAnimator(duration: 0.5, dampingRatio: 0.5) {
    self.stackView.alpha = 1
    self.stackView.transform = CGAffineTransform.identity
}

animation.startAnimation()
```

The final step is to modify the **prepareForNextQuestion()** method so that it animates the answer away ready to be replaced with a new question. We’re going to do this using a regular “ease in, ease out” animation, but I also want to show you how to add a completion block.

Right now the **prepareForNextQuestion()** method looks like this:

```
prompt.textColor = UIColor.black()
askQuestion()
```

What we *want* to happen is for the stack view to animate away, and for those two lines to be executed only when the animation is finished. To make that happen, you first create the animation you want, like this:

```
let animation = UIViewPropertyAnimator(duration: 0.5,
curve: .easeInOut) { [unowned self] in
```

```
    self.stackView.transform = CGAffineTransform(scaleX: 0.8, y: 0.8)
    self.stackView.alpha = 0
}
```

You then call **animation.addCompletion()** to pass it a closure that should be run when the animation completes, which in our case will be the existing two lines we have that reset the text color and call **askQuestion()** again. As it's a closure, we need to use **[unowned self]**, **self.prompt.textColor**, and **self.askQuestion()**, like this:

```
animation.addCompletion { [unowned self] position in
    self.prompt.textColor = UIColor.black()
    self.askQuestion()
}
```

Finally, we start the animation like this:

```
animation.startAnimation()
```

That's it. Putting all that together, the final **prepareForNextQuestion()** should be this:

```
func prepareForNextQuestion() {
    let animation = UIViewPropertyAnimator(duration: 0.5,
    curve: .easeInOut) { [unowned self] in
        self.stackView.transform = CGAffineTransform(scaleX: 0.8, y:
    0.8)
        self.stackView.alpha = 0
    }

    animation.addCompletion { [unowned self] position in
```

```
    self.prompt.textColor = UIColor.black()
    self.askQuestion()

}

animation.startAnimation()
```

That completes the project – good job!

Wrap up

This has been a project of several small parts, none of which were complicated individually, but together add up to a simple, smart, and useful app. iOS 10's increased push towards app widgets and 3D Touch is going to open a lot of interesting opportunities for developers to offer utilities straight to end users, and now hopefully you have all the tools required to build something new and useful.

Also in this project we covered **UIViewControllerAnimated** class introduced in iOS 10, which lets us control our animations in ways that were simply impossible earlier. I'll be going into more detail on this in a separate technique chapter, because it's worth getting into fine-grained detail in ways that wouldn't make sense in a real project!

Homework

If you want to take this project further, there are several tasks that are worth taking on:

- Fun: Allow users to edit words they have added already, perhaps by using something like **tableView(_:editActionsForRowAt:)** to provide a custom action next to the delete button when you slide.
- Tricky: In test mode, have the user hold down a button to trigger recording their voice to a file. When they release, use **SFSpeechRecognizer** to transcribe what they said. Tip: you can create your recognizer using a **Locale** object like **fr-FR** to have it transcribe French speech even when the user's system setting is English!
- Taxing: We added support for French in this tutorial, but there's no reason we couldn't support others. Initially you could just change "French" to "Spanish" wherever it appears in the project, but for a true Polyglot approach try making the app support French *and* Spanish for each English word.
- Mayhem: Let users mark test answers right and wrong, then update the widget to show their worst words. This would require the app saving a separate array of "PracticeWords" for the widget to load.

If all those seem a little too hard for you, here's something even easier: we set the widget's table view to be 440 points high when it's expanded, regardless of how many words are in the array. How about you try to make it use the minimum of either 440 or **words.count * 55** so that it shows no more than eight cells, but will show fewer if there aren't enough words to fill the widget?

Project 5

Alarmadillo

Setting up

Some people have busy lives. Teachers, for example, might have different classes in different places on different days, and if they're working as a supply teacher it can be pretty chaotic finding their way to and from every place they need to be.

In this project, we're going to build Alarmadillo: an app that helps organize complicated schedules by creating alarm groups that can be enabled or disabled easily. The user can create as many alarm groups as they want, and add as many alarms to each group as they need, giving them control over their schedule. So, whether they are a supply teaching remembering classes, or a patient who struggles to remember which pills they need to take on each day, Alarmadillo will help.

Of course, this project is really just an excuse for me to teach you more about iOS 10. In particular, we're going to be used the all-new notification framework that combines push notifications with local notifications, adds support for displaying media, and even lets users enter text as a response to the notification. I've designed Alarmadillo specifically to exercise this new functionality, but it's also a fully capable app all by itself. In fact, while writing it I had the opposite problem to what I had while coding Polyglot – Alarmadillo grew so big that I had to start taking things out and simplifying the code in order to make it a manageable project!

Even after all my simplification, it's a pretty massive project. So, I've designed it in such a way that all the iOS 10 work happens at the end. This means you have a choice, depending on how you like to learn:

1. If you want to make the complete project from scratch, just carry on reading below.
2. If you want to learn only the new iOS 10 parts, you should download my existing project, Project5-HalfWay, and turn to the chapter "Adding local notifications". The "half way" project contains the complete project up to that chapter, so you should be able to pick up from there.

If you're still reading, it means you want to create the complete project from scratch. Hardcore mode – I like it!

I'm a big believer in learning while doing, and I'm glad you agree – let's get started. Launch Xcode and select Single View Application. Name it Alarmadillo and select iPhone as your target device, then save it on your desktop.

Building the user interface

This project is going to have three main view controllers, plus an extra navigation controller to wrap it all up. The first view controller is easy enough because it will just show a list of the available alarm groups; the second will show settings for a given alarm group, as well as list the alarms it contains; the third will let users edit an alarm by giving it a title, description, and date, and also attach a picture.

Open Main.storyboard now, and you'll see the same empty view controller we've had in previous projects. Please delete it and replace it with a table view controller, then embed that inside a navigation controller, and mark the navigation controller as the Initial View Controller for the storyboard. Go to the attributes inspector and change its the table view's Style value from Plain to Grouped, because that's the style we'll be using elsewhere in the app.

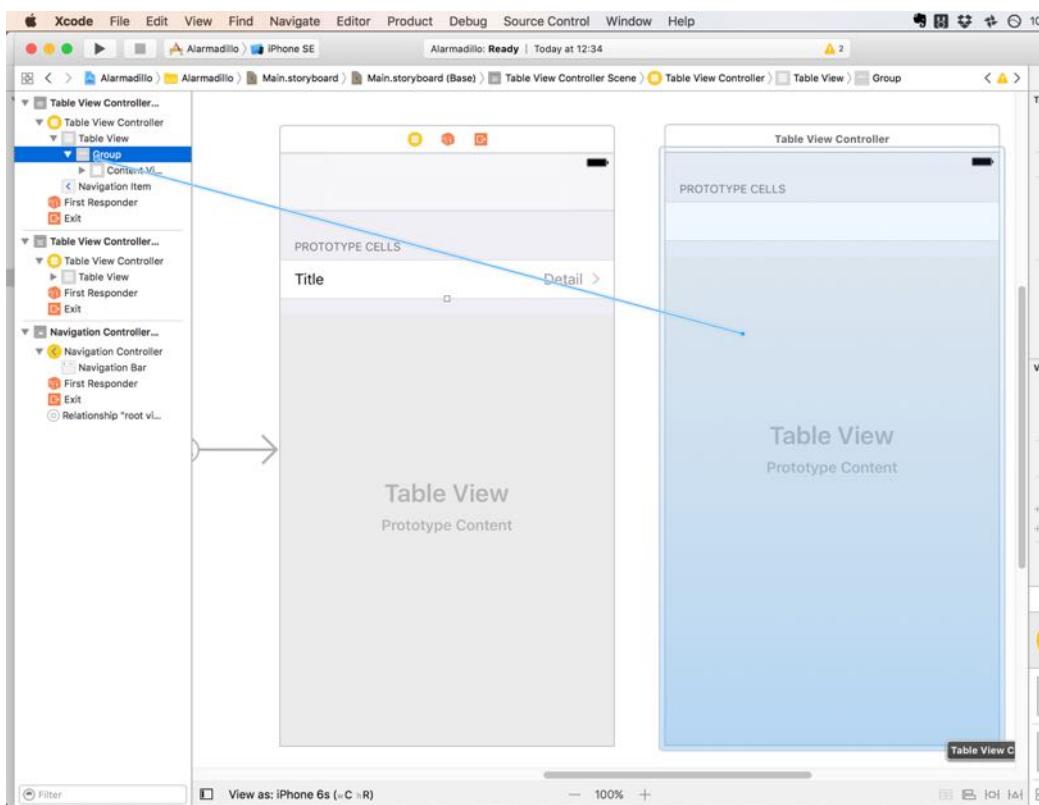
The table view should have one prototype cell already, so select that now and give it the identifier "Group". For Style please select Right Detail, and for Accessory please choose Disclosure Indicator so that there's a small arrow on the right of each cell.

That's the first view controller completed – I told you it was easy!

The second view controller is more complicated, because it needs to show some settings alongside a list of alarms in each group. We're going to let users name each alarm group, decide whether it should play a sound when triggered, and – most importantly – whether it is currently enabled. This lets them create lots of alarms inside lots of groups, then enable or disable them easily.

Drag another table view controller next to the one you added a few minutes ago, and again set the Style property of its table view to be Grouped.

We're going to connect the first controller to the second by using a segue. To make that happen, make sure the "Group" cell is selected in the document outline, then Ctrl-drag from there to the table view controller you just added. Choose "Show" from the list of segue options that appear, and you'll see a navigation bar appear at the top of the new table view controller because that's how it will appear. When the segue has been created, please select it and give it the identifier EditGroup.

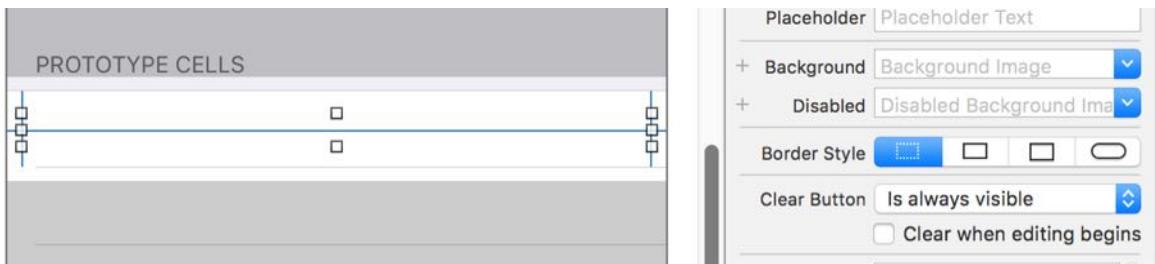


Ctrl-drag from a table view cell to another view controller to create a segue.

The next step is to create three prototype cells in the new table view. There is one by default, so please add two more. Select the first one and give it the identifier “EditText”, give the second the identifier “Switch”, and give the third the identifier “RightDetail”. We’re going to design each of these individually, because they are all different.

The first cell will let users name an alarm group, so please drop a text field in there. Ctrl-drag from that text field up to its parent view – the table view cell’s content view – and give it the constraints Leading Edge To Container Margin, Trailing Edge To Container Margin, and Center Vertically In Container. Use the size inspector to set the constants for each of those constraints to be 0, then use Resolve Auto Layout Issues > Update Frames to make the text field jump into place.

We want that text field to blend in with the rest of the table, and we can do that with only a few small changes. First, select the text field and choose the first option under Border Style – the one with the dotted lines around a square. Directly below that is the Clear Button dropdown; please change that to Is Always Visible.



Select the dotted line option for Border Style to make the textfield be borderless.

Change the text field's font size to be System 17.0 so that it matches other labels, then give it the placeholder text “Name this group”. Finally, change Capitalization to “Sentences”, and Return Key to “Done”.

In the original draft of this project I created **UITableViewCell** subclasses for each our cells, but as part of an attempt to shrink the project down I did away with them – we’re going to use view tags to identify things instead. Given that all our cells are trivial things this won’t cause a problem but you’re welcome to go the whole hog in your own code. So, for this cell I’d like you to give the text field the tag 1 so that we can identify it uniquely. Now select the cell itself and set its Selection option to “None” so that tapping the cell doesn’t turn it gray. That’s the first cell done!

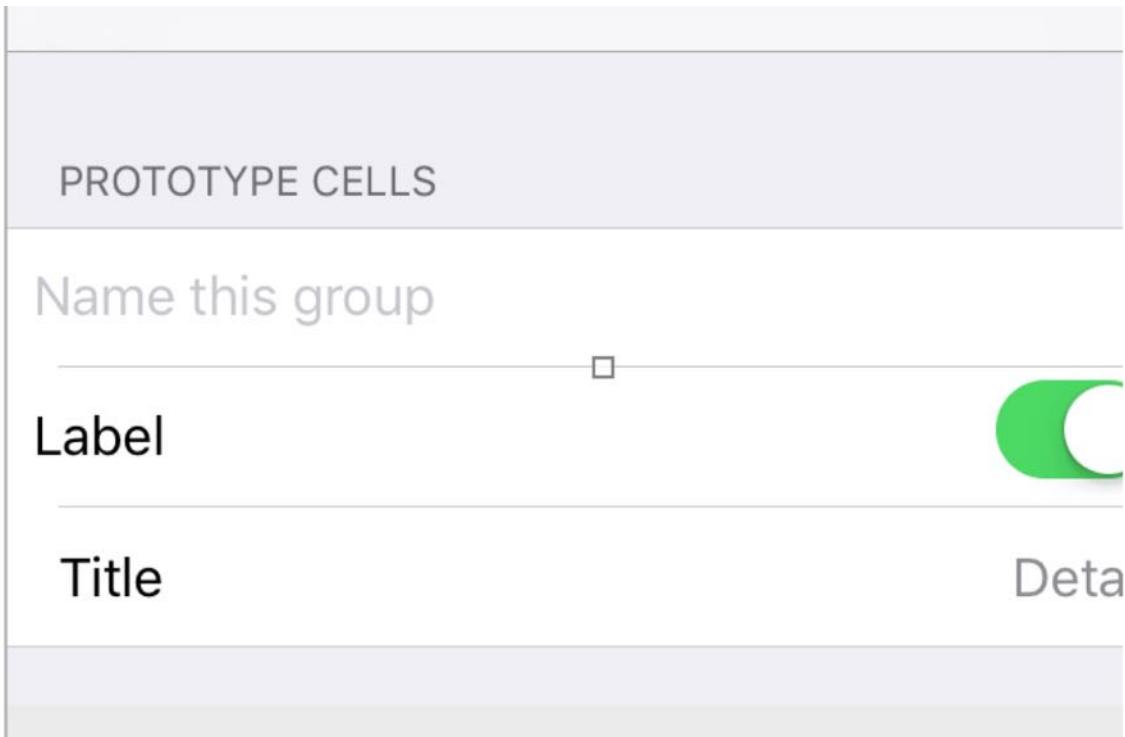
The second cell prototype – “Switch” – is going to have a label on the left and a **UISwitch** on the right, and will be used to let users enable or disable the alarm group and enable or disable its sounds. Drag into it a single **UILabel** along with a **UISwitch**, then give the label the tag 1 and the switch the tag 2. Please also give this cell a Selection value of “None” so that tapping it does nothing.

We only need a few constraints to make this cell layout correctly. First, Ctrl-drag from the switch out to the right, onto its parent view, and choose Trailing Space To Container Margin and Center Vertically In Container. For the label, Ctrl-drag to the left and onto its parent view then choose Leading Space To Container Margin and Center Vertically in Container. Again, set the constants for all those constraints to be 0, then use Update Frames to move the switch and the label into place. That’s the second cell done!

The final cell is much easier: just set its style to Right Detail then give it the accessory Disclosure Indicator.

Now, if you’re observant you’ll notice that the left edge of the top two labels is different from the left edge of the third. That used to be something you could fix in Xcode, but in the current

beta it seems to have disappeared so we'll need to fix that in code.



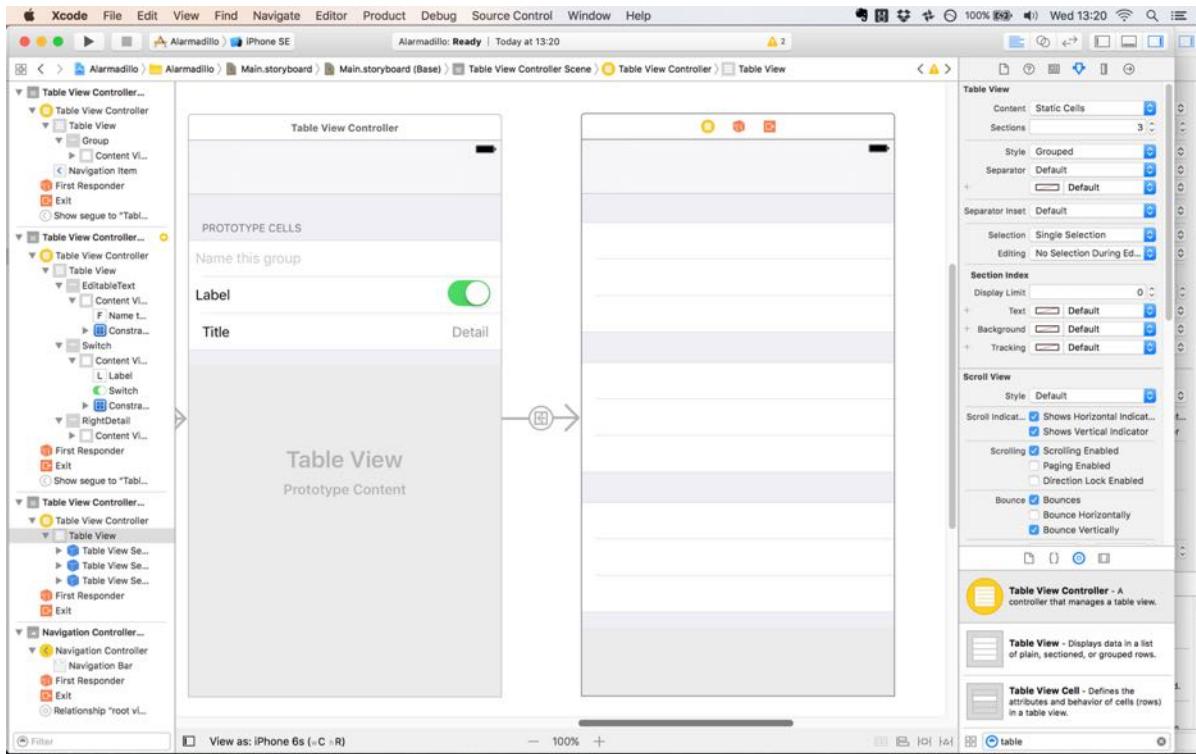
The default iOS table view cells have a specific left indent that won't be matched by our custom cells. That's OK – for now.

Editing alarms

Now it's time for the complicated view controller: editing alarms. This needs to let users add a name and description for each alarm, select a time for it to go off, then optionally also attach a picture. I tried out a few options here, but I found the one that worked best in terms of looking consistent and ensuring functionality across screen sizes was to use another table view controller. That being said, this one is a little different because we're going to use static cells rather than prototypes – we're going to design each cell individually.

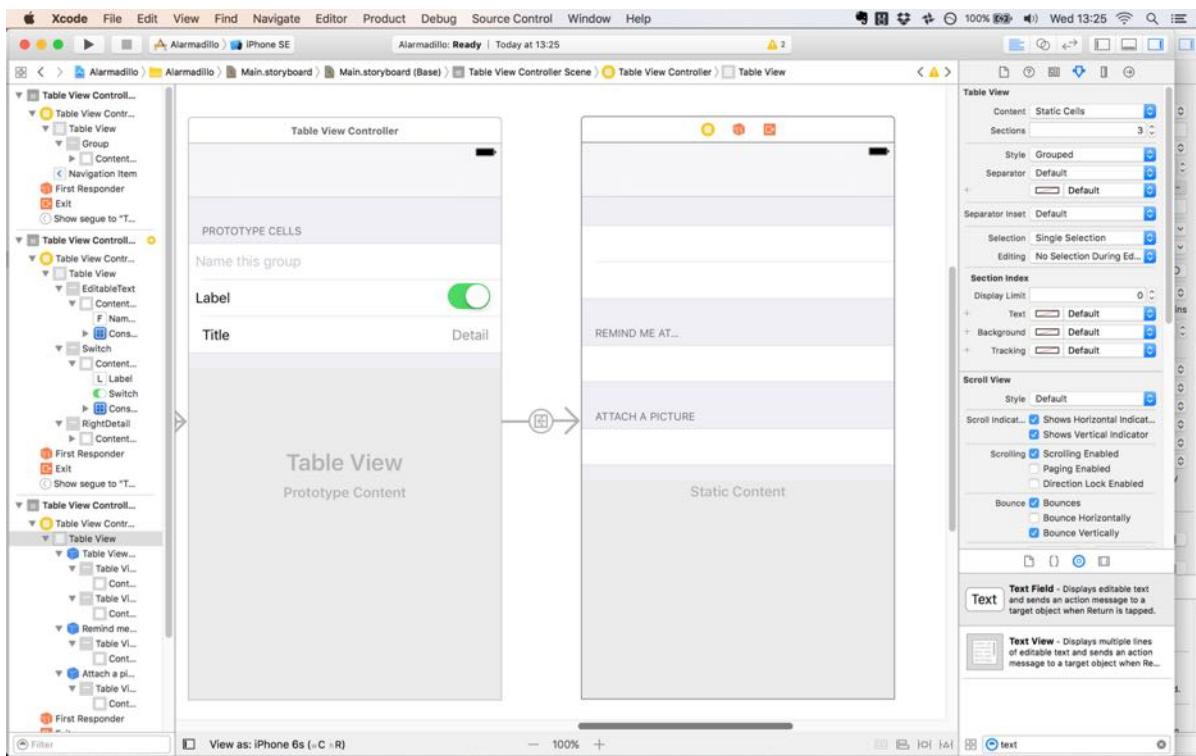
To begin, drag out a new table view controller onto the storyboard, then set the style property of its table view to Grouped to match the others. To connect that to our existing view controller flow, you need to Ctrl-drag from the "RightDetail" cell in the document outline and onto the new table view controller, then choose the "Show" segue. Again, I'd like you to select the segue you just created and give it a name, this time "EditAlarm".

Now for the interesting part: select the table view then change Content from Dynamic Prototypes to Static Cells. Use the box directly below “Content” to give it three sections, which will cause you to have three sections of three cells each.



When you add three sections to the table view, you'll see three sets of three cells by default.

Use the document outline to select the first table view section, and give it two rows. Select the second section and give it one row, but also set its Header to be “Remind me at...”. Now select the third section then give it one row, and the header “Attach a picture”.



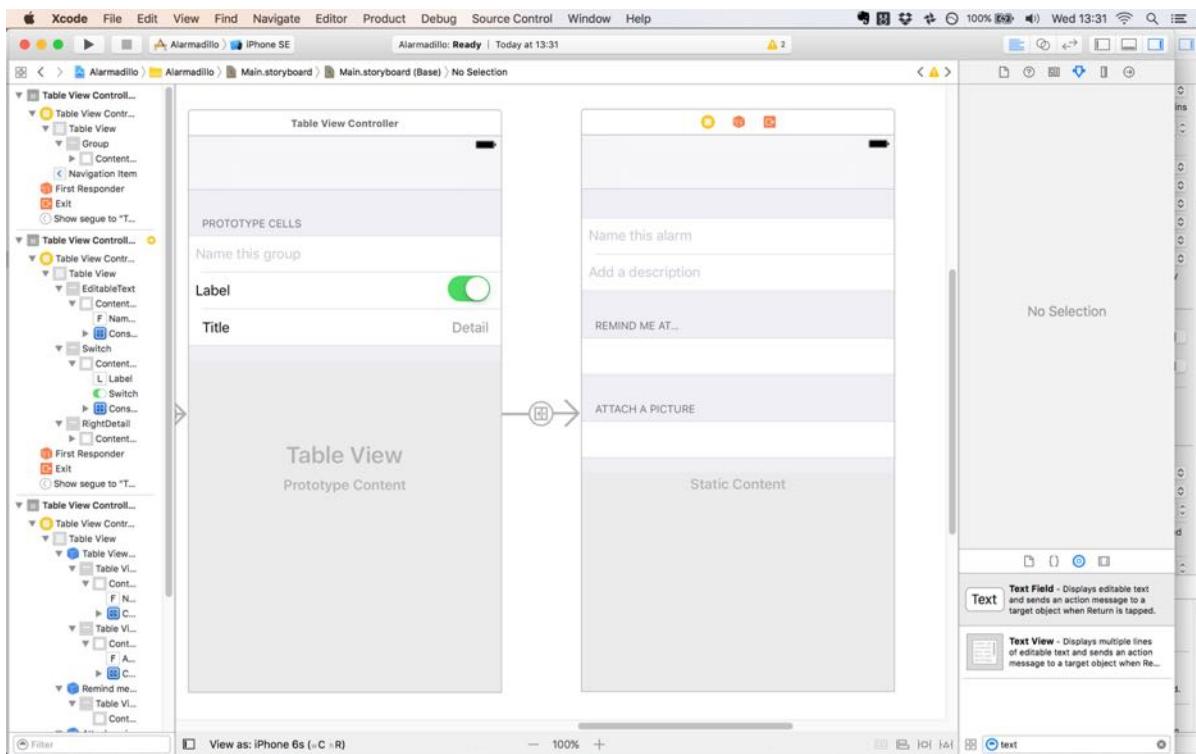
Using grouped tables with header text helps the different parts of your user interface stand out better.

The first two rows are identical in their content, and follow the same approach used with the “EditableText” cell from earlier. So, drag a **UITextField** into each of them, then make the same customizations as before:

- Give it the font size 17.
- Remove the border.
- Set Clear Button to “Is Always Visible”.
- Set Capitalization to “Sentences”.
- Change Return Key to “Done”.

There's no need to set a tag this time, because we'll be creating outlets directly – these are static cells, after all. However, I'd like you to give the top text field the placeholder “Name this alarm”, and the bottom text field the placeholder “Add a description”

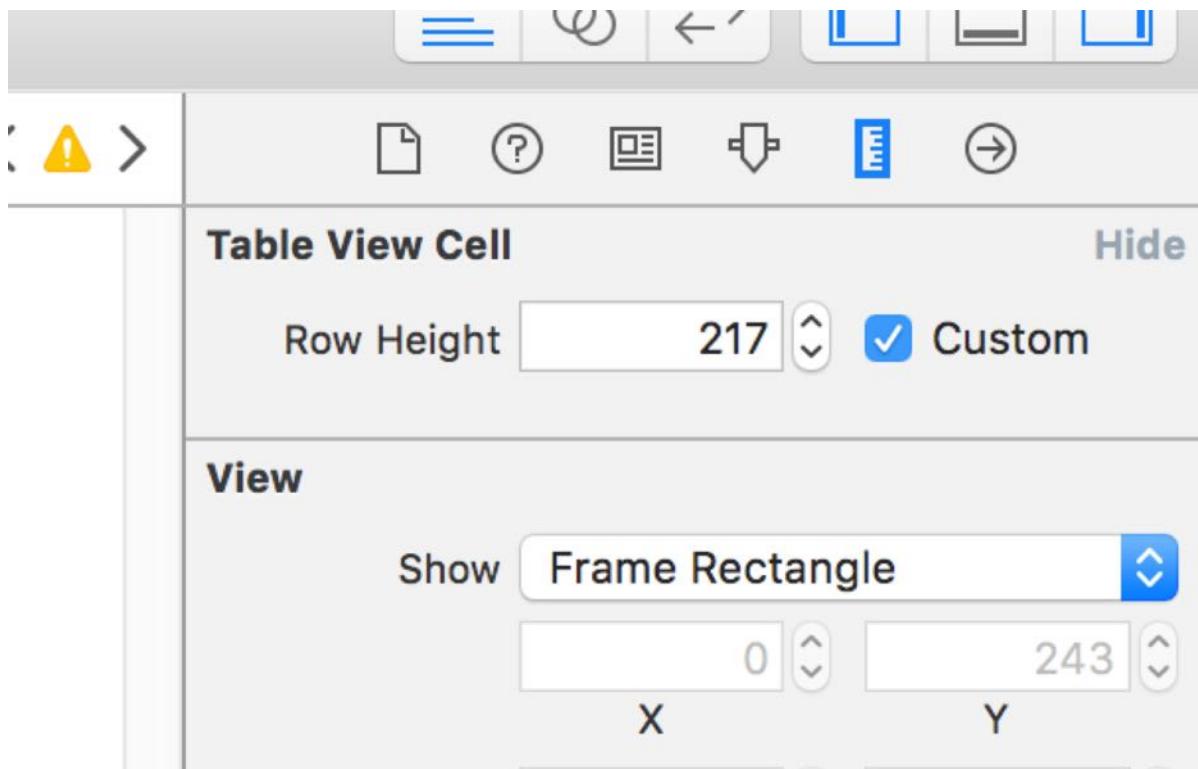
Add Leading Space To Container, Trailing Space To Container, and Center Vertically In Container constraints to both text fields, set the constants to 0, then use Update Frames to lay them out correctly.



We need two textfields at the top to let users input their data, but making them borderless helps them blend in with the table view.

The second section has just one cell, and is going to contain a date picker. As you saw in project 2, these don't play nicely with Auto Layout just yet, so we need to wrap the date picker inside a **UIView** so that it lays out correctly.

Start by selecting the empty cell that is already there in section two. Now go to the size inspector and change its size value from "Default" to 217 – that one point larger than the default date picker height because we need to take into account the separator line.

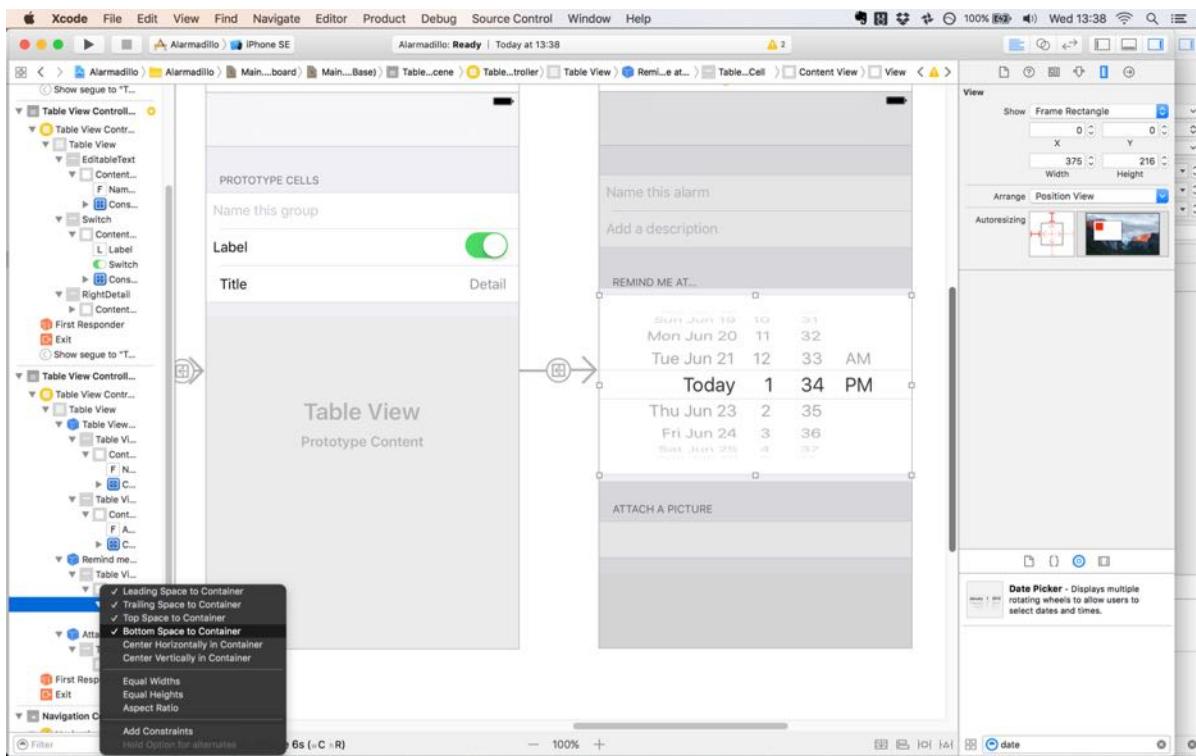


Date pickers are 216 points high by default, but we need to add one point more to allow for the table view separator.

That creates a large white space in the cell, so please drop a date picker in there now. When you've done that, select the date picker and go to Editor > Embed In > View so that we can fix its Auto Layout problems. The view that contains it will likely have a strange frame by default, so select it using the document outline then go to the size inspector and set its frame to be X: 0, Y: 0, Width: 375, Height: 216. Note: I used width 375 because that's the same width as the table view; if you're using a different simulated size, use that instead.

Resizing the view will have resized the date picker too, so you need to fix that before continuing. Select the date picker, then give it the frame X: 0, Y: 0, Width: 375, and Height: 216.

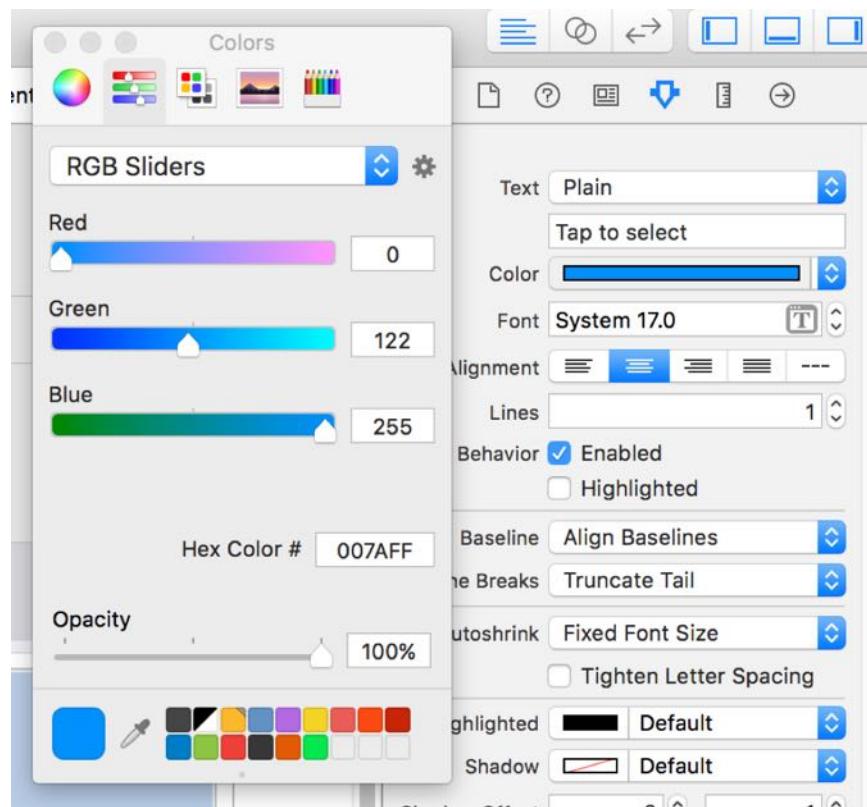
Finally, select the view that wraps the date picker, then give it the following constraints: Leading Edge To Container, Trailing Edge To Container, Top Space To Container, and Bottom Space To Container. That will ensure the view always resizes to fit the cell, and the date picker will automatically resize with it.



Wrap the date picker in a **UIView**, then add constraints to the view so that it always fills its container.

That's the second section complete, so all we need to do now is finish the third and final section. This is where we want users to attach a picture to their reminder, which might be a photo of the medicine bottle they need to take, a map of where their next class is, or whatever suits them best.

Select the lone cell in the third section and give it the height 200. Now drag a **UILabel** in there, as well as **UIImageView** – in that order, so the label appears behind the image. Select the label using the document outline, then give it the text “Tap to select image” and the constraints Center Horizontally In Container and Center Vertically In Container. Change the label’s text color from black to iOS blue - that’s Red 0, Green 122, Blue 255.

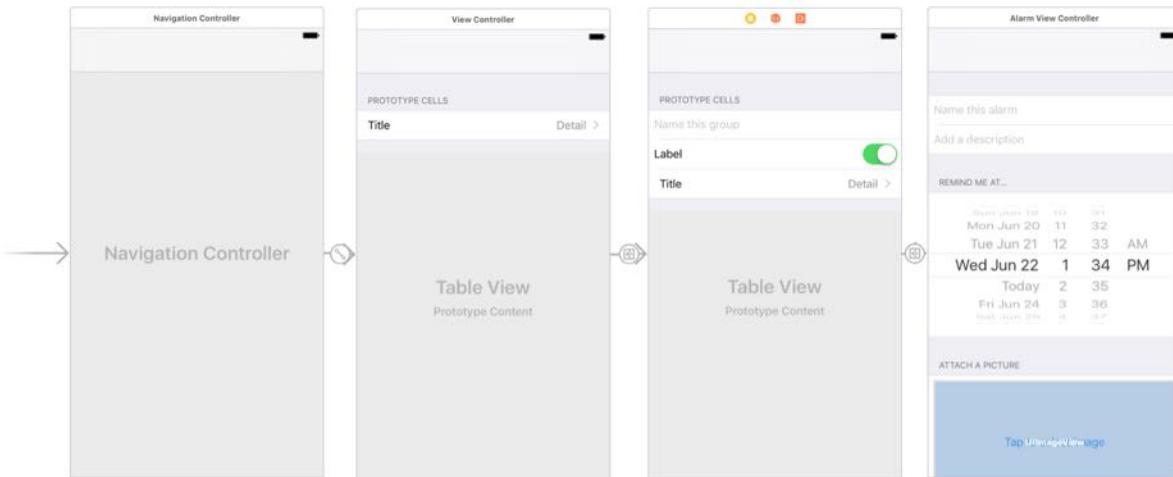


Set your label to have the iOS blue color. We're going to add a label behind the image view rather than use a button, because we need it to remain tappable even when obscured by the image.

Now select the image view and give it the constraints Leading Space To Container Margin, Trailing Space To Container Margin, Top Space To Container Margin, and Bottom Space To Container Margin. As per usual, select those constraints in the size inspector and set their constant values to 0. You will probably need to use Update Frames to make the label jump into place.

Drag a tap gesture recognizer onto the image view, then make sure you select the image view and check the box marked User Interaction Enabled. While you're there, I suggest changing the image view's mode from "Scale To Fill" to "Aspect Fit".

Before we're done, there's one small last change: make sure the Selection style for the first three rows is set to "None" so they don't turn gray when tapped.



The finished storyboard in full.

Actions and outlets

We just created a pretty big user interface up front, but I hope it lets you see how the app fits together. We already have one class for the main view controller, but we need to create two more: one for groups and one for alarms.

Let's start with the initial view controller, because that's the easiest. Our Xcode template already gave us `ViewController.swift`; all you need to do is change the `ViewController` class so that it inherits from `UITableViewController` instead of `UIViewController`.

Now we need to create two new view controllers, so go to `File > New > File`, then choose `iOS > Source > Cocoa Touch Class`. Choose `UIViewController` for `Subclass Of`, then name the new class “`GroupViewController`”. Make sure it's created in the `Alarmadillo` group with the yellow icon next to it, then when it's created change it to subclass from `UITableViewController` instead of `UIViewController`. Repeat all that again, this time creating a class called “`AlarmViewController`”. Trust me: this approach generates substantially cleaner code!

At this point you should have three view controllers, all of which inherit from `UITableViewController`. The next step is to connect them to the correct view controllers in the storyboard, so switch back to `Main.storyboard` and do the following:

1. Give the original table view controller – the one with the single prototype cell – the class

“ViewController”.

2. Give the middle table view controller – the one with three prototype cells – the class “GroupViewController”
3. Give the final table view controller – the one with the static cells and the date picker – the class “AlarmViewController”.

Now all that leaves is to create some outlets and actions. There aren’t too many of these, fortunately.

Let’s start with the group view controller first. Set the view controller to be the delegate of the “Name this group” text field. That is, Ctrl-drag from the text field to the Group View Controller line in the document outline. Now use the assistant editor to create an action for the **UISwitch** called **switchChanged()**, then make sure you change the Type value from “AnyObject” to “UISwitch”.

As for the alarms controller, set both of its text fields to use Alarm View Controller for their delegate, then create the following outlets:

- Name the first textfield **name**.
- Name the second textfield **caption**.
- Name the date picker **datePicker**.
- Name the image view **imageView**.
- Name the “Tap to select image” label **tapToSelectImage**.

Finally, I need you to create two new actions. First, Ctrl-drag from the date picker to make a new action called **datePickerChanged()**. Then Ctrl-drag from the tap gesture recognizer in the document outline to create another action called **imageViewTapped()**.

That’s it. Phew! I *told* you this project was going to be big, and we’re only just getting started...

Creating model classes

We now have classes to represent all three of our view controllers, but I'd like to create two more to represent the objects we'll be using in our app. They are classes rather than structs, because later on we'll be adding support for **NSCoding** to read and write these objects to disk.

So, go to File > New > File, then choose iOS > Source > Cocoa Touch Class, and create a new class called "Group" that subclasses **NSObject**. Repeat that again to create a class called "Alarm".

An alarm will be made up of five properties:

- **id**: a string containing a unique identifier for this alarm.
- **name**: the text the user entered for this alarm's name.
- **caption**: the text the user entered for this alarm's description. (**NSObject** already has a property called **description**, hence why we're using **caption**.)
- **time**: a **Date** object that stores the time the alarm should be triggered.
- **image**: the name of the image that is attached to this picture.

We'll also need an initializer to give all those properties default values, so replace the existing class in Alarm.swift with this:

```
class Alarm: NSObject {

    var id: String
    var name: String
    var caption: String
    var time: Date
    var image: String

    init(name: String, caption: String, time: Date, image: String) {
        self.id = UUID().uuidString
        self.name = name
        self.caption = caption
        self.time = time
    }
}
```

```
    self.image = image
}
}
```

Note that I create the **id** property inside the initializer rather than pass it in – it's just a random sequence of letters and numbers that identify the alarm uniquely.

That's the **Alarm** class finished for now, so we can move on to work with the **Group** class. This also needs five properties:

- **id**: a string containing a unique identifier for this group.
- **name**: the text the user entered for this group's name.
- **playSound**: a boolean that stores whether alarms in this group should trigger a sound.
- **enabled**: a boolean that stores whether this group of alarms is enabled.
- **alarms**: an array of **Alarm** objects that belong to this group.

Again, we'll need an initializer to set default values, so replace the existing **Group** class with this:

```
class Group: NSObject {

    var id: String
    var name: String
    var playSound: Bool
    var enabled: Bool
    var alarms: [Alarm]

    init(name: String, playSound: Bool, enabled: Bool, alarms: [Alarm]) {
        self.id = UUID().uuidString
        self.name = name
        self.playSound = playSound
        self.enabled = enabled
    }
}
```

```
    self.alarms = alarms
}
}
```

You might find it useful to place both of those classes in a group called “Model” – to do that, select both files then right-click and choose New Group From Selection.

With those two classes in place, we can create properties in our three view controllers to store them. Add this one to the **ViewController** class:

```
var groups = [Group]()
```

That will store the master list of all alarm groups in our app. When we’re in **GroupViewController** we need a property to store the group we’re currently editing, so add this property to GroupViewController.swift:

```
var group: Group!
```

Finally, in **AlarmViewController** we need to know which alarm we’re editing, so add this property:

```
var alarm: Alarm!
```

Listing groups

The **ViewController** class is, for now at least, the easiest one in the project because it just needs to display the groups and their names in its table view.

Let's start with something easy: I want to set **titleTextAttributes** for the navigation bar, just like we did in Polyglot. "Alarmadillo" is a playful name for an app, so we're going to use a playful font. There aren't many playful fonts built into iOS, but "Arial Rounded" comes pretty close. So, replace the existing **viewDidLoad()** method in **ViewController** with this:

```
override func viewDidLoad() {
    super.viewDidLoad()

    let titleAttributes = [NSFontAttributeName: UIFont(name: "Arial
Rounded MT Bold", size: 20)!]

    navigationController?.navigationBar.titleTextAttributes =
    titleAttributes

    title = "Alarmadillo"
}
```

Now add a couple of trivial methods: **numberOfSections** should always return 1 because we only have one section, and **numberOfRowsInSection** should return the value of **groups.count** so that we have one row for each group the user created.

Add these methods now:

```
override func numberOfSections(in tableView: UITableView) -> Int {
    return 1
}

override func tableView(_ tableView: UITableView,
numberOfRowsInSection section: Int) -> Int {
    return groups.count
}
```

```
}
```

I want users to be able to delete groups they no longer want, so we need to add a **commit** method for the table view that will delete the offending group and remove it from the table view. Add this method:

```
override func tableView(_ tableView: UITableView, commit editingStyle: UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
    groups.remove(at: indexPath.row)
    tableView.deleteRows(at: [indexPath], with: .automatic)
}
```

The final method we need to add is **cellForRowAt**, which is fractionally more complicated. This needs to pull out the correct group from the **groups** array, and assign its name to the cell's text label. If the alarm is enabled we're going to make its text color black; if it's disabled we'll red. Finally, we're going to set the **detailTextLabel** property to the number of alarms inside the group, so we're going to add a little bit of logic so that it handles singular and plural correctly.

Here's the code:

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "Group", for: indexPath)

    let group = groups[indexPath.row]
    cell.textLabel?.text = group.name

    if group.enabled {
        cell.textLabel?.textColor = UIColor.black()
    } else {
        cell.textLabel?.textColor = UIColor.red()
    }

    cell.detailTextLabel?.text = String(format: "%d alarm%@", group.alarms.count == 1 ? "" : "s", group.alarms.count)
}
```

```

    } else {
        cell.textLabel?.textColor = UIColor.red()
    }

    if group.alarms.count == 1 {
        cell.detailTextLabel?.text = "1 alarm"
    } else {
        cell.detailTextLabel?.text = "\(group.alarms.count) alarms"
    }

    return cell
}

```

If you run the app now you won't see anything, but only because the **groups** array is empty right now. To make the UI spring into life, try adding two dummy groups in the **viewDidLoad()** method, like this:

```

groups.append(Group(name: "Enabled group", playSound: true, enabled: true, alarms: []))
groups.append(Group(name: "Disabled group", playSound: true, enabled: false, alarms: []))

```

When you run the app *now* you should see two groups, one black and one red, so at least you can be sure it works!

Editing groups with segues

We have a screen for editing groups, and it can be reached in two ways:

1. If the user taps one of the “Group” table view cells it will trigger the segue we put in place, in which case we need to configure the **GroupViewController** with the group that was

tapped.

2. We're going to add a button in the navigation bar that the user can tap to add a new group. We're going to make that happen by adding a group to the array, then triggering the "EditGroup" segue with that object.

I don't often use segues in my tutorials, but they work quite well in this example, I think. As a reminder, when a segue is about to happen the following method gets called:

```
prepare(for segue: UIStoryboardSegue, sender: AnyObject?)
```

The first parameter is the segue that is happening, and the second can be any object at all. When the user taps on a "Group" cell to trigger a segue, **sender** will be the cell they tapped on. We can use this **sender** parameter when we create a new group: we can call **performSegue()** directly, and pass the new group in as the **sender** parameter. Add this method now:

```
func addGroup() {  
    let newGroup = Group(name: "Name this group", playSound: true,  
enabled: false, alarms: [])  
    groups.append(newGroup)  
  
    performSegue(withIdentifier: "EditGroup", sender: newGroup)  
}
```

With that in place, there are two ways **prepare(for segue:)** will get called: if it's in response to the user tapping a cell the **sender** parameter will be set to the cell they tapped, but if it's triggered through our **addGroup()** method above then it will be a **Group** object.

With that in mind, we can write the **prepare()** method so that it does the right thing: if we get sent a group we'll use that for the **GroupViewController**, otherwise we'll pull out the matching group from the **groups** array by reading the **indexPathForSelectedRow** property of our table view.

Here's the code, wit comments:

```
override func prepare(for segue: UIStoryboardSegue, sender: AnyObject?) {
    let groupToEdit: Group

    if sender is Group {
        // we were called from addGroup(); use what it sent us
        groupToEdit = sender as! Group
    } else {
        // we were called by a table view cell; figure out which
        // group we're attached to send send that
        guard let selectedIndexPath =
            tableView.indexPathForSelectedRow else { return }
        groupToEdit = groups[selectedIndexPath.row]
    }

    // unwrap our destination from the segue
    if let groupViewController = segue.destinationViewController as?
        GroupViewController {
        // give it whatever group we decided above
        groupViewController.group = groupToEdit
    }
}
```

Before you run the code, we need to add two more lines to **viewDidLoad()** so that the navigation controller push and pop works properly:

```
navigationItem.rightBarButtonItem =
    UIBarButtonItem(barButtonSystemItem: .add, target: self, action:
        #selector(addGroup))

navigationItem.backBarButtonItem = UIBarButtonItem(title: "Groups",
```

```
style: .plain, target: nil, action: nil)
```

The first one adds a + button to the navigation bar that calls **addGroup()** when tapped, and the second one sets up a back button that says “Groups” rather than “Alarmadillo”.

Even with those changes, the code still won’t do much when you run it: tapping a cell will take you a big empty view, and tapping the + button will take you to the same big empty view. Worse, when you tap to go back to the main view controller you won’t see any new group in the list – you’ll just see the same two test groups we coded in by hand.

This is easy to fix the simple way, but I’ll be asking you to return to this much later. For now, add this **viewWillAppear()** method to **ViewController**:

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)
    tableView.reloadData()
}
```

That causes the table view to refresh its data every time the user returns to it. This means if a group has been enabled or disabled, if a group has been renamed, or if alarms have been added or deleted inside a group, those changes will be reflected inside the table view.

You should be able to run the code and see a small improvement: you’ll still get taken to the same empty gray screen, but at least if you use the + button you’ll see new groups get added.

Listing alarms

Let's turn our focus to **GroupViewController**, because it has a lot of work to do. Right now our app can get to the screen, but nothing appears in there because we don't override any of the **UITableViewController** methods.

We have quite a few methods to fill in to bring this view controller to life, so I've tried to split it up by difficulty: we'll do some trivial tasks first, then a few intermediate tasks that are made easier because they are more or less repeating what we've done already, and finally some trickier tasks at the end.

Trivial tasks first...

First, we need precisely two sections: the first section to control options for the group (name, play sound, and enabled), and the second section to hold all the alarms in the current group. So, add this method to **GroupViewController**:

```
override func numberOfSections(in tableView: UITableView) -> Int {  
    return 2  
}
```

Second, I want to display the section title “Alarms” before the second section, but only if there are any alarms in the group. So, add this method:

```
override func tableView(_ tableView: UITableView,  
titleForHeaderInSection section: Int) -> String? {  
    // return nothing if we're the first section  
    if section == 0 { return nil }  
  
    // if we're still here, it means we're the second section - do we  
    // have at least 1 alarm?  
    if group.alarms.count > 0 { return "Alarms" }  
}
```

```
// if we're still here we have 0 alarms, so return nothing  
return nil  
}

---


```

Third, I want the table view to have exactly three rows in section one, but section two should have as many rows as there are alarms. Add this method to do just that:

```
override func tableView(_ tableView: UITableView,  
numberOfRowsInSection section: Int) -> Int {  
    if section == 0 {  
        return 3  
    } else {  
        return group.alarms.count  
    }  
}
```

Fourth – almost there! – we want users to be able to delete rows from the table, but only if the row is in the second section. To do that, return true from **canEditRowAt** for the rows that should be editable, like this:

```
override func tableView(_ tableView: UITableView, canEditRowAt  
indexPath: IndexPath) -> Bool {  
    return indexPath.section == 1  
}
```

Finally, when the user swipes to delete a row – which will only be possible in the second section thanks to the code above – we want to remove the deleted row from the active group's alarms and also the table, like this:

```
override func tableView(_ tableView: UITableView, commit editingStyle: UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
    group.alarms.remove(at: indexPath.row)
    tableView.deleteRows(at: [indexPath], with: .automatic)
}
```

Intermediate tasks

With the trivial changes out of the way we can focus on the bits that are a little harder, but only a little – we've covered most of these in the previous chapter, so this is just a chance for you to see how much you remember.

We set **GroupViewController** to be the delegate for the text field in the “EditableText” cell prototype so that we can determine what the user is doing. Let's conform to that protocol now by adding **UITextFieldDelegate** to the definition of **ViewController**, like this:

```
class GroupViewController: UITableViewController, UITextFieldDelegate {
```

That protocol only has optional methods, but it will at least enable code completion for all the **UITextField** callbacks. There are two in particular we care about: **textFieldDidEndEditing()** is called when the user is finished typing into the text field, and **textFieldShouldReturn()** is our chance to dismiss the keyboard.

When the user finishes typing into the text field, we're going to use that opportunity to update the **name** property of the current group, and we'll also put their new name into the navigation bar:

```
func textFieldDidEndEditing(_ textField: UITextField) {
    group.name = textField.text!
    title = group.name
```

```
}
```

And when the user taps “Done” while editing the text field, we want the keyboard to go away:

```
func textFieldShouldReturn(_ textField: UITextField) -> Bool {  
    textField.resignFirstResponder()  
    return true  
}
```

We also need to replicate a similar approach to creating new alarms as we used for creating alarm groups. That means:

1. Creating an **addAlarm()** method that adds an alarm to a group then triggers a segue.
2. Adding a bar button item in **viewDidLoad()** that will call **addAlarm()** when tapped.
3. Overriding the **prepare(for segue:)** method so that it can distinguish between being called by **addAlarm()** and being called by a table view cell segue.
4. Overriding **viewWillAppear()** so that the table view reloads its data when it returns, ensuring it stays synchronized with user changes.

Let’s tackle those items in order, starting with **addAlarm()**. This is almost identical to the **addGroup()** method from the previous chapter: it creates a group with some default values, add it to an array, then triggers a segue using the new group as the **sender** parameter. Add this now:

```
func addAlarm() {  
    let newAlarm = Alarm(name: "Name this alarm", caption: "Add an  
optional description", time: Date(), image: "")  
    group.alarms.append(newAlarm)  
  
    performSegue(withIdentifier: "EditAlarm", sender: newAlarm)  
}
```

The second step is to add a bar button item to call `addAlarm()`, again following **ViewController**. This time, though, we're not going to modify the back button because it will contain useful information (the group name), but we are going to set the view controller's initial title based on the group's name. This will change if the user renames the group, but we need a sensible default. Add this method now:

```
override func viewDidLoad() {
    super.viewDidLoad()

    navigationItem.rightBarButtonItem =
    UIBarButtonItem(barButtonSystemItem: .add, target: self, action:
    #selector(addAlarm))

    title = group.name
}
```

Step three is to set up the segue so that we pass the selected alarm onto the **AlarmViewController**, and this uses an identical technique to **ViewController**: if we receive an alarm pass it on, otherwise figure out which table view cell was tapped and use *that* alarm instead.

Here's the code:

```
override func prepare(for segue: UIStoryboardSegue, sender:
AnyObject?) {
    let alarmToDelete: Alarm

    if sender is Alarm {
        alarmToDelete = sender as! Alarm
    } else {
        guard let indexPath =
        tableView.indexPathForSelectedRow else { return }
    }
}
```

```
        alarmToEdit = group.alarms[selectedIndexPath.row]

    }

    if let alarmViewController = segue.destinationViewController as?
AlarmViewController {

    alarmViewController.alarm = alarmToEdit

}

}
```

Finally, we need to have identical code in **viewWillAppear()** to reload the table whenever it appears. This means that if users tap a cell to edit an alarm, rename it, then return to **GroupViewController**, we need the table to be updated to reflect that. So, add this:

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)

    tableView.reloadData()
}
```

The real work

The main piece of work is not yet done, but now it's time to face it: we need to create the **cellForRowAt** method. Now, normally this method isn't too difficult, but here there's quite a bit of logic required because we need to load one of four cell types depending on what index path was requested.

To make things easier to read, I've split **cellForRowAt** into two parts: the method itself, and a separate method that handles loading “EditableText” and “Switch” cells. This means all the alarms are handled directly in **cellForRowAt**, but the other cells that require custom configuration are handled elsewhere.

With that split, here's what `cellForRowAt` needs to do:

1. If we're in section 0, pass the call onto the `createGroupCell()` method, which we'll come to in a moment.
2. Otherwise, dequeue one of the "RightDetail" cells we created when designing the storyboard.
3. Pull out the alarm at the index path we're loading.
4. Feed its `time` property into the `DateFormatter` class to extract the time in the user's locale.

And that's it – hurray for pushing work out to a different method!

Here's the code, complete with numbers matching the list above:

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {  
    if indexPath.section == 0 {  
        // 1: pass the hard work onto a different method if we're in  
        // the first section  
        return createGroupCell(for: indexPath, in: tableView)  
    } else {  
        // 2: if we're here it means we're an alarm, so pull out a  
        // RightDetail cell for display  
        let cell = tableView.dequeueReusableCell(withIdentifier:  
            "RightDetail", for: indexPath)  
  
        // 3: pull out the correct alarm from the alarms array  
        let alarm = group.alarms[indexPath.row]  
  
        // 4: use the alarm to configure the cell, drawing on  
        // DateFormatter's localized date parsing  
        cell.textLabel?.text = alarm.name  
        cell.detailTextLabel?.text =  
            DateFormatter.localizedString(from: alarm.time, dateStyle: .none,
```

```
timeStyle: .short)

    return cell
}

}
```

That's the easy bit. The hard bit is the **createGroupCell()** method, but it's only hard because it's *long*. This one method is responsible for loading all the cells in the first section of **GroupViewController**, which means it needs to read the row being requested and load either an “EditableText” cell (for row 0), a “Switch” cell (for row 1), or another “Switch” cell (for row 2).

That last part is the source of a little complexity: we're using the same cell for two different things, so we need a way to identify which switch got changed. There are other ways this could be done, for example creating separate prototype cells, but I'm going for the easiest option: we're just going to assign switch a unique tag.

So, add these two properties to **GroupViewController** now:

```
let playSoundTag = 1001
let enabledTag = 1002
```

I've made them constants for easier referencing, because we can now use them inside **createGroupCell()**, and also re-use them inside the **switchChanged()** method we created earlier.

Let's tackle **createGroupCell()** now. I've added comments throughout to help you along:

```
func createGroupCell(for indexPath: IndexPath, in tableView:
UITableViewController) -> UITableViewCell {
    switch indexPath.row {
        case 0:
```

```

    // this is the first cell: editing the name of the group
    let cell = tableView.dequeueReusableCell(withIdentifier:
"EditableText", for: indexPath)

    // look for the text field inside the cell...
    if let cellTextField = cell.viewWithTag(1) as? UITextField {
        // ...then give it the group name
        cellTextField.text = group.name
    }

    return cell
}

case 1:
    // this is the "play sound" cell
    let cell = tableView.dequeueReusableCell(withIdentifier:
"Switch", for: indexPath)

    // look for both the label and the switch
    if let cellLabel = cell.viewWithTag(1) as? UILabel,
cellSwitch = cell.viewWithTag(2) as? UISwitch {
        // configure the cell with correct settings
        cellLabel.text = "Play Sound"
        cellSwitch.isOn = group.playSound

        // set the switch up with the playSoundTag tag so we know
        // which one was changed later on
        cellSwitch.tag = playSoundTag
    }

    return cell
}

default:

```

```

        // if we're anything else, we must be the "enabled" switch,
        which is basically the same as the "play sound" switch

        let cell = tableView.dequeueReusableCell(withIdentifier:
    "Switch", for: indexPath)

        if let cellLabel = cell.viewWithTag(1) as? UILabel,
cellSwitch = cell.viewWithTag(2) as? UISwitch {
            // obviously it's configured a little differently below!
            cellLabel.text = "Enabled"
            cellSwitch.isOn = group.enabled
            cellSwitch.tag = enabledTag
        }

        return cell
    }
}

```

With special tags being assigned to each **UISwitch**, we can finally fill in the **switchChanged()** method: if the changed switch has the tag **playSoundTag** then we update the **playSound** property of the current group, otherwise we update the **enabled** property:

```

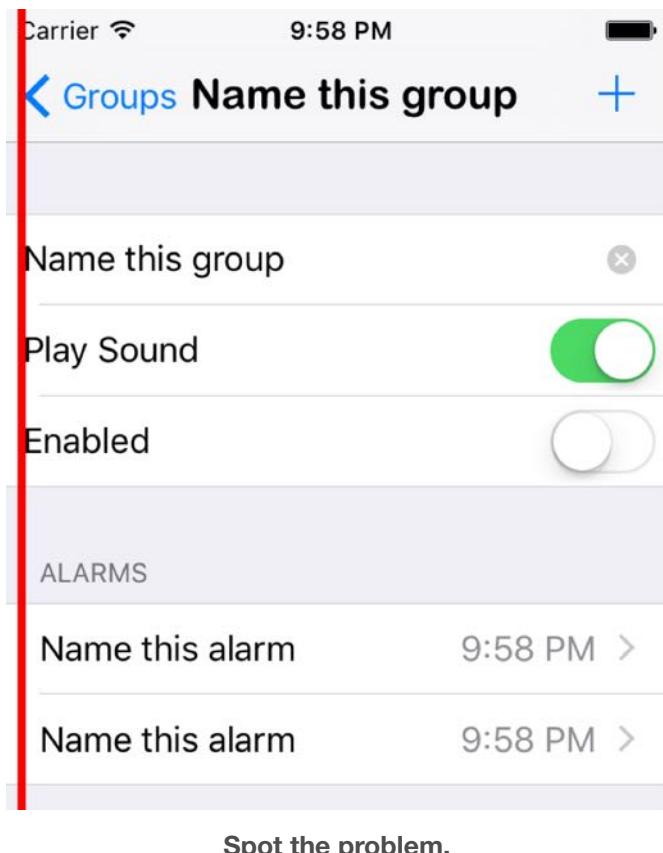
@IBAction func switchChanged(_ sender: UISwitch) {
    if sender.tag == playSoundTag {
        group.playSound = sender.isOn
    } else {
        group.enabled = sender.isOn
    }
}

```

Almost there!

After all that code, you'll be pleased to know that **GroupViewController** is almost entirely functional – you can now create groups, rename them and adjust their settings, then create alarms for those groups.

But there's one small problem, shown in the screenshot below:



Spot the problem.

No, the screen doesn't have a massive red line down it – that was added by me to demonstrate the actual problem: our custom-designed cells have text that's further to the left than the built-in cells.

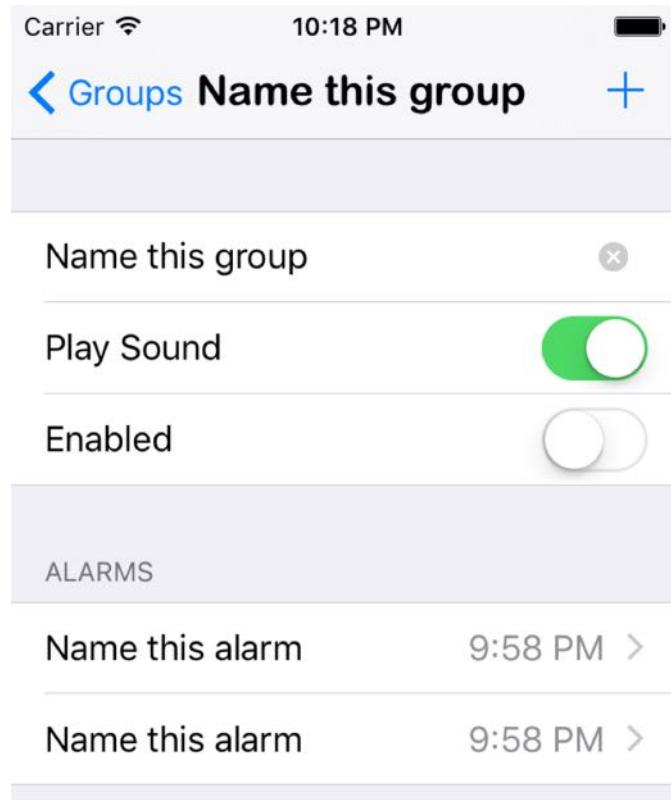
I don't know about you, but I'm a perfectionist and this kind of tiny glitch irritates me. Interface Builder used to have some options to fix this, but they seem to have disappeared in the current Xcode beta releases so we're forced to do it in code.

The fix is this: when you're about to display a cell, set its

`cell.preservesSuperviewLayoutMargins` property to be true, then do the same for the cell's content view. All you need to do is add this method to `GroupViewController` and the layout will magically update:

```
override func tableView(_ tableView: UITableView, willDisplay cell: UITableViewCell, forRowAt indexPath: IndexPath) {  
    cell.preservesSuperviewLayoutMargins = true  
    cell.contentView.preservesSuperviewLayoutMargins = true  
}
```

Run the app again and you'll see it looks much better!



Once we fix the margins with a little bit of code, our custom cells look identical to the built-in iOS cells.

Editing alarms

Now for **AlarmViewController**, which is where users edit the alarms they have created. This is the last view controller in the set, and fortunately it's a great deal simpler than **GroupViewController**.

Again I want to start by getting the easy stuff out of the way so I can focus on the code that matters. This view controller needs to respond to the user changing the name and description, and also needs to handle the image picker being shown when the tap gesture recognizer is triggered. So, please start by adding some new protocols to the class definition:

```
class AlarmViewController: UITableViewController,  
UITextFieldDelegate, UIImagePickerControllerDelegate,  
UINavigationControllerDelegate {
```

When we get the **textFieldDidEndEditing()** message we need to update the alarm using the new data. When the method gets called it passes in the text field that just changed, but we don't need to do it here because we have individual outlets for both text fields – we can just update both values and skip the **if** statement. Add this now:

```
func textFieldDidEndEditing(_ textField: UITextField) {  
    alarm.name = name.text!  
    alarm.caption = caption.text!  
    title = alarm.name  
}
```

We also need to implement **textFieldShouldReturn()** so that both text fields get rid of the keyboard when they are finished:

```
func textFieldShouldReturn(_ textField: UITextField) -> Bool {  
    textField.resignFirstResponder()  
    return true
```

```
}
```

The last easy change to make is to fill in the **datePickerChanged()** method. All that needs to do is save the date picker's current date into the **time** property of the current alarm, so update the method to this:

```
@IBAction func datePickerChanged(_ sender: UIDatePicker) {  
    alarm.time = datePicker.date  
}
```

Loading and saving a picture

We want the user to attach pictures to their reminders, which means filling in the **imageViewTapped()** method that's triggered by the tap gesture recognizer on the image. We can make that happen by reusing code from the Happy Days project, so fill in the method like this:

```
@IBAction func imageViewTapped(_ sender: UIImageView) {  
    let vc = UIImagePickerController()  
    vc.modalPresentationStyle = .formSheet  
    vc.delegate = self  
    present(vc, animated: true)  
}
```

Where things get more interesting is handling the *result* of the image picker, because that needs to do a number of things:

1. Dismiss the image picker
2. Fetch the image that was picked.
3. If the current alarm already has an image attached, delete it.

4. Generate a new filename for the image and assign it to the alarm image.
5. Write the new image to the user's documents directory.
6. Update the user interface.

Here's the code, with numbered comments matching the above:

```
func imagePickerController(_ picker: UIImagePickerController,
didFinishPickingMediaWithInfo info: [String : AnyObject]) {
    // 1: dismiss the image picker
    dismiss(animated: true, completion: nil)

    // 2: fetch the image that was picked
    guard let image = info[UIImagePickerControllerOriginalImage] as?
    UIImage else { return }

    let fm = FileManager()

    if alarm.image.characters.count > 0 {
        // 3: the alarm already has an image, so delete it
        do {
            let currentImage = try
            Helper.getDocumentsDirectory().appendingPathComponent(alarm.image)

            if fm.fileExists(atPath: currentImage.path ?? "") {
                try fm.removeItem(at: currentImage)
            }
        } catch {
            print("Failed to remove current image")
        }
    }

    do {
        // 4: generate a new filename for the image
    }
}
```

```

alarm.image = "\UUID().uuidString).jpg"

// 5: write the new image to the documents directory
let newPath = try
Helper.getDocumentsDirectory().appendingPathComponent(alarm.image)

let jpeg = UIImageJPEGRepresentation(image, 80)
try jpeg?.write(to: newPath)

} catch {
    print("Failed to save new image")
}

// 6: update the user interface
imageView.image = image
tapToSelectImage.isHidden = true
}

```

You'll notice I slipped in a couple of calls to **Helper.getDocumentsDirectory()**, which is the same method we've used previously to get the user's documents directory. This time I moved it into a struct called **Helper** because it will be used elsewhere.

So, go to File > New > File, then choose iOS > Source > Swift File, name it "Helper", then click Create. Now give it this content:

```

struct Helper {

    static func getDocumentsDirectory() -> URL {
        let paths =
FileManager.default.urlsForDirectory(.documentDirectory,
inDomains: .userDomainMask)

        let documentsDirectory = paths[0]

        return documentsDirectory
    }
}

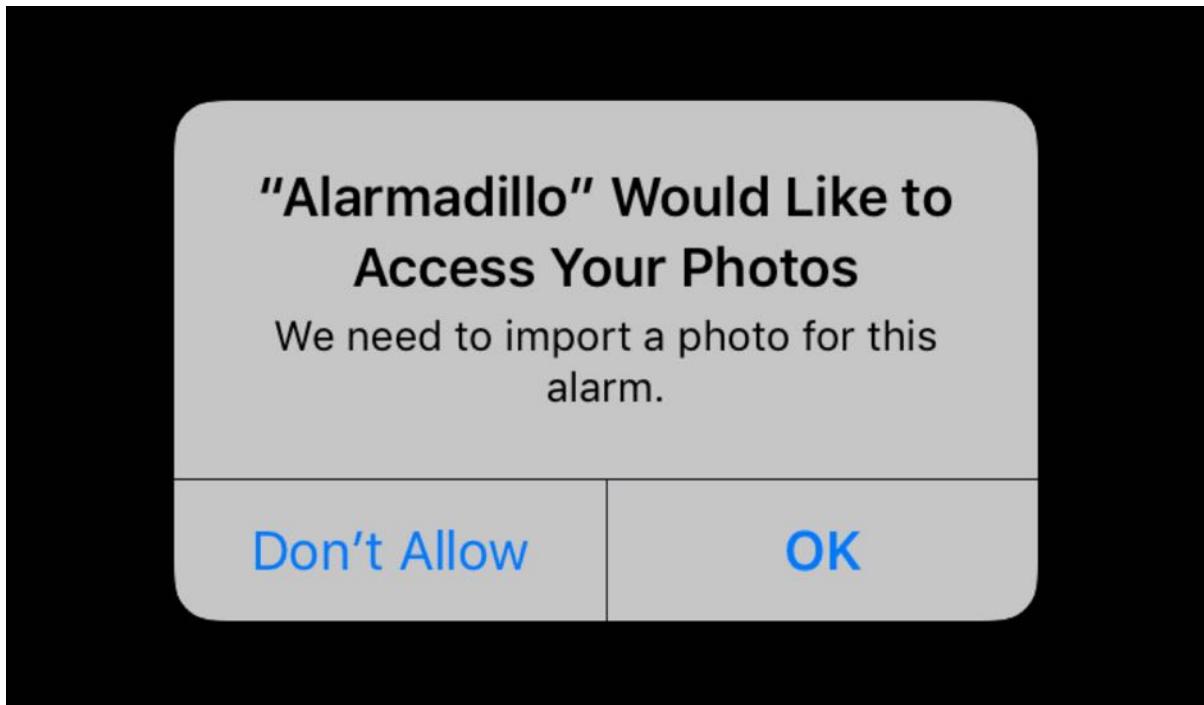
```

```
    }  
}
```

The last code change we need is to override **viewDidLoad()** to set up our user interface correctly. This involves setting things like the view controller's title and the date picker's date, but also loading an image if one exists. Add this now:

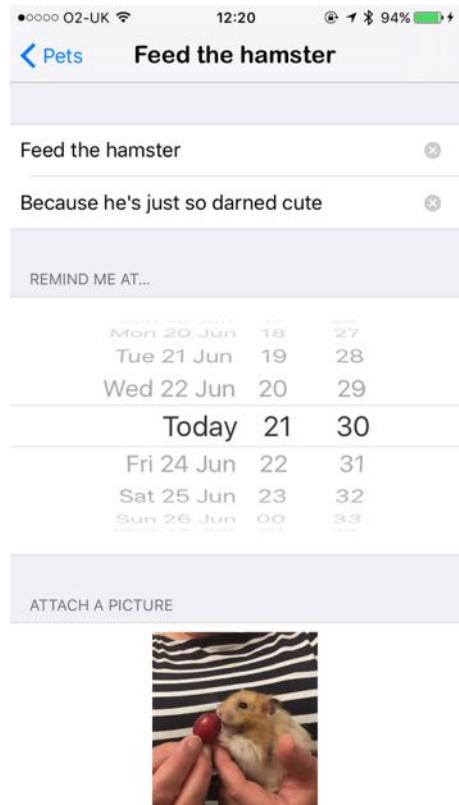
```
override func viewDidLoad() {  
    super.viewDidLoad()  
  
    title = alarm.name  
    name.text = alarm.name  
    caption.text = alarm.caption  
    datePicker.date = alarm.time  
  
    if alarm.image.characters.count > 0 {  
        // if we have an image, try to load it  
        if let imageFilename = try?  
            Helper.getDocumentsDirectory().appendingPathComponent(alarm.image) {  
                imageView.image = UIImage(contentsOfFile:  
                imageFilename.path!)  
            }  
  
            tapToSelectImage.isHidden = true  
    }  
}
```

Before you run your code, there's one last non-code thing we need to do, but you might have already figured it out: when you want to load photos, you need to explain why you want access. To do that, open your Info.plist file and add a key named "NSPhotoLibraryUsageDescription" with the text "We need to import a photo for this alarm."



Make sure you add a key in your Info.plist to explain to the user why you want access to their photo library.

You should be able to run the app now and use all of it – add groups, edit them, add alarms, edit those too, attach pictures, and choose dates. Things are starting to come together at last!



The app is starting to come together, and you can now create alarms and attach images to them.

Loading and saving data

Right now we let users create groups and alarms, but we don't actually write that data to disk. Well, that's not strictly true – we do write the image data to disk, but we then throw away the reference to it in the **alarm** objects we created, which is even worse!

Before this app can be considered done, we need to write code to save alarm data to disk, then read it back. The amount of data we actually need to store is quite small, because the images are being saved separately. This means we could use **User Defaults** and not have a problem, but that's not a solution I'm fond of because if you save too much there your app launch will slow down.

So, we're going to use **NSCoding** to read and write our objects, then use **NSKeyedArchiver** and **Data** to write all our data to disk. This is made complicated a little by the fact that there are lots of places where changes can happen: **ViewController** can delete groups, **GroupViewController** can rename groups and delete alarms, and **AlarmViewController** can edit alarms. All of those need to be able to trigger saving when something changes, because you rarely see a dedicated “Save” button in iOS apps.

There are lots of ways this can be tackled, not least by using dependency injection to pass a save delegate around. However, this easiest approach is to use **NotificationCenter**, which lets us send and receive notifications from anywhere in our app. What we're going to do is add a **save()** method to **ViewController**, **GroupViewController** and **AlarmViewController**, which then posts a notification that the main **ViewController** class can pick up and act on. Using this approach means you can change those **save()** methods to use a different approach later on if you want to.

Start by adding this method to both **GroupViewController** and **AlarmViewController**:

```
func save() {  
    NotificationCenter.default.post(name: Notification.Name("save") ,  
object: nil)  
}
```

That means “post the command ‘save’ to the rest of the app,” and any part of the app that wants to be notified when a save has been requested will get that message delivered.

We need to trigger those methods in various places. So, in **GroupViewController** you need to add a call to **save()** in these places:

- At the end of **switchChanged()**.
- At the end of **textFieldDidEndEditing()**.
- At the end of the **commit** method that deletes rows.
- At the end of the **addAlarm()** method.

Now in **AlarmViewController** add it in these places:

- At the end of **textFieldDidEndEditing()**.
- At the end of **datePickerChanged()**.
- Just after the line **try jpeg?.write(to: newPath)** in **didFinishPickingMediaWithInfo**.

We'll get onto **ViewController** in a moment, but first we need to add support for **NSCoding** to the **Group** and **Alarm** classes so that we can write them out.

We covered **NSCoding** back in Hacking with Swift project 12, but that was probably a long time ago for you so let's recap it briefly now. When you conform to this protocol, you need to add two new methods to your classes: one to load data from a saved state, and one to create that saved state from an existing object. To save data you make calls like this:

```
aCoder.encode(self.id, forKey: "id")
```

That means “take my **id** property and save it under the name ‘id’ so I can load it from there later.” To decode, you write this:

```
self.name = aDecoder.decodeObject(forKey: "name") as! String
```

You need to cast the result to whatever data type you saved, because **encode()** and **decodeObject()** can work with all sorts of data.

That's it! **NSCoding** isn't a complicated protocol, but I must admit it *is* a dull one because

you're just reading and writing the same stuff.

Add **NSCoding** to the protocol list for **Group**, then add these two methods:

```
required init?(coder aDecoder: NSCoder) {
    self.id = aDecoder.decodeObject(forKey: "id") as! String
    self.name = aDecoder.decodeObject(forKey: "name") as! String
    self.playSound = aDecoder.decodeBool(forKey: "playSound")
    self.enabled = aDecoder.decodeBool(forKey: "enabled")
    self.alarms = aDecoder.decodeObject(forKey: "alarms") as! [Alarm]
}

func encode(with aCoder: NSCoder) {
    aCoder.encode(id, forKey: "id")
    aCoder.encode(name, forKey: "name")
    aCoder.encode(playSound, forKey: "playSound")
    aCoder.encode(enabled, forKey: "enabled")
    aCoder.encode(alarms, forKey: "alarms")
}
```

Now add **NSCoding** to the list of protocols for **Alarm**, then add these two methods:

```
required init?(coder aDecoder: NSCoder) {
    self.id = aDecoder.decodeObject(forKey: "id") as! String
    self.name = aDecoder.decodeObject(forKey: "name") as! String
    self.caption = aDecoder.decodeObject(forKey: "caption") as!
String
    self.time = aDecoder.decodeObject(forKey: "time") as! Date
    self.image = aDecoder.decodeObject(forKey: "image") as! String
}
```

```
func encode(with aCoder: NSCoder) {  
    aCoder.encode(self.id, forKey: "id")  
    aCoder.encode(self.name, forKey: "name")  
    aCoder.encode(self.caption, forKey: "caption")  
    aCoder.encode(self.time, forKey: "time")  
    aCoder.encode(self.image, forKey: "image")  
}
```

The next step is to add some code to the **ViewController** class. With **NSCoding** support in place for **Group** and **Alarm**, there are still five more things to do:

1. Add a **save()** method to **ViewController** that performs the actual writing of data to disk.
2. Add a **load()** to read the saved data back out.
3. Call **save()** from the two places in **ViewController** that change our data.
4. Call **load()** from **viewWillAppear()** rather than just using **reloadData()** on the table.
5. Add an observer for the app-wide “save”.

Writing the **save()** method for **ViewController** is actually pretty trivial, because the **NSCoding** protocol does most of the work for us. All we need to do is figure out where we want to save it (using **getDocumentsDirectory()** again), then pass our **groups** array to **NSKeyedArchiver**. That will return a **Data** object that we can write to disk.

Here's the new **save()** method:

```
func save() {  
    do {  
        let path = try  
        Helper.getDocumentsDirectory().appendingPathComponent("groups")  
        let data = NSKeyedArchiver.archivedData(withRootObject:  
        groups)  
        try data.write(to: path)  
    } catch {
```

```
        print("Failed to save")
    }
}
```

Loading saved data is only fractionally harder: it's the same thing in reverse, with the addition that we need to create an empty array if there isn't one saved. I'm also going to have the table view reload itself when loading finishes, so that new data is reflected in the UI.

Here's the new method:

```
func load() {
    do {
        let path = try
            Helper.getDocumentsDirectory().appendingPathComponent("groups")
        let data = try Data(contentsOf: path)
        groups = NSKeyedUnarchiver.unarchiveObject(with: data) as?
        [Group] ?? [Group]()
    } catch {
        print("Failed to load")
    }

    tableView.reloadData()
}
```

The “failed to load” message will appear every time the app couldn't load something from disk, which means it will *definitely* appear the first time the app runs because there was nothing to load – don't worry about it!

The third task is to add calls to **save()** wherever **ViewController** changes the data. Right now, that's in **commit** and **addGroup()**, so please add a call to **save()** in both those places now.

The fourth task is to add a call to **load()** in **viewWillAppear()**, in place of the existing call to **reloadData()**, so just find this line:

```
tableView.reloadData()
```

And replace it with this:

```
load()
```

The final task is to add an observer for the “save” notification that gets posted from **GroupViewController** and **AlarmViewController**. Put this line of code into **viewDidLoad()**:

```
NotificationCenter.default.addObserver(self, selector:  
#selector(save), name: Notification.Name("save"), object: nil)
```

That's it! You should remove the two calls to **groups.append()** in **viewDidLoad()** because we don't need to insert dummy data any more – in fact doing so will probably just confuse you, because it will keep getting added as the app runs.

Adding local notifications

This has already been an epic project, and we're only just now making it to the new features in iOS 10. If you've been following along from scratch, I hope you've managed to learn from what we've implemented so far: segues, static table view cells, **NSCoding**, notifications, and more. If you skipped over that part because you want to focus only on what's new in iOS 10, that's fine too – there are so many new things to learn that it can be easier just to focus on the new stuff!

It's now time to explore the new notification framework in iOS 10. Apple claims that it has feature parity with the notifications from previous releases, which isn't quite true – at least not yet. But it does do more than enough for our purposes, and even adds things like being able to show media and allowing users to type responses directly into the notification.

I designed the Alarmadillo app specifically to let us exploit the power of the new notifications. We'll be displaying the name and description of each alert that gets triggered, we'll show any image that was attached, and we'll let users rename or delete groups right from within the notification on their lock screen.

To make all this work, we need to pull in the `UserNotifications` framework, so please start by adding this import to **ViewController** and also `AppDelegate.swift`:

```
import UserNotifications
```

We need to put it in the app delegate as well as **ViewController**, because that's where notifications are delivered.

In **ViewController**, we need to set up the notifications whenever the user takes any action. So, if they rename an alarm, delete it, change the time, attach a picture, or enable or disable a whole group, we need to recreate our alarms because something changed.

The easiest way to make that happen is by adding this line to the end of the `save()` method in **ViewController**:

```
updateNotifications()
```

We haven't written that method yet, but what it means is that every time we save – i.e., make any change – we also update our system notifications. You can try and do it in a more fine-grained way if you wish, but there isn't really much benefit.

The **updateNotifications()** method is actually quite easy, because I'm going to make it do only two things: request permission to show notifications, then pass the job of scheduling notifications onto a different method.

To request permission to show notifications, we need to read the user notification center for our application using **UNUserNotificationCenter.current**. Using that, we can request authorization to show badges, sounds, and alerts depending on our needs, all by calling the **requestAuthorization()** method and passing an array of what things we want to show along with a closure to run when permission has been granted. In that closure iOS will tell us whether permission was granted, at which point we can go ahead and schedule our notifications.

Here's the code for **updateNotifications()**:

```
func updateNotifications() {
    let center = UNUserNotificationCenter.current

    center.requestAuthorization(options: [.alert, .sound]) { [unowned self] (granted, error) in
        if granted {
            self.createNotifications()
        }
    }
}
```

Scheduling notifications

It takes a *lot* of work to create notifications: you need to figure out which groups and alarms

are enabled, set the title and description, add a sound and an image, set the alert time, and more.

To make it easier to understand, I split the code up into three methods, so please add these three method stubs now:

```
func createNotifications() {  
}  
  
func createNotificationRequest(group: Group, alarm: Alarm) ->  
UNNotificationRequest {  
}  
  
func createNotificationAttachments(alarm: Alarm) ->  
[UNNotificationAttachment] {  
}
```

The first of those, **createNotifications()**, is responsible for removing any notifications currently scheduled so that we have a clean slate, then going through all the groups and alarms to see which need to be scheduled. The actual job of creating individual notifications will be done in the **createNotificationRequest()** method, which will set up the title, attach a sound, then set a time. Finally, **createNotificationAttachments()** is responsible for adding the user's image to the notification, if one is set.

You've added stubs for those methods so that your code will compile as we progress, rather than you constantly being nagged with errors.

Let's start with **createNotifications()**. This needs to:

1. Remove any pending notification requests, i.e. things we had scheduled previously but hadn't been delivered.
2. Loop through every group, ignoring those groups that aren't enabled.
3. Loop through every alarm in the enabled groups and call **createNotificationRequest()** for each one.
4. That method will return a **UNNotificationRequest** object, which can then be passed to

the user notification center for delivery.

That's it: one small, meaningful chunk. Here's the code:

```
func createNotifications() {
    let center = UNUserNotificationCenter.current

    // 1: remove any pending notifications
    center.removeAllPendingNotificationRequests()

    for group in groups {
        // 2: ignore disabled groups
        guard group.enabled == true else { continue }

        for alarm in group.alarms {
            // 3: create a notification request from each alarm
            let notification = createNotificationRequest(group:
                group, alarm: alarm)

            // 4: schedule that notification for delivery
            center.add(notification) { error in
                if let error = error {
                    print("Error scheduling notification: \(error)")
                }
            }
        }
    }
}
```

That's one method down. The next one, **createNotificationRequest()**, needs to accept a

group and an alarm, and return a **UNNotificationRequest** object. That object encapsulates everything required to deliver the notification: its title and subtitle, whether it plays a sound or not, whether it has an attachment or not, what time it should be triggered, and more.

We'll also be using this method to attach to extra pieces of data to each notification: **categoryIdentifier** is an internal reference we'll use later to add buttons to the notification, and **userInfo** is a dictionary that contains any context information we want to attach to the notification – in our case we'll store a group and alarm ID. That user info dictionary will be handed back to us when the notification is triggered, which means we'll be able to read our context data back.

Before we dive into the code, there are a few things I want to go over first. There are several ways of triggering notifications, but the one we'll be using is called **UNCalendarNotificationTrigger** – “trigger the notification when this date is reached.” Another useful option is **UNTTimeIntervalNotificationTrigger**, which means “trigger the notification when a certain amount of time has passed from now.” This is helpful for debugging because it lets you test notifications quickly; I'll give you some code for that soon.

Like I said, we'll be using **UNCalendarNotificationTrigger** to specify when our alarms should go off. But cunningly, that trigger doesn't work with **Date** objects. Instead, it works with **DateComponents** objects, which are much more useful: you can specify any or all of the year, month, day, hour, minute, or second of the alert, and it will figure out the rest. In our case, our alarms might be stored as **Date** objects, but really we only care about the hour and minute. So, we'll pull out the hour and minute from each alarm, and create our **DateComponents** object from that so the notification will trigger at that time regardless of the day.

Even better, the initializer for **UNCalendarNotificationTrigger** has a **repeats** parameter: set that to true, and the alarm will repeat whenever its date components match. Want to repeat an alarm every hour, every day, or on the first day of every month? That's how you do it.

If you've never tried to pull out individual components from a **Date** object, you'll be pleased to know it's not hard. In fact, the **Calendar** object can do all the work for you. You can use code like this, for example, to pull out the hour component from a date:

```
let hour = Calendar.current.component(.hour, from: someDate)
```

One last thing: when you create a **UNNotificationRequest** object, you need to provide it with an identifier. This should be a unique name you can use later on if you want to check, edit, or remove a specific notification. We'll be generating a **UUID** string again because we don't really care about these identifiers.

Here's the code for **createNotificationRequest()**, with added comments to guide you through:

```
func createNotificationRequest(group: Group, alarm: Alarm) ->
UNNotificationRequest {
    // start by creating the content for the notification
    let content = UNMutableNotificationContent()

    // assign the user's name and caption
    content.title = alarm.name
    content.body = alarm.caption

    // give it an identifier we can attach to custom buttons later on
    content.categoryIdentifier = "alarm"

    // attach the group ID and alarm ID for this alarm
    content.userInfo = ["group": group.id, "alarm": alarm.id]

    // if the user requested a sound for this group, attach their
    default alert sound
    if group.playSound {
        content.sound = UNNotificationSound.default()
    }

    // use createNotificationAttachments to attach a picture for this
    alert if there is one
    content.attachments = createNotificationAttachments(alarm: alarm)
```

```

// get a calendar ready to pull out date components
let cal = Calendar.current

// pull out the hour and minute components from this alarm's date
var dateComponents = DateComponents()
dateComponents.hour = cal.component(.hour, from: alarm.time)
dateComponents.minute = cal.component(.minute, from: alarm.time)

// create a trigger matching those date components, set to repeat
let trigger = UNCalendarNotificationTrigger(dateMatching:
dateComponents, repeats: true)

// combine the content and the trigger to create a notification
request

let request = UNNotificationRequest(identifier:
UUID().uuidString, content: content, trigger: trigger)

// pass that object back to createNotifications() for scheduling
return request
}

```

That code is all correct and useful, but if you just want to test that notifications work try adding this trigger instead of the **UNCalendarNotificationTrigger**:

```
let trigger = UNTimeIntervalNotificationTrigger(timeInterval: 3,
repeats: false)
```

That will show an alert after a three-second delay has passed, which makes it perfect for testing out changes quickly.

That just leaves the **createNotificationAttachments()** method, which needs to attach an image to the notification if one was set. This method has a massive proviso in it that I totally missed when I got my hands on the iOS 10, so I'm going to write it for you really clearly:

When you attach an image to a notification, it gets moved into a separate location so that it can be guaranteed to exist when shown. That means you cannot just use the same file we placed into the documents directory, because it will get moved away – and thus lost.

So, let's break down what the **createNotificationAttachments()** method needs to do:

1. If there is no image attach to an alert, return nothing.
2. Get the full path to the alarm image by using **getDocumentsDirectory()**.
3. Create a full path to a temporary filename that we'll use to take a copy of the alert image.
4. Copy the alert image to the temporary file.
5. Create a **UNNotificationAttachment** object with a random identifier, pointing it at the file copy we just made.
6. Return that attachment back to **createNotificationRequest()**.

Remember, creating an attachment *moves* the file you attach rather than taking a copy, so we need to take our own copy otherwise it will get lost.

Here's the code for **createNotificationAttachments()**, complete with numbered comments matching the above:

```
func createNotificationAttachments(alarm: Alarm) ->
[UNNotificationAttachment] {

    // 1: return if there is no image to attach
    guard alarm.image.characters.count > 0 else { return [] }

    let fm = FileManager.default

    do {
        // 2: get the full path to the alarm image
        let imageURL = try
Helper.getDocumentsDirectory().appendingPathComponent(alarm.image)
```

```

    // 3: create a temporary filename
    let copyURL = try
        Helper.getDocumentsDirectory().AppendingPathComponent("\(UUID().uuidString).jpg")

    // 4: copy the alarm image to the temporary filename
    try fm.copyItem(at: imageURL, to: copyURL)

    // 5: create an attachment from the temporary filename,
    giving it a random identifier
    let attachment = try UNNotificationAttachment(identifier:
        UUID().uuidString, url: copyURL)

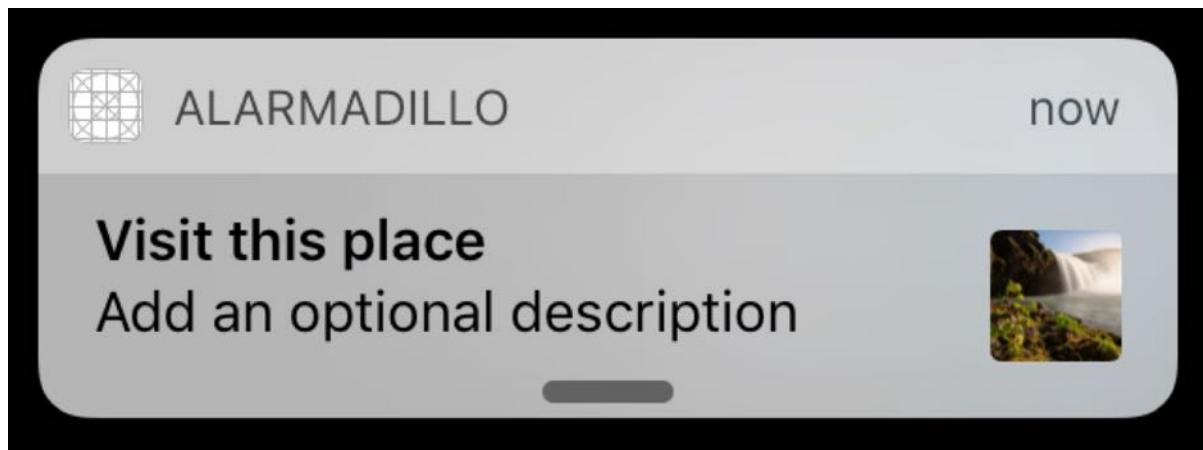
    // 6: return the attachment back to
    createNotificationRequest()

    return [attachment]
} catch {
    print("Failed to attach alarm image: \(error)")
    return []
}
}

```

If you didn't believe me when I told you earlier that "it takes a *lot* of work to create notifications," I think you believe me now. The framework is both powerful and flexible, but it's certainly not easy to get started with!

There's still more to do, but for now you can at least go ahead and test out basic notification delivery. You might find it easier to switch to the **UNTimeIntervalNotificationTrigger** trigger for the time being.



At this point you should be able to schedule and read notifications from the app.

Responding to user input

Here's where notifications get interesting: we can attach custom buttons to them that give users a chance to take immediate action, and we can even read a user's text input and act on it.

When we create the notification request in the previous chapter, I included this line of code:

```
content.categoryIdentifier = "alarm"
```

That assigns a category name to the notification, which is a free-text string of our own choosing. By itself it means nothing, but you can attach actions to that category and iOS will display them on your behalf.

To finish up this app, we're going to create three different kinds of action: one that will launch the app in the foreground, one that will require the user to enter some text, and one that will destroy content and thus require the user to unlock the device.

Registering notification categories is best done when your app runs, so we need to work in `AppDelegate.swift`. You'll see an existing `didFinishLaunchingWithOptions` method in there that gets run every time the app launches, and that's the perfect place to register our categories.

We're going to create a single category called "alarm", to match the `categoryIdentifier` we assigned when creating the notification request. Inside that category we're going to create three actions: "Show Group" will open the app and take the user to the group that triggered the alarm, "Destroy Group" will destroy the group that created the alarm but won't launch the app in the foreground, and "Rename Group" will ask the user to enter a new name for the group as part of the notification UI.

The first two of those, "Show Group" and "Destroy Group", are both done using the `UNNotificationAction` class. It has an initializer that accepts three values: an identifier string that tells you which button was tapped, a title string that is shown to the user, and a list of options. There are three options to choose from: `.authenticationRequired` will require the user to unlock their device before being activated, `.destructive` will mark the action with red text so the user knows it's dangerous, and `.foreground` will trigger a full app launch in the foreground when tapped.

The third action, “Rename Group”, will use the `UNTextInputNotificationAction` instead, because that will prompt the user to enter some text and tap a button to submit. This inherits from `UNNotificationAction` and so has the same identifier, title, and options initializer, but adds two more options: what title should be on the submit button, and what placeholder text should be in the text field they type into.

Once you’ve created all the actions you want, you wrap them into a `UNNotificationCategory` object. You need to give this the same identifier you used with `content.categoryIdentifier`, then give it two arrays of actions: the first, called `actions`, is an array of up to four actions you want to show; the second, called `minimalActions`, is an array of up to two actions that should be shown when space is limited.

The initializer also lets you specify an array of intent identifiers if you’re hooking to SiriKit (which we aren’t), as well as an array of options. Right now, there are only two things that can go into that options array: `.customDismissAction` requests that you be notified when the user dismisses your notification (i.e., swipes it away rather than choosing one of your actions), and `.allowInCarPlay` requests that the notification be allowed in CarPlay.

Finally, once you’ve created your actions and wrapped them inside a category, you register that category (and any others) with the system by using the `setNotificationCategories()` method of the user notification center. We’re going to do all that inside the `didFinishLaunchingWithOptions` method in `AppDelegate.swift`, but we’re also going to do one more thing: we’re going to set our `ViewController` to be the delegate for the user notification center.

This change is important, because it means the alerts come in they will get handled by the `ViewController` instance that is active in the app. This change is also new, because before iOS 10 all notifications came in to your app delegate so you needed to handle them there – being able to use a different object to process alerts is a huge improvement and makes for much simpler code.

Now, we want `didFinishLaunchingWithOptions` to set up our main view controller as the delegate for notifications, but it doesn’t have a property pointing to that. Instead, it has a `window` property. That `window` property has a `rootViewController` property, which will contain the `UINavigationController` we put in place early on. That navigation controller has a `viewControllers` property, which contains all the view controllers on its stack, and *that’s* what we need: `viewControllers[0]` should contain an instance of our `ViewController` class. Of course, “should” means nothing in code, so we’ll be using `if/let` to safely unwrap all the

above.

It's time to put all that into action with some real code. Here's a new version of **didFinishLaunchingWithOptions** for you to put into AppDelegate.swift:

```
func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
    let center = UNUserNotificationCenter.current

    if let navController = window?.rootViewController as? UINavigationController {
        if let viewController = navController.viewControllers[0] as? ViewController {
            // if we're here, it means we found our active `ViewController` object
            center.delegate = viewController
        }
    }

    // create the three actions we want for our alert
    let show = UNNotificationAction(identifier: "show", title: "Show Group", options: .foreground)

    let destroy = UNNotificationAction(identifier: "destroy", title: "Destroy Group", options: [.destructive, .authenticationRequired])

    let rename = UNTextInputNotificationAction(identifier: "rename", title: "Rename Group", options: [], textInputButtonTitle: "Rename", textInputPlaceholder: "Type the new name here")

    // wrap the actions inside a category
    let category = UNNotificationCategory(identifier: "alarm", actions: [show, rename, destroy], minimalActions: [show, rename], intentIdentifiers: [], options: [.customDismissAction])
}
```

```
// register the category with the system
center.setNotificationCategories([category])

return true
}
```

That code will *not* compile just yet, because **ViewController** needs to conform to the **UNUserNotificationCenterDelegate** delegate first, so please add that now.

Implementing the notification delegate

That delegate has two methods, but both are optional so your code should compile cleanly. The first method in the protocol is easy, and is called when a notification has come in while your app is running. If you don't implement it then the notification gets silently ignored, but if you choose to implement this method you can tell iOS you want some or all of the notification to be shown.

For example, in Alarmadillo let's make it so that if one of our notifications is triggered while the app is running we're only going to show the message – we're not going to play the sound no matter what the user requested. To do that, just implement **willPresent** like this:

```
func userNotificationCenter(_ center: UNUserNotificationCenter,
willPresent notification: UNNotification, withCompletionHandler
completionHandler: (UNNotificationPresentationOptions) -> Void) {
    completionHandler([.alert])
}
```

If you wanted the sound as well, just add it like this:

```
completionHandler([.alert, .sound])
```

That's the first method of **UNUserNotificationCenterDelegate** complete. The second, more challenging one is called **didReceive**, and is triggered when the user acts on a notification and it's down to you to respond. In our case we have three actions that we need to deal with: displaying a group, deleting a group, and renaming a group. We're going to start by writing individual methods for each of these inside the **ViewController** class, because none of them are difficult.

First, a method to display a group. This will be called – surprise! – **display()**, and expects to be given the ID of a group to show. Remember, we attached a **userInfo** dictionary to the notifications that contained the ID of the group and alarm that triggered it, so we'll have that ID passed in now.

The **display()** method, and indeed all three methods we're going to write, will start by calling **popToRootViewController()** on the navigation controller. This is required: we don't want to manipulate our data while the user is several screens deep, because they might later try to rename an alarm in a group that no longer exists. Going back to the root view controller before making any changes means we can be sure we're in a sensible, safe place before running any new code.

Once we're back at the root view controller, **display()** will loop through all the groups trying to find the one that was used to display the notification. When it finds it, we can call **performSegue()** using that group as the **sender** parameter, just like we did when creating new groups – hurray for code re-use!

Add this method to **ViewController** now:

```
func display(group groupID: String) {
    _ = navigationController?.popToRootViewController(animated:
false)

    for group in groups {
        if group.id == groupID {
```

```
        performSegue(withIdentifier: "EditGroup", sender: group)
    return
}
}

}
```

The second method is called **destroy()**, and will be triggered by the “Destroy Group” action. This is somewhat similar to **rename()** in that it starts by going back to the root view controller then loops over every group in the **groups** array. However, when it finds the matching group it will remove it from the **groups** array, then save those changes and reload the groups.

Here's the code:

```
func destroy(group groupID: String) {
    _ = navigationController?.popToRootViewController(animated:
false)

    for (index, group) in groups.enumerated() {
        if group.id == groupID {
            groups.remove(at: index)
            break
        }
    }

    save()
    load()
}
```

Finally, we need a **rename()** method that will accept a group ID as well as a string to use for the group's new name. Otherwise, this is almost identical to the previous two:

```

func rename(group groupID: String, newName: String) {
    _ = navigationController?.popToRootViewController(animated:
false)

    for group in groups {
        if group.id == groupID {
            group.name = newName
            break
        }
    }

    save()
    load()
}

```

Warning: The last two of those methods call **save()** and **load()** because they are making changes to the data. When you save the notifications Alaramadillo will automatically reschedule them, so if you left in the debugging code for interval triggers you'll find they get rescheduled every time you take an action!

Those are the three methods we want to run when the user takes one of the actions we registered in our category. The next job – and final one for this app – is to implement the second method of the **UNUserNotificationCenterDelegate** protocol, which is **didReceive()**. This is triggered when the user has acted on a notification and it's down to us to do something with it.

We already did most of the work by writing the three methods above, so really there's not much work left for **didReceive()**. However, there are still a few things it needs to do:

1. When the notification comes in, we need to pull out the **userInfo** dictionary we set because that contains the group ID that triggered the alarm. This is buried quite deep, as you'll see.
2. We also need to read the **actionIdentifier** value to see what action the user tapped, e.g. "show" or "rename".

3. It has a **completionHandler** block that must be called when you've finished work.

Point 2 has a small extra complexity that you need to be aware of: there are two extra action identifiers that you might get sent, above and beyond the three we registered. The first is **UNNotificationDefaultActionIdentifier**, which is sent to you when the user swipes on your notification to unlock their device – they didn't choose an action from your list, but they clearly want to know more about the notification. The second is **UNNotificationDismissActionIdentifier**, which is sent to you when the user dismisses your notification without choosing one of your actions. This isn't sent by default, but when we registered our “alarm” category we added the **.customDismissAction** option – that's what asks iOS to tell us when the user dismisses the notification.

I said that the **userInfo** dictionary is buried deep, and it is: we're given a **UNNotificationResponse**, which contains a **UNNotification** property, which contains a **UNNotificationRequest** property, which contains a **UNNotificationContent** property, which is what contains the **userInfo** dictionary. It will always be there, but it might not contain the value we expect so it's worth unwrapping it safely.

One last thing: if the “rename” action is triggered, then **didReceive()** will get called with a specific type of notification response called **UNTextInputNotificationResponse**. This is a subclass of a regular notification response, and provides a **userText** property that tells you what text the user entered.

That's it, time for the code. Add this method to **ViewController** now:

```
func userNotificationCenter(_ center: UNUserNotificationCenter,
didReceive response: UNNotificationResponse, withCompletionHandler
completionHandler: () -> Void) {
    // pull out the buried userInfo dictionary
    let userInfo = response.notification.request.content.userInfo

    if let groupID = userInfo["group"] as? String {
        // if we got a group ID, we're good to go!
        switch response.actionIdentifier {
            // the user swiped to unlock; do nothing
        }
    }
}
```

```

case UNNotificationDefaultActionIdentifier:
    print("Default identifier")

    // the user dismissed the alert; do nothing
case UNNotificationDismissActionIdentifier:
    print("Dismiss identifier")

    // the user asked to see the group, so call display()
case "show":
    display(group: groupID)
    break

    // the user asked to destroy the group, so call destroy()
case "destroy":
    destroy(group: groupID)
    break

    // the user asked to rename the group, so safely unwrap their
text response and call rename()
case "rename":
    if let textResponse = response as?
UNTextInputNotificationResponse {
        rename(group: groupID, newName:
textResponse.userText)
    }

    break

default:
    break
}

```

```
    }  
  
    // you need to call the completion handler when you're done  
    completionHandler()  
}
```

At last – at long, long last – the code is finished and the project is complete. Well done!



The finished product shows options directly below the attached image, and the user can type text directly in.

Wrap up

This has been a huge project and we've covered a lot of ground, but I hope you can see all the project code was needed in order to fully exercise the iOS 10 notifications. Being able to add pictures meant we could show rich media in our notifications; having cells with switches in meant we could enable or disable alert sounds; having names attached to groups and alarms meant users could rename them in alerts; attaching IDs to groups and alarms meant we could use that as context for notifications; being able to perform segues from various places became useful with the "show" notification; and much more.

So, it was a complex project, but it was also one carefully crafted to give you a reasonable, real-world test bed for notifications. And of course you may have learned a few other things along the way too, or at the very least had some practice.

Homework

- Fun: Create a second **UNNotificationCategory** for alarms that contain pictures, then give it an action that is "Remove picture".
- Tricky: Try creating a few groups, scheduling a notification then terminating the app. When the alert appears, use Delete Group. If you go back into the app, all your groups will be gone. Try to figure out why, then craft a fix. I've written a hint below if you get stuck.
- Taxing: Remove the calls to **reloadData()** in our view controllers, and replace it with something more specific.
- Nightmare: Every time any alarm changes, all enabled alarms are cancelled then rescheduled. If the app has lots of images, copying lots of images for no reason isn't useful. Remove the call to **removeAllPendingNotificationRequests()** in **createNotifications()**, and instead only reschedule alerts that have changed.

Project 6

Flip

Setting up

We've had some massive projects so far, which is both good and bad. Good, because I hope it means you've learned a lot, built some useful, practical apps you can adapt to your own needs, and also got value for money from the book. But also bad, because it can be quite hard going if you're always working on tough projects.

So, it's with great pleasure I want to introduce you to Flip, which is the first of two game projects in this book. It's not a complicated project, but it does give us some interesting problems to solve and introduces a great new iOS 10 feature: the Monte Carlo strategist in GameplayKit. This lets us create a high-performance AI opponent for our game with very little work using an approach inspired by randomness: rather than deeply evaluate every move, it tries out a few moves then focuses its attention only on the moves that look most promising.

Flip is a clone of Reversi, sometimes known under its trademarked name Othello. If you've never played it before, the rules aren't terribly hard:

- There's an eight by eight board of unmarked squares.
- There are two players, black and white, who take turns placing a piece of their color. Black always goes first.
- Both players start with two pieces on the game board in a particular formation.
- You can capture an opponent's pieces by sandwich them between two of your own. For example, if you have one black piece then one white piece like this **[b][w]** then placing a black piece after the while one will capture it: **[b][w][b]**.
- When pieces are capture they become your color.
- You can capture multiple pieces, e.g. placing the final black piece in **[b][w][w][w][b]** will become **[b][b][b][b][b]**.
- You can capture in any direction – horizontally, vertically, or diagonally – and you can capture in multiple directions at the same time.
- The only legal moves are those which capture pieces. If your move won't capture anything, you can't make it.
- If you can't play, you miss your turn. If your opponent also can't play, the game ends and the winner is the player with the most pieces of their color.
- The "pieces" are conventionally called stones.

I'll be explaining the rules more as we progress, and of course it will all become clear when you start playing.

For now, though, please launch Xcode and create a new Game project. Name it Flip, choose

SpriteKit for game technology, and iPad for the device target. Make sure “Integrate GameplayKit” is *not* checked.

Warning: If possible I recommend running this game on a real iOS 10 device. The iOS Simulator does *not* perform well with SpriteKit, and you can expect almost all animations to run slowly there.

Building the basic game

Even without including GameplayKit, the Xcode SpriteKit template gives us quite a lot of junk we don't need. Open GameScene.swift and give it this content:

```
import SpriteKit
import GameplayKit

class GameScene: SKScene {
    override func didMove(to view: SKView) {
    }

    override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
    }
}
```

That removes almost everything except stubs for **didMove()** and **touchesBegan()**. In GameViewController.swift, please find this line:

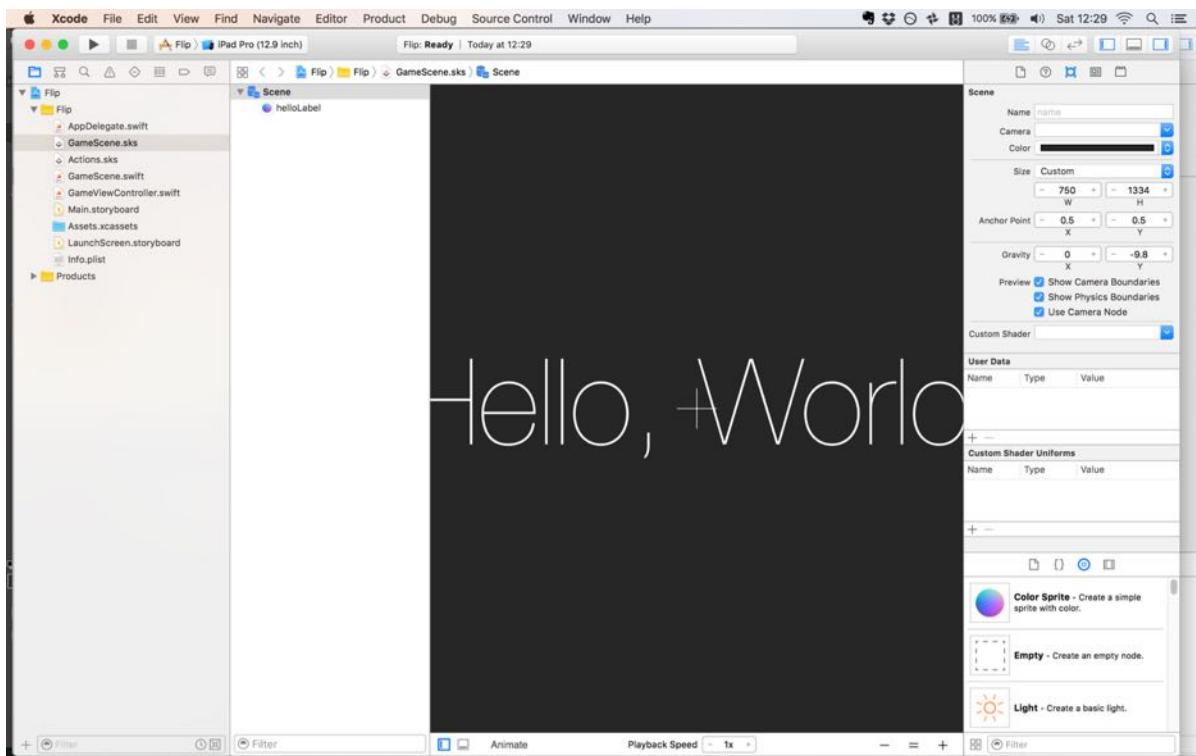
```
scene.scaleMode = .aspectFill
```

And replace it with this:

```
scene.scaleMode = .resizeFill
```

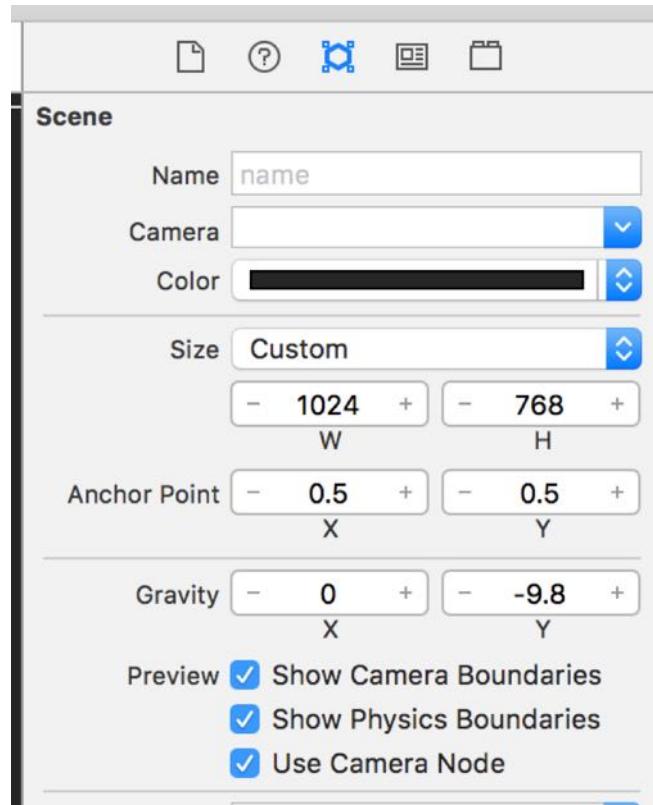
That will ensure our game draws correctly in landscape and portrait.

Open GameScene.sks and delete the massive “Hello, World” label that you’ll find there.



Apple's template includes the massive text Hello World, which is clearly never going to feature in any actual game

Now select the Scene itself, go to the Attributes inspector, and change its width to be 1024 and height to be 768.



Setting a custom size is the easiest way to make our scene the correct size for landscape iPads.

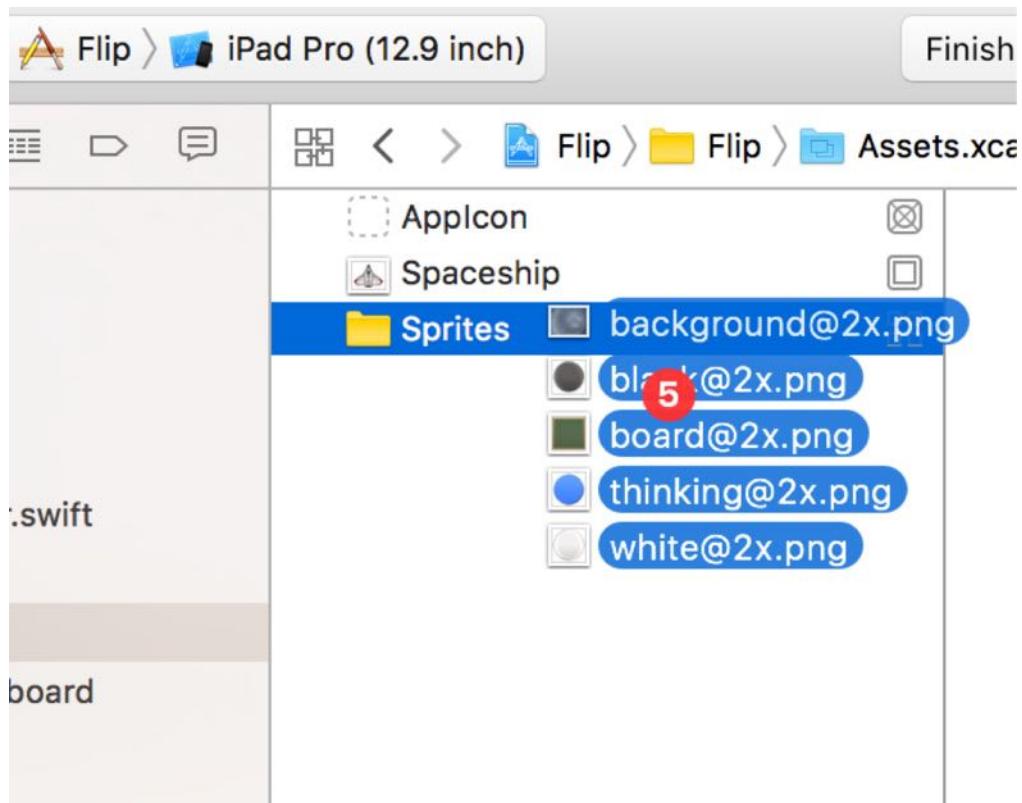
As for Actions.sks, you can just go ahead and delete that entirely.

By default, Assets.xcassets will contain the usual spaceship image, but we're going to replace that with the artwork for our game. Along with this book you'll see a zip file called Project6-Assets, which includes five assets:

- background@2x.png: a scratched metallic background that will fill the screen.
- black@2x.png: a stone that will represent the black player's stones.
- board@2x.png: the game board we'll be playing on. This fits great in portrait, landscape, and landscape with split view.
- thinking@2x.png: a virtual stone that will be used to represent where the AI intends to move.
- white@2x.png: a stone that will represent the white player's stones.

We're going to add those five into a sprite atlas inside the asset catalogue. To do that, right-click in the space below the spaceship image and choose New Sprite Atlas. Now for the

important bit: select all five images in Finder, then drag them onto the yellow folder marked “Sprites”. Doing so will import them all individually, ready to use.



Drag all five pictures into the Sprites folder, and Xcode will try to create a spritesheet from them all.

And remember: what do we say to the spaceship sprite? Not today. So, Before you leave the asset catalogue, delete the spaceship because it's not needed.

Adding the board and stones

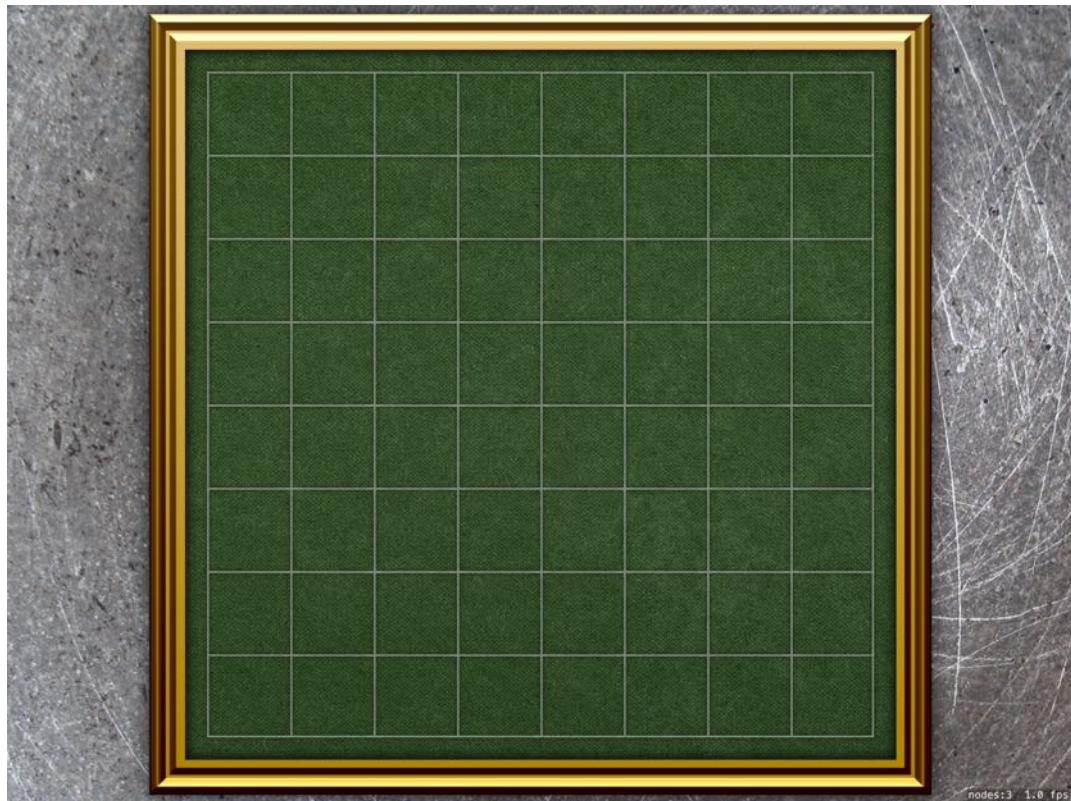
Now that we have a clean project ready to build on, the first thing we're going to do is add the background, game board, and stones in SpriteKit. This isn't quite as simple as just adding lots of **SKSpriteNode** objects to the screen because we need to position them carefully and also make sure the player's taps can be translated into moves easily.

First, the easy bits: we need to create the background and board pictures, and add them to the scene. Add this to the **didMove()** method:

```
let background = SKSpriteNode(imageNamed: "background")
background.blendMode = .replace
background.zPosition = 1
addChild(background)
```

```
let gameBoard = SKSpriteNode(imageNamed: "board")
gameBoard.name = "board"
gameBoard.zPosition = 2
addChild(gameBoard)
```

As usual, **blendMode** is set to **.replace** for the background image because it doesn't need alpha blending, and we're using the **zPosition** property to stack up the nodes correctly. I've given the game board a name because we'll be using that later for tap detection.



The result so far places the metal background on the screen, then adds the board on top.

Before we're able to add stones, we need to create a couple of new things. First, I'd like you to create a **StoneColor** enum that contains four values: black, white, empty, or choice. The first two will be used to mark player control; the third means no player owns it yet; and the fourth will be used to mark where the AI intends to go, which helps the user follow what's going on.

You can add this enum to any file, or alternatively create a new Swift file called StoneColor.swift. Either way, add this content:

```
enum StoneColor: Int {  
    case empty, choice, white, black  
}
```

Next we need to create a class to represent stones in the game. This is going to be a subclass of **SKSpriteNode** that has some additions:

1. Three static **SKTexture** objects, one for the black stone, one for the white stone, and one for the thinking virtual stone. This means we can change a stone's image just by switching the texture.
2. A **setPlayer()** method that accepts a player color and updates the texture appropriately.
3. A property for **row** and **col** that identifies where the stone is on the board. This will be used for tap detection.

So, go to File > New > File, then choose iOS > Source > Cocoa Class. Make it a subclass of **SKSpriteNode**, then name it Stone. By default it will import only UIKit, so please start by adding this:

```
import SpriteKit
```

Now give it this code:

```

class Stone: SKSpriteNode {

    // 1: create shared properties for the three possible textures
    // stones can have

    static let thinkingTexture = SKTexture(imageNamed: "thinking")
    static let whiteTexture = SKTexture(imageNamed: "white")
    static let blackTexture = SKTexture(imageNamed: "black")

    // 2: a method that will set the stone's texture to match the
    // player's color

    func setPlayer(_ player: StoneColor) {
        if player == .white {
            texture = Stone.whiteTexture
        } else if player == .black {
            texture = Stone.blackTexture
        } else if player == .choice {
            texture = Stone.thinkingTexture
        }
    }

    // 3: the row and column this stone belongs to
    var row = 0
    var col = 0
}

```

With the **StoneColor** enum and **Stone** class in place, the next step is to fill the board with stones. To make tap detection easy, we're going to create each stone when the game starts, giving each of them a clear texture so they are just spaces on the board. We'll then add each stone to an array of arrays so we can keep track of rows and columns more easily.

So, start by adding this property to **GameScene**:

```
var rows = [[Stone]]()
```

That's an array of arrays of **Stone** objects. Each stone will belong to an array that holds all the stones in a given row, and the **rows** array will hold all those arrays. This means we can reference a particular stone using syntax like **rows[4][3]** – that means five rows up (counting from the bottom) and four columns in (counting from the left).

When it comes to positioning, each stone is going to be 80x80 points, so for an 8x8 board that means the total size will be 640x640 points. SpriteKit positions everything from the center of its parent, so to find the position for the bottom-left stone we need to take 320 (half of 640) and subtract 40 (half of 80) to make 280. That means we need to start positioning at X: -280 and Y: -280 then work outwards from there. Because the stone graphics have a slight shadow on them, we'll be using -281 for the initial Y position to pull them down by just one point.

So, in total here's what the code needs to do:

1. Set up some constants for the initial X and Y offsets, as well as the stone size.
2. Count from 0 to 7, once for each row, creating a new array of **Stone** objects each time to store that row.
3. Count from 0 to 7, one for each column, creating a new **Stone** with a clear color at the correct stone size.
4. Position the new stone by using the initial X and Y offset, plus a multiple of stone size.
5. Tell the stone what row and column it has – we'll be using that later on for tap detection.
6. Add each stone to the game board and also to its column array.
7. Add each column array to the **rows** array.

The code for all that is below, with numbered comments matching the above. Put this into the **didMove()** method, below the existing code:

```
// 1: set up the constants for positioning
let offsetX = -280
let offsetY = -281
let stoneSize = 80
```

```

for row in 0 ..< 8 {
    // 2: count from 0 to 7, creating a new array of stones
    var colArray = [Stone]()

    for col in 0 ..< 8 {
        // 3: create from 0 to 7, creating a new stone object
        let stone = Stone(color: UIColor.clear(), size: CGSize(width:
stoneSize, height: stoneSize))

        // 4: place the stone at the correct position
        stone.position = CGPoint(x: offsetX + (col * stoneSize), y:
offsetY + (row * stoneSize))

        // 5: tell the stone its row and column
        stone.row = row
        stone.col = col

        // 6: add each stone to the game board and the column array
        gameBoard.addChild(stone)
        colArray.append(stone)
    }

    // 7: add each column to the rows array
    rows.append(colArray)
}

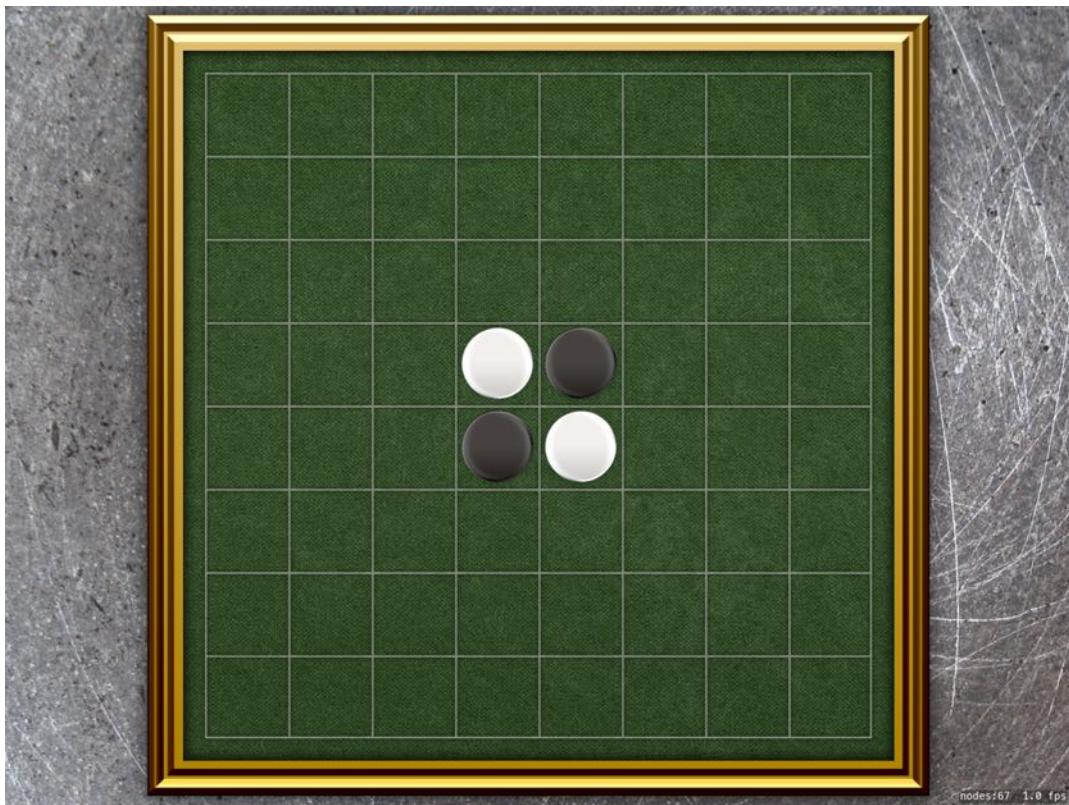
```

If you run that code you'll see it looks identical to what we had before, because all we're doing is creating lots of transparent squares.

However, we gave the **Stone** class a **setPlayer()** method, which means we can make it all come to life by adding just four more lines at the end of the **didMove()** method:

```
rows[4][3].setPlayer(.white)
rows[4][4].setPlayer(.black)
rows[3][4].setPlayer(.white)
rows[3][3].setPlayer(.black)
```

Run the game *now* and you'll see the correct starting position for the game of Reversi: two white stones and two black, placed diagonally.



This is the correct starting position for a game of Reversi: two whites and two black placed diagonally. Remember, black goes first!

Creating a model of our game

So far we've made our game *look* correct, but it's just a visual representation of the state of play. Underneath we need to keep track of the actual state of the board: which stones belong to each player, who the players actually are, and so on. We're going to put that into a new class called **Board**, which will also contain the logic required to check and apply moves that can then be replicated at the view layer – i.e., in SpriteKit.

So, go to File > New > File, then choose iOS > Source > Cocoa Touch Class. Name it "Board" and make it a subclass of **NSObject**. Give the new class the following two properties so that it knows how big the board is and can store who owns which stone:

```
static let size = 8  
var rows = [[StoneColor]]()
```

That's enough of the **Board** class for a few moments, so please go back to GameScene.swift. We need to create an instance of the **Board** class and store it as a property in **GameScene**, add lots of empty rows to it, then configure it to have the same starting position that you can see in SpriteKit.

First, add this property to **GameScene**:

```
var board: Board!
```

Now create that property in the **didMove()** method, just before the **let offsetX** line:

```
board = Board()
```

The next step is to add empty rows to the board model. We're already using **rows.append(colArray)** to add columns to the **rows** array, so we just need something to add to the **board.rows** array. Helpfully, Swift arrays have a dedicated initializer for creating arrays of repeated elements, so we can get what we want by adding just one line of code above the

existing `rows.append()` call in `didMove()`:

```
board.rows.append([StoneColor] (repeating: .empty, count: board.size))
```

That means “make a new array by repeating the value `.empty` enough times to fill the board,” which then gets appended to the `board.rows` array.

The last thing to do is set the model to have the same initial values as we gave the SpriteKit board. Add these four lines just before the end of `didMove()`:

```
board.rows[4][3] = .white  
board.rows[4][4] = .black  
board.rows[3][4] = .white  
board.rows[3][3] = .black
```

Now that we have a board in place, which has its own `size` property marking its size, you can change both the `for` loops in `didMove()` so that they use `Board.size` rather than the hard-coded value of 8:

```
for row in 0 ..< Board.size {  
    var colArray = [Stone]()  
  
    for col in 0 ..< Board.size {
```

The end result is the same, but it’s nice to avoid hard-coded values if possible!

Adding players and moves

We need to add two more model classes in order for the game to work. The `Player` class will

store the stone color for each player, an array of both players, and also a computed property to read the player's opponent.

Create a new **NSObject** subclass called “Player”, then give it the following code:

```
class Player: NSObject {

    // create two players, black and white, and store them in a
    static let allPlayers = [Player(stone: .black),
    Player(stone: .white)]

    // a property to store this player's color
    var stoneColor: StoneColor

    init(stone: StoneColor) {
        stoneColor = stone
    }

    // computed property to return the player's opponent
    var opponent: Player {
        if stoneColor == .black {
            return Player.allPlayers[1]
        } else {
            return Player.allPlayers[0]
        }
    }
}
```

Having the **opponent** computed property makes our code much easier later on, so it's worth adding.

The other other model object we're going to make is called **Move**, and will the row and column of a move. While it's possible to hijack something like **CGPoint** for this purpose, it makes our life harder in the long term because **CGPoint** stores floating-point numbers and we want integers.

So, create another **NSObject** subclass called "Move", and give it this code:

```
class Move: NSObject {
    var row: Int
    var col: Int

    init(row: Int, col: Int) {
        self.row = row
        self.col = col
    }
}
```

That's two properties, plus an initializer to set them. You might be wondering why we're not using structs for this and **Board**, but there's a simple answer: GameplayKit requires objects rather than structs.

Before we're done, there's just one more thing to add: the **Board** model needs to know which player is currently in control. This is used to figure out which moves are legal, because what is legal for one player may be illegal for the other.

So, add this property to the **Board** class:

```
var currentPlayer = Player.allPlayers[0]
```

Detecting legal moves

The first thing we're going to work towards is making a game that two humans players can use, so we need to write some logic to ensure the rules are followed. To that end, we're going to add a method to **Board** called **canMoveIn()**, which will run three checks and return if the move is possible.

The first test is whether the move is sensible, meaning that we don't want players to add a stone to row -1 or column 691. The second test is whether the move is new, meaning that we don't want players to add a stone to a position that already has a stone. The final test is whether the move is legal, which is the most complicated: a player can only make a move if it captures at least one stone, which means ensuring that one of their own stones is on the other side to make a stone sandwich, as it were.

The basic **canMoveIn()** method looks like this – please add the code below to **Board**:

```
func canMoveIn(row: Int, col: Int) -> Bool {  
    // test 1: check move is sensible  
  
    // test 2: check move hasn't been made already  
  
    // test 3: check the move is legal  
}
```

We're going to fill them in individually, starting with the first test: is the move within the bounds of the board? Add this method to **Board**:

```
func isInBounds(row: Int, col: Int) -> Bool {  
    if row < 0 { return false }  
    if col < 0 { return false }  
    if row >= Board.size { return false }  
    if col >= Board.size { return false }  
  
    return true
```

```
}
```

That will return true only if the row and column passed in are within the bounds of the board: no less than 0, and no more than the size of the board.

With that in place, the first test is complete: add this code below the “test 1” comment:

```
if !isInBounds(row: row, col: col) { return false }
```

The second test is to ensure the move hasn’t been made already. If you remember, we filled the **board.rows** array with lots of **.empty** values when the game first started. As players make moves, we change that value to **.white** or **.black** as appropriate, which means the second test is done by checking the row and column in question to see whether it has the value **.empty**, like this:

```
let stone = rows[row][col]
if stone != .empty { return false }
```

Now for the third and final test, which is the most complicated by a long way. A move is considered legal if it captures at least one of the opponent’s stones in any direction, and to check *that* we need to try every possible direction to look for a sequence of at least one opponent stone followed by one of our ours.

There are eight directions we need to check in: up and left, straight up, up and right, straight left, straight right, down and left, straight down, and down and right. Each of those can be presented using one of the **Move** objects we created, so add this property to the **Board** class:

```
static let moves = [
    Move(row: -1, col: -1), Move(row: 0, col: -1), Move(row: 1, col: -1),
    Move(row: -1, col: 0), Move(row: 1, col: 0), Move(row: -1, col: 1),
    Move(row: 0, col: 1), Move(row: 1, col: 1)
```

]

To explain that, let's examine the first one: `Move(row: -1, col: -1)`. That represent a move down one row (`row: -1`), and to the left one column (`col: -1`). The others represent the other possible moves. Using that array, we can draw lines from every starting point on the board by pulling out a move and applying it repeatedly to a starting point. So, if the starting point were row 3 column 2, then it would be row 2 column 1 (down and left one), followed by row 1 column 0 (down and left another one), at which point we'd hit the end of the board.

Once we're moving in a direction, we need to check the stones we find there. If we found an opponent's stone, we'll set a boolean flag to true so that we know the move is at least partially legal. If we find one of the player's stones *and have already found at least one opponent stone* then we will consider the test passed and return true immediately. If we find anything else – an empty space, or one of our stones *before* we found an opponent's stone – then that direction isn't going to work. If we've tried all directions and none of them worked, the move is invalid.

I'm going to give you the code in just a moment, but first I want to walk through exactly how it works just in case the above explanation doesn't quite make sense:

1. We need to loop through all possible moves, i.e. up and left, straight up, and so on.
2. Each time we start a new direction, we're going to set a boolean called `passedOpponent` to false.
3. We're going to assign our starting row and column to two movement variables. These variables will be adjusted as we move in a direction.
4. We'll loop through from 0 up to the size of the board, looking for moves.
5. Each time around the loop, we'll add the `row` and `col` properties of the current direction to the movement variables. So, if the move is row -1 and col -1, we'll subtract one from our movement variables each time.
6. We need to use the `isInBounds()` method to ensure the new position is valid.
7. If the stone belongs to the player's opponent, we set `passedOpponent`.
8. If the stone belongs to us and `passedOpponent` is set to true, we'll return true from the `canMoveIn()` method.
9. If we found anything else, we'll break out of the loop for the current move direction.
10. Finally, if we reach the end of the method it means all moves failed, so we return false for `canMoveIn()`.

That's it. I've written the code below, with numbered comments. Please put this below the

“test 3” comment:

```
// 1. loop through each possible move
for move in Board.moves {

    // 2. create a variable to track whether we've passed at least
    one opponent stone

    var passedOpponent = false

    // 3: set movement variables containing our initial row and
    column

    var currentRow = row
    var currentCol = col

    // 4: count from here up to the edge of the board, applying our
    move each time

    for _ in 0 ..< Board.size {

        // 5: add the move to our movement variables
        currentRow += move.row
        currentCol += move.col

        // 6: if our new position is off the board, break out of the
        loop
        guard isInBounds(row: currentRow, col: currentCol) else
        { break }

        let stone = rows[currentRow][currentCol]

        if (stone == currentPlayer.opponent.stoneColor) {

            // 7: we found an enemy stone
            passedOpponent = true

        } else if stone == currentPlayer.stoneColor && passedOpponent
    }
```

```

        // 8: we found one of our stones after finding an enemy
stone

        return true
    } else {
        // 9: we found something else; bail out
        break
    }
}

// 10: if we're still here it means we failed
return false

```

With that check in place, we can write a little of the **touchesBegan()** method in the **GameScene** class. We just have an empty method right now, but we can start to add some code there to figure out which stone was tapped. Remember, we fill every square with a stone when the game starts, but by default they just use a clear color so they are effectively invisible.

The code we're going to write in **touchesBegan()** needs to do the following:

1. Unwrap the first touch that was passed in.
2. Find the game board in the scene, or exit if it can't be found for some reason.
3. Figure out where on the game board the touch landed.
4. Pull out an array of all **SKNode** objects at that touch point
5. Filter out all nodes that weren't **Stone** nodes.
6. If no **Stone** was tapped, bail out.
7. Pass that stone's **row** and **col** property to the **canMoveIn()** method we just wrote.
8. Print a message whether the move is legal or not.

Obviously we'll make step 8 do something more interesting soon, but for now let's just make sure the algorithm works!

Here's the new **touchesBegan()** method, with numbered comments:

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
    // 1: unwrap the first touch
    guard let touch = touches.first else { return }

    // 2: find the game board, or return if it somehow couldn't be
    // found
    guard let gameBoard = childNode(withName: "board") else
    { return }

    // 3: figure out where on the game board the touch landed.
    let location = touch.location(in: gameBoard)

    // 4: pull out an array of all nodes at that location
    let nodesAtPoint = nodes(at: location)

    // 5: filter out all nodes that aren't stones
    let tappedStones = nodesAtPoint.filter { $0 is Stone }

    // 6: if no stone was tapped, bail out
    guard tappedStones.count > 0 else { return }

    let tappedStone = tappedStones[0] as! Stone

    // 7: pass the tapped stone's row and column into our new
    canMoveIn() method

    if board.canMoveIn(row: tappedStone.row, col: tappedStone.col) {

        // 8: print a message if the move is legal
        print("Move is legal")

    } else {

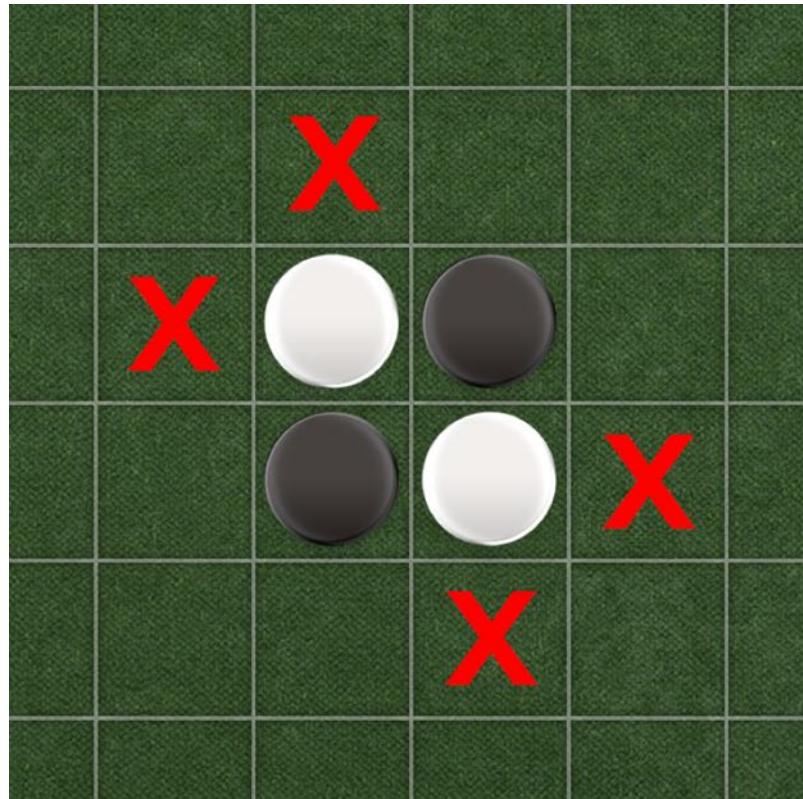
        print("Move is illegal")
    }
}
```

```
    }  
}  


---


```

Now that we've added the new `touchesBegan()` method, you can run the game now and try tapping squares to see whether the move is valid. Remember, black starts first, and in the screenshot below I've marked the moves you can make.



Now that the game has actual rules attached, it's black's turn to make a move. I've added red crosses to the moves you can make.

Capturing pieces

Now that we know which moves are legal and which aren't, we can proceed to the next step: placing a stone on the board, and capturing the opponent stones that it sandwiches. We're going to create a new method called `makeMove()` to do that, and it's about 50% the same as `canMoveIn()`. In fact, `makeMove()` should only be called if `canMoveIn()` has return true – if we get to `makeMove()` we're going to assume the move is valid.

Like I said, the `makeMove()` method is quite similar to `canMoveIn()`: because captures can take place in multiple directions, we try them all and see which ones work. The difference between the two methods is that `canMoveIn()` returns true as soon as a valid move is found, whereas `makeMove()` executes each move fully. Like `canMoveIn()`, the `makeMove()` method tries moving in each direction, but when it finds an opponent's stone it adds it to a "might capture" array. When it subsequently finds a friendly stone it captures all the stones in the "might capture" array, adds them to a "did capture" array, then breaks out of the loop because that direction has been exhausted. If no friendly stone is found, the array is cleared. Finally, it returns all the "did capture" stones so we can add some sort of animation in SpriteKit.

Yes, that's quite a lot, but keep your chin up: this is the last complicated method!

I'll put numbered comments in the code, but first here's a summary of what it all does:

1. Create an array for stones that have been captured.
2. Place a stone in the requested position and add it to the `didCapture` array.
3. Go through each direction and create movement variables from it, as well as a `mightCapture` array to handle possible captures in this direction.
4. Count from the start position to the edge of the board, applying the move each time.
5. If the new position is off the board, break out of the loop.
6. If we encounter an opponent's stone, add it to the `mightCapture` array as a possible stone for capturing.
7. If we encounter a friendly stone, add all the `mightCapture` stones to the `didCapture` list and change their owners.
8. Change the color of all stones in the `mightCapture` array, then exit the loop because this direction is complete.
9. If we find anything else – i.e., a gap on the board – we break out of the loop without flipping stones in `mightCapture`.
10. When all moves have been evaluated, `didCapture` will be full of all captured stones, so send it back.

So, the real changes are from step six onwards, but the concept is the same so **canMoveIn()**: go through all possible directions, draw a line from the start position along that direction, and see what can be captured.

Here's the code for **makeMove()** – put this into the **Board** class now:

```
func makeMove(player: Player, row: Int, col: Int) -> [Move] {
    // 1: create an array to hold all captured stones
    var didCapture = [Move]()

    // 2: place a stone in the requested position
    rows[row][col] = player.stoneColor
    didCapture.append(Move(row: row, col: col))

    for move in Board.moves {
        // 3: look in this direction for captured stones
        var mightCapture = [Move]()
        var currentRow = row
        var currentCol = col

        // 4: count from here up to the edge of the board, applying
        our move each time
        for _ in 0 ..< Board.size {
            currentRow += move.row
            currentCol += move.col

            // 5: make sure this is a sensible position to move to
            guard isOutOfBounds(row: currentRow, col: currentCol) else
            { break }

            let stone = rows[currentRow][currentCol]
            if stone == .empty || stone == .player(player) { break }

            if stone == .opponent(player) { mightCapture.append(Move(row: currentRow, col: currentCol)) }
        }
    }
}
```

```

        if stone == player.opponent.stoneColor {

            // 6: we found an enemy stone - add it to the list of
            possible captures

            mightCapture.append(Move(row: currentRow, col:
            currentCol))

        } else if stone == player.stoneColor {

            // 7: we found one of our stones - add the
            mightCapture array to didCapture

            didCapture.append(contentsOf: mightCapture)

            // 8: change all stones to the player's
            color, then exit the loop because we're finished in this direction

            mightCapture.forEach {

                rows[$0.row][$0.col] = player.stoneColor

            }

            break

        } else {

            // 9: we found something else; bail out

            break

        }

    }

}

// 10: send back the list of captured stones

return didCapture
}

```

Moving stones in SpriteKit

We've written the **makeMove()** method in the **Board** model, but it won't be reflected on the visible game board just yet. To do that, we need a new **makeMove()** method in the **GameScene** class that will call **makeMove()** on the board and use its return value – the list of captured stones – to update the user interface.

Fortunately, the board's **makeMove()** method does all the hard work – all the **makeMove()** method in **GameScene** needs to do is call **setPlayer()** on each captured stone to make it update its texture, then do a little animation.

Add this code to **GameScene** now:

```
func makeMove(row: Int, col: Int) {  
    // find the list of captured stones  
  
    let captured = board.makeMove(player: board.currentPlayer, row:  
        row, col: col)  
  
    for move in captured {  
        // pull out the sprite for each captured stone  
        let stone = rows[move.row] [move.col]  
  
        // update who owns it  
        stone.setPlayer(board.currentPlayer.stoneColor)  
  
        // make it 120% of its normal size  
        stone.xScale = 1.2  
        stone.yScale = 1.2  
  
        // animate it down to 100%  
        stone.run(SKAction.scale(to: 1, duration: 0.5))  
    }  
  
    // change players
```

```
    board.currentPlayer = board.currentPlayer.opponent  
}
```

There's just one last thing to do before the game becomes playable: inside the **touchesBegan()** method, replace the “move is legal” **print()** call with this:

```
makeMove(row: tappedStone.row, col: tappedStone.col)
```

OK, it's time to take a break: run the game now and you'll find it's completely playable – although I have to admit that playing as both black *and* white is confusing!

Capturing pieces

Now that we know which moves are legal and which aren't, we can proceed to the next step: placing a stone on the board, and capturing the opponent stones that it sandwiches. We're going to create a new method called `makeMove()` to do that, and it's about 50% the same as `canMoveIn()`. In fact, `makeMove()` should only be called if `canMoveIn()` has return true – if we get to `makeMove()` we're going to assume the move is valid.

Like I said, the `makeMove()` method is quite similar to `canMoveIn()`: because captures can take place in multiple directions, we try them all and see which ones work. The difference between the two methods is that `canMoveIn()` returns true as soon as a valid move is found, whereas `makeMove()` executes each move fully. Like `canMoveIn()`, the `makeMove()` method tries moving in each direction, but when it finds an opponent's stone it adds it to a "might capture" array. When it subsequently finds a friendly stone it captures all the stones in the "might capture" array, adds them to a "did capture" array, then breaks out of the loop because that direction has been exhausted. If no friendly stone is found, the array is cleared. Finally, it returns all the "did capture" stones so we can add some sort of animation in SpriteKit.

Yes, that's quite a lot, but keep your chin up: this is the last complicated method!

I'll put numbered comments in the code, but first here's a summary of what it all does:

1. Create an array for stones that have been captured.
2. Place a stone in the requested position and add it to the `didCapture` array.
3. Go through each direction and create movement variables from it, as well as a `mightCapture` array to handle possible captures in this direction.
4. Count from the start position to the edge of the board, applying the move each time.
5. If the new position is off the board, break out of the loop.
6. If we encounter an opponent's stone, add it to the `mightCapture` array as a possible stone for capturing.
7. If we encounter a friendly stone, add all the `mightCapture` stones to the `didCapture` list and change their owners.
8. Change the color of all stones in the `mightCapture` array, then exit the loop because this direction is complete.
9. If we find anything else – i.e., a gap on the board – we break out of the loop without flipping stones in `mightCapture`.
10. When all moves have been evaluated, `didCapture` will be full of all captured stones, so send it back.

So, the real changes are from step six onwards, but the concept is the same so **canMoveIn()**: go through all possible directions, draw a line from the start position along that direction, and see what can be captured.

Here's the code for **makeMove()** – put this into the **Board** class now:

```
func makeMove(player: Player, row: Int, col: Int) -> [Move] {
    // 1: create an array to hold all captured stones
    var didCapture = [Move]()

    // 2: place a stone in the requested position
    rows[row][col] = player.stoneColor
    didCapture.append(Move(row: row, col: col))

    for move in Board.moves {
        // 3: look in this direction for captured stones
        var mightCapture = [Move]()
        var currentRow = row
        var currentCol = col

        // 4: count from here up to the edge of the board, applying
        our move each time
        for _ in 0 ..< Board.size {
            currentRow += move.row
            currentCol += move.col

            // 5: make sure this is a sensible position to move to
            guard isOutOfBounds(row: currentRow, col: currentCol) else
            { break }
            let stone = rows[currentRow][currentCol]
            if stone == .empty {
                didCapture.append(Move(row: currentRow, col: currentCol))
            } else if stone == player {
                didCapture.append(Move(row: currentRow, col: currentCol))
                break
            } else if stone == .opposite {
                mightCapture.append(Move(row: currentRow, col: currentCol))
            }
        }
    }
}
```

```

        if stone == player.opponent.stoneColor {

            // 6: we found an enemy stone - add it to the list of
            possible captures

            mightCapture.append(Move(row: currentRow, col:
            currentCol))

        } else if stone == player.stoneColor {

            // 7: we found one of our stones - add the
            mightCapture array to didCapture

            didCapture.append(contentsOf: mightCapture)

            // 8: change all stones to the player's
            color, then exit the loop because we're finished in this direction

            mightCapture.forEach {

                rows[$0.row][$0.col] = player.stoneColor

            }

            break

        } else {

            // 9: we found something else; bail out

            break

        }

    }

}

// 10: send back the list of captured stones

return didCapture
}

```

Moving stones in SpriteKit

We've written the **makeMove()** method in the **Board** model, but it won't be reflected on the visible game board just yet. To do that, we need a new **makeMove()** method in the **GameScene** class that will call **makeMove()** on the board and use its return value – the list of captured stones – to update the user interface.

Fortunately, the board's **makeMove()** method does all the hard work – all the **makeMove()** method in **GameScene** needs to do is call **setPlayer()** on each captured stone to make it update its texture, then do a little animation.

Add this code to **GameScene** now:

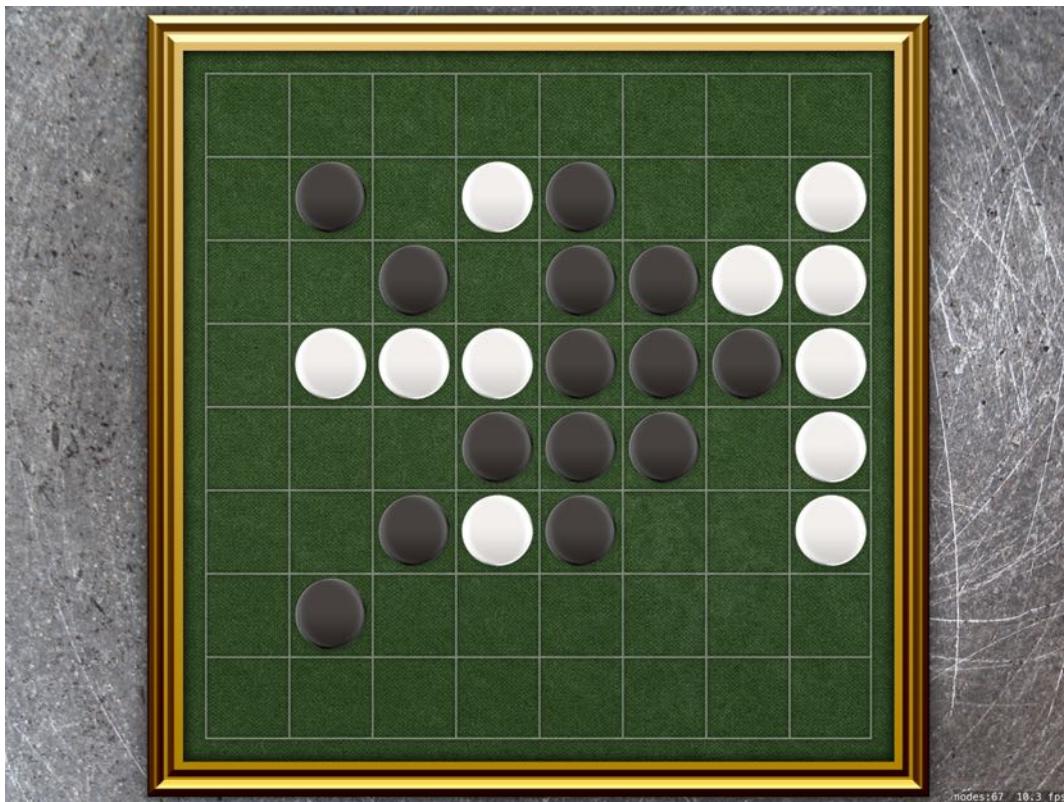
```
func makeMove(row: Int, col: Int) {  
    // find the list of captured stones  
  
    let captured = board.makeMove(player: board.currentPlayer, row:  
        row, col: col)  
  
    for move in captured {  
        // pull out the sprite for each captured stone  
        let stone = rows[move.row] [move.col]  
  
        // update who owns it  
        stone.setPlayer(board.currentPlayer.stoneColor)  
  
        // make it 120% of its normal size  
        stone.xScale = 1.2  
        stone.yScale = 1.2  
  
        // animate it down to 100%  
        stone.run(SKAction.scale(to: 1, duration: 0.5))  
    }  
  
    // change players
```

```
    board.currentPlayer = board.currentPlayer.opponent  
}
```

There's just one last thing to do before the game becomes playable: inside the **touchesBegan()** method, replace the "move is legal" **print()** call with this:

```
makeMove(row: tappedStone.row, col: tappedStone.col)
```

OK, it's time to take a break: run the game now and you'll find it's completely playable – although I have to admit that playing as both black *and* white is confusing!



The game is now playable, so if you're still not sure how the rules work here's your chance to try it yourself!

Monte Carlo strategy

When GameplayKit was introduced in iOS 9, it brought with it **GKMinMaxStrategist**: a tool that is able to act as an AI opponent in games by trying out all possible moves to a specific depth before selecting the best of them.

Minimax AI, as implemented in **GKMinMaxStrategist**, is guaranteed to find the best move available to the computer as long as a few rules are followed:

1. Players must take turns.
2. The game must be zero sum, i.e. if one player wins the other must lose.
3. There must be no randomness in the moves, i.e. no dice rolling or other chance.
4. You must be able to provide the computer with the complete situation, which means no hidden moves or actions.

What happens then is that the computer tries every possible move, followed by every possible subsequent move, followed by every subsequent subsequent move, and so on. When a specific lookahead depth is reached – when the computer has tried every move and every follow-on move from that up to a specific number of times – it evaluates the scores from every move combination and picks the best one.

Using the above technique, a minimax AI has literally evaluated every possible move combination up to a specific depth, so as long as that depth is big enough it will be able to play a perfect game. The problem is that it's expensive. If we imagine that at any point a player could make eight moves, then a look ahead depth of 8 means the minimax strategist needs to evaluate $8 * 8 * 8 * 8 * 8 * 8 * 8 * 8$ moves. That's 16,777,216 move combinations. Now, you might think that an eight-move lookahead is a lot, but remember it's a two-player game so that's only four moves per player. Plus, even in a "simple" game like Reversi, there are usually more than eight moves available, so in practicality a solid AI might need to evaluate 100 million positions or more in order to play solidly.

iOS 10 introduces an alternative strategist that is designed to operate significantly faster than minimax, but at the cost of having less than perfect results. It's called **GKMonteCarloStrategist** and its gambling-inspired name might give you a clue as to how it works: it uses random sampling to take an educated guess at which move is best, then explores the most promising paths in more detail.

To contrast Monte Carlo with minimax, consider this simplified situation:

- Move 1 looks like a terrible move, but it leads to move 2 that turns out to be a game-winning move.
- Move 3 looks like a great move, but it leads to move 4 that turns out to be a bad move.
- Move 5 looks like an OK move, and it leads to move 6 that also looks OK.

If you were using minimax, it would evaluate 1 and 2, 3 and 4, then 5 and 6, before deciding that move 1 was the best because it ultimate results in move 2. If you were using Monte Carlo, it might take one look at move 1's result and decide it wasn't worth further investigation. It would evaluate 3 and consider it worth investigating further, so it would investigate move 4. It might also investigate moves 5 and 6. It would then choose either 4 or 6, neither of which are the ideal move.

That's a simplified example – it's a bit smarter than *that* – but the concept is what matters: think of Monte Carlo strategists as a focused minimax search that explores only possibilities that look promising. This means it can potentially go to a greater search depth than minimax because it can ignore some paths in order to examine others more closely.

We're going to add a **GKMonteCarloStrategist** AI to our game, but you're welcome to try using **GKMinMaxStrategist** later on so you can see which you prefer.

In order to simulate moves, both minimax and Monte Carlo strategists work by producing lots of copies. They need to take a copy of the game board, apply a move, copy it again, apply another move, and so on. This is one of the reasons why we've kept our models separate from the SpriteKit views – it means the copies are quite light, and thus fast.

In order to make this technique work, we need to make a few changes to our model classes. Don't worry: the changes are moderately small and purely additive, which means the game will still work for two human players if you want that.

Modifying players

We already have a **Player** class that stores a color for each player, as well as a computed property to find that player's opponent. GameplayKit needs players to have one other property, which is an integer called **playerId**. It can have any value you like, as long as it's unique amongst all players.

So, open the **Player** class now and add this property:

```
var playerId: Int
```

That needs to be initialized somewhere to a unique value. The easiest thing to do is set it in **init()** to be the raw value of the player's stone color – that will automatically be a unique integer.

Add this line to the end of **init()** in the **Player** class:

That's all it takes to make the **Player** class ready for GameplayKit, so we can go ahead and tell Swift that we're compatible. Start by adding this import to the file:

```
import GameplayKit
```

Second, add the protocol **GKGameModelPlayer** to the class, like this:

```
class Player: NSObject, GKGameModelPlayer {
```

Done!

Modifying moves

It's even easier to make moves ready for GameplayKit. First, add the same import:

```
import GameplayKit
```

Second, make the **Move** class conform to **GKGameModelUpdate** like this:

```
class Move: NSObject, GKGameModelUpdate {
```

Finally, add a single new property to the class:

```
var value = 0
```

That gets used internally by GameplayKit to store how good it considers each move to be. This means GameplayKit will re-use the same **Move** class inside its own calculations, which helps keep our code cleaner.

Modifying the board

Our game board takes a little more work to adapt to GameplayKit, because all that copying is done using another protocol, **NSCopying**. To make it easier to understand, I'm going to try to break it down into smaller, simpler chunks that we can combine into full GameplayKit compatibility.

The first change is nice and easy, because it's just adding this import to Board.swift:

```
import GameplayKit
```

Second, we're going to add a **getScores()** method that counts the number of black and white stones and returns those counts as a tuple. There's no magic logic here, it just loops through every column in every row and adds to two variables. Add this method to the **Board** class:

```
func getScores() -> (black: Int, white: Int) {
    var black = 0
    var white = 0
```

```

rows.forEach {
    $0.forEach {
        if $0 == .black {
            black += 1
        } else if $0 == .white {
            white += 1
        }
    }
}

return (black, white)
}

```

As GameplayKit examines each move, it will ask us whether we consider it a win for a specific player. Now, most of the time the game won't actually be over so it's down to us to look at the scores and decide whether it looks good or bad. This is an area you can experiment with later on, but for now we're going to report a game as a win if the AI is at least 10 points ahead of the player.

When GameplayKit asks us whether a move is a win, it will pass in a **GKGameModelPlayer** instance. Behind the scenes we know that's really an instance of our **Player** class, but we still need to safely typecast it. We'll then call **getScores()** and compare black against white to decide whether we consider the move to be a winning one. Here's the code:

```

func isWin(for player: GKGameModelPlayer) -> Bool {
    guard let playerObject = player as? Player else { return false }
    let scores = getScores()

    if playerObject.stoneColor == .black {
        return scores.black > scores.white + 10
    }
}

```

```
    } else {
        return scores.white > scores.black + 10
    }
}
```

Next, we need to write a method to provide the strategist with every possible legal move for it to choose from. If there are no moves available – if the game is over – then it should return nil, but we'll also be returning nil from this method when `isWin()` reports true to tell it we think we've already found a good enough move to go with. Just as with `isWin()`, GameplayKit will pass us an object we need to safely typecast – this time from `GKGameModelPlayer` to **Player**.

Getting a list of all possible moves is just a matter of looping through every column in every row and calling `canMoveIn()` on it. If that returns true the move is possible and should be added to the list of moves to consider.

Add this method to the **Board** class now:

```
func gameModelUpdates(for player: GKGameModelPlayer) ->
[GKGameModelUpdate]? {
    // safely unwrap the player object
    guard let playerObject = player as? Player else { return nil }

    // if the game is over exit now
    if isWin(for: playerObject) || isWin(for: playerObject.opponent)
    {
        return nil
    }

    // if we're still here prepare to send back a list of moves
    var moves = [Move]()
}

// try every column in every row
```

```

        for row in 0 ..< Board.size {
            for col in 0 ..< Board.size {
                if canMoveIn(row: row, col: col) {
                    // this move is possible; add it to the list
                    moves.append(Move(row: row, col: col))
                }
            }
        }

        return moves
    }

```

Once we've given GameplayKit a list of possible moves, it will start trying them all. Of course, it doesn't care about the rules of Reversi, so instead it will pick one of the moves it wants to explore and ask us to make that move. That's trivial for us to do, because we already added a **makeMove()** method that applies a move to the game board, so all we need to do is add the same sort of typecast as before (we get a **Move** object masquerading as a **GKGameModelUpdate**) then switch players after each move.

Add this method to the **Board** class:

```

func apply(_ gameModelUpdate: GKGameModelUpdate) {
    guard let move = gameModelUpdate as? Move else { return }
    _ = makeMove(player: currentPlayer, row: move.row, col: move.col)
    currentPlayer = currentPlayer.opponent
}

```

There are just two more methods we need to implement, but before we do that I want to explain how GameplayKit works in a little more detail. We already have our **Board** class that represents the underlying state of the game. GameplayKit needs to be able to make changes to that to simulate moves happening, but of course we don't want those changes to happen

on the real game model otherwise we would lose our place. So, GameplayKit takes a copy of the game board and applies its move there, then it takes another copy and applies the subsequent move there, and so on.

Now, clearly making 16 million copies to evaluate every move is going to be extraordinarily expensive, so GameplayKit optimizes things: it will take *some* copies, but it will try to reuse copies as often as possible. So, we need to handle two cases: create a full copy of a **Board** object, and reuse an existing **Board** object. The second is really a subset of the first, so we're going to use one from inside the other.

First, we need to implement a method called **setGameModel()**. This will pass us a **Board** object masquerading as a **GKGameModel** object, so we need to start by safely typecasting it as a **Board**. Once that's done, GameplayKit wants us to set our current state to be the same as the board it passed us. So, we'll set our **currentPlayer** property to be the same as the other board's **currentPlayer** property, and our **rows** property to be the same as the other board's **rows** property. That's it; those are the only two properties we need to copy.

Add this method to **Board**:

```
func setGameModel(_ gameModel: GKGameModel) {
    guard let board = gameModel as? Board else { return }
    currentPlayer = board.currentPlayer
    rows = board.rows
}
```

Now we need to create the method GameplayKit will call when it needs to create a new **Board** copy. All this needs to do is create a new empty **Board** object, call **setGameModel()** on it, then return it. Here's the code:

```
func copy(with zone: NSZone? = nil) -> AnyObject {
    let copy = Board()
    copy.setGameModel(self)
    return copy
}
```

```
}
```

You'll get a small error with that code, but don't worry: we'll fix it in just a moment.

That's the last of the methods we need, but we also need to implement two properties. GameplayKit expects our board to have an array called **player** that holds an optional array of **GKGameModelPlayer** objects. It also expects to see a property called **activePlayer** that returns an optional **GKGameModelPlayer** object. We have our own values already, so we can just create computed properties to map one to the other. Add these to the **Board** class now:

```
var players: [GKGameModelPlayer]? {
    return Player.allPlayers
}

var activePlayer: GKGameModelPlayer? {
    return currentPlayer
}
```

That's all the requirements for GameplayKit, but the code won't compile until we tell Swift that the **Board** class conforms to the **GKGameModel** protocol. So, change the class definition to this:

```
class Board: NSObject, GKGameModel {
```

We've modified all our model classes for GameplayKit, so we're ready to proceed to the next step...

Making an AI player

At this point in the project we've given GameplayKit enough information to form a credible AI opponent to our player: it can get a list of all legal moves, try them out, and evaluate the effectiveness of the end result.

However, we still haven't added an AI opponent. To do that we need to create an instance of the Monte Carlo strategist, point it at our board, then ask it to make moves. Your GameScene.swift file should already have `import GameplayKit` at the top, but if not please add it now.

The first task is to declare and initialize an instance of `GKMonteCarloStrategist`. Start by adding this property in the `GameScene` class:

```
var strategist: GKMonteCarloStrategist!
```

We're going to initialize that object at the end of the `didMove()` method. It has four properties that are of interest:

- **budget**: how many total moves the strategist should evaluate. Higher values help it find better results, but also slow things down.
- **explorationParameter**: set to 4 by default, which means "focus as much on trying new things as focusing on moves that look good." We'll set this to 1, which means "when moves look good, follow them as far as you can."
- **randomSource**: the Monte Carlo algorithm includes a fair chunk of randomness, so it needs a way to generate random numbers.
- **gameModel**: the board it should look at to generate moves.

Add these five lines to the end of the `didMove()` method:

```
strategist = GKMonteCarloStrategist()
strategist.budget = 100
strategist.explorationParameter = 1
strategist.randomSource = GKRANDOMSource.sharedRandom()
strategist.gameModel = board
```

The main piece of work will be done in a new method called **makeAIMove()**. This is going to do several things:

1. Push all work to a background thread so that the AI doesn't cause the game to freeze.
2. Get the current time.
3. Calculate the best AI move.
4. Use the previous time reading to figure out how long the AI took to think.
5. Set the AI's tile choice to have the **.thinking** texture.
6. Wait for three seconds minus how long the AI took to think, so the AI will always appear to think for at least three seconds.
7. Then make the AI's move for real.

Here's the code, with numbered comments. Please add this method to the end of **GameScene**:

```
func makeAIMove() {  
    // 1: push all work to a background thread  
    DispatchQueue.global(attributes: .qosUserInitiated).async  
{ [unowned self] in  
    // 2: get the current time  
    let strategistTime = CFAbsoluteTimeGetCurrent()  
  
    // 3: calculate the best AI move  
    guard let move = self.strategist.bestMoveForActivePlayer()  
    as? Move else { return }  
  
    // 4: figure out how much time the AI spent thinking  
    let delta = CFAbsoluteTimeGetCurrent() - strategistTime  
  
    // 5: set the AI's chosen tile to the "thinking" texture  
    DispatchQueue.main.async { [unowned self] in  
        self.rows[move.row][move.col].setPlayer(.choice)  
    }  
}
```

```

    }

    // 6: wait for at least three seconds
    let aiTimeCeiling = 3.0
    let delay = min(aiTimeCeiling - delta, aiTimeCeiling)

    // 7: after at least three seconds have passed, make the move
    for real

        DispatchQueue.main.after(when: .now() + delay) { [unowned
self] in

            self.makeMove(row: move.row, col: move.col)
        }
    }
}

```

Using the **.thinking** state and inserting a three-second delay helps the user follow what the computer is up to; without it, the game can get hard to track because the computer plays so quickly!

There's only one more thing to do now, which is to hand control over to the AI when it's white's turn. In the **touchesBegan()** method of **GameScene**, add this code just after the call to **makeMove()**:

```

if board.currentPlayer.stoneColor == .white {

    makeAIMove()
}

```

That's it! You're done - go and play the game, and enjoy the fruits of your labor!

Wrap up

That's our first game project completed, and I hope you enjoyed the ride. Reversi isn't a complicated game, but it's complete enough to provide some interesting problems to solve: what are legal moves? How should capturing be calculated? What does a winning move look like for the computer?

To add a little extra spice, I snuck in a couple of functional calls here and there – if you're new to functional programming, you might like to read my Pro Swift book.

Homework

- Fun: When it's the AI's turn, the user can actually go ahead and make moves, which really screws things up – can you stop that from happening? Also, can you think of a way of showing legal moves to the user so they know where they can move?
- Tricky: When a player chooses an illegal move we use `print()` to output a message to Xcode, which is clearly not going to work in production. Can you make a more attractive user interface?
- Taxing: Right now the game doesn't end because there's no logic in place to see when the player can't move – can you add that? Once that's done, are you able to make the `isWin()` algorithm a bit smarter?
- Nightmare: I've shown you how to implement `GKMonteCarloStrategist`, but can you switch over to `GKMinMaxStrategist` and compare the two? This was covered in Hacking with Swift project 34.

Project 7

Dead Storm Rising

Setting up

In this project we're going to build a 2D war game called Dead Storm Rising – with apologies to Tom Clancy. To make the game's map we'll be drawing on the new tile map system that SpriteKit introduces in iOS 10, as well as other core features such as **SKCameraNode**, **SKEmitterNode**, and more.

The original code I wrote for the game ended up being over 1000 lines, which would have made for a book all by itself. After simplifying the code as much as possible, I ended up with over 1500 lines, which just goes to show that refactoring code to make it easier doesn't always make it shorter! Lots of code wrangling later, the finished project is now down to about 600 lines, which is long but still feasible for a single project. So, if you think "wow, this project is long," I hope you'll believe me when I say that this is the cut-down, simplified version!

I have provided lots of assets for this game, including several that didn't make the cut into the final tutorial. If you're enjoying the game and want to take it further, there's a big pile of suggestions in the homework!

For now, launch Xcode and create a new Game project named "DeadStormRising". Select SpriteKit for the game technology and iPad for the device, then click Next and Create.

We need to clean this project so it's in a usable state:

- Delete Actions.sks.
- Go into GameScene.sks, then delete the massive "Hello World" label. Give the scene a width of 1024 and a height of 768.
- In Assets.xcassets delete the Spaceship graphic.
- In GameScene.swift, delete everything except the stub for the **didMove()** and **touchesBegan()** methods. Remove the GameplayKit import too.

The final version of GameScene.swift should look like this:

```
import SpriteKit

class GameScene: SKScene {

    override func didMove(to view: SKView) {
```

```
    }

    override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
        ...

    }
}
```

The last thing to do is copy in the assets for this project. Open Assets.xcassets, then right-click below “AppIcon” and choose New Sprite Atlas. Download the Project7-Assets.zip file, then extract it. Inside you’ll find a “Sprites” folder – please drag all the files in there into that sprite atlas folder.

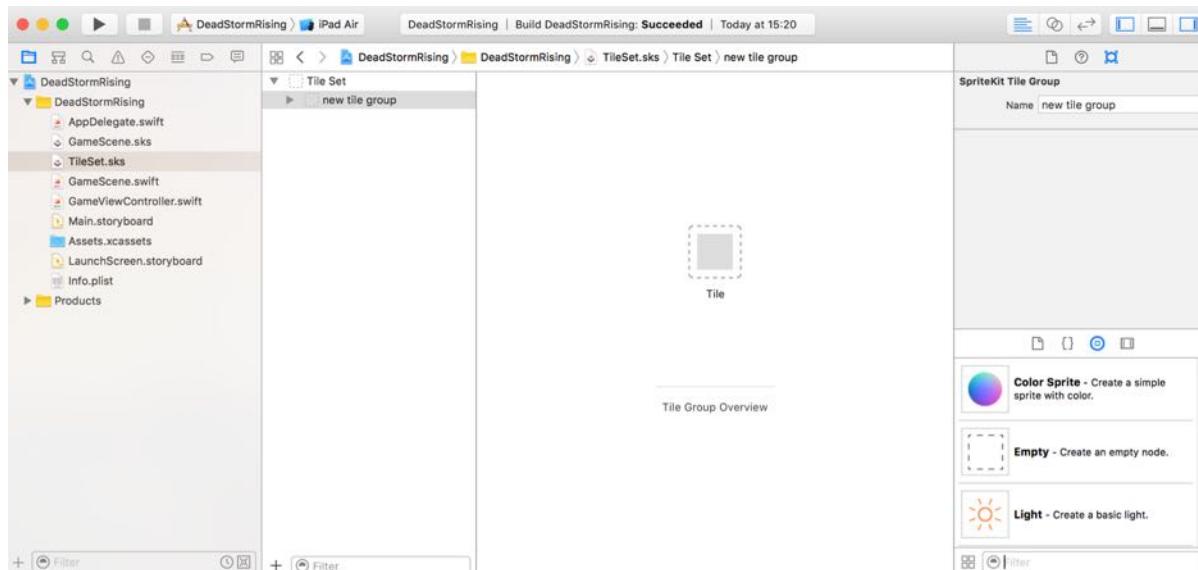
That’s it: the project is now cleaned and prepped ready for work to begin.

Note: The graphics for this project were taken from the Kenney Game Assets, and are released under the public domain. For more information, see <https://kenney.itch.io/kenney-donation>.

SpriteKit tile maps

Tile maps are a major new feature of SpriteKit in iOS 10, and enable extremely efficient drawing of large amounts of terrain. Rather than drawing huge maps by hand, tile maps are grids of smaller images that combine to form large and varied maps. SpriteKit automatically manages the tiles to keep their overhead as low as possible, and even builds a tile map editor directly into Xcode so you can literally paint your maps by clicking around.

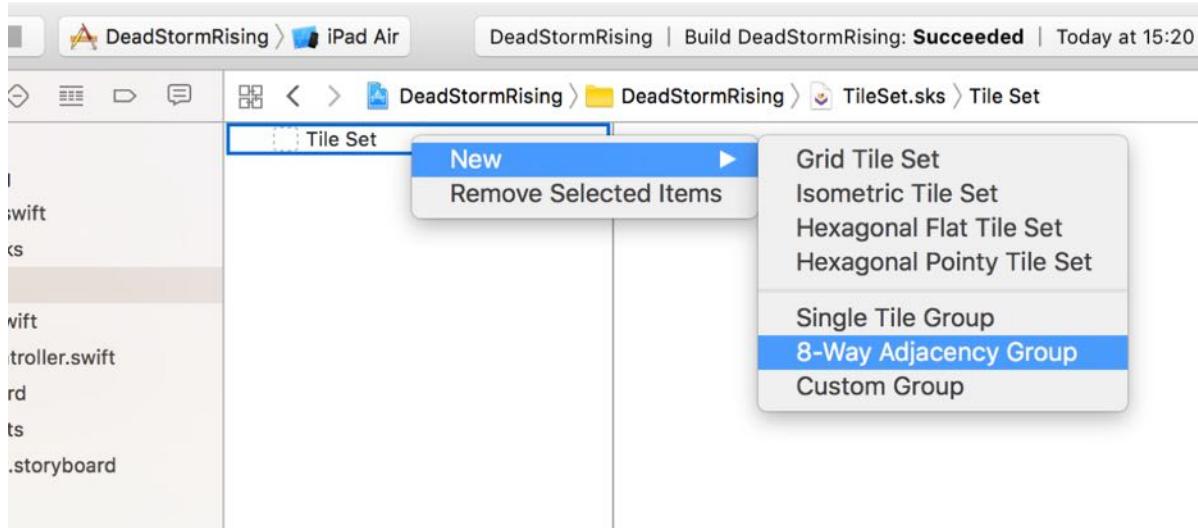
This is how we're going to start: we're going to create a tile map for the game then draw out a map. To start, go to File > New > File, then choose iOS > Source > SpriteKit Tile Set. Name it TileSet then click Create.



You'll get an empty tile set with an empty "new tile group" by default.

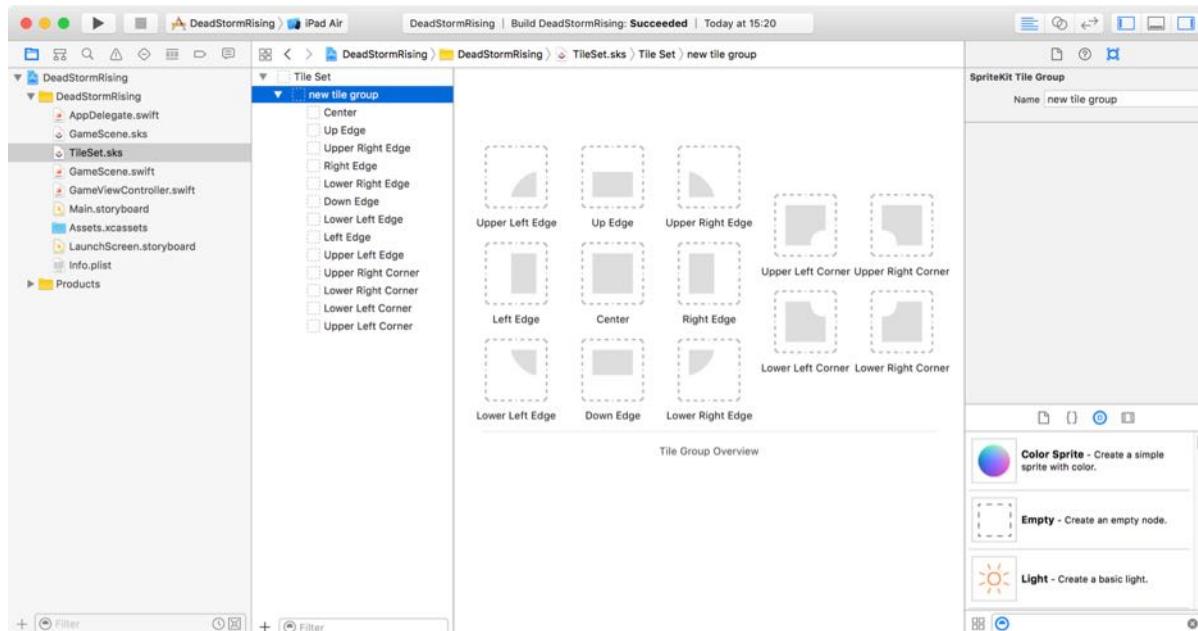
By default you'll get an empty tile set with an empty "new tile group" inside it. Please select "new tile group" then press delete because it's not much use.

Now select "Tile Set", then right-click on it and choose New > 8-Way Adjacency Group. If you only see four options, you might have tried to right-click on "Tile Set" without selecting it first.



You need to create a new 8-Way Adjacency Group, which lets you place specially shaped tiles into placeholder images.

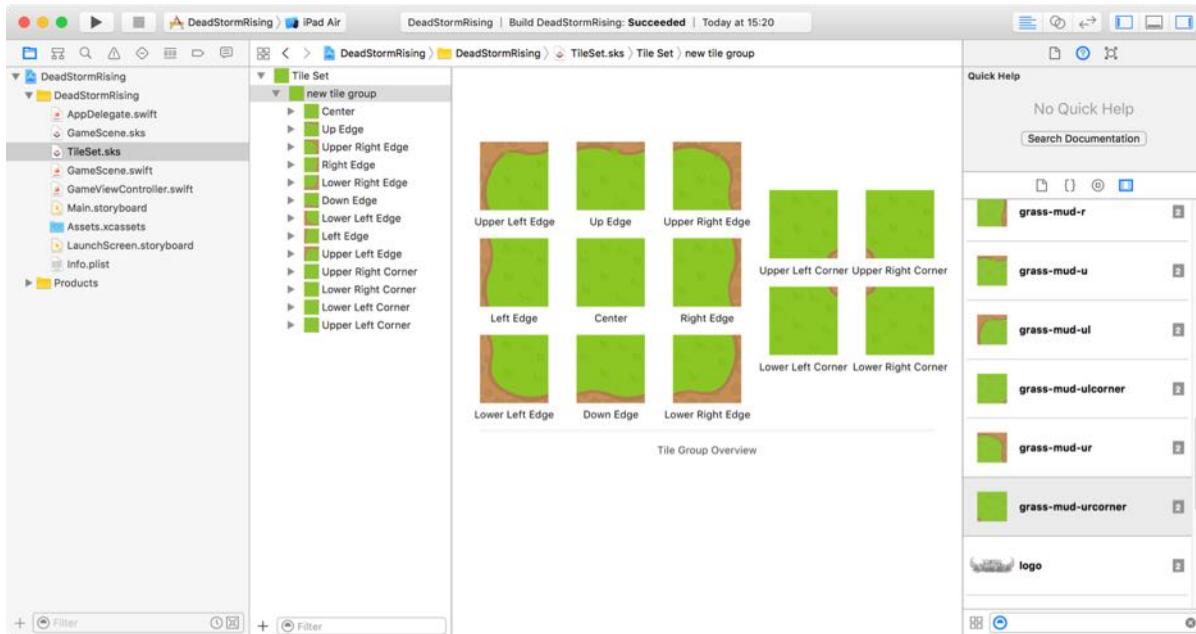
When you create a new adjacency group, you might find it's invisible by default because the disclosure arrow next to "Tile Set" is closed. If you open that, you should see another "new tile group" in there, and if you select *that* you'll see lots of boxes to fill: Upper Left Edge, Up Edge, Upper Right Edge, and so on.



This is what an 8-Way Adjacency Group looks like by default: lots of placeholder images to fill.

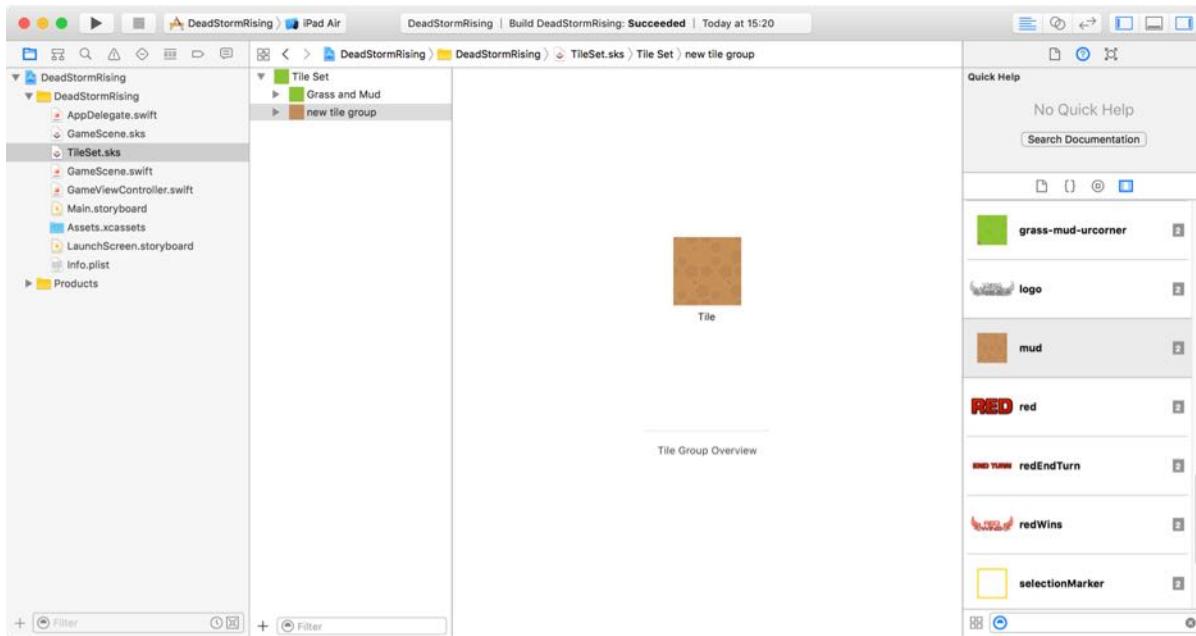
Your job is to fill each of those slots with the correct grass and mud tile graphics. I've tried to name them all usefully: "grass-mud-ul" should be placed in the upper left edge, for example. The easiest way to do that is to use the media library – press **Ctrl+Alt+Cmd+4** to activate it in the bottom-right corner of Xcode, then drag images across into the slots.

When you're done, all 13 slots should be filled with different images.



Once the adjacency group is filled you're all set to start using this tile group in the Xcode scene editor.

Name that adjacency group "Grass and Mud", then right-click on the "Tile Set" group and choose New > Single Tile Group. That will create one image box for you to fill, and I'd like you to fill that with the "mud" picture from the media library.

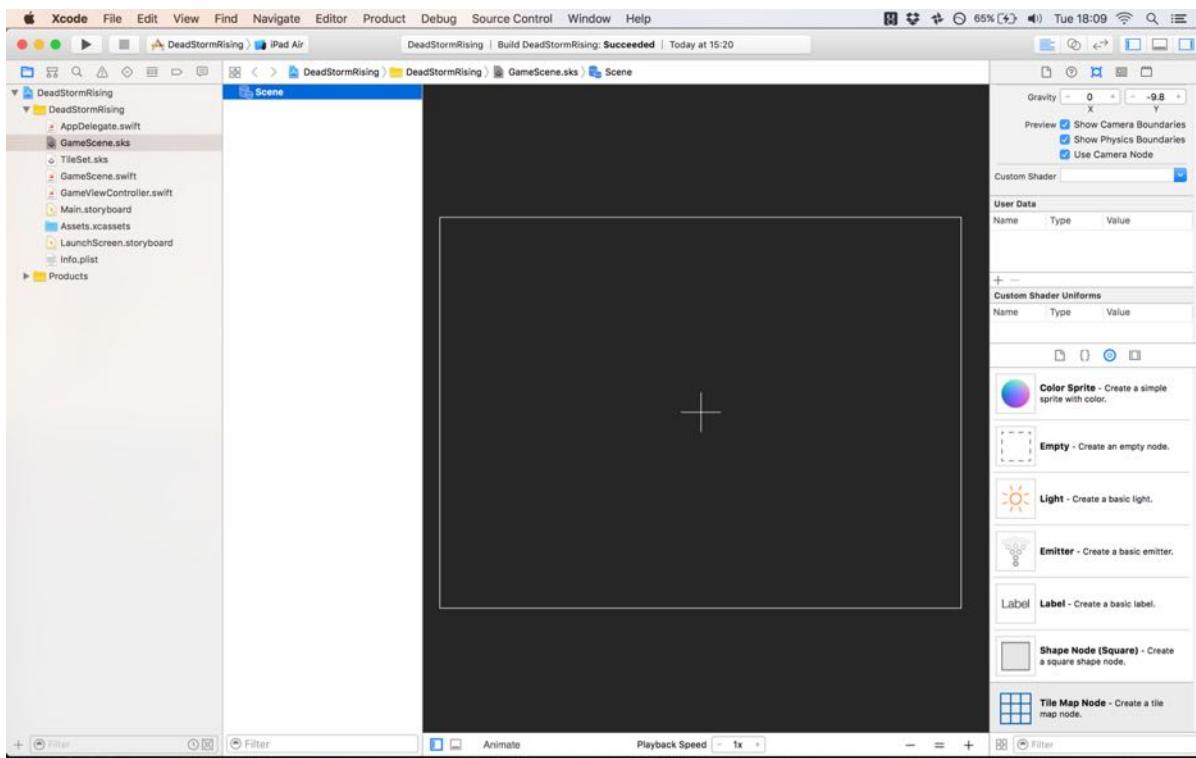


Add a second tile to the set, this one just containing a mud tile.

Warning: Before you leave TileSet.sks, press Cmd+S to save your work. At this point in the beta, Xcode really likes losing tile set data.

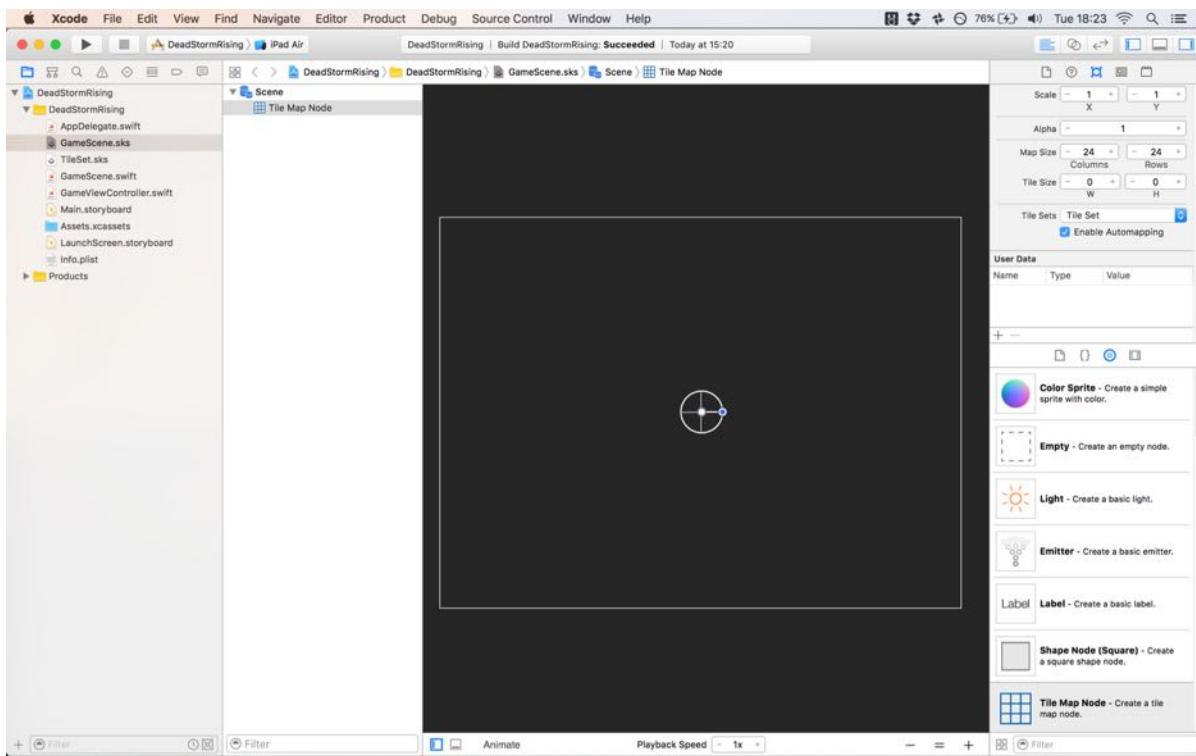
Drawing a map

Open GameScene.sks, which will right now just contain a large, black space. At the bottom of the scene editor you'll see Playback Speed and to the right some zoom buttons. Please click the - button a few times, at least until you can see the full size of the game scene as a rectangle.



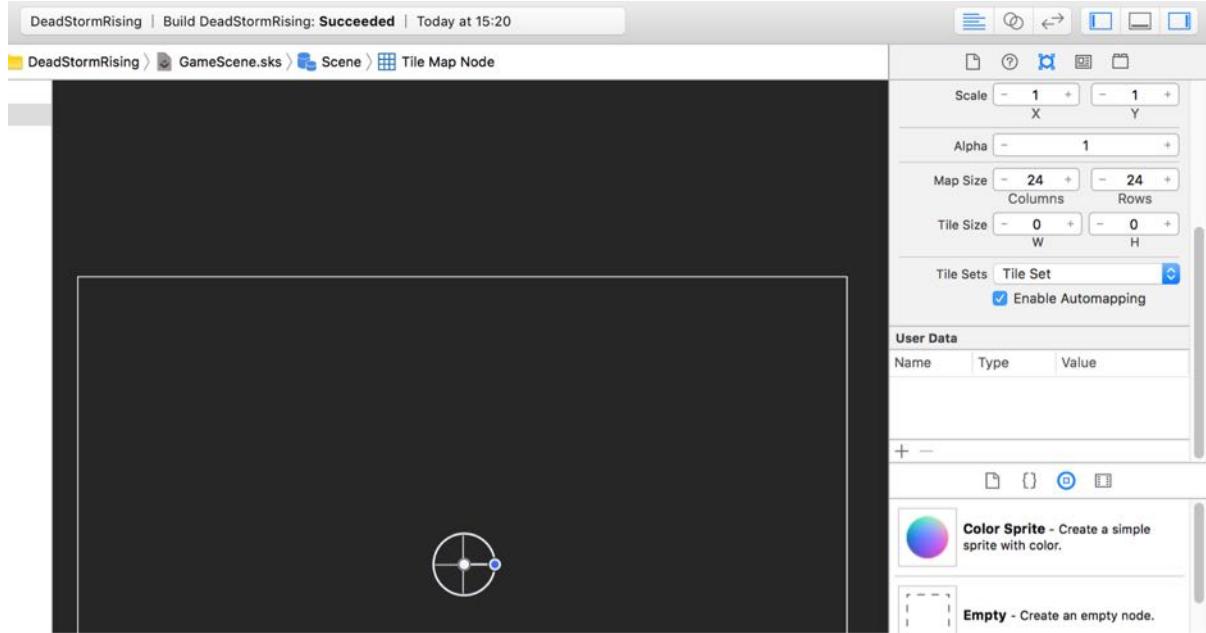
Zoom all the way out until you can see your game scene represented as a rectangle.

Switch to the object library (Ctrl+Alt+Cmd+3), then drag a Tile Map Node out onto the canvas. You'll probably see a small circle appear in the scene editor, which represents the tile map node.



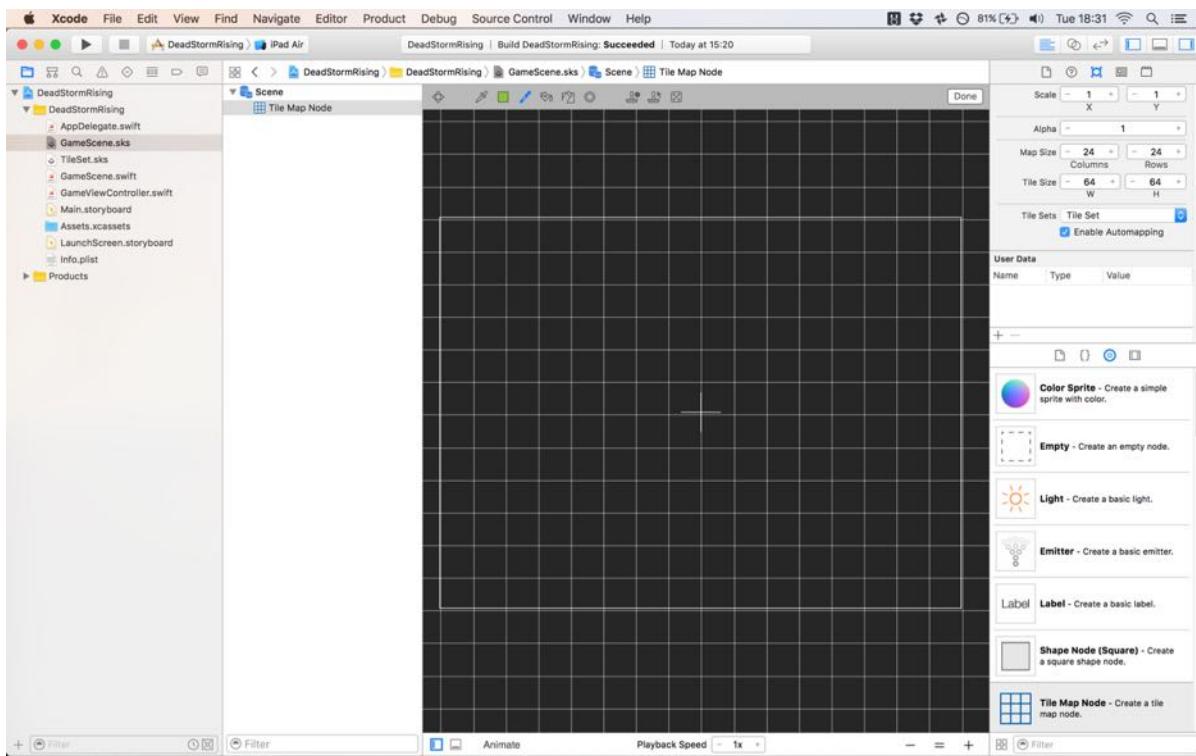
When you add a tile map node to your scene, you might just see a tiny circle in the middle. It's a bit glitchy right now!

Select the tile map node using the document outline, then go to the attributes inspector. It's possible you find it has Tile Size set to 0 width and 0 height – please change that to 64 and 64, because our tiles are 64x64 points in size.



Xcode has a tendency of creating tile map nodes with a tile width and height of 0. If you see this screenshot, change the numbers to 64 and 64!

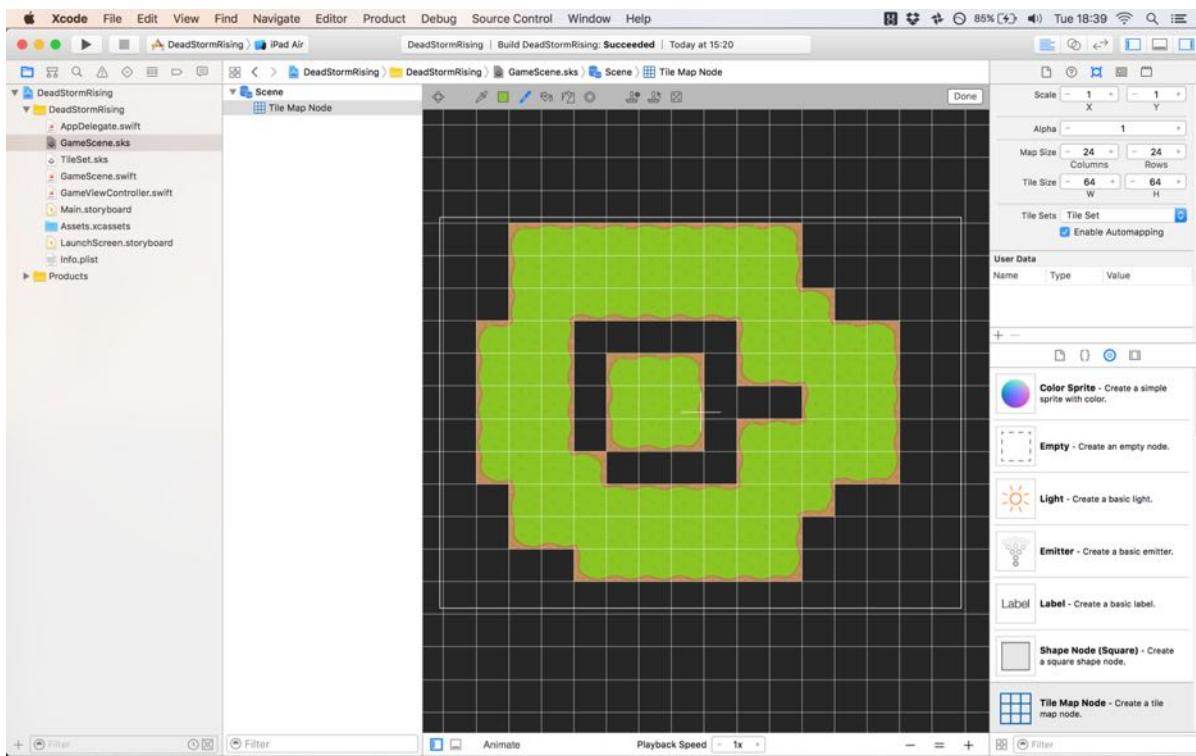
Once that's done, double-click the circle in the scene editor to activate the map editor, and you'll see a huge grid appear – those are the tiles in your map, so all you need to do is fill them in.



When you see a grid like this, it means the tile editor is enabled. Make all the changes you want, then click Done to finish.

Above the grid you'll see lots of small buttons appear, and all being well one of them will be a small green square. That means our grass tile is selected, and next to it you should see a small brush icon in blue, meaning that painting mode is active.

Xcode uses a system called automapping to make painting map nodes fast. If you try clicking around on the grid, you'll see lots of grass and mud tiles appear. Xcode intelligently places the tiles using the adjacency group we created earlier, so you can make corners, edges, islands, and more really quickly.

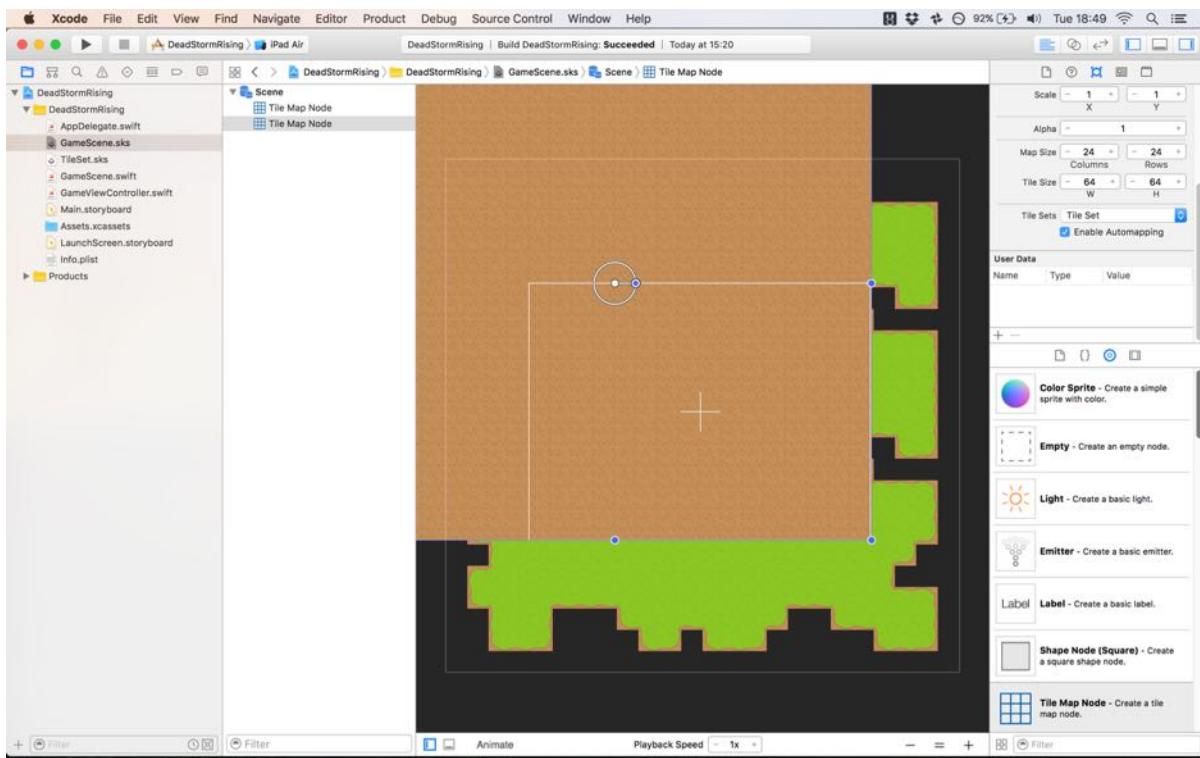


Xcode's automapping system means you can generate maps quickly just by scribbling with your mouse.

Immediately to the right of the paint button are two more tools that you will find useful for creating maps: a fill bucket does a flood fill of your selected tile, and next to that is an eraser tool to remove tiles.

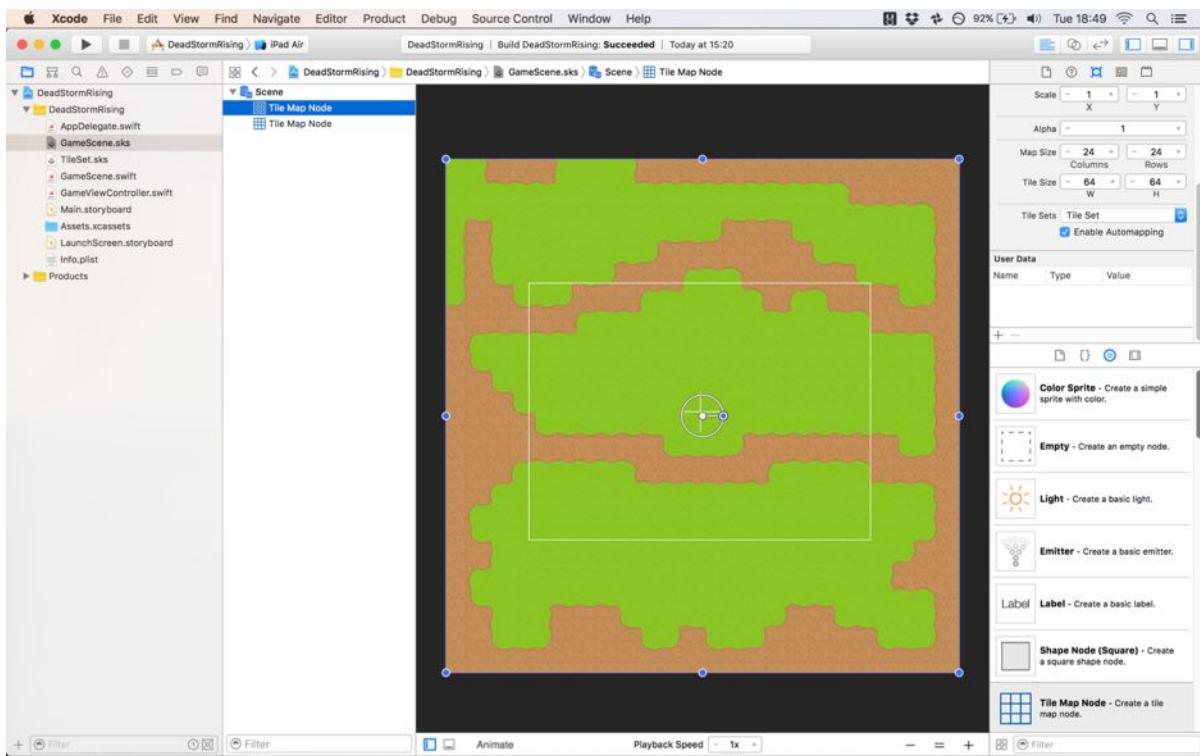
Please go ahead and fill the grid with a variety of grass tiles. When you're done, I'd like you to create a second tile map node, but make sure you place it well above and to the left of the first tile map node so that it's easier to work with. Make sure its tile size is set to 64x64 again, then set Tile Sets to have the value "Tile Set".

Double click the new tile map node, then click the green square in its toolbar and change to "Mud". Select the flood fill tool and fill the entire tile map node with mud tiles.



We have two tile map nodes: one with a variation of grass and empty pockets, and one with pure mud.

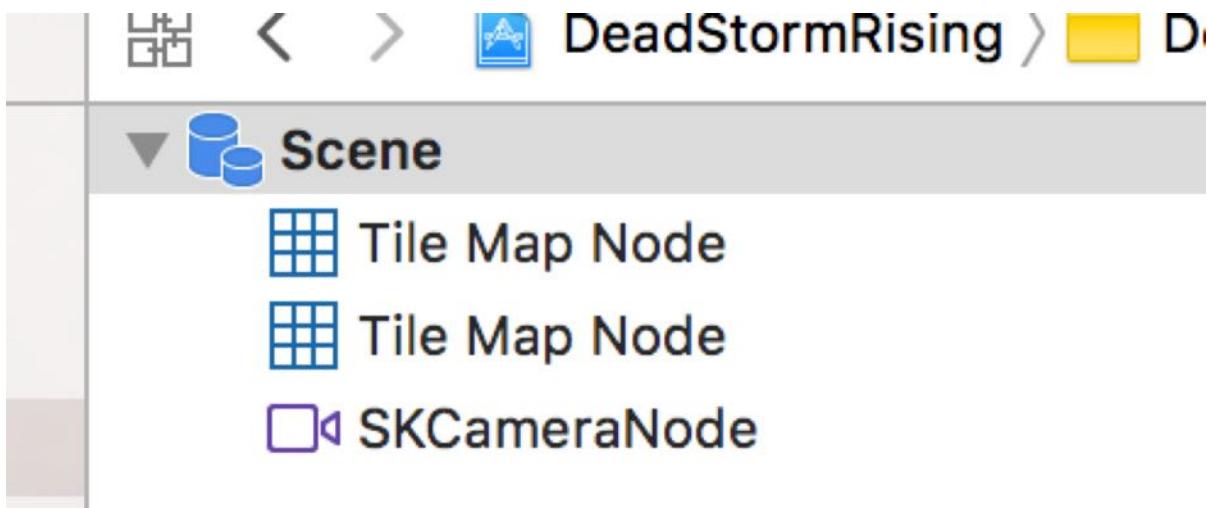
Next, move the mud tile map directly over the grass tile map, so they occupy the same position. Finally, use the document outline to move the mud tile map *behind* the grass tile map, and you should see something like the picture below.



The finished product: two tile maps overlapped to create a neat-looking map that didn't take much work.

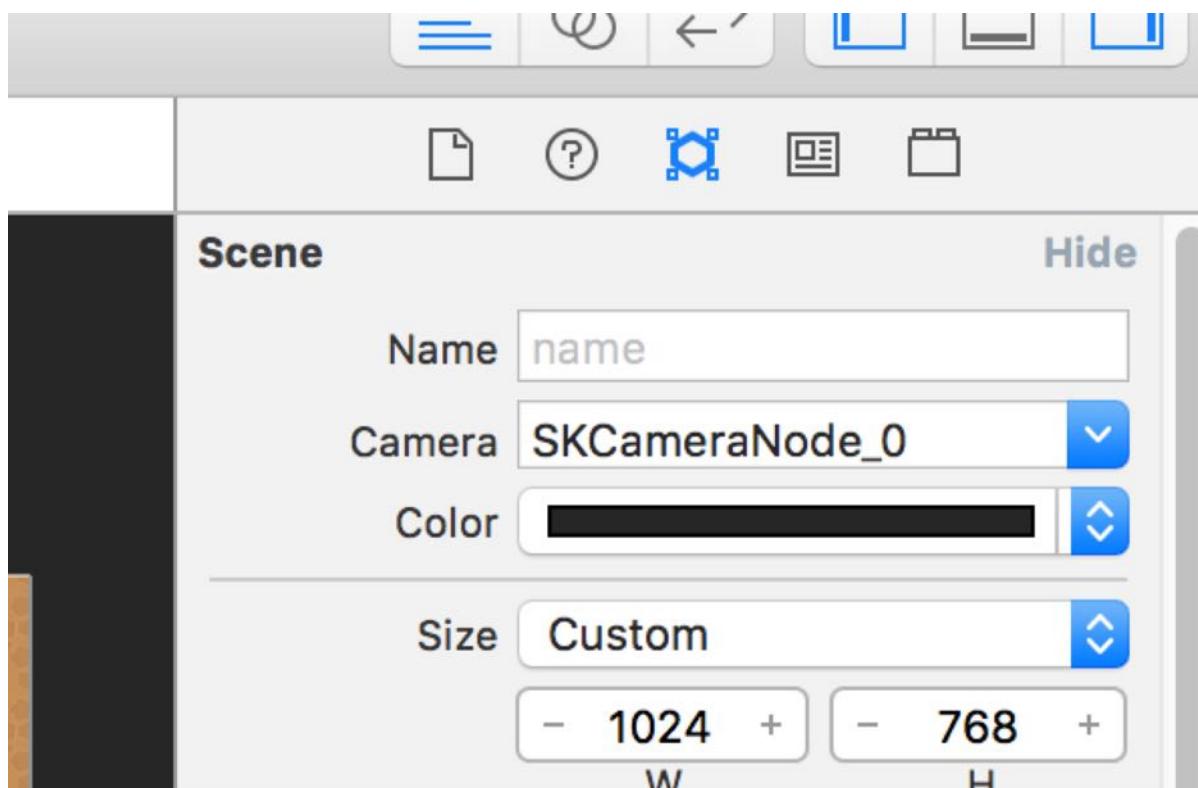
It doesn't matter what kind of grass and mud shapes you draw – it's just for decoration, really. As far as we're concerned, grass and mud are the same thing.

The last thing we're going to do is create a camera node that will be used for our game view. Drag a camera node into the scene, beneath the two existing tile map node.



Place a camera node into the scene, alongside the two tile map nodes. We'll be using this to move around the game.

Finally, select the scene itself, then go to the attributes inspector and set the Camera value to the name of the camera that was just created.



Select the scene then set its Camera value. This will attach it to the camera property in code.

That's the map finished! You can return to this later on if you want, and edit it to your heart's content.

Moving around the map

We're going to tackle two big tasks up front: panning around the map with touches, then selecting and moving units. It will take quite a bit of work, but it does mean you'll be able to see the game start to come together very early on!

Start by adding four properties to the **GameScene** class:

```
var lastTouch = CGPoint.zero  
var originalTouch = CGPoint.zero  
var cameraNode: SKCameraNode!
```

We're going to be using the first two to track touches. The first, **lastTouch**, will be used to track movement inside **touchesMoved()**, and the second, **originalTouch**, will be used to compare where the touch started and where it ended, so we can figure out whether the user tapped the screen or was just moving around. The first property, **cameraNode**, will be used to hold the **SKCameraNode** we created in the scene editor.

Put together, those three properties allow us to move the camera around the game world. When the app starts we'll point **cameraNode** to the existing **camera** property of the scene, but this way we're force unwrapping it only once rather than littering ? and ! throughout our code. When the touch starts we set both **lastTouch** and **originalTouch** based on the position of the touch inside the view. When **touchesMoved()** is called we can figure out what movement happened by subtracting the new touch position from **lastTouch**, and move **cameraNode** to reflect that change. The **lastTouch** property is then updated with the new touch position, and the process repeats.

Important: Touches are calculated using the **SKView** that contains our game scene, not the game scene itself. This is not how you normally do touch detection, but it's important: as the camera moves so does their position in the game scene, so you'll get entirely wrong results from reading the scene location.

Time for some code. In **didMove()** we need to assign the existing **camera** property to **cameraNode** so that we don't need to continually unwrap something we know will be there:

```
override func didMove(to view: SKView) {  
    cameraNode = camera!  
}
```

In **touchesBegan()** we need to pull out the touch, and figure out its location in the **SKView**. We'll assign that to both **lastTouch** and **originalTouch** - the former will change every time **touchesMoved()** is called, but the latter won't change until **touchesBegan()** is called again:

```
override func touchesBegan(_ touches: Set<UITouch>, with event:  
UIEvent?) {  
    guard let touch = touches.first else { return }  
    lastTouch = touch.location(in: self.view)  
    originalTouch = lastTouch  
}
```

And in **touchesMoved()** we need do a little bit of maths. We know where the touch was previously, because it's stored in **lastTouch**, so we need to compare that against the current touch position and adjust the camera appropriately.

There's a small complication here, which is that UIKit and SpriteKit have different Y axes: SpriteKit counts up from the bottom whereas UIKit counts down from the top. So, to calculate the X position we subtract the current touch location from the previous one, but to calculate the Y position it's the other way around.

Once we have the new position, we make that the new position for **cameraNode** so that the game moves, then update **lastTouch** to be the new touch position so we can calculate the movement from there next time.

Here's the code for **touchesMoved()**:

```
override func touchesMoved(_ touches: Set<UITouch>, with event:  
UIEvent?) {  
    guard let touch = touches.first else { return }
```

```

let touchLocation = touch.location(in: self.view)

    let newX = cameraNode.position.x + (lastTouch.x -
touchLocation.x)

    let newY = cameraNode.position.y + (touchLocation.y -
lastTouch.y)

    cameraNode.position = CGPoint(x: newX, y: newY)

lastTouch = touchLocation
}

```

Go ahead and run the game now and you'll see it works already – you can touch to pan around the entire map, and admire your map-building handiwork.

Adding units

This is a war game, which means adding units the player can move around. As with Flip in the previous chapter, we'll be creating our game sprites as subclasses of **SKSpriteNode** so that we can attach extra data. However, this time there's an extra complication: as well as controlling tanks, players will also be able to capture bases that let them build things, so really we have two kinds of game item: a unit that can move around, and a base that can build things.

To model this sensibly, we're going to create a class called "GamelItem" that inherits from **SKSpriteNode**, then "Base" and "Unit" classes that inherit from "GamelItem".

Go ahead and create the three new classes now: **GamelItem**, **Unit**, and **Base**. Make sure all three have **import SpriteKit** at the top of their files. Remember, the **GamelItem** class should inherit from **SKSpriteNode**, then **Base** and **Unit** should inherit from **GamelItem**. Those are all the classes we'll be using in this game, but if you wanted to extend it to something more complex you might want to consider a protocol-oriented approach to avoid fragile architectures.

At any given time one player is in control of the game, and all game items have an owner,

although initially that will be nobody for bases. To represent that please create a **Player** enum in GameScene.swift:

```
enum Player {  
    case none, red, blue  
}
```

Make sure you place that *outside* the **GameScene** class.



A screenshot of the Xcode code editor. On the left, there's a sidebar with file names: 'troller.swift', 'd', 's', and 'storyboard'. The main area shows the following Swift code:

```
troller.swift 9 import SpriteKit  
d 10  
s 11 enum Player {  
storyboard 12     case none, red, blue  
13 }  
14  
15 class GameScene: SKScene {  
16     var lastTouch = CGPoint.zero  
17     var originalTouch = CGPoint.zero  
18     var cameraNode: SKCameraNode!  
}
```

Place the Player enum outside the GameScene class in order for it be available everywhere.

With that enum in place, we can create two more properties. In **GameScene** add this:

```
var currentPlayer = Player.red
```

And in **GamelItem** add this:

```
var owner = Player.none
```

Before we dive into the code to create some initial units, there's one more major thing we need to define: Z positions. These tell SpriteKit how to layer sprites that are being drawn to

the screen, allowing us to position some sprites above others. You can type these directly in as you code, but when you have more than two or three it becomes hard to track and maintain. An alternative is to split them up into individual constants, but we need them to be available everywhere in the program so they are the same.

The solution we'll be using is to group all our Z positions in an enum. If you don't mind using **rawValue** everywhere then you could create an enum like this:

```
enum zPositions: CGFloat {
    case unit = 10
}
```

However, I *do* mind using **rawValue** everywhere because I think it adds clutter throughout your code. So instead we're going to create an enum called **zPositions** that has several constants attached to it – one for each thing we'll display. Doing it this way lets us place the constants into a namespace (the enum name), but also makes clear our intention that we don't want people to create new instances of it – if we used a struct instead, someone might try and create a new **zPositions** instance, for example.

There are seven game objects we need to work with:

- **base** is the lowest thing; everything should be drawn above them.
- **bullet** is next up; these are things fired from tanks.
- **unit** are next, and are the tanks players drive around.
- **smoke** and **fire** are next, which are particle systems we'll create for explosions.
- Then there's **selectionMarker** which we'll use to draw a reticule over the unit the player tapped.
- Finally there's **menuBar**, which is drawn over everything else.

I've given them Z positions ranging from 10 to 100, giving you enough space to insert new ones later on if you continue to expand the project.

Add this code to GameScene.swift, next to the existing **Player** enum:

```
enum zPositions {
```

```
static let base: CGFloat = 10
static let bullet: CGFloat = 20
static let unit: CGFloat = 30
static let smoke: CGFloat = 40
static let fire: CGFloat = 50
static let selectionMarker: CGFloat = 60
static let menuBar: CGFloat = 100
}
```

We'll only be using one of them for now, but at least they are all in place so we can use the rest later.

OK, enough of the boring stuff: it's time to put some tanks on the screen. I've given you two colors to work with as separate images, because it's likely you'll want to make the different sides stand out using a little more than just color. For now, though, we have "tankRed" and "tankBlue".

Every tank – regardless of its side – will be a **Unit** instance. I've named it **Unit** rather than **Tank** because again I think this is an area you could easily expand later – rocket launchers, humvees, and so on. We'll be adding to the **Unit** class as we go, but for now we're just going to use them as basic sprite nodes.

Each side will start with five tanks, the red side on the bottom, and the blue side off the screen above. We're going to rotate the blue tanks by 180 degrees, so they face downwards while the red tanks face upwards. Later on, we'll place some bases between both sides that can be captured, so players have a choice between rushing into a fight or trying to win territory.

Add this property to the **GameScene** class now:

```
var units = [Unit]()
```

That will store all units in the game, at least until they get destroyed. The next step is to

create a new method, **createStartingLayout()**, that will set up the initial units. Add this to **GameScene**:

```
func createStartingLayout() {  
    for i in 0 ..< 5 {  
        // create five red tanks  
        let unit = Unit(imageNamed: "tankRed")  
  
        // mark them as owned by red  
        unit.owner = .red  
  
        // position them neatly  
        unit.position = CGPoint(x: -128 + (i * 64), y: -320)  
  
        // give them the correct Z position  
        unit.zPosition = zPositions.unit  
  
        // add them to the `units` array for easy look up  
        units.append(unit)  
  
        // add them to the SpriteKit scene  
        addChild(unit)  
    }  
  
    for i in 0 ..< 5 {  
        // create five blue tanks  
        let unit = Unit(imageNamed: "tankBlue")  
        unit.owner = .blue  
        unit.position = CGPoint(x: -128 + (i * 64), y: 704)  
        unit.zPosition = zPositions.unit  
    }  
}
```

```

// these are rotated 180 degrees
unit.zRotation = CGFloat.pi

units.append(unit)
addChild(unit)

}

}

```

I chose those Y positions because they fitted with the map I drew, but you may need to adjust them or your map to fit.

To make that code work, add a call to **createStartingLayout()** in **didMove()** then run your game again – you should see five red tanks, then five blue tanks up off the top of the screen.



At this point you should see ten tanks when the game starts: five red on the screen, and five blue way up off the screen

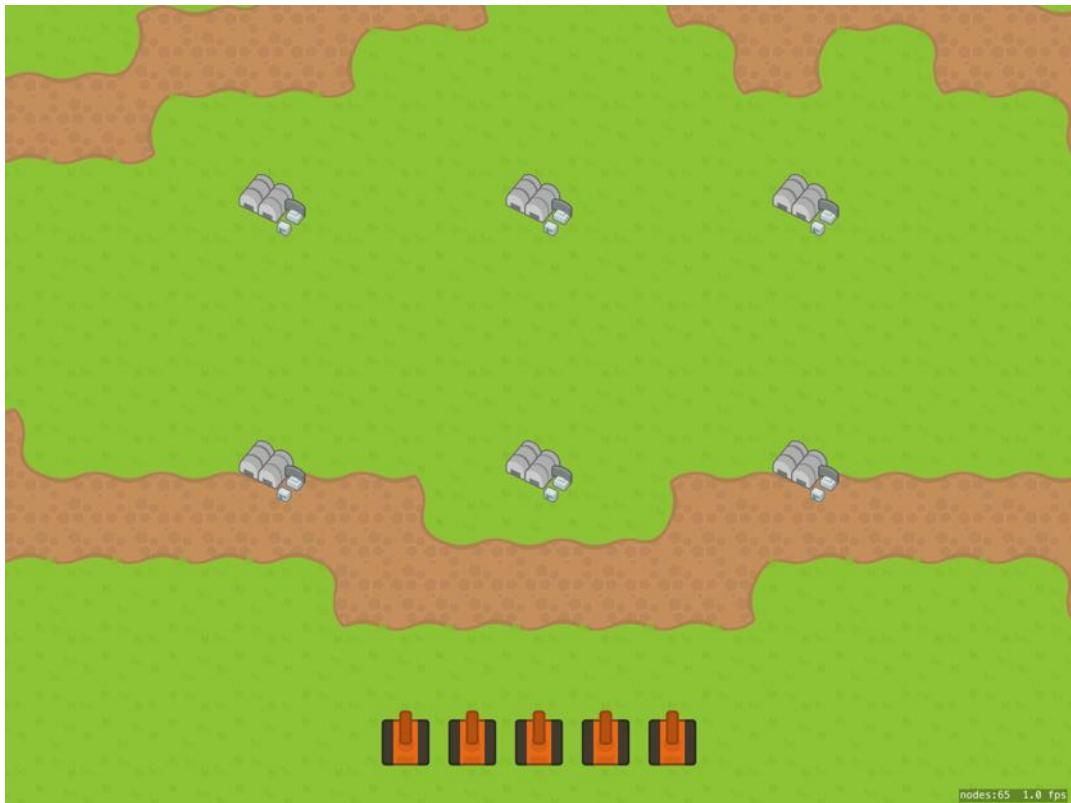
Adding bases

Every game needs a purpose, and in Dead Storm Rising that purpose will be to capture all the bases in the game. These will be small huts that units can land on, capture, then use to build more tanks in later turns. We'll be adding more later on, but for now let's just place them on the map.

You're welcome to place bases with some sort of clever precision in your own code, but for here we're just going to place them in a 3x3 grid. Again, you will need to adjust the Y coordinate so the bases look good for you, but something like this ought to come close enough:

```
for row in 0 ..< 3 {  
    for col in 0 ..< 3 {  
        let base = Base(imageNamed: "base")  
        base.position = CGPoint(x: -256 + (col * 256), y: -64 + (row  
* 256))  
        base.zPosition = zPositions.base  
        bases.append(base)  
        addChild(base)  
    }  
}
```

Add that to the start of the new **createStartingLayout()** method and run the game again – you should see 9 bases out there between the 10 tanks, ready to be captured.



The game places nine bases in spaced positions around the board, but you could easily place them by hand to make things more interesting.

Selecting game items

The next thing we're going to tackle is how players can give orders to units: moving them around the squares, and attacking enemy tanks. That's really two problems: how you *select* things, and then how you tell them what to do. There are lots of ways of doing this, but few ways are easy enough to explain in a tutorial.

I tried a few different approaches, and the easiest one is this: when the user selects a unit, show a set of squares around that unit that represent how far it can move. When the user taps one of those squares, we'll move the unit there. If a square happens to be over an enemy tank, we'll color it red and it will trigger an attack when tapped.

Laying out the movement squares isn't hard. What *is* hard is detecting what got tapped, and it's more complicated than you might think. What got tapped – was it one of the tanks? Was it a move instruction? Was it a base? Was it a menu? What happens if there's a tank and a base on the same square – which one gets selected?

One way to simplify things is to add rules that limit what can happen. For example, we're going to code this game so that one square can be occupied by only one unit at a time. Units will be allowed to occupy the same square as bases, but when the user taps on that square we'll make it always select the unit. It's a simple rule – "bases can't build things if there's a unit there already" – but again it makes selection a little easier. Note: it's a *little* easier, but still complicated.

Selecting units is done in two methods: **touchesEnded()** is called when the touch finishes and we're ready to figure out what happened, but when we're sure a tap has happened we're going to pass that on to a method called **nodesTapped()** that will carry out our selection logic.

To make sure your code compiles cleanly, start by adding this stub for **nodesTapped()** into **GameScene**:

```
func nodesTapped(at point: CGPoint) {  
}
```

The **touchesEnded()** method needs to do much the same thing as **touchesMoved()**: figure

out where the user touched, compare it to the previous touch, then move the camera appropriately. However, once that's done there's more: it needs to calculate the distance between the user's first touch – stored in `originalTouch` during `touchesBegan()` – and their final touch, then decide whether that's small enough to be considered a tap rather than a swipe. Of course, in theory it's possible a user will start a swipe, move around a fair amount, then move back to exactly where they started, but that's extra logic you can add later.

There are several ways of calculating distance, but in this game we're going to make the units move in very precise ways: they can move up, down, left, or right, but not diagonally. In a single move they might move up two spaces and left two in order to reach the position the player wants, but it's still made up of two simple directions. This movement is commonly called "Manhattan distance", which uses rectilinear geometry to calculate the absolute difference of two Cartesian coordinates. Try to imagine a taxi cab moving through a city where the blocks are square: you can move two blocks up then two to the right, or two to the right and two up, or one up, one right, one up, one right – the actual distance traveled is the same.

It's extremely fast to calculate the Manhattan distance between two points: subtract the target X from the start X, subtract the target Y from the start Y, then take the absolute values of each ("do nothing if they are positive, otherwise remove the negative sign") then add them together.

We're going to put the code for this into a `CGPoint` extension to keep our codebase clean, so create a new Swift file named `CGPoint+Additions` and give it this content:

```
import UIKit

extension CGPoint {
    func manhattanDistance(to: CGPoint) -> CGFloat {
        return (abs(x - to.x) + abs(y - to.y))
    }
}
```

Note that I've changed `import Foundation` to `import UIKit` so that we get the definition for `CGPoint`.

Now, back to **touchesEnded()**: this will call **touchesMoved()** to ensure we have the very latest information for the touch and camera position. But it will then use the new **manhattanDistance()** method to calculate the distance between the user's original touch and their final touch, and if that's less than certain threshold – I chose 44 points – then pass the touch on to **nodesTapped()**. Note, though, that the location passed into **nodesTapped()** must be the touch's position in the game scene, not in the **SKView** as we've been using so far – we need to know what the user tapped, rather than caring about relative movement.

Add this method now:

```
override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?) {  
    guard let touch = touches.first else { return }  
    touchesMoved(touches, with: event)  
  
    let distance = originalTouch.manhattanDistance(to: lastTouch)  
  
    if distance < 44 {  
        nodesTapped(at: touch.location(in: self))  
    }  
}
```

Which nodes can be tapped?

When **nodesTapped()** has been called, we can be pretty sure that the user has tapped on the screen rather than moving around and accidentally triggering something. At this point, we need to figure out what was tapped, and what it means.

So far we have two game items, a **Unit** and a **Base**, so it could be either of those. I also explained that we'll be adding move squares to the scene that represent where the player can move to or attack, so it might also be one of those. We'll be using some logic to figure

out what was tapped: moves have the first priority, then units, then cities. However, we need to add a little more: users should clearly not be able to select units and cities that don't belong to them, for example. We're also going to add a new **isAlive** property to the **Unit** class, which means we can stop users from trying to select units that are dead.

Let's start with the **Unit** class, because that's easy enough: if a unit is dead, it should not be selectable. We can do that by adding a single property to the class:

```
var isAlive = true
```

So, all units start alive, but we can change that later. As we're about to tackle moving and attacking, we might as well add two more properties to the same class:

```
var hasMoved = false  
var hasFired = false
```

We'll be using those two to ensure that each unit moves and fires only once per turn.

Back in **GameScene**, we need to track *which* item is selected: not only whether it was a base or a unit, but *which* one of those. This is as simple as adding a property to **GameScene** to track which **GameItem** instance is selected, so add this now:

```
var selectedItem: GameItem?
```

What got tapped?

We have all the code in place to be able to select items, so now it's just figuring out what got tapped. Remember, a given square might be occupied by at least three things: a move square (where a selected unit can move), a unit (alive or dead, ours or the opponent's), and a base. We need to be able to handle any of those being tapped, so the easiest thing to do is

catch all of them then figure out later on which one we want.

Let's write the first half of **nodesTapped()** now. It's going to use the **nodes(at:)** SpriteKit method to get an array of all nodes at the position that was tapped, then loop through all of them to figure out what was at that spot. Again, it might be nothing, or it might be one to three things. So, we're going to use the **is** operator to do a type check for **Unit** and **Base**, then use a node name comparison to look for "move" – that's the name we'll be giving to move squares later on. By the end of the loop we'll know which move, unit, or base was tapped, and can then make a choice based on that information.

Here's the first part of **nodesTapped()**:

```
func nodesTapped(at point: CGPoint) {
    let tappedNodes = nodes(at: point)

    var tappedMove: SKNode!
    var tappedUnit: Unit!
    var tappedBase: Base!

    for node in tappedNodes {
        if node is Unit {
            tappedUnit = node as! Unit
        } else if node is Base {
            tappedBase = node as! Base
        } else if node.name == "move" {
            tappedMove = node
        }
    }
}
```

The second part of **nodesTapped()** is where we have to apply our selection logic by evaluating what was tapped and deciding what that means. The logic is pretty simple:

1. If **tappedMove** is not nil then we have a move or attack.
2. Otherwise if **tappedUnit** is not nil then the user is trying to select or deselect a unit.
3. Otherwise if **tappedBase** is not nil then the user is trying to select a base.
4. Otherwise the user tapped somewhere else, and we should deselect whatever we had selected.

Within steps 2 and 3 we also need to check that players are trying to select units and bases they own, and within step 2 we'll also check that the unit is actually alive.

Add this code to the end of **nodesTapped()** now:

```

if tappedMove != nil {
    // move or attack
} else if tappedUnit != nil {
    // user tapped a unit
    if selectedItem != nil && tappedUnit == selectedItem {
        // it was already selected; deselect it
        selectedItem = nil
    } else {
        // don't let us control enemy units or dead units
        if tappedUnit.owner == currentPlayer && tappedUnit.isAlive {
            selectedItem = tappedUnit
        }
    }
} else if tappedBase != nil {
    // user tapped a base
    if tappedBase.owner == currentPlayer {
        // and it's theirs - select it
        selectedItem = tappedBase
    }
} else {

```

```
// user tapped something else; deselect
selectedItem = nil
}
```

Showing what's selected

That's all our selection logic for now; we'll be coming back to it later. First, though, we need to make the user's selection visible somehow. Sure, we've made a **selectedItem** property so the game knows what's selected, but we haven't given the user any visual feedback on that selection.

The most common system in these games is to show some sort of targeting reticule over whatever the user selected. I already created a simple one for you in the game assets, so now it's just a matter of showing it at the right place then hiding it when appropriate.

Start by adding this property to the **GameScene** class:

```
var selectionMarker: SKSpriteNode!
```

That will hold the sprite that highlights which game item is selected. The next step is to create two methods, **showSelectionMarker()** and **hideSelectionMarker()** that will show or hide the selection marker. Showing it will require a few steps:

1. Call **removeAllActions** on the **selectionMarker** property to ensure it's fully reset.
2. Move the **selectionMarker** sprite to the position of the currently selected item.
3. Give it an alpha of 1 so that it's fully visible.
4. Add a rotation action to make it spin around gently, then make that action repeat forever.

That last step isn't needed, but I think it helps make the game look a little more alive – try it and see what you think.

Here's the code for **showSelectionMarker()**:

```
func showSelectionMarker() {  
    guard let item = selectedItem else { return }  
    selectionMarker.removeAllActions()  
  
    selectionMarker.position = item.position  
    selectionMarker.alpha = 1  
  
    let rotate = SKAction.rotate(byAngle: -CGFloat.pi, duration: 1)  
    let repeatForever = SKAction.repeatForever(rotate)  
    selectionMarker.run(repeatForever)  
}
```

The code for **hideSelectionMarker()** is much simpler, because it just needs to remove any actions (i.e. the spinning) then set the alpha to 0. Here's the code:

```
func hideSelectionMarker() {  
    selectionMarker.removeAllActions()  
    selectionMarker.alpha = 0  
}
```

To connect those we're going to add a property observer to **selectedItem**. Initially it's just needs to trigger either **showSelectionMarker()** or **hideSelectionMarker()**, but soon we'll be making it do more things. I love property observers, but I *don't* love it when they contain lots of code because I think it clutters things up. So, we'll use a property observer, but it's just going to call a new method called **selectedItemChanged()**, which in turn will show or hide the selection marker.

Add this method now:

```
func selectedItemChanged() {
```

```
if let item = selectedItem {  
    showSelectionMarker()  
} else {  
    hideSelectionMarker()  
}  
}
```

You'll get a warning that **item** is unused, but don't worry: that will be changing soon enough.

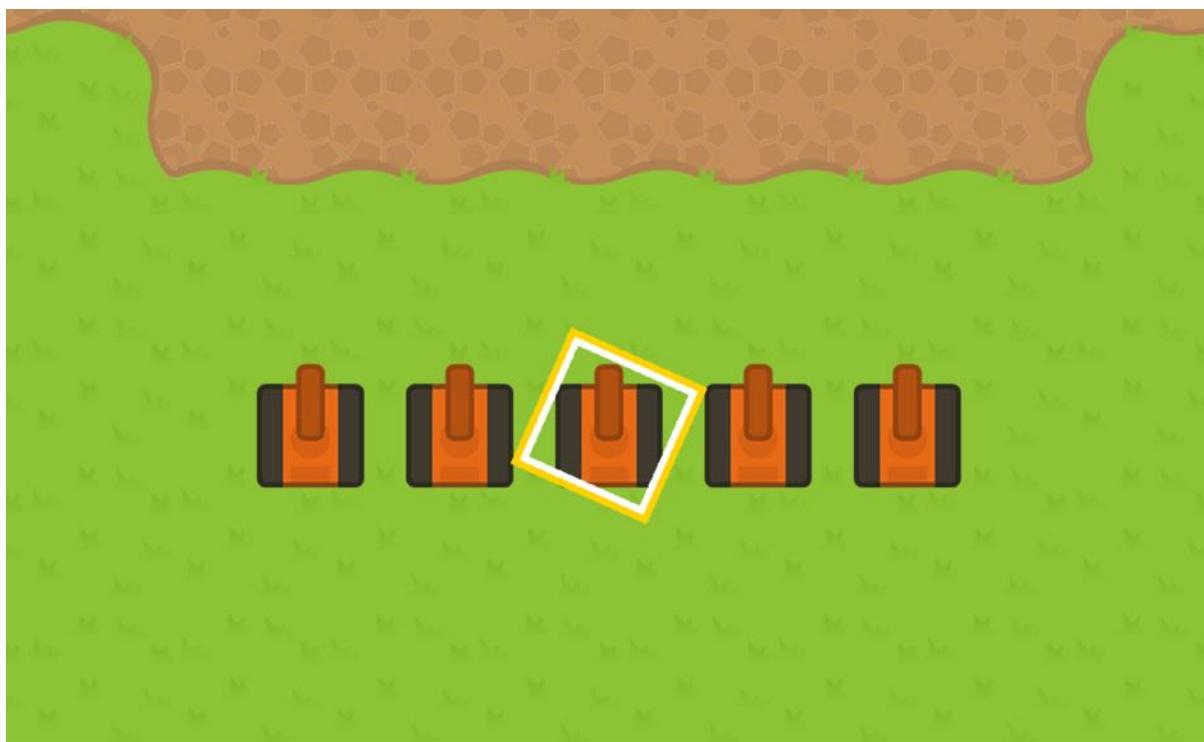
Before that code works, we need to do two more things, both of which are trivial. First, we need to actually create the **selectionMarker** sprite, so add this code to the **didMove()** method:

```
selectionMarker = SKSpriteNode(imageNamed: "selectionMarker")  
selectionMarker.zPosition = zPositions.selectionMarker  
addChild(selectionMarker)  
hideSelectionMarker()
```

Second, add the **didSet** property observer to **selectedItem** so that it calls the new **selectedItemChanged()** method:

```
var selectedItem: GameItem? {  
    didSet {  
        selectedItemChanged()  
    }  
}
```

If you run the game now you'll see you can tap on any of the red tanks to make the selection marker appear, then tap on it again to deselect. Progress!



When a unit or base is selected, we display a spinning square around it so the user knows what they are controlling.

Moving and attacking

Now that we know what unit is selected, it's time to let them move around and attack enemy tanks. This is going to require a number of steps, but only one of them is complicated!

What we're going to do is draw a set of squares around a selected unit, colored white if they can move there or colored red if it's an attack. We could create these squares each time they were needed, but there's no point – we can just re-use the ones we have.

Start by adding this property to the **GameScene** class:

```
var moveSquares = [SKSpriteNode]()
```

We'll be using that to store all the squares that show where the selected unit can move.

We're going to let units move up to four squares at a time, which might be four directly left or right, three left and one up, two right and two down, and so on. That means having 41 sprite nodes in total inside the **moveSquares** array, which can be positioned as needed.

Add this loop to the start of the **didMove()** method:

```
for _ in 0 ..< 41 {  
    let moveSquare = SKSpriteNode(color: UIColor.white(), size:  
        CGSize(width: 64, height: 64))  
    moveSquare.alpha = 0  
    moveSquare.name = "move"  
    moveSquares.append(moveSquare)  
    addChild(moveSquare)  
}
```

That creates exactly 41 white squares and adds them all to the **moveSquares** array.

To decide whether each square is a move or an attack, we need to know which units are at a

given position. For example, given the position X: 256 Y: 128, what units are at that position?

This is harder than you might expect. Every **SKNode** has a **position** property that holds a **CGPoint** identifying its precise coordinates on the screen, and there's a built-in **equalTo()** method on **CGPoint** that returns true if it matches another point. The problem is, these are stored as floating-point numbers: X: 256 might actually be stored as X: 256.0000135, which means it's difficult to compare two **CGPoint** instances accurately.

To make this work, we're going to write a **unitsAt()** method that returns an array of units that are at a specific position, plus or minus a small amount. This will use the **filter()** method of arrays to create a new array containing only units that are within range of the target position. We'll calculate that by calculating the unit's X distance and Y distance, adding them together, then returning true if that total is less than 10. Note: 10 is just a number I made up – it allows for 5 points of variation in each axis, which ought to be more than enough.

Here's how the method *could* look:

```
func unitsAt(position: CGPoint) -> [Unit] {
    return units.filter {
        let diffX = abs($0.position.x - position.x)
        let diffY = abs($0.position.y - position.y)
        return diffX + diffY < 10
    }
}
```

Don't add that just yet, though. Think about it: we're also going to need something similar for bases, and in the future you might add power ups or other items. I don't know about you, but I like to avoid code duplication as much as is sensible, so a much smarter thing to do is create an extension to the **Array** class that will contain this code whenever the array contains elements that are a subclass of **GamelItem**.

So, rather than adding the **unitsAt()** method above – or indeed adding **basesAt()** or any other duplicates – let's create an extension to **Array** when it contains **GamelItem** elements. You can put this into a new file called **Array+Additions.swift** if you want, or just place it somewhere in **GameScene.swift** outside of the **GameScene** class:

```
extension Array where Element: GameItem {  
    func itemsAt(position: CGPoint) -> [Element] {  
        return filter {  
            let diffX = abs($0.position.x - position.x)  
            let diffY = abs($0.position.y - position.y)  
            return diffX + diffY < 20  
        }  
    }  
}
```

If you do choose to put that into its own file, make sure you change **import Foundation** to **import UIKit** so that the **CGPoint** type gets imported.

Showing and hiding moves

The next step in this process is to show and hide the move options as needed. We created them all and added them to the game, so this is really just a matter of positioning and coloring them appropriately. In fact *hiding* the move options is trivial:

```
func hideMoveOptions() {  
    moveSquares.forEach {  
        $0.alpha = 0  
    }  
}
```

That just loops over all the sprites in the **moveSquares** array and hides them all.

Showing moves is harder:

1. We'll then loop from -5 to +5 rows, and -5 to +5 columns, and calculate how far that is from 0.
2. If the answer is less than or equal to 4 – in any direction – we'll consider that a possible move.
3. For each possible move we'll calculate the exact map coordinates of that square, then use our new **itemsAt()** method to figure out which units are there.
4. If there's a unit there and it either belongs to us or is dead, we'll skip this move.
5. If there's any other unit there, this is an attacking move.
6. If it's an attack, we'll color the square red and use it – but only if the unit hasn't already fired.
7. If it's a regular move, we'll color the square white and use it – but only if the unit hasn't already moved.
8. If this square is still a valid move, we'll set its alpha to 0.35 and position it appropriately.

One more thing: throughout the loop we'll be keeping a **counter** variable equal to how many squares have been placed already, so that we know which one to modify next.

That's quite a lot of work for one method, but I think you'll appreciate the end result!

Add this method to **GameScene** now:

```
func showMoveOptions() {
    guard let selectedUnit = selectedItem as? Unit else { return }

    var counter = 0

    // 1: loop from 5 squares to the left and right, bottom to top
    for row in -5 ..< 5 {
        for col in -5 ..< 5 {
            // 2: only allow moves that are 4 or fewer spaces
            let distance = abs(col) + abs(row)
            guard distance <= 4 else { continue }

            // ...
        }
    }
}
```

```

        // 3: calculate the map position for this square then see
which items are there

        let squarePosition = CGPoint(x: selectedUnit.position.x +
CGFloat(col * 64), y: selectedUnit.position.y + CGFloat(row * 64))

        let currentUnits = units.itemsAt(position:
squarePosition)

        var isAttack = false

        if currentUnits.count > 0 {

            if currentUnits[0].owner == currentPlayer ||

currentUnits[0].isAlive == false {

                // 4: if there's a unit there and it's ours or
dead, ignore this move

                continue

            } else {

                // 5: if there's any other unit there, this is an
attack

                isAttack = true

            }

        }

        if isAttack {

            // 6: if this is an attack and we haven't already
fired, color the square red

            guard selectedUnit.hasFired == false else
{ continue }

            moveSquares[counter].color = UIColor.red()

        } else {

            // 7: if this is a move and we haven't already moved,
color the square white

            guard selectedUnit.hasMoved == false else
{ continue }

            moveSquares[counter].color = UIColor.white()

        }

    }

```

```

        // 8: position the square, make it partially visible,
then add 1 to the counter so the next move square is used

        moveSquares[counter].position = squarePosition
        moveSquares[counter].alpha = 0.35
        counter += 1
    }
}

}

```

To make that work, we just need to add a handful of lines of code to the **selectedItemChanged()** method. First, we need to add a call to **hideMoveOptions()** to the start of the method, which ensures that all previous move options are removed. Then we need to adjust the conditional statement after so that if the selected item is a **Unit** we call **showMoveOptions()**.

Here's the new **selectedItemChanged()** method:

```

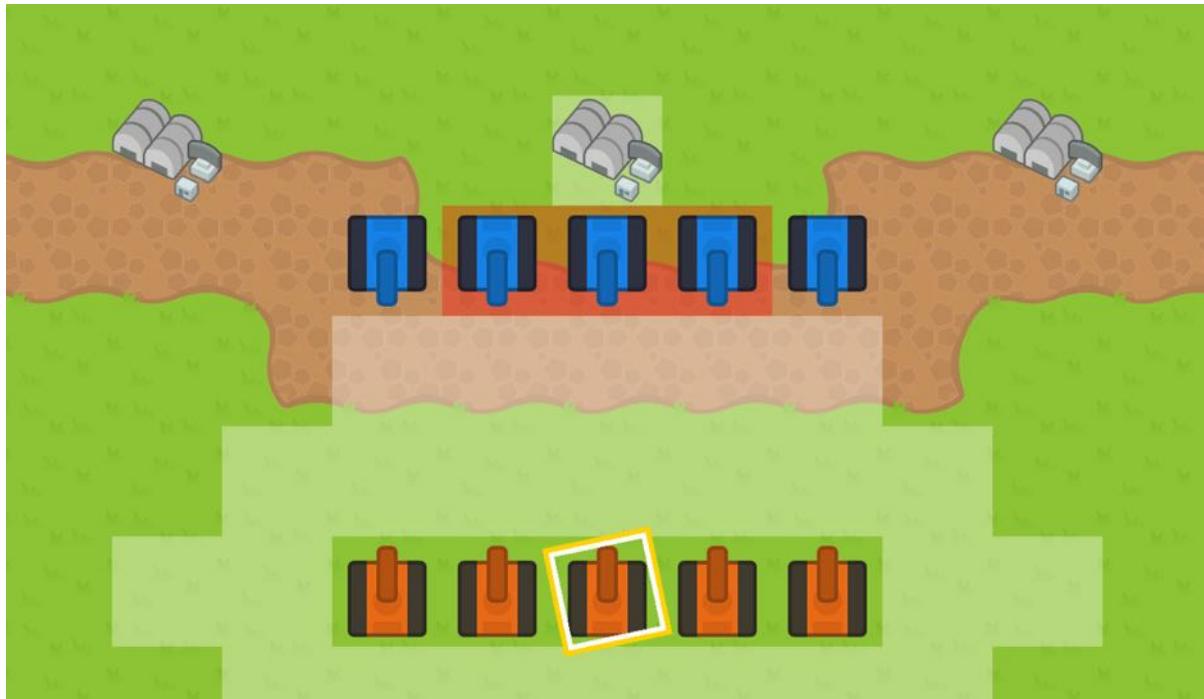
func selectedItemChanged() {
    hideMoveOptions()

    if let item = selectedItem {
        showSelectionMarker()

        if selectedItem is Unit {
            showMoveOptions()
        }
    } else {
        hideSelectionMarker()
    }
}

```

Go ahead and run the game now and you'll find you can tap on units and see a neat selection of moves available to you. What's more, it automatically ignores squares where your other tanks are, so we can be sure only one unit occupies a tile at any time.



We're now able to see where a unit can move and attack just by selecting it.

Unit, move thyself!

We're going to add two methods to the **Unit** class so that it knows how to move and attack. So, please start by opening `Unit.swift` and adding these two stubs:

```
func move(to target: SKNode) {  
}  
  
func attack(target: Unit) {  
}
```

We'll be filling them in soon enough, but first let's add support for those two methods to the **GameScene** class. Inside the **nodesTapped()** method we added a comment saying **// move or attack** that is reached when the **tappedMove** variable is not nil – i.e., when the user has selected one of the white or red squares for their unit.

We're going to replace that comment with some code that executes the correct method on **Unit** depending on what's in the square. We'll use the **itemsAt()** method again, passing it the position of the move square that was tapped – if we get back 0 we'll call **move()** on the unit, otherwise we'll call **attack()**.

Put this code where the **// move or attack** comment was:

```
guard let selectedUnit = selectedItem as? Unit else { return }
let tappedUnits = units.itemsAt(position: tappedMove.position)

if tappedUnits.count == 0 {
    selectedUnit.move(to: tappedMove)
} else {
    selectedUnit.attack(target: tappedUnits[0])
}

selectedItem = nil
```

Notice that **selectedItem** is cleared after the move or attack, which will cause the move squares and selection marker to disappear, as you'd expect.

Now let's look at the **move()** method. This is going to do three simple things:

1. If the unit has moved already, return immediately. Otherwise, set **hasMoved** to true.
2. If the unit's current X position is different to its target X position, we'll create an **SKAction** to make it follow a path from current to target X.
3. If the unit's Y position needs to change, we'll do the same thing for Y.

Both 2 and 3 will generate an **SKAction** moving the unit in one axis, so we'll store them in an array that can be executed as an **SKAction** sequence.

We covered moving a sprite along a path in Hacking with Swift project 20, but in case your memory is fuzzy: you create a new **UIBezierPath**, move it to your starting point, add a line to your finishing point, then use its **cgPath** property to pull out the finished path ready for SpriteKit. SpriteKit is able to interpret these points as relative to a sprite's existing position, and it will even rotate the sprite so that it's pointing in the direction it's moving.

Here's the code for **move()**:

```
func move(to target: SKNode) {
    // 1: refuse to let this unit move twice
    guard hasMoved == false else { return }
    hasMoved = true

    var sequence = [SKAction]()

    // 2: if we need to move along the X axis now, calculate that
    movement and add it to the sequence array
    if position.x != target.position.x {
        let path = UIBezierPath()
        path.moveTo(CGPoint.zero)
        path.addLine(to: CGPoint(x: target.position.x - position.x,
y: 0))
        sequence.append(SKAction.follow(path.cgPath, asOffset: true,
orientToPath: true, speed: 200))
    }

    // 3: repeat for the Y axis
    if position.y != target.position.y {
        let path = UIBezierPath()
```

```

        path.move(to: CGPoint.zero)
        path.addLine(to: CGPoint(x: 0, y: target.position.y -
position.y))
        sequence.append(SKAction.follow(path.cgPath, asOffset: true,
orientToPath: true, speed: 200))
    }

    // run the complete sequence of moves
    run(SKAction.sequence(sequence))
}

```

Run the game now and give it a try – the combination of the move squares must the two-axis movement looks pretty good, I think!

Firing on all cylinders

Making the units fire takes a little more work because we'd like a number of things to happen:

1. The tank should turn to face its target.
2. It needs to fire a bullet, which should fly towards its target.
3. When the bullet hits its target, we need to create an explosion and remove the bullet.
4. Any unit that is hit by a bullet needs to take damage and potentially be destroyed.

We can separate some of those into individual parts. For example, turning one **SKNode** to face another one can be done with a dash of mathematics: pass the Y delta and X delta to the **atan2()** method to get the new angle in radians. By “delta” I mean the difference between the current node’s position and its target’s position.

Add this method to the **Unit** class:

```

func rotate(toFace node: SKNode) {
    let angle = atan2(node.position.y - position.y, node.position.x -

```

```
position.x)

zRotation = angle - (CGFloat.pi / 2)

}
```

The second thing we can do is make units take damage when they are hit. This will mostly be done with one property, **health**, that will make the unit respond appropriately. We need a simple way for units to appear damaged, so we're going to go with the time-honored way of making things flash when hurt. The faster a unit flashes, the more damaged it is.

Each unit will start with a **health** property of 3. Whenever it changes, we'll either make the unit flash at a rate of $0.25 * \text{health}$ (so lower health means faster flashing), or, if it reaches a health of 0, we'll remove the flashing and make it dead.

Add this property, including a sizable property observer, to the **Unit** class:

```
// give each unit 3 health points by default

var health = 3 {

    didSet {
        // remove any existing flashing for this unit
        removeAllActions()

        // if we still have health...
        if health > 0 {
            // make fade out and fade in actions
            let fadeOut = SKAction.fadeAlpha(to: 0.5, duration: 0.25
* Double(health))

            let fadeIn = SKAction.fadeAlpha(to: 1, duration: 0.25 *
Double(health))

            // put them together and make them repeat forever
            let sequence = SKAction.sequence([fadeOut, fadeIn])
            let repeatForever = SKAction.repeatForever(sequence)
```

```
        run(repeatForever)
    } else {
        // if the tank is destroyed, change its texture to a
        // burnt out tank
        texture = SKTexture(imageNamed: "tankDead")

        // force it to have 100% alpha
        alpha = 1

        // mark it as dead so it can't be moved any more
        isAlive = false
    }
}
```

We're going to wrap that property in a method to help keep the class neatly encapsulated, so add this method to **Unit** now:

```
func takeDamage() {
    health -= 1
}
```

That will adjust the **health** property and thus trigger the property observer.

The final step is to create some explosion particle emitters. You can do this yourself if you want, you can import mine from the assets, or you can import mine them tweak them as much as you want.

If you want to use mine, copy Smoke.sks, Fire.sks, and spark.png from my assets into your project now. If you want to make your own, go to File > New > File, then choose iOS > Resource > SpriteKit Particle File, and choose whichever template you want. Please make one for smoke and one for fire, but make sure they have a sensible “maximum” value for the

number of particles to generate otherwise your explosion won't end!

That's all the preparation required, so we can now look at the **attack()** method itself. This has to do quite a lot, but the end result is pretty impressive!

Here's what this one method needs to do:

1. Ensure the unit hasn't already fired. If it has, bail out.
2. Turn the tank to face its target using the **rotate()** method we wrote.
3. Create a new bullet sprite and give it the correct bullet sprite for our tank. That is, if we are a red tank we'll use the "bulletRed" image.
4. Position the bullet underneath the tank so it looks like it's fired from the barrel.
5. Add the bullet to the unit's parent node, which is our game scene.
6. Create a path from the unit's position to its target's position
7. Create an **SKAction** for that path, this time setting **asOffset** to false because we're using exact positions.
8. Create an action that causes the target tank to take damage.
9. Create one last action that will produce smoke and fire when the bullet finishes its movement.
10. Combine the three actions together into a sequence, then add **removeFromParent()** to the end to destroy the bullet.
11. Run that sequence on the bullet sprite: it will fly forward, damage the target, cause an explosion, then remove itself from the scene.

The only part of that you are likely to be new to is creating **SKAction** instances from closures. There are lots of built-in actions for doing things like changing colors, scales, fading in and out, and more, but you can also create your own actions by using **SKAction.run()** to provide your own closure. Once it's in there, you can add these custom actions to sequences, which is exactly what we need to do here.

For example, we're going to use something like this:

```
let damageTarget = SKAction.run { [unowned target] in
    target.takeDamage()
}
```

That will call **takeDamage()** on our unit's target when the action is executed.

OK, it's time for the **attack()** method. Here's the code, complete with numbered comments matching the list above:

```
func attack(target: Unit) {  
    // 1: make sure this unit hasn't fired already  
    guard hasFired == false else { return }  
    hasFired = true  
  
    // 2: turn it to face its target  
    rotate(toFace: target)  
  
    // 3: create a new bullet and give it the same color as this tank  
    let bullet: SKSpriteNode  
  
    if owner == .red {  
        bullet = SKSpriteNode(imageNamed: "bulletRed")  
    } else {  
        bullet = SKSpriteNode(imageNamed: "bulletBlue")  
    }  
  
    // 4: place the bullet underneath the unit  
    bullet.zPosition = zPositions.bullet  
  
    // 5: add the bullet to our parent - i.e. the game scene  
    parent?.addChild(bullet)  
  
    // 6: draw a path from the bullet to the target  
    let path = UIBezierPath()
```

```

path.move(to: position)
path.addLine(to: target.position)

// 7: create an action for that movement
let move = SKAction.follow(path.cgPath, asOffset: false,
orientToPath: true, speed: 500)

// 8: create an action that makes the target take damage
let damageTarget = SKAction.run { [unowned target] in
    target.takeDamage()
}

// 9: create an action for the smoke and fire particle emitters
let createExplosion = SKAction.run { [unowned self] in
    // create the smoke emitter
    if let smoke = SKEmitterNode(fileName: "Smoke") {
        smoke.position = target.position
        smoke.zPosition = zPositions.smoke
        self.parent?.addChild(smoke)
    }

    // create the fire emitter over the smoke emitter
    if let fire = SKEmitterNode(fileName: "Fire") {
        fire.position = target.position
        fire.zPosition = zPositions.fire
        self.parent?.addChild(fire)
    }
}

// 10: create a combined sequence: bullet moves, target takes
damage, explosion is created, then the bullet is removed from the

```

```
game

    let sequence = [move, damageTarget, createExplosion,
SKAction.removeFromParent()]

    // 11: run that sequence on the bullet
    bullet.run(SKAction.sequence(sequence))

}
```

That code works just fine, but it's hard to test right now because we placed the tanks quite far apart.

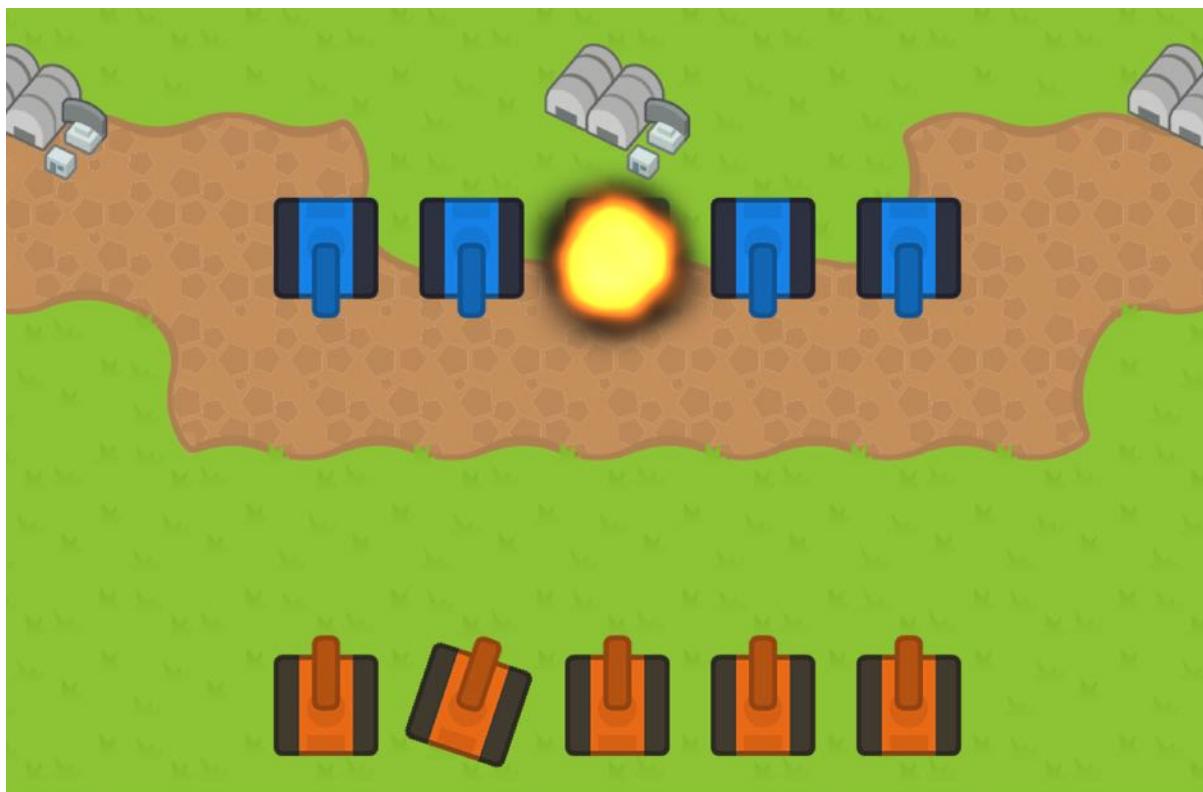
To make testing easier, find **createStartingLayout()** and change the coordinates of the blue tanks. Originally we had this:

```
unit.position = CGPointMake(x: -128 + (i * 64), y: 704)
```

Please change it to this:

```
unit.position = CGPointMake(x: -128 + (i * 64), y: -128)
```

That will position the blue tanks only a couple of squares away from the red tanks, so you should be able to try the **attack()** method. Select any red tank to bring up its move options, and you should see some squares marked red to show that will attack an enemy tank. Tap one of those red squares, and you should see the bullet and explosion. Keep on firing, and the opponent tank should be completely destroyed.



Explosions! All games work better with explosions.

Controlling the game

Now that we can move and attack, it's time to add the central element to the game: letting players capture bases and build new units there. To make this possible, we're going to add a simple menu bar that displays who is control, and adds options to capture bases, build units, and end the turn.

We're going to be storing each of the menu bar sprites in its own **SKSpriteNode**. If you want to continue down the path of having a single menu bar you might want to consider making a dedicated **MenuBar** class, but equally you might want to break this information up into two or more bars that display different kinds of information. I've left them loose, so you're welcome to re-arrange them how you please.

First, add these five properties to the **GameScene** class:

```
var menuBar: SKSpriteNode!
var menuBarPlayer: SKSpriteNode!
var menuBarEndTurn: SKSpriteNode!
var menuBarCapture: SKSpriteNode!
var menuBarBuild: SKSpriteNode!
```

The first will hold the menubar itself, which will a translucent black triangle. The second will hold the name of the current player, so there's no doubt who is in control. The third will hold a button saying "End Turn" in either red or blue. The fourth will hold a button saying "Capture" that will appear when a unit is able to capture a base. Finally, the fifth one will hold a button saying "Build" when a base is selected and can build a new unit.

I don't want to clutter up the **didMove()** method with lots of menu bar code, so please add the new **createMenuBar()** method shown below:

```
func createMenuBar() {
    // create a translucent black menu bar, positioned near the top
    menuBar = SKSpriteNode(color: UIColor(white: 0, alpha: 0.66),
                           size: CGSize(width: 1024, height: 60))
```

```

menuBar.position = CGPoint(x: 0, y: 354)
menuBar.zPosition = zPositions.menuBar
cameraNode.addChild(menuBar)

// add the "Red's Turn" sprite to the top-left corner
menuBarPlayer = SKSpriteNode(imageNamed: "red")
menuBarPlayer.anchorPoint = CGPoint(x: 0, y: 0.5)
menuBarPlayer.position = CGPoint(x: -512 + 20, y: 0)
menuBar.addChild(menuBarPlayer)

// add the red "End Turn" sprite to the top-right corner
menuBarEndTurn = SKSpriteNode(imageNamed: "redEndTurn")
menuBarEndTurn.anchorPoint = CGPoint(x: 1, y: 0.5)
menuBarEndTurn.position = CGPoint(x: 512 - 20, y: 0)
menuBarEndTurn.name = "endturn"
menuBar.addChild(menuBarEndTurn)

// add the "Capture" button to the center of the menu bar
menuBarCapture = SKSpriteNode(imageNamed: "capture")
menuBarCapture.position = CGPoint(x: 0, y: 0)
menuBarCapture.name = "capture"
menuBar.addChild(menuBarCapture)
hideCaptureMenu()

// add the "Build" button to the center of the menu bar
menuBarBuild = SKSpriteNode(imageNamed: "build")
menuBarBuild.position = CGPoint(x: 0, y: 0)
menuBarBuild.name = "build"
menuBar.addChild(menuBarBuild)
hideBuildMenu()

```

```
}
```

There are a few interesting things in that code:

1. Note that the **menuBar** sprite is added as a child to **cameraNode**. This means it will stay fixed even when the user scrolls.
2. All the buttons are added as children of **menuBar**, so we position them relative to the bar rather than the whole screen.
3. I've adjusted the **anchorPoint** property of **menuBarPlayer** and **menuBarEndTurn** so that we can align them easily – the former to the left, and the latter to the right.
4. The End Turn, Capture, and Build buttons all have unique names we can identify, which will let us do tap detection soon.
5. It calls two methods that don't exist yet: **hideCaptureMenu()** and **hideBuildMenu()**.

Let's start by making the code compile, which means adding the **hideCaptureMenu()** and **hideBuildMenu()** methods. All they will do right now is set the alpha to 0 for those two bars – I've made them a method so that you can add animation or other functionality easily. Along with the two **hide...** methods we'll also write **show...** methods that set the alpha to 1.

So, add these four methods to **GameScene** now:

```
func hideCaptureMenu() {  
    menuBarCapture.alpha = 0  
}  
  
func showCaptureMenu() {  
    menuBarCapture.alpha = 1  
}  
  
func hideBuildMenu() {  
    menuBarBuild.alpha = 0  
}
```

```
func showBuildMenu() {  
    menuBarBuild.alpha = 1  
}
```

Finally, add a call to **createMenuBar()** in the **didMove()** method, triggering all the sprite creation.

Updating the game status

Now that we have a menu bar, the next step is to update the **selectedItemChanged()** method so that it shows Build and Capture at the right times.

Right now, it has this code:

```
func selectedItemChanged() {  
    hideMoveOptions()  
  
    if let item = selectedItem {  
        showSelectionMarker()  
  
        if selectedItem is Unit {  
            showMoveOptions()  
        }  
    } else {  
        hideSelectionMarker()  
    }  
}
```

We need to extend that a little so that when the selection is changed we always start by hiding the build and capture menus. If we selected a unit and its over a base, show the

capture menu; if we selected a base then show the build menu.

Here's the new code for the method:

```
func selectedItemChanged() {
    // hide all buttons by default
    hideMoveOptions()
    hideCaptureMenu()
    hideBuildMenu()

    if let item = selectedItem {
        // something was selected; show the selection marker
        showSelectionMarker()

        if selectedItem is Unit {
            // it was a unit - let the user choose where to move
            showMoveOptions()

            // get the list of bases at this item's position
            let currentBases = bases.itemsAt(position: item.position)

            if currentBases.count > 0 {
                if currentBases[0].owner != currentPlayer {
                    // we found a base we didn't already own
                    showCaptureMenu()
                }
            }
        } else {
            // the user selected a base
            showBuildMenu()
        }
    }
}
```

```

        }

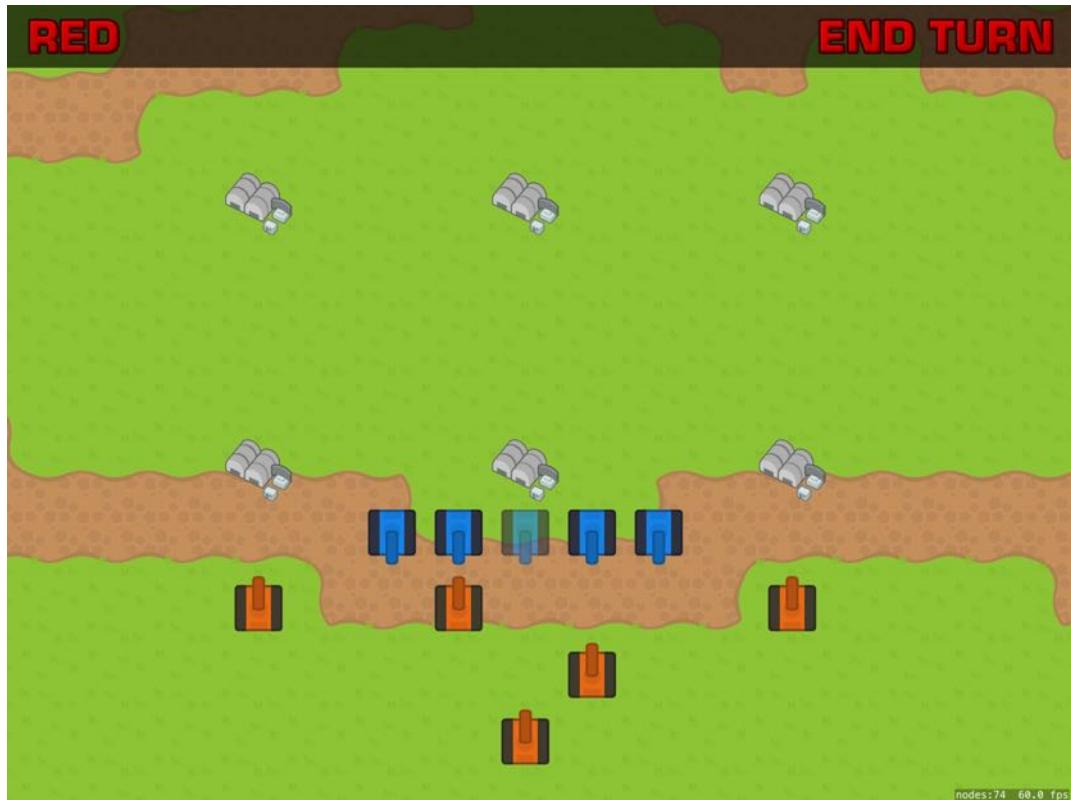
    } else {

        // nothing was selected
        hideSelectionMarker()

    }

}

```



Adding a menu bar across the top helps players understand who is in control and what they can do.

Ending turns

If you run the game you'll see you can move a tank over a base and have the capture button appear in the menu bar. However, tapping it won't do anything, and neither will tapping End Turn.

To make End Turn work, we need some new methods. First, some easy ones: we need a method in **Base** and one in **Unit** that will reset each item when a turn ends. For bases, that means adding a property called **hasBuilt** that tracks whether the base has built a unit this turn, and setting that back to false when the turn ends so it can build another unit next turn.

Give the **Base** class this code:

```
class Base: GameItem {  
    var hasBuilt = false  
  
    func reset() {  
        hasBuilt = false  
    }  
}
```

For units it's a little trickier: if the unit is alive, we're going to reset its **hasFired** and **hasMoved** properties to false so they can move and fire the following turn. However, if they are dead – i.e. **isAlive** is false – we're going to make them fade away then be destroyed so the map doesn't end up cluttered with dozens of ex-tanks.

Here's the **reset()** method for the **Unit** class:

```
func reset() {  
    if isAlive == true {  
        hasFired = false  
        hasMoved = false  
    } else {  
        let fadeAway = SKAction.fadeOut(withDuration: 0.5)  
        let sequence = [fadeAway, SKAction.removeFromParent()]  
        run(SKAction.sequence(sequence))  
    }  
}
```

```
    }  
}  


---


```

With those two in place we can write an **endTurn()** method. It will switch the controlling player, reset all bases and units, remove any units that are now dead, then set **selectedItem** to nil to clear any previous selection.

Switching the controlling player might seem just a matter of changing the **currentPlayer** property, but we're also going to change the **menuBarEndTurn** and **menuBarPlayer** textures to the correct colors.

Here's the code for **endTurn()** – put it into **GameScene** please:

```
func endTurn() {  
    if currentPlayer == .red {  
        // switch the controlling player  
        currentPlayer = .blue  
  
        // update the two textures  
        menuBarEndTurn.texture = SKTexture(imageNamed: "blueEndTurn")  
        let setTexture = SKAction.setTexture(SKTexture(imageNamed:  
"blue"), resize: true)  
        menuBarPlayer.run(setTexture)  
    } else {  
        currentPlayer = .red  
        menuBarEndTurn.texture = SKTexture(imageNamed: "redEndTurn")  
        let setTexture = SKAction.setTexture(SKTexture(imageNamed:  
"red"), resize: true)  
        menuBarPlayer.run(setTexture)  
    }  
  
    // reset all bases and units
```

```

bases.forEach { $0.reset() }

units.forEach { $0.reset() }

// remove any dead units
units = units.filter { $0.isAlive }

// clear whatever was selected
selectedItem = nil
}

```

There are two pieces of that method that may have caught you by surprise.

First, we change the **menuBarEndTurn** texture just by assigning to its **texture** property, but we change the controlling player texture by using an **SKAction**. This is required because we need the sprite to resize to fit the new image – “BLUE” is bigger than “RED”, and we don’t want them both squeezed in the same space.

Second, dead units are removed using a single call to **filter()**: if the unit is alive, include it in the **units** array. The two calls to **forEach()** also use a functional approach, and I hope you’ll agree it keeps the code clean and simple!

Capturing bases

We haven’t connected the **endTurn()** method to the **nodesTapped()** methods just yet, because before we do that I want to create the code for capturing bases and building units.

When a player moves one of their units over a base that is either neutral or owned by the enemy, we’re going to show a “capture” menu button that will transfer control over to them. This is going to change the **owner** property of the base, set its **hasBuilt** property to true so that bases can’t build a tank the same turn they are captured, then apply a color blend to the base image so it’s visibly owned by one player.

All that will be wrapped up in a single method, **setOwner()**. Add this to the **Base** class now:

```
func setOwner(_ owner: Player) {
    self.owner = owner
    hasBuilt = true
    self.colorBlendFactor = 0.9

    if owner == .red {
        color = UIColor(red: 1, green: 0.4, blue: 0.1, alpha: 1)
    } else {
        color = UIColor(red: 0.1, green: 0.5, blue: 1, alpha: 1)
    }
}
```

That effectively means that bases handle most of the capturing process themselves, but there's still a little extra work we need to do inside the **GameScene** class. We're going to create a new method there called **captureBase()** that will be triggered when a unit is selected. It's going to check whether there is in fact a base there to capture, and that it's not already owned by the player. If both of those are true, then it will call the **setOwner()** method we just defined and then clear the **selectedItem** property.

Here's the code for **captureBase()**; add this to the **GameScene** class:

```
func captureBase() {
    guard let item = selectedItem else { return }
    let currentBases = bases.itemsAt(position: item.position)

    // if we have a base here...
    if currentBases.count > 0 {
        // ...and it's not owned by us already
        if currentBases[0].owner != currentPlayer {
            // capture it!
        }
    }
}
```

```
        currentBases[0].setOwner(currentPlayer)
        selectedItem = nil
    }
}

}
```

Building new units

We now have code to trigger ending the turn and capturing bases, although we still haven't written the code necessary to make those actions happen. We'll come to that in just a moment, but first I want to add the code for one final action: building units.

Bases will be responsible for building things, so we're going to give them a **buildUnit()** method. However, as with capturing, we still need **GameScene** to do a little of the work because we need new units to be in the **units** array so we can manipulate them.

The **buildUnit()** method needs to do several things:

1. Check that each base builds only one thing per turn.
2. Create a new unit either using "tankRed" or "tankBlue".
3. Mark the new unit has already having moved and fired this turn so players need to wait for next turn to use them.
4. Assign the new unit the same owner and position as the base.
5. Give the unit the same Z position we use for all units.
6. Return it back to the caller.

Here's the code – add this to the **Base** class:

```
func buildUnit() -> Unit? {
    // 1: ensure bases build only one thing per turn
    guard hasBuilt == false else { return nil }
    hasBuilt = true
```

```

// 2: create the new unit
let unit: Unit

if owner == .red {
    unit = Unit(imageNamed: "tankRed")
} else {
    unit = Unit(imageNamed: "tankBlue")
}

// 3: mark it as having moved and fired already
unit.hasMoved = true
unit.hasFired = true

// 4 give it the same owner and position as this base
unit.owner = owner
unit.position = position

// 5: give it the correct Z position
unit.zPosition = zPositions.unit

// 6: send it back to the caller
return unit
}

```

Note that the method returns **Unit?** because it will return nil if the base built something already.

Tying it all together

We've now created three complete actions: ending the turn, capturing a base, and building new units. What we *haven't* done yet is call those methods from anywhere, because they are all called from the same place: the **nodesTapped()** method. That method tries to figure out what to do when the screen was tapped, and we're going to extend that now so that it understands the menu bar buttons being tapped.

We already have a check in there for **node.name == "move"**, so we can start by just copying that and adding more cases. However, whenever any of the menubar buttons are tapped we *must* call **return** immediately after acting on the tap. This is important: if we don't call **return** it's possible one action will be taken followed swiftly by another – with the second action potentially being a bit confusing. Remember, these nodes are just layered up things on the screen: if someone taps "Capture" and we *don't* exit the **nodesTapped()** method, they might accidentally select a different unit afterwards because their tap also triggered something else.

If we match the node name "endturn" then we'll call **endTurn()**; if we match "capture" we call **captureBase()**; so far, so easy. But handling building does require a little more logic because we need to make sure the user actually has a base selected. We'll then call the new **buildUnit()** method and check its response – if we got a valid **Unit** back we'll add that to the **units** array and also add it to the game scene.

Replace the current **for node in tappedNodes** loop in the **nodesTapped()** method with this new one:

```
for node in tappedNodes {  
    if node is Unit {  
        tappedUnit = node as! Unit  
    } else if node is Base {  
        tappedBase = node as! Base  
    } else if node.name == "move" {  
        tappedMove = node  
    } else if node.name == "endturn" {  
        endTurn()  
        return  
    } else if node.name == "capture" {
```

```
captureBase()

return

} else if node.name == "build" {

    guard let selectedBase = selectedItem as? Base else
{ return }

    if let unit = selectedBase.buildUnit() {

        // we got a new unit back!
        units.append(unit)
        addChild(unit)

    }

selectedItem = nil
return
}
```

With that code in place, the game is done – you can move, attack, capture, and build, and hopefully conquer your opponent using vast amounts of flaming death.

Wrap up

This has been a big project, but trust me: I had to simplify so much just to get it to this size! SpriteKit's new tile map nodes are a brilliant way of creating interesting, varied maps just by piecing together small graphics. Keeping your map designs separate from your code also means that designers and coders can work alongside much more easily, which is always a good thing.

There are so very many ways you could take this project further, and I've listed some ideas below in the homework section. Hopefully you've have your own ideas, though, and are maybe even inspired by how easy it is!

Homework

- Fun: Add a pre-game screen with the “Dead Storm Rising” logo you’ll find in my assets. Add another screen for “Red Wins” or “Blue Wins” depending on who dies first. Don’t let users execute another order until the first one finished.
- Tricky: Create two new particle emitter types: a small puff of smoke that is emitted when a bullet is fired, and a continuous stream of smoke that can be attached to damaged tanks.
- Taxing: Create another unit type that can be built in bases, such as a heavy tank that moves slower but does more damage.
- Nightmare: Have a go at crafting a simple AI. Don’t be afraid to make it use random behavior to decide precisely where tanks move – players often see random behavior and interpret it as things like “ah, the computer is trying to flank me!” (Note: this is a situation where the GameplayKit AI options are *not* suitable.)

Project 8

Techniques

Animations

This is the first technique project in the book. If you didn't follow Hacking with Swift, these projects are much smaller and simpler than the others: the goal is to teach one specific aspect of iOS development without building a working app around it.

In this project we're going to play around with the new iOS 10 animation framework, powered by **UIViewControllerAnimated**. We used it briefly in project 4 to make a stack view appear and disappear, but I want to demonstrate a few other important things to you so you can start to feel more comfortable with it.

To start, open Xcode and create a new Single View Application project targeting iPad. Please lock it so that it supports landscape only, then embed its view controller inside a navigation controller.

Scrubbing animations

iOS 10's animation framework allows us to interactively adjust the position of an animation, making it jump to any point in time that we need. This is *incredible*: imagine a first run screen where you go from showing thing 1 to thing 5 over a series of steps – thanks to **UIViewControllerAnimated** you can now set up one animation then just move it through various positions automatically.

We're going to construct a simple piece of scaffolding to demonstrate this. I've written it all in code to make explanation easier: we're going to create a **UISlider** then fix it to the bottom of our view, spanning the full width.

Open ViewController.swift and add this code to **viewDidLoad()**:

```
let slider = UISlider()  
slider.translatesAutoresizingMaskIntoConstraints = false  
view.addSubview(slider)  
  
slider.bottomAnchor.constraint(equalTo: view.bottomAnchor).isActive =  
true
```

```
slider.widthAnchor.constraint(equalTo: view.widthAnchor).isActive = true
```

It's not going to win awards for beautiful design, but that isn't the point.

When that slider is dragged from left to right, it will count from 0 to 1 and we're going to use that to manipulate an animation of a red box sliding across the screen.

Add this code to **viewDidLoad()**:

```
let redBox = UIView(frame: CGRect(x: -64, y: 0, width: 128, height: 128))  
redBox.translatesAutoresizingMaskIntoConstraints = false  
redBox.backgroundColor = UIColor.red()  
redBox.center.y = view.center.y  
view.addSubview(redBox)
```

That creates a 128x128 red box, centered vertically and part-way off the left edge of the screen. Even though we're going to manipulate it elsewhere in the app, we *don't* need a property for it – **UIViewPropertyAnimator** works using closures, so it will capture the box for us.

Let's create that now. Add a property for the animator:

```
var animator: UIViewPropertyAnimator!
```

We're going to make the animation move the box from the left to the right, while spinning around and scaling down to nothing. All that will happen over two seconds, with an ease-in-ease-out curve. Add this to the end of **viewDidLoad()**:

```
animator = UIViewPropertyAnimator(duration: 2, curve: .easeInOut)  
{ [unowned self, redBox] in
```

```
    redBox.center.x = self.view.frame.width  
  
    redBox.transform = CGAffineTransform(rotationAngle:  
CGFloat.pi).scaleBy(x: 0.001, y: 0.001)  
}
```

That doesn't actually *run* the animation, which is OK for now. Instead, it creates the animation and stores it away in the **animator** property, ready for us to manipulate.

At this point, we have a slider on the screen and a red box too, so we just need to connect it all. When the slider is moved, its **.valueChanged** event will be triggered, and we can add a method to catch that. We can actually feed the slider's **value** property – the number from 0.0 to 1.0 – directly into the **fractionComplete** property of our **UIViewControllerAnimated**, which controls how much of the animation has happened, and UIKit will take care of the rest for us.

Add this method to **ViewController**:

```
func sliderChanged(_ sender: UISlider) {  
    animator.fractionComplete = CGFloat(sender.value)  
}
```

To make that get called by the slider, add this to **viewDidLoad()**:

```
slider.addTarget(self, action: #selector(sliderChanged),  
for: .valueChanged)
```

That's it! We've created the user interface, prepared an animation, then connected the slider's value to the animation's progress. If you run the app now you'll see you can drag the slider from left to right and back again to manipulate the box – you literally have exact control over its position in the animation.

Start, stop, reverse

As well as being interactively scrubbed, you can also pause and continue animations, and even set them in reverse if you need to. To demonstrate this, we're going to add a couple of buttons to the navigation bar – add this to `viewDidLoad()`:

```
let play = UIBarButtonItem(barButtonSystemItem: .play, target: self, action: #selector(playTapped))  
let flip = UIBarButtonItem(barButtonSystemItem: .rewind, target: self, action: #selector(reverseTapped))  
navigationItem.rightBarButtonItem = [play, flip]
```

You'll get two compiler errors there because we haven't created `playTapped()` or `reverseTapped()` just yet, but both of them are trivial. To play animations, call the `startAnimation()` method on the `UIViewControllerAnimated`; to reverse animations, just set the `isReversed` property to true.

Here's the code for the two missing methods:

```
func playTapped() {  
    animator.startAnimation()  
}  
  
func reverseTapped() {  
    animator.isReversed = true  
}
```

You should now be able to tap the play button to set the animation moving, then at any point tap the rewind button make the animation change direction. That's neat, but we can make the play button do more.

Along with `startAnimation()` we also have `pauseAnimation()` and `stopAnimation()`. The difference is subtle but important because it exposes a little of the underlying animation

system of iOS.

When you animate a view from position A to position B, iOS immediately moves it to B. It might not *look* like it does, but it does. This is the gap between the model layout and the presentation layer: the model – the underlying values – is immediately changed when an animation begins, whereas the presentation – the bit you can see – is what smoothly slides across the screen.

That's where the difference between pausing and stopping comes in: when you pause an animation it means you intend to continue it, so the model and presentation layers will remain different: the model will stay at the end point, and the presentation will stay wherever the view is on the screen right now. When you stop an animation it means it's over, and the model layer will be updated to match the presentation layer.

When you want to continue a paused animation – which you should, otherwise you should have stopped it! – there's a special **continueAnimation()** method that lets you adjust the way the animation works. You can specify new timing parameters if you want, or just pass **nil** to continue with the existing parameters. You can also specify a **durationFactor** parameter, which is multiplied against the original animation duration to figure out how fast to make the animation when it continues.

To demonstrate this, we can build on the **playTapped()** method so that the animation will play, pause, or continue depending on the current animation state. If it's already running – which is the iOS way of saying “in flight”, which includes paused – then we'll read the **isRunning** parameter to determine whether to pause or continue the animation. Otherwise, we'll just call **startAnimation()** to set it in motion.

Here's the new code for **playTapped()**:

```
func playTapped() {
    // if the animation has started
    if animator.state == .active {
        // if it's current in motion
        if animator.isRunning {
            // pause it
            animator.pauseAnimation()
```

```
        } else {
            // continue at the same speed
            animator.continueAnimation(withTimingParameters: nil,
durationFactor: 1)
        }
    } else {
        // not started yet; start it now
        animator.startAnimation()
    }
}
```

You should now be able to tap the play button to start the animation, then tap it again and again to pause and unpause it. You can even hit the reverse button, then pause and unpause that – nice!

Animation completion

We used animation completion in project 4, but it's worth revisiting just briefly because it has an interesting complication when it comes to ending animations.

To demonstrate this, add a completion at the end of `viewDidLoad()`:

```
animator.addCompletion { [unowned self] position in
    if position == .end {
        self.view.backgroundColor = UIColor.green()
    } else {
        self.view.backgroundColor = UIColor.black()
    }
}
```

Try running it now, and you'll see that if the square reaches the right-hand edge the screen will turn green, but if you reverse it and it reaches the left-hand edge it will turn black. The important thing is that both these circumstances are considered *completed*, hence why the completion block is called.

Let's add to this mix the **stopAnimation()** method, which stops an animation and updates the model layer to match the presentation layer. It takes a single, unnamed boolean parameter, although I suspect that will change in future iOS 10 betas because unnamed boolean parameters are terrible for legibility. If you specify "true" it will effectively destroy the animation, and your completion block will never be called. If you specify "false" it will mark the animation as stopped, but you still have the option of completing it later on.

To demonstrate this in action, try changing the **playTapped()** method to the code below:

```
func playTapped() {
    if animator.state == .active {
        animator.stopAnimation(false)
        animator.finishAnimation(at: .end)
    } else {
        animator.startAnimation()
    }
}
```

That will stop and finish the animation rather than pausing it. The nice thing about this approach is that you can call **finishAnimation()** at any point – it doesn't need to be directly after the **stopAnimation()** call.

Wrap up

The new animation system is designed to complete the existing **UIView** methods we already have. They are a bit more verbose, yes, but by breaking animation components up into parts we have much more control.

Units

iOS 10 introduces a new system for calculating distance, length, area, volume, duration, and many more measurements. It's not hard to grasp, and in fact you can learn it all in about 20 minutes - a perfect technique project, I think!

The best way to experiment with measurements is using a Swift playground, so please create one now.

There are three important data types you need to learn: **Unit** describes a unit of measurement (e.g. Celsius or meters), **Measurement** provides a single value combined with a unit (e.g. 32 degrees Celsius), and **MeasurementFormatter** is used to convert a measurement to a human-readable format.

Let's start with something simple. If you're six feet tall, you'd create a **Measurement** instance like this:

```
let heightFeet = Measurement(value: 6, unit: UnitLength.feet)
```

Note that Swift can't infer **.feet** to mean **UnitLength.feet** because there are lots of **Unit** subclasses as you'll see soon.

Once you have a measurement ready, you can convert it to other units like this:

```
let heightInches = heightFeet.converted(to: UnitLength.inches)
let heightSensible = heightFeet.converted(to: UnitLength.meters)
```

You should see "72.0 in" and "1.8288 m" in the playground output, showing that the conversion process has worked.

The **UnitLength** class, like all unit subclasses, spans a huge range of units from old to futuristic. For example, you can convert feet to furlongs – a unit of measurement that comes from the Old English words furh (furrow) and lang (long) that describes the length of the furrow in one acre of a plowed field – or you can convert it to astronomical units, which is

equal to the average distance between the Earth and the Sun, or about 150 million kilometers:

```
let furlongs = heightFeet.converted(to: UnitLength.furlongs)
let heightAUs = heightFeet.converted(to:
UnitLength.astronomicalUnits)
```

Once you've used one unit, the rest work identically. Here are some more examples to get you started:

```
// convert degrees to radians
let degrees = Measurement(value: 180, unit: UnitAngle.degrees)
let radians = degrees.converted(to: .radians)

// convert square meters to square centimeters
let squareMeters = Measurement(value: 4, unit: UnitArea.squareMeters)
let squareCentimeters =
squareMeters.converted(to: .squareCentimeters)

// convert bushels to imperial teaspoons
let bushels = Measurement(value: 6, unit: UnitVolume.bushels)
let teaspoons = bushels.converted(to: .imperialTeaspoons)
```

Honestly, I have no idea what the bushels to imperial teaspoons ratio is, but it's nice to be given the option!

Calculating and printing

Once you have a **Measurement** instance set up, you can manipulate them with some basic calculations. For example, we had this measurement from earlier:

```
let heightFeet = Measurement(value: 6, unit: UnitLength.feet)
```

If you multiply that by another number – integer or double – you will get another **Measurement** instance back with that applied. For example:

```
let doubleHeight = heightFeet * 2
```

You can also manipulate them by using another **Measurement** instance. For example, if you ran two laps around a field, one in 49 seconds and the other in 58 seconds, you could add them together then get the result in minutes:

```
let firstLap = Measurement(value: 49, unit: UnitDuration.seconds)
let secondLap = Measurement(value: 58, unit: UnitDuration.seconds)
let total = (firstLap + secondLap).converted(to: .minutes)
```

You can even mix your units in the calculation. That means we could add a third lap – so slow it's measured in minutes! – alongside the first two, and it would be converted all the same:

```
let firstLap = Measurement(value: 49, unit: UnitDuration.seconds)
let secondLap = Measurement(value: 58, unit: UnitDuration.seconds)
let thirdLap = Measurement(value: 1.3, unit: UnitDuration.minutes)
let total = (firstLap + secondLap + thirdLap).converted(to: .minutes)
```

Once you've manipulated the values as much as you want, you can render them human-readable using the **MeasurementFormatter** class. It's a bit glitchy right now, but the workaround is easy enough. Here's an example that prints "32 degrees Celsius":

```
let formatter = MeasurementFormatter()  
let distance = Measurement(value: 32, unit: UnitTemperature.celsius  
as Unit)  
formatter.unitStyle = .long  
let result = formatter.string(from: distance)
```

The workaround is the **as Unit** part – I’m pretty sure that will disappear in subsequent beta releases of iOS 10. You can change **.long** to be **.medium** or **.short** to see other ways of writing the same thing.

Wrap up

The new measurement system solves a small but important problem, and solves it pretty thoroughly. Its ability to blend units together in a single calculation, or convert one measurement to any other kind of unit, should mean there isn’t much it can’t do – it should be able to find a home in your app pretty easily!

Core Data

Among the many and varied APIs offered to us by Apple, Core Data is by far the one that offered the least attractive cost to benefit ratio. To be clear, that's not me saying it was bad, just that it had a pretty epic learning curve for something as fundamental to the Apple development ecosystem.

That's all changed in iOS 10. Apple has made a handful of changes that dramatically simplify learning and using Core Data, so if you're using it today you should be able to delete a huge amount of your code to make it leaner and more efficient. If you aren't using Core Data today – perhaps because you were intimidated by its complexity – then now is a good time to reconsider.

I created a full Core Data app from scratch in Hacking with Swift project 38, and there isn't much point me creating another whole project here. Instead, I want to walk you through what's changed and why, with code examples, so you can get started straight away.

Core Data before iOS 10

Using Core Data in iOS 9 and earlier required a number of steps:

1. Create entities that represent your data.
2. Convert those to **NSManagedObject** subclasses.
3. Load the finished model into an **NSManagedObjectModel**.
4. Create an **NSPersistentStoreCoordinator** object, responsible for reading from and writing to disk.
5. Set up an NSURL pointing to the database on disk where the saved objects live, e.g. Project38.sqlite.
6. Load that database into the **NSPersistentStoreCoordinator** so it knows where we want it to save.
7. Create an **NSManagedObjectContext** and point it at the persistent store coordinator.

And that was just to get your data loaded. Once you're actually up and running, you had to use code like below just to create new objects:

```
if let myObj =  
NSEntityDescription.insertNewObjectForEntityForName("MyObject",
```

```
inManagedObjectContext: managedObjectContext) as? MyObject {  
    // do stuff  
}
```

Because Core Data wasn't designed with generics in mind, you had to do a lot of typecasting. Code like this was the norm:

```
if let objects = try managedObjectContext.execute(fetch) as?  
[MyObject] {  
    // do stuff  
}
```

Once you actually had your project working, Core Data still didn't get much easier. It didn't take much work to trigger dangerous faults – when missing data needed to be fetched but was no longer available – and it took considerable thinking to ensure your architecture allowed high performance reads and writes.

Core Data in iOS 10

The existing Core Data stack you know (and perhaps love) continues to exist in iOS 10, but Apple has made sweeping changes to simplify it. If you're already using Core Data this gives you the chance to refactor your code, and probably delete a large chunk of it along the way; if you're not using Core Data this definitely makes the learning curve significantly lower.

Let's start with the biggest change: of the seven steps I listed to get Core Data up and running, only the first – “Create entities that represent your data” – is required in iOS 10. Xcode 8 now automatically generates **NSManagedObject** subclasses for you, and it does so at build time so you never even see them. You can still create as hard source files if you want by going to the same Editor > Create NSManagedObject Subclass option, and you can even control what kind of code is generated by changing the “Codegen” option in the model editor.

Most of the remaining steps are replaced by an all-new class that unifies loading the

managed object model, creating a persistent store, figuring out where the database is, then loading that into a managed object context. This new class – **NSPersistentContainer** – does all that work for you, and if you create one successfully then you can be guaranteed to have a valid managed object context to work with.

Let's look at this in code. Previously – i.e. in Swift 2.2 and iOS 9 – you would have written code like this:

```
func startCoreData() {  
    let modelURL =  
        NSBundle.mainBundle().URLForResource("CoreDataTest", withExtension:  
            "momd")!  
  
    let managedObjectModel = NSManagedObjectModel(contentsOfURL:  
        modelURL)!  
  
    let coordinator =  
        NSPersistentStoreCoordinator(managedObjectModel: managedObjectModel)  
  
    let url =  
        getDocumentsDirectory().URLByAppendingPathComponent("Project38.sqlite")  
  
    do {  
        try coordinator.addPersistentStoreWithType(NSSQLiteStoreType,  
            configuration: nil, URL: url, options: nil)  
  
        managedObjectContext =  
            NSManagedObjectContext(concurrencyType: .MainQueueConcurrencyType)  
  
        managedObjectContext.persistentStoreCoordinator = coordinator  
    } catch {  
        print("Failed to initialize the application's saved data")  
        return  
    }  
}
```

Note: **getDocumentsDirectory()** is our usual helper method.

Using iOS 10 and Swift 3 all that code disappears, and is replaced with this instead:

```
let container = NSPersistentContainer(name: "CoreDataTest")

container.loadPersistentStores { storeDescription, error in
    if let error = error {
        print("Unresolved error \(error)")
    }
}
```

When you do that, **NSPersistentContainer** will automatically look for the model names “CoreDataTest” in your application bundle, and load a persistent store for it in your application’s documents directory.

If the container loads correctly – i.e., you get no errors in the above code – you can be guaranteed that you have a persistent data store and managed object context available for you to use. In fact, the data store becomes a non-issue because you don’t really need to touch it any more – instead, just use the **viewContext** property of the container to get a managed object context for running queries. This context will run Core Data queries on the main queue, which makes it perfect for your user interface.

And how do those queries look? Beautiful, quite frankly. I already showed you this old code:

```
if let myObj =
NSEntityDescription.insertNewObjectForEntityForName("MyObject",
inManagedObjectContext: managedObjectContext) as? MyObject {
    // do stuff
}
```

Well, that’s all been swept away in iOS 10. Now the auto-generated **NSManagedObject** subclasses come with a built-in initializer that accepts a managed object context, like this:

```
let myObj = MyObject(context: container.viewContext)
```

That's it: no more optionality, no more stringly typed entity names, and under half the code too.

When it came to read objects with fetch requests, there was more string typing. When you wrote code like this:

```
let fetch = NSFetchedRequest(entityName: "MyObject")
```

...it meant that the **NSFetchRequest** instance had no idea of what object type it was working with, so you needed to typecast the results like this:

```
if let objects = try managedObjectContext.executeFetchRequest(fetch)  
as? [MyObject] {  
    // do stuff  
}
```

Again, that has been swept away in iOS 10: all **NSManagedObject** subclasses now have a **fetchRequest()** method that creates a type-safe **NSFetchRequest**, meaning that when you execute it you'll get an array of your specific object back, rather than something that needs to be typecasted.

Right now, this approach only works if you use explicit typing when creating your fetch request, although I presume that's a bug and I've filed a radar for it with Apple. Until it gets changed, you need to write code like this:

```
let fetch: NSFetchedRequest<MyObject> = MyObject.fetchRequest()
```

To execute that fetch request and get results back, call the **execute()** method on it directly, like this:

```
if let objects = try? fetch.execute() {  
    // do stuff  
}
```

Again, it's significantly shorter, and the addition of type safety is huge.

But wait... there's more!

Not only has Apple massively reduced the surface area of its Core Data APIs, but they also made two other major changes that make Core Data simpler and safer to use.

First, they added a connection pool for the SQL data store. Specifically, this means multiple concurrent requests can be working on the same database, which dramatically simplifies your code architecture. Think about it: every time you fault you can now go off and fetch the data you need without needing to worry about what other background tasks might be running. There is one proviso, but it's quite sensible: while you may have several connections *reading* data, you may only have one *write* active at any given time.

Second, and most impressively, Apple added a new technology called query generations – that's “generations” the noun like when you say “the new generation of students”. This is an entirely new approach that completely eliminates the problem of unfulfilled faults, which cause your app to crash because certain data no longer exists.

By default, Core Data in iOS 10 works just as it did in iOS 9: if you request data that got deleted, you'll get a crash. But if you want, you can now pin your data to a specific generation of data: you can say “no matter what happens, I want to use the data that's current as of this point in time.” So if you request some data and it has some faults, you can read those faults and never worry about them disappearing from under you. It's like a snapshot in time for queries: you can effectively say “run this query on my database as it was at 10am,” and it doesn't matter that at 10:05 all your data got deleted – as far as you're concerned, it still exists.

This is called “pinning”: you can pin your managed object context to a specific generation (identified using opaque tokens rather than specific times), or pin it to whatever is the latest generation. Once it’s pinned, it has a single, stable view of your data: all reads will see the same data until you choose to advance your position. Previously you would need to write code for preventative prefetching – i.e., aggressively fetching extra data that the user might want just in case it disappeared – but that just isn’t needed any more.

To pin your current context to the latest generation – which means “use the latest data as of right now, but if someone else moves it subsequently don’t let it affect me” – you would use code like this:

```
try
container.viewContext.setQueryGenerationFrom(NSQueryGenerationToken.current)
```

Of course, you can go ahead and make all the changes you want in your pinned generation. When you want to write those changes back to CoreData, call **mergeChanges()** on your managed object context, which will also move you to the latest generation. In doing this, your objects *won’t* automatically be refreshed to contain the latest data from the updates – if you want that to happen, call **refreshAllObjects()** on your managed object context to update any data in registered objects.

Wrap up

Core Data is still Core Data, but Apple has significantly lowered the barrier to entry. Rather than force you to jump through a variety of hoops just to get started, the new **NSPersistentContainer** class does almost all the work for you, and the addition of generics also go a long way to making your code easier to write and easier to read.

The addition of multiple concurrent read requests is great, because it’s something you benefit from immediately. However, query generations are the stand out new feature this year: it’s now trivial to create queries that are consistent with each other, which in turn makes your app simpler and more stable.

The combination of all these make iOS 10 the biggest release for Core Data in years, and genuinely has me excited to start using it again. Maybe Apple will fix Core Image in iOS 11...

Raw photography

iOS 10 revamps photography yet again, this time unleashing a whole range of new features with a whole new API: **AVCapturePhotoOutput**. This replaces the venerable **AVCaptureStillImageOutput**, and in doing so helps split up the code into smaller, easier to manage parts.

Before we start, a small piece of background information: a raw photo is the name for the unprocessed data delivered directly from a camera's sensor. It takes a lot of processing to convert a raw file to a JPEG, not least noise reduction, exposure adjustment, and sharpening, as well as the secret sauce Apple adds to make the JPEGs look great, so there are significant downsides to working with raw files. However, there are equally significant advantages: raw files contain far more information than JPEGs, and so can be adjusted in ways that would otherwise be impossible. If you dramatically under- or over-expose a picture, for example, you could fix that in the raw file pretty trivially; it's remarkable what you can do with them. It's a pretty common theme that beginners use JPEG, prosumers use raw, and professionals go back to JPEGs – because they know their camera well enough to ensure they get the picture right first time.

Setting up

To demonstrate raw photography we need to create a small app. Broadly I try to keep technique projects to be as simple as possible so that you can focus on the technology, but here we need to do a little bit of work to demonstrate **AVCapturePhotoOutput** fully.

Please create a new Single View Application named PaleoCam – it's the raw food diet of photography, after all. It doesn't matter whether you target iPhone or iPad, but you do need to have a physical device to hand because the simulator doesn't support using the camera.

We're going to create the user interface entirely in code because it's easier to explain, and we're going to start by creating a custom **UIView** subclass to handle previewing the camera output. This is a neat hack that makes it easy to wrap the **AVCaptureVideoPreviewLayer** camera previewing layer in a way that gives us Auto Layout support.

So, start by creating a new Cocoa Touch Class called "CapturePreviewView", making it a subclass of "UIView". This is going to have only one method: **layerClass()** is a class method – i.e. something that gets called on the class rather than instances of the class – that returns

what layer type should be used for drawing. It's **CALayer** by default, but we're going to make it return **AVCaptureVideoPreviewLayer** so that we get a camera preview layer.

If you wanted to make the **CapturePreviewView** more robust, you could add computed properties to it that exposed the most important parts of its **AVCaptureVideoPreviewLayer** layer more cleanly, but we're only using one of its properties once, so I'm not going to bother.

Here's the complete code for the **CapturePreviewView** class:

```
import AVFoundation
import UIKit

class CapturePreviewView: UIView {
    override class func layerClass() -> AnyClass {
        return AVCaptureVideoPreviewLayer.self
    }
}
```

Note the **import AVFoundation** line at the top – this is required to get the definition of **AVCaptureVideoPreviewLayer**.

We're done with the **CapturePreviewView** class, so please switch to **ViewController.swift**. We need to give this class three properties:

1. **session** will be an instance of the **AVCaptureSession** class, which is our app-wide configuration for using the camera.
2. **photoOutput** will be an instance of the **AVCapturePhotoOutput** class, which is what we use to take photos with a specific configuration.
3. **capturePreview** will hold our **CapturePreviewView** view.

Two of those can be created up front with an empty initializer, and the third is going to be a force unwrapped optional because we'll create it in **viewDidLoad()**. The first two require the AVFoundation framework, so please add these two imports now:

```
import AVFoundation
import Photos
```

We'll be using the Photos framework later, but there's no harm adding it now. Now that's done, add these three properties to the **ViewController** class:

```
let session = AVCaptureSession()
let photoOutput = AVCapturePhotoOutput()
var capturePreview: CapturePreviewView!
```

In the **viewDidLoad()** method we need to create the **CapturePreviewView()** instance and give it some Auto Layout constraints so that it takes up the full screen. We're also going to create a "Take Photo" button that sits at the bottom of the screen, that will trigger a method called **takePhoto()**.

There are two small complexities here:

1. We haven't created the **takePhoto()** method yet, so you'll get a compile error.
2. We need to connect our **session** property to the **session** property of the **AVCaptureVideoPreviewLayer** instance.

The first problem is fixed just by adding an empty **takePhoto()** method, like this:

```
func takePhoto() {
}
```

The second problem is fixed with some typecasting. Our **capturePreview** property is an instance of the **CapturePreviewView** class, which has a **layerClass()** method on it that makes it use **AVCaptureVideoPreviewLayer** rather than a plain **CALayer** for its layer type. However, we aren't storing that choice anywhere, which means if we read **capturePreview.layer** we'll get told that it's a **CALayer** even though really it's a subclass. To

fix this we'll just typecast `capturePreview.layer` as a `AVCaptureVideoPreviewLayer` before setting its `session` property.

Add this code to `viewDidLoad()` now:

```
// create a new capture preview
capturePreview = CapturePreviewView()
capturePreview.translatesAutoresizingMaskIntoConstraints = false
view.addSubview(capturePreview)

// make it fill the screen
capturePreview.topAnchor.constraint(equalTo: view.topAnchor).isActive
= true

capturePreview.bottomAnchor.constraint(equalTo:
view.bottomAnchor).isActive = true

capturePreview.leadingAnchor.constraint(equalTo:
view.leadingAnchor).isActive = true

capturePreview.trailingAnchor.constraint(equalTo:
view.trailingAnchor).isActive = true

// typecast its layer and connect its session to ours
(capturePreview.layer as! AVCaptureVideoPreviewLayer).session =
session

// create a new button
let takePhoto = UIButton(type: .custom)
takePhoto.translatesAutoresizingMaskIntoConstraints = false
takePhoto.setTitle("Take Photo", for: [])
takePhoto.setTitleColor(UIColor.red(), for: [])
view.addSubview(takePhoto)

// align it to 20 points up from the bottom
```

```
takePhoto.centerXAnchor.constraint(equalTo:  
view.centerXAnchor).isActive = true  
  
takePhoto.bottomAnchor.constraint(equalTo: view.bottomAnchor,  
constant: -20).isActive = true  
  
  
// make it call takePhoto() when tapped  
  
takePhoto.addTarget(self, action: #selector(takePhoto),  
for: .touchUpInside)
```

Before we're done with the set up, I'd like you to add two keys to the Info.plist file. First, add **NSCameraUsageDescription** and set its value to be "We need to take photos". Second, add **NSPhotoLibraryUsageDescription** then set its value to be "We need to save photos". The first one is used immediately because we use the camera; the second one will be used later because we're going to save photos to the camera roll.

Configuring a session

At this point the app won't do anything, but that's OK – we've put in place the framework for more development, so the next step is to configure a session for capturing photos.

Configuring a session means doing three things:

1. Giving it a sensible preset. In our case that means using **AVCaptureSessionPresetPhoto**, which prepares the session to take high-quality photos.
2. Creating a video capture device and adding it to the session.
3. Adding our existing **AVCapturePhotoOutput** property, **photoOutput**, to the session, so that we save photos.

The latter two of those will throw exceptions if they fail, and we also need to use the **canAddInput()** and **canAddOutput()** methods before even trying. However, there is one further complexity: if you're going to make multiple changes to a session, you should wrap the whole thing inside calls to **session.beginConfiguration()** and **session.commitConfiguration()** so that the changes are batched together. This ensures you never have half a change, and is also more efficient for the device.

We're going to create a new **configureSession()** method to handle all this. If any part of the method fails – if we can't add an input or output, or if doing so triggers an exception – then the method will commit whatever configuration was made so far and return false. Otherwise, it will commit everything then return true.

Before we dive into the (commented) code, there's one more thing: if you want to be able to capture high-resolution photos, you need to set **isHighResolutionCaptureEnabled** to true on your **AVCapturePhotoOutput** object. That doesn't actually capture high-resolution images, it just prepares the system to be able to do so. More on this later!

Add this method to the **ViewController** class now:

```
func configureSession() -> Bool {  
    // prepare to make changes  
    session.beginConfiguration()  
  
    // apply the most useful preset fo our needs  
    session.sessionPreset = AVCaptureSessionPresetPhoto  
  
    do {  
        // create a video capture device  
        let videoCaptureDevice =  
            AVCaptureDevice.defaultDevice(withMediaType: AVMediaTypeVideo)  
  
        // try creating a device input for the video capture device -  
        // note: this might throw an exception!  
        let videoDeviceInput = try AVCaptureDeviceInput(device:  
            videoCaptureDevice)  
  
        // check whether we can add this device input to our session  
        if session.canAddInput(videoDeviceInput) {  
            // we can - add it!  
            session.addInput(videoDeviceInput)
```

```

    } else {
        // we can't - escape!
        print("Failed to add video device input")
        session.commitConfiguration()
        return false
    }

    // check whether we can add our photo output
    if session.canAddOutput(photoOutput) {
        // we can - add it!
        session.addOutput(photoOutput)

        // request high-res photo support
        photoOutput.isHighResolutionCaptureEnabled = true
    } else {
        // we can't - escape!
        print("Failed to add photo output")
        session.commitConfiguration()
        return false
    }
}

} catch {
    // something went wrong - escape!
    print("Failed to create device input: \\" + error + "\\")
    session.commitConfiguration()
    return false
}

// if we made it here then everything went well - commit the
configuration
session.commitConfiguration()

```

```
// start the capture session  
session.startRunning()  
  
// return success  
return true  
}
```

At this point, you have enough code to run the app and see it sort of working – you should be able to see a live preview of the camera, at least.

Taking photos

We have a preview layer and we have a live stream of what the camera sees. All that's left now is to capture a snapshot when the “Take Photo” button is tapped. This is where the new iOS 10 code comes in, because it's time to put **AVCapturePhotoOutput** to work.

There are three stages to this process that we might care about:

1. When “Take Photo” is tapped, we're going to configure settings for the picture. Each picture must be configured from scratch – you can't re-use existing settings.
2. When we tell the **AVCapturePhotoOutput** object to take a photo, we'll get a callback saying that our settings have been evaluated and resolved. That might mean we got what we asked for, or it might mean iOS had to make changes – we find out here.
3. Finally, we get callbacks when the picture has been taken. There is one for raw photos, and another for JPEG photos – more on that soon.

WARNING: Apple makes it crystal clear that you cannot re-use existing photo settings; you need to configure each picture from scratch.

As you might imagine, all these callbacks mean that we need to use the delegate pattern, specifically **AVCapturePhotoCaptureDelegate**, so please add that to the class now. This protocol only has optional callbacks, however in practice several of them are required depending on what you're doing – if you don't implement them your app will just crash.

Completing stage 1 means filling in the `takePhoto()` method. This will create a new `AVCapturePhotoSettings` object, which is where all photo configuration is done, but it needs to do that using a raw photo pixel format. This is a cunning complexity of raw: every sensor is built differently, and you need to know what kind of sensor you're working with. Helpfully, Apple provides an array of all available pixel formats, and orders it so that the most useful – the one you will actually want – comes first.

Once you've created your `AVCapturePhotoSettings` object, you can configure it how you please. For example, we already requested support for high-resolution photos, but here's where we actually set one up. You can also request the flash to be on, off, or auto, you can ask for image stabilization, and so on. This is the difference between stage 1 and stage 2: stage 1 is where we request the settings we want (e.g. automatic flash), and stage 2 is where we get told which settings were resolved and selected (e.g. flash was not needed).

Once you have your settings ready, it's time to call `capturePhoto()` on the `AVCapturePhotoOutput` object. This takes your settings as its first parameter and a delegate as its second. Add this code to `takePhoto()` now:

```
let rawFormat =
    photoOutput.availableRawPhotoPixelFormatTypes.first!.uint32Value
let photoSettings = AVCapturePhotoSettings(rawPixelFormatType:
    rawFormat)

photoSettings.isHighResolutionPhotoEnabled = true
photoSettings.flashMode = .auto

photoOutput.capturePhoto(with: photoSettings, delegate: self)
```

Stage 2 is where you get a callback with what settings got selected. We're not going to use it here because it isn't needed, but if you want to experiment here's the method stub:

```
func capture(_ captureOutput: AVCapturePhotoOutput,
willBeginCaptureForResolvedSettings resolvedSettings:
    AVCaptureResolvedPhotoSettings) {
```

```
// update the UI?  
}
```

That just leaves stage 3: receiving the photo once it's taken. This is done using another delegate callback that gets sent the sample buffer straight from the sensor, a thumbnail version of the picture if you requested one (which we didn't), along with any settings or error information.

All we care about is pulling the raw file out of the sample buffer. iOS uses Adobe's Digital Negative format (DNG) for raw images, and there's a single method on **AVCapturePhotoOutput** that does all this work for us: **dngPhotoDataRepresentation()**.

Add this new method to **ViewController** now:

```
func capture(_ captureOutput: AVCapturePhotoOutput,  
didFinishProcessingRawPhotoSampleBuffer rawSampleBuffer:  
CMSampleBuffer?, previewPhotoSampleBuffer: CMSampleBuffer?,  
resolvedSettings: AVCaptureResolvedPhotoSettings, bracketSettings:  
AVCaptureBracketedStillImageSettings?, error: NSError?) {  
  
    // make sure we actually have a filled sample buffer  
    guard let rawSampleBuffer = rawSampleBuffer else { return }  
  
    // try to convert the sample buffer to DNG raw format  
    guard let data =  
    AVCapturePhotoOutput.dngPhotoDataRepresentation(forRawSampleBuffer:  
rawSampleBuffer, previewPhotoSampleBuffer: previewPhotoSampleBuffer)  
else { return }  
}
```

Note that both values need to be unwrapped safely, because they are optional.

And that's it: you have a DNG raw file to do with as you please. But what to do with it? You could enable iTunes documents sharing to let the user copy the files off so they can process them using something like Photoshop. Alternatively, you could use the new Core Image raw filters to open the raw file directly, manipulate it, then convert it to a **UIImage**.

This technique project isn't about Core Image, but I think it's worth examining just briefly. You can open a raw file by creating a **CIFilter** instance out of it, but you do need to give it the source type hint of "com.adobe.raw-image" so it knows to expect a DNG. Once that's done you can apply any Core Image manipulations however you want – for example the new **kCIIInputLuminanceNoiseReductionAmountKey** adjusts noise reduction – then create a **CGImage** or **UIImage** as needed.

For example, if you wanted to convert the **data** variable from the previous code into an image that gets saved to the user's camera roll, you would do something like this:

```
let rawOptions = [String(kCGImageSourceTypeIdentifierHint):  
"com.adobe.raw-image"]  
  
let context = CIContext(options: nil)  
let filter = CIFilter(imageData: data, options: rawOptions)  
  
if let result = filter?.outputImage {  
    if let cgImage = context.createCGImage(result, from:  
result.extent) {  
        let image = UIImage(cgImage: cgImage)  
        UIImageWriteToSavedPhotosAlbum(image, nil, nil, nil)  
    }  
}
```

Working with raw and JPEG

As you've now see, iOS 10 makes it easy to capture raw images using the new **AVCapturePhotoOutput** class. However, it adds a second pro feature at the same time: when you capture a photo, it can give you both the raw and the processed JPEG at the same time, so you get the best of both worlds.

To make this work, you need to modify the `takePhoto()` method so that it has a raw pixel format as well as a processed photo codec. Again, the array of processed codecs is sorted by usefulness, so you can just use `first!` to get back the most sensible option: JPEG.

To try out this double capture, start by changing the existing settings code in `takePhoto()` to this:

```
let rawFormat =  
    photoOutput.availableRawPhotoPixelFormatTypes.first!.uint32Value  
let processedFormat = photoOutput.availablePhotoCodecTypes.first!  
  
let photoSettings = AVCapturePhotoSettings(rawPixelFormatType:  
    rawFormat, processedFormat: [AVVideoCodecKey: processedFormat])
```

As you can see, the code for specifying which processed format codec to use is pretty ugly, but I'm afraid it's unavoidable.

Now that we're requesting two formats for each picture, two callback methods will be triggered when a photo is taken. We already have one in place for when a raw image is captured, and that will still be called as before. However, we'll now also get a subtly different callback for when the JPEG is automatically converted for us.

This new callback – `didFinishProcessingPhotoSampleBuffer` rather than `didFinishProcessingRawPhotoSampleBuffer` – works much the same: we need to safely unwrap the buffer, and safely convert it to JPEG data. However, once that's done we can immediately create a `UIImage` from it and save it to the user's photo album.

Here's an example implementation:

```
func capture(_ captureOutput: AVCapturePhotoOutput,  
            didFinishProcessingPhotoSampleBuffer photoSampleBuffer:  
            CMSampleBuffer?, previewPhotoSampleBuffer: CMSampleBuffer?,  
            resolvedSettings: AVCaptureResolvedPhotoSettings, bracketSettings:  
            AVCaptureBracketedStillImageSettings?, error: NSError?) {  
  
    guard let photoSampleBuffer = photoSampleBuffer else { return }  
}
```

```
guard let data =  
AVCapturePhotoOutput.jpegPhotoDataRepresentation(forJPEGSampleBuffer:  
photoSampleBuffer, previewPhotoSampleBuffer:  
previewPhotoSampleBuffer) else { return }  
  
if let image = UIImage(data: data) {  
    UIImageWriteToSavedPhotosAlbum(image, nil, nil, nil)  
}  
}
```

Wrap up

The ability to create and manipulate raw photos on-device could be a real game changer for iOS photography apps – of which there are probably thousands. If you’re not convinced of the benefits of raw images, try making a small app to manipulate the raw data using Core Image and you’ll see you can perform truly astonishing transformations on the image without sacrificing much quality.

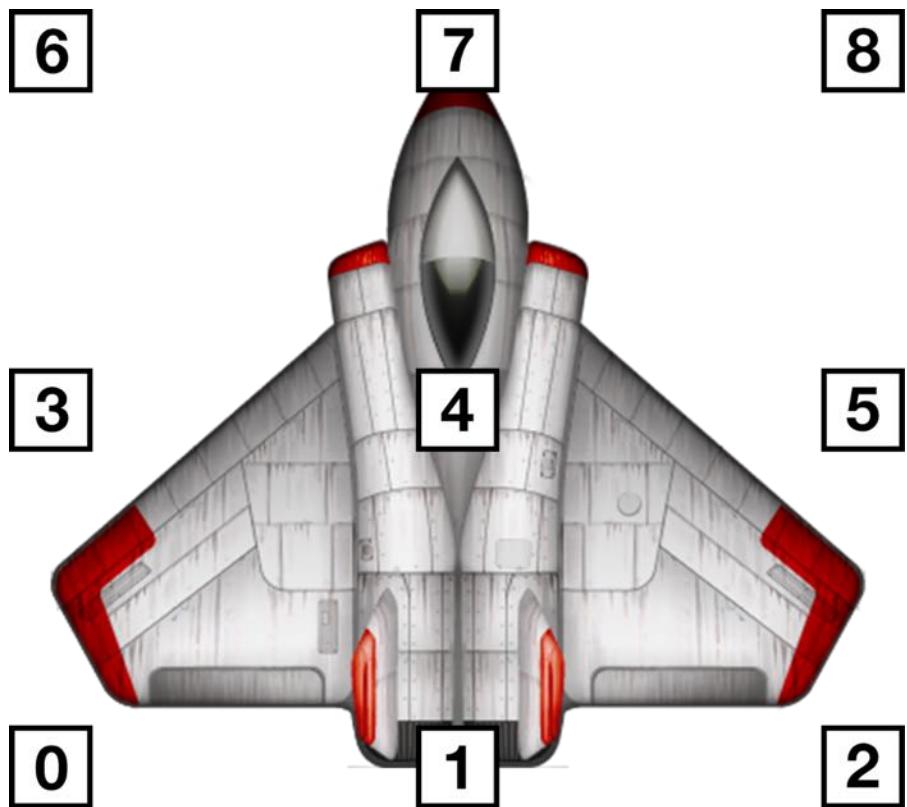
Warp Geometry

SpriteKit introduces a new way of manipulating the shapes of sprites on the screen: it divides the sprite up into small squares, then lets you stretch those small squares into different positions. The result is that you can warp a sprite in various ways – you can effectively pull different parts of it however you want.

Obviously there's a cost to doing this: a sprite is effectively one quad by default, which is a fancy way of saying it's a rectangle. Behind the scenes that becomes two triangles, pieced together to look like a single rectangle. If we split that single sprite into two rows and two columns, you end up with four times as many triangles; if you split it into 10 rows and 10 columns, you end up with a huge number of triangles just to represent one tiny sprite – your game will slow down massively.

Helpfully, SpriteKit employs a technique called automatic quad subdivision: even though you create two rows and two columns, it will automatically split them further if it needs to, allowing to warp transformation to look silky smooth.

We'll be working with a 2x2 warp here, which means two rows and two columns. That results in nine points in total: bottom left, bottom center, bottom right, left, center, right, top left, top center, and top right. They are positioned in a grid, as you can see in the picture below.



Warp transformations are specified as a grid of individual points.

When you create a warp transformation, you start by telling SpriteKit where to place its grid points. You then tell it where those same grid points are once the transformation has finished – like a before and after. These points should be positioned from 0 to 1, left to right and bottom to top.

For example, to give the right-hand grid point is at X: 1 (the right) Y: 0.5 (the center). If we wanted to make the sprite bulge outwards at that point, we could give it X: 2 Y: 0.5, meaning that it was twice as far out as it normally would be.

Making a warp transformation sandbox

It's all very well me showing you a picture of how warp transformation works, but the only way to master it is to set up your own transformation then try poking in various numbers.

Launch Xcode and create a new Game project called WarpTransformations. Make it target iPad and use SpriteKit for its game technology.

The default SpriteKit project comes with a spaceship picture that we will actually use for once, but it also comes with a lot of other junk. Open GameScene.sks and delete the “Hello World” label, then open GameScene.swift and remove everything but method stubs for **didMove()** and **touchesBegan()**, like this:

```
import SpriteKit

class GameScene: SKScene {
    override func didMove(to view: SKView) {

    }

    override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
    }
}
```

We’re going to start by displaying the SpriteKit spaceship in the center of the screen. Add a property to store it in:

```
var spaceship: SKSpriteNode!
```

Now add this code to **didMove()**:

```
spaceship = SKSpriteNode(imageNamed: "Spaceship")
addChild(spaceship)
```

That adds the spaceship to the scene, but it *does* prepare it for a warp transform. Sure, you can apply one later, but it won't animate – in order for SpriteKit to be able to animate a warp, the transform needs to have the same number of rows and columns as any existing transform. Right now we have *no* transform applied, so we're going to start by applying a warp transform that does nothing – it will leave the sprite exactly how it looks right now.

We'll need this transform a lot, because after every warp we're going to reset the sprite. So, we're going to create a property called `src` that contains the source positions in the grid as an array. As I said already, this needs to be nine positions between X: 0.0 Y: 0.0 and X: 1.0 Y: 1.0, specify exactly where each point is on the sprite. As this is our “before” grid, we'll be spacing the points evenly. Each point needs to be a `vector_float2`, which is an X and Y coordinate pair.

Add this property now:

```
var src = [
    // bottom row: left, center, right
    vector_float2(0.0, 0.0),
    vector_float2(0.5, 0.0),
    vector_float2(1.0, 0.0),

    // middle row: left, center, right
    vector_float2(0.0, 0.5),
    vector_float2(0.5, 0.5),
    vector_float2(1.0, 0.5),

    // top row: left, center, right
    vector_float2(0.0, 1.0),
    vector_float2(0.5, 1.0),
    vector_float2(1.0, 1.0)
]
```

I added comments in there so you can hopefully see how all nine points are positioned. For example, the middle point is X: 0.5 Y: 0.5, meaning that it's in the center of both axes.

In order to complete our setup, we're going to create a warp geometry that uses the `src` positions twice: the same for before and after. This means it will look identical to a sprite without a warp geometry attached, but it also means that when we apply a new geometry with the same number of rows and columns SpriteKit will be able to animate the change.

Add this code to the end of `didMove()`:

```
let warp = SKWarpGeometryGrid(columns: 2, rows: 2, sourcePositions:  
src, destPositions: src)  
spaceship.warpGeometry = warp
```

Stretch and squeeze

At this point we have a spaceship sprite visible on screen, and it has a warp geometry attached – albeit one that does nothing just yet. To bring that to life we're going to fill in the `touchesBegan()` method so that every time the user taps the screen we warp the spaceship in different ways.

First, add this property to the `GameScene` class:

```
var warpType = 0
```

Each time `touchesBegan()` is called we'll add one to that up to a maximum of 3, and use it to decide which transformation to apply.

The initial version of `touchesBegan()` is just going to repeat what we have already: it will create a warp transformation with the same source and destination transforms. It will, however, lay the groundwork for doing more advanced things in a few minutes, and it gives me a chance to introduce you to an important new `SKAction` initializer: `animate()`.

In `didMove()` we added this line of code:

```
spaceship.warpGeometry = warp
```

That immediately updates our spaceship to have the specified warp transformation, which is fine when you want something to snap into place. But if you want an animation to happen – even source point moving to match its destination point – then you need to create an **SKAction** using its `animate()` transition. This takes an array of warp geometries to apply, along with an array of times to use when animating to those geometries. Unusually, the times are specified cumulatively, so if you specify **[4, 8]** it doesn't mean the first animation will take four seconds and the second animation will take eight seconds. Instead, they will both take four seconds, because there's a four-second gap between the two numbers.

Using the `animate()` initializer returns an optional **SKAction** because it might fail, so we'll unwrap it carefully.

Here's the first draft of the `touchesBegan()` method – it has a placeholder **switch** block in there that we'll be filling in later:

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {  
    // create a destination transform by copying the source transform  
    var dst = src  
  
    // this is where we'll be modifying the destination transform  
    switch warpType {  
        default:  
            break  
    }  
  
    // create a new warp geometry by mapping from src to dst  
    let newWarp = SKWarpGeometryGrid(columns: 2, rows: 2,  
        sourcePositions: src, destPositions: dst)
```

```

    // pull out the existing warp geometry so we have something to
    animate back to

    let oldWarp = spaceship.warpGeometry!

    // try to create an SKAction with these two warps; each will
    animate over 0.5 seconds

    if let action = SKAction.animate(withWarps: [newWarp, oldWarp],
    times: [0.5, 1]) {

        // run it on the spaceship sprite
        spaceship.run(action)

    }

    // add 1 to the warp type so that we get a different
    transformation next time

    warpType += 1

    // if we're higher than 3 wrap back to 0
    if warpType > 3 {

        warpType = 0
    }
}

```

That's a whole lot more code, but it *still* won't do anything. In **didMove()** we create the initial warp transformation so that the sprite is ready to be animated. Here in **touchesBegan()** we're creating a new warp geometry that is identical to the existing one, and we're animating to it – so again, nothing happens.

However, we're now in the perfect position to change that with only a small amount of code. We have a **switch** block that is ready to start manipulating the **dst** array based on the value of **warpType**, so let's add a new case now.

The first case we're going to add is for when **warpType** is 0. We're going to make it stretch

the nose of the spaceship upwards, and it should animate back to its starting point too.

To do that, add this **case** to the **switch** block:

```
case 0:  
    // stretch the nose up  
    dst[7] = vector_float2(0.5, 1.5)
```

The finished **switch** block should look like this:

```
switch warpType {  
case 0:  
    // stretch the nose up  
    dst[7] = vector_float2(0.5, 1.5)  
default:  
    break  
}
```

What that's doing is adjusting destination grid position 7 - the top center point - so that it is now X: 0.5 Y: 1.5. It was at Y: 1.0, so this means it will stretch further up along the Y axis.

Go ahead and run the code now to make sure it's working – you should see the spaceship as before, but clicking the screen should cause its nosecone to stretch up then back again.

Let's add another case:

```
case 1:  
    // stretch the wings down  
    dst[0] = vector_float2(0, -0.5)  
    dst[2] = vector_float2(1, -0.5)
```

That changes the first and third points – the bottom left and bottom right corners – so that their Y position moves from 0 to -0.5, pulling them downwards.

Here's another one, more complicated:

```
case 2:  
    // squash the ship vertically  
    dst[0] = vector_float2(0.0, 0.25)  
    dst[1] = vector_float2(0.5, 0.25)  
    dst[2] = vector_float2(1.0, 0.25)  
  
    dst[6] = vector_float2(0.0, 0.75)  
    dst[7] = vector_float2(0.5, 0.75)  
    dst[8] = vector_float2(1.0, 0.75)
```

That pushes the bottom three points up while also pulling the top three points down, causing the spaceship to look squashed.

And finally, we can tell the six points that are off the horizontal center to have the opposite positions they would normally have, causing the spaceship to look like it's turning over:

```
case 3:  
    // flip it left to right  
    dst[0] = vector_float2(1.0, 0.0)  
    dst[2] = vector_float2(0.0, 0.0)  
  
    dst[3] = vector_float2(1.0, 0.5)  
    dst[5] = vector_float2(0.0, 0.5)
```

```
dst[6] = vector_float2(1.0, 1)
dst[8] = vector_float2(0.0, 1)
```

Wrap up

Now that you have a simple sandbox in place, I encourage you to spend a little time toying around with it to see what effects you can achieve. You know where each point lies on the grid, so try things out: squeeze the plane horizontally, make its wings stretch out, make an **SKAction** that animates between several transforms, and so on – it's the only way to really feel like you understand it.