HACKING WITH SWIFT

PROJECTS 1-39

Learn to make iOS apps with real projects

Paul Hudson

Contents

About this book

The **Hacking with Swift** tutorial series is designed to make it easy for beginners to get started coding for iPad and iPhone using the Swift programming language.

My teaching method skips out a lot of theory. It skips out the smart techniques that transform 20 lines of easy-to-understand code into 1 line of near-magic. It ignores coding conventions by the dozen. And perhaps later on, once you've finished, you'll want to go back and learn all the theory I so blithely walked past. But let me tell you this: the problem with learning theory by itself is that your brain doesn't really have any interest in remembering stuff just for the sake of it.

You see, here you'll be learning to code on a Need To Know basis. Nearly everything you learn from me will have a direct, practical application to something we're working on. That way, your brain can see exactly why a certain technique is helpful and you can start using it straight away.

This book has been written on the back of my personal motto: "Programming is an art. Don't spend all your time sharpening your pencil when you should be drawing." We'll be doing some "sharpening" but a heck of a lot more "drawing" – if that doesn't suit your way of learning, you should exit now.

The three golden rules

The series is crafted around a few basic tenets, and it's important you understand them before continuing:

1. Follow the series: The tutorials are designed to be used in order, starting at the beginning and working through to the end. The reason for this is that concepts are introduced sequentially on a need-to-know basis – you only learn about something when you really have to in order to make the project work.
2. Don't skip the games and techniques: The tutorials follow a sequence: app, game, technique, app, game, technique, etc. That is, you develop an app, then you develop a game, then we focus on a particular iOS component together to help make your apps better. The apps and games are, of course, standalone projects that you can go on to develop as

you wish, whereas the technique tutorials will often be used to improve or prepare you for other projects.

3. Get ready to hack: This is not designed to be the one-stop learning solution for all your Swift needs. It's called "Hacking with Swift" because the goal of each project is to reach the end with as little complication as possible – we're hacking, or playing around, with the language, not trying to give you Comp Sci 101.

I can't re-iterate that last point enough. What I have found time and time again is that any tutorial, no matter how carefully written or what audience it's aimed at, will fail to fit the needs of many possible readers. And these people get angry, saying how the tutorial is wrong, how the tutorial is lame, how their tutorial would be much better if only they had the time to write it, and so on.

Over the last 12 years of writing, I have learned to ignore minority whinging and move on, because what matters is that this tutorial is useful to *you*.

You'd be surprised by how many people think the path to success is through reading books, attending classes or, well, doing pretty much anything except sitting down in front of a computer and typing. Not me. I believe the best way to learn something is try to it yourself and see how it goes.

Sure, going to classes might re-enforce what you've learned, or it might teach you some time-saving techniques, but ultimately I've met too many people with computing degrees who stumble when asked to write simple programs. Don't believe me? Try doing a Google search for "fizz buzz test", and you'll be surprised too.

So, dive in, make things, and please, please, please have fun – because if you're not enjoying yourself, Swift coding probably isn't for you.

To download the files for any Hacking with Swift project, or if you'd like to learn more about the series, visit the series website and see the full range: **hackingwithswift.com**.

If you spot any errors in this book, either typos or technical mistakes, please do let me know so I can correct them as soon as possible. The best way to get in touch is on Twitter **@twostraws**, but you can also email **paul@hackingwithswift.com**.

Xcode, Swift and iOS

I'm not going to talk much, because I want to get straight into coding. However, there are some points you do need to know:

 • You should install the latest Xcode from the Mac App Store. It's free, and includes everything you need to make iOS apps in the iOS Simulator. Most of the projects in this series will be developed in the simulator, but a couple will require a device because the technology isn't available in the simulator – things like Touch ID and the accelerometer, for example. Projects that require a device also require you to have an active iOS developer account with Apple so that you can deploy your project to a device.
 • Swift is a relatively new language, and is evolving quickly. Every new release of Xcode seems to change something or other, and often that means code that used to work now no longer does. At the time of writing, Swift is mature enough that the changes are relatively minor, so hopefully you can make them yourself. If not, check to see if there's an update of the project files on **hackingwithswift.com**.
 • These projects are designed to work with iOS 9.0 or later, which is the version that runs on the majority of devices. You can downgrade them to 7.0 with relatively few changes if you desperately want to maximise your reach, but it's really not worth it at this point.

Important note: if any bugs are found in the project files, or if Swift updates come out that force syntax changes, I'm going to be making changes to the projects on the website and updating this book as needed. Please make sure you read the release notes for each project to see what's changed, and **follow me on Twitter @twostraws** if you want to be notified of updates.

I'm also happy to answer questions on Twitter if you encounter problems, so please feel free to get in touch!

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.

Dedication

This book is dedicated to my daughter Charlotte, aka "Bonk", who has provided lots of hugs and lots of happiness at every point in its creation.

Introduction
Swift for Complete Beginners

If you want to learn the language all at once before you start making apps, this is for you.

How to install Xcode and create a playground

Xcode is Apple's programming application for developers. It's free from the Mac App Store, and it's required to do iPhone and iPad development. So, your first action is to **click here to install Xcode from the Mac App Store** – it's quite a big download, so start downloading it now and carry on reading.

While that's downloading, I can explain a couple of the absolute basics to you:

 • **iOS** is the name of the operating system that runs on all iPhones and iPads. It's responsible for all the basic operations of the phone, such as making phone calls, drawing on the screen, and running apps.
 • **Swift** is Apple's modern programming language that lets you write apps for iOS. It contains the functionality for building programs, but doesn't handle anything like user interfaces, audio or networking.
 • **Swift 1.2** was the first major update to Swift, tweaking various language features and improving others.
 • **Swift 2** is the second major update to Swift, and it's the version used across all of Hacking with Swift.
 • **UIKit** is Apple's user interface toolkit. It contains things like buttons, text boxes, navigation controls and more, and you drive it using Swift.
 • **Cocoa Touch** is the name given for Apple's vast collection of frameworks for iOS. It includes UIKit to do user interfaces, but also SpriteKit for making 2D games, SceneKit for making 3D games, MapKit for maps, Core Graphics for drawing, Core Animation for animating things, iAd for placing adverts, and much more.
 • **NeXTSTEP** is an operating system created by a company that Steve Jobs founded called NeXT. It was bought by Apple, at which point Jobs was placed back in control of the company, and put NeXTSTEP technology right into the core of Apple's development platform.
 • **iOS Simulator** is a tool that comes with Xcode that looks and works almost exactly like a real iPhone or iPad. It lets you test your app very quickly without having to use a real device.
 • **Playgrounds** are miniature Swift testing environments that let you type code and see the results immediately. You don't build real apps with them, but they are great for learning. We'll be using playgrounds in this introduction.
 • **Crashes** are when your code goes disastrously wrong and your app cannot recover. If a user is running your app it will just disappear and they'll be back on the home screen. If you're running in Xcode, you'll see a crash report.

• **Taylor Swift** has nothing to do with the Swift programming language. This is a shame, as you might imagine, but I'll try to make up for this shortfall by using her songs in this tutorial. Deal with it.

That's it for the basics – if Xcode still hasn't finished downloading then why not watch some Taylor Swift videos while you wait? The examples in this tutorial will certainly make a lot more sense…

Variables and constants

Every useful program needs to store data at some point, and in Swift there are two ways to do it: variables and constants. A variable is a data store that can have its value changed whenever you want, and a constant is a data store that you set once and can never change. So, variables have values that can vary, and constants have values that are constant – easy, right?

Having both these options might seem pointless, after all you could just create a variable then never change it – why does it need to be made a constant? Well, it turns out that many programmers are – shock! – less than perfect at programming, and we make mistakes. One of the advantages of separating constants and variables is that Xcode will tell us if we've made a mistake. If we say, "make this date a constant, because I know it will never change" then 10 lines later try to change it, Xcode will refuse to build our app.

Constants are also important because they let Xcode make decisions about the way it builds your app. If it knows a value will never change, it is able to apply optimizations to make your code run faster.

In Swift, you make a variable using the **var** keyword, like this:

```
var name = "Tim McGraw"
```

Let's put that into a playground so you can start getting feedback. Delete everything in there apart from the **import UIKit** line (that's the bit that pulls in Apple's core iOS framework and it's need later on), and add that variable. You should see the picture below.
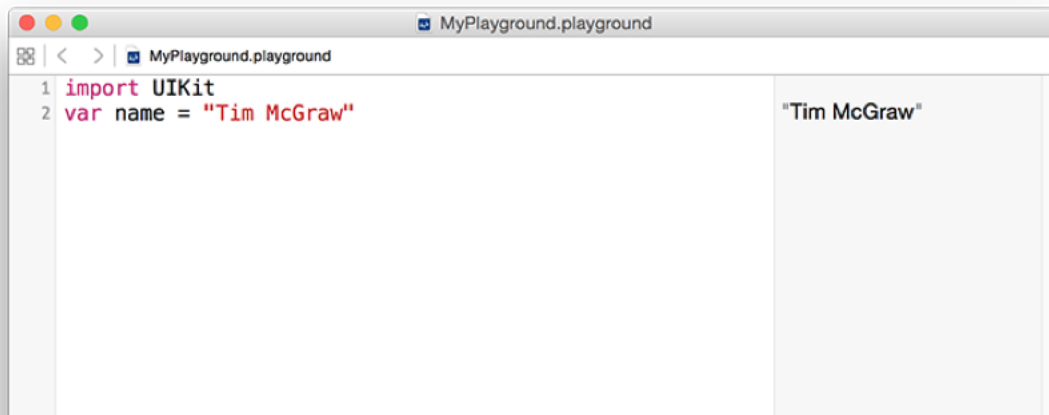
In Xcode playgrounds, you type your code on the left and see results on the right a second later.

Because this is a variable, you can change it whenever you want, but you shouldn't use the **var** keyword each time – that's only used when you're declaring new variables. Try writing this:

```
var name = "Tim McGraw"
name = "Romeo"
```

So, the first line creates the **name** variable and gives it an initial value, then the second line updates the **name** variable so that its value is now "Romeo". You'll see both values printed in the results area of the playground.

Now, what if we had made that a constant rather than a variable? Well, constants use the **let** keyword rather than **var**, so you can change your first line of code to say **let name** rather than **var name** like this:

```
import UIKit
let name = "Tim McGraw"
name = "Romeo"
```

But now there's a problem: Xcode is showing a red warning symbol next to line three, and it

should have drawn a squiggly underline underneath **name**. If you click the red warning symbol, Xcode will tell you the problem: "Cannot assign to 'let' value 'name'" – which is Xcode-speak for "you're trying to change a constant and you can't do that."
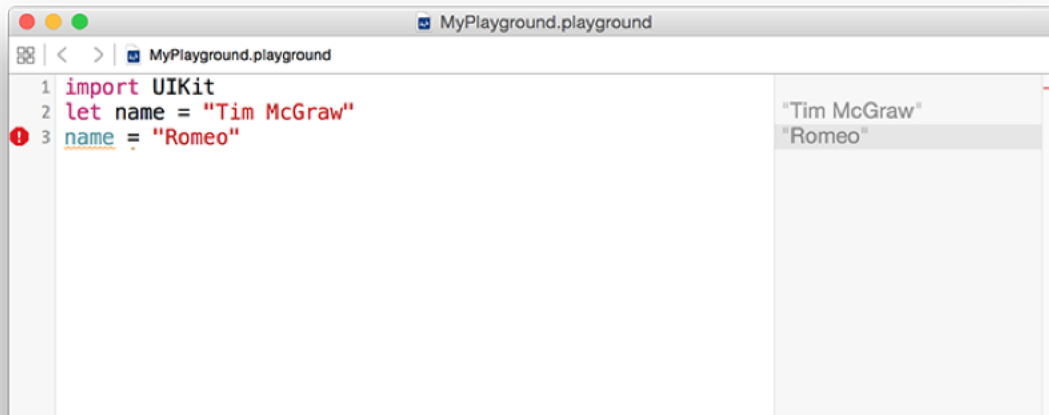
If you try to change a constant in Swift, Xcode will refuse to build your app.

So, constants are a great way to make a promise to Swift and to yourself that a value won't change, because if you do try to change it Xcode will refuse to run. Swift developers have a strong preference to use constants wherever possible because it makes your code easier to understand. In fact, in the very latest versions of Swift, Xcode will actually tell you if you make something a variable then never change it!

Important note: variable and constant names must be unique in your code. You'll get an error if you try to use the same variable name twice, like this:

```
var name = "Tim McGraw"
var name = "Romeo"
```

If the playground finds an error in your code, it will either flag up a warning in a red box, or will just refuse to run. You'll know if the latter has happened because the text in the results pane has gone gray rather than its usual black.

Types of Data

There are lots of kinds of data, and Swift handles them all individually. You already saw one of the most important types when you assigned some text to a variable, but in Swift these are called a **String** – literally a string of characters.

Strings can be long (e.g. a million letters or more), short (e.g. 10 letters) or even empty (no letters), it doesn't matter: they are all strings in Swift's eyes, and all work the same. Swift knows that **name** should hold a string because you assign a string to it when you create it: "Tim McGraw". If you were to rewrite your code to this it would stop working:

```
var name

name = "Tim McGraw"
```

This time Xcode will give you an error message that won't make much sense just yet: "Type annotation missing in pattern". What it means is, "I can't figure out what data type **name** is because you aren't giving me enough information."

At this point you have two options: either create your variable and give it an initial value on one line of code, or use what's called a type annotation, which is where you tell Swift what data type the variable will hold later on, even though you aren't giving it a value right now.

You've already seen how the first option looks, so let's look at the second: type annotations. We know that **name** is going to be a string, so we can tell Swift that by writing a colon then **String**, like this:

```
var name: String

name = "Tim McGraw"
```

You'll have no errors now, because Swift knows what type of data **name** will hold in the future.

Note: some people like to put a space before and after the colon, making **var name : String**, but they are *wrong* and you should try to avoid mentioning their wrongness in polite

company.

The lesson here is that Swift always wants to know what type of data every variable or constant will hold. Always. You can't escape it, and that's a good thing because it provides something called type safety – if you say "this will hold a string" then later try and put a rabbit in there, Swift will refuse.

We can try this out now by introducing another important data type, called **Int**, which is short for "integer." Integers are round numbers like 3, 30, 300, or -16777216. For example:

```
var name: String
name = "Tim McGraw"


var age: Int
age = 25
```

That declares one variable to be a string and one to be an integer. Note how both **String** and **Int** have capital letters at the start, whereas **name** and **age** do not – this is the standard coding convention in Swift. A coding convention is something that doesn't matter to Swift (you can write your names how you like!) but does matter to other developers. In this case, data types start with a capital letter, whereas variables and constants do not.

Now that we have variables of two different types, you can see type safety in action. Try writing this:

```
name = 25
age = "Time McGraw"
```

In that code, you're trying to put an integer into a string variable, and a string into an integer variable – and, thankfully, Xcode will throw up errors. You might think this is pedantic, but it's actually quite helpful: you make a promise that a variable will hold one particular type of data, and Xcode will enforce that throughout your work.

Before you go on, please delete those two lines of code causing the error, otherwise nothing in your playground will work going forward!

Float and Double

Let's look at two more data types, called **Float** and **Double**. This is Swift's way of storing numbers with a fractional component, such as 3.1, 3.141, 3.1415926, and -16777216.5. There are two data types for this because you get to choose how much accuracy you want, but most of the time it doesn't matter so the official Apple recommendation is always to use **Double** because it has the highest accuracy.

Try putting this into your playground:

```
var latitude: Double
latitude = 36.166667

var longitude: Float
longitude = -86.783333
```

You can see both numbers appear on the right, but look carefully because there's a tiny discrepancy. We said that **longitude** should be equal to -86.783333, but in the results pane you'll see -86.78333 – it's missing one last 3 on the end. Now, you might well say, "what does 0.000003 matter amongst friends?" but this is ably demonstrating what I was saying about accuracy.

Because these playgrounds update as you type, we can try things out so you can see exactly how **Float** and **Double** differ. Try changing the code to be this:
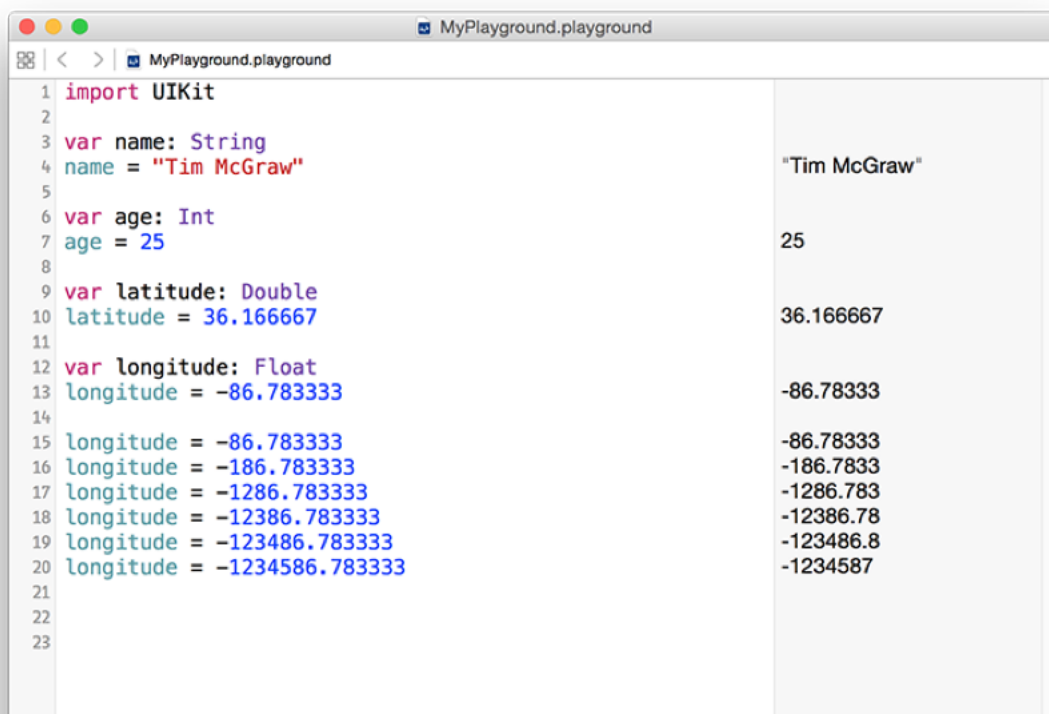
```
var longitude: Float
longitude = -86.783333
longitude = -186.783333
longitude = -1286.783333
longitude = -12386.783333
```

```
longitude = -123486.783333
longitude = -1234586.783333
```

That's adding increasing numbers before the decimal point, while keeping the same amount of numbers after. But if you look in the results pane you'll notice that as you add more numbers before the point, Swift is removing numbers after. This is because it has limited space in which to store your number, so it's storing the most important part first – being off by 1,000,000 is a big thing, whereas being off by 0.000003 is less so.

In Swift a Float holds much less data than a Double, so you should use Double where possible.

Now try changing the **Float** to be a **Double** and you'll see Swift prints the correct number out every time:

```
var longitude: Double
```

This is because, again, **Double** has twice the accuracy of **Float** so it doesn't need to cut your

number to fit. Doubles still have limits, though – if you were to try a massive number like 123456789.123456789 you would see it gets cut down to 123456789.1234568.

Boolean

Swift has a built-in data type that can store whether a value is true or false, called a **Bool**, short for Boolean. Bools don't have space for "maybe" or "perhaps", only absolutes: true or false. For example:

```
var stayOutTooLate: Bool
stayOutTooLate = true


var nothingInBrain: Bool
nothingInBrain = true


var missABeat: Bool
missABeat = false
```

Using type annotations wisely

As you've learned, there are two ways to tell Swift what type of data a variable holds: assign a value when you create the variable, or use a type annotation. If you have a choice, the first is always preferable because it's clearer. For example:

```
var name = "Tim McGraw"
```

…is preferred to:

```
var name: String
name = "Tim McGraw"
```

This applies to all data types. For example:

```
var age = 25
var longitude = -86.783333
var nothingInBrain = true
```

This technique is called *type inference*, because Swift can infer what data type should be used for a variable by looking at the type of data you want to put in there. When it comes to numbers like -86.783333, Swift will always infer a **Double** rather than a **Float**.

For the sake of completeness, I should add that it's possible to specify a data type and provide a value at the same time, like this:

```
var name: String = "Tim McGraw"
```

Operators

Operators are those little symbols you learned in your very first math classes: + to add, - to subtract, * to multiply, / to divide, = to assign value, and so on. They all exist in Swift, along with a few extras.

Let's try a few basics – please type this into your playground:

```
var a = 10
a = a + 1
a = a - 1
a = a * a
```

In the results pane, you'll see 10, 11, 10 and 100 respectively. Now try this:

```
var b = 10
b += 10
b -= 10
```

+= is an operator that means "add then assign to." In our case it means "take the current value of **b**, add 10 to it, then put the result back into **b**." As you might imagine, -= does the same but subtracts rather than adds. So, that code will show 10, 20, 10 in the results pane.

Some of these operators apply to other data types. As you might imagine, you can add two doubles together like this:

```
var a = 1.1
var b = 2.2
var c = a + b
```

When it comes to strings, + will join them together. For example:

```swift
var name1 = "Tim McGraw"
var name2 = "Romeo"
var both = name1 + " and " + name2
```

That will write "Tim McGraw and Romeo" into the results pane.

Comparison operators

Swift has a set of operators that perform comparisons on values. For example:

```swift
var a = 1.1
var b = 2.2
var c = a + b


c > 3
c >= 3
c > 4
c < 4
```

That shows off greater than (>), greater than or equal (>=), and less than (<). In the results window you'll see true, true, false, true – these are Booleans, because the answer to each of these statements can only ever be true or false.

If you want to check for equality, you can't use = because already has a meaning: it's used to give a variable a value. So, Swift has an alternative in the form of ==, meaning "is equal to." For example:

```
var name = "Tim McGraw"

name == "Tim McGraw"
```

That will show "true" in the results pane. Now, one thing that might catch you out is that in Swift strings are case-sensitive, which means "Tim McGraw", "TIM MCGRAW" and "TiM mCgRaW" are all considered different. If you use == to compare two strings, you need to make sure they have the same letter case.

There's one more operator I want to introduce you to, and it's called the "not" operator: !. Yes, it's just an exclamation mark. This makes your statement mean the opposite of what it did. For example:

```
var stayOutTooLate = true

stayOutTooLate

!stayOutTooLate
```

That will print out true, true, false – with the last value there because it flipped the previous true.

You can also use ! with = to make != or "not equal". For example:

```
var name = "Tim McGraw"

name == "Tim McGraw"

name != "Tim McGraw"
```

String interpolation

This is a fancy name for what is actually a very simple thing: combining variables and constants inside a string.

Clear out all the code you just wrote and leave only this:

```
var name = "Tim McGraw"
```

If we wanted to print out a message to the user that included their name, string interpolation is what makes that easy: you just write a backslash, then an open parenthesis, then your code, then a close parenthesis, like this:

```
var name = "Tim McGraw"
"Your name is \(name)"
```

The results pane will now show "Your name is Tim McGraw" all as one string, because string interpolation combined the two for us.

Now, we could have written that using the + operator, like this:

```
var name = "Tim McGraw"
"Your name is " + name
```

…but that's not as efficient, particularly if you're combining multiple variables together. In addition, string interpolation in Swift is smart enough to be able to handle a variety of different data types automatically. For example:

```
var name = "Tim McGraw"
var age = 25
```

```
var latitude = 36.166667
```

```
"Your name is \(name), your age is \(age), and your latitude is \
(latitude)"
```

Doing that using + is much more difficult, because Swift doesn't let you add integers and doubles to a string.

At this point your result may no longer fit in the results pane, so either resize your window or hover over the result and click the + button that appears to have it shown inline.

One of the powerful features of string interpolation is that everything between **\(** and **)** can actually be a full Swift expression. For example, you can do mathematics in there using operators, like this:

```
var age = 25
```

```
"You are \(age) years old. In another \(age) years you will be \(age
* 2)."
```

Arrays

Arrays let you group lots of values together into a single collection, then access those values by their position in the collection. Swift uses type inference to figure out what type of data your array holds, like so:

```
var oddNumbers = [2, 4, 6, 8]
var songs = ["Shake it Off", "You Belong with Me", "Back to December"]
```

As you can see, Swift uses brackets to mark the start and end of an array, and each item in the array is separated with a comma.

When it comes to reading items out an array, there's a catch: Swift starts counting at 0. This means the first item is 0, the second item is 1, the third is 2, and so on. Try putting this into your playground:

```
var songs = ["Shake it Off", "You Belong with Me", "Back to December"]
songs[0]
songs[1]
songs[2]
```

That will print "Shake it Off", "You Belong with Me", and "Back to December" in the results pane.

An item's position in an array is called its index, and you can read any item from the array just by providing its index. However, you do need to be careful: our array has three items in, which means indexes 0, 1 and 2 work great. But if you try and read **songs[3]** your playground will stop working – and if you tried that in a real app it would crash!

Because you've created your array by giving it three strings, Swift knows this is an array of strings. You can confirm this by using a special command in the playground that will print out the data type of any variable, like this:

```
var songs = ["Shake it Off", "You Belong with Me", "Back to
December"]

songs.dynamicType
```

That will print **Swift.Array<Swift.String>** into the results pane, telling you that Swift considers **songs** to be an array of strings.

Let's say you made a mistake, and accidentally put a number on the end of the array. Try this now and see what the results pane prints:

```
var songs = ["Shake it Off", "You Belong with Me", "Back to
December", 3]

songs.dynamicType
```

This time you'll see something else: **Swift.Array<NSObject>**. If you get an error saying "Type of expression is ambiguous" you probably deleted the **import UIKit** line by accident. Think back to the absolute basics I outlined while you were downloading Xcode: one of the things I mentioned was a company that Steve Jobs founded called NeXT, which made an operating system called NeXTSTEP, which in turn got bought by Apple and placed at the core of its development platform.

Well, here's the proof: now that our array holds both strings and numbers, Swift is behind the scenes converting the array to hold the data type **NSObject** – literally "an object from NeXTSTEP." This is the iOS way of saying "any kind of data," because the NeXT technologies are so deeply embedded in the Apple platform that literally everything builds on it.

Type safety is important, and although it's neat that Swift is trying to be helpful by making our array hold any kind of data, this particular case was an accident. Fortunately, I've already said that you can use type annotations to specify exactly what type of data you want an array to store. To specify the type of an array, write the data type you want to store with brackets around it, like this:

```
var songs: [String] = ["Shake it Off", "You Belong with Me", "Back to
December", 3]
```

Now that we've told Swift we want to store only strings in the array, it will refuse to run the
code because it has a 3 on the end.

Creating arrays

If you make an array using the syntax shown above, Swift creates the array and fills it with
the values we specified. Things aren't quite so straightforward if you want to create the array
then fill it later – this syntax doesn't work:

```
var songs: [String]
songs[0] = "Shake it Off"
```

The reason is one that will seem needlessly pedantic at first, but has deep underlying
performance implications so I'm afraid you're just stuck with it. Put simply, writing **var songs:
[String]** tells Swift "the **songs** variable will hold an array of strings," but *it doesn't actually
create that array*. It doesn't allocate any RAM, or do any of the work to actually create a Swift
array. It just says that at some point there will be an array, and it will hold strings.

There are a few ways to express this correctly, and the one that probably makes most sense
at this time is this:

```
var songs: [String] = []
```

That uses a type annotation to make it clear we want an array of strings, and it assigns an
empty array (that's the **[]** part) to it.

You'll also commonly see this construct:

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

That means the same thing: the **()** tells Swift we want to create the array in question, which is then assigned to **songs** using type inference. This option is two characters shorter, so it's no surprise programmers prefer it!

Array operators

You can use a limited set of operators on arrays. For example, you can merge two arrays by using the + operator, like this:

```
var songs = ["Shake it Off", "You Belong with Me", "Love Story"]
var songs2 = ["Today was a Fairytale", "White Horse", "Fifteen"]
var both = songs + songs2
```

You can also use += to add and assign, like this:

```
both += ["Everything has Changed"]
```

Dictionaries

As you've seen, Swift arrays are a collection where you access each item using a numerical index, such as **songs[0]**. Dictionaries are another common type of collection, but they differ from arrays because they let you access values based on a key you specify.

To give you an example, let's imagine how we might store data about a person in an array:

```
var person = ["Taylor", "Alison", "Swift", "December",
"taylorswift.com"]
```

To read out that person's middle name, we'd use **person[1]**, and to read out the month they were born we'd use **person[3]**. This has a few problems, not least that it's difficult to remember what index number is assigned to each value in the array! And what happens if the person has no middle name? Chances are all the other values would move down one place, causing chaos in your code.

With dictionaries we can re-write this to be far more sensible, because rather than arbitrary numbers you get to read and write values using a key you specify. For example:

```
var person = ["first": "Taylor", "middle": "Alison", "last": "Swift",
"month": "December", "website": "taylorswift.com"]
person["middle"]
person["month"]
```

It might help if I use lots of whitespace to break up the dictionary on your screen, like this:

```
var person = [
        "first": "Taylor",
        "middle": "Alison",
        "last": "Swift",
        "month": "December",
```

```
        "website": "taylorswift.com"
    ]


person["middle"]
person["month"]
```

As you can see, when you make a dictionary you write its key, then a colon, then its value. You can then read any value from the dictionary just by knowing its key, which is much easier to work with.

As with arrays, you can store a wide variety of values inside dictionaries, although the keys are most commonly strings.

Conditional statements

Sometimes you want code to execute only if a certain condition is true, and in Swift that is represented primarily by the **if** and **else** statements. You give Swift a condition to check, then a block of code to execute if that condition is true.

You can optionally also write **else** and provide a block of code to execute if the condition is false, or even **else if** and have more conditions. A "block" of code is just a chunk of code marked with an open brace – **{** – at its start and a close brace – **}** – at its end.

Here's a basic example:

```
var action: String
var person = "hater"


if person == "hater" {

  action = "hate"

}
```

That uses the == (equality) operator introduced previously to check whether the string inside **person** is exactly equivalent to the srting "hater". If it is, it sets the **action** variable to "hate". Note that open and close braces, also known by their less technical name of "curly brackets" – that marks the start and end of the code that will be executed if the condition is true.

Let's add **else if** and **else** blocks:

```
var action: String
var person = "hater"


if person == "hater" {

  action = "hate"
} else if person == "player" {
```

```
    action = "play"

} else {

    action = "cruise"

}
```

That will check each condition in order, and only one of the blocks will be executed: a person is either a hater, a player, or anything else.

Evaluating multiple conditions

You can ask Swift to evaluate as many conditions as you want, but they all need to be true in order for Swift to execute the block of code. To check multiple conditions, use the **&&** operator – it means "and". For example:

```
var action: String

var stayOutTooLate = true

var nothingInBrain = true


if stayOutTooLate && nothingInBrain {

    action = "cruise"

}
```

Because **stayOutTooLate** and **nothingInBrain** are both true, the whole condition is true, and **action** gets set to "cruise." Swift uses something called short-circuit evaluation to boost performance: if it is evaluating multiple things that all need to be true, and the first one is false, it doesn't even both evaluating the rest.

Looking for the opposite of truth

This might sound deeply philosophical, but actually this is important: sometimes you care whether a condition is not true, i.e. is false. You can do this with the ! (not) operator that was introduced earlier. For example:

```
if !stayOutTooLate && !nothingInBrain {
  action = "cruise"
}
```

This time, the **action** variable will only be set if both **stayOutTooLate** and **nothingInBrain** are false – the ! has flipped them around.

Loops

Computers are great at doing boring tasks billions of times in the time it took you to read this sentence. When it comes to repeating tasks in code, you can either copy and paste your code multiple times, or you can use *loops* – simple programming constructs that repeat a block of code for as long as a condition is true.

To demonstrate this, I want to introduce you to a special debugging function called **print()**: you give it some text to print, and it will print it. If you're running in a playground like we are, you'll see your text appear in the results window. If you're running a real app in Xcode, you'll see your text appear in Xcode's log window. Either way, **print()** is a great way to get a sneak peek at the contents of a variable.

Take a look at this code:

```
print("1 x 10 is \(1 * 10)")
print("2 x 10 is \(2 * 10)")
print("3 x 10 is \(3 * 10)")
print("4 x 10 is \(4 * 10)")
print("5 x 10 is \(5 * 10)")
print("6 x 10 is \(6 * 10)")
print("7 x 10 is \(7 * 10)")
print("8 x 10 is \(8 * 10)")
print("9 x 10 is \(9 * 10)")
print("10 x 10 is \(10 * 10)")
```

When it has finished running, you'll have the 10 times table in your playground results pane. But it's hardly efficient code, and in fact a much cleaner way is to loop over a range of numbers using what's called the closed range operator, which is three periods in a row: **...**Using the closed range operator, we could re-write that whole thing in three lines:

```
for i in 1...10 {
  print("\(i) x 10 is \(i * 10)")
```

```
}
```

The results pane just shows "(10 times)" for our loop, meaning that the loop was run 10 times. If you want to know what the loop actually did, hover over the "(10 times)" then click the + button that appears on the right. You'll see a box saying "10 x 10 is 100" appear inside your code, and if you hover over *that* two further buttons will appear. From those two, click the one on the right, and you should see the picture below
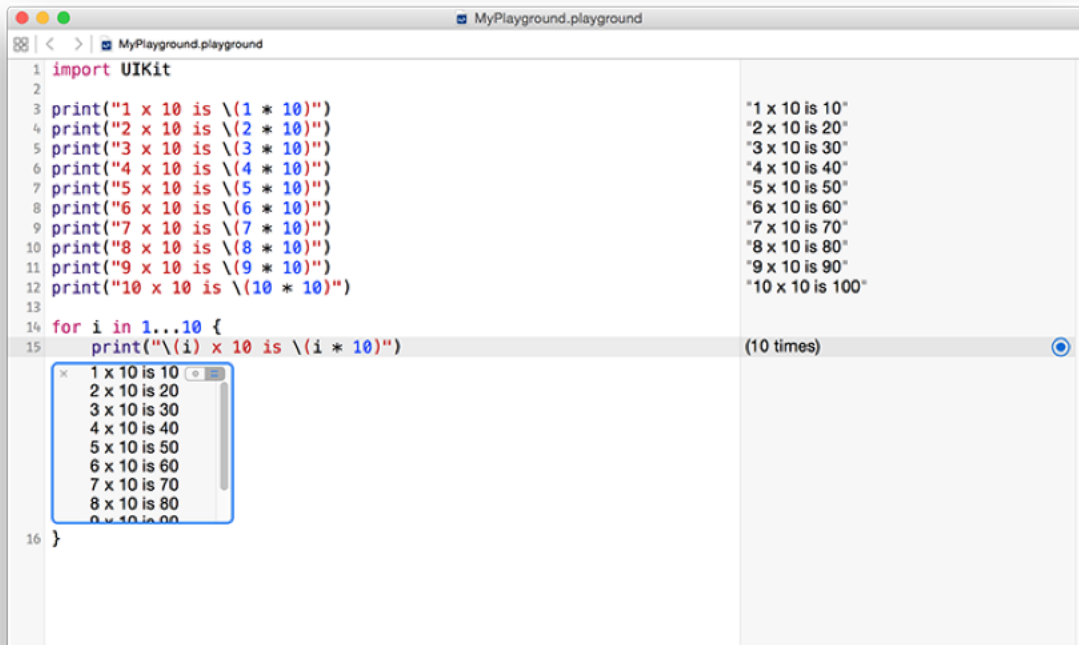
When a Swift playground runs through a loop it only shows how many times that loop was run. If you want to inspect the values more closely, click the + button in the results area.

What the loop does is count from 1 to 10 (including 1 and 10), assigns that number to the constant **i**, then runs the block of code inside the braces.

If you don't need to know what number you're on, you can use an underscore instead. For example, we could print some Taylor Swift lyrics like this:

```
var str = "Fakers gonna"
```

```
for _ in 1 ... 5 {
```

```
    str += " fake"

}


print(str)
```

That will print "Fakers gonna fake fake fake fake fake" by adding to the string each time the loop goes around.

There's a variant of the closed range operator called the half open range operator, and they are easily confused. The half open range operator looks like **..<** and counts from one number up to and excluding another. For example, **1 ..< 5** will count 1, 2, 3, 4.

Looping over arrays

Swift provides a very simple way to loop over all the elements in an array. Because Swift already knows what kind of data your array holds, it will go through every element in the array, assign it to a constant you name, then run a block of your code. For example, we could print out a list of great songs like this:

```
var songs = ["Shake it Off", "You Belong with Me", "Back to
December"]


for song in songs {

  print("My favorite song is \(song)")

}
```

You can also use the **for i in** loop construct to loop through arrays, because you can use that constant to index into an array. We could even use it to index into two arrays, like this:

```
var people = ["players", "haters", "heart-breakers", "fakers"]
```

```
var actions = ["play", "hate", "break", "fake"]


for i in 0 ... 3 {

    print("\(people[i]) gonna \(actions[i])")

}
```

You might wonder what use the half open range operator has, but it's particularly useful for working with arrays because they count from zero. So, rather than counting from 0 up to and including 3, we could count from 0 up to and *excluding* the number of items in an array. Remember: they count from zero, so if they have 4 items the maximum index is 3, which is why we need to use *excluding* for the loop.

To count how many items are in an array, use **someArray.count**. So, we could rewrite our code like this:

```
var people = ["players", "haters", "heart-breakers", "fakers"]
var actions = ["play", "hate", "break", "fake"]


for i in 0 ..< people.count {

    print("\(people[i]) gonna \(actions[i])")

}
```

Inner loops

You can put loops inside loops if you want, and even loops inside loops inside loops – although you might suddenly find you're doing something 10 million times, so be careful!

We can combine two of our previous loops to create this:

```
var people = ["players", "haters", "heart-breakers", "fakers"]
```

```
var actions = ["play", "hate", "break", "fake"]

for i in 0 ..< people.count {
  var str = "\(people[i]) gonna"

  for _ in 1 ... 5 {
    str += " \(actions[i])"
  }

  print(str)
}
```

That outputs "players gonna play play play play play", then "haters gonna…" Well, you get the idea.

One important note: although programmers conventionally use **i**, **j** and even **k** for loop constants, you can name them whatever you please: **for personNumber in 0 ..< count(people)** is perfectly valid.

While loops

There's a third kind of loop you'll see, which repeats a block of code until you tell it to stop. This is used for things like game loops where you have no idea in advance how long the game will last – you just keep repeating "check for touches, animate robots, draw screen, check for touches…" and so on, until eventually the user taps a button to exit the game and go back to the main menu.

These loops are called **while** loops, and they look like this:

```
var counter = 0
```

```
while true {
  print("Counter is now \(counter)")
  counter += 1

  if counter == 556 {
    break
  }
}
```

That code introduces a new keyword, called **break**. It's used to exit a **while** or **for** loop at a point you decide. Without it, the code above would never end because the condition to check is just "true", and true is always true. Without that **break** statement the loop is an infinite loop, which is A Bad Thing.

These **while** loops work best when you're using unknown data, such as downloading things from the internet, reading from a file such as XML, looking through user input, and so on. This is because you only know when to stop the loop after you've run it a sufficient number of times.

There is a counterpart to **break** called **continue**. Whereas breaking out of a loop stops execution immediately and continues directly after the loop, continuing a loop only exits the current iteration of the loop – it will jump back to the top of the loop and pick up from there.

As an example, consider the code below:

```
var songs = ["Shake it Off", "You Belong with Me", "Back to
December"]

for song in songs {
  if song == "You Belong with Me" {
    continue
  }
```

```
    print("My favorite song is \(song)")
}
```

That loops through three Taylor Swift songs, but it will only print the name of two. The reason for this is the **continue** keyword: when the loop tries to use the song "You Belong with Me", **continue** gets called, which means the loop immediately jumps back to the start – the **print()** is call never made, and instead the loop continues straight on to **Back to December**.

Switch case

You've seen **if** statements and now loops, but Swift has another type of flow control called **switch/case**. It's easiest to think of this as being an advanced form of **if**, because you can have lots of matches and Swift will execute the right one.

In the most basic form of a **switch/case** you tell Swift what variable you want to check, the provide a list of possible cases for that variable. Swift will find the first case that matches your variable, then run its block of code. When that block finishes, Swift exits the whole **switch/case** block.

Here's a basic example:

```
let liveAlbums = 2

switch liveAlbums {
case 0:
  print("You're just starting out")

case 1:
  print("You just released iTunes Live From SoHo")

case 2:
  print("You just released Speak Now World Tour")

default:
  print("Have you done something new?")
}
```

We could very well have written that using lots of **if** and **else if** blocks, but this way is clearer and that's important.

One advantage to **switch/case** is that Swift will ensure your cases are exhaustive. That is, if there's the possibility of your variable having a value you don't check for, Xcode will refuse to build your app. In situations where the values are effectively open ended, like our **liveAlbums** integer, you need to include a **default** case to catch these potential values. Yes, even if you "know" your data can only fall within a certain range.

Swift can apply some evaluation to your case statements in order to match against variables. For example, if you wanted to check for a range of possible values, you could use the closed range operator like this:

```swift
let studioAlbums = 5

switch studioAlbums {
case 0...1:
  print("You're just starting out")

case 2...3:
  print("You're a rising star")

case 4...5:
  print("You're world famous!")

default:
  print("Have you done something new?")
}
```

One thing you should know is that **switch/case** blocks in Swift don't fall through like they do in some other languages you might have seen. If you're used to writing **break** in your **case** blocks, you should know this isn't needed in Swift. Instead, you use the **fallthrough** keyword to make one case fall into the next – it's effectively the opposite. Of course, if you have no idea what any of this means, that's even better: don't worry about it!

Functions

Functions let you define re-usable pieces of code that perform specific pieces of functionality. Usually functions are able to receive some values to modify the way they work, but it's not required.

Let's start with a simple function:

```
func favoriteAlbum() {

  print("My favourite is Fearless")

}
```

If you put that code into your playground, nothing will be printed. And yes, it is correct. The reason nothing is printed is that we've placed the "My favourite is Fearless" message into a function called **favoriteAlbum()**, and that code won't be called until we ask Swift to run the **favoriteAlbum()** method. To do that, add this line of code:

```
favoriteAlbum()
```

That runs the function (or "calls" it), so now you'll see "My favourite is Fearless" printed out.

As you can see, you define a function by writing **func**, then your function name, then open and close parentheses, then a block of code marked by open and close braces. You then call that function by writing its name followed by an open and close parentheses.

Of course, that's a silly example – that function does the same thing no matter what, so there's no point it existing. But what if we wanted to print a different album each time? In that case, we could tell Swift we want our function to accept a value when it's called, then use that value inside it.

Let's do that now:

```
func favoriteAlbum(album: String) {
```

```
    print("My favourite is \(album)")

}
```

That tells Swift we want the function to accept one value (called a "parameter"), named "album", that should be a string. We then use string interpolation to write that favorite album directly into our output message.

You might still be wondering what the point is, given that it's still just one line of code. Well, imagine we used that function in 20 different places around a big app, then your head designer comes along and tells you to change the message to "I love Fearless so much – it's my favorite!" Do you really want to find and change all 20 instances in your code? Probably not. With a function you change it once, and everything updates.

You can make your functions accept as many parameters as you want, so let's make it accept name and a year:

```
func printAlbumRelease(name: String, year: Int) {

  print("\(name) was released in \(year)")

}


printAlbumRelease("Fearless", year: 2008)

printAlbumRelease("Speak Now", year: 2010)

printAlbumRelease("Red", year: 2012)
```

You might wonder why the first parameter doesn't need a label. Well, this is an old Apple naming convention: function names should be designed to be read, including their first parameter. So Apple would call that function something like **printReleaseDateOfAlbumNamed()** so that a human reading the code aloud would say "release date of album named Fearless, year 2008."

Return values

Swift functions can return a value by writing **->** then a data type. Once you do this, Swift will ensure that your function will return a value no matter what, so again this is you make a promise about what your code does.

As an example, let's write a function that returns true if an album is one of Taylor Swift's, or false otherwise. This needs to accept one parameter (the name of the album to check) and will return a Boolean. Here's the code:

```
func albumsIsTaylor(name: String) -> Bool {
  if name == "Taylor Swift" { return true }

  if name == "Fearless" { return true }

  if name == "Speak Now" { return true }

  if name == "Red" { return true }

  if name == "1989" { return true }


  return false
}
```

If you wanted to try your new **switch/case** knowledge, this function is a place where it would work well.

You can now call that by passing the album name in and acting on the result:

```
if albumsIsTaylor("Red") {
  print("That's one of hers!")
} else {
  print("Who made that?!")
}


if albumsIsTaylor("Blue") {
```

```
    print("That's one of hers!")
} else {
    print("Who made that?!")
}
```

Optionals

Swift is a very safe language, by which I mean it works hard to ensure your code never fails in surprising ways.

One of the most common ways that code fails is when it tries to use data that is bad or missing. For example, imagine a function like this:

```swift
func getHaterStatus() -> String {

  return "Hate"

}
```

That function doesn't accept any parameters, and it returns a string: "Hate". But what if today is a particularly sunny day, and those haters don't feel like hating – what then? Well, maybe we want to return nothing: this hater is doing no hating today. Now, when it comes to a string you might think an empty string is a great way to communicate nothing, and that might be true sometimes. But how about numbers – is 0 an "empty number"? Or -1?

Before you start trying to create imaginary rules for yourself, Swift has a solution: optionals. An optional value is one that might have a value or might not.

When we used **-> String** it means "this will definitely return a string," which means this function cannot return no value, and thus can be called safe in the knowledge that you'll always get a value back. If we wanted to tell Swift that this function might return a value or it might not, we need to use this instead:

```swift
func getHaterStatus() -> String? {

  return "Hate"

}
```

Note the extra question mark: that means "optional string." Now, in our case we're still returning "Hate" no matter what, but let's go ahead and modify that function further: if the weather is sunny, the haters have turned over a new leaf and have given up their life of

hating, so want to return no value. In Swift, this "no value" has a special name: **nil**.

Change the function to this:

```
func getHaterStatus(weather: String) -> String? {
  if weather == "sunny" {
    return nil
  } else {
    return "Hate"
  }
}
```

That accepts one string parameter (the weather) and returns one string (hating status), but that return value might be there or it might not – it's nil. In this case, it means we might get a string, or we might get nil.

Now for the important stuff: Swift wants your code to be really safe, and trying to use a nil value is a bad idea – it might crash your code, it might screw up your app logic, or it might make your user interface show the wrong thing. As a result, when you declare a value as being optional, Swift will make sure you handle it safely.

Let's try this now: add these lines of code to your playground:

```
var status: String
status = getHaterStatus("rainy")
```

The first line creates a string variable, and the second assigns to it the value from **getHaterStatus()** – and today the weather is rainy, so those haters are hating for sure.

That code will not run, because we said that **status** is of type **String**, which requires a value, but **getHaterStatus()** might not provide one because it returns an optional string. Swift simply will not let you make this mistake, which is extremely helpful because it effectively

stops dead a whole class of common bugs.

To fix the problem, we need to make the **status** variable a **String?**, or just remove the type annotation entirely and let Swift use type inference. The first option looks like this:

```
var status: String?
status = getHaterStatus("rainy")
```

And the second like this:

```
var status = getHaterStatus("rainy")
```

Regardless of which you choose, that value might be there or might not, and by default Swift won't let you use it dangerously. As an example, imagine a function like this:

```
func takeHaterAction(status: String) {
  if status == "Hate" {
    print("Hating")
  }
}
```

That takes a string and prints a message depending on its contents. This function takes a **String** value, and *not* a **String?** value – you can't pass in an optional here, it wants a real string, which means we can't call it using the **status** variable.

Swift has two solutions. Both are used, but one is definitely preferred over the other. The first solution is called optional unwrapping, and it's done inside a conditional statement using special syntax. It does two things at the same time: checks whether an optional has a value, and if so unwraps it into a non-optional type then runs a code block.

The syntax looks like this:

```
if let unwrappedStatus = status {

  // unwrappedStatus contains a non-optional value!

} else {

  // in case you you want an else block, here you go…

}
```

These **if let** statements check and unwrap in one succinct line of code, which makes them very common. Using this method, we can safely unwrap the return value of **getHaterStatus()** and be sure that we only call **takeHaterAction()** with a valid, non-optional string. Here's the complete code:

```
func getHaterStatus(weather: String) -> String? {

  if weather == "sunny" {

    return nil

  } else {

    return "Hate"

  }

}


func takeHaterAction(status: String) {

  if status == "Hate" {

    print("Hating")

  }

}


if let status = getHaterStatus("rainy") {

  takeHaterAction(status)

}
```

If you understand this concept, you're welcome to skip down to the title that says "Force unwrapping optionals". If you're still not quite sure about optionals, carry on reading.OK, if you're still here it means the explanation above either made no sense, or you sort of understood but could probably use some clarification. Optionals are used extensively in Swift, so you really do need to understand them. I'm going to try explaining again, in a different way, and hopefully that will help!

Here's a new function:

```
func yearAlbumReleased(name: String) -> Int {
  if name == "Taylor Swift" { return 2006 }
  if name == "Fearless" { return 2008 }
  if name == "Speak Now" { return 2010 }
  if name == "Red" { return 2012 }
  if name == "1989" { return 2014 }


  return 0
}
```

That takes the name of a Taylor Swift album, and returns the year it was released. But if we call it with the album name "Lantern" because we mixed up Taylor Swift with Hudson Mohawke (an easy mistake to make, right?) then it returns 0 because it's not one of Taylor's albums. But does 0 make sense here? Sure, if the album was released back in 0 AD when Caesar Augustus was emperor of Rome, 0 might make sense, but here it's just confusing – people need to know that 0 means "not recognized."

A much better idea is to re-write that function so that it either returns an integer (when a year was found) or nil (when nothing was found), which is easy thanks to optionals. Here's the new function:

```
func yearAlbumReleased(name: String) -> Int? {
  if name == "Taylor Swift" { return 2006 }
```

```
    if name == "Fearless" { return 2008 }

    if name == "Speak Now" { return 2010 }

    if name == "Red" { return 2012 }

    if name == "1989" { return 2014 }


    return nil

}
```

Now that it returns nil, we need to unwrap the result using **if let** because we need to check whether a value exists or not.

If you understand this concept, you're welcome to skip down to the title that says "Implict and force unwrapping optionals". If you're still not quite sure about optionals, carry on reading.OK, if you're still here it means you're really struggling with optionals, so I'm going to have one last go at explaining them.

Here's an array of names:

```
var items = ["James", "John", "Sally"]
```

If we wanted to write a function that looked in that array and told us the index of a particular name, we might write something like this:

```
func positionOfString(items: [String], str: String) -> Int {

    for i in 0 ..< items.count {

        if items[i] == str {

            return i

        }

    }


    return 0
```

```
}
```

That loops through all the items in the array, returning its position if it finds a match, otherwise returning 0.

Now try running these three lines of code:

```
let jamesPosition = positionOfString(items, str: "James")
let johnPosition = positionOfString(items, str: "John")
let sallyPosition = positionOfString(items, str: "Sally")
let bobPosition = positionOfString(items, str: "Bob")
```

That will output 0, 1, 2, 0 – the positions of James and Bob are the same, even though one exists and one doesn't. This is because I used 0 to mean "not found." The easy fix might be to make -1 not found, but whether it's 0 or -1 you still have a problem because you have to remember that specific number means "not found."

The solution is optionals: return an integer if you found the match, or nil otherwise. In fact, this is exactly the approach the built-in "find in array" functions work: **someArray.indexOf(someValue)**.

When you work with these "might be there, might not be" values, Swift forces you to unwrap them before using them, thus acknowledging that there might not be a value. That's what **if let** syntax does: if the optional has a value then unwrap it and use it, otherwise don't use it at all.

If you're *still* not sure how optionals work, then the best thing to do is ask me on Twitter and I'll try to help: you can find me **@twostraws**.

Force unwrapping optionals

Swift lets you override its safety by using the exclamation mark character: !. If you know that an optional definitely has a value, you can force unwrap it by placing this exclamation mark after it. Please be careful, though: if you try this on a variable that does not have a value, your code will crash.

To put together a working example, here's some foundation code:

```
func yearAlbumReleased(name: String) -> Int? {
  if name == "Taylor Swift" { return 2006 }
  if name == "Fearless" { return 2008 }
  if name == "Speak Now" { return 2010 }
  if name == "Red" { return 2012 }
  if name == "1989" { return 2014 }


  return nil
}


var year = yearAlbumReleased("Red")


if year == nil {
  print("There was an error")
} else {
  print("It was released in \(year)")
}
```

That gets the year an album was released. If the album couldn't be found, **year** will be set to nil, and an error message will be printed. Otherwise, the year will be printed.

Or will it? Well, **yearAlbumReleased()** returns an optional string, and this code doesn't use **if let** to unwrap that optional. As a result, it will print out the following: "It was released in Optional(2012)" – probably not what we wanted!

At this point in the code, we have already checked that we have a valid value, so it's a bit pointless to have another **if let** in there to safely unwrap the optional. So, Swift provides a solution – change the second **print()** call to this:

```
print("It was released in \(year!)")
```

Note the exclamation mark: it means "I'm certain this contains a value, so force unwrap it now."

Implicitly unwrapped optionals

You can also use this exclamation mark syntax to create implicitly unwrapped optionals, which is where some people really start to get confused. So, please read this carefully!

 • A regular variable must contain a value. Example: **String** must contain a string, even if that is string empty, i.e. **""**.
 • An optional variable might contain a value, or might not. It must be unwrapped before it is used. Example: **String?** might contain a string, or it might contain nil. The only way to find out is to unwrap it.
 • An implicitly unwrapped optional might contain a value, or might not. But it does *not* need to be unwrapped before it is used. Swift won't check for you, so you need to be extra careful. Example: **String!** might contain a string, or it might contain nil – and it's down to you to use it appropriately.

There are two main times you're going to meet implicitly unwrapped optionals. The first is when you're working with Apple's APIs: these frequently return implicitly unwrapped optionals because their code pre-dates Swift and that was how things were done in Ye Olde Ages Of Programming.

The second is when you're working with user interface elements in UIKit. These need to be declared up front, but you can't use them until they have been created by iOS – and it likes to create user interface elements at the last possible moment to avoid any unnecessary work. Having to continually unwrap values you definitely know will be there is annoying, so these are made implicitly unwrapped.

Don't worry if you find implicitly unwrapped optionals a bit hard to grasp - it will become clear as you work with the language.

Optional chaining

Working with optionals can feel a bit clumsy sometimes, and all the unwrapping and checking can become so onerous that you might be tempted to throw some exclamation marks to force unwrap stuff so you can get on with work. Be careful, though: if you force unwrap an optional that doesn't have a value, your code will crash.

Swift has two techniques to help make your code less complicated. The first is called optional chaining, which lets you run code only if your optional has a value. Put the below code into your playground to get us started:

```
func albumReleasedYear(year: Int) -> String? {

  switch year {

  case 2006: return "Taylor Swift"

  case 2008: return "Fearless"

  case 2010: return "Speak Now"

  case 2012: return "Red"

  case 2014: return "1989"

  default: return nil

  }

}


let album = albumReleasedYear(2006)

print("The album is \(album)")
```

That will output "The album is Optional("Taylor Swift")" into the results pane.

If we wanted to convert the return value of **albumReleasedYear()** to be uppercase letters (that is, "TAYLOR SWIFT" rather than "Taylor Swift") we could use the **uppercaseString** value of that string. For example:

```
let str = "Hello world"
```

```
print(str.uppercaseString)
```

The problem is, **albumReleasedYear()** returns an optional string: it might return a string or it might return nothing at all. So, what we really mean is, "if we got a string back make it uppercase, otherwise do nothing." And that's where optional chaining comes in, because it provides exactly that behavior.

Try changing the last two lines of code to this:

```
let album = albumReleasedYear(2006)?.uppercaseString

print("The album is \(album)")
```

Note that there's a question mark in there, which is the optional chaining: everything after the question mark will only be run if everything before the question mark has a value. This doesn't affect the underlying data type of **album**, because that line of code will now either return nil or will return the uppercase album name – it's still an optional string.

Your optional chains can be as long as you need, for example:

```
let album =
albumReleasedYear(2006)?.someOptionalValue?.someOtherOptionalValue?.w
hatever
```

Swift will check them from left to right until it finds nil, at which point it stops.

The nil coalescing operator

This simple Swift feature makes your code much simpler and safer, and yet has such a grandiose name that many people are scared of it. This is a shame, because the nil coalescing operator will make your life easier if you take the time to figure it out!

What it is does is let you say "use value A if you can, but if value A is nil then use value B." That's it. It's particularly helpful with optionals, because it effectively stops them from being optional because you provide a non-optional value B. So, if A is optional and has a value, it gets used (we have a value.) If A is present and has no value, B gets used (so we still have a value). Either way, we definitely have a value.

To give you a real context, try using this code in your playground:

```
let album = albumReleasedYear(2006) ?? "unknown"
print("The album is \(album)")
```

That double question mark is the nil coalescing operator, and in this situation it means "if **albumReleasedYear()** returned a value then put it into the **album** variable, but if **albumReleasedYear()** returned nil then use 'unknown' instead."

If you look in the results pane now, you'll see "The album is Taylor Swift" printed in there – no more optionals. This is because Swift can now be sure it will get a real value back, either because the function returned one or because you're providing "unknown". This in turn means you don't need to unwrap anything or risk crashes – you're guaranteed to have real data to work with, which makes your code safer and easier to work with.

Enumerations

Enumerations – usually just called "enum" and pronounced "ee-num" - are a way for you to define your own kind of value in Swift. In some programming languages they are simple little things, but Swift adds a huge amount of power to them if you want to go beyond the basics.

Let's start with a simple example from earlier:

```
func getHaterStatus(weather: String) -> String? {

  if weather == "sunny" {

    return nil

  } else {

    return "Hate"

  }

}
```

That function accepts a string that defines the current weather. The problem is, a string is a poor choice for that kind of data – is it "rain", "rainy" or "raining"? Or perhaps "showering", "drizzly" or "stormy"? Worse, what if one person writes "Rain" with an uppercase R and someone else writes "Ran" because they weren't looking at what they typed?

Enums solve this problem by letting you define a new data type, then define the possible values it can hold. For example, we might say there are five kinds of weather: sun, cloud, rain, wind and snow. If we make this an enum, it means Swift will accept only those five values – anything else will trigger an error. And behind the scenes enums are usually just simple numbers, which are a lot faster than strings for computers to work with.

Let's put that into code:

```
enum WeatherType {

  case Sun, Cloud, Rain, Wind, Snow

}
```

```
func getHaterStatus(weather: WeatherType) -> String? {

    if weather == WeatherType.Sun {

        return nil

    } else {

        return "Hate"

    }

}


getHaterStatus(WeatherType.Cloud)
```

Take a look at the first three lines: line 1 gives our type a name, **WeatherType**. This is what you'll use in place of **String** or **Int** in your code. Line 2 defines the five possible cases our enum can be, as I already outlined. Convention has these start with a capital letter, so Sun, Cloud, etc. And line 3 is just a closing brace, ending the enum.

Now take a look at how it's used: I modified the **getHaterStatus()** so that it takes a **WeatherType** value. The conditional statement is also rewritten to compare against **WeatherType.Sun**, which is our value. Remember, this check is just a number behind the scenes, which is lightning fast.

Now, go back and read that code again, because I'm about to rewrite it with two changes that are important. All set?

```
enum WeatherType {

    case Sun

    case Cloud

    case Rain

    case Wind

    case Snow

}


func getHaterStatus(weather: WeatherType) -> String? {
```

```
  if weather == .Sun {

    return nil

  } else {

    return "Hate"

  }

}



getHaterStatus(.Cloud)
```

I made two differences there. First, each of the weather types are now on their own line. This might seem like a small change, and indeed in this example it is, but it becomes important soon. The second change was that I wrote **if weather == .Sun** – I didn't need to spell out that I meant **WeatherType.Sun** because Swift knows I am comparing against a **WeatherType** variable, so it's using type inference.

Note that at this time Xcode is unable to use code completion to suggest enums if you use this short form. If you type them in full, e.g. **WeatherType.Sun**, you will get code completion.

Enums are particularly useful inside **switch/case** blocks, particularly because Swift knows all the values your enum can have so it can ensure you cover them all. For example, we might try to rewrite the **getHaterStatus()** method to this:

```
func getHaterStatus(weather: WeatherType) -> String? {

  switch weather {

  case .Sun:

    return nil

  case .Cloud, .Wind:

    return "dislike"

  case .Rain:

    return "hate"

  }

}
```

Yes, I realise "haters gonna dislike" is hardly a great lyric, but its academic anyway because this code won't build: it doesn't handle the **.Snow** case, and Swift wants all cases to be covered. You either have to add a case for it or add a default case.

Enums with additional values

One of the most powerful features of Swift is that enumerations can have values attached to them that you define. To extend our increasingly dubious example a bit further, I'm going to add a value to the **.Wind** case so that we can say how fast the wind is. Modify your code to this:

```
enum WeatherType {

    case Sun

    case Cloud

    case Rain

    case Wind(speed: Int)

    case Snow

}
```

As you can see, the other cases don't need a speed value – I put that just into wind. Now for the real magic: Swift lets us add extra conditions to the **swift/case** block so that a case will match only if those conditions are true. This uses the **let** keyword to access the value inside a case, then the **where** keyword for pattern matching.

Here's the new function:

```
func getHaterStatus(weather: WeatherType) -> String? {

    switch weather {
```

```
    case .Sun:

        return nil

    case .Wind(let speed) where speed < 10:

        return "meh"

    case .Cloud, .Wind:

        return "dislike"

    case .Rain, .Snow:

        return "hate"

    }

}


getHaterStatus(WeatherType.Wind(speed: 5))
```

You can see **.Wind** appears in there twice, but the first time is true only if the wind is slower than 10 kilometers per hour. If the wind is 10 or above, that won't match. The key is that you use **let** to get hold of the value inside the enum (i.e. to declare a constant name you can reference) then use a **where** condition to check.

Swift evaluates **switch/case** from top to bottom, and stops as soon as it finds match. This means that if **case .Cloud, .Wind:** appears before **case .Wind(let speed) where speed < 10:** then it will be executed instead – and the output changes. So, think carefully about how you order cases!

Structs

Structs are complex data types, meaning that they are made up of multiple values. You then create an instance of the struct and fill in its values, then you can pass it around as a single value in your code. For example, we could define a **Person** struct type that contains two properties: **clothes** and **shoes**:

```
struct Person {
  var clothes: String
  var shoes: String
}
```

When you define a struct, Swift makes them very easy to create because it automatically generates what's called a memberwise initializer. In plain speak, it means you create the struct by passing in initial values for its two properties, like this:

```
let taylor = Person(clothes: "T-shirts", shoes: "sneakers")
let other = Person(clothes: "short skirts", shoes: "high heels")
```

Once you have created an instance of a struct, you can read its properties just by writing the name of the struct, a period, then the property you want to read:

```
print(taylor.clothes)
print(other.shoes)
```

If you assign one struct to another, Swift copies it behind the scenes so that it is a complete, standalone duplicate of the original. Well, that's not strictly true: Swift uses a technique called "copy on write" which means it only actually copies your data if you try to change it.

To help you see how struct copies work, put this into your playground:

```
struct Person {

  var clothes: String

  var shoes: String

}


let taylor = Person(clothes: "T-shirts", shoes: "sneakers")

let other = Person(clothes: "short skirts", shoes: "high heels")


var taylorCopy = taylor

taylorCopy.shoes = "flip flops"


taylor

taylorCopy
```

That creates two **Person** structs, then creates a third one called **taylorCopy** as a copy of **taylor**. What happens next is the interesting part: the code changes **taylorCopy**, and prints both it and **taylor**. If you look in your results pane (you might need to resize it to fit) you'll see that the copy has a different value to the original: changing one did not change the other.

Classes

Swift has another way of build complex data types called classes. They look similar to structs, but have a number of important differences, including:

 • You don't get an automatic memberwise initializer for your classes; you need to write your own.
 • You can define a class as being based off another class, adding any new things you want.
 • If you copy an object, both copies point at the same data by default.

All three of those are massive differences, so I'm going to cover them in more depth before continuing.

Initializing an object

If we were to convert our **Person** struct into a **Person** class, Swift wouldn't let us write this:

```
class Person {

  var clothes: String

  var shoes: String

}
```

This is because we're declaring the two properties to be **String**, which if you remember means they absolutely must have a value. This was fine in a struct because Swift automatically produces a memberwise initializer for us that forced us to provide values for the two properties, but this doesn't happen with classes so Swift can't be sure they will be given values.

There are three solutions: make the two values optional strings, give them default values, or write our own initializer. The first option is clumsy because it introduces optionals all over our code where they don't need to be. The second option works, but it's a bit wasteful unless those default values will actually be used. That leaves the third option, and really it's the right one: write our own initializer.

To do this, create a function inside the class called **init()** that takes the two parameters we care about:

```
class Person {
  var clothes: String
  var shoes: String

  init(clothes: String, shoes: String) {
    self.clothes = clothes
    self.shoes = shoes
  }
}
```

There are two things that might jump out at you in that code. First, you don't write **func** before your **init()** function, because it's special. Second, because the parameter names being passed in are the same as the names of the properties we want to assign, you use **self.** to make your meaning clear – "the **clothes** property of this object should be set to the **clothes** parameter that was passed in." You can give them unique names if you want – it's down to you.

There are two more things you ought to know but can't see in that code. First, when you write a function inside a class, it's called a *method* instead. In Swift you write **func** whether it's a function or a method, but the distinction is preserved when you talk about them.

Second, Swift requires that all non-optional properties have a value by the end of the initializer, or by the time the initializer calls any other method – whichever comes first.

Class inheritance

The second difference between classes and structs are that classes can build on each other to produce greater things, known as *class inheritance*. This is a technique used extensively in

Cocoa Touch, even in the most basic programs, so it's something you should get to grips with.

Let's start with something simple: a **Singer** class that has properties, which is their name and age. As for methods, there will a simple initializer to handle setting the properties, plus a **sing()** method that outputs some words:

```
class Singer {
  var name: String
  var age: Int

  init(name: String, age: Int) {
    self.name = name
    self.age = age
  }

  func sing() {
    print("La la la la")
  }
}
```

We can now create an instance of that object by calling that initializer, then read out its properties and call its method:

```
var taylor = Singer(name: "Taylor", age: 25)
taylor.name
taylor.age
taylor.sing()
```

That's our basic class, but we're going to build on it: I want to define a **CountrySinger** class

that has everything the **Singer** class does, but when I call **sing()** on it I want to print "Trucks, girls, and liquor" instead.

You could of course just copy and paste the original **Singer** into a new class called **CountrySinger** but that's a lazy way to program and it will come back to haunt if you make later changes to **Singer** and forget to copy them across. Instead, Swift has a smarter solution: we can define **CountrySinger** as being based off **Singer** and it will get all its properties and methods for us to build on:

```
class CountrySinger: Singer {


}
```

That colon is what does the magic: it means "**CountrySinger** extends **Singer**." Now, that new **CountrySinger** class (called a subclass) doesn't add anything to **Singer** (called the parent class, or superclass) yet. We want it to have its own **sing()** method, but in Swift you need to learn a new keyword: **override**. This means "I know this method was implemented by my parent class, but I want to change it for this subclass."

Having the **override** keyword is helpful, because it makes your intent clear. It also allows Swift to check your code: if you don't use **override** Swift won't let you change a method you got from your superclass, or if you use **override** and there wasn't anything to override, Swift will point out your error.

So, we need to use **override func**, like this:

```
class CountrySinger : Singer {
  override func sing() {
    print("Trucks, girls, and liquor")
  }
}
```

Now modify the way the **taylor** object is created:

```
var taylor = CountrySinger(name: "Taylor", age: 25)
taylor.sing()
```

If you change **CountrySinger** to just **Singer** you should be able to see the different messages appearing in the results pane.

Now, to make things more complicated, we're going to define a new class called **HeavyMetalSinger**. But this time we're going to store a new property called **noiseLevel** defining how loud this particular heavy metal singer likes to scream down their microphone.

This causes a problem, and it's one that needs to be solved in a very particular way:

• Swift wants all non-optional properties to have a value.
• Our **Singer** class doesn't have a **noiseLevel** property.
• So, we need to create a custom initializer for **HeavyMetalSinger** that accepts a noise level.
• That new initializer also needs to know the **name** and **age** of the heavy metal singer, so it can pass it onto the superclass **Singer**.
• Passing on data to the superclass is done through a method call, and you can't make method calls in initializers until you have given all your properties values.
• So, we need to set our own property first (**noiseLevel**) then pass on the other parameters for the superclass to use.

That might sound awfully complicated, but in code it's straightforward. Here's the **HeavyMetalSinger** class, complete with its own **sing()** method:

```
class HeavyMetalSinger : Singer {
  var noiseLevel: Int

  init(name: String, age: Int, noiseLevel: Int) {
    self.noiseLevel = noiseLevel
    super.init(name: name, age: age)
```

```
    }

  override func sing() {
    print("Grrrrr rargh rargh rarrrrgh!")
  }
}
```

Notice how its initializer takes three parameters, then calls **super.init()** to pass **name** and **age** on to the **Singer** superclass - but only after its own property has been set. You'll see **super** used a lot when working with objects, and it just means "call a method on the class I inherited from. It's usually used to mean "let my parent class do everything it needs to do first, then I'll do my extra bits."

Class inheritance is a big topic so don't fret if it's not clear just yet. However, there is one more thing you need to know: class inheritance often spans many levels. For example, A could inherit from B, and B could inherit from C, and C could inherit from D, and so on. This lets you build functionality and re-use up over a number of classes, helping to keep your code modular and easy to understand.

Values vs References

When you copy a struct, the whole thing is duplicated, including all its values. This means that changing one copy of a struct doesn't change the other copies – they are all individual. With classes, each copy of an object points at the same original object, so if you change one they all change. Swift calls structs "value types" because they just point at a value, and classes "reference types" because objects are just shared references to the real value.

This is an important difference, and it means the choice between structs and classes is an important one:

 • If you want to have one shared state that gets passed around and modified in place, you're looking for classes. You can pass them into functions or store them in arrays, modify them in there, and have that change reflected in the rest of your program.

• If you want to avoid shared state where one copy can't affect all the others, you're looking for structs. You can pass them into functions or store them in arrays, modify them in there, and they won't change wherever else they are referenced.

If I were to summarise this key difference between structs and classes, I'd say this: classes offer more flexibility, whereas structs offer more safety. As a basic rule, you should always use structs until you have a reason to use classes.

Properties

Structs and classes (collectively: "types") can have their own variables and constants, and these are called properties. These let you attach values to your types to represent them uniquely, but because types can also have methods you can have them behave according to their own data.

Let's take a look at an example now:

```
struct Person {
  var clothes: String
  var shoes: String

  func describe() {
    print("I like wearing \(clothes) with \(shoes)")
  }
}


let taylor = Person(clothes: "T-shirts", shoes: "sneakers")
let other = Person(clothes: "short skirts", shoes: "high heels")
taylor.describe()
other.describe()
```

As you can see, when you use a property inside a method it will automatically use the value that belongs to the same object.

Property observers

Swift lets you add code to be run when a property is about to be changed or has been changed. This is frequently a good way to have a user interface update when a value

changes, for example.

There are two kinds of property observer: **willSet** and **didSet**, and they are called before or after a property is changed. In **willSet** Swift provides your code with a special value called **newValue** that contains what the new property value is going to be, and in **didSet** you are given **oldValue** to represent the previous value.

Let's attach two property observers to the **clothes** property of a **Person** struct:

```
struct Person {
  var clothes: String {
    willSet {
      updateUI("I'm changing from \(clothes) to \(newValue)")
    }

    didSet {
      updateUI("I just changed from \(oldValue) to \(clothes)")
    }
  }
}


func updateUI(msg: String) {
  print(msg)
}


var taylor = Person(clothes: "T-shirts")
taylor.clothes = "short skirts"
```

That will print out the messages "I'm changing from T-shirts to short skirts" and "I just changed from T-shirts to short skirts."

Computed properties

It's possible to make properties that are actually code behind the scenes. We already used the **uppercaseString** property of strings, for example – that's something that gets calculated as needed, rather than every string always storing an uppercase version of itself.

To make a computed property, place an open brace after your property then use either **get** or **set** to make an action happen at the appropriate time. For example, if we wanted to add a **ageInDogYears** property that automatically returned a person's age multiplied by seven, we'd do this:

```
struct Person {

  var age: Int


  var ageInDogYears: Int {

    get {

      return age * 7

    }

  }

}


var fan = Person(age: 25)

print(fan.ageInDogYears)
```

Computed properties are common in Apple's code, but less common in user code. Many programmers prefer to use methods because their behavior is clearer.

Static properties and methods

Swift lets you create properties and methods that belong to a type, rather than to instances of a type. This is helpful for organizing your data meaningfully by storing shared data.

Swift calls these shared properties "static properties", and you create one just by using the **static** keyword. Once that's done, you access the property by using the full name of the type. Here's a simple example:

```
struct TaylorFan {
    static var favoriteSong = "Shake it Off"


    var name: String
    var age: Int
}


let fan = TaylorFan(name: "James", age: 25)
print(TaylorFan.favoriteSong)
```

So, a Taylor Swift fan has a name and age that belongs to them, but they all have the same favorite song.

Because static methods belong to the class rather than to instances of a class, you can't use it to access any non-static properties from the class.

Access control

This is an important feature and one you need to understand, but sadly it's one that doesn't work in Swift playgrounds so you'll just have to take my word for it. Access control lets you specify what data inside structs and classes should be exposed to the outside world, and you get to choose three modifiers:

 • Public: this means everyone can read and write the property.
 • Internal: this means only your Swift code can read and write the property.
 • Private: this means that only Swift code in the same file as the type can read and write the property.

Most of the time you don't need to specify access control, but sometimes you'll want to explicitly set a property to be private because it stops others from accessing it directly. This is useful because your own methods can work with that property, but others can't, thus forcing them to go through your code to perform certain actions.

To declare a property private, just do this:

```
class TaylorFan {

  private var name: String!

}
```

Reminder: this does nothing on Swift playgrounds, because one playground is effectively one file and thus can read and write any data it likes.

Polymorphism and type casting

Because classes can inherit from each other (e.g. **CountrySinger** can inherit from **Singer**) it means one class is effectively a superset of another: class B has all the things A has, with a few extras. This in turn means that you can treat B as type B or as type A, depending on your needs.

Confused? Let's try some code:

```
class Album {
  var name: String

  init(name: String) {
    self.name = name
  }
}


class StudioAlbum: Album {
  var studio: String

  init(name: String, studio: String) {
    self.studio = studio
    super.init(name: name)
  }
}


class LiveAlbum: Album {
  var location: String

  init(name: String, location: String) {
    self.location = location
```

```
    super.init(name: name)

  }

}
```

That defines three classes: albums, studio albums and live albums, with the latter two both inheriting from **Album**. Because any instance of **LiveAlbum** is inherited from **Album** it can be treated just as either **Album** or **LiveAlbum** – it's both at the same time. This is called "polymorphism," but it means you can write code like this:

```
var taylorSwift = StudioAlbum(name: "Taylor Swift", studio: "The
Castles Studios")

var fearless = StudioAlbum(name: "Speak Now", studio: "Aimeeland
Studio")

var iTunesLive = LiveAlbum(name: "iTunes Live from SoHo", location:
"New York")


var allAlbums: [Album] = [taylorSwift, fearless, iTunesLive]
```

There we create an array that holds only albums, but put inside it two studio albums and a live album. This is perfectly fine in Swift because they are all descended from the **Album** class, so they share the same basic behavior.

We can push this a step further to really demonstrate how polymorphism works. Let's add a **getPerformance()** method to all three classes:

```
class Album {

  var name: String


  init(name: String) {

    self.name = name

  }
```

```
  func getPerformance() -> String {

    return "The album \(name) sold lots"

  }

}


class StudioAlbum: Album {

  var studio: String


  init(name: String, studio: String) {

    self.studio = studio

    super.init(name: name)

  }


  override func getPerformance() -> String {

    return "The studio album \(name) sold lots"

  }

}


class LiveAlbum: Album {

  var location: String


  init(name: String, location: String) {

    self.location = location

    super.init(name: name)

  }


  override func getPerformance() -> String {

    return "The live album \(name) sold lots"

  }

}
```

The **getPerformance()** method exists in the **Album** class, but both child classes override it. When we create an array that holds **Albums**, we're actually filling it with subclasses of albums: **LiveAlbum** and **StudioAlbum**. They go into the array just fine because they inherit from the **Album** class, but they never lose their original class. So, we could write code like this:

```
var taylorSwift = StudioAlbum(name: "Taylor Swift", studio: "The
Castles Studios")

var fearless = StudioAlbum(name: "Speak Now", studio: "Aimeeland
Studio")

var iTunesLive = LiveAlbum(name: "iTunes Live from SoHo", location:
"New York")


var allAlbums: [Album] = [taylorSwift, fearless, iTunesLive]


for album in allAlbums {

  print(album.getPerformance())

}
```

That will automatically use the override version of **getPerformance()** depending on the subclass in question. That's polymorphism in action: an object can work as its class and its parent classes, all at the same time.

Converting types with type casting

You will often find you have an object of a certain type, but really you know it's a different type. Sadly, if Swift doesn't know what you know, it won't build your code. So, there's a solution, and it's called type casting: converting an object of one type to another.

Chances are you're struggling to think why this might be necessary, but I can give you a very

simple example:

```
for album in allAlbums {
  print(album.getPerformance())
}
```

That was our loop from a few minutes ago. The **allAlbums** array holds the type **Album**, but we know that really it's holding one of the subclasses: **StudioAlbum** or **LiveAlbum**. Swift doesn't know that, so if you try to write something like **print(album.studio)** it will refuse to build because only **StudioAlbum** objects have that property.

Type casting in Swift comes in three forms, but most of the time you'll only meet two: **as?** and **as!**, known as optional downcasting and forced downcasting. The former means "I think this conversion might be true, but it might fail," and the second means "I know this conversion is true, and I'm happy for my app to crash if I'm wrong." When I say "conversion" I don't mean that the object literally gets transformed. Instead, it's just converting how Swift treats the object – you're telling Swift that an object it thought was type A is actually type E.

The question and exclamation marks should give you a hint of what's going on, because this is very similar to optional territory. For example, if you write this:

```
for album in allAlbums {
  let studioAlbum = album as? StudioAlbum
}
```

Swift will make **studioAlbum** have the data type **StudioAlbum?**. That is, an optional studio album: the conversion might have worked, in which case you have a studio album you can work with, or it might have failed, in which case you have nil. This is most commonly used with **if let** to automatically unwrap the optional result, like this:

```
for album in allAlbums {
  print(album.getPerformance())
```

```
   if let studioAlbum = album as? StudioAlbum {

     print(studioAlbum.studio)

   } else if let liveAlbum = album as? LiveAlbum {

     print(liveAlbum.location)

   }

}
```

That will go through every album and print its name, because that's common to the **Album** class and all its subclasses. It then checks whether it can convert the **album** value into a **StudioAlbum**, and if it can it prints out the studio name. The same thing is done for the **LiveAlbum** in the array.

Forced downcasting is when you're really sure an object of one type can be treated like a different type, but if you're wrong your program will just crash. Forced downcasting doesn't need to return an optional value, because you're saying the conversion is definitely going to work – if you're wrong, it means you wrote your code wrong.

To demonstrate this in a non-crashy way, let's strip out the live album so that we just have studio albums in the array:

```
var taylorSwift = StudioAlbum(name: "Taylor Swift", studio: "The
Castles Studios")

var fearless = StudioAlbum(name: "Speak Now", studio: "Aimeeland
Studio")


var allAlbums: [Album] = [taylorSwift, fearless]


for album in allAlbums {

  let studioAlbum = album as! StudioAlbum

  print(studioAlbum.studio)

}
```

That's obviously a contrived example, because if that really were your code you would just change **allAlbums** so that it had the data type **[StudioAlbum]**. Still, it shows how forced downcasting works, and the example won't crash because it makes the correct assumptions.

Swift lets you downcast as part of the array loop, which in this case would be more efficient. If you wanted to write that forced downcast at the array level, you would write this:

```
for album in allAlbums as! [StudioAlbum] {
  print(album.studio)
}
```

That no longer needs to downcast every item inside the loop, because it happens when the loop begins. Again, you had better be correct that all items in the array are **StudioAlbums**, otherwise your code will crash.

Swift also allows optional downcasting at the array level, although it's a bit more tricksy because you need to use the nil coalescing operator to ensure there's always a value for the loop. Here's an example:

```
for album in allAlbums as? [LiveAlbum] ?? [LiveAlbum]() {
  print(album.location)
}
```

What that means is, "try to convert **allAlbums** to be an array of **LiveAlbum** objects, but if that fails just create an empty of live albums and use that instead" – i.e., do nothing. It's possible to use this, but I'm not sure you'd really want to!

Closures

You've met integers, strings, doubles, floats, Booleans, arrays, dictionaries, structs and classes so far, but there's another type of data that is used extensively in Swift, and it's called a closure. These are complicated, but they are so powerful and expressive that they are used pervasively in Cocoa Touch, so you won't get very far without understanding them.

A closure can be thought of as a variable that holds code. So, where an integer holds 0 or 500, a closure holds lines of Swift code. It's different to a function, though, because closures are a data type in their own right: you can pass a closure as a parameter or store it as a property. Closures also capture the environment where they are created, which means they take a copy of the values that are used inside them.

You never *need* to design your own closures so don't be afraid if you find the following quite complicated. However, Cocoa Touch will often ask you to write closures to match its needs, so you at least need to know how they work. Let's take a Cocoa Touch example first:

```
let vw = UIView()


UIView.animateWithDuration(0.5, animations: {

  vw.alpha = 0

})
```

UIView is a data type in UIKit that represents the most basic kind of user interface container. Don't worry about what it does for now, all that matters is that it's the basic user interface component. **UIView** has a method called **animateWithDuration()** and it lets you change the way your interface looks using animation – you describe what's changing and over how many seconds, and Cocoa Touch does the rest.

The **animateWithDuration()** method takes two parameters in that code: the number of seconds to animate over, and a closure containing the code to be executed as part of the animation. I've specified half a second as the first parameter, and for the second I've asked UIKit to adjust the view's alpha (that's opacity) to 0, which means "completely transparent."

This method needs to use a closure because UIKit has to do all sorts of work to prepare for the animation to begin, so what happens is that UIKit takes a copy of the code inside the

braces (that's our closure), stores it away, does all its prep work, then runs our code when it's ready. This wouldn't be possible if we just run our code directly.

The above code also shows how closure's capture their environment: I declared the **vw** constant outside of the closure, then used it inside. Swift detects this, and makes that data available inside the closure too.

Swift's system of automatically capturing a closure's environment is very helpful, but can occasionally trip you up: if object A stores a closure as a property, and that property also references object A, you have something called a strong reference cycle and you'll have unhappy users. This is a substantially more advanced topic than you need to know right now, so don't worry too much about it just yet.

Trailing closures

As closures are used so frequently, Swift can apply a little syntactic sugar to make your code easier to read. The rule is this: if the last parameter to a method takes a closure, you can eliminate that parameter and instead provide it as a block of code. For example, we could convert the previous code to this:

```
let vw = UIView()


UIView.animateWithDuration(0.5) {

  vw.alpha = 0

}
```

It does make your code shorter and easier to read, so this syntax form – known as trailing closure syntax – is preferred.

Wrap up

That's the end of our tour around the Swift programming language. I haven't tried to cover everything in the language, but that's OK because you have all the important stuff, all the sometimes-important stuff, and all the nice-to-know stuff – the interesting-but-rarely-used features you'll either come across in a later project or through extended experience with the language.

If you find yourself a bit confused about how exactly some features are used, don't worry: as soon as you start coding apps with Swift things will start to make a lot more sense!

At this point, you have two options: start on Project 1 of Hacking with Swift, where you make your initial iOS app and recap some of the basic features of Swift, or if you're really desperate to learn more about the language I have four articles that teach you **the new features of Swift 2.0 by example**.

Project 1

Storm Viewer

Get started coding in Swift by making an image viewer app and learning key concepts.

Setting up

In this project you'll produce an application that lets users scroll through a list of images, then select one to view. It's deliberately simple, because there are many other things you'll need to learn along the way, so strap yourself in – this is going to be long!

Launch Xcode, and choose "Create a new project" from the welcome screen. Choose Master-Detail Application from the list and click Next. For Product Name enter Project1, then make sure you have Swift selected for language and Universal for devices.
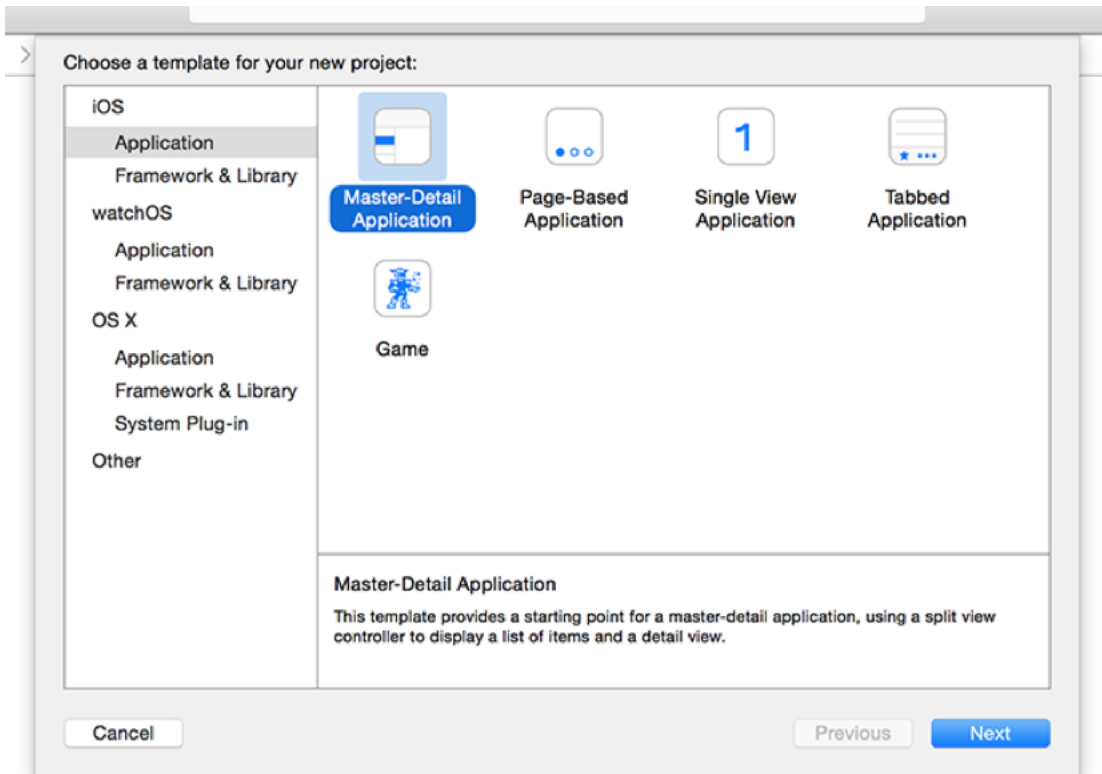
Creating a new Master-Detail Application project in Xcode.

One of the fields you'll be asked for is "Organisation Identifier", which is a unique identifier usually made up of your personal web site domain name in reverse. For example, I would use **com.hackingwithswift** if I were making an app. You'll need to put something valid in there if you're deploying to devices, but otherwise you can just use **com.example**.
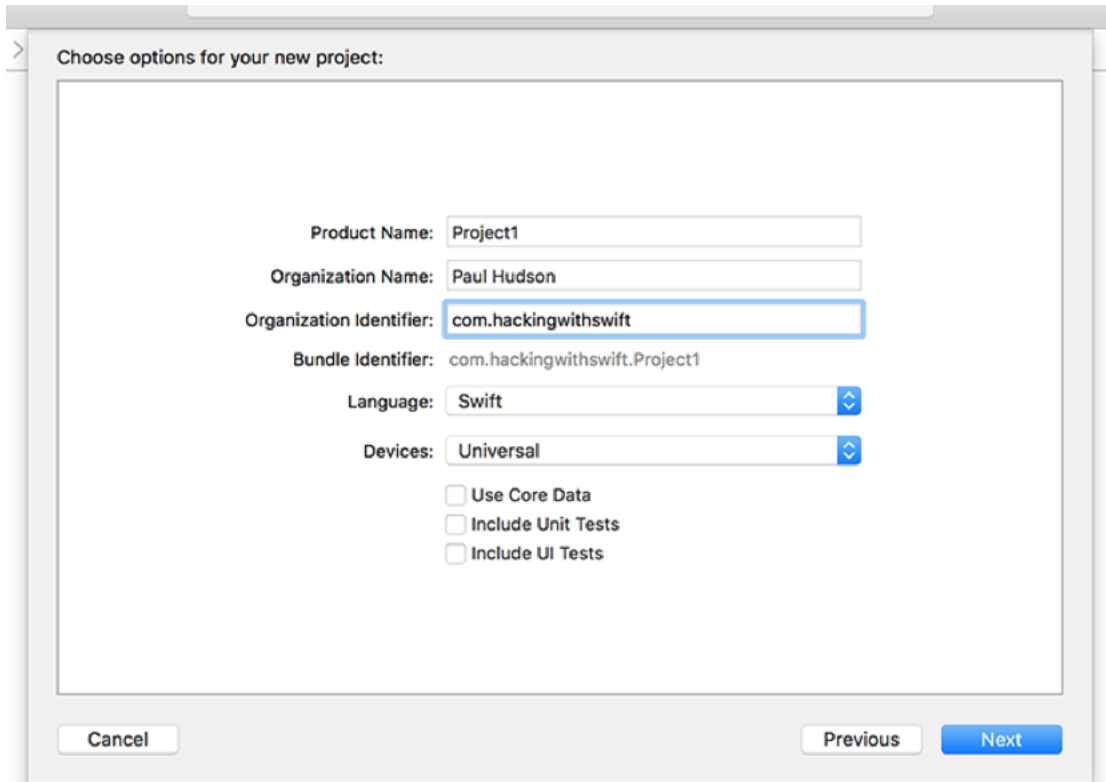
Setting your Organization Identifier in Xcode.

Important note: some of Xcode's project templates have checkboxes saying "Use Core Data", "Include Unit Tests" and "Include UI Tests". Please ensure these boxes are unchecked for this project and indeed all projects in this series.

Now click Next again and you'll be asked where you want to save the project – your desktop is fine. Once that's done, you'll be presented with the example project that Xcode made for you. The first thing we need to do is make sure you have everything set up correctly, and that means running the project as-is.

When you run a project, you get to choose what kind of device the iOS Simulator should pretend to be, or you can also select a physical device if you have one plugged in. These options are listed under the Product > Destination menu, and you should see iPad 2, iPad Air, iPhone 6, and so on.

There's also a shortcut for this menu: at the top-left of Xcode's window is the play and stop button, but to the right of that it should say Project1 then a device name. You can click on that device name to select a different device.

For now, please choose iPhone 5s, and click the Play triangle button in the top-left corner. This will compile your code, which is the process of converting it to instructions that iPhones can understand, then launch the simulator and run the app. As you'll see when you interact with the app, this basic project adds dates to the table when you click the + button, you can delete them either by swiping or using the Edit button, and tapping on a date brings in a new screen showing the date in its center.
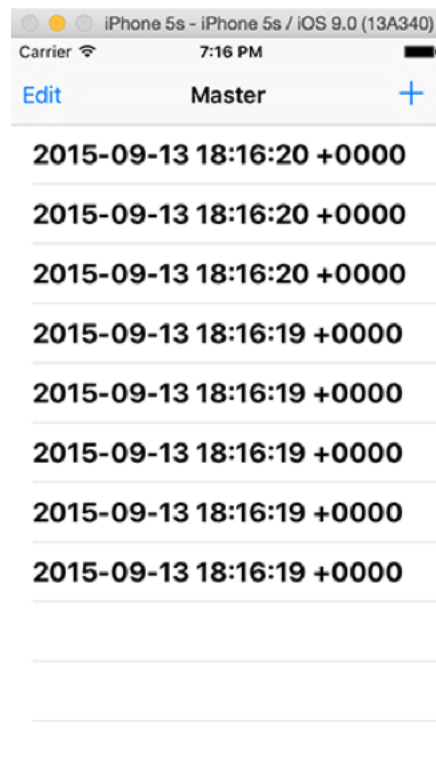
The basic Master-Detail Application project in Xcode.

You'll be starting and stopping projects a lot as you learn, so there are three basic tips you need to know:

• You can run your project by pressing Cmd+R. This is equivalent to clicking the play button.
• You can stop a running project by pressing Cmd+.
• If you have made changes to a running project, just press Cmd+R again. Xcode will prompt you to stop the current run before starting another. Make sure you check the "Do not show this message again" box to avoid being bothered in the future.

This project is all about letting users select images to view, so you're going to need to import some pictures. Download the files for this project from **GitHub**, and look in the Project1 folder. You'll see another folder in there called Project1, and inside that a folder called Content.

I want you to drag that Content folder straight into your Xcode project, just under where it says "Info.plist". A window will appear asking how you want to add the files: make sure "Copy items if needed" is checked, and "Create groups" is selected. **Important: do not choose "Create folder references" otherwise your project will not work.**
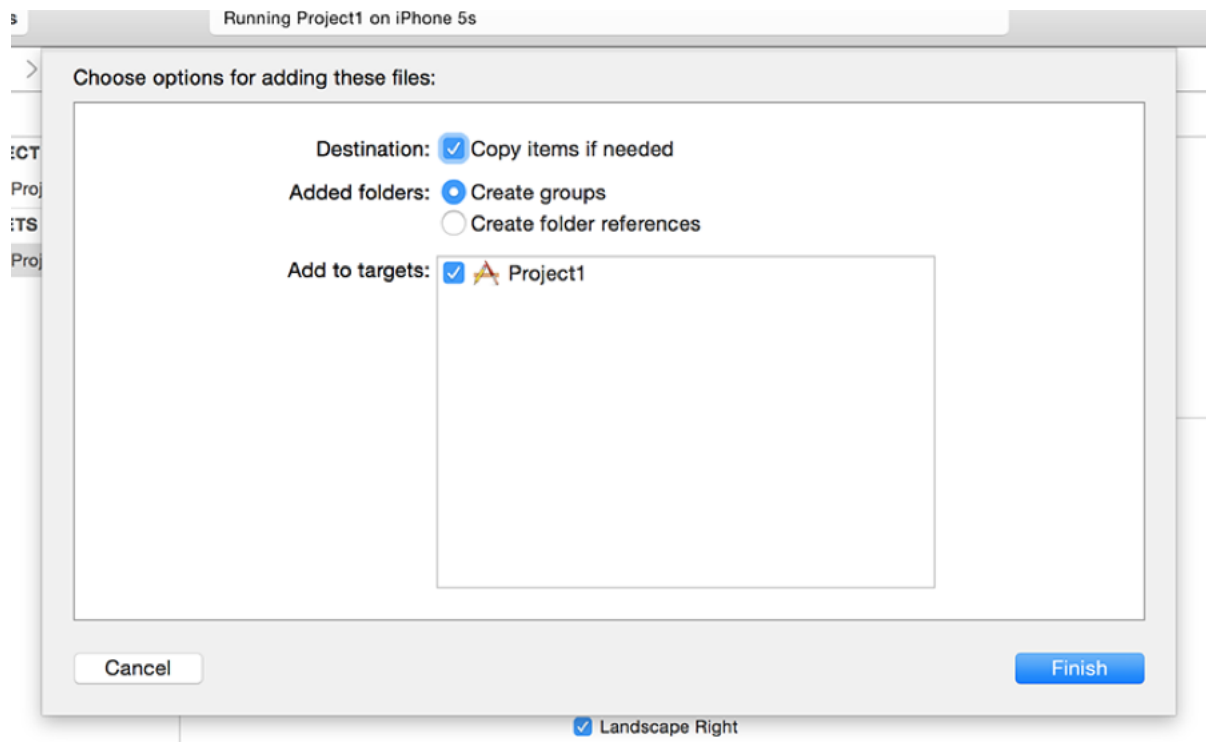
When you add items to Xcode, make sure you choose Create Folder References.

Click Finish and you'll see a yellow Content folder appear in Xcode. If you see a blue one, you didn't select "Create groups", and you'll have problems following this tutorial!

Deleting skeleton code

Apple's example contains lots of code we don't need, so let's zap it: select the file MasterViewController.swift to open it for editing. Around line 17 you'll see the code starting with **override func viewDidLoad() {**, and there'll be some more lines of code until it reaches a **}** on a line all by itself on line 28. If you're not sure which **}** I mean, it's the one that is aligned directly beneath the first letter in **override**.

No line numbers? If your Xcode isn't showing line numbers by default, I suggest you turn them on. Go to the Xcode menu and choose Preferences, then choose the Text Editing tab and make sure "Line numbers" is checked.

This block of code is the **viewDidLoad()** method, which is code that gets called when the system has finished creating the screen and is giving you the chance to configure it.

The method starts at the **func viewDidLoad() {** line and ends on the **}** not far below. These symbols, known as *braces* (or sometimes *curly brackets*) are used to mark chunks of code, and it's convention to indent lines inside braces so that it's easy to identify where code blocks start and end. But enough of the theory: almost everything inside this method is not needed, so delete its contents except for the line **super.viewDidLoad()**.

Note: when I say "delete its contents" I mean leave the **func viewDidLoad() {** and **}** intact, but remove everything in between except for that one line. So, it should look like this:

```
override func viewDidLoad() {

    super.viewDidLoad()

}
```

This method does nothing now, but that's OK: we'll be filling it in later.

Next up, look for the **insertNewObject()** method, which starts with **func insertNewObject() {** and ends with a closing brace a few lines later. Delete the entire method – yes, even the **func insertNewObject(sender: AnyObject) {** and **}** parts.

These methods start with **func**, which is short for "function", which for nearly all intents and purposes is identical to a method in Swift. There is one small exception, but you won't come

to it until project 24 so for now please just consider functions and methods identical.

Finally, delete the very last method in the file. This one has a very peculiar name, and it's a bit of a hangover from Apple's previous language, Objective C. The method is quite long, and although we don't need it here, you do need to understand what it means. Here's the method definition:

```
override func tableView(tableView: UITableView, commitEditingStyle
editingStyle: UITableViewCellEditingStyle, forRowAtIndexPath
indexPath: NSIndexPath)
```

Let's break that down…

 • **override** This means the method has been defined already, and we want to override the existing definition with this new definition. If you didn't override it, then the previously defined method would execute, and in this instance it would do nothing.
 • **func tableView** The method's name is **tableView**, which doesn't sound very useful. But the way Apple defines methods is to ensure that the information that gets passed into them – the parameters – are named usefully, and in this case the very first thing that gets passed in is the table view that triggered the code. A table view, as you might have gathered, is the scrolling thing that's containing all the dates in the example project, and is a core component in iOS.
 • **tableView: UITableView** As promised, the first thing that gets passed into to the method is the table view that triggered the code, and here it is. But this contains two pieces of information: **tableView** is the name that we can use to reference the table view inside the method, and **UITableView** is the data type – the bit that describes what it is. Everything that begins with "UI" is something Apple provides as part of its iOS development kit, so **UITableView** is the default Apple table view.
 • **commitEditingStyle editingStyle: UITableViewCellEditingStyle** This provides the really important part of the method: what it's trying to do. We know it involves a table view because that's the name of the method, but the **commitEditingStyle** part is the actual action: this code will be triggered when the user tries to commit the editing style of a table view. Translated, that means this code will be called when the user tries to add or delete data from the table.

But wait: there's more! When we had **tableView: UITableView**, it meant we could reference the table view by using the "tableView" parameter, but here the parameter is **commitEditingStyle**, which would be quite a clumsy way to reference the parameter. So Swift lets you give parameters external names: one used when passing data in (**commitEditingStyle**, in this case) and one to be used inside the method (**editingStyle**). Finally, there's the colon and **UITableViewCellEditingStyle**, which means that the **editingStyle** parameter will be of data type **UITableViewCellEditingStyle**.

 • **forRowAtIndexPath indexPath: NSIndexPath** Here's that sneaky parameter naming again, but again hopefully you can see how it's useful: people calling the method would write **forRowAtIndexPath**, and inside the method you would use just **indexPath**. It's of data type **NSIndexPath**, which holds two pieces of information: a table section and a table row. We aren't using sections here, but we are using rows – in the example project, every date that is inserted is one row.

I'm not going to pretend it's easy to understand how Swift methods look and work, but the best thing to do is not worry too much if you don't understand right now because after a few hours of coding they will be second nature. At the very least you do need to know that these methods are referred to using their name (**tableView**) and any named parameters. Parameters without names are just referenced as underscores: _.

So, the method you just deleted is referred to as **tableView(_:commitEditingStyle:forRowAtIndexPath:)** – clumsy, I know, which is why most people usually just talk about the important bit, for example, "in the **commitEditingStyle** method."

The last things to delete are a little more subtle: look for the **tableView(_:cellForRowAtIndexPath:)** method (I'll just call it **cellForRowAtIndexPath** from now on) and you'll see these two lines of code:

```
let object = objects[indexPath.row] as! NSDate
cell.textLabel!.text = object.description
```

I want you to delete the part that says **as! NSDate** and the part that says **.description**. Your final code should look like this:

```
let object = objects[indexPath.row]
cell.textLabel!.text = object
```

When you make this change, Xcode will start telling you there are problems. This is perfectly normal – there are more changes to come.

Now find the **prepareForSegue()** method and you'll see another **as NSDate** in there that you should delete, so the line should read this:

```
let object = objects[indexPath.row]
```

If you were able to run the project now, you would see it's basically useless because the add button has been removed. But you can't run the program just yet, because it has problems that need to be fixed. That's OK, though, because we're about to bring it to life again…

Listing our images

The images I've provided you with come from the National Oceanic and Atmospheric Administration (NOAA), which is a US government agency and thus produces public domain content that we can freely reuse. Once they are copied into your project, Xcode will automatically build them into your finished app so that you can access them.

Behind the scenes, an iOS (and OS X) app is actually a directory containing lots of files: the binary itself (that's the compiled version of your code, ready to run), all the media assets your app uses, any visual layout files you have, plus a variety of other things such as metadata and security entitlements.

These app directories are called bundles, and they have the file extension .app. Because our media files are loose inside the folder, we can ask the system to tell us all the files that are in there then pull out the ones we want. You may have noticed that all the images start with the name "nssl" (short for National Severe Storms Laboratory), so our task is simple: list all the files in our app's directory, and pull out the ones that start with "nssl".

As I said before, the **viewDidLoad()** method starts at **func viewDidLoad() {** and ends at the **}** a few lines later. We're going to put some more code into that method, just beneath the line that says **super.viewDidLoad()**:

```
let fm = NSFileManager.defaultManager()
let path = NSBundle.mainBundle().resourcePath!
let items = try! fm.contentsOfDirectoryAtPath(path)

for item in items {
  if item.hasPrefix("nssl") {
    objects.append(item)
  }
}
```

I already told you that any data types that start with UI belong to Apple's iOS development kit, but that's only partially true. UI stands for User Interface, and so these types primarily

relate to things the user can interact with – **UITableView** is a table, **UIButton** is a button, **UITextField** is a text entry field, and so on. But there are lots of other data types that Apple provides you with, and here we can see two: **NSFileManager** and **NSBundle**.

To cut a long story short, NS is short for NeXTSTEP, software that Apple bought in 1997. NeXSTEP developed technology that still today lies at the heart of iOS. **NSBundle** and **NSFileManager** are data types that can do some great work for you, but they don't have a visual component. Incidentally, these NS data types nearly always existing on OS X identically to iOS, whereas all those UI things are available only on iOS.

Enough history, let's look at what this code does:

• **let fm = NSFileManager.defaultManager()** This declares a constant called **fm** and assigns it the value returned by **NSFileManager.defaultManager()**. This is a data type that lets us work with the filesystem, and in our case we'll be using it to look for files.

• **let path = NSBundle.mainBundle().resourcePath!** This declares a constant called **path** that is set to the resource path of our app's bundle. Remember, a bundle is a directory containing our compiled program and all our assets. So, this line says, "tell me where I can find all those images I added to my app."

• **let items = try! fm.contentsOfDirectoryAtPath(path)** This declares a third constant called **items** that is set to the contents of the directory at a path. Which path? Well, the one that was returned by the line before. As you can see, Apple's long method names really does make their code quite self-descriptive!

• **for item in items {** This starts a *loop*. Loops are a block of code that execute multiple times. In this case, the loop executes once for every item we found in the app bundle. Note that the line has an opening brace at the end: that signals the start of a new block of code, and there's a matching closing brace four lines beneath.

Everything inside those braces will be executed each time the loop goes around. We could translate this line as "treat items as a series of text strings, then pull out each one of those text strings, give it the name **item**, then run the following code block…" We use text strings because **contentsOfDirectoryAtPath()** returns a list of filenames.

• **if item.hasPrefix("nssl") {** This is the first line inside our loop. By this point, we'll have the first filename ready to work with, and it'll be called **item**. To decide whether it's one we care about or not, we use the **hasPrefix()** method: it takes one parameter (the prefix to search for) and returns either true or false.

That "if" at the start means this line is a conditional statement: if the item has the prefix "nssl", then… that's right, another opening brace to mark another new code block. This time, the code will be executed only if **hasPrefix()** returned true.

 • **objects.append(item)** This code will be executed only if **hasPrefix()** returned true, and it adds the matching filename to the end of a list called **objects**. No, we didn't create that – that was in the Xcode sample project.

In just those few lines of code, there's quite a lot to take in, so let's recap:

 • We use **let** to declare constants. Constants are pieces of data that we want to reference, but that we know won't have a changing value. For example, your birthday is a constant, but your age is not – your age is a variable, because it varies.
 • Swift coders really like to use constants in places most other developers use variables. This is because when you're actually coding you start to realise that most of the data you store doesn't actually change very much, so you might as well make it constant. Doing so allows the system to do some optimisation, and also adds some extra safety because if you try to change a constant the compiler will issue errors.
 • Text in Swift is represented using the **String** data type. Swift strings are extremely powerful and guaranteed to work with any language you can think of – English, Chinese, Klingon and more.
 • Collections of values are called arrays, and are usually restricted to holding one data type at a time. An array of strings is written as **[String]** and can hold only strings. If you try to put numbers in there, the compiler will emit errors. There is a special data type called **AnyObject** that means, as you might imagine, any data type can be placed inside
 • The **try!** keyword is not used frequently, but we're using it here to mean "I realise calling this code might fail, but I'm certain it won't." If the code does fail, our app will crash. At the same time, if the code fails it means our app can't read its own data, so something must be seriously wrong, which is why **try!** is OK here.
 • You can use **for someVar in someArray** to loop through every item in an array. Swift pulls out each item and runs the code inside your loop once for each item.

If you're extremely observant you might have noticed one tiny, tiny little thing that also one of the most complicated parts of Swift, so I'm going to keep it as simple as possible for now, then expand more over time: it's the exclamation mark at the end of **NSBundle.mainBundle().resourcePath!** No, that wasn't a typo from me. If you take away the exclamation mark the code will no longer work, so clearly Xcode thinks it's important – and indeed it is. Swift has three ways of working with data:

1. A variable or constant that holds the data. For example, **foo: String** is a string called **foo**.

2. A variable or constant that might hold the data, but we're not sure. This is called an optional type, and looks like this: **foo: String?** You can't use these directly, instead you need to "unwrap" them first.

3. A variable or constant that might hold the data, and in fact we're sure it does – at least once it has first been set. This is called an implicitly unwrapped optional, and looks like this: **foo: String!** You *can* use these directly.

When I explain this to people, they nearly always get optional and implicitly unwrapped optional confused, largely because they aren't all that different. Implicitly unwrapped optionals – the !s – serve two purposes: they make code easier to work with, and they make for easier compatibility with Apple's vast collection of existing APIs.

We'll look at optionals in more depth later, but for now what matters is that **NSBundle.mainBundle().resourcePath** may or may not return a string, so what it returns is a **String?** – that is, an optional string. By adding the exclamation mark to the end we are force unwrapping the optional string, which means we're saying, "I'm sure this will return a real string, it will never be **nil**, so please just give it to me as a regular string."

Important warning: if you ever try to use a constant or variable that has a **nil value, your app will crash.** As a result, some people have named **!** the "crash" operator because it's easy to get wrong. The same is true of **try!**, which is also easy to get wrong. Don't worry if this all sounds hard for now – you'll be using it more later, and it make more sense over time.

Before we move on to the next piece of work, there's one small change to make: now you know what **AnyObject**, **String** and arrays are, you should be able to see that at the top of **MasterViewController** there is a line that says **var objects = [AnyObject]()**, which defines our array as containing **AnyObject**. With all the changes we've made, we now know for sure that the objects we're adding will always be strings, so we can change that declaration like so:

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

With that change, Swift will now make sure we're always putting in and taking out strings from the objects array, which means it's helping keep our code safe. Your project should now build correctly, and if you run it you should see your image names inside the table. Success! Let's make it more interesting…

Our project so far: the image names are listed inside the UITableView.

Introducing Interface Builder

As you saw when you first ran the template app, selecting a date brought in a new screen showing the date all by itself. This was all done with a smooth animation from left to right, and included adding a Back button so you can get back to the previous screen. You might also have noticed that you can swipe from the left edge to the right to go back to the table view.

All this behavior was provided for us by two important iOS data types: **UISplitViewController** and **UINavigationController**. From those names you can divine two things:

1. The "UI" means it's a user interface component designed for iOS.
2. The "Controller" part means it provides functionality.

Controllers are part of the holy trinity of software development: Model, View, Controller. In an ideal world, every part of your app can be split into one of these three types: it's either a model (something that describes the data you are working with), a view (the user interface of your app), or a controller (the code that sends model data to and from the view).

In reality, things are rarely this clean, and it's extremely common to have a Fat Controller problem: lots of code that ought to be in models and views ends up in your controller. It's not ideal, but you can always go back later and rewrite the code to be neater. (Spoiler: you will never do this.)

Now, you might be thinking that you haven't written any code using split view controllers or navigation controllers, and you'd be right. This is because Apple has a dedicated tool for editing the visual layouts of your apps, and it's called Interface Builder. You'll see the file Main.storyboard in your project, so select that now to show Interface Builder.

Interface Builder (also known as the storyboard editor) is designed to show a visual flow of your program. You will need to zoom out, though: your user interface is surprisingly complicated for such a simple app! Hold down Cmd and tap the - key once to zoom out one level. You should now be able to see that there are five squares arranged on the screen: one saying Split View Controller, two saying Navigation Controller, one saying Table View and one saying "Detail view content goes here."
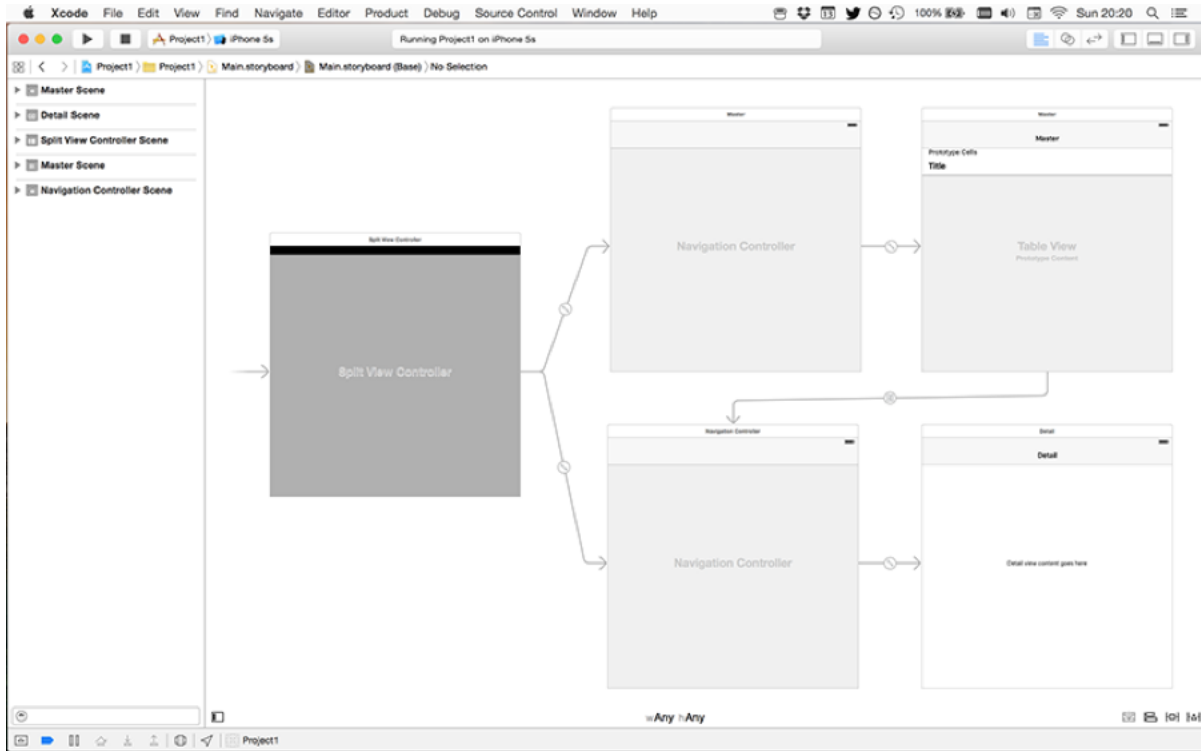
Our simple application has a storyboard with five view controllers.

Between each of the squares are arrows moving from left to right, showing the program flow. These are called *segues* (pronounced segways, like the motorised scooter.) The segue with two circles connected by a line is called a relationship segue, and it's used to connect the split view controller to both navigation controllers, and also to connect both navigation controllers to the squares immediately to their right.
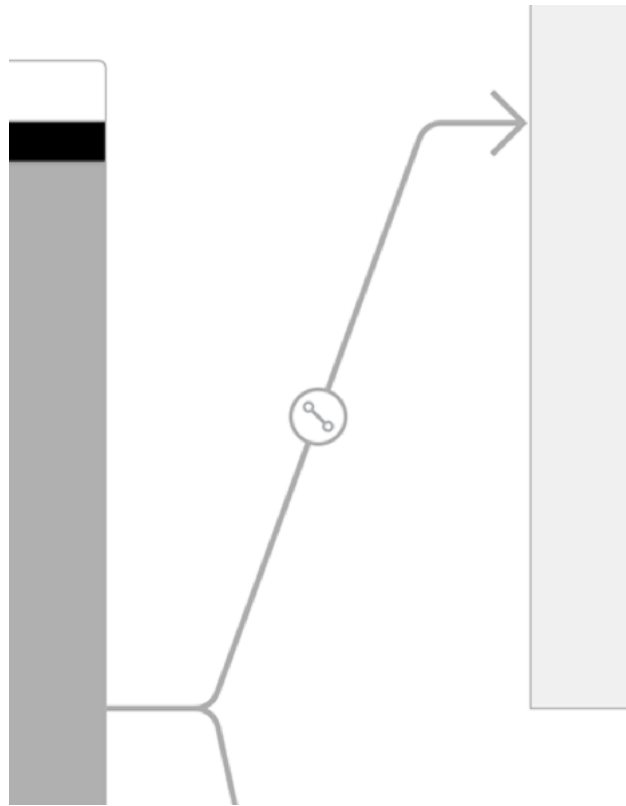
Relationship Segues have an icon showing two circles connected by a line.

We'll talk about view controllers in a moment, but for now all you need to know is that these relationship segue describe one screen of content being embedded inside another. So, the relationship segue between the top navigation controller and the table view to its right means that the table view is inside the navigation controller.

Where this gets complicated – and extraordinarily clever – is when you look at the relationship segues between the split view controller and its navigation controllers. It has two navigation controllers as its relationship, which means both are embedded inside it. Can you see them both? Nope. But that's because we asked to use the "iPhone 5s" device simulator earlier on.

If you change that to be "iPad 2" then run it again, yuo'll see something different: in portrait, you'll see a button at the top saying "Master" that, when tapped, makes the table of picture names slide in from the left side. If you press Cmd+Right Arrow on your keyboard, you'll see both the table view and the detail view at the same time, as the screen automatically splits in two.
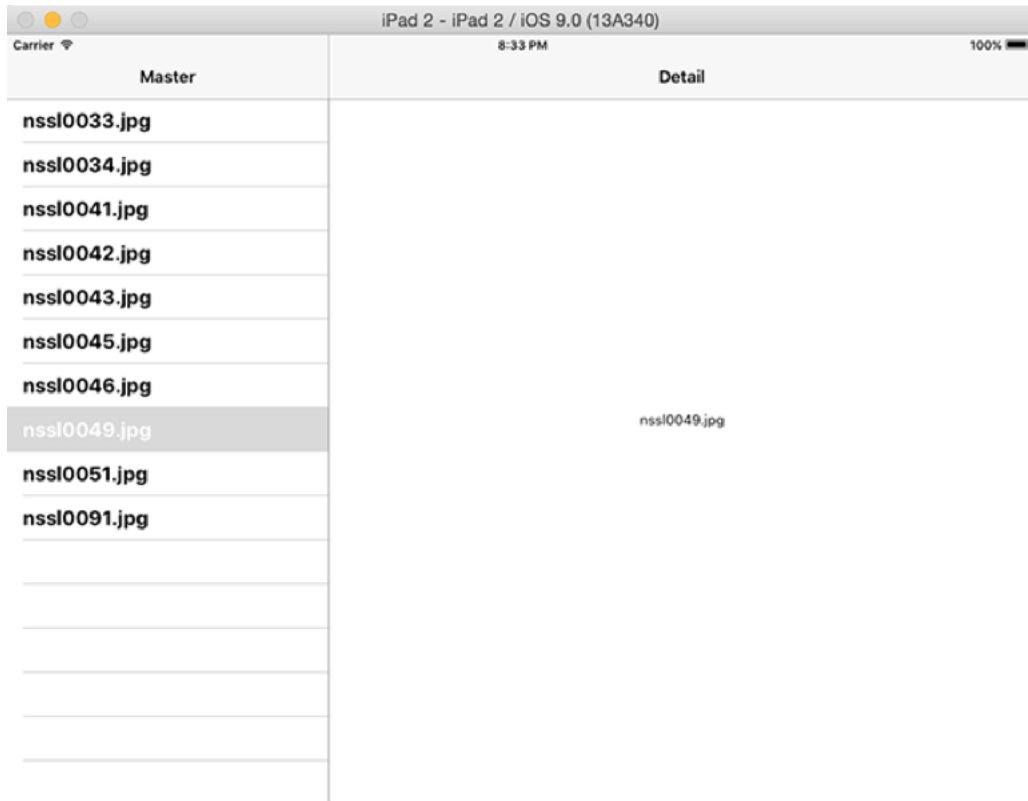
Our same project looks very different when running on iPad, even though the code is the same.

This is what Apple calls "adaptive user interfaces," which means you design your app once and iOS automatically determines the best way of showing it on various devices. We've now seen three different ways of seeing the same information: iPhone, iPad portrait and iPad landscape, all from the same code. So, even though it means having a split view controller and two navigation controllers, the end result means that our app looks great on all devices.

Back inside Interface Builder again, you'll notice there's a second kind of segue going from the table view at the top to the navigation controller on the bottom. This segue looks like two rectangles with a left arrow over them, and it's called a "Show Detail" segue, and it creates the behavior you already saw: the detail view controller slides in from the right. It's an adaptive segue (part of the "adaptive user interfaces" way of working) and it means "on iPhone, animate in from the right, but on iPad just change the detail view conntroller."
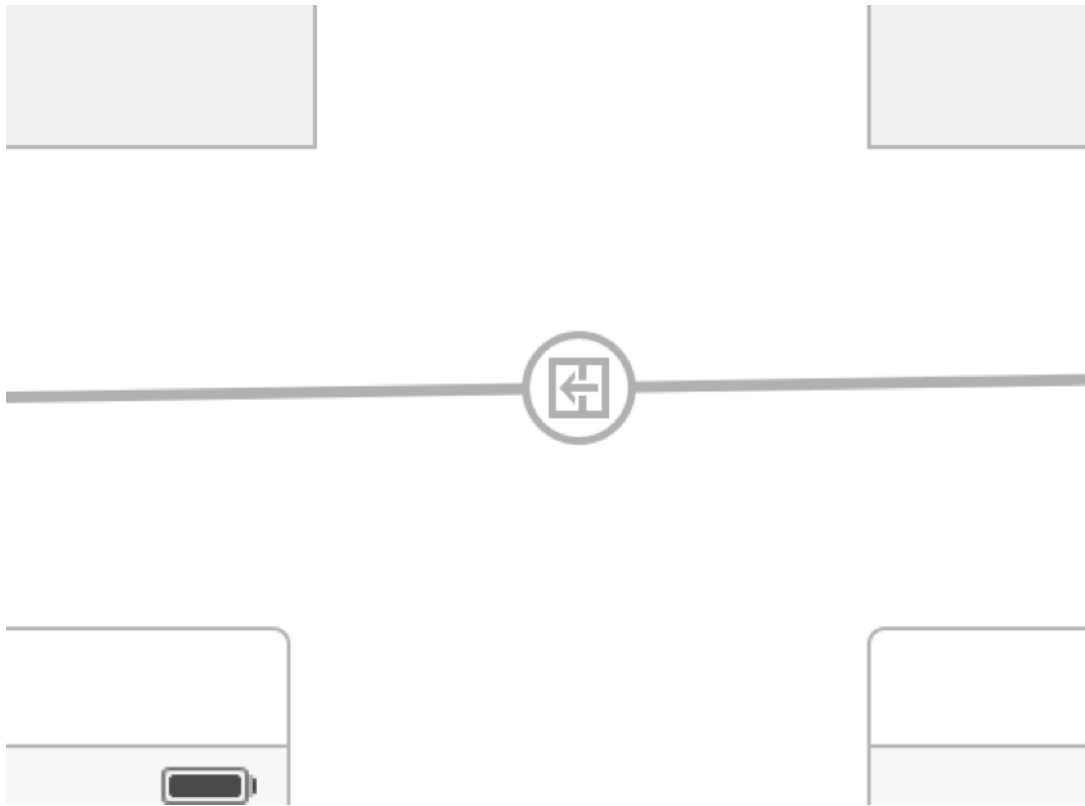
The show detail segue looks like two rectangles with a left arrow over them.

This is a good time to touch on view controllers just briefly, because they are one of the most important components in iOS. The data type **UIViewController** is used to represent all screens in your app, and comes with a huge array of functionality built in. For example, it can rotate with the device, it can respond if the device is running out of memory, it can show and hide other view controllers, and so on.

View controllers can also be used to represent parts of screens in your app, and this is done in a clever way. As an example, the mail app on iPhone has a view controller to show the messages in your inbox, and when you tap on one it brings in a new view controller to show message detail.

This set up is perfect for the iPhone, where screen space is limited and you can really only do one thing at a time. But on iPad, the same mail app shows the messages on the left and the detail on the right, which is two view controllers at the same time. Behind the scenes, this uses the exact same split view controller you're using here, and iOS just adapts to make sure the right layout is automatically used. Hurray for code reuse!

In our current app, we have five view controllers: the split view controller, two navigation controllers, our master view controller (the table view) and the detail view controller. Each of these is based on the **UIViewController** data type, but add their own functionality on top using a technique known as inheritance – each custom view controller literally *inherits* the functionality of **UIViewController**, before adding its own.

So, the split view controller is responsible for choosing the right layout depending on the device, the navigation controller adds the title bar at the top, the master controller adds the table view and the code we wrote with **NSFileManager**, and the detail controller has the text showing the date that was selected. You can have multiple levels of inheritance, which is where data type D inherits from C, which itself inherits from B, and B in turn inherits from A. It might sound a little confusing, but it does mean you get to share as much code as possible.

Now, enough thinking – time for some action. We need to change the detail view controller because right now it has some text in the middle for showing dates. We scrapped all that date code and instead want to show images, so we need to redo its user interface. Double-click on the detail view controller to zoom in and select it. In the middle you'll see the text "Detail view content goes here" inside a **UILabel**, which is a simple view type that shows text that can't be edited. Select it and press delete.

We're going to replace that label with a large **UIImageView**, which, as you might guess, is a UI component that shows images. iOS comes with an awesome collection of view types that you can drag and drop into place using storyboards, and these are all stored inside the object library. This is usually visible in the bottom-right corner, and starts with "View Controller," "Storyboard Reference," and so on. If you can't see it, press Ctrl+Alt+Cmd+3 and it should appear.

Some users have the object library set to icon view, which means you'll see just an arrow followed by a series of yellow circles. This isn't helpful for a beginner, so look to the bottom of the object library and you'll see a search box, and directly to the left of that you'll see a button that uses list view instead of icon view.

In the picture below you can see my object library, with the image view selected. At the bottom is the search area where you can filter the list of items, and just to the left of the search field is the icon/list toggle button.
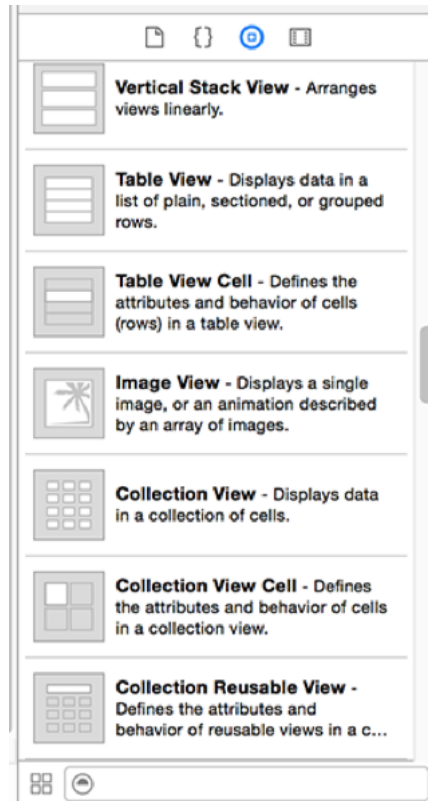
The object library, with the image view selected.

This object library contains all the built-in view types that you can use, and you'll see there a quite a few. You can look at them all you want later on, but for now use the search box: type "image" to bring up the Image View component. Click and drag the image view from the object library onto the detail view controller, then let go. Now drag its edges so that it fills the entire view controller – yes, even under the gray navigation bar that says "Detail".
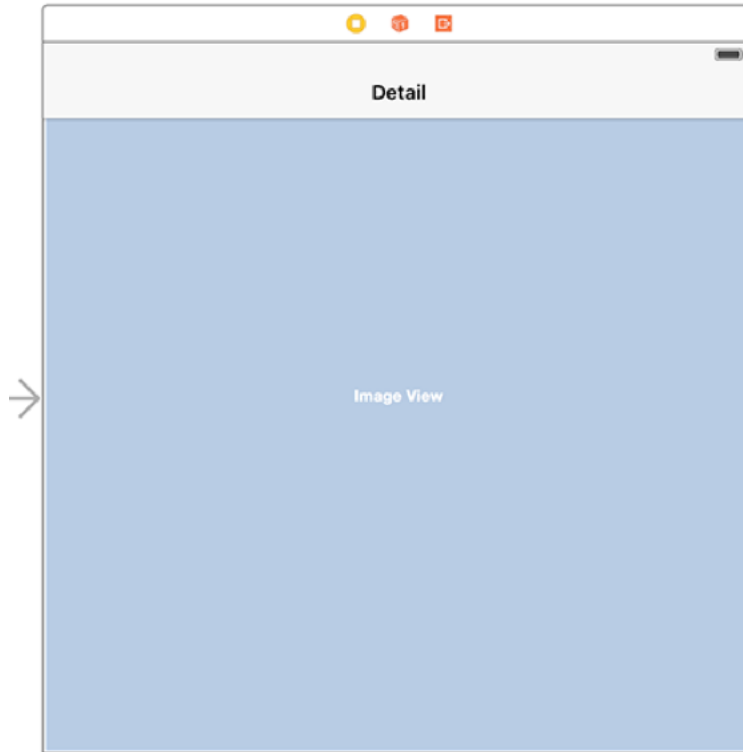
Make your UIImageView take up the full screen.

This image view has no content right now, so it's filled with a pale blue background and the word **Image View**. We won't be assigning any content to it right now, though – that's something we'll do when the program runs. Instead, we need to tell the image view how to size itself for our screen, whether that's iPhone or iPad.

This might seem strange at first, after all you just placed it to fill the view controller, and it has the same size as the view controller, so that should be it, right? Well, not quite. For a start, the detail view controller you can see on the storyboard is square, and have you ever seen a square iPhone? And what about when you have multiple devices to support: iPhone 4s, iPhone 5s and iPhone 6 all have different sizes, never mind iPad and iPad Pro, so how should the image view respond?

iOS has an answer for this. And it's a brilliant answer that in many ways works like magic to do what you want. It's called Auto Layout: it lets you define rules for how your views should be laid out, and it automatically makes sure those rules are followed. But it has two rules of its own, both of which must be followed by you:

• Your layout rules must be complete. That is, you can't specify only an X position for something, you must also specify a Y position. If it's been a while since you were at school, "X" is position from the left of the screen, and "Y" is position from the top of the screen.
• Your layout rules must not conflict. That is, you can't specify that a view must be 10 points away from the left edge, 10 points away from the right edge, and 1000 points wide. An iPhone 5 screen is only 320 points wide, so your layout is impossible. Auto Layout will try to recover from these problems by breaking rules until it finds a solution, but the end result is never what you want.

You can create Auto Layout rules – known as *constraints* – entirely inside Interface Builder, and it will warn you if you aren't following the two rules. It will even help you correct any mistakes you make by suggesting fixes!

We're going to create four constraints now: one each for the top, bottom, left and right of the image view so that it expands to fill the detail view controller regardless of its size. There are lots of ways of adding Auto Layout constraints, but the easiest way right now is to use the Pin button: a tiny, almost anonymous button in the bottom-right of Interface Builder.

There are four buttons down there: the first has three rectangles with a line pointing down, the second has two rectangles one above the other, the third has a square with lines either side, and the fourth has a triangle with lines either side. All without titles – who said Apple were geniuses at user interface design? The one we want is the Pin button, which is the third one: a square with lines either side. You can hover over it to see the tooltip, which should say Pin.

In the picture below you can see the strip of buttons at the bottom of my Interface Builder view. The button on the left toggles the document outline, and the group of four buttons towards the right control Auto Layout. You want to click the third one of the four on the right.
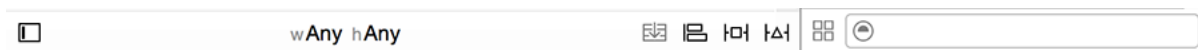
Buttons across the bottom of Interface Builder.

This Pin menu is what we're going to use now, so make sure your image view is selected then click the Pin button. A popup will appear, with the title Add New Constraints. Deselect the "Constrain to margins" checkbox, then click on the four red dotted lines near the top to turn them into solid red lines.

This will create constraints that set the distance between the image view and its superview –

the view that it's sitting inside, in this case our detail view. As you're clicking these red lines, the button at the bottom of the popup will change to "Add 4 Constraints", so click that now to dismiss the popup and add the constraints.
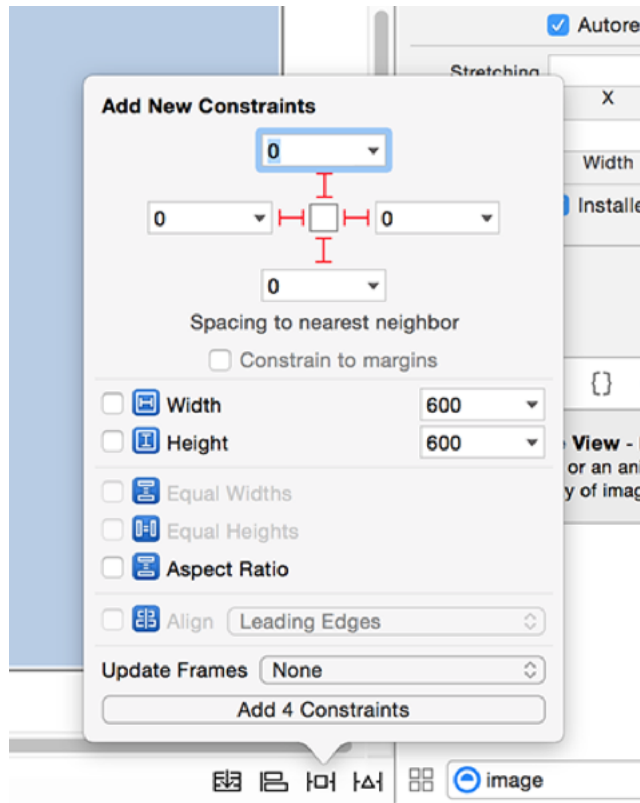
Adding Auto Layout constraints using the Pin menu.

Visually, your layout will look pretty much identical once you've added the constraints, but there are two subtle differences. First, there's a thin blue line surrounding the **UIImageView** on the detail view controller, which is Interface Builder's way of showing you that the image view has a correct Auto Layout definition.

Then in the Document Outline pane you'll see a new entry for "Constraints" beneath the image view. If you're not sure where to look, or if your document outline pane is hidden, select the image view then go to the Editor menu and choose Reveal in Document Outline to show the document outline and highlight the image view. All four constraints that were added are hidden under that Constraints item, and you can expand it to view them individually.

With the constraints added, there's one more thing to do here before we're finished with Interface Builder, and that's to connect our new image view to some code. You see, having

the image view inside the layout isn't enough – if we actually want to reference the image view inside code, we need to create a property for it that's attached to the layout.

What's a property? Well, you've already met constants (using **let**) and variables (using **var**), but these are just temporary so they last only until the method ends. A property is the name given to a constant or variable when it's attached to a data type such as our detail view controller. The image view property we're going to create exists for as long as the detail view controller does, because it belongs to it.

The Xcode template project had a **UILabel** in the detail view controller, and it had a corresponding property attached. We're going to delete that property, then create a new one.

To make this process a little bit easier, Xcode has a special display layout called the Assistant Editor. This splits your Xcode editor into two: the view you had before on top, and a related view at the bottom. In this case, it's going to show us Interface Builder on top, and the code for the detail view controller below.

Xcode decides what code to show based on what item is selected in Interface Builder, so make sure the image view is selected now and choose View > Assistant Editor > Show Assistant Editor from the menu. You can also use the keyboard shortcut Alt+Cmd+Return to do the same, or click the assistant editor button. This is the second of six button at the top-right of your Xcode window, and looks like two overlapping circles.

In the picture below, you can see the buttons in the top-right of my Xcode window when I'm in Interface Builder. The first row are the standard editor, the assistant editor, the version editor, then three buttons to control showing and hiding panes on the bottom, left and right of Xcode. The second row shows various Interface Builder inspectors, with the third, fourth and fifth inspectors being the most commonly used.
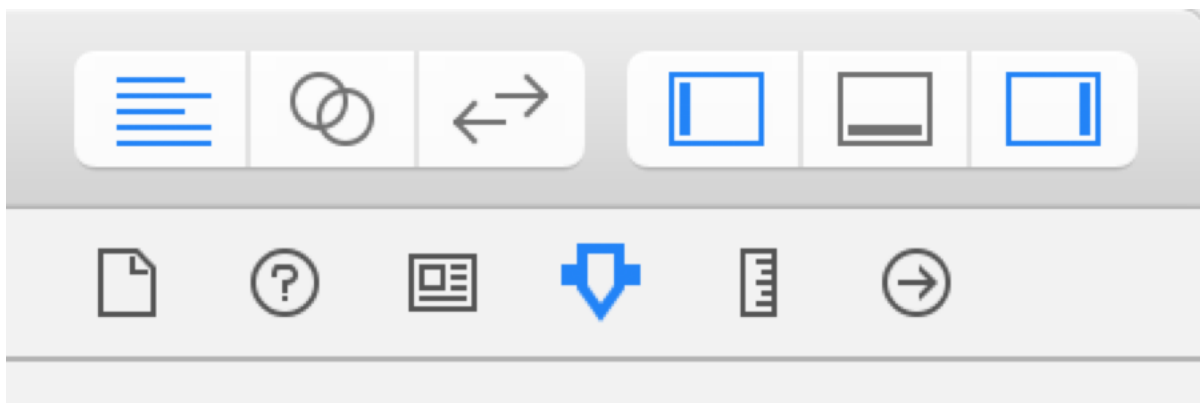
The assistant editor looks like two overlapping circles.

You should now see the detail view controller in Interface Builder in the top pane, and in the bottom pane the source code for a file we haven't looked at yet: DetailViewController.swift. This is (predictably) the code that manages the detail view controller, and you'll see in there near the top this following line:

```
@IBOutlet weak var detailDescriptionLabel: UILabel!
```
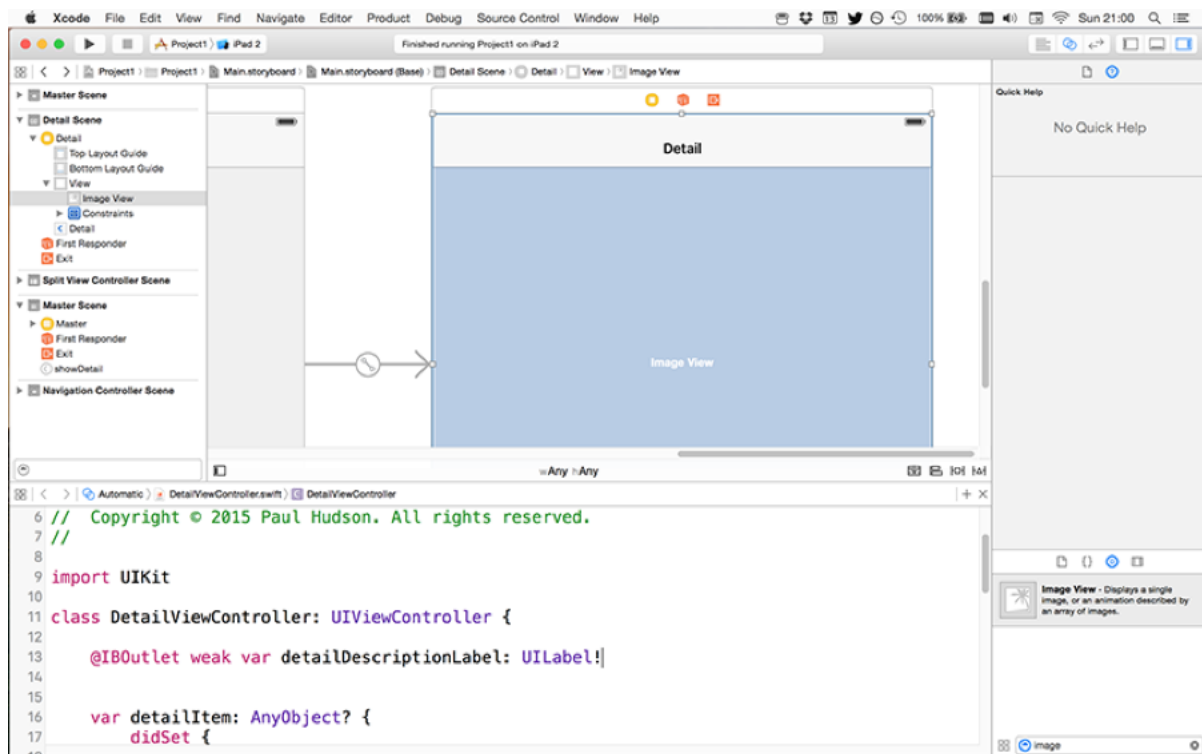

Using the Assistant Editor you can view your layout and code at the same time.

Some bits of that are new, so let's break down the whole line:

• **@IBOutlet**: This attribute is used to tell Xcode that there's a connection between this line of code and Interface Builder.
• **weak**: This tells iOS that we don't want to own the object in memory. This is because the object has been placed inside a view, so the view owns it.
• **var**: This declares a new variable or variable property. We already used **let** to declare

constants, and this is how you declare variables. Remember, a constant's value can be set only once, whereas a variable's value can change.

  • **detailDescriptionLabel**: This was the property name assigned to the **UILabel**. Note the way capital letters are used: variables and constants should start with a lowercase letter, then use a capital letter at the start of any subsequent words. For example, **myAwesomeVariable**. This is sometimes called camel case.

  • **UILabel!**: This declares the property to be of type **UILabel**, and again we see the implicitly unwrapped optional symbol: !. This means that that **UILabel** may be there or it may not be there, but we're certain it definitely will be there by the time we want to use it.

If you were struggling to understand implicitly unwrapped optionals (don't worry; they are complicated!), this code might make it a bit clearer. You see, when the detail view controller is created, its view hasn't been loaded yet – it's just some code running on the CPU.

When the basic stuff has been done (allocating enough memory to hold it all, for example), iOS goes ahead and loads the layout from the storyboard, then connects all the IBOutlets from the storyboard to the code.

So, when the detail controller is first made, the **UILabel** doesn't exist because it hasn't been created yet – but we still need to have some space for it in memory. At this point, the property is **nil**, or just some empty memory. But when the view gets loaded and the outlet gets connected, the **UILabel** will point to a real **UILabel**, not to **nil**, so we can start using it.

In short: it starts life as **nil**, then gets set to a value before we can use it, so we're certain it won't ever be **nil** by the time we want to use it – a textbook case of implicitly unwrapped optionals.

(PS: If you still don't understand implicitly unwrapped optionals, that's perfectly fine – keep on going and they'll become clear over time.)

Back to the project: we don't have a **UILabel** any more, so you can delete that whole line of code. Don't worry: learning all that **@IBOutlet** stuff wasn't in vain, because we're about to create a new outlet for the image view. This is done, again, visually: hold down Ctrl then click and drag from the **UIImageView** in the top pane to the where the **@IBOutlet** line was in the bottom pane. If you've forgotten, it was just below the line that says, **class DetailViewController: UIViewController {**.

As you're Ctrl-dragging from the image view, a blue line will appear. When you move over the code, a tooltip will appear next to your pointer saying "Insert Outlet or Outlet Collection," and

a second blue line will appear horizontally to show exactly where the outlet will be inserted.
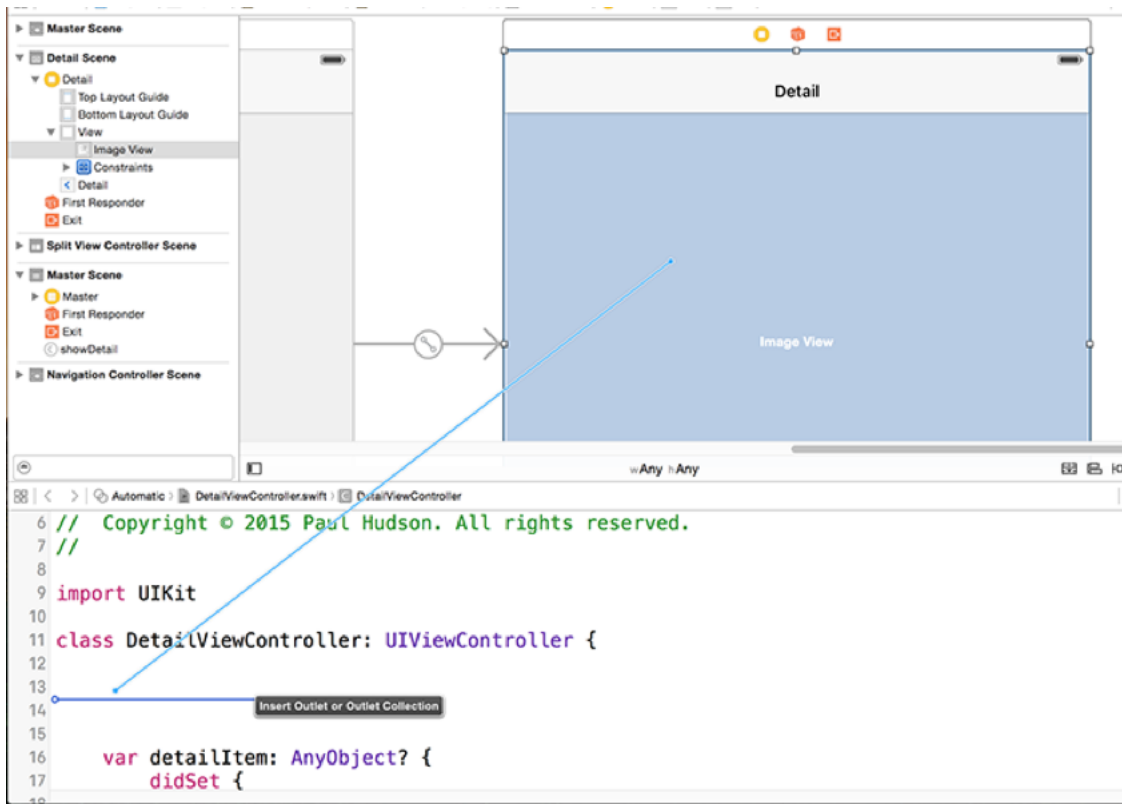
Ctrl+drag from your storyboard into the code, and a message will appear saying 'Insert Outlet or Outlet Collection'.

Let go of your mouse button and a popup will appear asking you to fill in some information. Right now, all you need to do is give your outlet a name, so call it **detailImageView** and click Connect. You'll see this line of code gets written for you:

```
@IBOutlet weak var detailImageView: UIImageView!
```
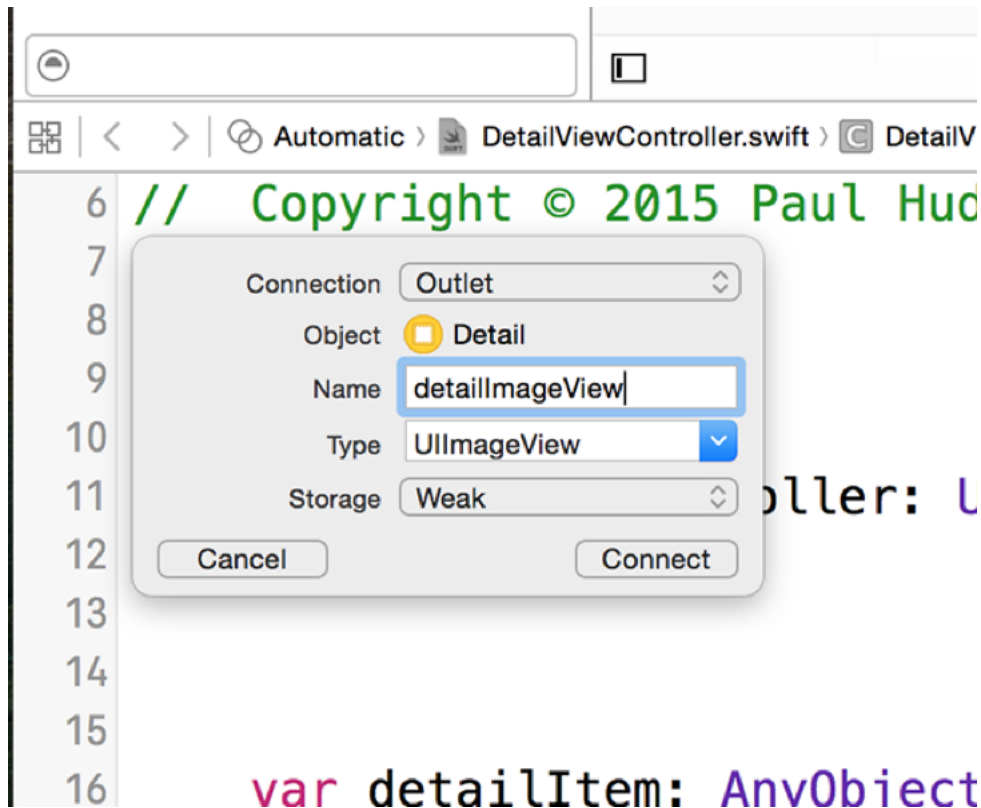
Name your UIImageView outlet detailImageView.

Yes, that's mostly the same as the previous code, but now it's an image view rather than a label. Crucially, if you look to the left of this line of code, you'll see a gray circle with a ring around it. This is Xcode's way of telling you that this outlet has a connection in place in Interface Builder. If it didn't have a connection, you'd just see an empty ring.

Sending new data

In making this layout change, we've broken our code – you'll notice there's a red line in the scroll bar of our code editor, and also a red warning symbol at the top of the Xcode window. This is because other parts of DetailViewController.swift reference the **UILabel**, and we've just deleted that property. So we need to make a few changes to this file so that it knows how to handle our new data, then make some final changes to MasterViewController.swift in order to pass the new data correctly.

First, look for this code in DetailViewController.swift:

```
var detailItem: AnyObject? {

  didSet {

    // Update the view.

    self.configureView()

  }

}
```

This declares a property called **detailItem**, giving it the type **AnyObject?** – that's Swift's way of saying it might be any kind of object, or it might nothing at all. But this property has a twist, because it has a property observer attached, in the form of **didSet**.

The **didSet** property observer is a block of code that will be executed any time this property's value has been changed. There's also a corresponding **willSet** property observer that can execute code just before a property is changed, but it's not used as often as **didSet**.

In this code, **didSet** is being used to call **self.configureView()**, which means "call the **configureView()** method on myself." The **self.** isn't actually needed, so you can delete it if you want. You might be interested to know that there are two trains of thought with regards to using "self." when referring to variables and methods.

The first group of people never like to use **self.** unless it's required, because when it's required it's actually important and meaningful, so using it in places where it isn't required can confuse matters. The other group of people always like to use **self.** whenever it's possible, even when it's not required. To be fair to this group, using **self.** everywhere was

good practice in Objective C, and it can be a hard habit to break.

Before we fix the errors in our code, we're going to introduce some more. You see, **detailItem** is defined as **AnyObject?**, which is silly. We know exactly what kind of data is going to be sent across from the master view controller, because we're working with an array of strings. The only thing **detailItem** can be is a **String**, albeit an optional one.

So, change **AnyObject?** to be **String?**, then press Cmd+B to compile your project. This creates another error, but that's OK because we're about to fix it!

The error will be in the **configureView()** method of DetailViewController.swift. Here's its current code:

```
if let detail: AnyObject = self.detailItem {

  if let label = self.detailDescriptionLabel {

    label.text = detail.description

  }

}
```

The **detailItem** property is an optional data type: it was **AnyObject?** and now it's **String?**. This means we need to unwrap the value before we can use it – we need to check whether there is a value there, and, if so, extract it. The most common way to extract values from optionals is to use an "if let" condition, which is what this current code is doing. For example:

```
if let foo = bar {

  doStuff(foo)

}
```

That code means "see if the **bar** variable contains a value, and if it does unwrap that value and place it inside another variable called **foo**, then call **doStuff()** and pass in **foo** as a parameter." So, if **bar** was of type **String?**, **foo** will be of type **String** because it won't be optional any more. If **bar** didn't have a value, then **doStuff()** would never be called. The advantage of this "if let" syntax is that it both checks for the existence of a value and

unwraps it all it once.

With all that in mind, let's take a look at the code again:

```
if let detail: AnyObject = self.detailItem {
  if let label = self.detailDescriptionLabel {
    label.text = detail.description
  }
}
```

You'll see that it first checks whether **detailItem** has a value, and if so it unwraps it into a new constant called **detail**. It then checks whether **detailDescriptionLabel** has a value, and if so it unwraps it into another new constant called **label**. If both of these had values and were unwrapped, then the label's **text** property is set to be **detail.description**. In the case of the dates that were originally in there, this would print the date as text. These lines are erroring because we don't have a label any more, so please delete them and replace them with this:

```
if let detail = self.detailItem {
  if let imageView = self.detailImageView {
    imageView.image = UIImage(named: detail)
  }
}
```

This changes a few things:

 • We no longer need to declare the **detail** constant as **AnyObject**, because **self.detailItem** is of type **String?** not **AnyObject?**.
 • We unwrap the property **detailImageView** into **imageView**, because we have an image view rather than a label now.
 • Rather than setting the text of the label, we're setting the image of the image view.

This new code also introduces a new data type, called **UIImage**. This doesn't have "View" in its name like **UIImageView** does, so it's not a view – it's not something that's actually visible to users. Instead, **UIImage** is the data type you'll use to load image data, such as PNG or JPEGs.

When you create a **UIImage**, it takes a parameter called **named** that lets you specify the name of the image to load. **UIImage** then looks for this filename in your app's bundle, and loads it. By passing in the **detail** constant here, which was created by unwrapping **detailItem**, this will load the image that was selected by the user.

That's the only error inside DetailViewController.swift, but you're probably curious how the selected image name gets passed from the master view controller to the detail. To find out, go to MasterViewController.swift. If you're still using the assistant editor view, switch back to the standard editor by pressing Cmd+Return, by clicking the first of the six buttons in the top-right of the Xcode window, or by choosing the View > Standard Editor > Show Standard Editor menu item.

The image name is passed from MasterViewController.swift inside the **prepareForSegue()** method. This gets called when a segue is about to execute, and gives you a chance to configure the new view controller with information for it to use. In our project, this is where we tell the detail view controller what item was selected, and it's done using this existing code:

```
override func prepareForSegue(segue: UIStoryboardSegue, sender:
AnyObject?) {

  if segue.identifier == "showDetail" {

    if let indexPath = self.tableView.indexPathForSelectedRow {

      let object = objects[indexPath.row]

      let controller = (segue.destinationViewController as!
UINavigationController).topViewController as! DetailViewController

      controller.detailItem = object

      controller.navigationItem.leftBarButtonItem =
self.splitViewController?.displayModeButtonItem()

      controller.navigationItem.leftItemsSupplementBackButton = true

    }

  }
```

```
}
```

This is pretty complicated code, but only these three lines are important:

```
let object = objects[indexPath.row]

let controller = (segue.destinationViewController as!
UINavigationController).topViewController as! DetailViewController

controller.detailItem = object
```

To make it easier to understand, we're going to rewrite it. Replace those three with these:

```
let navigationController = segue.destinationViewController as!
UINavigationController

let controller = navigationController.topViewController as!
DetailViewController

controller.detailItem = objects[indexPath.row]
```

The first line looks for the destination view controller of the segue, which is where the "show detail" segue pointed to. If you remember, that was the navigation controller, so this line of code forces that to be treated as a navigation controller using **as! UINavigationController**. This means "I know you *think* that the destination view controller is a regular **UIViewController**, but trust me: it's actually a **UIViewController**."

This is required because when **prepareForSegue()** is called the new view controller that is about to be shown has already been created, and it's our job to customise it with any data it needs. We can access the new view controller by reading the **segue.destinationViewController** property, but the problem is that it could be any kind of view controller – Swift knows that it's definitely some sort of **UIViewController**, but has no idea that behind the scenes it's actually an instance of **UIViewController**.

Sometimes this doesn't matter. If you just want to show something and don't care to modify its values, you don't need to do any typecasting or indeed do any of this code – the segue

will just happen. But **prepareForSegue()** exists specifically so that we can do any customization before the new view controller is shown, and in our case that means setting a property on the detail view controller.

We need to typecast **destinationViewController** to be **UINavigationController** because of what happens in line two: a navigation controller has a property called **topViewController** that points to whichever view controller it is showing right now. For us that's our detail view controller, but iOS doesn't know that – it thinks **topViewController** is just a regular **UIViewController**. So we need to use **as!** again to override this: we're telling Swift that this object is definitely our **DetailViewController** data type.

Finally, once we've found the navigation controller (line 1), found the detail view controller inside it (line 2), we can update the **detailItem** property of that detail view controller to be the image that was selected. This is done by setting its **detailItem** property to be **objects[indexPath.row]**.

The **objects** property, you might recall, is an array of strings that we loaded from the app bundle. Arrays are an ordered collection of values, one after the other, and you can read them out by specifying their position in the array counting from 0. So, the first item in the **objects** array is **objects[0]**, the second is **objects[1]**, the tenth is **objects[9]** and so on.

So what's **indexPath.row**? Well, **indexPath** is set a couple of lines above to be whatever row is selected in the table – i.e., the row that the user just tapped to trigger the segue. The **row** property of that **indexPath** is the position inside the table, so by using **objects[indexPath.row]** we're saying "get the item in the **objects** array at the position where the user just tapped."

Final tweaks

At this point you have a working project: you can press Cmd+R to run it, flick through the images in the table, then tap one to view it. But before this project is complete, there are two other small changes we're going to make that makes the end result a little more polished.

First, you might have noticed that all the images are being stretched to fill the screen. This isn't an accident – it's the default setting of **UIImageView**. This takes just a few clicks to fix: choose Main.storyboard, select the image view in the detail view controller, then choose the Attributes Inspector. This is in the right-hand pane, near the top, and is the fourth of six inspectors, just to the left of the ruler icon.

If you don't fancy hunting around for it, just press Cmd+Alt+4 to bring it up. The stretching is caused by the view mode, which is a dropdown button that defaults to "Scale to Fit." Change that to be "Aspect Fit," and this first problem is solved.
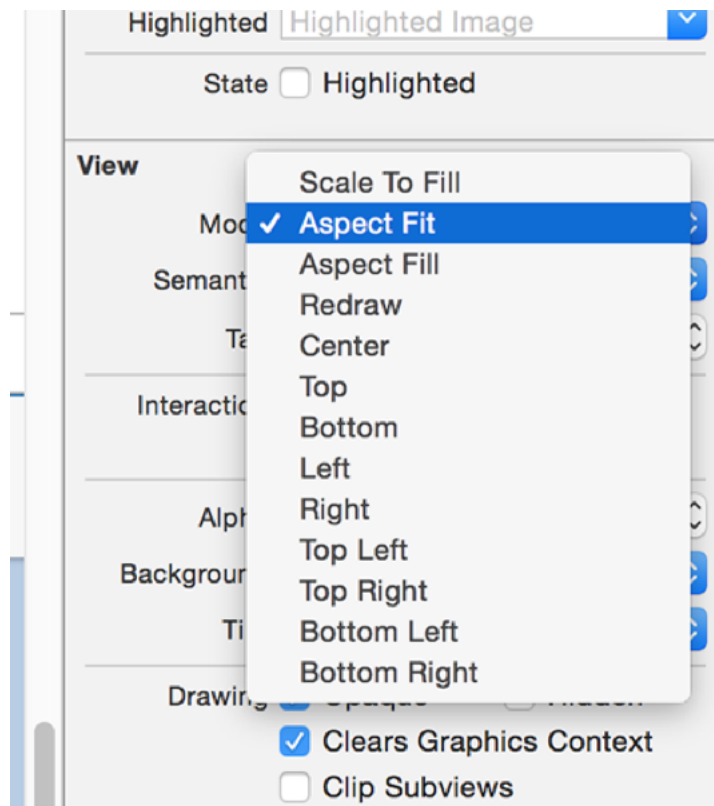
The Aspect Fit content mode for UIImageViews forces them to resize their images so they are fully visible.

If you were wondering, Aspect Fit sizes the image so that it's all visible. There's also Aspect

Fill, which sizes the image so that there's no space left blank – this usually means cropping either the width or the height. If you use Aspect Fill, the image effectively hangs outside its view area, so you should make sure you enable Clip Subviews to avoid the image overspilling.

The second change we're going to make is to allow users to view the images fullscreen, with no navigation bar getting in their way. There's a really easy way to make this happen, and it's a property on **UINavigationController** called **hidesBarsOnTap**. When this is set to true, the user can tap anywhere on the current view controller to hide the navigation bar, then tap again to show it.

Be warned: you need to set it carefully when working with iPhones. If we had it set on all the time then it would affect taps in the table view, which would cause havoc when the user tried to select things. So, we need to enable it when showing the detail view controller, then disable it when hiding.

You already met the method **viewDidLoad()**, which is called when the view controller's layout has been loaded. There are several others that get called when the view is about to be shown, when it has been shown, when it's about to go away, and when it has gone away. These are called, respectively, **viewWillAppear()**, **viewDidAppear()**, **viewWillDisappear()** and **viewDidDisappear()**. We're going to use **viewWillAppear()** and **viewWillDisappear()** to modify the **hidesBarsOnTap** property so that it's set to true only when the detail view controller is showing.

Open DetailViewController.swift, then add these two new methods directly below the end of the **viewDidLoad()** method:

```
override func viewWillAppear(animated: Bool) {
  super.viewWillAppear(animated)
  navigationController?.hidesBarsOnTap = true
}


override func viewWillDisappear(animated: Bool) {
  super.viewWillDisappear(animated)
  navigationController?.hidesBarsOnTap = false
}
```

There are some important things to note in there:

 • We're using override for each of these methods, because they already have defaults defined in **UIViewController** and we're asking it to use ours instead. Don't worry if you aren't sure when to use override and when not, because if you don't use it and it's required Xcode will tell you.
 • Both methods have a single parameter: whether the action is animated or not. We don't really care in this instance, so we ignore it.
 • Both methods use the **super** prefix: **super.viewWillAppear()** and **super.viewWillDisappear()**. This means "tell my parent data type that these methods were called." In this instance, it means that it passes the method on to **UIViewController**, which may do its own processing.
 • All view controllers have an optional property called **navigationController**, which, if set, lets us reference the navigation controller we are inside. It's optional because not all view controllers are inside a navigation controller. In Swift you can use a question mark inside a statement to evaluate an optional, and execution of the line will continue only if the optional could be unwrapped. So, if we're not inside a navigation controller, the **hidesBarsOnTap** lines will do nothing.

If you run the app now, you'll see that you can tap to see a picture full size, and it will no longer be stretched. While you're viewing a picture you can tap to hide the navigation bar at the top, then tap to show it again. We're done!

Wrap up

This has been a very simple project in terms of what it can do, but you've also learned a huge amount about Swift, Xcode and storyboards. I know it's not easy, but trust me: you've made it this far, so you're through the hardest part.

To give you an idea of how far you've come, here are just some of the things we've covered: constants and variables, method overrides, table views and image views, app bundles, **NSFileManager**, typecasting, arrays, loops, optionals, view controllers, storyboards, outlets, Auto Layout, **UIImage** and more.

Yes, that's a *huge* amount, and to be brutally honest chances are you'll forget half of it. But that's OK, because we all learn through repetition, and if you continue to follow the rest of this series you'll be using all these and more again and again until you know them like the back of your hand.

If you want to spend a little more time on this app, try investigating the **title** property of your two view controllers. This lets you customise the text that appears in the navigation bar at the top when the view controller is being shown – it's easy to make this show the name of the image that was selected.

Project 2
Guess the Flag

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

Setting up

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

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

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

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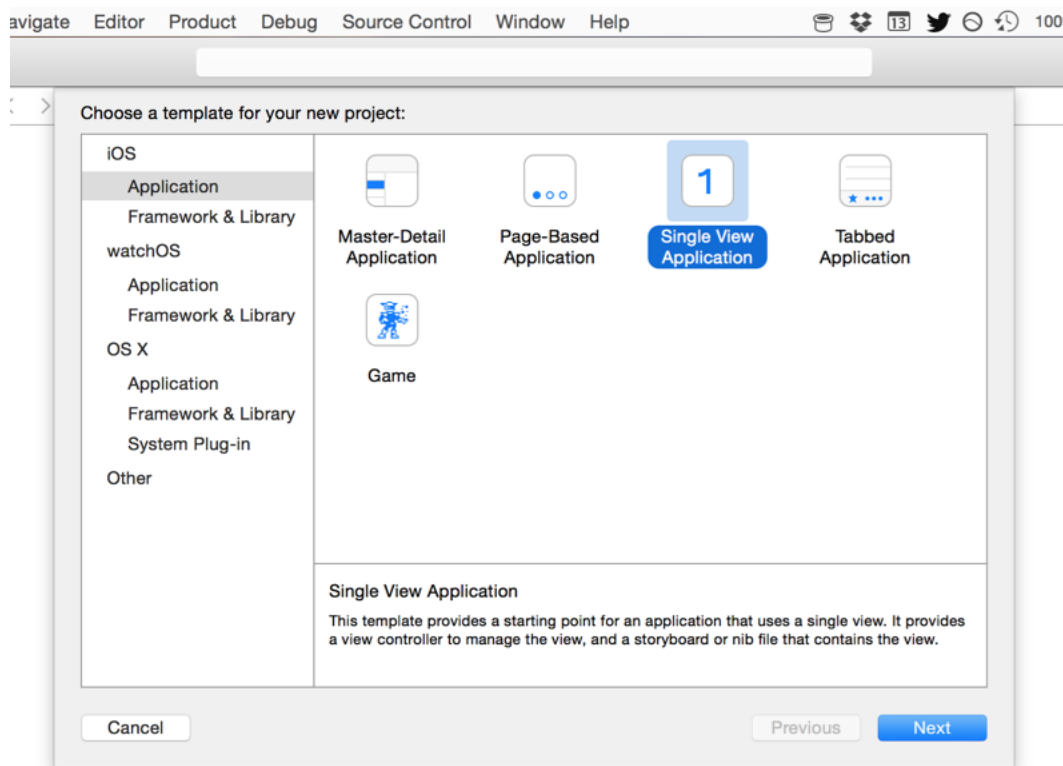
Designing your layout

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

A Single View Application project template gives you one **UIViewController**, called **ViewController**, and a storyboard called Main.storyboard that contains the layout for our single view controller. Choose that storyboard now to open Interface Builder, and you'll see a big, blank space ready for your genius to begin.
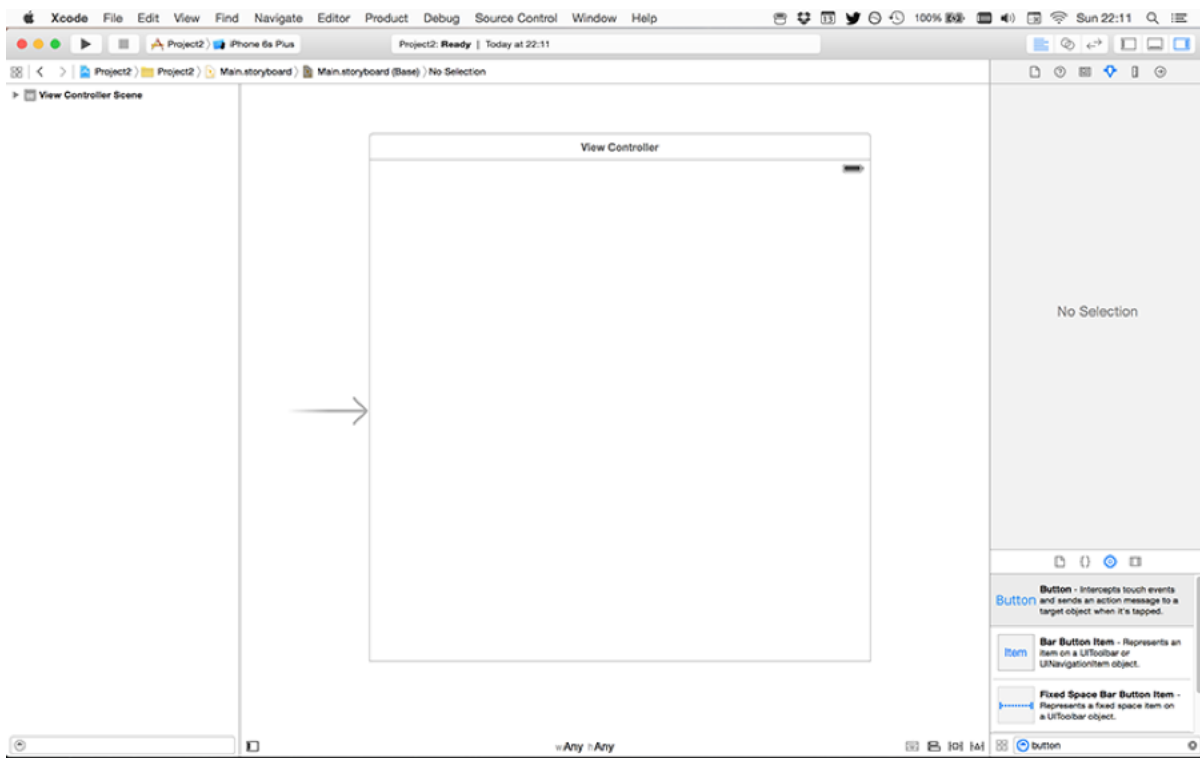
The Single View Application gives you one, large, empty view controller to customize.

In our game, we're going to show users three flags, with the name of the country to guess shown in the navigation bar at the top. What navigation bar? Well, there isn't one, or at least not yet. Single View Application projects don't come with a navigation controller as standard, but it's trivial to add one: click inside the view controller, then go to the Editor menu and choose Embed In > Navigation Controller.

With the new navigation controller in place, scroll so you can see our empty view controller

again, and draw out three **UIButton**s onto the canvas. This is a new view type, but as you might imagine it's just a button that users can tap. Each of them should be 200 wide by 100 high. You can set these values exactly by using the size inspector in the top-right of the Xcode window.
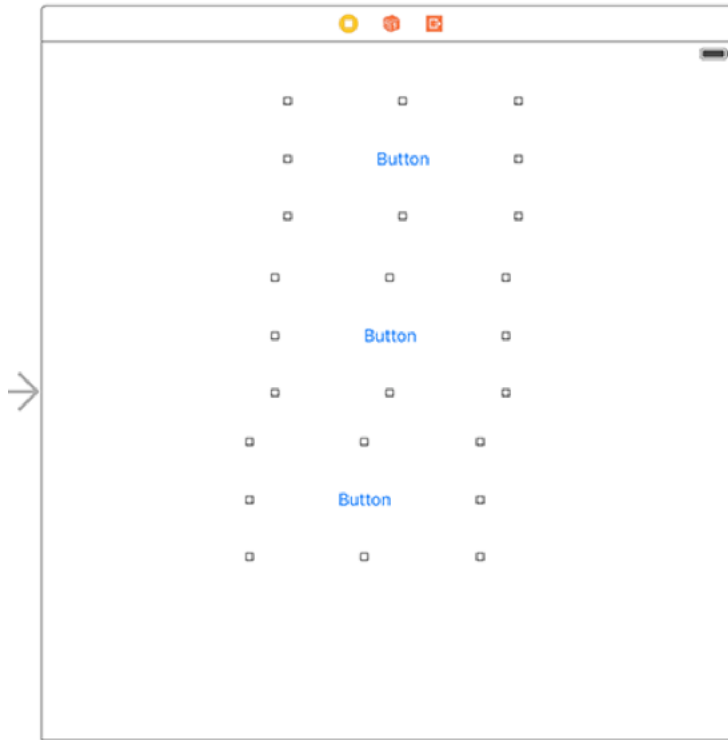
Draw three buttons onto your view.

In the "old days" of iOS 6 and earlier, these **UIButtons** had a white background color and rounded edges so they were visibly tappable, but from iOS 7 onwards buttons have been completely flat with just some text. That's OK, though; we'll make them more interesting soon.

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

In the picture below you can see the size inspector, which is the quickest and easiest way to position and size views if you know exactly where you want them.
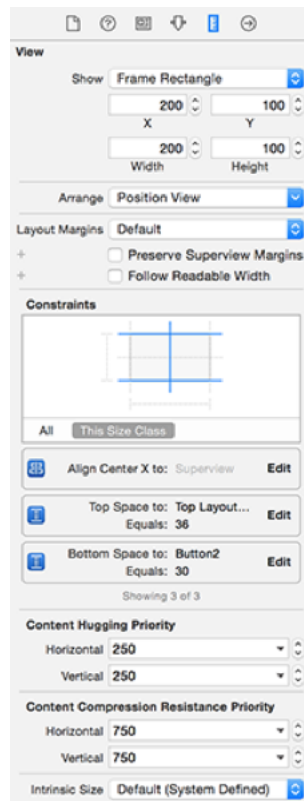
Use the size inspector to enter exact values for X, Y, width and height – it's much faster than dragging things around.

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

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

Select the top button, then Ctrl-drag from there directly upwards to just outside itself – i.e., onto the white area of the view controller. As you do this, the white area will turn blue to show that it's going to be used for Auto Layout.
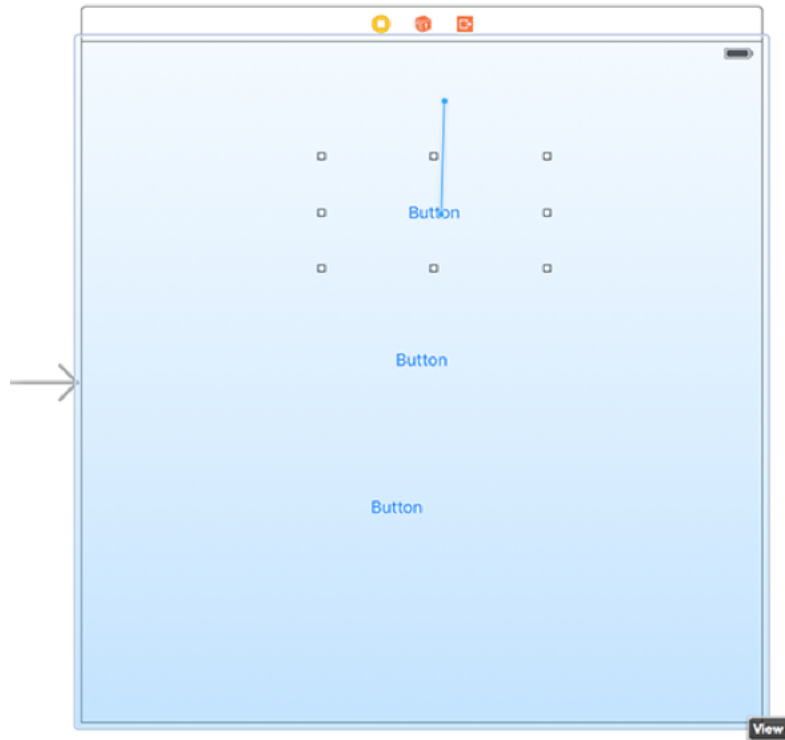
The Single View Application gives you one, large, empty view controller to customize.

When you let go of the mouse button, you'll be presented with a list of possible constraints to create. In that list are two we care about: "Top Space to Top Layout Guide" and "Center Horizontally in Container."

You have two options when creating multiple constraints like this: you can either select one then Ctrl-drag again and select the other, or you can hold down shift before selecting an item in the menu, and you'll be able to select more than one at a time. That is, Ctrl-drag from the button straight up to the white space in the view controller, let go of the mouse button and Ctrl so the menu appears, then hold down Shift and choose "Top Space to Top Layout Guide" and "Center Horizontally in Container."
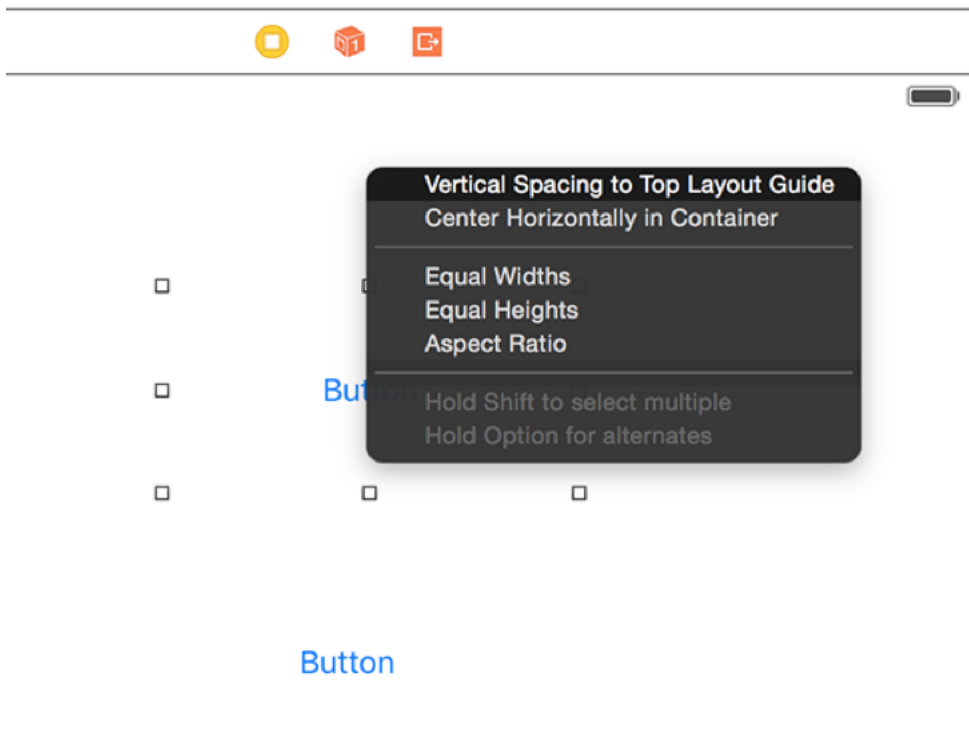
Xcode will ask you which Auto Layout constraints you want to make.

That's the first flag complete, so before we go any further let's bring it to life by adding some example content so you can see how it looks.

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

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

Select all 36 flag pictures from the project files, and drag them into the Xcode window to beneath where it says "AppIcon" in our asset catalog. This will create 12 new entries in the asset catalog, one for each country.
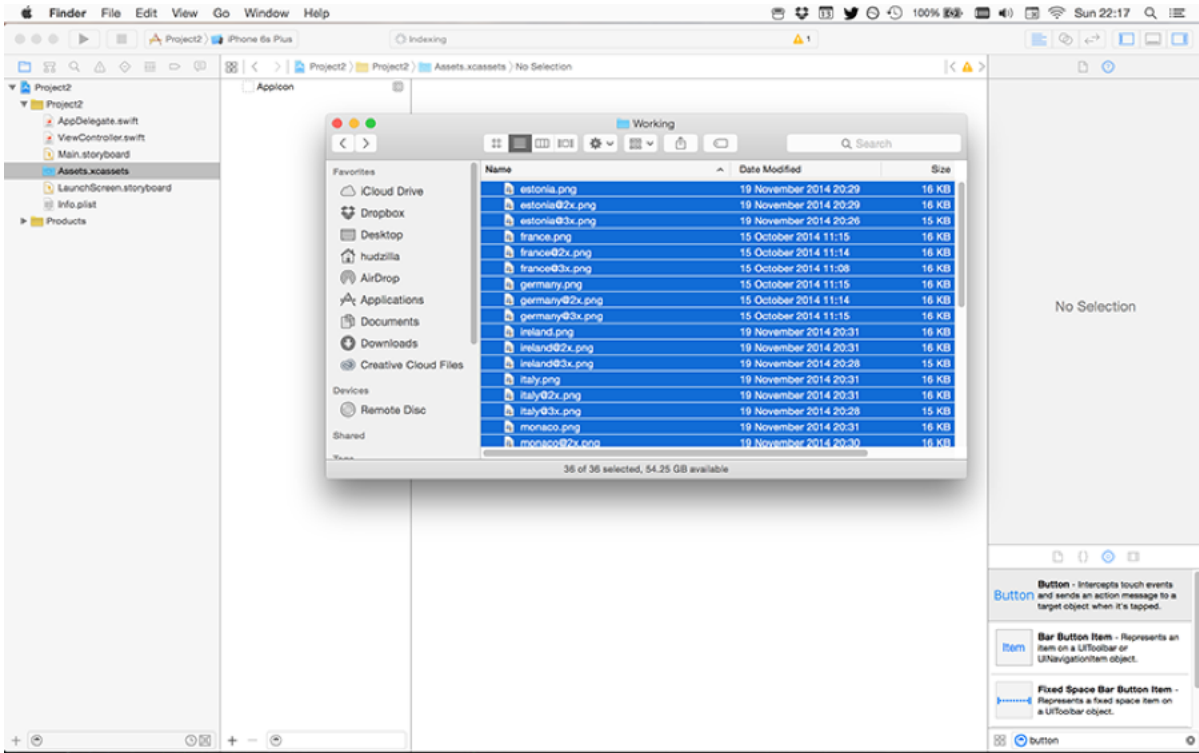
Select all the flag images in Finder.
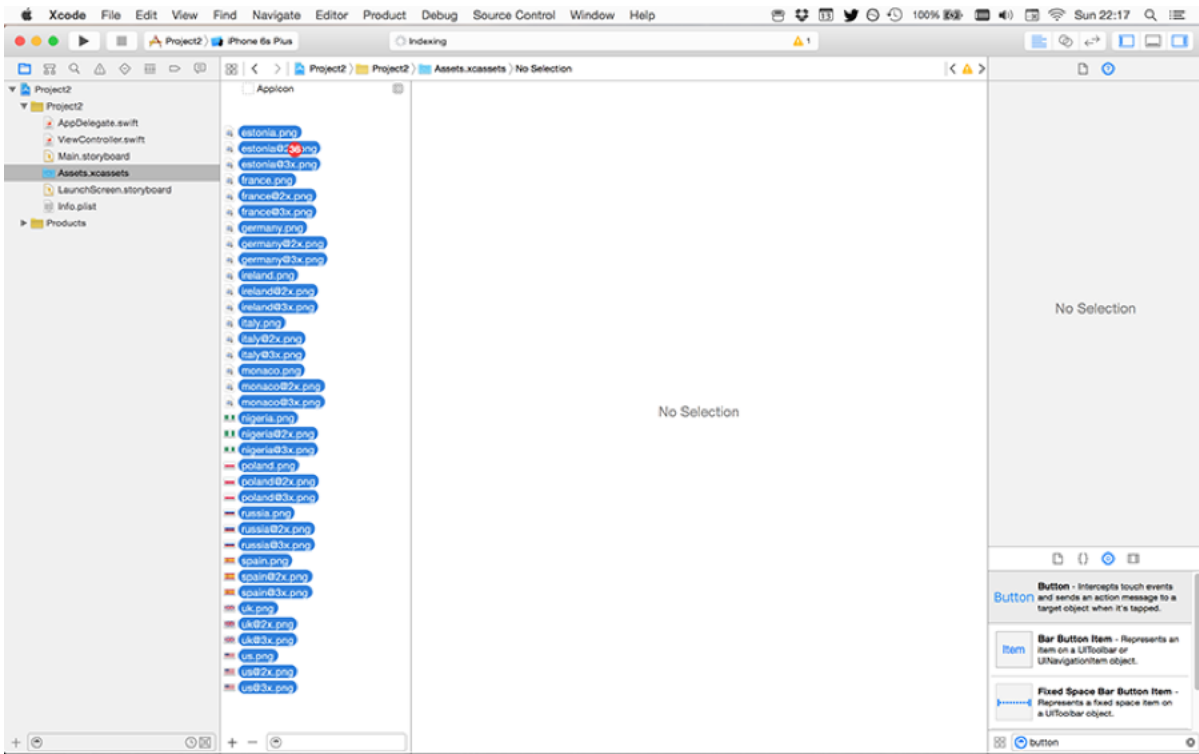
Drag them into the asset catalog inside Xcode.
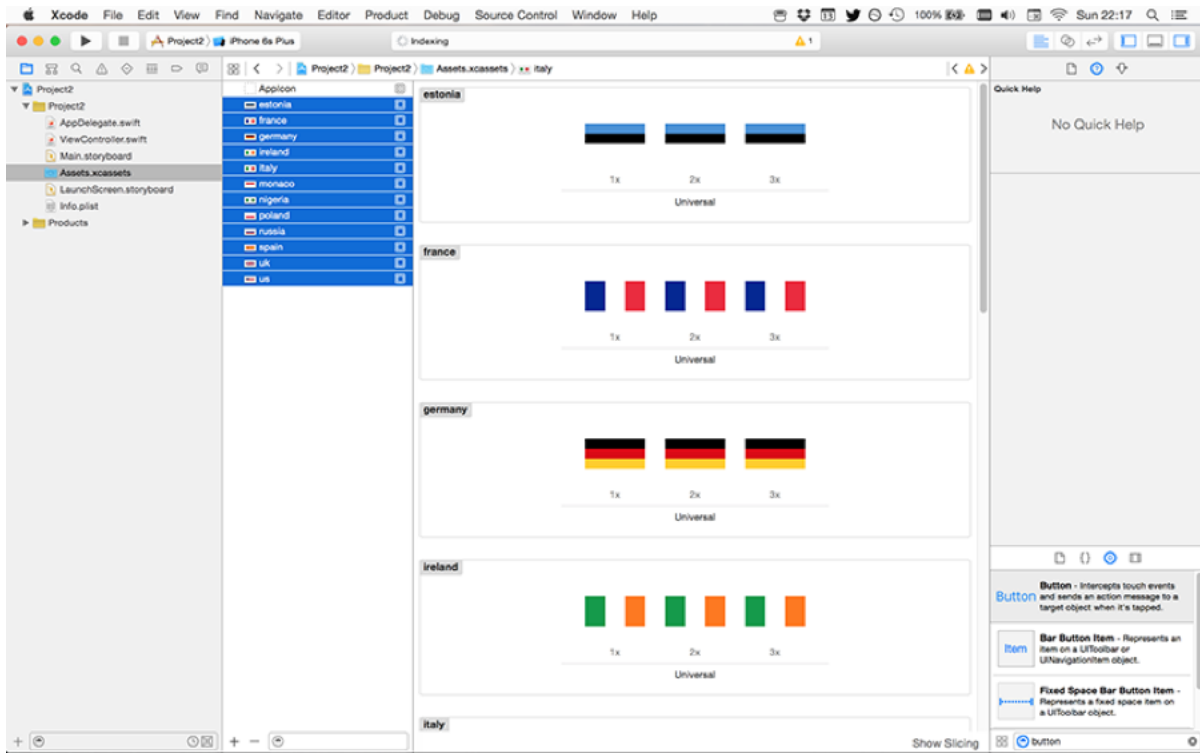
When you let go, 12 new entries will be created in the asset catalog, one for each country.

As much as I hate diversions, this one is important: iOS assets come in the sizes 1x, 2x and 3x. A 1x image is just called its regular name, eg hello.png, and is used on all non-retina devices – that's iPhone, iPhone 3G, iPhone 3GS, iPad, iPad 2 and iPad Mini.

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

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

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

image depending on the user's device and what images you have provided.

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

All this is important because when we imported the images into our asset catalog, they were automatically placed into 1x, 2x and 3x buckets. This is because I had named the files correctly: france.png, france@2x.png, france@3x.png, and so on.

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

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

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

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

This is easy enough to do – from the four buttons at the bottom of Interface Builder, click the

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

The last step before we're finished with Interface Builder for now is to add some outlets for our three flag buttons, so that we can reference them in code. Activate the assistant editor, then Ctrl-drag from the first flag to create an outlet called **button1**, then from the second flag to create **button2**, and from the third flag to create **button3**.
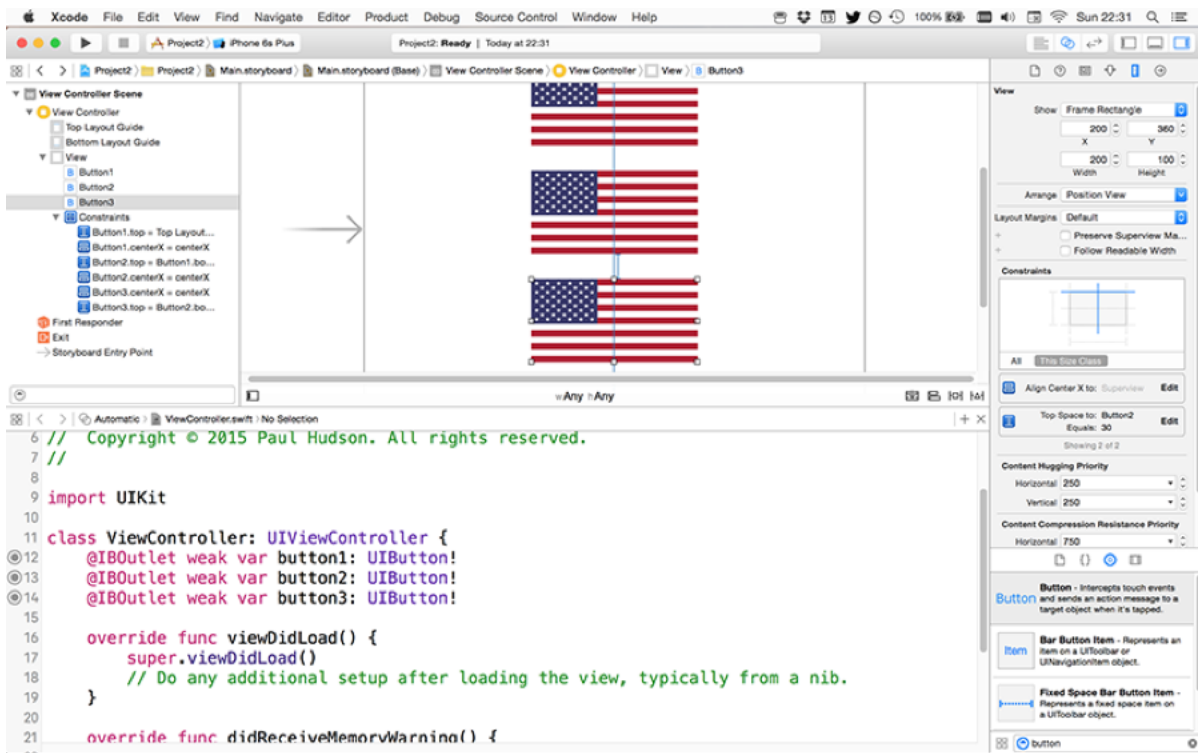
Create three outlets, one for each flag button.

We'll come back to it later on, but for now we're done with Interface Builder. Select ViewController.swift and go back to the standard editor (that is, press Cmd+return turn off the assistant editor) so we can get busy with some coding.

Making the basic game work

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

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

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

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

• **var a = 0** tells Swift that we want to put the number 0 into **score**. 0 is what's known as an integer (**Int** in Swift), which means it's a whole number. 556 is an integer. 100000001 is an integer. 3.14159 is not, because it's not a whole number. By "whole number" I mean a number that has nothing after its decimal point, but for the sake of clarity I should add that integers can be negative too.
• **var a = [String]()** This tells Swift that we want to put an array of strings into **a**. The syntax looks a little strange because it both declares what type we want, which is **[String]**, and also creates it: those **()** literally make this a method call that creates the **[String]** array.

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

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

• **var score = 0** This makes an **Int** (integer), so it holds whole numbers.
• **var score = 0.0** This makes a **Double**, which is one of several ways of holding decimal

numbers, e.g. 3.14159.

- **var score = "hello"** This makes a **String**, so it holds text.
- **var score = ""** Even though there's no text in the quote marks, this still makes a **String**.
- **var score = ["hello"]** This makes a **[String]** with one item, so it's an array where every item is a **String**.
- **var score = ["hello", "world"]** This makes a **[String]** with two items, so it's an array where every item is a String.

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

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

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

```
countries.append("estonia")

countries.append("france")

countries.append("germany")

countries.append("ireland")

countries.append("italy")

countries.append("monaco")

countries.append("nigeria")

countries.append("poland")

countries.append("russia")

countries.append("spain")

countries.append("uk")

countries.append("us")
```

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

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

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

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

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

```
askQuestion()
```

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

Add this new method underneath **viewDidLoad()**:

```
func askQuestion() {
  button1.setImage(UIImage(named: countries[0]), forState: .Normal)
```

```
button2.setImage(UIImage(named: countries[1]), forState: .Normal)

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

The first line is easy enough: we're declaring a new method called **askQuestion()**, and it takes no parameters. The next three use **UIImage(named:)** and read from an array by position, both of which we used in project 1, so that bit isn't new either. However, the rest of those lines is new, and shows off two things:
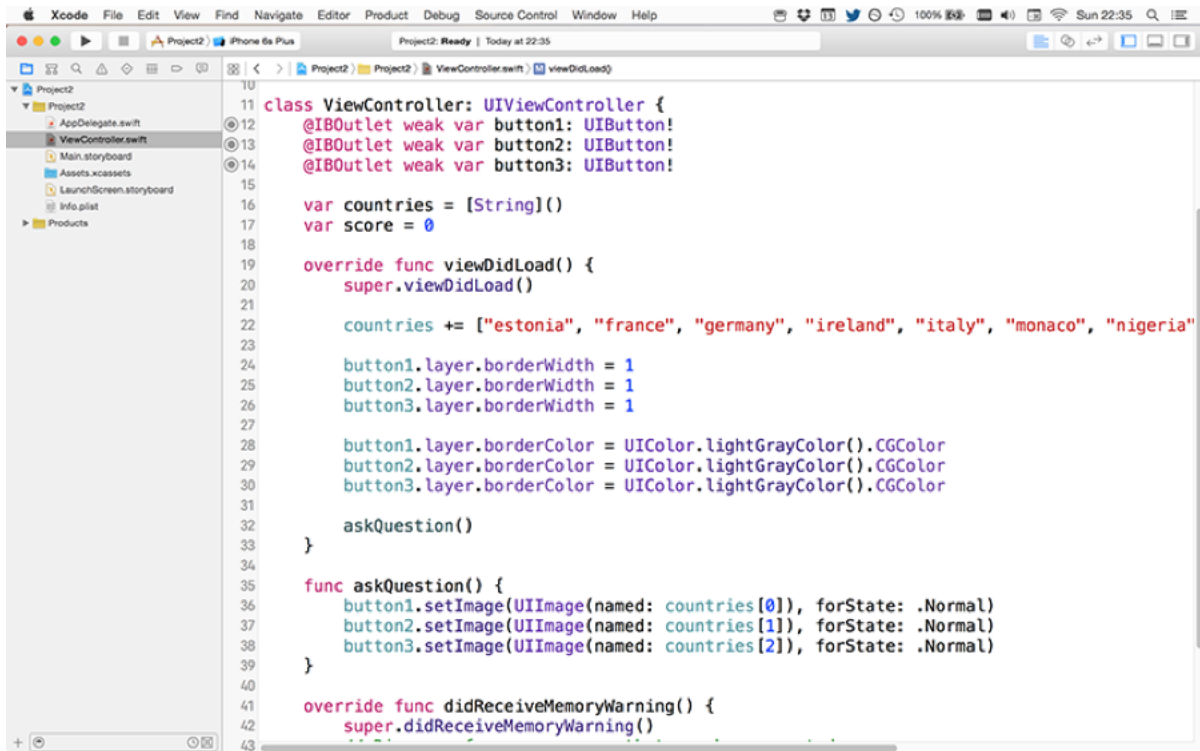
 • **button1.setImage()** assigns a **UIImage** to the button. We have the US flag in there right now, but this will change it when **askQuestion()** is called.
 • **forState: .Normal** The **setImage()** method takes a second parameter: which state of the button should be changed? We're specifying **.Normal**, which means "the standard state of the button."

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

Enums solve this problem by letting us use meaningful names for things. In place of 0 we can write **.Normal**, and in place of 1 we can write **.Disabled**, and so on. This makes code easier to write and easier to read, without having any performance impact. Perfect!

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

The actual data type **setImage()** expects is called **UIControlState**, and Swift is being clever: it knows to expect a **UIControlState** value in there, so when we write **.Normal** it understands that to mean "the **Normal** value of **UIControlState**." You could, if you wanted, write the line out in full as **UIControlState.Normal**, but that's not common.

This is how your code should look at this point.

At this point, the game is in a fit state to run, so press Cmd+R now to launch the iPhone Simulator and give it a try. You'll notice two problems: 1) we're showing the Estonian and French flags, both of which have white in them so it's hard to tell whether they are flags or just blocks of color, and 2) the "game" isn't much fun, because it's always the same three flags!

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

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

The Estonian flag has a white stripe at the bottom, and because our view controller has a white background that whole stripe is invisible. We can fix that by giving the layer of our

buttons a **borderWidth** of 1, which will draw a one point black line around them. Put these three lines in **viewDidLoad()** directly before it calls **askQuestion()**:

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

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

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

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

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

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

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

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

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

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

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

Guess which flag?

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

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

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

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

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

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

```
import GameplayKit
```

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

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

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

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

```
var correctAnswer = 0
```

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

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

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

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

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

The solution here is simple: uppercase the entire string. This is done using the **uppercaseString** property of any string, so all we need to do is read the string out from the countries array at the position of **correctAnswer**, then uppercase it. Add this to the end of the **askQuestion()** method, just after **correctAnswer** is set:

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

With that done, you can run the game and it's now almost playable: you'll get three different flags each time, and the flag the player needs to tap on will have its name shown at the top.
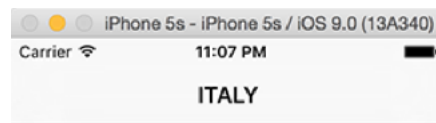
Your game so far: three different flags, with one correct answer shown at the top.

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

From outlets to actions

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

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

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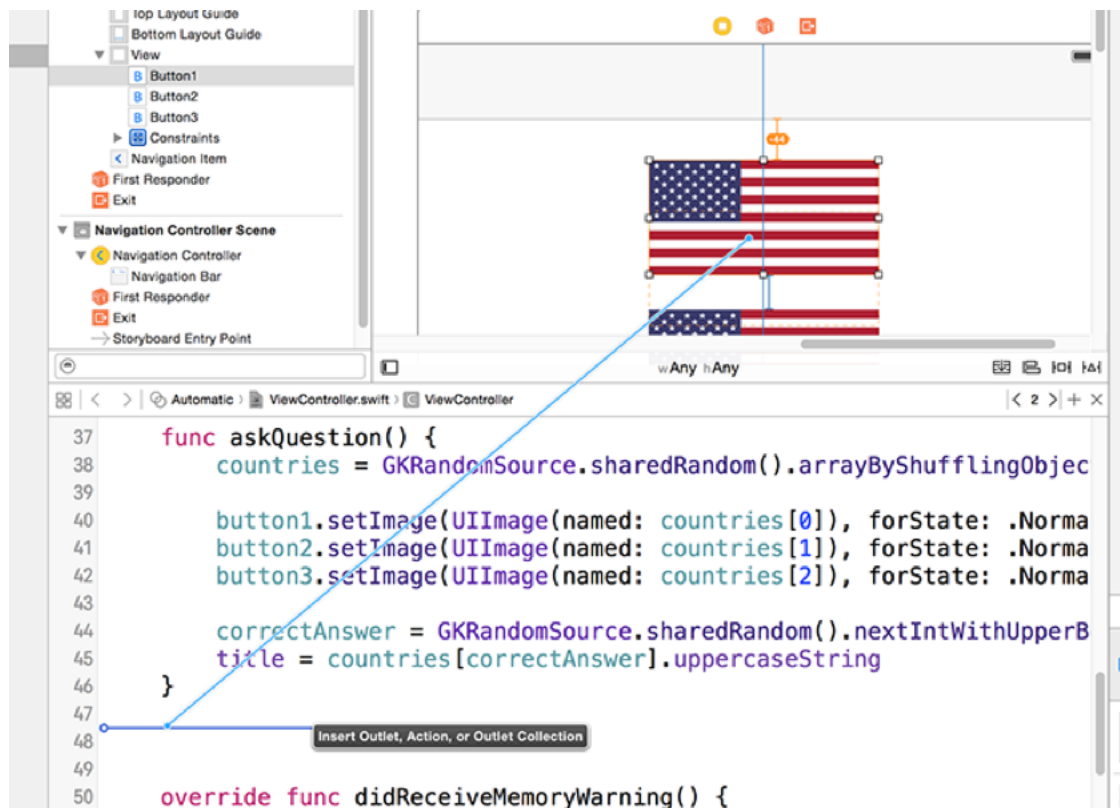
Creating an action in the Xcode assistant editor is very similar to creating an outlet.

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

you'll make problems for yourself!

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

Here's what Xcode will write for you:

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

}
```

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

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

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

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

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

But how do we know whether the correct button was tapped? Right now, all the buttons look the same, but behind the scenes all views have a special identifying number that we can set, called its Tag. This can be any number you want, so we're going to give our buttons the numbers 0, 1 and 2. This isn't a coincidence: our code is already set to put flags 0, 1 and 2 into those buttons, so if we give them the same tags we know exactly what flag was tapped.

Select the second flag (not the first one!), then look in the attributes inspector (Alt+Cmd+4) for the input box marked Tag. You'll need to scroll down, because **UIButton**s have lots of properties to work with! Once you find it (it's about two-thirds of the way down, just above the color and alpha properties), make sure it's set to 1.
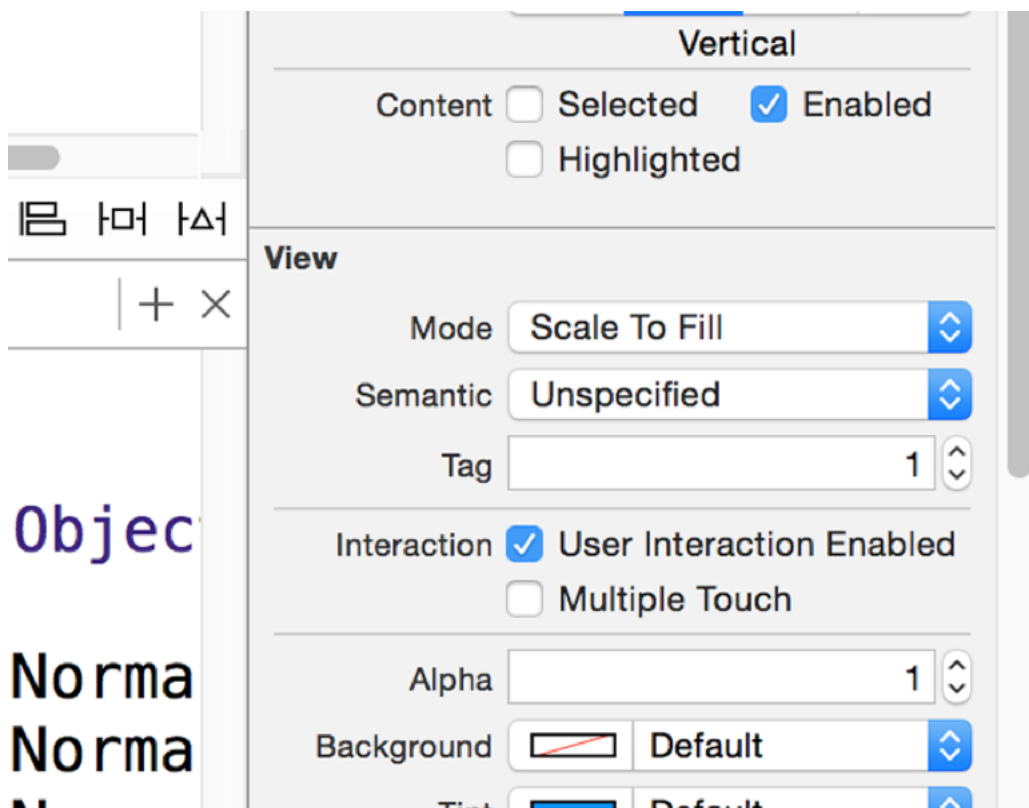
Setting a tag in Interface Builder is a quick and easy way to distinguish your views.

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

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

method.

This method needs to do three things:

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

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

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

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

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

```
var title: String


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

There are two new things here:

1. We're using the **==** operator. This is the equality operator, and checks if the value on the

left matches the value on the right. In this, it will be true if the tag of the button that was tapped equals the **correctAnswer** variable we saved in **askQuestion()**.

2. We have an **else** statement. When you write any **if** condition, you open a brace (curly bracket), write some code, then close the brace, and that code will be executed if the condition evaluates to true. But you can also give Swift some code that will be executed if the condition evaluates to false, and that's the "else" block. Here, we set one title if the answer was correct, and a different title if it was wrong.

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

The first thing to learn is called string interpolation. This is a Swift feature that lets you put variables and constants directly inside strings, and it will replace them with their current value when the code is executed. Right now, we have an integer variable called **score**, so we could put that into a string like this:

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

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

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

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

```
let ac = UIAlertController(title: title, message: "Your score is \
(score).", preferredStyle: .Alert)
ac.addAction(UIAlertAction(title: "Continue", style: .Default,
```

```
handler: askQuestion))
presentViewController(ac, animated: true, completion: nil)
```

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

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

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

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

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

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

The final line calls **presentViewController()**, which takes three parameters: a view controller to present, whether to animate the presentation, and another closure that should be executed when the presentation animation has finished. We send our **UIAlertController** for the first parameter, true for the second (animation is always nice!), and take a different shortcut for the closure: we give it **nil**. This means "do nothing," and you'll be using it a lot

with closures!

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

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

```
func askQuestion() {
```

…to this:

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

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

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

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

```
askQuestion(nil)
```

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

to this:

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

Now, go ahead and run your program in the simulator, because it's done!
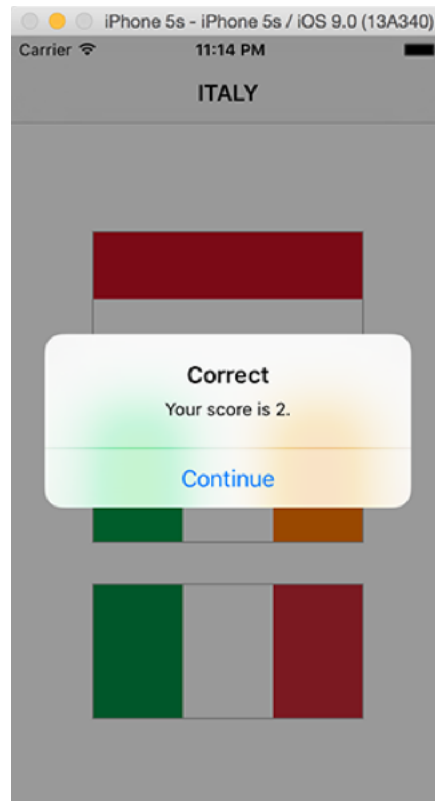
Tapping buttons now works - you see whether you were right or wrong.

Wrap up

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

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

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

Project 3
Social Media

Let users share to Facebook and Twitter by modifying project 1.

About technique projects

As you should know, this series follows the order app, game, technique. Project 1 was an app letting users browse images on their phone, project 2 was a game that lets players guess flags, so now it's time for the first technique project.

The goal with technique projects is to pick out one iOS technology and focus on it in depth. Some will be easy, some others not so much, but I promise to try to keep them as short as possible because I know you want to focus on making real things.

This first technique project is going to be really simple, because we're going to modify project 1 to do something it doesn't currently do: allow users to share images with their friends.

Activity View Controllers

Sharing things using iOS uses a standard, powerful component that other apps can plug into. As a result, it should be your first port of call when adding sharing to an app. This component is called **UIActivityViewController**: you tell it what kind of data you want to share, and it figures out how best to share it.

As we're working with images, **UIActivityViewController** will automatically gives us functionality to share by iMessage, by email and by Twitter and Facebook, as well as saving the image to the photo library, assigning it to contact, printing it out via AirPrint, and more. It even hooks into AirDrop and the iOS 8 extensions system so that other apps can read the image straight from us.

Best of all, it takes just a handful of lines of code to make it all work. But before we touch **UIActivityViewController**, we first need to give users a way to trigger sharing, and the way we're going to use is to add a bar button item.

Project 1, if you recall, used a **UINavigationController** to let users move between two screens. By default, a **UINavigationController** has a bar across the top, called a **UINavigationBar**, and as developers we can add buttons to this navigation bar that call our methods.

Let's create one of those buttons now. First, take a copy of your existing Project1 folder (the whole thing), and rename it to be Project3. Now launch it in Xcode, open the file DetailViewController.swift, and find the **viewDidLoad()** method. It's mostly empty right now – it just calls **super.viewDidLoad()** then **configureView()**. Directly beneath **configureView()**, I want you to add this code:

```
navigationItem.rightBarButtonItem =
UIBarButtonItem(barButtonSystemItem: .Action, target: self, action:
"shareTapped")
```

This is easily split into two parts: on the left we're assigning to the **rightBarButtonItem** of our view controller's **navigationItem**. This is navigation item is used by the navigation bar so that it can show relevant information. In this case, we're setting the right bar button item, which is a button that appears on the right of the navigation bar when this view controller is visible.

On the right we create a new instance of the **UIBarButtonItem** data type, setting it up with three parameters: a system item, a target, and an action. The system item we specify is **.Action**, but you can type **UIBarButtonSystemItem.** to have code completion tell you the many other options available. The **.Action** system item displays an arrow coming out of a box, signalling the user can do something when it's tapped.

The **target** and **action** parameters go hand in hand, because combined they tell the **UIBarButtonItem** what method should be called. The **action** parameter is saying "when you're tapped, call the **shareTapped()** method," and the target parameter tells the button that the method belongs to the current view controller – **self**.

If you don't like the look of the various system bar button items available, you can create with one with your own title or image instead. However, it's generally preferred to use the system items where possible because users already know what they do.

With the bar button created, it's time to create the **shareTapped()** method. Are you ready for this huge, complicated amount of code? Here goes! Put this just after the **viewWillDisappear()** method:

```
func shareTapped() {

  let vc = UIActivityViewController(activityItems:
[detailImageView.image!], applicationActivities: [])

  vc.popoverPresentationController?.barButtonItem =
navigationItem.rightBarButtonItem

  presentViewController(vc, animated: true, completion: nil)

}
```

That's it. With those three lines of code, **shareTapped()** can send photos via AirDrop, post to Twitter, and much more. You have to admit, iOS can be pretty amazing sometimes!

The third line is old; we already learned about **presentViewController()** in project 2. However, lines 1 and 2 are new, so let me explain what they do: line 1 creates a **UIActivityViewController**, which is the iOS method of sharing content with other apps and services, and line 2 tells iOS where the activity view controller should be anchored – where it should appear from.

On iPhone, activity view controllers automatically take up the full screen, but on iPad they appear as a popover that allows the user to see what they were working on below. This line of code tells iOS to anchor the activity view controller to the right bar button item (our share button), but this only has an effect on iPad – on iPhone it's ignored.

Let's focus on how activity view controllers are created. As you can see in the code, you pass in two items: an array of items you want to share, and an array of any of your own app's services you want to make sure are in the list. We're passing an empty array into the second parameter, because our app doesn't have any services to offer. But if you were to extend this app to have something like "Other pictures like this", for example, then you would include that functionality here.

So, the real focus is on the first parameter: we're passing in **[detailImageView.image!]**. If you recall, the image was being displayed in a **UIImageView** called **detailImageView**, and **UIImageView** has an optional property called **image**, which holds a **UIImage**. But it's optional, so there may be an image or there may not. And **UIActivityViewController** doesn't want maybe or maybe not, it wants facts.

Fortunately, we know for a fact that our image view has an image, because we set it! In fact, that's the whole point of this view controller. So we use **detailImageView.image!** with that exclamation mark on the end to force unwrap the optional. That then gets put into an array by itself, and send to **UIActivityViewController**.

And… that's it. No, really. We're done: your app now supports sharing. If you want to try pushing your skills further, try modifying **shareTapped()** so that it shows a message if no image was selected. This is only possible on iPad, but it's worth catching.
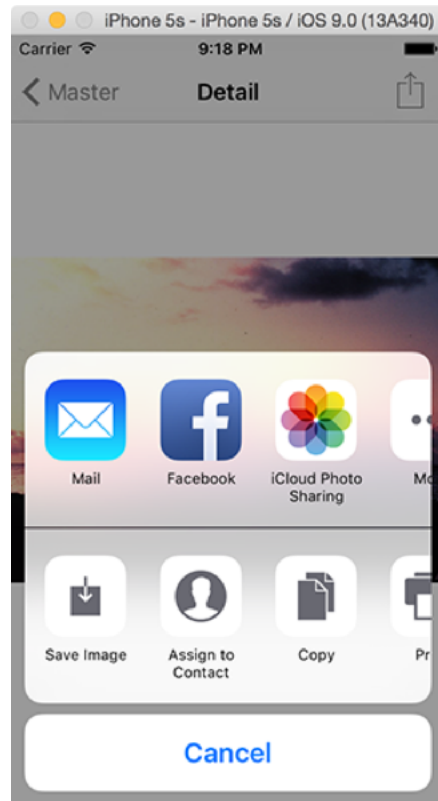
UIActivityViewController lets your users share, print, save and more – all in just two lines of code!

Twitter and Facebook

OK, so I would feel guilty if I didn't spend a *little* more time with you showing you other ways to share things, in particular there's built-in support for Facebook and Twitter sharing in iOS and both are quite easy to use.

iOS includes a framework called "Social", which is designed to post to social networks like Facebook and Twitter. We can use both of these in our app to share the image the user is looking at, and it has the added benefit that the user is immediately prompted to enter their tweet / Facebook post – there's no initial view controller there asking them how they want to share.

Happily, using the Social framework to post to social media is also straightforward, and has the advantage of simple method calls that are self-describing. In fact, I'm just going to go ahead and show you the code, and see what you think:

```
let vc = SLComposeViewController(forServiceType:
SLServiceTypeFacebook)

vc.setInitialText("Look at this great picture!")

vc.addImage(detailImageView.image!)

vc.addURL(NSURL(string: "http://www.photolib.noaa.gov/nssl"))

presentViewController(vc, animated: true, completion: nil)
```

Apart from the **SLComposeViewController** component, which as you can see is created with the Facebook service type, the only other new thing in there is **NSURL**. This is a new data type, and one that might seem a little redundant at first: it stores and processes URLs like www.yoursite.com.

Now, clearly to you and me a URL is a text string, so it seems strange to have a dedicated class when a plain old string would do. However, iOS uses URLs for more things than just websites. For example, you can get a file URL to a local file, or you can get a URL to a document securely stored in iCloud. And even if it were just about website URLs, even then there's things like "is it HTTP or HTTPS?" and "is there a username and password in the URL?"

We're going to use **NSURL** again in the next project, but right now its use is quite simple:

we're attaching the URL to the National Severe Storm Laboratory so that people can browse there for more photos. The **NSURL(string:)** method converts the string "http://www.photolib.noaa.gov/nssl" into a full **NSURL** instance, which can then be passed to **addURL()**.

Now that code is in place, you will find it doesn't actually compile. This is because the Social framework is separate from the default iOS frameworks, so it isn't included as standard. If you want to import the Social framework (and you will want to if you ever want your code to compile again!), just scroll to the top of the view controller and look for the line **import UIKit**. Now put this line directly above:

```
import Social
```

UIKit is the name for Apple's iOS development framework, so this includes both UIKit and the Social framework.

Finally, if you want to use Twitter instead, just specify **SLServiceTypeTwitter** for the service type; the rest of the code stays the same.
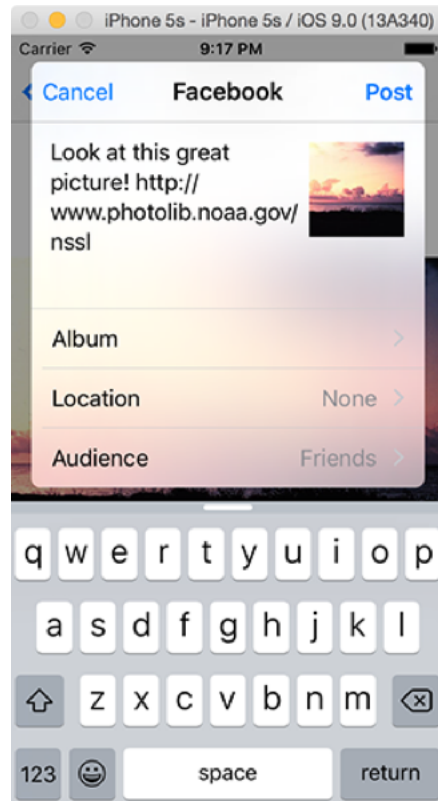
If you want to share to a particular service, such as Facebook and Twitter, you should use the Social framework.

Wrap up

This was a deliberately short technique project taking an existing app and making it better. I hope you didn't get too bored, and hope even more that some of the new material sunk in because we covered **UIBarButtonItem**, **UIActivityViewController**, the Social framework, and **NSURL**.

I hope you can see how trivial it is to add social media to your apps, and it can make a huge difference to helping spread the word about your work once your apps are on the App Store. I hope this project has also shown you how easy it is to go back to previous projects and improve them with only a little extra effort.

Project 4
Easy Browser

Embed Web Kit and learn about delegation, KVO, classes and UIToolbar.

Setting up

In this project you're going to build on your new knowledge of **UIBarButtonItem**, **UIAlertController** and **NSURL** by producing a simple web browser app. Yes, I realise this is another easy project, but learning is as much about tackling new challenges as going over what you've already learned.

To sweeten the deal, I'm going to use this opportunity to teach you lots of new things: **WKWebView** (Apple's extraordinary web widget), **UIToolbar** (a toolbar component that holds **UIBarButtonItem**s), **UIProgessView**, delegation, classes and structs, key-value observing, and how to create your views in code. Plus, this is the last easy app project, so enjoy it while it lasts!

To get started, create a new Xcode project using the Single View Application template, and call it Project4. Choose iPhone for the device, and make sure Swift is selected for the language, then save the project on your desktop. Open up Main.storyboard, select the view controller, and choose Editor > Embed In > Navigation Controller – that's our storyboard finished.

Creating a simple browser

In projects 1 and 2, we used Interface Builder for a lot of layout work, but here our layout will be so simple we can do the entire thing in code. You see, before we were adding buttons and images to our view, but in this project the web view is going to take up all the space so it might as well *be* the view controller's main view.

So far, we've been using the **viewDidLoad()** method to configure our view once its layout has loaded. This time we need to override the actual loading of the view – we don't want that empty thing on the storyboard, we want our own code. It will still be placed inside the navigation controller, but the rest is up to us.

Open ViewController.swift for editing, and *before* **viewDidLoad()** put this:

```
override func loadView() {

  webView = WKWebView()

  webView.navigationDelegate = self

  view = webView

}
```

It isn't at all necessary to put **loadView()** before **viewDidLoad()** – you could put it anywhere between **class ViewController: UIViewController {** down to the last closing brace in the file. But I do encourage you to structure your methods in an organised way, and because **loadView()** gets called before **viewDidLoad()** it makes sense to position the code above it too.

Anyway, there are only three things we care about, because by now you should understand why we need to use the **override** keyword. (Hint: it's because there's a default implementation, which is to load the layout from the storyboard!) First, we create a new instance of Apple's **WKWebView** web browser component and assign it to a variable called **webView**. Third, we make our view (the root view of the view controller) that web view.

Yes, I missed out the second line, and that's because it introduces new concept: delegation. Delegation is what's called a *programming pattern* – a way of writing code – and it's used extensively in iOS. And for good reason: it's easy to understand, easy to use, and extremely

flexible.

A *delegate* is one thing acting in place of another, effectively answering questions and responding to events on its behalf. In our example, we're using WKWebView: Apple's powerful, flexible and efficient web renderer. But as smart as **WKWebView** is, it doesn't know (or care!) how our application wants to behave, because that's our custom code.

The delegation solution is brilliant: we can tell **WKWebView** that we want to be told when something interesting happens. In our code, we're setting the web view's **navigationDelegate** property to **self**, which means "when any web page navigation happens, please tell me."

When you do this, two things happen:

1. You must conform to the protocol. This is a fancy way of saying, "if you're telling me you can handle being my delegate, here are the methods you need to implement." In the case of **navigationDelegate**, all these methods are optional, meaning that we don't *need* to implement any methods.
2. Any methods you do implement will now be given control over the **WKWebView**'s behavior. Any you don't implement will use the default behavior of **WKWebView**.

Before we get any further, you may have noticed that your code doesn't actually compile. There are three reasons, but all three take just seconds to fix.

Reason #1: Swift doesn't know what **WKWebView** is. As you can see, it doesn't start with "UI" so **WKWebView** is not part of UIKit. As a result, we need to import a new framework before we can use it. The "WK" in **WKWebView** stands for WebKit, so go to the top of your file and modify it to this:

```
import UIKit
import WebKit
```

Reason #2: We haven't declared a **webView** property, and yet we're assigning to it. To fix this, go to the line before our **loadView()** method and add this:

```
var webView: WKWebView!
```

That declares an implicitly unwrapped optional **WKWebView** instance called **webView**. I'll explain one more time just so we're clear: that ! at the end of **WKWebView** is needed because the property starts out as **nil** before being set later on.

Reason #3: When you set any delegate, you need to conform to the protocol. Yes, all the **navigationDelegate** protocol methods are optional, but we Swift doesn't know that yet. All it knows is that we're promising we're a suitable delegate for the web view, and yet haven't implemented the protocol.

The fix for this is simple, but I'm going to hijack it to introduce something else the same time, because this is an opportune moment. First, the fix: find this line:

```
class ViewController: UIViewController {
```

…and change it to this:

```
class ViewController: UIViewController, WKNavigationDelegate {
```

That's the fix. But what I want to discuss is the **class** bit, because I've been using words like "data type", "component" and "instance" so far, without really being clear – and I promise you there are developers out there that are absolutely seething as a result. Hello, haters!

There are two types of complex data types in Swift: structures ("structs") and classes. They are extremely similar in Swift, and really there are only two differences likely to matter to you at this stage, or indeed any stage over the next six months or so.

The first difference is that one class can inherit from another. We already talked about this in project 1, where our view controller inherited from **UIViewController**. This class inheritance means you get to build on all the amazing power when you inherit from **UIViewController**, and add your own customisations on top.

The second difference is that when you pass a struct into a method, a *copy* gets passed in. This means any changes you make in the method won't affect the struct outside of the method. On the other hand, when you pass an instance of a class into a method, it's passed *by reference*, meaning that the object inside the method is the same one outside the method; any changes you make will stay.

In terms of which is which: **Int**, **Double**, **Float**, **String** and **Array** are all structs, **UIViewController** and any **UIView** are all classes. In practice, this means that whenever you pass an array into a method, it gets copied. That might sound grossly inefficient, particularly if the array contains a huge amount of data, but don't fret about it: Swift will avoid any performance penalty as best it can using a technique called copy on write.

Back to our code: all this is important, because I want you to understand exactly what the line of code does. Here it is again:

```
class ViewController: UIViewController, WKNavigationDelegate {
```

As you can see, the line kicks off with "class", showing that we're declaring a new class here. The line ends with an opening brace, and everything from that opening brace to the closing brace at the end of the file form part of our class. The next part, **ViewController**, is the name of our class. Not a great name in a big project, but for a Single View Application template project it's fine.

The interesting stuff comes next: there's a colon, followed by **UIViewController**, then a comma and **WKNavigationDelegate**. If you're feeling fancy, this part is called a type inheritance clause, but what it really means is that this is the definition of what the new **ViewController** class is made of: it inherits from **UIViewController** (the first item in the list), and implements the **WKNavigationDelegate** protocol.

The order here really is important: the parent class (superclass) comes first, then all protocols implemented come next, all separated by commas. We're saying that we conform to only one protocol here (**WKNavigationDelegate**) but you can specify as many as you need to.

So, the complete meaning of this line is "create a new subclass of **UIViewController** called **ViewController**, and tell the compiler that we promise we're safe to use as a **WKNavigationDelegate**."

This program is almost doing something useful, so before you run it let's add three more lines. Please place these in the **viewDidLoad()** method, just after the **super** call:

```
let url = NSURL(string: "https://www.hackingwithswift.com")!
webView.loadRequest(NSURLRequest(URL: url))
webView.allowsBackForwardNavigationGestures = true
```

The first line creates a new **NSURL**, as you saw in the previous project. I'm using hackingwithswift.com as an example website, but please change it to something you like. **Warning: you need to ensure you use https:// for your websites, because iOS 9 does not like apps sending or receiving data insecurely.** If this is something you want to override, **click here to read about App Transport Security in iOS 9**.

The second line does two things: it creates a new **NSURLRequest** object from that NSURL, and gives it to our web view to load.

Now, this probably seems like pointless obfuscation from Apple, but **WKWebViews** don't load websites from strings like www.hackingwithswift.com, or even from an **NSURL** made out of those strings. You need to turn the string into an **NSURL**, then put the **NSURL** into an **NSURLRequest**, and **WKWebView** will load *that*. Fortunately it's not hard to do!

Warning: Your URL must be complete, and valid, in order for this process to work. That means including the **https://** part.

The third line enables a property on the web view that allows users to swipe from the left or right edge to move backward or forward in their web browsing. This is a feature from the Safari browser that many users rely on, so it's nice to keep it around.

Press Cmd+R to run your app now, and you should be able to view your website. Step one done!
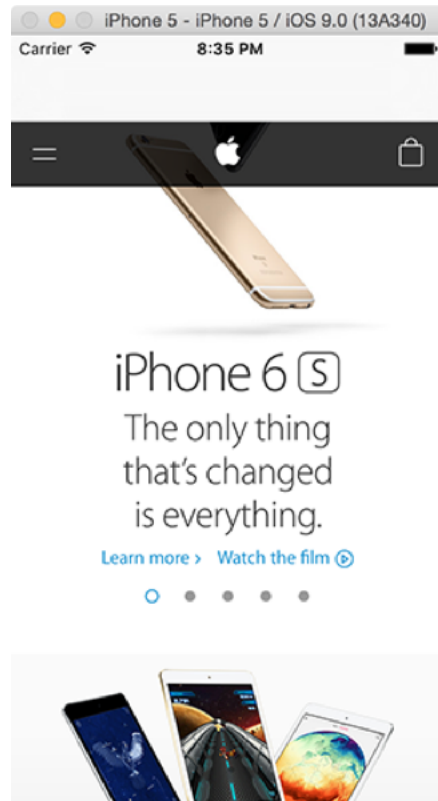
Your Action.js file needs to be in the Copy Bundle Resources build phase of your extension.

Choosing a website

We're going to lock this app down so that it opens websites selected by the user. The first step to doing this is to give the user the option to choose from one of our selected websites, and that means adding a button to the navigation bar.

Somewhere in **viewDidLoad()** (but always after it has called **super.viewDidLoad()**), add this:

```
navigationItem.rightBarButtonItem = UIBarButtonItem(title: "Open",
style: .Plain, target: self, action: "openTapped")
```

We did exactly this in the previous project, except here we're using a custom title for our bar button rather than a system icon. It called the **openTapped()** method, which doesn't exist, when the button is tapped, so let's add that now. Put this method below **viewDidLoad()**:

```
func openTapped() {

  let ac = UIAlertController(title: "Open page…", message: nil,
preferredStyle: .ActionSheet)

  ac.addAction(UIAlertAction(title: "apple.com", style: .Default,
handler: openPage))

  ac.addAction(UIAlertAction(title: "hackingwithswift.com",
style: .Default, handler: openPage))

  ac.addAction(UIAlertAction(title: "Cancel", style: .Cancel,
handler: nil))

  presentViewController(ac, animated: true, completion: nil)

}
```

Warning: if you did *not* set your app to be targeted for iPhone at the beginning of this chapter, the above code will not work correctly. Yes, I know I told you to set iPhone, but a lot of people skip over things in their rush to get ahead. If you chose iPad or Universal, you will need to add **ac.popoverPresentationController?.barButtonItem = self.navigationItem.rightBarButtonItem** to the **openTapped()** method before presenting the alert controller.

We used the **UIAlertController** class in project 2, but here it's slightly different for three reason:

1. We're using **nil** for the message, because this alert doesn't need one.
2. We're using the **preferredStyle** of **.ActionSheet** because we're prompting the user for more information.
3. We're adding a dedicated Cancel button using style **.Cancel**. It has a handler of **nil** which will just hide the alert controller.

Both our website buttons point to the **openPage()** method, which, again, doesn't exist yet. This is going to be very similar to how we loaded the web page before, but now you will at least see why the handler method of **UIAlertAction** takes a parameter telling you which action was selected!

Add this method directly beneath the **openTapped()** method you just made:

```
func openPage(action: UIAlertAction!) {

  let url = NSURL(string: "https://" + action.title!)!

  webView.loadRequest(NSURLRequest(URL: url))

}
```

This method takes one parameter, which is the **UIAlertAction** object that was selected by the user. Obviously it won't be called if Cancel was tapped, because that had a **nil** handler rather than **openPage**.

What the method does is use the **title** property of the action (apple.com, hackingwithswift.com), put "https://" in front of it to satisfy App Transport Security, then construct an **NSURL** out of it. It then wraps that inside an **NSURLRequest**, and gives it to the web view to load. All you need to do is make sure the websites in the **UIAlertController** are correct, and this method will load anything.

You can go ahead and test the app now, but there's one small change we can add to make the whole experience more pleasant: setting the title in the navigation bar. Now, we are the web view's navigation delegate, which means we will be told when any interesting navigation happens, such as when the web page has finished loading. We're going to use this to set the navigation bar title.

As soon as we told Swift that our **ViewController** class conformed to the **WKNavigationDelegate** protocol, Xcode updated its code completion system to support all the **WKNavigationDelegate** methods that can be called. As a result, if you go below the **openPage()** method and start typing "web" you'll see a list of all the **WKNavigationDelegate** methods we can use.

Scroll through the list of options until you see **didFinishNavigation** and press return to have Xcode fill in the method for you. Now modify it to this:

```
func webView(webView: WKWebView, didFinishNavigation navigation:
WKNavigation!) {

  title = webView.title

}
```

All this method does it update our view controller's **title** property to be the title of the web view, which will automatically be set to the page title of the web page that was most recently loaded.

Press Cmd+R now to run the app, and you'll see things are starting to come together: your initial web page will load, and when the load finishes you'll see its page title in the navigation bar.
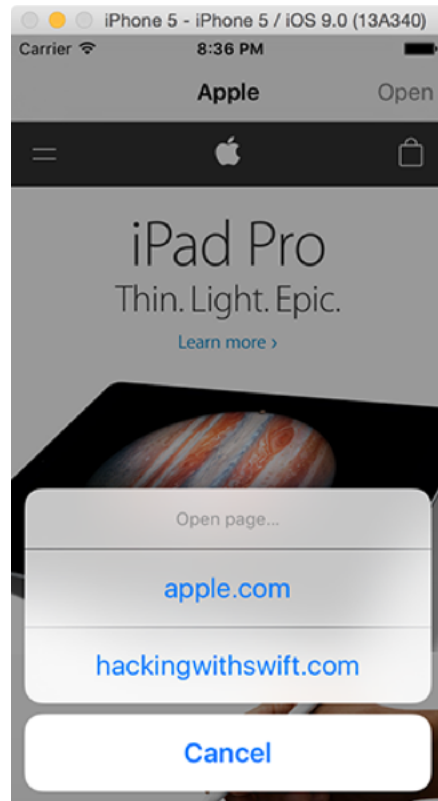
Using a UIAlertController is an easy way to let users choose which website to visit.

Monitoring page loads

Now is a great time to meet two new **UIView** subclasses: **UIToolbar** and **UIProgressView**. **UIToolbar** holds and shows a collection of **UIBarButtonItem** objects that the user can tap on. We already saw how each view controller has a **rightBarButton** item, so a **UIToolbar** is like having a whole bar of these items. **UIProgressView** is a colored bar that shows how far a task is through its work, sometimes called a "progress bar."

The way we're going to use **UIToolbar** is quite simple: all view controllers automatically come with a **toolbarItems** array that automatically gets read in when the view controller is active inside a **UINavigationController**.

This is very similar to the way **rightBarButtonItem** is shown only when the view controller is active. All we need to do is set the array, then tell our navigation controller to show its toolbar, and it will do the rest of the work for us.

We're going to create two **UIBarButtonItems** at first, although one is special because it's a flexible space. This is a unique **UIBarButtonItem** type that acts like a spring, pushing other buttons to one side until all the space is used.

In **viewDidLoad()**, put this new code directly below where we set the **rightBarButtonItem**:

```
let spacer = UIBarButtonItem(barButtonSystemItem: .FlexibleSpace,
target: nil, action: nil)

let refresh = UIBarButtonItem(barButtonSystemItem: .Refresh, target:
self, action: "refreshTapped")


toolbarItems = [spacer, refresh]

navigationController?.toolbarHidden = false
```

The first line is new, or at least part of it is: we're creating a new bar button item using the special system item type **.FlexibleSpace**, which creates a flexible space. It doesn't need a target or action because it can't be tapped. The second line you've seen before, although now we're calling the **refreshTapped()** method.

The last two lines are new: the first puts an array containing the flexible space and the refresh

button, then sets it to be our view controller's toolbarItems array. The second sets the navigation controller's **toolbarHidden** property to be false, show the toolbar will be shown – and its items will be loaded from our current view.

That code will compile and run, and you'll see the refresh button neatly aligned to the right – that's the effect of the flexible space automatically taking up as much room as it can on the left. But if you tap it you'll see an immediate crash.

Back in Xcode, you might even notice a long crash log being printed near the bottom of the screen – just scroll to the top and read from there, because most of the rest of the text isn't helpful right now. You should see something like this:

```
[Project4.ViewController refreshTapped]: unrecognized selector sent
to instance
```

A "selector" is a fancy way of saying "method." Well, that's not strictly true – it's a bit cleverer than that. But it's not something that Swift has much use for, so as far as you're concerned it means method. So this error message says, "I tried to call **refreshTapped()**, but couldn't find it" – and it's right! Let's add this new method just before the closing brace of our class:

```
func refreshTapped() {

    webView.reload()

}
```

It's not complicated, as you can see: **WKWebView** has a **reload()** method that we can call, and it just reloads the web page. In fact, it's so simple to do that we could easily remove this method entirely and have our refresh **UIBarButtonItem** call the web view directly! Change the definition of the **refresh** button to be this:

```
let refresh = UIBarButtonItem(barButtonSystemItem: .Refresh, target:
webView, action: "reload")
```

Having **webView** for the target and "reload" for the action means that when the bar button is tapped **webView.reload()** is called. Easy!

The next step is going to be to add a **UIProgressView** to our toolbar, which will show how far the page is through loading. However, this requires two new pieces of information:

 • You can't just add random **UIView** subclasses to a **UIToolbar**, or to the **rightBarButtonItem** property. Instead, you need to wrap them in a special **UIBarButtonItem**, and use that instead.
 • Although **WKWebView** tells us how much of the page has loaded using its **estimatedProgress** property, the **WKNavigationDelegate** system doesn't tell us when this value has changed. So, we're going to ask iOS to tell us using a powerful technique called key-value observing, or KVO.

First, let's create the progress view and place it inside the bar button item. Begin by declaring the property at the top of the **ViewController** class next to the existing **WKWebView** property:

```
var progressView: UIProgressView!
```

Now place this code directly before the **let spacer =** line in viewDidLoad():

```
progressView = UIProgressView(progressViewStyle: .Default)
progressView.sizeToFit()
let progressButton = UIBarButtonItem(customView: progressView)
```

All three of those lines are new, so let's go over them:

1. The first line creates a new **UIProgressView** instance, giving it the default style. There is an alternative style called **.Bar**, which doesn't draw an unfilled line to show the extent of the progress view, but the default style looks best here.
2. The second line tells the progress view set its layout size so that it fits its contents fully.
3. The last line creates a new **UIBarButtonItem** using the **customView** parameter, which is where we wrap up our **UIProgressView** in a **UIBarButtonItem** so that it can go into our

toolbar.

With the new **progressButton** item created, we can put it into our toolbar items anywhere we want it. The existing spacer will automatically make itself smaller to give space to the progress button, so I'm going to modify my **toolbarItems** array to this:

```
toolbarItems = [progressButton, spacer, refresh]
```

That is, progress view first, then a space in the center, then the refresh button on the right.

If you run the app now, you'll just see a thin gray line for our progress view – that's because it's default value is 0, so there's nothing colored in. Ideally we want to set this to match our webView's **estimatedProgress** value, which is a number from 0 to 1, but WKNavigationDelegate doesn't tell us when this value has changed.

Apple's solution to this is huge. Apple's solution is powerful. And, best of all, Apple's solution is almost everywhere in its toolkits, so once you learn how it works you can apply it elsewhere. It's called key-value observing (KVO), and it effectively lets you say, "please tell me when the property X of object Y gets changed by anyone at any time."

We're going to use KVO to watch the **estimatedProgress** property, and I hope you'll agree that it's extremely easy. First, we add ourselves as an observer of the property on the web view by adding this to **viewDidLoad()**:

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

The **addObserver()** method takes four parameters: who the observer is (we're the observer, so we use **self**), what property we want to observe (we want the **estimatedProgress** property), which value we want (we want the value that was just set, so we want the new one), and a context value.

forKeyPath and **context** bear a little more explanation. **forKeyPath** isn't named **forProperty** because it's not just about entering a property name. You can actually specify a path: one property inside another, inside another, and so on. More advanced key paths can even add

functionality, such as averaging all elements in an array!

context is easier: if you provide a unique value, that same context value gets sent back to you when you get your notification that the value has changed. This allows you to check the context to make sure it was your observer that was called. There are some corner cases where specifying (and checking) a context is required to avoid bugs, but you won't reach them during any of this series.

Warning: in more complex applications, all calls to **addObserver()** should be matched with a call to **removeObserver()** when you're finished observing – for example, when you're done with the view controller.

Once you have registered as an observer using KVO, you *must* implement a method called **observeValueForKeyPath()**. This tells you when an observed value has changed, so add this method now:

```
override func observeValueForKeyPath(keyPath: String?, ofObject
object: AnyObject?, change: [String : AnyObject]?, context:
UnsafeMutablePointer<Void>) {

  if keyPath == "estimatedProgress" {

    progressView.progress = Float(webView.estimatedProgress)

  }

}
```

As you can see it's telling us which key path was changed, and it also sends us back the context we registered earlier so you can check whether this callback is for you or not.

In this project, all we care about is whether the **keyPath** parameter is set to **estimatedProgress** – that is, if the **estimatedProgress** value of the web view has changed. And if it has, we set the **progress** property of our progress view to the new **estimatedProgress** value.

Minor note: **estimatedProgress** is a **Double**, which as you should remember is one way of representing decimal numbers like 0.5 or 0.55555. Unhelpfully, **UIProgressView**'s **progress** property is a **Float**, which is another (lower-precision) way of representing decimal numbers. Swift doesn't let you put a **Double** into a **Float**, so we need to create a new **Float** from the

Double.

If you run your project now, you'll see the progress view fills up with blue as the page loads.

Refactoring for the win

Our app has a fatal flaw, and there are two ways to fix it: double up on code, or refactor. Cunningly, the first option is nearly always the easiest, and yet counter-intuitively also the hardest.

The flaw is this: we let users select from a list of websites, but once they are on that website they can get pretty much anywhere else they want just by following links. Wouldn't it be nice if we could check every link that was followed so that we can make sure it's on our safe list?

One solution – doubling up on code – would have us writing the list of accessible websites twice: once in the **UIAlertController** and once when we're checking the link. This is extremely easy to write, but it can be a trap: you now have two lists of websites, and it's down to you to keep them both up to date. And if you find a bug in your duplicated code, will you remember to fix it in the other place too?

The second solution is called refactoring, and it's effectively a rewrite of the code. The end result should do the same thing, though. The purpose of the rewrite is to make it more efficient, make it easier to read, reduce its complexity, and to make it more flexible. This last use is what we'll be shooting for: we want to refactor our code so there's a shared array of allowed websites.

Up where we declared our two properties **webView** and **progressView**, add this:

```
var websites = ["apple.com", "hackingwithswift.com"]
```

That's an array containing the websites we want the user to be able to visit.

With that array, we can modify the web view's initial web page so that it's not hard-coded. In **viewDidLoad()**, change the initial web page to this:

```
let url = NSURL(string: "https://" + websites[0])!
webView.loadRequest(NSURLRequest(URL: url))
```

So far, so easy. The next change is to make our **UIAlertController** use the websites for its list of **UIAlertAction**s. Go down to the **openTapped()** method and replace these two lines:

```
ac.addAction(UIAlertAction(title: "apple.com", style: .Default,
handler: openPage))

ac.addAction(UIAlertAction(title: "hackingwithswift.com",
style: .Default, handler: openPage))
```

…with this loop:

```
for website in websites {
  ac.addAction(UIAlertAction(title: website, style: .Default,
handler: openPage))
}
```

That will add one **UIAlertAction** object for each item in our array. Again, not too complicated.

The final change is something new, and it belongs to the **WKNavigationDelegate** protocol. If you find space for a new method and start typing "web" you'll see the list of **WKWebView**-related code completion options. Look for the one called **decidePolicyForNavigationAction** and let Xcode fill in the method for you.

This delegate callback allows us to decide whether we want to allow navigation to happen or not every time something happens. We can check which part of the page started the navigation, we can see whether it was triggered by a link being clicked or a form being submitted, or, in our case, we can check the URL to see whether we like it.

Now that we've implemented this method, it expects a response: should we load the page or should we not? When this method is called, you get passed in a parameter called **decisionHandler**. This actually holds a function, which means if you "call" the parameter, you're actually calling the function.

If your brain has just turned to soup, let me try to clarify. In project 2 I talked about closures: chunks of code that you can pass into a function like a variable and have executed at a later

date. This **decisionHandler** is also a closure, except it's the other way around – rather than giving someone else a chunk of code to execute, you're being given it and are required to execute it.

And make no mistake: you *are required* to do something with that **decisionHandler** closure. That might make sound an extremely complicated way of returning a value from a method, and that's true – but it's also underestimating the power a little! Having this **decisionHandler** variable/function means you can show some user interface to the user "Do you really want to load this page?" and call the closure when you have an answer.

So, we need to evaluate the URL to see whether it's in our safe list, then call the **decisionHandler** with a negative or positive answer. Here's the code for the method:

```
func webView(webView: WKWebView, decidePolicyForNavigationAction
navigationAction: WKNavigationAction, decisionHandler:
(WKNavigationActionPolicy) -> Void) {

  let url = navigationAction.request.URL


  if let host = url!.host {

    for website in websites {

      if host.rangeOfString(website) != nil {

        decisionHandler(.Allow)

        return

      }

    }

  }


  decisionHandler(.Cancel)
}
```

There are some easy bits, but they are outweighed by the hard bits so let's go through every line in detail to make sure:

1. First, we set the constant **url** to be equal to the **NSURL** of the navigation. This is just to make the code clearer.

2. Second, we use if/let syntax to unwrap the value of the optional **url.host**. Remember I said that **NSURL** does a lot of work for you in parsing URLs properly? Well, here's a good example: this line says, "if there is a host for this URL, pull it out" – and by "host" it means "website domain" like apple.com. NB: we need to unwrap this carefully because not all URLs have hosts.

3. Third, we loop through all sites in our safe list, placing the name of the site in the **website** variable.

4. Fourth, we use the **rangeOfString()** String method to see whether each safe website exists somewhere in the host name.

5. Fifth, if the website was found (if **rangeOfString()** is not **nil**) then we call the decision handler with a positive response: allow loading.

6. Sixth, if the website was found, after calling the **decisionHandler** we use the **return** statement. This means "exit the method now."

7. Last, if there is no host set, or if we've gone through all the loop and found nothing, we call the decision handler with a negative response: cancel loading.

The **rangeOfString()** method can take quite a few parameters, however all but the first are optional so the above usage is fine. To use it, call **rangeOfString()** on one string, giving it another string as a parameter, and it will tell you where it was found, or **nil** if it wasn't found at all.

You've already met the **hasPrefix()** method in project 1, but **hasPrefix()** isn't suitable here because our safe site name could appear anywhere in the URL. For example, slashdot.org redirects to m.slashdot.org for mobile devices, and **hasPrefix()** would fail that test.

The **return** statement is new, but it's one you'll be using a lot from now on. It exits the method immediately, executing no further code. If you said your method returns a value, you'll use the **return** statement to return that value.

Your project is complete: press Cmd+R to run the finished app, and enjoy!

Wrap up

Another project done, another huge pile of things learned. You should be starting to get into the swing of things by now, but don't let yourself become immune to your success. In this tutorial alone you've learned about **loadView()**, **WKWebView**, delegation, classes and structs, **NSURLRequest**, **UIToolbar**, **UIProgressView**, KVO and more, so you should be proud of your fantastic accomplishments!

There is a lot of scope for improvement with this project, so where you start is down to you. I would suggest at the very least that you investigate changing the initial view controller to a table view like in project 1, where users can go choose their website from a list rather than just having the first in the array loaded up front.

Once you have completed project 5, you might like to return here to add in the option to load the list of websites from a file, rather than having them hard-coded in an array.

Project 5

Word Scramble

Create an anagram game while learning about closures and booleans.

Setting up

Projects 1 to 4 were all fairly easy, because my goal was to teach you as much about Swift without scaring you away, while also trying to make something useful. But now that you're hopefully starting to become familiar with the core tools of iOS development, it's time to change up a gear and tackle something a bit meatier.

In this project you're going to learn how to make a word game that deals with anagrams, but as per usual I'll be hijacking it as a method to teach you more about iOS development. This time around we're going back to the table views as seen in project 1, but you're also going to learn how to load text from files, how to ask for user input in **UIAlertController**, and get a little more insight to how closures work.

In Xcode, create a new Master-Detail Application called Project5. Select iPhone for your target, then click Next to save it somewhere. Right-click on DetailViewController.swift in the project navigator (the pane on the left showing all your files; Cmd+1 shows it), then choose Delete and click "Move to Trash" when prompted.

Doing this will cause quite a few errors to appear in your project, but it's easily fixed – we just need to delete quite a bit of Apple's template! First, open the AppDelegate.swift file, and look for this code near the top of the file:

```
func application(application: UIApplication,
didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?)
-> Bool {

  // Override point for customization after application launch.

  let splitViewController = self.window!.rootViewController as!
UISplitViewController

  let navigationController =
splitViewController.viewControllers[splitViewController.viewControlle
rs.count-1] as! UINavigationController


navigationController.topViewController!.navigationItem.leftBarButtonI
tem = splitViewController.displayModeButtonItem()

  splitViewController.delegate = self

  return true

}
```

Please delete everything in the body of that method except **return true**, like this:

```
func application(application: UIApplication,
didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?)
-> Bool {

  return true

}
```

Staying in that same file, scroll to the bottom to find this method with an extremely long signature:

```
func splitViewController(splitViewController: UISplitViewController,
collapseSecondaryViewController
secondaryViewController:UIViewController, ontoPrimaryViewController
primaryViewController:UIViewController) -> Bool {
```

Please delete that whole method.

Next, open the MasterViewController.swift file, and delete everything except **super.viewDidLoad()** from the **viewDidLoad()** method, then delete the **viewWillAppear()**, **insertNewObject()** and **prepareForSegue()** methods entirely. Finally, find the **tableView(_:canEditRowAtIndexPath:)** and **tableView(_:commitEditingStyle:forRowAtIndexPath:)** methods and delete those too.

The last code change is to find and delete this property near the top of the class:

```
var detailViewController: DetailViewController? = nil
```

That's all the code cleaned up, but we need to do a little Interface Builder surgery before we're complete. As you saw back in project 1, Apple's Master-Detail Application template sets up a split view controller plus two navigation controllers, as well as a table view

controller and a regular view controller with a detail label – this is all rather overkill for our simple application, so we're going to delete 3/5ths of that to make our whole app simpler.

So, open Main.storyboard in Interface Builder. Now select and delete the Split View Controller (the one on the far left with a dark gray background color), then delete the bottom navigation controller and its associated view controller. You should be left with just two things: a navigation controller, and, to its right, a table view controller.

Apple had it set up so that the split view controller was the initial view controller, which is what gets shown when the application launches. We just deleted that, so we need to pick a new initial view controller instead. To do that, select the remaining navigation controller, go to the Attributes Inspector (Alt+Cmd+4), and select the "Is Initial View Controller" checkbox that's about half way down the list of options. If it's worked, you should see an arrow appear to the left of the navigation controller.

The finished Interface Builder layout should look like the screenshot below:
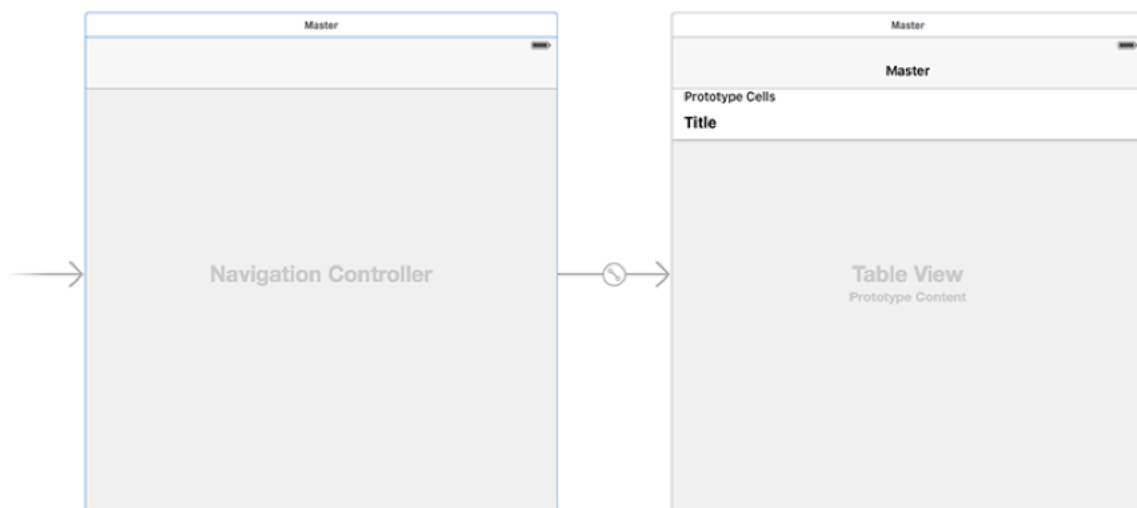
We've turned Apple's five view controllers into just two, which is much easier to manage.

The project is now so small that you might wander why we didn't just start from scratch! Still, there's a fair amount remaining, and deleting stuff can be quite cathartic. Even better: your project is now a clean table view project, ready for customisation – let's do this!

(PS: Is it faster to start with a Single View application? Possibly, but possibly not; at least this way you get some experience in cleaning up Apple's templates. Plus, deleting stuff is fun!)

Reading from disk

We're going to make an anagram game, where the user is asked to make words out of a larger word. We're going to put together a list of possible starter words for the game, and that list will be stored in a separate file. But how we get the text from the file into the app? Well, it turns out that Swift's **String** data type makes it a cinch – thanks, Apple!

If you haven't already downloaded the assets for this project from **hackingwithswift.com**, please do so now. In the Content folder you'll find the file start.txt. Please drag that into your Xcode project, making sure that "Copy items if needed" is checked.

The start.txt file contains over 12,000 eight-letter words we can use for our game, all stored one word per line. We need to turn that into an array of words we can play with. Behind the scenes, those line breaks are marked with a special line break character that is usually expressed as "\n". So, we need to load that word list into a string, then split it into an array by breaking up wherever we see **\n**.

First, go to the start of your class and make a new array. There will be an existing one there from Apple's template, so put this alongside:

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

While you're there, you might as well change Apple's array to be **[String]** rather than **[AnyObject]**, because we'll only ever be storing strings in there. You'll need to adjust the table view's **cellForRowAtIndexPath** method from this:

```
let object = objects[indexPath.row] as NSDate
cell.textLabel!.text = object.description
```

…to this:

```
let object = objects[indexPath.row]
cell.textLabel!.text = object
```

We did this in project 1 too, so hopefully it's not too hard. This is a result of the **objects** array no longer containing **AnyObject**, but just strings.

Second, loading our array. This is done in three parts: finding the path to our start.txt file, loading the contents of that file, then splitting it into an array.

Finding a path to a file is something you'll do a lot, because even though you know the file is called "start.txt" you don't know where it might be on the filesystem. So, we use a built-in method of **NSBundle** to find it: **pathForResource()**. This takes as its parameters the name of the file and its path extension, and returns a **String?** – i.e., you either get the path back or you get **nil**.

Loading a file into a string is also something you'll need to get familiar with, and again there's an easy way to do it: when you create an **String** instance, you can ask it to create itself from the contents of a file at a particular path. You can also provide it with parameters to tell you the text encoding that was used, but for this project we don't care.

Finally, we need to split our single string into an array of strings based on wherever we find a line break (**\n**). This is as simple as another method call on **String**: **componentsSeparatedByString()**. Tell it what string you want to use as a separator (for us, that's **\n**), and you'll get back an array.

Before we get onto the code, there are two things you should know: **pathForResource()** and creating an **String** from the contents of a file both return **String?**, which means we need to check and unwrap the optional using if/let syntax.

OK, time for some code. Put this into **viewDidLoad()**, after the **super** call:

```
if let startWordsPath =
NSBundle.mainBundle().pathForResource("start", ofType: "txt") {

  if let startWords = try? String(contentsOfFile: startWordsPath,
usedEncoding: nil) {

    allWords = startWords.componentsSeparatedByString("\n")

  }

} else {
```

```
    allWords = ["silkworm"]

}
```

If you look carefully, there's a new keyword in there: **try?**. You already saw **try!** previously, and really we could use that here too because we're loading a file from our app's bundle so any failure is likely to be catastrophic. However, this way I have a chance to teach you something new: **try?** means "call this code, and if it throws an error just send me back **nil** instead." This means the code you call will always work, but you need to unwrap the result carefully.

As you can see, that code carefully checks for and unwraps the contents of our start file, then converts it to an array. When it has finished, **allWords** will contain 12,000+ strings ready for us to use in our game.

To prove that everything is working before we continue, let's create a new method called **startGame()**. This will be called every time we want to generate a new word for the player to work with:

```
func startGame() {

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

  title = allWords[0]

  objects.removeAll(keepCapacity: true)

  tableView.reloadData()

}
```

Note the first line there is what shuffles the words arrow. We used this is in project 2, so you should remember that you need to import the GameplayKit framework in order for that to work.

Line 2 sets our view controller's title to be the first word in the array, once it has been shuffled. This will be the word the player has to find.

Line 3 removes all values from the **objects** array. This array was created for us by the Xcode template, and we'll be using it to store the player's answers so far. We aren't adding anything to it right now, so **removeAll()** won't do anything just yet.

Line 4 is the interesting part: it calls the **reloadData()** method of **tableView**. That table view was declared up near the… hey, wait a minute! Where's that table view come from? Well, we certainly didn't make it. Instead, it's made for us because – *dramatic drum roll* – **MasterViewController** doesn't inherit from **UIViewController**.

Yes, I know I said **UIViewController** is used for all the screens in your app, and it *is*, just sometimes not directly. In this case, **MasterViewController** inherits from **UITableViewController**, and **UITableViewController** inherits from **UIViewController**. There's an inheritance chain, with each component addings its own functionality.

Don't fear: most of your view controllers will inherit either directly from **UIViewController** or will go through **UITableViewController** first. It's just that table views are so pervasive in iOS that Apple took the opportunity to bake in some extra behavior for developers.

So what does **UITableViewController** do that **UIViewController** doesn't? Well, it comes with a full-screen table, for starters. When the view is shown, **UITableViewController** automatically flashes the scrollbars of the table so users know they can scroll, and if the keyboard appears then **UITableViewController** automatically adjusts itself so its content doesn't go beneath the keyboard.

Anyway, **UITableViewController** is what we are based on for **MasterViewController**, and that's where our **tableView** comes from. Calling **reloadData()** will cause the table view to check how many rows it has and reload them all.

Our table view doesn't have any rows yet, so this won't do anything. However, the method is ready to be used, and allows us to check we've loaded all the data correctly, so add this just before the end of **viewDidLoad()**:

```
startGame()
```

Now press Cmd+R to run the app, and you ought to see an eight-letter word at the top, ready for play to begin.

Our game so far is less than impressive, but trust me: it's all going to come together shortly!

Pick a word, any word

This game will prompt the user to enter a word that can be made from the eight-letter prompt word. For example, if the eight-letter word is "agencies", the user could enter "cease." We're going to solve this with **UIAlertController**, because it's a nice fit, and also gives me the chance to introduce some new teaching. I'm all about ulterior motives!

Add this code to **viewDidLoad()**, just after the call to **super**:

```
navigationItem.rightBarButtonItem =
UIBarButtonItem(barButtonSystemItem: .Add, target: self, action:
"promptForAnswer")
```

That creates a new UIBarButtonItem using the "add" system item, and configured it to run a method called **promptForAnswer()** when tapped. This method is going to show a **UIAlertController** with space for the user to enter an answer, and when the user clicks Submit to that alert controller the answer is checked to make sure it's valid.

Before I give you the code, let me explain what you need to know.

You see, we're about to use a closure, and things get a little complicated. As a reminder, these are chunks of code that can be treated like a variable – we can send the closure somewhere, where it gets stored away and executed later. To make this work, Swift takes a copy of the code and captures any objects it references, so it can use them later.

But there's a problem: what if the closure references the view controller? Then what could happen is a strong reference cycle: the view controller owns an object that owns a closure that owns the view controller, and nothing could ever be destroyed.

I'm going to try (and likely fail!) to give you a metaphorical example, so please bear with me. Imagine if you built two cleaning robots, red and blue. You told the red robot, "don't stop cleaning until the blue robot stops," and you told the blue robot "don't stop cleaning until the red robot stops." When would they stop cleaning? Never, because neither will make the first move.

This is the problem we are facing with a strong reference cycle: object A owns object B, and object B owns a closure that referenced object A. And when closures are created, they

capture everything they use, thus object B owns object A.

Strong reference cycles used to be hard to find, but you'll be glad to know Swift makes them trivial. In fact, Swift makes it so easy that you will use its solution even when you're not sure if there's a cycle simply because you might as well.

So, please brace yourself: we're about to take our first look at actual closures. The syntax will hurt. And when you finally understand it, you'll come across examples online that make your brain hurt all over again.

Ready? Here's the **promptForAnswer()** method:

```
func promptForAnswer() {

  let ac = UIAlertController(title: "Enter answer", message: nil,
preferredStyle: .Alert)

  ac.addTextFieldWithConfigurationHandler(nil)


  let submitAction = UIAlertAction(title: "Submit", style: .Default)
{ [unowned self, ac] (action: UIAlertAction!) in

    let answer = ac.textFields![0]

    self.submitAnswer(answer.text!)

  }


  ac.addAction(submitAction)


  presentViewController(ac, animated: true, completion: nil)

}
```

That one method introduces quite a few new things, but before we look at them let's eliminate the easy stuff.

 • Creating a new **UIAlertController**: we did that in project 2.
 • **addTextFieldWithConfigurationHandler()** just adds an editable text field to the

UIAlertController. We could do more with it, but it's enough for now.

 • **addAction()** is used to add a **UIAlertAction** to a **UIAlertController**. We used this in project 2 also.

 • **presentViewController** is also from project 2. Clearly project 2 was brilliant!

That leaves the tricky stuff: creating **submitAction**. These handful of lines of code demonstrate no fewer than five new things to learn, all of which are important. I'm going to sort them easiest first, starting with **UITextField**.

You've already seen **UILabel**: it's a simple **UIView** subclass that shows a string of uneditable text on the screen. **UITextField** is similar, except it's editable. We added a single text field to the **UIAlertController** using its **addTextFieldWithConfigurationHandler()** method, and we now read out the value that was inserted.

Next up is something called *trailing closure syntax*. I know, I know: you haven't even learned about regular closures yet, and you're already having to learn about trailing closures! Well, they are related, and trailing closures aren't hard, so give it a chance.

Here's part of a line of code from project 2:

```
UIAlertAction(title: "Continue", style: .Default, handler:
askQuestion)
```

This is from a similar situation: we're using **UIAlertController** and **UIAlertAction** to add buttons that the user can tap on. Back then, we used a separate method (**askQuestion()**) to avoid having to explain closures too early, but you can see that I'm passing **askQuestion** to the **handler** parameter of the **UIAlertAction**.

Closures can be thought of as a bit like anonymous functions. That is, rather than passing the name of a function to execute, we're just passing a lump of code. So we could conceptually rewrite that line to be this:

```
UIAlertAction(title: "Continue", style: .Default, handler: { CLOSURE
CODE HERE })
```

But that has one critical problem: it's ugly! If you're executing lots of code inside the closure, it looks strange to have a one-line function call taking a 10-line parameter.

So, Swift has a solution, called trailing closure syntax. Any time you are calling a method that expects a closure as its final parameter – and there are many of them – you can eliminate that final parameter entirely, and pass it inside braces instead. This is optional and automatic, and would turn our conceptual code into this:

```
UIAlertAction(title: "Continue", style: .Default) {

  CLOSURE CODE HERE

}
```

Everything from the opening brace to the close is part of the closure, and is passed into the **UIAlertAction** creation as its last parameter. Easy!

Next, **(action: UIAlertAction!) in**. If you remember project 2, we had to modify the **askQuestion()** method so that it accepted a **UIAlertAction** parameter saying what button was tapped, like this:

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

We had no choice but to do that, because the **handler** parameter for **UIAlertAction** expects a method that takes itself as a parameter. And that's what's happening here: we're giving the UIAlertAction some code to execute when it is tapped, and it wants to know that that code accepts a parameter of type **UIAlertAction**.

The **in** keyword is important: everything before that describes the closure; everything after that *is* the closure. So **(action: UIAlertAction!) in** means that it accepts one parameter in, of type **UIAlertAction**.

I used this way of writing the closure because it's so similar to that used in project 2. However, Swift knows what kind of closure this needs to be, so we could simplify it a little: from this…

```
(action: UIAlertAction!) in
```

…to this:

```
action in
```

In our current project, we could simplify this even further: we don't make any reference to the **action** parameter inside the closure, which means we don't need to give it a name at all. In Swift, to leave a parameter unnamed you just use an underscore character, like this:

```
_ in
```

Fourth and fifth are going to be tackled together: **unowned** and **self.**.

Swift "captures" any constants and variables that are used in a closure, based on the values of the closure's surrounding context. That is, if you create an integer, a string, an array and another class outside of the closure, then use them inside the closure, Swift captures them.

This is important, because the closure references the variables, and might even change them. But I haven't said yet what "capture" actually means, and that's because it depends what kind of data you're using. Fortunately, Swift hides it all away so you don't have to worry about it…

…except for those strong reference cycles I mentioned. Those you need to worry about. That's where objects can't even be destroyed because they all hold tightly on to each other – known as *strong referencing*.

Swift's solution is to let you declare that some variables aren't held onto quite so tightly. It's a two-step process, and it's so easy you'll find yourself doing it for everything just in case.

First, you must tell Swift what variables you don't want strong references for. This is done in one of two ways: **unowned** or **weak**. These are somewhat equivalent to implicitly unwrapped

optionals (unowned) and regular optionals (weak): a weakly owned reference might be **nil**, so you need to unwrap it; an unowned reference is one you're certifying cannot be **nil** and so doesn't need to be unwrapped, however you'll hit a problem if you were wrong.

In our code we use this: **[unowned self, ac]**. That declares **self** (the current view controller) and **ac** (our **UIAlertController**) to be captured as unowned references inside the closure. It means the closure can use them, but won't create a strong reference cycle because we've made it clear the closure doesn't own either of them.

But that's not enough for Swift. Inside our method we're calling the **submitAnswer()** method of our view controller. We haven't created it yet, but you should be able to see it's going to take the answer the user entered and try it out in the game.

This **submitAnswer()** method is external to the closures current context, so when you're writing it you might not realise that calling **submitAnswer()** implicitly requires that **self** be captured by the closure. That is, the closure can't call **submitAnswer()** if it doesn't capture the view controller.

We've already declared that **self** is unowned by the closure, but Swift wants us to be absolutely sure we know what we're doing: every call to a method or property of the current view controller must prefixed with "**self.**", as in **self.submitAnswer()**.

In project 1, I told you there were two trains of thought regarding use of **self**, and said, "The first group of people never like to use **self.** unless it's required, because when it's required it's actually important and meaningful, so using it in places where it isn't required can confuse matters."

Implicit capture of **self** in closures is that place when using **self** is required and meaningful: Swift won't let you avoid it here. By restricting your use of **self** to closures, you can easily check your code doesn't have any reference cycles by searching for "self" – there ought not to be too many to look through!
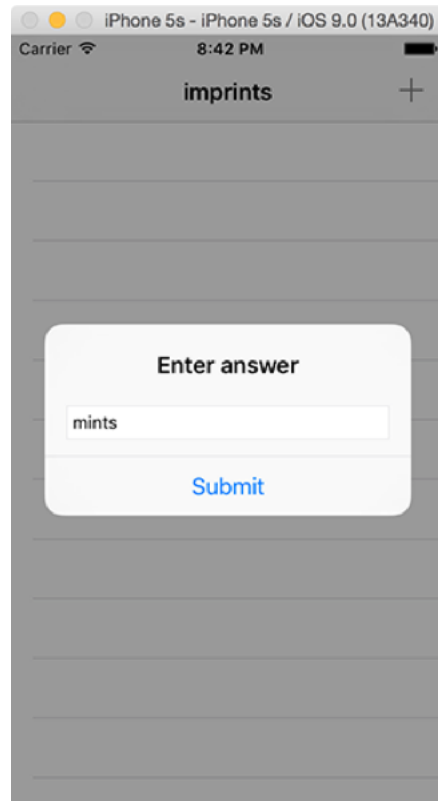
You can add multiple text fields to an alert controller, which is perfect for accepting quick user input.

Prepare for submission

OK, you can breathe again: we're done with closures for now. I know that wasn't easy, but once you understand basic closures you really have come a long in your Swift adventure.

We're going to do some much easier coding now, because believe it or not we're not that far from making this game actually work!

First, let's make your code compile again, because right now it's calling **self.submitAnswer()** and we haven't made that method yet. So, add this new method somewhere in the class:

```
func submitAnswer(answer: String) {

}
```

That's right, it's empty – it's enough to make the code compile cleanly so we can carry on.

We have now gone over the structure of a closure: trailing closure syntax, unowned self, a parameter being passed in, then the need for **self.** to make capturing clear. We haven't really talked about the actual content of our closure, because there isn't a lot to it:

```
let answer = ac.textFields![0]

self.submitAnswer(answer.text!)
```

The first line force unwraps the array of text fields (it's optional because there might not be any; we can force unwrap because we know we added one), then tells Swift to treat it as a **UITextField**. The second line pulls out the text from the text field and passes it to our (all-new-albeit-empty) **submitAnswer()** method.

This method needs to check whether the player's word can be made from the given letters. It needs to check whether the word has been used already, because obviously we don't want duplicate words. It also needs to check whether the word is actually a valid English word, because otherwise the user can just type in nonsense.

If all three of those checks pass, **submitAnswer()** needs to add the word to the **objects**

array, then insert a new row in the table view. We could just use the table view's **reloadData()** method to force a full reload, but that's not very efficient when we're changing just one row.

Here's our first pass at filling in the **submitAnswer()** method:

```
func submitAnswer(answer: String) {

  let lowerAnswer = answer.lowercaseString


  if wordIsPossible(lowerAnswer) {

    if wordIsOriginal(lowerAnswer) {

      if wordIsReal(lowerAnswer) {

        objects.insert(answer, atIndex: 0)


        let indexPath = NSIndexPath(forRow: 0, inSection: 0)

        tableView.insertRowsAtIndexPaths([indexPath],
withRowAnimation: .Automatic)

      }

    }

  }

}
```

Ignore the **wordIsPossible()**, **wordIsOriginal()** and **wordIsReal()** methods for now – let's focus on the rest of the code first.

If a user types "cease" as a word that can be made out of our started word "agencies", it's clear that is correct because there is one "c", two "e"s, one "a" and one "s". But what if they type "Cease"? Now it has a capital C, and "agencies" doesn't have a capital C. Yes, that's right: strings are case-sensitive, which means Cease is not cease is not CeasE is not CeAsE.

The solution to this is quite simple: all the starter words are lowercase, so when we check the player's answer we immediately lowercase it using its **lowercaseString** property. This is stored in the **lowerAnswer** constant because we want to use it several times.

We then have three if statements, one inside another. These are called nested statements, because you nest one inside the other. Only if all three statements are true (the word is possible, the word hasn't been used yet, and the word is a real word), does the main block of code execute.

Once we know the word is good, we do three things: insert the new word into our **objects** array at index 0. This means "add it to the start of the array," and means that the newest words will appear at the top of the table view.

The next two things are related: we insert a new row into the table view. Given that the table view gets all its data from the objects array, this might seem strange. After all, we just inserted the word into the **objects** array, so why do we need to insert anything into the table view?

The answer is animation. Like I said, we could just call the **reloadData()** method and have the table do a full reload of all rows, but it means a lot of extra work for one small change, and also causes a jump – the word wasn't there, and now it is.

This can be hard for users to track visually, so using **insertRowsAtIndexPaths()** lets us tell the table view that a new row has been placed at a specific place in the array so that it can animate the new cell appearing. Adding one cell is also significantly easier than having to reload everything, as you might imagine!

There are two quirks here that require a little more detail. First, **NSIndexPath** is something we looked at briefly in project 1, as it contains a section and a row for every item in your table. As with project 1 we aren't using sections here, but the row number should equal the position we added the item in the array – position 0, in this case.

Second, the **withRowAnimation** parameter lets you specify how the row should be animated in. Whenever you're adding and removing things from a table, the **.Automatic** value means "do whatever is the standard system animation for this change." In this case, it means "slide the new row in from the top."

Your code won't compile yet, because we still have three missing methods that need to be done. Add these three beneath the **submitAnswer()** method to get everything working again:

```
func wordIsPossible(word: String) -> Bool {
    return true
```

```
}


func wordIsOriginal(word: String) -> Bool {

  return true

}


func wordIsReal(word: String) -> Bool {

  return true

}
```

We'll look at what that does shortly, then fill it out with lots of real code. But for now, press Cmd+R to play back what you have – you should be able to tap the + button and enter words into the alert.

Returning values

None of our custom methods have returned values so far. We used the **return** keyword briefly in project 4 to exit the web view's **decidePolicyForNavigationAction** method early, but it didn't send any data back. So, let's take things another step forward.

As I said, the **return** keyword exits a method at any time it's used. If you use **return** by itself, it exits the method and does nothing else. But if you use **return** with a value, it sends that value back to whatever called the method.

Before you can send a value back, you need to tell Swift that you expect to return a value. Swift will automatically check that a value is returned and it's of the right data type, so this is important. We just put in stubs (empty methods that do nothing) for three new methods, each of which returns a value. Let's take a look at one in more detail:

```swift
func wordIsOriginal(word: String) -> Bool {

  return true

}
```

The method is called **wordIsOriginal()**, and it takes one parameter that's a string. But before the opening brace there's something new: **-> Bool**. This tells Swift that the method will return a boolean value, which is the name for a value that can be either true or false.

The body of the method has just one line of code: **return true**. This is how the **return** statement is used to send a value back to its caller: we're returning true from this method, so the caller can use this method inside an **if** statement to check for true or false.

This method can have as much code as it needs in order to evaluate fully whether the word has been used or not, including calling any other methods it needs. We're going to change it so that it calls another method, which will check whether our **objects** array already contains the word that was provided. Replace its current **return true** code with this:

```swift
return !objects.contains(word)
```

There are two new things here. First, **contains()** is a method that checks whether the array specified in parameter 1 (**objects**) contains the value specified in parameter 2 (**word**). If it does contain the value, **contains()** returns true; if not, it returns false. Second, the **!** symbol. You've seen this before as the way to force unwrap optional variables, but here it's something different: it means *not*.

The difference is small but important: when used before a variable or constant, **!** means "not" or "opposite". So if **contains()** returns true, **!** flips it around to make it false, and vice versa. When used after a variable or constant, **!** means "force unwrap this optional variable."

This is used because our method is called **wordIsOriginal()**, and should return true if the word has never been used before. If we had used **return objects.contains(word)**, then it would do the opposite: it would return true if the word had been used and false otherwise. So, by using **!** we're flipping it around so that the method returns true if the word is new.

That's one method down. Next is the **wordIsPossible()**, which also takes a string as its only parameter and returns a **Bool** – true or false. This one is more complicated, but I've tried to make the algorithm as simple as possible.

How can we be sure that "cease" can be made from "agencies", using each letter only once? The solution I've adopted is to loop through every letter in the player's answer, seeing whether it exists in the eight-letter start word we are playing with. If it does exist, we remove the letter from the start word, then continue the loop. So, if we try to use a letter twice, it will exist the first time, but then get removed so it doesn't exist the next time, and the check will fail.

You already met the **rangeOfString()** method in project 4, so this should be straightforward:

```
func wordIsPossible(word: String) -> Bool {
    var tempWord = title!.lowercaseString

    for letter in word.characters {
        if let pos = tempWord.rangeOfString(String(letter)) {
            tempWord.removeAtIndex(pos.startIndex)
        } else {
            return false
```

```
    }

  }


  return true
}
```

Our usage of **rangeOfString()** is a little different than from project 4. Remember, **rangeOfString()** returns an optional position of where the item was found – meaning that it might be **nil**. So, we wrap the call into an if/let to safely unwrap the optional.

The usage is also different because we use **String(letter)** rather than just **letter**. This is because our **for** loop is used on a string, and it pulls out every letter in the string as a new data type called **Character** – i.e., a single letter. **rangeOfString()** expects a string, not a character, so we need to create a string from the character using **String(letter)**.

If the letter was found in the string, we use **removeAtIndex()** to remove the used letter from the **tempWord** variable. This is why we need the **tempWord** variable at all: because we'll be removing letters from it so we can check again the next time the loop goes around.

The method ends with **return true**, because this line is reached only if every letter in the user's word was found in the start word no more than once. If any letter isn't found, or is used more than possible, one of the **return false** lines would have been hit, so by this point we're sure the word is good.

Important: we have told Swift that we are returning a boolean value from this method, and it will check every possible outcome of the code to make sure a boolean value is returned no matter what.

Time for the final method. Replace the current **wordIsReal()** method with this:

```
func wordIsReal(word: String) -> Bool {
  let checker = UITextChecker()
  let range = NSMakeRange(0, word.characters.count)
  let misspelledRange = checker.rangeOfMisspelledWordInString(word,
```

```
range: range, startingAt: 0, wrap: false, language: "en")


    return misspelledRange.location == NSNotFound
}
```

There's a new class here, called **UITextChecker**. This is an iOS class that is designed to spot spelling errors, which makes it perfect for knowing if a given word is real or not. We're creating a new instance of the class and putting it into the **checker** constant for later.

There's a new function call here too, called **NSMakeRange()**. This is used to make a string range, which is a value that holds a start position and a length. We want to examine the whole string, so we use 0 for the start position and the string's length for the length.

Next, we call the **rangeOfMisspelledWordInString()** method of our **UITextChecker** instance. This wants five parameters, but we only care about the first two and the last one: the first parameter is our string, **word**, the second is our range to scan (the whole string), and the last is the language we should be checking with, where **en** selects English.

Parameters three and four aren't useful here, but for the sake of completeness: parameter three selects a point in the range where the text checker should start scanning, and parameter four lets us set whether the **UITextChecker** should start at the beginning of the range if no misspelled words were found starting from parameter three. Neat, but not helpful here.

Calling **rangeOfMisspelledWordInString()** returns an **NSRange** structure, which tells us where the misspelling was found. But what we care about was whether any misspelling was found, and if nothing was found our **NSRange** will have the special location **NSNotFound**. Usually location would tell you where the misspelling started, but **NSNotFound** is telling us the word is spelled correctly – i.e., it's a valid word.

Here the **return** statement is used in a new way: as part of an operation involving ==. This is a very common way to code, and what happens is that == returns true or false depending on whether **misspelledRange.location** is equal to **NSNotFound** That true or false is then given to **return** as the return value for the method.

We could have written that same line across multiple lines, but it's not common:

```
if misspelledRange.location == NSNotFound {

  return true

} else {

  return false

}
```

That completes the third of our missing methods, so the project is almost complete. Run it now and give it a thorough test!

Or else what?

There remains one problem to fix with our code, and it's quite a tedious problem. If the word is possible and original and real, we add it to the list of found words then insert it into the table view. But what if the word isn't possible? Or if it's possible but not original? In this case, we reject the word and don't say why, so the user gets no feedback.

So, the final part of our project is to give users feedback when they make an invalid move. This is tedious because it's just adding **else** statements to all the **if** statements in **submitAnswer()**, each time configuring a message to show to users.

Here's the adjusted method:

```
func submitAnswer(answer: String) {

  let lowerAnswer = answer.lowercaseString


  let errorTitle: String
  let errorMessage: String


  if wordIsPossible(lowerAnswer) {

    if wordIsOriginal(lowerAnswer) {

      if wordIsReal(lowerAnswer) {

        objects.insert(answer, atIndex: 0)


        let indexPath = NSIndexPath(forRow: 0, inSection: 0)
        tableView.insertRowsAtIndexPaths([indexPath],
withRowAnimation: .Automatic)


        return

      } else {
        errorTitle = "Word not recognised"
        errorMessage = "You can't just make them up, you know!"
      }
```

```
    } else {

      errorTitle = "Word used already"

      errorMessage = "Be more original!"

    }

  } else {

    errorTitle = "Word not possible"

    errorMessage = "You can't spell that word from '\
(title!.lowercaseString)'!"

  }


  let ac = UIAlertController(title: errorTitle, message:
errorMessage, preferredStyle: .Alert)

  ac.addAction(UIAlertAction(title: "OK", style: .Default, handler:
nil))

  presentViewController(ac, animated: true, completion: nil)

}
```

As you can see, every **if** statement now has a matching **else** statement so that the user gets appropriate feedback. All the **else**s are effectively the same (albeit with changing text): set the values for **errorTitle** and **errorMessage** to something useful for the user. The only interesting exception is the last one, where we use string interpolation (remember project 2?) to show the view controller's title as a lowercase string.

If the user enters a valid answer, a call to **return** forces Swift to exit the method immediately once the table has been updated. This is helpful, because at the end of the method there is code to create a new **UIAlertController** with the error title and message that was set, add an OK button with a **nil** handler (i.e., just dismiss the alert), then show the alert. So, this error will only be shown if something went wrong.

This demonstrates one important tip about Swift constants: both **errorTitle** and **errorMessage** were declared as constants, which means their value cannot be changed once set. I didn't give either of them an initial value, and that's OK – Swift lets you do this as long as you do provide a value before the constants are read, and also as long as you don't try to change the value again later on.

Other than that, your project is done. Go and play!

Wrap up

You've made it this far, so your Swift learning really is starting to come together, and I hope this project has shown you that you can make some pretty advanced things with your knowledge.

In this project, you learned a little bit more about **UITableView**: how to reload their data and how to insert rows. You also learned how to add text fields to **UIAlertController** so that you can accept user input. But you also learned some serious core stuff: more about Swift strings, closures, method return values, booleans, **NSRange**, and more. These are things you're going to use in dozens of projects over your Swift coding career, and things we'll be returning to again and again in this series.

You may already have plans for how you'd like to improve this game, but if not here are four ideas to get you started:

1. Disallow answers that are shorter than three letters. The easiest way to accomplish this is to put a check into **wordIsReal()** that returns false if the word length is under three letters.
2. Refactor all the **else** statements we just added so that they call a new method called **showErrorMessage()**. This should accept an error message and a title, and do all the **UIAlertController** work from there.
3. Disallow answers that are just the start word. Right now, if the start word is "agencies" the user can just submit "agencies" as an answer, which is too easy – stop them from doing that.
4. Fix our start.txt loading code. If we the **pathForResource()** call returns **nil** we load an array containing one word: silkworm. But what if **pathForResource()** succeeds, but creating an **NSString** using **contentsOfFile** fails? Then the array is empty! Make a new **loadDefaultWords()** method that can be used for both failures.

Project 6

Auto Layout

Get to grips with Auto Layout using practical examples and code.

Setting up

In this technique project you're going to learn more about Auto Layout, the powerful and expressive way iOS lets you design your layouts. We used it in project 2 to make sure our flag buttons were positioned correctly, but that project has a problem: if you rotate your device, the flags don't fit on the screen!

So, we're first going to fix project 2 so that it demonstrates more advanced Auto Layout techniques (while also making the flags stay on the screen correctly!), then take a look at ways you can use Auto Layout in code.

First: take a copy of project 2, call it project6a, then open it in Xcode. All set? Then let's begin…

Advanced Auto Layout

When you run the project, it looks fine in portrait, but is unplayable on landscape because some of the buttons are hidden. You have two options: either disable landscape mode, or make your layout work across both orientations.

Disabling orientations isn't a great solution, but sometimes it's the *right* solution. Most games, for example, fix their orientation because it just doesn't make sense to support both. If you want to do this, press Cmd+1 to show the project navigator on the left of your Xcode window, select your project (it's the first item in the pane), then to the right of where you just clicked will appear another pane showing "PROJECT" and "TARGETS", along with some more information in the center.

Please note: This project and targets list can be hidden by clicking the disclosure button in the top-left of the project editor (directly beneath the icon with four squares), and you may find yours is already hidden. I strongly recommend you show this list – hiding it will only make things harder to find, so please make sure it's visible!

In the picture below you can see the project editor, with the device orientations at the bottom. This is the collapsed view of projects and targets, so there's a dropdown arrow at the top that says "Project2" (just above where it says Identity in bold), and to the left of that is the button to show the projects and targets list.
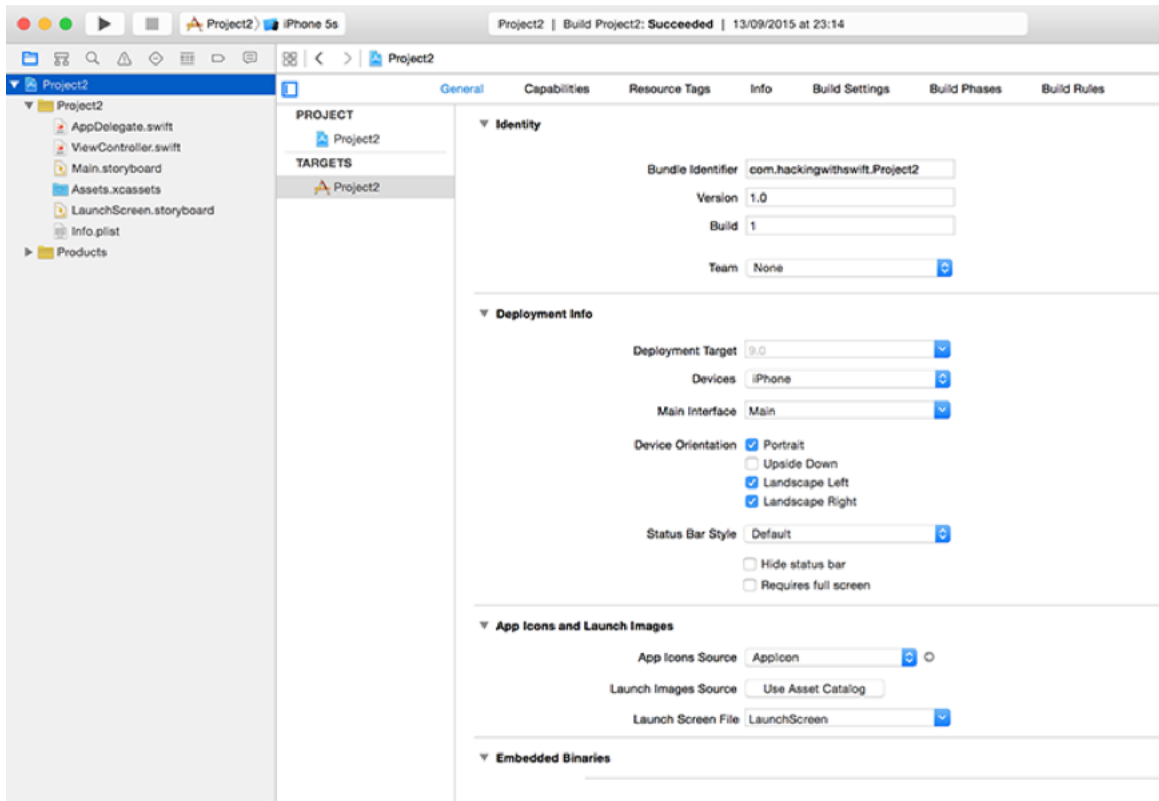
Use the project editor to set the device orientations you want to support.

This view is called the project editor, and contains a huge number of options that affect the way your app works. You'll be using this a lot in the future, so remember how to get here! Select Project 2 under TARGETS, then choose the General tab, and scroll down until you see four checkboxes called Device Orientation. You can select only the ones you want to support.

You'll need to support selective orientations in some later projects, but for now let's take the smart solution: add extra rules to Auto Layout so it can make the layout work great in landscape mode.

Open Main.storyboard in Interface Builder, select the bottom flag, then Ctrl-drag from the flag to the white space directly below the flag – in the view controller itself. The direction you drag is important, so please drag straight down.

When you release your mouse button, a popup will appear that includes the option "Bottom Space to Bottom Layout Guide" – please select that. This creates a new Auto Layout constraint that the bottom of the flag must be at least X points away from the bottom of the

view controller, where X is equal to whatever space there is in there now.

Although this is a valid rule, it will screw up your layout because we now have a complete set of exact vertical rules: the top flag should be 36 points from the top, the second 30 from the first, the third 30 from the second, and the third X away from the bottom. It's 140 for me, but yours might be different.

Because we've told Auto Layout exactly how big all the spaces should be, it will add them up and divide the remaining space among the three flags however it thinks best. That is, the flags must now be stretched vertically in order to fill the space, which is almost certainly what we don't want.

Instead, we're going to tell Auto Layout where there is some flexibility, and that's in the new bottom rule we just created. The bottom flag doesn't need to be 140 points away from the bottom layout guide – it just needs to be *some* distance away, so that it doesn't touch the edge. If there is more space, great, Auto Layout should use it, but all we care about is the minimum.

Select the third flag to see its list of constraints drawn in blue, then (carefully!) select the bottom constraint we just added. In the utilities view on the right, choose the attributes inspector (Alt+Cmd+4), and you should see Relation set to Equal and Constant set to 140 (or some other value, depending on your layout).

What you need to do is change Equal to be "Greater Than or Equal", then change the Constant value to be 20. This sets the rule "make it at least 20, but you can make it more to fill up space." The layout won't change visually while you're doing this, because the end result is the same. But at least now that Auto Layout knows it has some flexibility beyond just stretching the flags!

Our problem is still not fixed, though: in landscape, and iPhone 4s has just 320 points of space to work with, so Auto Layout is going to make our flags fit by squashing one or maybe even two of them. Squashed flags aren't good, and having uneven sizes of flags isn't good either, so we're going to add some more rules.

Select the second button, then Ctrl-drag to the first button. When given the list of options, choose Equal Heights. Now do the same from the third button to the second button. This rule ensures that at all times the three flags have the same height, so Auto Layout can no longer squash one button to make it all fit and instead has to squash all three equally.

That fixes part of the problem, but in some respects it has made things worse. Rather than having one squashed flag, we now have three! But with one more rule, we can stop the flags from being squashed ever. Select the first button, then Ctrl-drag a little bit upwards – but stay within the button! When you release your mouse button, you'll see the option "Aspect Ratio", so please choose it.

The Aspect Ratio constraint solves the squashing once and for all: it means that if Auto Layout is forced to reduce the height of the flag, it will reduce its width by the same proportion, meaning that the flag will always look correct. Add the Aspect Ratio constraint to the other two flags, and run your app again. It should work great in portrait and landscape, all thanks to Auto Layout!

Auto Layout in code

Create a new Single View Application project in Xcode, name it Project6b and set its target to be iPhone. We're going to create some views by hand, then position them using Auto Layout. Put this into your **viewDidLoad()** method:

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

  let label1 = UILabel()
  label1.translatesAutoresizingMaskIntoConstraints = false
  label1.backgroundColor = UIColor.redColor()
  label1.text = "THESE"

  let label2 = UILabel()
  label2.translatesAutoresizingMaskIntoConstraints = false
  label2.backgroundColor = UIColor.cyanColor()
  label2.text = "ARE"

  let label3 = UILabel()
  label3.translatesAutoresizingMaskIntoConstraints = false
  label3.backgroundColor = UIColor.yellowColor()
  label3.text = "SOME"

  let label4 = UILabel()
  label4.translatesAutoresizingMaskIntoConstraints = false
  label4.backgroundColor = UIColor.greenColor()
  label4.text = "AWESOME"

  let label5 = UILabel()
  label5.translatesAutoresizingMaskIntoConstraints = false
```

```
    label5.backgroundColor = UIColor.orangeColor()

    label5.text = "LABELS"


    view.addSubview(label1)

    view.addSubview(label2)

    view.addSubview(label3)

    view.addSubview(label4)

    view.addSubview(label5)

}
```

Put to one side what that code does for a moment, please add this method somewhere after **viewDidLoad()**:

```
override func prefersStatusBarHidden() -> Bool {

    return true

}
```

That tells iOS we don't want to show the iOS status bar on this view controller – that's the bit that tells you what time it is.

OK, back to **viewDidLoad()**: all that code creates five **UILabel** objects, each with unique text and a unique background color. All five views then get added to the view belonging to our view controller by using **view.addSubview()**. We also set the property **translatesAutoresizingMaskIntoConstraints** to be **false** on each label, because by default iOS generates Auto Layout constraints for you based on a view's size and position. We'll be doing it by hand, so we need to disable this feature.

If you run the app now, you'll see seem some colorful labels at the top, overlapping so it looks like it says "LABELS ME". That's because our labels are placed in their default position (at the top-left of the screen) and are all sized to fit their content.

We're going to do is add some constraints that say each label should start at the left edge of its superview, and end at the right edge. We're going to do this using a technique called Auto

Layout Visual Format Language (VFL), which is kind of like a way of drawing the layout you want with a series of keyboard symbols.

Before we do that, we need to create a dictionary of the views we want to lay out. This is a new data type, but it's quite easy to understand. You've already met arrays, which hold values that can be read using their order, for example **myArray[0]** reads the first item from the array. Rather than making you use numbers, dictionaries let you specify any object as the access method (known as the "key"), for example you could read out via a string: **myDictionary["name"]**.

The reason this is needed for VFL will become clear shortly, but first here's the dictionary you need to add below the last call to **addSubview()**:

```
let viewsDictionary = ["label1": label1, "label2": label2, "label3":
label3, "label4": label4, "label5": label5]
```

That creates a dictionary with strings (the keys) and our **UILabel**s as its values (the values). So, to get access to **label1**, we can now use **viewsDictionary["label1"]**. This might seem redundant, but wait just a moment longer: it's time for some Visual Format Language!

Add these lines directly below the **viewsDictionary** that was just created:

```
view.addConstraints(NSLayoutConstraint.constraintsWithVisualFormat("H
:|[label1]|", options: [], metrics: nil, views: viewsDictionary))

view.addConstraints(NSLayoutConstraint.constraintsWithVisualFormat("H
:|[label2]|", options: [], metrics: nil, views: viewsDictionary))

view.addConstraints(NSLayoutConstraint.constraintsWithVisualFormat("H
:|[label3]|", options: [], metrics: nil, views: viewsDictionary))

view.addConstraints(NSLayoutConstraint.constraintsWithVisualFormat("H
:|[label4]|", options: [], metrics: nil, views: viewsDictionary))

view.addConstraints(NSLayoutConstraint.constraintsWithVisualFormat("H
:|[label5]|", options: [], metrics: nil, views: viewsDictionary))
```

That's a lot of code, but actually it's just the same thing five times over. As a result, we could

easily rewrite those fix in a loop, like this:

```
for label in viewsDictionary.keys {

view.addConstraints(NSLayoutConstraint.constraintsWithVisualFormat("H
:|[\(label)]|", options: [], metrics: nil, views: viewsDictionary))

}
```

Note that we're using string interpolation to put the key ("label1", etc) into the VFL.

Let's eliminate the easy stuff, then focus on what remains.

 • **view.addConstraints()**: this adds an array of constraints to our view controller's view. This array is used rather than a single constraint because VFL can generate multiple constraints at a time.
 • **NSLayoutConstraint.constraintsWithVisualFormat()** is the Auto Layout method that converts VFL into an array of constraints. It accepts lots of parameters, but the important ones are the first and last.
 • We pass **[]** (an empty array) for the options parameter and **nil** for the metrics parameter. You can use these options to customise the meaning of the VFL, but for now we don't care.

That's the easy stuff. So, let's look at the Visual Format Language itself: **"H:|[label1]|"**. As you can see it's a string, and that string describes how we want the layout to look. That VFL gets converted into Auto Layout constraints, then added to the view.

The **H:** parts means that we're defining a horizontal layout; we'll do a vertical layout soon. The pipe symbol, |, means "the edge of the view." We're adding these constraints to the main view inside our view controller, so this effectively means "the edge of the view controller." Finally, we have **[label1]**, which is a visual way of saying "put **label1** here". Imagine the brackets, [and], are the edges of the view.

So, **"H:|[label1]|"** means "horizontally, I want my **label1** to go edge to edge in my view." But there's a hiccup: what is "label1"? Sure, *we* know what it is because it's the name of our variable, but variable names are just things for humans to read and write – the variable names aren't actually saved and used when the program runs.

This is where our **viewsDictionary** dictionary comes in: we used strings for the key and **UILabels** for the value, then set "label1" to be our label. This dictionary gets passed in along with the VFL, and gets used by iOS to look up the names from the VFL. So when it sees **[label1]**, it looks in our dictionary for the "label1" key and uses its value to generate the Auto Layout constraints.

That's the entire VFL line explained: each of our labels should stretch edge-to-edge in our view. If you run the program now, that's sort of what you'll see, although it highlights our second problem: we don't have a vertical layout in place, so although all the labels sit edge-to-edge in the view, they all overlap.

We're going to fix this with another set of constraints, but this time it's just one (long) line.

```
view.addConstraints(NSLayoutConstraint.constraintsWithVisualFormat("V
:|[label1]-[label2]-[label3]-[label4]-[label5]", options: [],
metrics: nil, views: viewsDictionary))
```

That's identical to the previous five, except for the VFL part. This time we're specifying **V:**, meaning that these constraints are vertical. And we have multiple views inside the VFL, so lots of constraints will be generated. The new thing in the VFL this time is the **-** symbol, which means "space". It's 10 points by default, but you can customise it.

Note that our vertical VFL doesn't have a pipe at the end, so we're not forcing the last label to stretch all the way to the edge of our view. This will leave whitespace after the last label, which is what we want right now.

If you run your program now, you'll see all five labels stretching edge-to-edge horizontally, then spaced neatly vertically. It would have taken quite a lot of Ctrl-dragging in Interface Builder to make this same layout, so I hope you can appreciate how powerful VFL is!

Sizes, metrics and priorities

We have a working layout now, but it's quite basic: the labels aren't very high, and without a rule regarding the bottom of the last label it's possible the views might be pushed off the bottom edge.

To begin to fix this problem, we're going to add a constraint for the bottom edge saying that the bottom of our last label must be at least 10 points away from the bottom of the view controller's view. We're also going to tell Auto Layout that we want each of the five labels to be 88 points high. Replace the previous vertical constraints with this:

```
view.addConstraints(NSLayoutConstraint.constraintsWithVisualFormat("V
:|[label1(==88)]-[label2(==88)]-[label3(==88)]-[label4(==88)]-
[label5(==88)]-(>=10)-|", options: [], metrics: nil, views:
viewsDictionary))
```

The difference here is that we now have numbers inside parentheses: **(==88)** for the labels, and **(>=10)** for the space to the bottom. Note that when specifying the size of a space, you need to use the - before and after the size: a simple space, -, becomes **-(>=10)-**.

We are specifying two kinds of size here: **==** and **>=**. The first means "exactly equal" and the second "greater than or equal to." So, our labels will be forced to be an exact size, and we ensure that there's some space at the bottom while also making it flexible – it will definitely be at least 10 points, but could be 100 or more depending on the situation.

Actually, wait a minute. I didn't want 88 points for the label size, I meant 80 points. Go ahead and change all the labels to 80 points high.

Whoa there! It looks like you just received an email from your IT director: he thinks 80 points is a silly size for the labels; they need to be 64 points, because all good sizes are a power of 2.

And now it looks like your designer and IT director are having a fight about the right size. A few punches later, they decide to split the difference and go for a number in the middle: 72. So please go ahead and make the labels all 72 points high.

Bored yet? You ought to be. And yet this is the kind of pixel-pushing it's easy to fall into,

particularly if your app is being designed by committee.

Auto Layout has a solution, and it's called *metrics*. All these calls to **constraintsWithVisualFormat()** have been sent **nil** for their metrics parameter, but that's about to change. You see, you can give VFL a set of sizes with names, then use those sizes in the VFL rather than hard-coding numbers. For example, we wanted our label height to be 88, so we could create a metrics dictionary like this:

```
let metrics = ["labelHeight": 88]
```

Then, whenever we had previously written **==88**, we can now just write **labelHeight**. So, change your current vertical constraints to be this:

```
view.addConstraints(NSLayoutConstraint.constraintsWithVisualFormat("V
:|[label1(labelHeight)]-[label2(labelHeight)]-[label3(labelHeight)]-
[label4(labelHeight)]-[label5(labelHeight)]->=10-|", options: [],
metrics: ["labelHeight": 88], views: viewsDictionary))
```

So when your designer / manager / inner-pedant decides that 88 points is wrong and you want some other number, you can change it change in one place to have everything update.

Before we're done, we're going to make one more change that makes the whole user interface much better, because right now it's still imperfect. To be more specific, we're forcing all labels to be a particular height, then adding constraints to the top and bottom. This still works fine in portrait, but in landscape you're unlikely to have enough room to satisfy all the constraints.

With our current configuration, you'll see this message when the app is rotated to landscape: "Unable to simultaneously satisfy constraints." This means your constraints simply don't work given how much screen space there is, and that's where *priority* comes in. You can give any layout constraint a priority, and Auto Layout will do its best to make it work.

Constraint priority is a value between 1 and 1000, where 1000 means "this is absolutely required" and anything less is optional. By default, all constraints you have are priority 1000, so Auto Layout will fail to find a solution in our current layout. But if we make the height

optional – even as high as priority 999 – it means Auto Layout can find a solution to our layout: shrink the labels to make them fit.

It's important to understand that Auto Layout doesn't just discard rules it can't meet – it still does its best to meet them. So in our case, if we make our 88-point height optional, Auto Layout might make them 78 or some other number. That is, it will still do its best to make them as close to 88 as possible. TL;DR: constraints are evaluated from highest priority down to lowest, but all are taken into account.

So, we're going to make the label height have priority 999 (i.e., very important, but not required). But we're also going to make one other change, which is to tell Auto Layout that we want all the labels to have the same height. This is important, because if all of them have optional heights using **labelHeight**, Auto Layout might solve the layout by shrinking one label and making another 88.

From its point of view it has at least managed to make some of the labels 88, so it's probably quite pleased with itself, but it makes our user interface look uneven. So, we're going to make the first label use **labelHeight** at a priority of 999, then have the other labels adopt the same height as the first label. Here's the new VFL line:

```
"V:|[label1(labelHeight@999)]-[label2(label1)]-[label3(label1)]-
[label4(label1)]-[label5(label1)]->=10-|"
```

It's the **@999** that assigns priority to a given constraint, and using **(label1)** for the sizes of the other labels is what tells Auto Layout to make them the same height.

That's it: your Auto Layout configuration is complete, and the app can now be run safely in portrait and landscape.

Wrap up

There are two types of iOS developer in the world: those who use Auto Layout, and fools. It has bit of a steep learning curve (and we didn't even use the hard way of adding constraints!), but it's an extremely expressive way of creating great layouts that adapt themselves automatically to whatever device they find themselves running on – now and in the future.

Most people recommend you do as much as you can inside Interface Builder, and with good reason – you can drag lines about until you're happy, you get an instant preview of how it all looks, and it will warn you if there's a problem (and help you fix it.) But, as you've seen, creating constraints in code is remarkably easy thanks to the Visual Format language, so you might find yourself mixing the two to get the best results.

Project 7
Whitehouse Petitions

Make an app to parse Whitehouse petitions using JSON and a tab bar.

Setting up

This project will take a data feed from a website and parse it into useful information for users. As per usual, this is just a way of teaching you some new iOS development techniques, but let's face it – you already have two apps and two games under your belt, so you're starting to build up a pretty good library of work!

This time, you'll be learning about **UITabBarController**, **NSData**, and more. You'll also be using a data format called JSON, which is a popular way to send and receive data online. It's not easy to find interesting JSON feeds that are freely available, but the option we'll be going for is the "We the people" Whitehouse petitions in the US, where Americans can submit requests for action, and others can vote on it.

Some are entirely frivolous ("We want the US to build a Death Star"), but it has good, clean JSON that's open for everyone to read, which makes it perfect. Lots to learn, and lots to do, so let's get started: create a new project in Xcode by choosing the Master-Detail Application template. Now name it Project7, set its target to be iPad, and save it somewhere.

Creating the basic UI

The Master-Detail application template gives us a lot of stuff we don't need, but rather than delete it we're just going to modify it as needed. For the user interface we need to make only a handful of changes, so please open Main.storyboard in Interface Builder.

In your storyboard, you'll see there are two navigation controllers: one at the top that connects to a table view on its right, and another at the bottom that connects to a view saying "Detail view content goes here" on its right. Choose the top one, then choose Editor > Embed In > Tab Bar Controller. Like **UINavigationController**, **UITabBarController** is a common element in iOS user interface design. A tab bar is that strip of icons across the bottom that shows various screens, and it appears in the App Store, in the music app, in the phone app, and more.
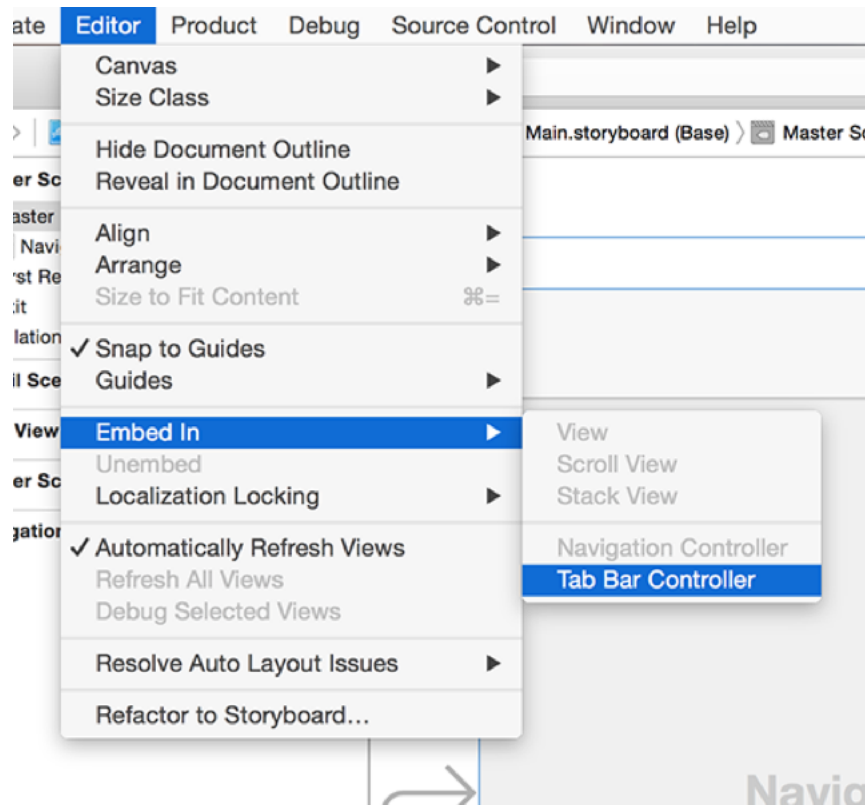
Interface Builder can embed a view controller directly inside a tab bar controller with one menu click.

Behind the scenes, **UITabBarController** manages an array of view controllers that the user can choose between. You can do often do most of the work inside Interface Builder, but not in this project. We're going to use one tab to show recent petitions, and another to show

popular petitions, which is the same thing really – all that's changing is the data source.

Doing everything inside the storyboard would mean duplicating our view controllers, which is A Bad Idea, so instead we're just going to design one of them in the storyboard then create a duplicate of it using code.

Now that our navigation controller is inside a tab bar controller, it will have acquired a gray strip along its bottom in Interface Builder. If you click that now, it will select a new type of object called a **UITabBarItem**, which is the icon and text used to represent a view controller in the tab bar. In the attributes inspector (Alt+Cmd+4) change System Item from "Custom" to "Most Recent".

One important thing about **UITabBarItem** is that when you set its system item, it assigns both an icon and some text for the title of the tab. If you try to change the text to your own text, the icon will be removed and you need to provide your own. This is because Apple has trained users to associate certain icons with certain information, and they don't want you using those icons incorrectly!

Select the navigation controller itself (just click where it says Navigation Controller in big letters in the center of the view controller), then press Alt+Cmd+3 to select the identity inspector. We haven't been here before, because it's not used that frequently. However, here I want you to type "NavController" in the text box to the right of where it says, "Storyboard ID". We'll be needing that soon enough!

In the picture below you can see how the identity inspector should look when configured for your navigation controller. You'll be using this inspector in later projects to give views a custom class by changing the first of these four text boxes.
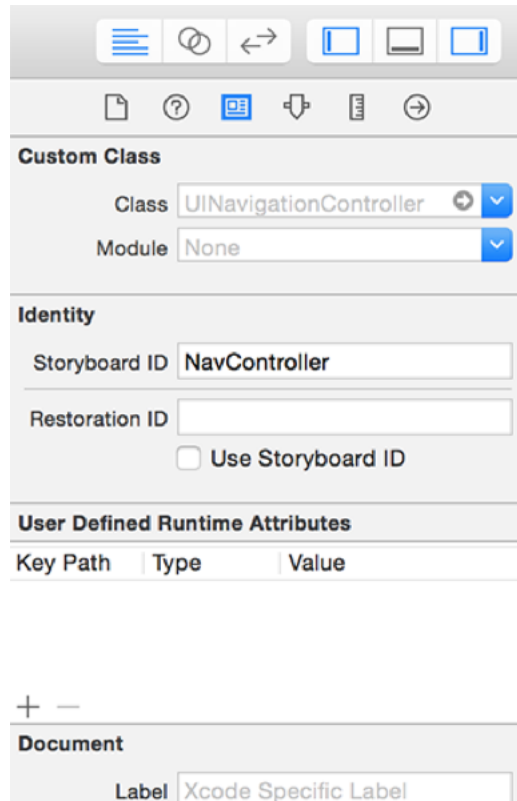
Use the identity inspector to configure your navigation controller with a storyboard identifier.

The other change we're going to make to the storyboard is in the table view controller. Click once where it says "Title" (just below where it says Prototype Cells), and in the attributes inspector you'll some options for the cell. You'll know when you have it right because it will say "Table View Cell" at the top of the attributes inspector. Change the style from "Basic" to "Subtitle", which adds an extra line of text underneath the title.
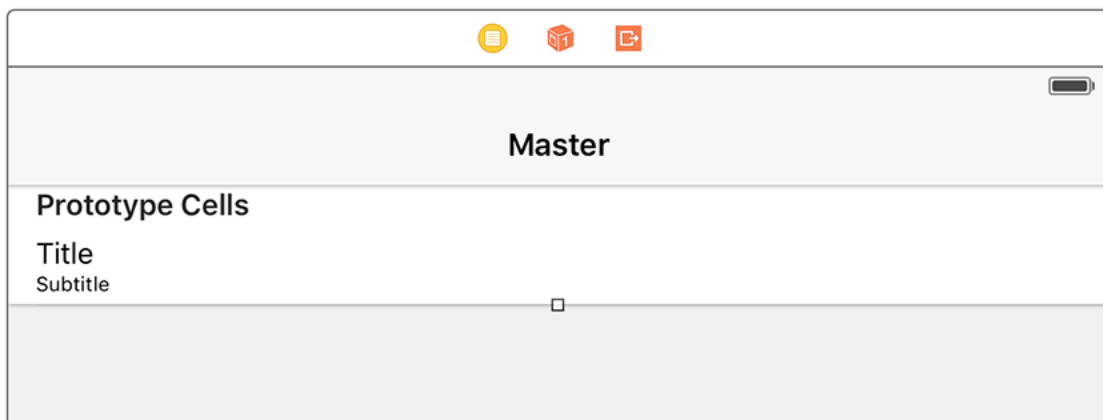
Changing the table view cell style from Basic to Subtitle adds a second line of text we can customize.

We're done with Interface Builder, so please open the file MasterViewController.swift so we can make a few basic changes. First, delete the **insertNewObject()** method entirely, delete everything from **viewDidLoad()** except the call to **super.viewDidLoad()**, delete the table view's **commitEditingStyle** and **canEditRowAtIndexPath** methods, and finally delete the **as! NSDate** text from the **prepareForSegue()** and **cellForRowAtIndexPath** methods – not the whole line, just the bit that says **as! NSDate**.

Step one is now complete: we have a basic user interface in place, and we've cleaned the unwanted cruft out of the Xcode template. Now for some real code…

Parsing JSON

JSON – short for JavaScript Object Notation – is a way of describing data. It's not the easiest to read yourself, but it's compact and easy to parse for computers, which makes it popular online where bandwidth is at a premium.

In project 6 you learned about using dictionaries with Auto Layout, and in this project we're going to use dictionaries more extensively. What's more, we're going to put dictionaries inside an array to make an array of dictionaries, which should keep our data in order.

You declare a dictionary using square brackets, then entering its key type, a colon, and its value type. For example, a dictionary that used strings for its keys and **UILabel**s for its values would be declared like this:

```
var labels = [String: UILabel]()
```

And as you'll recall, you declare arrays just by putting the data type in brackets, like this:

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

Putting these two together, we want to make an array of dictionaries, with each dictionary holding a string for its key and another string for its value. So, it looks like this:

```
var objects = [[String: String]]()
```

Put that in place of the current **objects** definition at the top of MasterViewController.swift – it holds **AnyObject** right now, which won't do the job.

It's now time to parse some JSON, which means to process it and examine its contents. This isn't easy in Swift, so a number of helper libraries have appeared that do a lot of the heavy lifting for you. We're going to use one of them now: download the files for this project from **GitHub** then look for a file called SwiftyJSON.swift. Add that your project, making sure "Copy

items if needed" is checked.

SwiftyJSON lets us read through JSON in an extremely intuitive way: you can effectively treat almost everything as a dictionary, so if you know there's a value called "information" that contains another value called "name", which in turns contains another value called "firstName", you can use **json["information"]["name"]["firstName"]** to get the data, then ask for it as a Swift value by using the **string** property.

Before we do the parsing, here is a tiny slice of the actual JSON you'll be receiving:

```
{
  "metadata":{
    "responseInfo":{
      "status":200,
      "developerMessage":"OK",
    }
  },
  "results":[
    {
      "title":"Legal immigrants should get freedom before
undocumented immigrants – moral, just and fair",

      "body":"I am petitioning President Obama's Administration to
take a humane view of the plight of legal immigrants. Specifically,
legal immigrants in Employment Based (EB) category. I believe, such
immigrants were short changed in the recently announced reforms via
Executive Action (EA), which was otherwise long due and a welcome
announcement.",
      "issues":[
        {
          "id":"28",
          "name":"Human Rights"
        },
        {
          "id":"29",
```

```
        "name":"Immigration"

      }

    ],

    "signatureThreshold":100000,

    "signatureCount":267,

    "signaturesNeeded":99733,

  },

  {

    "title":"National database for police shootings.",

    "body":"There is no reliable national data on how many people
are shot by police officers each year. In signing this petition, I am
urging the President to bring an end to this absence of visibility by
creating a federally controlled, publicly accessible database of
officer-involved shootings.",

    "issues":[

      {

        "id":"28",

        "name":"Human Rights"

      }

    ],

    "signatureThreshold":100000,

    "signatureCount":17453,

    "signaturesNeeded":82547,

  }

  ]

}
```

You'll actually be getting between 2000-3000 lines of that stuff, all containing petitions from US citizens about all sorts of political things. It doesn't really matter (to us) what the petitions are, we just care about the data structure. In particular:

1. There's a metadata value, which contains a **responseInfo** value, which in turn contains a

status value. Status 200 is what internet developers use for "everything is OK."

2. There's a results value, which contains a series of petitions.

3. Each petition contains a title, a body, some issues it relates to, plus some signature information.

4. JSON has strings and integers too. Notice how the strings are all wrapped in quotes, whereas the integers aren't.

Now that you have a basic understanding of the JSON we'll be working with, it's time to write some code. We're going to update the **viewDidLoad()** method for **MasterViewController** so that it downloads the data from the Whitehouse petitions server, converts it to a SwiftyJSON object, and checks that the status value is equal to 200.

To make this happen, we're going to use **NSURL** alongside a new NS class called **NSData**. This is a class designed to hold data in any form, which might be a string, it might be an image, or it might be something else entirely. You already saw that **NSString** can be created using **contentsOfFile** to load data from disk. Well, **NSData** (and **NSString**) can be created using **contentsOfURL**, which downloads data from a URL (specified using **NSURL**) and makes it available to you.

Here's the new **viewDidLoad** method:

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


  let urlString = "https://api.whitehouse.gov/v1/petitions.json?
limit=100"


  if let url = NSURL(string: urlString) {
    if let data = try? NSData(contentsOfURL: url, options: []) {
      let json = JSON(data: data)


      if json["metadata"]["responseInfo"]["status"].intValue == 200 {
        // we're OK to parse!
      }
```

```
        }

    }

}
```

Let's focus on the new stuff:

 • **urlString** points to the Whitehouse.gov server, accessing the petitions system.
 • We use if/let to make sure the **NSURL** is valid, rather than force unwrapping it. Later on you can return to this to add more URLs, so it's good play it safe.
 • We create a new **NSData** object using its **contentsOfURL** method. This returns the content from an **NSURL**, which is all we need – hence why we're using **[]** for options. This might throw an error (i.e., if the internet connection was down), so we also need to use **try?**.
 • If the **NSData** object was created successfully, we create a new JSON object from it. This is a SwiftyJSON structure.
 • Finally, we have our first bit of JSON parsing: if there is a "metadata" value and it contains a "responseInfo" value that contains a "status" value, return it as an integer, then compare it to 200.
 • The "we're OK to parse!" line starts with **//**, which begins a comment line in Swift. Comment lines are ignored by the compiler; we write them as notes to ourselves.

The reason SwiftyJSON is so good at JSON parsing is because it has optionality built into its core. If any of "metadata", "responseInfo" or "status" don't exist, this call will return 0 for the status – we don't need to check them all individually. If you're reading a string value, SwiftyJSON will return either the string it found, or if it didn't exist then an empty string.

This code isn't perfect, in fact far from it. In fact, by downloading data from the internet in **viewDidLoad()** our app will lock up until all the data has been transferred. There are solutions to this, but to avoid complexity they won't be covered until project 9.

For now, we want to focus on our JSON parsing. We already have an **objects** array that is ready to accept dictionaries of data. We want to parse that JSON into dictionaries, with each dictionary having three values: the title of the petition, its body text, and how many signatures it has. Once that's done, we need to tell our table view to reload itself.

Are you ready? Because this code is shockingly simple given how much work it's doing:

```swift
func parseJSON(json: JSON) {

    for result in json["results"].arrayValue {

        let title = result["title"].stringValue

        let body = result["body"].stringValue

        let sigs = result["signatureCount"].stringValue

        let obj = ["title": title, "body": body, "sigs": sigs]

        objects.append(obj)

    }


    tableView.reloadData()

}
```

Place that method just underneath **viewDidLoad()** method, then replace the existing **// we're OK to parse!** line in **viewDidLoad()** with this:

```swift
parseJSON(json)
```

The **parseJSON()** method reads the "results" array from the JSON object it gets passed. If you look back at the JSON snippet I showed you, that results array contains all the petitions ready to read. When you use **arrayValue** with SwiftyJSON, you either get back an array of objects or an empty array, so we use the return value in our loop.

For each result in the results array, we read out three values: its title, its body, and its signature count, with all three of them being requested as strings. The signature count is actually a number when it comes in the JSON, but SwiftyJSON converts it for us so we can put it inside our dictionary where all the keys and values are strings.

Each time we're accessing an item in our **result** value using **stringValue**, we will either get its value back or an empty string. Regardless, we'll have *something*, so we construct a new dictionary from all three values then use **objects.append()** to place the new dictionary into our array.

Once all the results have been parsed, we tell the table view to reload, and the code is

complete.

Well, the could *would* have been complete if the Whitehouse actually used good HTTPS. Even though the URL we're hitting starts with **https://api.whitehouse.gov**, the form of HTTPS is so weak at the time of writing that iOS 9 doesn't trust it. So, if you try running this code you'll get an error: "NSURLSession/NSURLConnection HTTP load failed (kCFStreamErrorDomainSSL, -9802)".

The solution here is to have the Whitehouse upgrade to more secure HTTPS. Failing that, we can ask iOS to allow an exception for this insecure domain by customizing its App Transport Security Settings. This is an annoyance, and I wouldn't show it to you unless it was strictly necessary, but I'm afraid there's no other choice.

So: look in the project navigator for a file called Info.plist. Right-click on it, and choose Open As > Source Code. It should end like this:

```
</dict>
</plist>
```

Just before that, I'd like you to paste this:

```
<key>NSAppTransportSecurity</key>
<dict>
  <key>NSExceptionDomains</key>
  <dict>
    <key>whitehouse.gov</key>
    <dict>
      <key>NSIncludesSubdomains</key>
      <true/>
      <key>NSThirdPartyExceptionAllowsInsecureHTTPLoads</key>
      <true/>
    </dict>
```

```
    </dict>
</dict>
```

That adds an App Transport Security exception that means iOS won't refuse to work with the Whitehouse's weak certification.
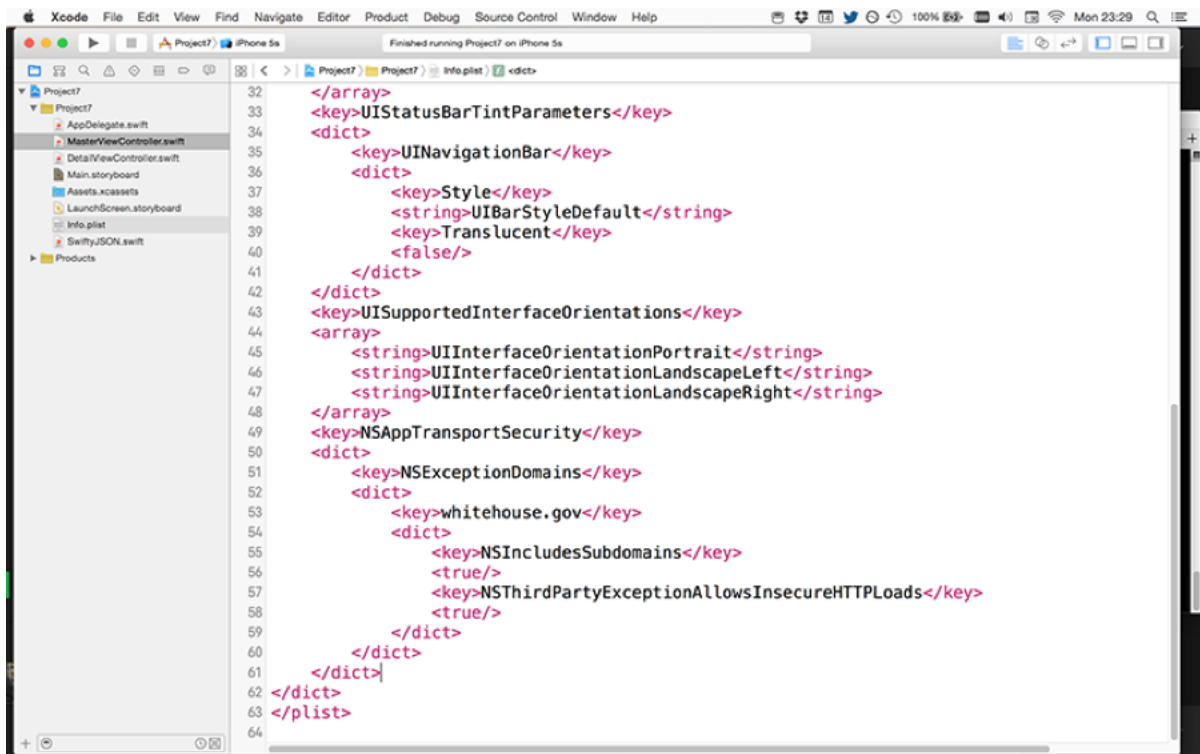
Because the Whitehouse APIs don't use sufficient security, you need to add an App Transport Security exception.

You can run the program now, although it's not optimal: you'll see some strangely formatted text in the rows. This is because the built-in Xcode project has the following in the table view's **cellForRowAtIndexPath** method inside MasterViewController.swift:

```
cell.textLabel!.text = object.description
```

The text label for the cell expects a string, not a dictionary, but Xcode's default template code uses **object.description** to ask the object to describe itself in string format. For

dictionaries, this prints out a nicely formatted key/value layout that shows you what the dictionary contains.

We want to modify this so that it prints out the **title** value of our dictionary, but we also want to use the subtitle text label that got added when we changed the cell type from "Basic" to "Subtitle" in the storyboard. To do that, change the current line (the one above) to this:

```
cell.textLabel!.text = object["title"]
cell.detailTextLabel!.text = object["body"]
```

We set the **title**, **body** and **sigs** keys in the dictionary, and now we can read them out to configure our cell correctly.

If you run the app now, you'll see things are starting to come together quite nicely – every table row now shows the petition title, and beneath it shows the first few words of the petition's body. The subtitle automatically shows "…" at the end when there isn't enough room for all the text, but it's enough to give the user a flavor of what's going on.

Rendering a petition

After all the JSON parsing, it's time for something easy: we need to update the **DetailViewController** class so that it can draw the petition content in a nice way. The easiest way for rendering complex content from the web is nearly always to use a **WKWebView**, and we're going to use the same technique from project 4 to modify **DetailViewController** so it has a web view.

Replace *all* the **DetailViewController** code with this:

```
import UIKit
import WebKit


class DetailViewController: UIViewController {
  var webView: WKWebView!
  var detailItem: [String: String]!


  override func loadView() {
    webView = WKWebView()
    view = webView
  }


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

This is almost identical to the code from project 4, but you'll notice I've added a **detailItem** property that will contains our dictionary of data.

That was the easy bit. The hard bit is that we can't just drop the petition text into the web view, because it will probably look tiny. Instead, we need to wrap it in some HTML, which is a

whole other language with its own rules and its own complexities.

Now, this series isn't called "Hacking with HTML," so I don't intend to go into much detail here. However, I will say that the HTML we're going to use tells iOS that the page fits mobile devices, and that we want the font size to be 150% of the standard font size. All that HTML will be combined with the **body** value from our dictionary, then sent to the web view.

Place this in **viewDidLoad()**, directly beneath the call to **super.viewDidLoad()**:

```
guard detailItem != nil else { return }


if let body = detailItem["body"] {

  var html = "<html>"

  html += "<head>"

  html += "<meta name=\"viewport\" content=\"width=device-width,
initial-scale=1\">"

  html += "<style> body { font-size: 150%; } </style>"

  html += "</head>"

  html += "<body>"

  html += body

  html += "</body>"

  html += "</html>"

  webView.loadHTMLString(html, baseURL: nil)

}
```

There's a new Swift statement in there that is important: **guard**. This is used to create an "early return," which means you set your code up so that it exits immediately if critical data is missing. In our case, we don't want this code to run if **detailItem** isn't set, so guard will run **return** is **detailItem** is set to **nil**.

I've tried to make the HTML as clear as possible, but if you don't care for HTML don't worry about it. What matters is that there's a Swift string called **html** that contains everything needed to show the page, and that's passed in to the web view's **loadHTMLString()** method

so that it gets loaded. This is different to the way we were loading HTML before, because we aren't using a website here, just some custom HTML.

That's it for the detail view controller, it really is that simple. Go ahead and run the project now by pressing Cmd+R or clicking play, then tap on a row to see more detail about each petition.

Finishing touches

Before this project is finished, we're going to make two changes. First, we're going to add another tab to the **UITabBarController** that will show popular petitions, and second we're going to make our **NSData** loading code a little more resilient by adding error messages.

As I said previously, we can't really put the second tab into our storyboard because both tabs will host a **MasterViewController** and doing so would require me to duplicate the view controllers in the storyboard. You can do that if you want, but please don't – it's a maintenance nightmare!

Instead, we're going to leave our current storyboard configuration alone, then create the second view controller using code. This isn't something you've done before, but it's not hard and we already took the first step, as you'll see.

Open the file AppDelegate.swift. This has been in all our projects so far, but it's not one we've had to work with until now. Look for the **didFinishLaunching** method, which should be at the top of the file. This gets called by iOS when the app is ready to be run, and we're going to hijack it to insert a second **MasterViewController** into our tab bar.

It should already have some default Apple code in there, but we're going to add some more just before the **return true** line:

```
let tabBarController = splitViewController.viewControllers[0] as!
UITabBarController


let storyboard = UIStoryboard(name: "Main", bundle: nil)

let vc =
storyboard.instantiateViewControllerWithIdentifier("NavController")
as! UINavigationController

vc.tabBarItem = UITabBarItem(tabBarSystemItem: .TopRated, tag: 1)


tabBarController.viewControllers?.append(vc)
```

Every line of that is new, so let's dig in deeper:

• Our storyboard automatically creates a window in which all our view controllers are shown. This window needs to know what its initial view controller is, and that gets set to its **rootViewController** property. This is all handled by our storyboard.

• In the Master-Detail Application template, the root view controller is the **UISplitViewController**, which itself has a property called **viewControllers**. This stores two items: the first is the view controller on the left (our table view) and the second is the view controller on the right (our detail view).

• We need to create a new **MasterViewController** by hand, which first means getting a reference to our Main.storyboard file. This is done using the **UIStoryboard** class, as shown. You don't need to provide a bundle, because **nil** means "use the current app bundle."

• We create our view controller using the extraordinarily long method **instantiateViewControllerWithIdentifier()**, passing in the storyboard ID of the view controller we want. Earlier we set our navigation controller to have the storyboard ID of "NavController", so we use pass that in and typecast the result to be a **UINavigationController**.

• We create a new **UITabBarItem** object for the new view controller, giving it the "Top Rated" icon and the tag 1. That tag is important, but not just yet.

• We add the new view controller to our tab bar controller's **viewControllers** array, which will cause it to appear in the tab bar.

So, the code creates a duplicate **MasterViewController** wrapped inside a navigation controller, gives it a new tab bar item to distinguish it from the existing tab, then adds it to the list of visible tabs. This lets us use the same class for both tabs without having to duplicate things in the storyboard.

The reason we gave a tag of 1 to the new **UITabBarItem** is because it's an easy way to identify it. Remember, both tabs contain a **MasterViewController**, which means the same code is executed. Right now that means both will download the same JSON feed, which makes having two tabs pointless. But if you modify **urlString** in MasterViewController's **viewDidLoad()** method to this, it will work much better:

```
let urlString: String


if navigationController?.tabBarItem.tag == 0 {

  urlString = "https://api.whitehouse.gov/v1/petitions.json?limit=100"

} else {
```

```
    urlString = "https://api.whitehouse.gov/v1/petitions.json?
signatureCountFloor=10000&limit=100"

}
```

That adjusts the code so that the first instance of **MasterViewController** loads the original JSON, and the second loads only petitions that have at least 10,000 signatures.

The project is almost done, but we're going to make one last change. Our current loading code isn't very resilient: we have lots of **if** statements checking that things are working correctly, but no **else** statements showing an error message if there's a problem. This is easily fixed by adding a new **showError()** method that creates a **UIAlertController** showing a general failure message:

```
func showError() {

   let ac = UIAlertController(title: "Loading error", message: "There
was a problem loading the feed; please check your connection and try
again.", preferredStyle: .Alert)

   ac.addAction(UIAlertAction(title: "OK", style: .Default, handler:
nil))

   presentViewController(ac, animated: true, completion: nil)

}
```

You can now adjust the JSON downloading and parsing code to call this error method everywhere a condition fails, like this:

```
if let url = NSURL(string: urlString) {

  if let data = try? NSData(contentsOfURL: url, options: []) {

    let json = JSON(data: data)


    if json["metadata"]["responseInfo"]["status"].intValue == 200 {

      parseJSON(json)

    } else {
```

```
        showError()
    }

  } else {
    showError()
  }

} else {
  showError()
}
```

Now that error messages are shown when the app hits problems, we're done!

Wrap up

As your Swift skill increases, I hope you're starting to feel the balance of these projects move away from explaining the basics and toward presenting and dissecting code. Working with JSON is something you're going to be doing time and time again in your Swift career, and you've cracked it in about an hour of work – while also learning about **NSData**, **UITabBarController**, and more. Not bad!

If you're looking to extend this project some more, you might like to look at the original API documentation – it's at **https://petitions.whitehouse.gov/developers** and contains lots of options. If you add more view controllers to the tab bar, you'll find you can add up to five before you start seeing a "More" button. This More tab hides all the view controllers that don't fit into the tab bar, and it's handled for you automatically by iOS.

Project 8

7 Swifty Words

Build a word-guessing game and master strings once and for all.

Setting up

This is the final game you'll be making with UIKit; every game after this one will use Apple's SpriteKit library for high-performance 2D drawing. To make this last UIKit effort count, we're going to have a fairly complicated user interface so you can go out with a bang. We're also going to mix in some great new Swift techniques, including property observers, searching through arrays, modulo, array enumeration, ranges and more!

Of course, you're probably wondering what kind of game we're going to make, and I have some bad news for you: it's another word game. But there's good news too: it's a pretty darn awesome word game, based on the popular indie game 7 Little Words. This will also be our first game exclusively targeting iPad, and you'll soon see why – we're using a lot of space in our user interface!

So, go ahead and create a new Single View Application project in Xcode, this time selecting iPad for your device, then save it somewhere. Now go to the project editor and deselect Portrait and Upside Down orientations.

What's that? You don't know where the project editor is? I'm sure I told you to remember where the project editor was! OK, here's how to find it, one last time, quoted from project 6:

Press Cmd+1 to show the project navigator on the left of your Xcode window, select your project (it's the first item in the pane), then to the right of where you just clicked will appear another pane showing "PROJECT" and "TARGETS", along with some more information in the center. The left pane can be hidden by clicking the disclosure button in the top-left of the project editor, but hiding it will only make things harder to find, so please make sure it's visible!This view is called the project editor, and contains a huge number of options that affect the way your app works. You'll be using this a lot in the future, so remember how to get here! Select Project 2 under TARGETS, then choose the General tab, and scroll down until you see four checkboxes called Device Orientation. You can select only the ones you want to support.

Buttons... buttons everywhere.

Our user interface for this game is going to have two large **UILabel**s, one small **UILabel**, one large **UITextField**, twenty (count 'em!) big **UIButton**s, then two small **UIButton**s. This is probably the most complicated user interface we're going to make in this entire series, so don't worry if it takes you 20 minutes or so to put together – the end result is definitely worth it!

The picture below shows how your finished layout should look if you've followed all the instructions. If you're seeing something slightly different, that's OK. If you're seeing something *very* different, you should probably try again!
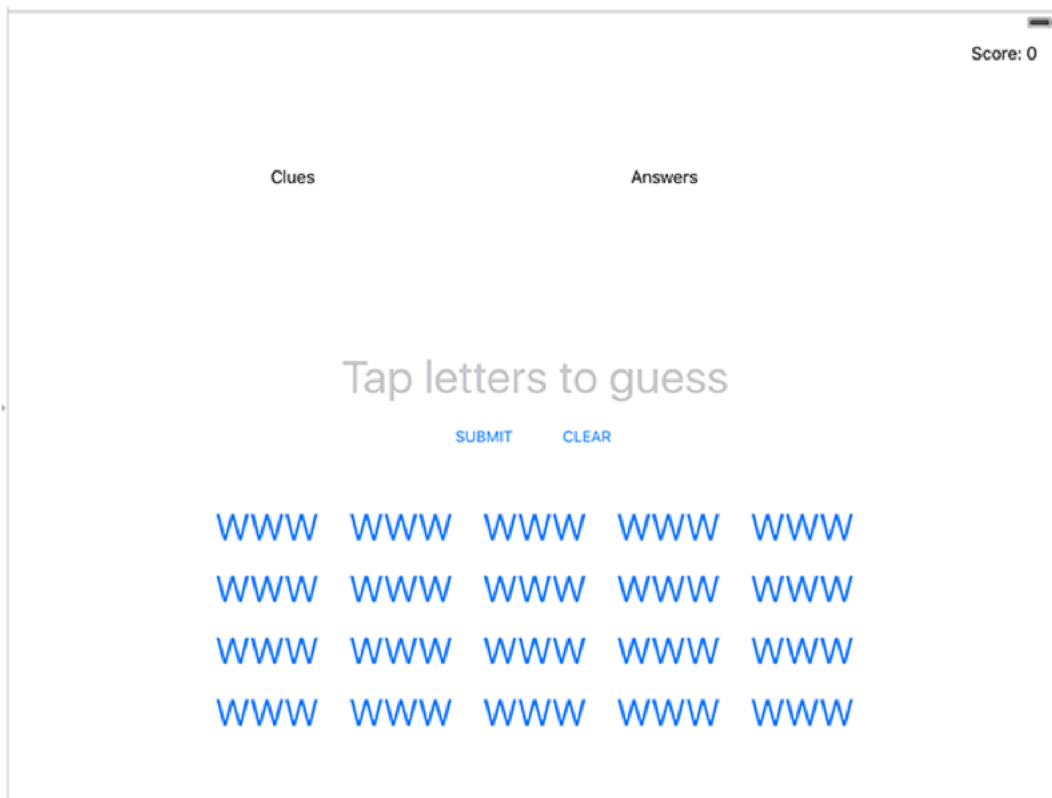
Your finished layout should look like this.

Our game is designed for iPads, and specifically for iPads in landscape orientation. Using a square Interface Builder canvas is great for when you want to support a variety of devices where things resize, but here we want one single design with lots of things on. So, we're going to make the canvas the exact right size: go to the file inspector (Alt+Cmd+1) and deselect the checkbox that says, "Use Size Classes".

You'll be asked what size class data you want to keep, so select iPad then choose Disable Size Classes. Now select your view controller, go to the attributes inspector, then change Orientation from "Inferred" to "Landscape". This gives us a canvas that shaped and sized exactly correctly for an iPad screen, so we're all set to go!

Let's start with the twenty big buttons, because you need to follow my instructions carefully and once these are placed you'll be able to see the big plan.

What you need to do is place twenty **UIButton**s in a grid that's five across and four down. The top-left button should be at X:200 and Y:470. All buttons should be 120 wide and 60 high; each column should be 130 points apart, and each row should be 60 points apart. That ought to be enough for you to make the entire grid, but for the sake of clarity, this means that:

 • The second button on the top row should be at X:330 Y:470.
 • The third button on the top row should be at X:460 Y:470. (and so on)
 • The first button on the second row should be at X:200 Y:530.
 •
The second button on the second row should be at X:330 Y:530. (and so on)

Once you've placed all the buttons, click and drag over them so they are all selected, then tap your left cursor key eight times to move every button eight points to the left. On the eighth tap, you should see a long, blue, vertical line appear in the center of the button group, which is telling you the buttons are centered horizontally, so these buttons are placed correctly.

With the buttons still selected, go to the attributes inspector and change Type to be "Custom" and Tag to be 1001. The first one disables an Apple animation that will otherwise cause problems later, and the second change sets the tag for all the buttons simultaneously. You should also click the small T button next to the button font, and in the popover that appears make sure the size is set to 36.

When designing this, I gave these buttons the text WWW because that's the largest string they'll need to hold. You should also give them a text color; I chose a shade of blue similar to the iOS default.

Now create two more buttons, both 75 wide and 44 high. Place the first at X:425 Y:390 and give it the title SUBMIT, and place the second at X:525 Y:390 and give it the title CLEAR.

That's all the buttons we'll need for the game.

Place a text field and make it 535 wide by 80 high, then position it at X: 245 Y: 315. You'll find that you can't resize the text field's height by default, and that's because it has a rounded rectangle border around it that must be an exact size. To change the height you must make the textfield bordlerless: in the identity inspector, choose the first of the four options next to Border Style, then you can adjust the height freely.

Give this text field the placeholder text "Tap letters to guess", then give it a nice and big font size – 44 points ought to do. Finally, make its text aligned to the center rather than the left, so everything lines up neatly.

Now create two labels. Make the first one 400 wide by 280 high, with position X:255 Y:20, then give it the text "Clues". Make the second one 165 wide by 280 high, and position it at X: 605 Y:20, then give it the text "Answers". Both of these should be given font size 24, but where it says "Number of lines" set the value to be 0 – that means "let this text go over as many lines as it needs."

Finally, create one last label of width 170 and height 40, at X:830 and Y:20. Set this to have text alignment right and the text "Score: 0".

That's the layout complete. If you've played any games like 7 Little Words before, you'll already know exactly how the user interface functions. If not, we'll show seven clues in the label marked "Clues", and each of those clues can be spelled by tapping the letters in the buttons. When a user has spelled the word they want, they click Submit to try it out, and if the answer is correct they'll see it in the Answers label. Otherwise, that answers label just shows the number of letters in the correct answer.

Before you exit Interface Builder, switch to the assistant editor and create four outlets: one for the clue label (call it **cluesLabel**), one for the Answers label (call it **answersLabel**), one for the text field (call it **currentAnswer**) and one for the score (call it **scoreLabel**). Please also create two actions: one from the submit button (call it **submitTapped()**) and one from the clear button (call it **clearTapped()**).

That's it! That's the most complicated storyboard you'll make in any project in this entire series. Fortunately, from here on the rest is all coding and lots of fun, so let's get onto the best bit…

Loading a level

This game asks players to spell seven words out of various letter groups, and each word comes with a clue for them to guess. It's important that the total number of letter groups adds up to 20, as that's how many buttons you have. I created the first level for you, and it looks like this:

```
HA|UNT|ED: Ghosts in residence

LE|PRO|SY: A Biblical skin disease

TW|ITT|ER: Short but sweet online chirping

OLI|VER: Has a Dickensian twist

ELI|ZAB|ETH: Head of state, British style

SA|FA|RI: The zoological web

POR|TL|AND: Hipster heartland
```

As you can see, I've used the pipe symbol to split up my letter groups, meaning that one button will have "HA", another "UNT", and another "ED". There's then a colon and a space, followed by a simple clue. This level is in the files for this project you should download from **GitHub**. You should copy level1.txt into your Xcode project as you have done before.

Our first task will be to load the level and configure all the buttons to show a letter group. We're going to need three arrays to handle this: one to store all the buttons, one to store the buttons that are currently being used to spell an answer, and one for all the possible solutions. Further, we need two integers: one to hold the player's score, which will start at 0 but obviously change during play, and one to hold the current level.

So, declare these properties just below the current **@IBOutlet**s from Interface Builder:

```swift
var letterButtons = [UIButton]()

var activatedButtons = [UIButton]()

var solutions = [String]()


var score = 0
```

```
var level = 1
```

Now, you'll notice we don't have any **@IBOutlet** references to any of our buttons, and that's entirely intentional: it wouldn't be very smart to create an **@IBOutlet** for every button. Interface Builder does have a solution to this, called *Outlet Collections*, which are effectively an IBOutlet array, but even that solution requires you to Ctrl-drag from every button and quite frankly I don't think you have the patience after spending so much time in Interface Builder!

As a result, we're going to take a simple shortcut. And this shortcut will also deal with calling methods when any of the buttons are tapped, so all in all it's a clean and easy solution. The shortcut is this: all our buttons have the tag 1001, so we can loop through all the views inside our view controller, and modify them only if they have tag 1001. Add this code to your **viewDidLoad()** method beneath the call to **super**:

```
for subview in view.subviews where subview.tag == 1001 {

  let btn = subview as! UIButton

  letterButtons.append(btn)

  btn.addTarget(self, action: "letterTapped:",
forControlEvents: .TouchUpInside)

}
```

As you can see, **view.subviews** is an array containing all the **UIView**s that are currently placed in our view controller, which is all the buttons and labels, plus that text field. I've used a more enhanced version of a regular **for** loop that adds a **where** condition so that the only items inside the loop are subviews with that tag. If we find a view with tag 1001, we typecast it as a **UIButton** then append it to our buttons array.

We also take this opportunity to use a new method, called **addTarget()**. This is the code version of Ctrl-dragging in a storyboard and it lets us attach a method to the button click. You should remember **.TouchUpInside** from all the button actions you have made, because that's the event that means the button was tapped.

But there is a curiosity: the action is **letterTapped:**, with a colon on the end. That's not a typo – it's meaningful. And what it means is that it will call the **letterTapped()** method that takes

exactly one parameter. Yes, I know it's a bit strange, but that colon really does mean "it takes one parameter."

The parameter that gets sent will be the button that is clicked, which is perfect for us because we can read the letter group on the button and use it to spell words. But more on that later – for now, we want to finish loading the level.

We're going to isolate level loading into a single level, called **loadLevel()**. This needs to do two things: load and parse our level text file in the format I showed you earlier, then randomly assign letter groups to buttons. In project 5 you already learned how to create **Strings** using **contentsOfFile** to load files from disk, and we'll be using that to load our level. In that same project you learned how to use **componentsSeparatedByString()** to split up a string into an array, and we'll use that too.

We'll also need to use the array shuffling code from the GameplayKit framework that we've used before. But: there are some new things to learn, honest! First, we'll be using the **enumerate()** method to loop over an array. We haven't used this before, but it's helpful because it passes you each object from an array as part of your loop, as well as that object's position in the array. You're also going to meet the **characters.count** property of strings, which returns how many letters are in that string.

There's also a new string method to learn, called **stringByReplacingOccurrencesOfString()**. This lets you specify two parameters, and replaces all instances of the first parameter with the second parameter. We'll be using this to convert "HA|UNT|ED" into HAUNTED so we have a list of all our solutions.

Before I show you the code, watch out for how I use the method's three variables: **clueString** will store all the level's clues, **solutionString** will store how many letters each answer is (in the same position as the clues), and **letterBits** is an array to store all letter groups: HA, UNT, ED, and so on.

Here's the **loadLevel()** method:

```
func loadLevel() {
  var clueString = ""
  var solutionString = ""
  var letterBits = [String]()
```

```swift
    if let levelFilePath = NSBundle.mainBundle().pathForResource("level
\(level)", ofType: "txt") {

        if let levelContents = try? String(contentsOfFile: levelFilePath,
usedEncoding: nil) {

            var lines = levelContents.componentsSeparatedByString("\n")

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


            for (index, line) in lines.enumerate() {

                let parts = line.componentsSeparatedByString(": ")

                let answer = parts[0]

                let clue = parts[1]


                clueString += "\(index + 1). \(clue)\n"


                let solutionWord =
answer.stringByReplacingOccurrencesOfString("|", withString: "")

                solutionString += "\(solutionWord.characters.count) letters
\n"

                solutions.append(solutionWord)


                let bits = answer.componentsSeparatedByString("|")

                letterBits += bits
            }
        }
    }


    // Now configure the buttons and labels
}
```

If you read all that and it made sense first time, great! You can skip over the next few paragraphs and jump to the bit the bold text "All done!". If you read it and only some made sense, these next few paragraphs are for you.

First, the method uses **pathForResource()** and **String**'s **contentsOfFile** to find and load the level string from the disk. String interpolation is used to combine "level" with our current level number, making "level1.txt". The text is then split into an array by breaking on the **\n** character (that's line break, remember), then shuffled so that the game is a little different each time.

Our loop uses the **enumerate()** method to go through each item in the **lines** array. This is different to how we normally loop through an array, but **enumerate()** is helpful here because it tells us where each item was in the array so we can use that information in our clue string. In the code above, **enumerate()** will place the item into the **line** variable and its position into the **index** variable.

We already split the text up into lines based on finding **\n**, but now we split each line up based on finding **:** , because each line has a colon and a space separating its letter groups from its clue. We put the first part of the split line into **answer** and the second part into **clue**, for easier referencing later.

Now, here's something new: you've already seen how string interpolation can turn **level\(level)** into "level1" because the **level** variable is set to 1, but here we're adding to the **clueString** variable using **\(index + 1)**. Yes, we're actually doing basic math in our string interpolation. This is needed because the array indexes start from 0, which looks strange to players, so we add 1 to make it count from 1 to 7.

Next comes our new string method call, **stringByReplacingOccurrencesOfString()**. We're asking it to replace all instances of | with an empty string, so HA|UNT|ED will become HAUNTED. We then use **characters.count** to get the length of our string then use that in combination with string interpolation to add to our solutions string.

Finally, we make yet another call to **componentsSeparatedByString()** to turn the string "HA| UNT|ED" into an array of three elements, then add all three to our **letterBits** array

All done!Time for some more code: our current **loadLevel()** method ends with a comment saying **// Now configure the buttons and labels**, and we're going to fill that in with the final part of the method. This needs to set the **cluesLabel** and **answersLabel** text, shuffle up our buttons and letter groups, then assign letter groups to buttons.

Before I show you the actual code, there's a new string method to introduce, and it's another long one: **stringByTrimmingCharactersInSet()** removes any letters you specify from the start and end of a string. It's most frequently used with the parameter **.whitespaceAndNewlineCharacterSet()**, which trims spaces, tabs and line breaks, and we need exactly that here because our clue string and solutions string will both end up with an extra line break.

Put this code where the comment was:

```
cluesLabel.text =
clueString.stringByTrimmingCharactersInSet(.whitespaceAndNewlineChara
cterSet())

answersLabel.text =
solutionString.stringByTrimmingCharactersInSet(.whitespaceAndNewlineC
haracterSet())


letterBits =
GKRandomSource.sharedRandom().arrayByShufflingObjectsInArray(letterBi
ts) as! [String]

letterButtons =
GKRandomSource.sharedRandom().arrayByShufflingObjectsInArray(letterBu
ttons) as! [UIButton]


if letterBits.count == letterButtons.count {

  for i in 0 ..< letterBits.count {

    letterButtons[i].setTitle(letterBits[i], forState: .Normal)

  }

}
```

That code uses yet another type of loop, and this time it's a range: **for i in 0 ..< letterButtons.count** means "count from 0 up to but not including the number of buttons." This is useful because we have as many items in our **letterBits** array as our **letterButtons** array. Looping from 0 to 19 (inclusive) means we can use the **i** variable to set a button to a letter group.

The **..<** operator is called the "half-open range operator" because it does not include the upper limit. Instead, it counts to one below. There's a closed range operator, **...**, which includes the upper limit, but we don't want that here because an array of 20 items will have numbers 0 to 19.

Before you run your program, make sure you add a call to **loadLevel()** in your **viewDidLoad()** method. Once that's done, you should be able to see all the buttons and clues configured correctly. Now all that's left is to let the player, well, *play*.

It's play time!

We need to add three more methods to our view controller in order to finish this game: one to handle letter buttons being tapped, another to handle the current word being cleared, and a third to handle the current word being submitted. The first two are extremely easy, so let's get those done so we can get onto the serious stuff.

First, we already used the **addTarget()** method in **viewDidLoad()** to make all our letter buttons call the method **letterTapped()**, and you should remember that actually had to specify **letterTapped:** because we want to receive the button that was tapped as a parameter for our method. Add this method now somewhere in your code:

```
func letterTapped(btn: UIButton) {
    currentAnswer.text = currentAnswer.text! + btn.titleLabel!.text!
    activatedButtons.append(btn)
    btn.hidden = true
}
```

That does three things: gets the text from the title label of the button that was tapped and appends it to the current text of the answer text field, then appends the button to the **activatedButtons** array, and finally hides the button. We need to force unwrap both the title label and its text, because both might not exist – and yet we know they do.

The **activatedButtons** array is being used to hold all buttons that the player has tapped before submitting their answer. This is important because we're hiding each button as it is tapped, so when the user taps "Clear" we need to know which buttons are currently in use so we can re-show them. You already created an empty **@IBAction** method for clear being tapped, so fill it in like this:

```
@IBAction func clearTapped(sender: AnyObject) {
    currentAnswer.text = ""

    for btn in activatedButtons {
```

```
      btn.hidden = false

   }


   activatedButtons.removeAll()

}
```

As you can see, this method removes the text from the current answer text field, unhides all the activated buttons, then removes all the items from the **activatedButtons** array.

That just leaves one final method, and you already created its stub: the **submitTapped()** method for when the player taps the submit button.

This method will use another new function called **indexOf()**, which searches through an array for an item and, if it finds it, tells you its position. The return value is optional so that in situations where nothing is found you won't get a value back, so we need to unwrap its return value carefully.

If the user gets an answer correct, we're going to change the answers label so that rather than saying "7 LETTERS" it says "HAUNTED", so they know which ones they have solved already. The way we're going to do this is delightfully simple: **indexOf()** will tell us which solution matched their word, and that we can use that position to find the matching clue text. All we need to do is split the answer label text up by **\n**, replace the line at the solution position with the solution itself, then re-join the clues label back together.

You've already learned how to use **componentsSeparatedByString()** to split text into an array, and now it's time to meet its counterpart: **joinWithSeparator()**. This makes an array into a single string, with each array element separated by the string specified in its parameter.

Once that's done, we clear the current answer text field and add one to the score. If the score is evenly divisible by 7, we know they have found all seven words so we're going to show a **UIAlertController** that will prompt the user to go to the next level.

The "evenly divisible" task is easy to do in Swift (and indeed any sensible programming language) thanks to a dedicated modulo operator: %. Modulo is division with remainder, so 10 % 3 means "tell me what number remains when you divide 10 evenly into 3 parts". 3 goes

into 10 three times (making nine), with remainder 1, so 10 % 3 is 1, 11 % 3 is 2, and 12 % 3 is 0 – i.e., 12 divides perfectly into 3 with no remainder. If score % 7 is 0, we know they have answered all seven words correctly.

That's all the parts explained, so here's the final **submitTapped()** method:

```
@IBAction func submitTapped(sender: AnyObject) {
  if let solutionPosition = solutions.indexOf(currentAnswer.text!) {
    activatedButtons.removeAll()


    var splitClues =
answersLabel.text!.componentsSeparatedByString("\n")

    splitClues[solutionPosition] = currentAnswer.text!

    answersLabel.text = splitClues.joinWithSeparator("\n")


    currentAnswer.text = ""

    score += 1


    if score % 7 == 0 {
      let ac = UIAlertController(title: "Well done!", message: "Are
you ready for the next level?", preferredStyle: .Alert)

      ac.addAction(UIAlertAction(title: "Let's go!", style: .Default,
handler: levelUp))

      presentViewController(ac, animated: true, completion: nil)

    }

  }

}
```

The **levelUp()** call in there is just to get you started – there isn't a level up here, because I only created one level! But if you wanted to make more levels and continue the game, you'd need a **levelUp()** method something like this:

```swift
func levelUp(action: UIAlertAction!) {
    level += 1
    solutions.removeAll(keepCapacity: true)

    loadLevel()

    for btn in letterButtons {
        btn.hidden = false
    }
}
```

As you can see, that code clears out the existing **solutions** array before refilling it inside **loadLevel()**. Then of course you'd need to create level2.txt, level3.txt and so on. To get you started, I've made an example level2.txt for you inside the Content folder – try adding that to the project and see what you think. Any further levels are for you to do – just make sure there's a total of 20 letter groups each time!

Property observers

There's one last thing to cover before this project is done, and it's really small and really easy: property observers.

Right now we have a property called **score** that is set to 0 when the game is created and increments by one whenever an answer is found. But we don't do anything with that score, so our score label is never updated.

One solution to this problem is to use something like **scoreLabel.text = "Score: \(score)"** whenever the score value is changed, and that's perfectly fine to begin with. But what happens if you're changing the score from several places? You need to keep all the code synchronised, which is unpleasant.

Swift has a simple and classy solution called property observers, and it lets you execute code whenever a property has changed. To make them work, you need to declare your data type explicitly (in our case we need an **Int**), then use either **didSet** to execute code when a property has just been set, or **willSet** to execute code before a property has been set.

In our case, we want to add a property observer to our **score** property so that we update the score label whenever the score value was changed. So, change your **score** property to this:

```
var score: Int = 0 {
  didSet {
    scoreLabel.text = "Score: \(score)"
  }
}
```

Note that when you use a property observer like this, you need to explicitly declare its type otherwise Swift will complain.

Using this method, any time **score** is changed by anyone, our score label will be updated. That's it, the project is done!

Wrap up

Yes, it took quite a lot of storyboard work to get this project going, but I hope it has shown you that you can make some great games using just the UIKit tools you already know.

Of course, at the same time as making another game, you've made several steps forward in your iOS development quest, this time learning about **addTarget()**, **enumerate()**, **countElements()**, **find()**, **join()**, **stringByReplacingOccurrencesOfString()**, property observers, range operators and the difference between Swift strings and **NSString**.

Looking at that list, it should be clear that you are increasingly dealing with specific bits of code (i.e., functions like **find()**) when you're developing UIKit projects. This is because you're starting to build up a great repertoire of code, so there is simply less to teach. That's not to say there isn't a lot of new things still to come – in fact, the next few projects all introduce several big new things – but it does mean your knowledge is starting to mature.

If you're looking to improve this project, see if you can make it deduct points if the player makes an incorrect guess - this is just a matter of extending the **submitAnswer()** method so that if **find()** failed to find the guess then you remove points.

Project 9
Grand Central Dispatch

Learn how to run complex tasks in the background with GCD.

Setting up

In this technique project we're going to return to project 7 to solve a critical problem using one of the most important Apple frameworks available: Grand Central Dispatch, or GCD. I already mentioned the problem to you, but here's a recap from project 7:

By downloading data from the internet in **viewDidLoad()** *our app will lock up until all the data has been transferred. There are solutions to this, but to avoid complexity they won't be covered until project 9.*We're going to solve this problem by using GCD, which will allow us to fetch the data without locking up the user interface. But be warned: even though GCD might seem easy at first, it opens up a new raft of problems, so be careful!

If you want to keep your previous work for reference, take a copy of project 7 now and call it project 9. Otherwise, just modify it in place.

Why is locking the UI bad?

The answer is two-fold. First, we used **NSData**'s **contentsOfURL** to download data from the internet, which is what's known as a *blocking* call. That is, it blocks execution of any further code in the method until it has connected to the server and fully downloaded all the data.

Second, behind the scenes your app actually executes multiple sets of instructions at the same time, which allows it to take advantage of having two CPU cores, or even three as in the iPad Air 2. Each CPU can be doing something independently of the others, which hugely boosts your performance. These code execution processes are called *threads*, and come with a number of important provisos:

1. Threads execute the code you give them, they don't just randomly execute a few lines from **viewDidLoad()** each. This means by default your own code executes on only one CPU, because you haven't created threads for other CPUs to work on.
2. All user interface work must occur on the main thread, which is the initial thread your program is created on. If you try to execute code on a different thread, it might work, it might fail to work, it might cause unexpected results, or it might just crash.
3. You don't get to control when threads execute, or in what order. You create them and give them to the system to run, and the system handles executing them as best it can.
4. Because you don't control the execution order, you need to be extra vigilant in your code to ensure only one thread modifies your data at one time.

Points 1 and 2 explain why our call is bad: if all user interface code must run on the main thread, and we just blocked the main thread by using **NSData**'s **contentsOfURL**, it causes the entire program to freeze – the user can touch the screen all they want, but nothing with happen. When the data finally downloads (or just fails), the program will unfreeze. This is a terrible experience, particularly when you consider that iPhones are frequently on poor-quality data connections.

Broadly speaking, if you're accessing any remote resource, you should be doing it on a *background thread* – i.e., any thread that is not the main thread. If you're executing any slow code, you should be doing it on a background thread. If you're executing any code that can be run in parallel – e.g. adding a filter to 100 photos – you should be doing it on multiple background threads.

The power of GCD is that it takes away a lot of the hassle of creating and working with multiple threads, known as *multithreading*. You don't have to worry about creating and destroying threads, and you don't have to worry about ensuring you have created the optimal

number of threads for the current device. GCD automatically creates threads for you, and executes your code on them in the most efficient way it can.

To fix our project, you need to learn three new GCD functions, but the most important one is called **dispatch_async()** – it means "run the following code asynchronously," i.e. don't block (stop what I'm doing right now) while it's executing. Yes, that seems simple, but there's a sting in the tail: you need to use closures. Remember those? They are your best friend. No, really.

GCD 101

We're going to use **dispatch_async()** twice: once to push some code to a background thread, then once more to push code back to the main thread. This allows us to do any heavy lifting away from the user interface where we don't block things, but then update the user interface safely on the main thread – which is the only place it can be safely updated.

When you call **dispatch_async()**, you must tell it where you want the code to run. GCD works with a system of queues, which are much like a real-world queue: they are First In, First Out (FIFO) blocks of code. What this means is that your GCD calls don't create threads to run in, they just get assigned to one of the existing threads for GCD to manage.

GCD creates for you a number of queues, and places tasks in those queues depending on how important you say they are. All are FIFO, meaning that each block of code will be taken off the queue in the order they were put in, but more than one code block can be executed at the same time so the finish order isn't guaranteed.

"How important" some code is depends on something called "quality of service", or QoS, which decides what level of service this code should be given. Obviously at the top of this is the main queue, which runs on your main thread, and should be used to schedule any work that must update the user interface immediately even when that means blocking your program from doing anything else. But there are four background queues that you can use, each of which has their own QoS level set:

1. User Interactive: this is the highest priority background thread, and should be used when you want a background thread to do work that is important to keep your user interface working. This priority will ask the system to dedicate nearly all available CPU time to you to get the job done as quickly as possible.
2. User Initiated: this should be used to execute tasks requested by the user that they are now waiting for in order to continue using your app. It's not as important as user interactive work – i.e., if the user taps on buttons to do other stuff, that could should be executed first – but it is important because you're keeping the user waiting.
3. The Utility queue: this should be used for long-running tasks that the user is aware of, but not necessarily desperate for now. If the user has requested something and can happily leave it running while they do something else with your app, you should use Utility.
4. The Background queue: this is for long-running tasks that the user isn't actively aware of, or at least doesn't care about its progress or when it completes.

Those QoS queues affect the way the system prioritises your work: User Interactive and User

Initiated tasks will be executed as quickly as possible regardless of their effect on battery life, Utility tasks will be executed with a view to keeping power efficiency as high as possible without sacrificing too much performance, whereas Background tasks will be executed with power efficiency as its priority.

GCD automatically balances work so that higher priority queues are given more time than lower priority ones, even if that means temporarily delaying a background task because a user interactive task just came in.

Enough talking, time for some action: we're going to use **dispatch_async()** to make all our loading code run in the background queue with User Initiated quality of service. It's actually only two lines of code different:

```
dispatch_async(dispatch_get_global_queue(QOS_CLASS_USER_INITIATED,
0)) { [unowned self] in
```

…before the code you want to run in the background, then a closing brace at the end.

dispatch_async() takes one parameter, then a closure to execute asynchronously. The parameter it takes is which queue you want to use, and we're going to use one of two functions: **dispatch_get_global_queue()** asks for a queue with a particular quality of service setting, and **dispatch_get_main_queue()** will use the main queue.

The **dispatch_get_global_queue()** also takes parameters, but the second one is always 0 – Apple has left it there in case they want to make changes in the future, but until you hear otherwise just put a 0 in there. The first parameter is the name of the queue you want to use, and in our code that's **QOS_CLASS_USER_INITIATED**: we want our code to execute quickly because we know the user is waiting for it.

Because **dispatch_async()** uses closures, we start with **[unowned self] in** to avoid strong reference cycles, but otherwise our loading code is the same as before. To help you place it correctly, here's how the loading code should look:

```
dispatch_async(dispatch_get_global_queue(QOS_CLASS_USER_INITIATED,
0)) { [unowned self] in
    if let url = NSURL(string: urlString) {
```

```swift
    if let data = try? NSData(contentsOfURL: url, options: []) {
      let json = JSON(data: data)


      if json["metadata"]["responseInfo"]["status"].intValue == 200 {
        self.parseJSON(json)
      } else {
        self.showError()
      }
    } else {
      self.showError()
    }
  } else {
    self.showError()
  }
}
```

Note that because our code is now inside a closure, we need to prefix our method calls with **self.** otherwise Swift complains.

If you want to try the other QoS queues, you could also use **QOS_CLASS_USER_INTERACTIVE**, **QOS_CLASS_UTILITY** or **QOS_CLASS_BACKGROUND**.

Back to the main thread

With this change, our code is both better and worse. It's better because it no longer blocks the main thread while the JSON downloads from Whitehouse.gov. It's worse because we're pushing work to the background thread, *and any further code called in that work will also be on the background thread*.

So, if the download fails, **loadError()** will be called on the background thread and its **UIAlertController** will be created and shown on the background thread. If the download succeeds, the JSON will be parsed on the background thread and the table view's **reloadData()** will be called on the background thread.

Now, it's OK to parse the JSON there, but *it's never OK to do user interface work on the background thread*.

That's so important it bears repeating twice: **it's never OK to do user interface work on the background thread**.

If you're on a background thread and want to execute code on the main thread, you need to call **dispatch_async()** again, this time using the function **dispatch_get_main_queue()** because you want to run on the main thread. There are two ways we could modify our code to have **dispatch_get_main_queue()** before every call to **showError()** and **parseJSON()**, but that's both ugly and inefficient.

Instead, it's better to place the **dispatch_async()** call inside **showError()**, wrapping up the whole **UIAlertController** code, and also inside **parseJSON()**, but only where the table view is being reloaded. The actual JSON parsing can happily stay on the background thread.

So, inside the **parseJSON()** method find this code:

```
tableView.reloadData()
```

…and replace it this new code, bearing in mind again the need for **[unowned self] in** and **self.** to keep away strong reference cycles:

```
dispatch_async(dispatch_get_main_queue()) { [unowned self] in
```

```
    self.tableView.reloadData()

}
```

And now change the **showError()** method to this:

```
func showError() {
    dispatch_async(dispatch_get_main_queue()) { [unowned self] in

        let ac = UIAlertController(title: "Loading error", message:
"There was a problem loading the feed; please check your connection
and try again.", preferredStyle: .Alert)

        ac.addAction(UIAlertAction(title: "OK", style: .Default, handler:
nil))

        self.presentViewController(ac, animated: true, completion: nil)

    }

}
```

The code is almost identical, it's just a matter of wrapping it up neatly to avoid strong reference cycles.

At this point, this code is in a good place: we do all the slow work off the main thread, then push work back to the main thread when we want to do user interface work. This background/foreground bounce is common, and you'll see it again in later projects.

Wrap up

Although I've tried to simplify things as much as possible, GCD still isn't easy. That said, it's much easier than the alternatives: GCD automatically handles thread creation and management, automatically balances based on available system resources, and automatically factors in Quality of Service to ensure your code runs as efficiently as possible. The alternative is doing all that yourself!

There's a lot more we could cover (not least how to create your own queues!) but really you have more than enough to be going on with, and certainly more than enough to complete the rest of this series. We'll be using GCD again, so it might help to keep this reference close to hand!

Project 10

Names to Faces

Get started with UICollectionView and the photo library.

Setting up

This is a fun, simple and useful project that will let you create an app to help store names of people you've met. If you're a frequent traveller, or perhaps just bad at putting names to faces, this project will be perfect for you.

And yes, you'll be learning lots along the way: this time you'll meet **UICollectionViewController**, **UIImagePickerController** and **NSUUID**. Plus you'll get to do more with your old pals **CALayer**, **UIAlertController**, **NSData** and closures. But above all, you're going to learn how to make a new data type from scratch for the first time.

Create a new Single View Application project in Xcode, call it Project10, set its target to any device you want, then save it somewhere. This should be second nature to you by now – you're becoming a veteran!

Designing UICollectionView cells

Open Interface Builder with Main.storyboard, then embed the initial view controller inside a navigation controller. Now, using the object library (Ctrl+Alt+Cmd+3) search for a Collection View and drag it onto the view controller so that it takes up the full space. **Important:** don't choose the one with a yellow icon, because that's a Collection View Controller rather than a **UICollectionView**. They are different!

With the collection view selected, go to the size inspector and set Cell Size to have the width 140 and height 180. Now set the section insets for top, bottom, left and right to all be 10.

Collection views are extremely similar to table views, with the exception that they display as grids rather than as simple rows. But while the display is different, the underlying method calls are so similar that you would be able to dive right in if it weren't for the fact that we need to write all the code ourselves this time – there's no collection view template code we can modify!

Using the assistant editor, create an outlet for the collection view called **collectionView**, then return back to the standard editor. Trying to edit a storyboard while you can only see half the screen isn't easy!

When you create a collection view, you get one initial collection view cell defined for you, called a *prototype*. This is the empty square you'll see in the top-left corner. This works the same as with table views; you'll remember we changed the initial cell in project 7 so that we could add subtitles.

Select that collection view cell now, then go to the attributes inspector and change its Background from "Default" (transparent) to white. Now place a **UIImageView** in there, with X: 10, Y:10, width 120 and height 120. We'll be using this to show pictures of people's faces.

Place a **UILabel** in there too, with X:10, Y:134, width 120 and height 40. In the attributes inspector, change the label's font by clicking the T button and choosing "Custom" for font, "Marker Felt" for family, and "Thin" for style. Give it the font size 16, which is 1 smaller than the default, then set its alignment to be centered and its number of lines property to be 2.
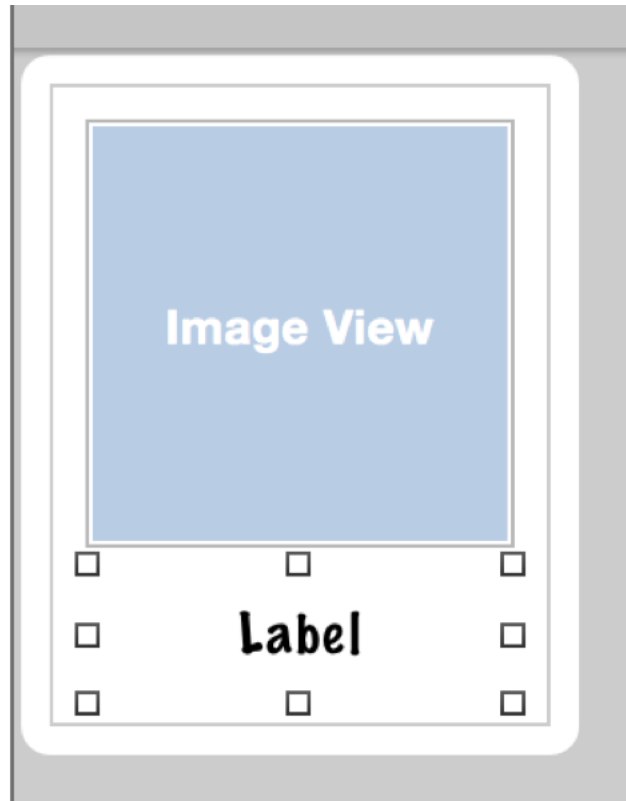
Your collection view cell design should have one image view and one label.

So far this has been fairly usual storyboard work, but now we're going to do two things we've never done before: assign a delegate from a storyboard, then create a custom class for our collection view cell. The first is simple: select the collection view (not the cell!) by clicking the large empty area, then Ctrl-drag from there up to where it says "View Controller" in the document outline on the left.

When you let go, you'll be asked what kind of connection you want to make, and you'll see two options you haven't seen before: **dataSource** and **delegate**. I want you to select both, which means you'll need to Ctrl-drag twice.

What you've just done is tell the collection view that your **ViewController** class (the one Xcode made for you as part of its template) will provide all the data for the collection view (it's data source) and will also respond to any interesting events from the collection view (its delegate). We'll need to return to this some more soon.

Now for something harder: we need to create a custom class for our collection view cell. This is because our collection view cell has two views that we created – the image view and the

label – and we need a way to manipulate this in code. The shortcut way would be to give them unique tags and give them variables when the app runs, but we're going to do it The Proper Way this time so you can learn.

Read this carefully, because you'll need to do it a lot from here on. In the project navigator, the top thing is a blue project icon that says "Project10 / 2 targets". Beneath that is a yellow folder that also says Project10. Right-click on that and choose "New File…". Whenever I say to create a new file in later projects, please follow this exact procedure.

In the picture below you can see how adding a new file should look: right-click on the yellow folder that says Project10 (or whatever you named your project) then look for New File in the popup menu.
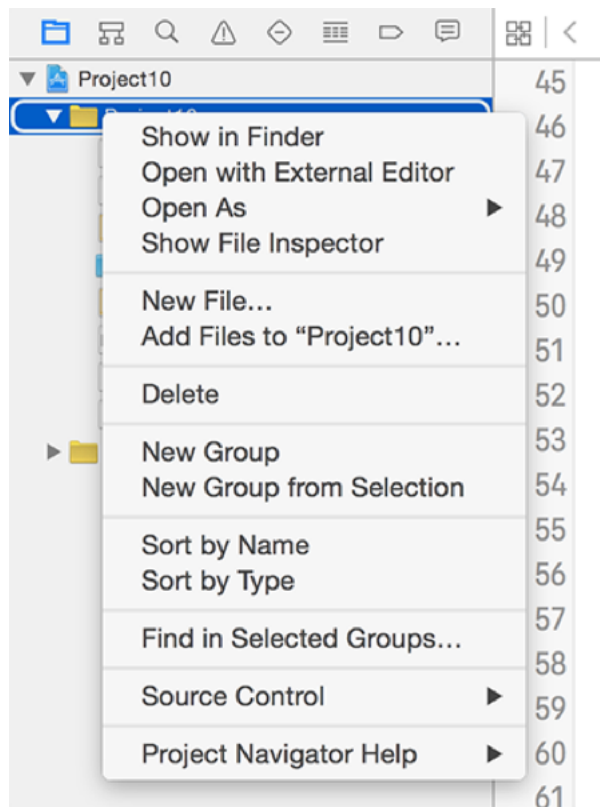
How to add a new file to a project in Xcode.

You'll be asked to choose a template for your new file. From the left, make sure "Source" is selected under the "iOS" heading, then choose Cocoa Touch Class and click Next. You'll be asked to fill in two text fields: where it says "Subclass of" you should enter "UICollectionViewCell", and where it says "Class" enter "PersonCell". Click Next then Create,

and Xcode will create a new class called **PersonCell** that inherits from **UICollectionViewCell**.

This new class needs to be able to represent the collection view layout we just defined in Interface Builder, so it just needs two outlets. Modify the class to this:

```
@IBOutlet weak var imageView: UIImageView!
@IBOutlet weak var name: UILabel!
```

Now go back to Interface Builder and select the collection view cell in the document outline. Select the identity inspector (Cmd+Alt+3) and you'll see next to Class the word "UICollectionViewCell" in gray text. That's telling us that the cell is its default class type.

We want to use our custom class here, so enter "PersonCell" and hit return. You'll see that "PersonCell" now appears in the document outline. While you're there, go to the attributes inspector and give the cell the identifier "Person".

Now that Interface Builder knows that the cell is actually a **PersonCell**, we can connect its outlets. Go to the connections inspector (it's the last one, so Alt+Cmd+6) with the cell selected and you'll see **imageView** and **name** in there, both with empty circles to their right. That empty circle has exactly the same meaning as when you saw it with outlets in code: there is no connection between the storyboard and code for this outlet.

To make a connection from the connections inspector, just click on the empty circle next to **imageView** and drag a line over the view you want to connect. In our case, that means dragging over the image view in our custom cell. Now connect **name** to the label, and you're almost done with the storyboard.
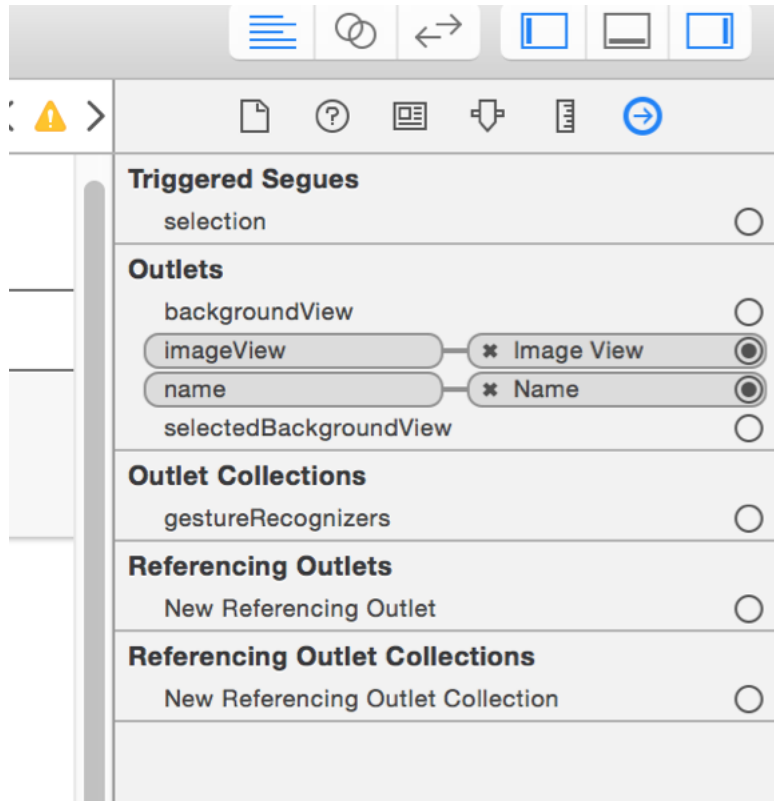
You can create and destroy Interface Builder outlets using the connections inspector.

The last thing to do is make sure the collection view stays edge to edge with the view controller regardless of device. With the collection view selected, click the Pin button, which is the third of four buttons in the bottom-right corner of the Interface Builder pane.

Deselect "Constrain to margins" then click all four dotted red lines near the top of the popup and click "Add 4 Constraints." We did this in project 1, but I appreciate that was some time ago!

Enough storyboarding: time for some code…

Data sources and delegates

In project 4 we made ourselves the delegate of a **WKWebView**'s navigation, and as soon as that happened there were compiler problems. We also had to tell Swift that we conformed to the **WKNavigationDelegate** protocol in order to make the code work.

WKNavigationDelegate is a very easy protocol to conform to, because all its methods are optional. That means you don't need to implement anything, you just need to say that you conform to the protocol.

When you make your view controller to be the data source and delegate of a collection using a storyboard, you won't get any compiler problems, but the code won't work. You don't get any compiler problems because the compiler doesn't check storyboard connections, but the code won't work because the **UICollectionViewDataSource** protocol has two non-optional methods that we need to implement.

The best next step from here is to tell Swift in code that your view controller conforms to the **UICollectionViewDataSource** and **UICollectionViewDelegate** protocols, because that way it can make sure your code is valid. So, go to ViewController.swift and modify your view controller's class definition to this:

```
class ViewController: UIViewController, UICollectionViewDataSource,
UICollectionViewDelegate {
```

Once you've made this change, you'll see compiler errors because we don't fully conform to the **UICollectionViewDataSource** protocol. Specifically, you must tell the collection view how many items of data it should expect and what each item should contain. To begin with, let's put together the most basic implementation that allows our code to build cleanly. Add these two methods:

```
func collectionView(collectionView: UICollectionView,
numberOfItemsInSection section: Int) -> Int {

    return 10

}
```

```
func collectionView(collectionView: UICollectionView,
cellForItemAtIndexPath indexPath: NSIndexPath) ->
UICollectionViewCell {

   let cell =
collectionView.dequeueReusableCellWithReuseIdentifier("Person",
forIndexPath: indexPath) as! PersonCell

   return cell

}
```

We haven't looked at any of this code before, so I want to pull it apart in detail before continuing:

 • **collectionView(_:numberOfItemsInSection:)** This must return an integer, and tells the collection view how many items you want to show in its grid. I've returned 10 from this method, but soon we'll switch to using an array.
 • **collectionView(_:cellForItemAtIndexPath:)** This must return an object of type **UICollectionViewCell**. We already designed a prototype in Interface Builder, and configured the **PersonCell** class for it, so we need to create and return one of these.
 • **dequeueReusableCellWithReuseIdentifier()** This crazy long method name does something really important: it creates a collection view cell using the reuse identified we specified, in this case "Person" because that was what we typed into Interface Builder earlier. But even better, this method will automatically try to reuse collection view cells, so as soon as a cell scrolls out of view it can be recycled so that we don't have to keep creating new ones.

Note that we need to typecast our collection view cell as a **PersonCell** because we'll soon want to access its **imageView** and **name** outlets.

These two new methods both come from the **UICollectionViewDataSource** protocol, but both are remarkably similar to the **UITableViewDataSource** delegate that is the equivalent for table views – you can go back and open project 1 again to see just how similar!

Press Cmd+R to run your project now, and you'll see the beginning of things start to come together: the prototype cell you designed in Interface Builder will appear 10 times, and you can scroll up and down to view them all. As you'll see, you can fit two cells across the screen, which is what makes the collection view different to the table view. Plus, if you rotate to landscape you'll see it automatically (and beautifully) animates the movement of cells so they take up the full width.
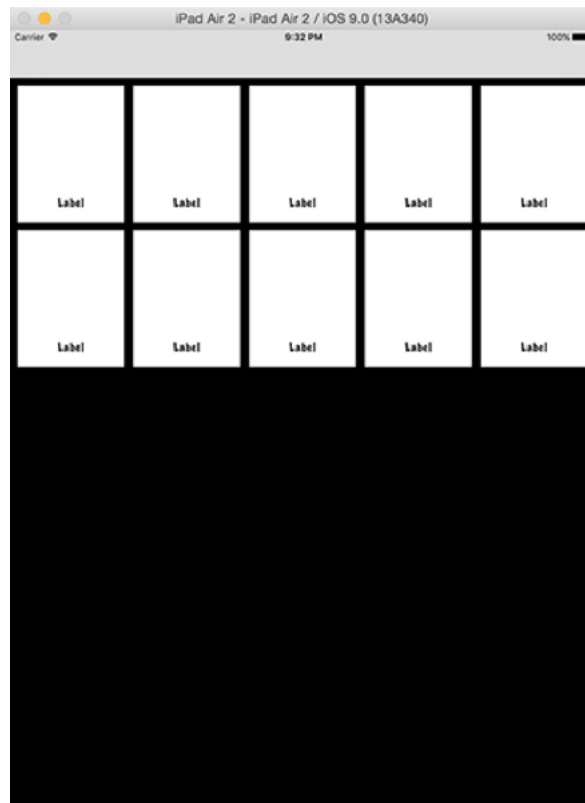
A UICollectionView filled with our basic cell design.

Importing photos

There are lots of events that make up the **UICollectionViewDelegate** protocol to handle when the user interacts with a cell, but we'll come back to that later. For now, let's look at how to import pictures using **UIImagePickerController**. This new class is designed to let users select an image from their camera to import into an app. When you first create a **UIImagePickerController**, iOS will automatically ask the user whether the app can access their photos.

First, we need to create a button that lets users add people to the app. This is as simple as putting the following into the **viewDidLoad()** method:

```
navigationItem.leftBarButtonItem =
UIBarButtonItem(barButtonSystemItem: .Add, target: self, action:
"addNewPerson")
```

The **addNewPerson()** method is where we need to use the **UIImagePickerController**, but it's so easy to do I'm just going to show you the code:

```
func addNewPerson() {

   let picker = UIImagePickerController()

   picker.allowsEditing = true

   picker.delegate = self

   presentViewController(picker, animated: true, completion: nil)

}
```

There are only two interesting things in there. First, we set the **allowsEditing** property to be true, which allows the user to crop the picture they select. Second, when you set **self** as the delegate, you'll need to conform not only to the **UIImagePickerControllerDelegate** protocol, but also the **UINavigationControllerDelegate** protocol.

The first of those protocols is useful, telling us when the user either selected a picture or cancelled the picker. The second, **UINavigationControllerDelegate**, really is quite pointless

here, so don't worry about it beyond just modifying your class declaration to include the protocol.

When you conform to the **UIImagePickerControllerDelegate** protocol, you don't need to add any methods because both are optional. But they aren't really – they are marked optional for whatever reason, but your code isn't much good unless you implement them! Let's tackle them individually, starting with **imagePickerControllerDidCancel()**. Add this to your class:

```
func imagePickerControllerDidCancel(picker: UIImagePickerController)
{
    dismissViewControllerAnimated(true, completion: nil)
}
```

So, if the user cancels the image picker, we dismiss it. This is required because the default **UIImagePickerController** behaviour is to take up the full screen, so we need to hide it to return to our view controller.

The much more complicated delegate method is **imagePickerController(_, didFinishPickingMediaWithInfo:)**, which returns when the user selected an image and it's being returned to you. This method needs to do several things:

• Extract the image from the dictionary that is passed as a parameter.
• Generate a unique filename for it.
• Convert it to a JPEG, then write that JPEG to disk.
• Dismiss the view controller.

To make all this work you're going to need to learn a few new things.

First, it's very common for Apple to send you a dictionary of several pieces of information as a method parameter. This can be hard to work with sometimes because you need to know the names of the keys in the dictionary in order to be able to pick out the values, but you'll get the hang of it over time.

This dictionary parameter will contain one of two keys: **UIImagePickerControllerEditedImage** (the image that was edited) or **UIImagePickerControllerOriginalImage**, and realistically it should only ever be the former

unless you change the **allowsEditing** property.

The problem is, we don't know if these values exist as **UIImage**s, so we can't just extract them straight into **UIImage**s. Instead, we need to use an optional method of typecasting, **as?**, along with if/let syntax. Using this method, we can be sure we always get the right thing out.

Second, we need to generate a unique filename for every image we import. This is so that we can copy it to our app's space on the disk without overwriting anything, and if the user ever deletes the picture from their photo library we still have our copy. We're going to use a new class for this, called **NSUUID**, which generates a Universally Unique Identifier and is perfect for a random filename.

Third, once we have the image, we need to write it to disk. You're going to need to learn two new pieces of code: **UIImageJPEGRepresentation()** converts a **UIImage** to an **NSData**, and there's a method on NSData called **writeToFile()** that, well, writes its data to disk.

Writing information to disk is easy enough, but finding where to put it is tricky. All apps that are installed have a directory called Documents where you can save private information for the app, and it's also automatically synchronised with iCloud. The problem is, it's not obvious how to find that directory, so I have a method I use called **getDocumentsDirectory()** that does exactly that – you don't need to understand how it works, but you do need to copy it into your code.

With all that in mind, here are the new methods:

```
func imagePickerController(picker: UIImagePickerController,
didFinishPickingMediaWithInfo info: [String : AnyObject]) {

  var newImage: UIImage


  if let possibleImage = info["UIImagePickerControllerEditedImage"]
as? UIImage {

    newImage = possibleImage

  } else if let possibleImage =
info["UIImagePickerControllerOriginalImage"] as? UIImage {

    newImage = possibleImage
```

```
  } else {

    return

  }


  let imageName = NSUUID().UUIDString

  let imagePath =
getDocumentsDirectory().stringByAppendingPathComponent(imageName)


  if let jpegData = UIImageJPEGRepresentation(newImage, 80) {

    jpegData.writeToFile(imagePath, atomically: true)

  }


  dismissViewControllerAnimated(true, completion: nil)

}


func getDocumentsDirectory() -> NSString {

  let paths =
NSSearchPathForDirectoriesInDomains(.DocumentDirectory, .UserDomainMa
sk, true)

  let documentsDirectory = paths[0]

  return documentsDirectory

}
```

Again, it doesn't matter how **getDocumentsDirectory()** works, but if you're curious: its first parameter asks for the documents directory, and its second parameter adds that we want the path to be relative to the user's home directory. This returns an array that nearly always contains only one thing: the user's documents directory. So, we pull out the first element and return it.

Now onto the code that matters: as you can see I've declared a **newImage** variable up front, then it gets assigned to whichever value in the dictionary has an image. If we can't find an image for some reason, we exit the method. We then create an **NSUUID** object, and use its **UUIDString** property to extract the unique identifer as a string data type.

The code then creates a new constant, **imagePath**, which takes the string result of **getDocumentsDirectory()** and calls a new method on it: **stringByAppendingPathComponent()**. This is used when working with file paths, and adds one string (**imageName** in our case) to a path, including whatever path separator is used on the platform.

Now that we have a **UIImage** containing an image and a path where we want to save it, we need to convert the **UIImage** to an **NSData** object so it can be saved. To do that, we use the **UIImageJPEGRepresentation()** function, which takes two parameters: the **UIImage** to convert to JPEG and a quality value between 0 and 100.

Once we have an **NSData** object containing our JPEG data, we just need to unwrap it safely then write it to the file name we made earlier. That's done using the **writeToFile()** method, which takes a filename as its first parameter and a boolean as its second. That second parameter, "atomically", should generally be true. It means "write to a temporary file first, then rename it to be the file you asked," which has the benefit that the file doesn't appear to exist until it has been fully written.

So: users can pick an image, and we'll save it to disk. But this still doesn't do anything – you won't see the picture in the app, because we aren't doing anything with it beyond writing it to disk. To fix that, we need to create a custom class to hold custom data…

Custom classes

You already created your first custom class when you created the collection view cell. But this time we're going to do something very simple: we're going to create a class to hold some data for our app. So far you've seen how we can create arrays of strings by using **[String]**, but what if we want to hold an array of *people*?

Well, the solution is to create a custom class. Create a new file (you remember my explicit, "read this carefully" instructions, right?) and choose Cocoa Touch Class. Click Next and name the class Person, type **NSObject** for "Subclass of", then click Next and Create to create the file.

NSObject is what's called a *universal base class* for all Cocoa Touch classes. That means all UIKit classes ultimately come from **NSObject**, as do all NS objects like **NSString**. You don't have to inherit from **NSObject** in Swift, but you did in Objective C and in fact there are some behaviors you can only have if you do inherit from it. More on that in project 12, but for now just make sure you inherit from **NSObject**.

We're going to add two properties to our class: a name and a photo for every person. So, add this inside the **Person** definition:

```
var name: String
var image: String
```

When you do that, you'll see errors: "Class 'Person' has no initializers." This is a term I've skipped over so far, but now is a good time to introduce it: an initializer method is something that creates instances of a class. You've been using these all along: the **contentsOfFile** method for **NSString** is an initializer, as is **UIAlertController(title:message:preferredStyle:)**.

Swift is telling you that you aren't satisfying one of its core rules: objects of type **String** can't be empty. Remember, **String!** and **String?** can both be **nil**, but plain old **String** can't – it must have a value. Without an initializer, it means the object will be created and these two variables won't have values, so you're breaking the rules.

To fix this problem, we need to create an **init()** method that accepts two parameters, one for the name and one for the image. We'll then save that to the object so that both variables

have a value, and Swift is happy.

Doing this gives you the chance to learn something else: another required usage of the **self** keyword. Here's the code:

```
init(name: String, image: String) {
    self.name = name
    self.image = image
}
```

As you can see, the method takes two parameters: **name** and **image**. These are perfectly valid parameter names, but also happen to be the same names used by our class. So if we were to write something like this…

```
name = name
```

…then it would be confusing. Are you assigning the parameter to itself? Are we assigning the class's property to the parameter? To solve this problem, you use **self.** to clarify which is which, so **self.image = image** can only mean one thing: assign the parameter to the class's property.

Our custom class is done; it's just a dumb data store for now. If you're the curious type, you might wonder why I used a class here rather than a struct. This question is even more pressing once you know that structs have an automatic initializer method made for them that looks exactly like ours. Well, the answer is: you'll have to wait and see. All will become clear in project 12!

With that custom class done, we can start to make our project much more useful: every time a picture is imported, we can create a Person object for it and add it to an array to be shown in the collection view.

So, go back to ViewController.swift, and add this declaration for a new array:

```
var people = [Person]()
```

Every time we add a new person, we need to create a new **Person** object with their details. This is as easy as modifying our initial image picker success method so that it creates a **Person** object, adds it to our **people** array, then reloads the collection view. Put this code before the call to **dismissViewControllerAnimated()**:

```
let person = Person(name: "Unknown", image: imageName)
people.append(person)
collectionView.reloadData()
```

That stores the image name in the **Person** object and gives them a default name of "Unknown", before reloading the collection view.

Can you spot the problem? If not, that's OK, but you should be able to spot it if you run the program.

The problem is that although we've added the new person to our array and reloaded the collection view, we aren't actually using the **people** array with the collection view – we just return 10 for the number of items and create an empty collection view cell for each one! Let's fix that…

Connecting up the people

We need to make three final changes to this project in order to finish: show the correct number of items, show the correct information inside each cell, then make it so that when users tap a picture they can set a person's name.

Those methods are all increasingly difficult, so we'll start with the first one. Right now, your collection view's **numberOfItemsInSection** method just has **return 10** in there, so you'll see 10 items regardless of how many people are in your array. This is easily fixed:

```
func collectionView(collectionView: UICollectionView,
numberOfItemsInSection section: Int) -> Int {

    return people.count

}
```

Next, we need to update the collection view's **cellForItemAtIndexPath** method so that it configures each **PersonCell** cell to have the correct name and image of the person in that position in the array. This takes a few steps:

 • Pull out the person from the **people** array at the correct position.
 • Set the **name** label to the person's name.
 • Create a **UIImage** from the person's image filename, adding it to the value from **getDocumentsDirectory()** so that we have a full path for the image.

We're also going to use this opportunity to give the image views a border and slightly rounded corners, then give the whole cell matching rounded corners, to make it all look a bit more interesting. This is all done using **CALayer**, so that means we need to convert the **UIColor** to a **CGColor**. Anyway, here's the new code:

```
func collectionView(collectionView: UICollectionView,
cellForItemAtIndexPath indexPath: NSIndexPath) ->
UICollectionViewCell {

    let cell =
collectionView.dequeueReusableCellWithReuseIdentifier("Person",
forIndexPath: indexPath) as! PersonCell
```

```
    let person = people[indexPath.item]


    cell.name.text = person.name


    let path =
getDocumentsDirectory().stringByAppendingPathComponent(person.image)
    cell.imageView.image = UIImage(contentsOfFile: path)


    cell.imageView.layer.borderColor = UIColor(red: 0, green: 0, blue:
0, alpha: 0.3).CGColor
    cell.imageView.layer.borderWidth = 2
    cell.imageView.layer.cornerRadius = 3
    cell.layer.cornerRadius = 7


    return cell
}
```

The only new thing in there is setting the **cornerRadius** property, which rounds the corners of a **CALayer** – or in our case the **UIView** being drawn by the **CALayer**.

With that done, the app works: you can run it with Cmd+R, import photos, and admire the way they all appear correctly in the app. But don't get your hopes up, because we're not done yet – you still can't assign names to people!

For this last part of the project, we're going to recap how to add text fields to a **UIAlertController**, just like you did in project 5. All of the code is old, but I'm going to go over it again to make sure you fully understand.

First, the delegate method we're going to implement is the UICollectionView's **didSelectItemAtIndexPath** method, which is triggered when the user taps a cell. This method needs to pull out the **Person** object at the array index that was tapped, then show a **UIAlertController** asking users to rename the person.

Adding a text field to an alert controller is done with the

addTextFieldWithConfigurationHandler() method. We'll also need to add two actions: one to cancel the alert (with a **nil** handler), and one to save the change. To save the changes, we need to add a closure that pulls out the text field value and assigns it to the person's **name** property, then we'll also need to reload the collection view to reflect the change.

That's it! The only thing that's new, and it's hardly new at all, is the setting of the **name** property. Put this new method into your class:

```
func collectionView(collectionView: UICollectionView,
didSelectItemAtIndexPath indexPath: NSIndexPath) {

  let person = people[indexPath.item]


  let ac = UIAlertController(title: "Rename person", message: nil,
preferredStyle: .Alert)

  ac.addTextFieldWithConfigurationHandler(nil)


  ac.addAction(UIAlertAction(title: "Cancel", style: .Cancel,
handler: nil))


  ac.addAction(UIAlertAction(title: "OK", style: .Default) { [unowned
self, ac] _ in

    let newName = ac.textFields![0]

    person.name = newName.text!


    self.collectionView.reloadData()

  })


  presentViewController(ac, animated: true, completion: nil)

}
```

Finally, the project is complete: you can import photos of people, then tap on them to rename. Well done!

Wrap up

UICollectionView and **UITableView** are the most common ways of showing lots of information in iOS, and you now know how to use both. You should be able to go back to project 1 and recognise a lot of very similar code, and that's by intention – Apple has made it easy to learn both view types by learning either one.

You've also learned another batch of iOS development, this time **UIImagePickerController**, **NSUUID**, custom classes and more. You might not realise it yet, but you have enough knowledge now to make a huge range of apps!

If you wanted to take this app further you could add a second **UIAlertController** that is shown when the user taps a picture, and asks them whether they want to rename the person or delete them.

You could also try **picker.sourceType = .Camera** when creating your image picker, which will tell it to create a new image by taking a photo. This is only available on devices (not on the simulator!) so you might want to check the return value of **UIImagePickerController.isSourceTypeAvailable()** before trying to use it!

Before we finish, you may have spotted one problem with this app: if you quit the app and relaunch, it hasn't remembered the people you added. Worse, the JPEGs are still stored on the disk, so your app takes up more and more room without having anything to show for it!

This is quite intentional, and something we'll return to fix in project 12. Before then, let's take a look at another game…

Project 11

Pachinko

Dive into SpriteKit to try your hand at fast 2D games.

Setting up

This project is going to feel like a bit of a reset for you, because we're going to go back to basics. This isn't because I like repeating myself, but instead because you're going to learn a wholly new technology called SpriteKit.

So far, everything you've made has been based on UIKit, Apple's user interface toolkit for iOS. We've made several games with it, and it really is very powerful, but even UIKit has its limits – and fast 2D games aren't its strong suit.

A much better solution is called SpriteKit, and it's Apple's fast and easy toolkit designed specifically for 2D games. It includes sprites, fonts, physics, particle effects and more, and it's built into every iOS device. What's not to like?

So, this is going to be a long tutorial because you're going to learn an awful lot. To help keep you sane, I've tried to make the project as iterative as possible. That means we'll make a small change and discuss the results, then make another small change and discuss the results, until the project is finished.

And what are we building? Well, we're going to produce a game similar to pachinko, although a lot of people know it by the name "Peggle." To get started, create a new project in Xcode and choose Game. Name it Project11, set it to target iPad, set its Game Technology to be SpriteKit, then save it somewhere.

Before we start, please configure your project so that it runs only in landscape mode.

Warning: When working with SpriteKit projects, I strongly recommend you use the iPad 2 simulator rather than iPad Air – you'll get much faster frame rates, making it much more suitable for testing.

Falling boxes

The first thing you should do is run your game and see what a default SpriteKit template game looks like. You should see a large gray window saying "Hello, World!", and when you tap a spinning spaceship should appear. In the bottom right is a node count (how many things are on screen right now) and a frame rate. You're aiming for 60 frames per second all the time, if possible.

From the project navigator, select Images.xcassets, then choose the spaceship image and and press the delete key. Now go to GameScene.swift, and delete everything inside the **didMoveToView()** and **touchesBegan()** methods (but leave the methods themselves.)

With the template stuff deleted, I'd like you to import the assets for the project. If you haven't already downloaded the code for this project, please do so now. You should copy the entire Content folder from the example project into your own, making sure the "Copy items if needed" box is checked.

Let's kick off this project by ditching the gray background and replacing it with a picture. If you want to place an image in your game, the class to use is called **SKSpriteNode**, and it can load any picture from your app bundle just like **UIImage**.

To place a background image, we need to load the file called background.jpg, then position it in the center of the screen. Unlike UIKit, SpriteKit works with the center of objects – i.e., the point 0,0 refers to the horizontal and vertical center of a node. Also unlike UIKit, SpriteKit's Y axis starts at the bottom edge, so a higher Y number places a node higher on the screen. To place the background image in the center of a landscape iPad, we need to place it at the position X:512, Y:384.

We're going to do two more things, both of which are only needed for this background. First, we're going to give it the blend mode **.Replace**. Blend modes determine how a node is drawn, and SpriteKit gives you many options. The **.Replace** option means "just draw it, ignoring any alpha values," which makes it fast for things without gaps such as our background. We're also going to give the background a **zPosition** of -1, which in our game means "draw this behind everything else."

To add any node to the current screen, you use the **addChild()** method. As you might expect, SpriteKit doesn't use **UIViewController** like our UIKit apps have done. Yes, there is a view controller in your project, but it's there to host your SpriteKit game. The equivalent of screens in SpriteKit are called *scenes*.

When you add a node to your scene, it becomes part of the node tree. Using **addChild()** you can add nodes to other nodes to make a more complicated tree, but in this game we're going to keep it simple.

Add this code to the **didMoveToView()** method, which is sort of the equivalent of SpriteKit's **viewDidLoad()** method:

```
let background = SKSpriteNode(imageNamed: "background.jpg")

background.position = CGPoint(x: 512, y: 384)

background.blendMode = .Replace

background.zPosition = -1

addChild(background)
```

If you run the app now you'll see a dark blue image for the background rather than plain gray – hardly a massive improvement, but this is just the beginning.
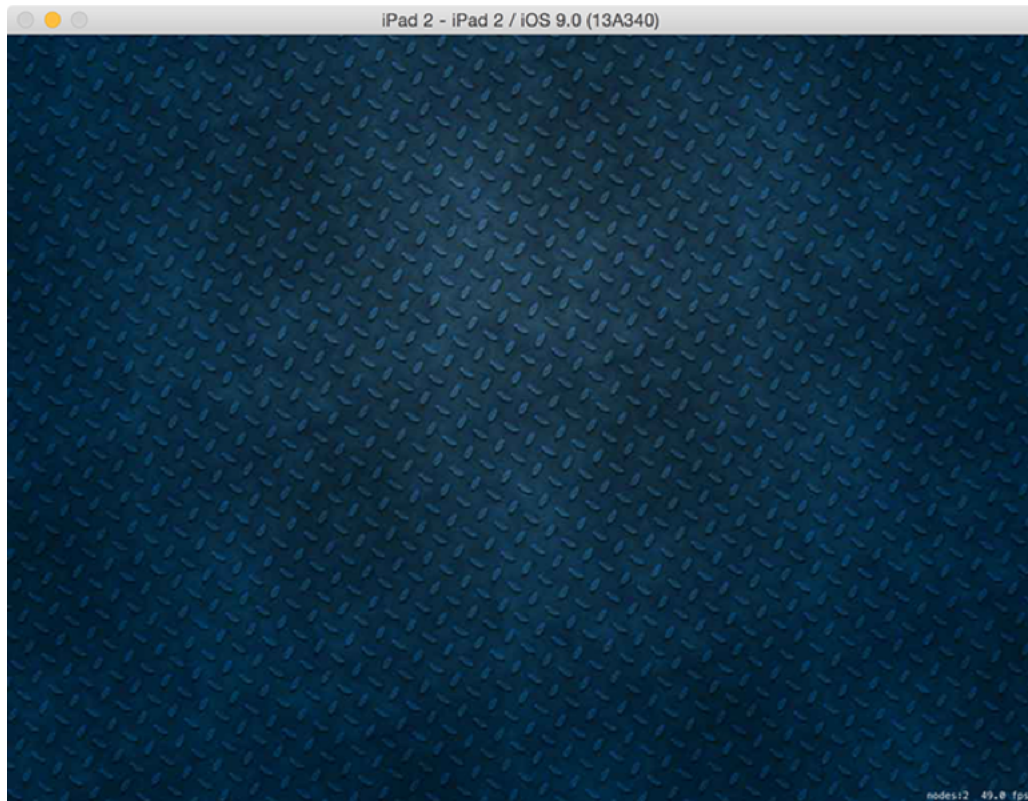
The very start of our game just has a large plain background image. Don't worry, it gets more fun – honest!

Let's do something a bit more interesting, so add this to the **touchesBegan()** method:

```
if let touch = touches.first {

  let location = touch.locationInNode(self)

  let box = SKSpriteNode(color: UIColor.redColor(), size:
CGSize(width: 64, height: 64))

  box.position = location

  addChild(box)

}
```

We haven't used **touchesBegan()** before, so the first two lines needs to be explained. This method gets called (in UIKit and SpriteKit) whenever someone startes touching their device. It's possible they started touching with multiple fingers at the same time, so we get passed a

new data type called **Set**. This is just like an array, except each object can appear only once.

We want to know where the screen was touched, so we use a conditional typecast plus if/let to pull out any of the screen touches from the **touches** set, then use its **locationInNode()** method to find out where the screen was touched in relation to "self" - i.e., the game scene. **UITouch** is a UIKit class that is also used in SpriteKit, and provides information about a touch such as its position and when it happened.

The third line is also new, but it's still **SKSpriteNode**. We're just writing some example code for now, so this line generates a node filled with a color (red) at a size (64x64). The **CGSize** struct is new, but also simple: it just holds a width and a height in a single structure.

The code sets the new box's position to be where the tap happened, then adds it to the scene. No more talk: press Cmd+R to make sure this all works, then tap around the screen to make boxes appear.

OK, I admit: that's still quite boring. Let's make it even more interesting – are you ready to see quite how powerful SpriteKit is? Just before setting the position of our new box, add this line:

```
box.physicsBody = SKPhysicsBody(rectangleOfSize: CGSize(width: 64, height: 64))
```

And just before the end of **didMoveToView()**, add this:

```
physicsBody = SKPhysicsBody(edgeLoopFromRect: frame)
```

The first line of code adds a physics body to the box that is a rectangle of the same size as the box. The second line of code adds a physics body to the whole scene that is a line on each edge, effectively acting like a container for the scene.

If you run the scene now, I hope you can't help but be impressed: you can tap on the screen to create boxes, but now they'll fall to the ground and bounce off. They'll also stack up as you tap more often, and fall over realistically if your aim isn't spot on.

This is the power of SpriteKit: it's so fast and easy to make games that there really is nothing holding you back. But we're just getting warmed up!

Bouncing balls

You're not going to get rich out of red rectangles, so let's use balls instead. Take the box code out (everything after **let location =** in **touchesBegan()**) and replace it with this instead:

```
let ball = SKSpriteNode(imageNamed: "ballRed")

ball.physicsBody = SKPhysicsBody(circleOfRadius: ball.size.width /
2.0)

ball.physicsBody!.restitution = 0.4

ball.position = location

addChild(ball)
```

There are two new things there. First, we're using the **circleOfRadius** initializer for **SKPhysicsBody** to add circular physics to this ball, because using rectangles would look strange. Second, we're giving the ball's physics body a restitution (bounciness) level of 0.4, where values are from 0 to 1.

(NB: the physics body of a node is optional, because it might not exist. We know it exists because we just created it, so you'll always see **physicsBody!** to force unwrap the optional.)

When you run the game now, you'll be able to tap on the screen to drop bouncy balls. It's fractionally more interesting, but let's face it: this is still a dreadful game.

Easily fixed, though: we're going to add something for the balls to bounce off. In the Content folder I provided you with is a picture called "bouncer.png", so we're going to place that in the game now.

Just before the end of the **didMoveToView()** method, add this:

```
let bouncer = SKSpriteNode(imageNamed: "bouncer")

bouncer.position = CGPoint(x: 512, y: 0)

bouncer.physicsBody = SKPhysicsBody(circleOfRadius:
bouncer.size.width / 2.0)

bouncer.physicsBody!.dynamic = false
```

```
addChild(bouncer)
```

There's a new data type in there called **CGPoint**, but, like **CGSize**, it's very simple: it just holds X/Y co-ordinates. Remember, SpriteKit's positions start from the center of nodes, so X: 512 Y:0 means "centered horizontally on the bottom edge of the scene."

Also new is the **dynamic** property of a physics body. When this is true, the object will be moved by the physics simulator based on gravity and collisions. When it's false (as we're setting it) the object will still collide with other things, but it won't ever be moved as a result.

Using this code, the bouncer node will be placed on the screen and your balls can collide with it – but it won't move. Give it a try now.

Adding a bouncer took five lines of code, but our game needs more than one bouncer. In fact, I want five of them, evenly distributed across the screen. Now, you *could* just copy and paste the code five times then change the positions, but I hope you realise there's a better way: creating a method that does all the work, then calling that method each time we want a bouncer.

The method code itself is nearly identical to what you just wrote, with the only change being that we need to position the box at the **CGPoint** specified as a parameter:

```
func makeBouncerAt(position: CGPoint) {

  let bouncer = SKSpriteNode(imageNamed: "bouncer")

  bouncer.position = position

  bouncer.physicsBody = SKPhysicsBody(circleOfRadius:
bouncer.size.width / 2.0)

  bouncer.physicsBody!.dynamic = false

  addChild(bouncer)

}
```

With that method in place, you can place a bouncer in one line of code: just call **makeBouncerAt()** with a position, and it will be placed and given a non-dynamic physics body automatically.

To show this off, delete the bouncer code from **didMoveToView()**, and replace it with this:

```
makeBouncerAt(CGPoint(x: 0, y: 0))
makeBouncerAt(CGPoint(x: 256, y: 0))
makeBouncerAt(CGPoint(x: 512, y: 0))
makeBouncerAt(CGPoint(x: 768, y: 0))
makeBouncerAt(CGPoint(x: 1024, y: 0))
```

If you run the game now you'll see five bouncers evenly spread across the screen, and the balls you drop bounce off any of them. It's still not a game, but we're getting there. Now to add something between the bouncers…

Spinning slots

The purpose of the game will be to drop your balls in such a way that they land in good slots and not bad ones. We have bouncers in place, but we need to fill the gaps between them with something so the player knows where to aim.

We'll be filling the gaps with two types of target slots: good ones (colored green) and bad ones (colored red). As with bouncers, we'll need to place a few of these, which means we need to make a method. This needs to load the slot base graphic, position it where we said, then add it to the scene, like this:

```
func makeSlotAt(position: CGPoint, isGood: Bool) {

  var slotBase: SKSpriteNode


  if isGood {

    slotBase = SKSpriteNode(imageNamed: "slotBaseGood")

  } else {

    slotBase = SKSpriteNode(imageNamed: "slotBaseBad")

  }


  slotBase.position = position

  addChild(slotBase)

}
```

Unlike **makeBouncerAt()**, this method has a second parameter – whether the slot is good or not – and that affects which image gets loaded. But first, we need to call the new method, so add these lines just before the calls to **makeBouncerAt** in **didMoveToView()**:

```
makeSlotAt(CGPoint(x: 128, y: 0), isGood: true)

makeSlotAt(CGPoint(x: 384, y: 0), isGood: false)

makeSlotAt(CGPoint(x: 640, y: 0), isGood: true)

makeSlotAt(CGPoint(x: 896, y:0), isGood: false)
```

The X positions are exactly between the bouncers, so if you run the game now you'll see bouncer / slot / bouncer / slot and so on.

One of the obvious-but-nice things about using methods to create the bouncers and slots is that if we want to change the way slots look we only need to change it in one place. For example, we can make the slot colors look more obvious by adding a glow image behind them:

```
func makeSlotAt(position: CGPoint, isGood: Bool) {
  var slotBase: SKSpriteNode
  var slotGlow: SKSpriteNode

  if isGood {
    slotBase = SKSpriteNode(imageNamed: "slotBaseGood")
    slotGlow = SKSpriteNode(imageNamed: "slotGlowGood")
  } else {
    slotBase = SKSpriteNode(imageNamed: "slotBaseBad")
    slotGlow = SKSpriteNode(imageNamed: "slotGlowBad")
  }

  slotBase.position = position
  slotGlow.position = position

  addChild(slotBase)
  addChild(slotGlow)
}
```

That basically doubles every line of code, changing "Base" to "Glow", but the end result is quite pleasing and it's clear now which slots are good and which are bad.

We could even make the slots spin slowly by using a new class called **SKAction**. SpriteKit actions are ridiculously powerful and we're going to do some great things with them in later projects, but for now we just want the glow to rotate very gently.

Before we look at the code to make this happen, you need to learn a few things up front:

 • Angles are specified in radians, not degrees. This is true in UIKit too. 360 degrees is equal to the value of 2 x Pi – that is, the mathematical value π. Therefore π radians is equal to 180 degrees.
 • Rather than have you try to memorise it, there is a built-in value of π called **M_PI**. Problem is, it has the data type **Double**, and SpriteKit likes to use the data type **CGFloat**.
 • Yes **CGFloat** is yet another way of representing decimal numbers, just like **Double** and **Float**. So, if you want to use **M_PI** in SpriteKit you need to create a **CGFloat** out of it. Lovely.
 • When you create an action it will execute once. If you want it to run forever, you create another action to wrap the first using the **repeatActionForever()** method, then run that.

Our new code will rotate the node by 90 degrees (available as the constant **M_PI_2**) over 10 seconds, repeating forever. Put this code just before the end of the **makeSlotAt()** method:

```
let spin = SKAction.rotateByAngle(CGFloat(M_PI_2), duration: 10)
let spinForever = SKAction.repeatActionForever(spin)
slotGlow.runAction(spinForever)
```

If you run the game now, you'll see that the glow spins around very gently. It's a simple effect, but it makes a big difference.

With bouncers and slots now in place, this pachinko game is starting to come together.

Collision detection

Just by adding a physics body to the balls and bouncers we already have some collision detection because the objects bounce off each other. But it's not being detected by *us*, which means we can't do anything about it.

In this game, we want the player to win or lose depending on how many green or red slots they hit, so we need to make a few changes:

1. Add rectangle physics to our slots.
2. Name the slots so we know which is which, then name the balls too.
3. Make our scene the contact delegate of the physics world – this means, "tell us when contact occurs between two bodies."
4. Create a method that handles contacts and does something appropriate.

The first step is easy enough: add these two lines just before you call **addChild()** for **slotBase**:

```
slotBase.physicsBody = SKPhysicsBody(rectangleOfSize: slotBase.size)
slotBase.physicsBody!.dynamic = false
```

The slot base needs to be non-dynamic because we don't want it to move out of the way when a player ball hits.

The second step is also easy, but bears some explanation. As with UIKit, it's easy enough to store a variable pointing at specific nodes in your scene for when you want to make something happen, and there are lots of times when that's the right solution.

But for general use, Apple recommends assigning names to your nodes, then checking the name to see what node it is. We need to have three names in our code: good slots, bad slots and balls. This is really easy to do – just modify your **makeSlotAt()** method so the **SKSpriteNode** creation looks like this:

```
if isGood {
  slotBase = SKSpriteNode(imageNamed: "slotBaseGood")
```

```
    slotGlow = SKSpriteNode(imageNamed: "slotGlowGood")

    slotBase.name = "good"

} else {

    slotBase = SKSpriteNode(imageNamed: "slotBaseBad")

    slotGlow = SKSpriteNode(imageNamed: "slotGlowBad")

    slotBase.name = "bad"

}
```

Then add this to the code where you create the balls:

```
ball.name = "ball"
```

We don't need to name the bouncers, because we don't actually care when their collisions happen.

Now comes the tricky part, which is setting up our scene to be the contact delegate of the physics world. The initial change is easy: we just need to conform to the **SKPhysicsContactDelegate** protocol then assign the physics world's **contactDelegate** property to be our scene. But by default, you still won't get notified when things collide.

What we need to do is change the **contactTestBitMask** property of our physics objects, which sets the contact notifications we want to receive. This needs to introduce a whole new concept – bitmasks – and really it doesn't matter at this point, so we're going to take a shortcut for now, then return to it in a later project.

Let's set up all the contact delegates and bitmasks now. First, make your class conform to the **SKPhysicsContactDelegate** protocol by modifying its definition to this:

```
class GameScene: SKScene, SKPhysicsContactDelegate {
```

Then assign the current scene to be the physics world's contact delegate by putting this line of code in **didMoveToView()**, just below where we set the scene's physics body:

```
physicsWorld.contactDelegate = self
```

Now for our shortcut: we're going to tell the slot base node and the all the ball nodes to set their **contactTestBitMask** property to be equal to their **collisionBitMask**. Two bitmasks, with confusingly similar names but quite different jobs.

The **collisionBitMask** bitmask means "which nodes should I bump into?" By default, it's set to everything, which is why our ball are already hitting each other and the bouncers. The **contactTestBitMask** bitmask means "which collisions do you want to know about?" and by default it's set to nothing. So by setting **contactTestBitMask** to the value of **collisionBitMask** we're saying, "tell me about every collision."

This isn't particularly efficient in complicated games, but it will make no difference at all in this current project. And, like I said, we'll return to this in a later project to explain more. Until then, add this line just before you set each ball's **restitution** property:

```
ball.physicsBody!.contactTestBitMask =
ball.physicsBody!.collisionBitMask
```

And put this line of code directly before setting the **dynamic** property of the bouncers in **makeBouncerAt()**:

```
bouncer.physicsBody!.contactTestBitMask =
bouncer.physicsBody!.collisionBitMask
```

Those are all the configuration changes required for us to detect collisions, so now it's time to write the code that does the hard work.

But first, a little explanation: when contact between two physics bodies occurs, we don't know what order it will come in. That is, did the ball hit the slot, or did the slot hit the ball? I know this sounds like pointless philosophising, but it's important because we need to know which one is the ball!

Before looking at the actual contact method, I want to look at two other methods first, because this is our ultimate goal. The first one, **collisionBetweenBall()** will be called when a ball collides with something else. The second one, **destroyBall()** is going to be called when we're finished with the ball and want to get rid of it.

Put these new methods into to your code:

```
func collisionBetweenBall(ball: SKNode, object: SKNode) {
  if object.name == "good" {
    destroyBall(ball)
  } else if object.name == "bad" {
    destroyBall(ball)
  }
}


func destroyBall(ball: SKNode) {
  ball.removeFromParent()
}
```

The **removeFromParent()** method removes a node from your node tree. Or, in plain English, it removes the node from your game.

You might look at that and think it's utterly redundant, because no matter what happens it's effectively the same as writing this:

```
func collisionBetweenBall(ball: SKNode, object: SKNode) {
  ball.removeFromParent()
}
```

But trust me on this: we're going to make these methods do more shortly, so get it right now

and it will save refactoring later.

With those two in place, our contact checking method almost writes itself. We'll get told which two bodies collided, and the contact method needs to determine which one is the ball so that it can call **collisionBetweenBall()** with the correct parameters. This is as simple as checking the names of both properties to see which is the ball, so here's the new method to do contact checking:

```
func didBeginContact(contact: SKPhysicsContact) {

  if contact.bodyA.node!.name == "ball" {

    collisionBetweenBall(contact.bodyA.node!, object:
contact.bodyB.node!)

  } else if contact.bodyB.node!.name == "ball" {

    collisionBetweenBall(contact.bodyB.node!, object:
contact.bodyA.node!)

  }

}
```

If you're particularly observant, you may have noticed that we don't have a special case in there for when both bodies are balls – i.e., if one ball collides with another. This is because our **collisionBetweenBall()** method will ignore that particular case, because it triggers code only if the other node is named "good" or "bad".

Run the game now and you'll start to see things coming together: you can drop balls on the bouncers and they will bounce, but if they touch one of the good or bad slots the balls will be destroyed. It works, but it's boring. Players want to score points so they feel like they achieved something, even if that "something" is just nudging up a number on a CPU.

Scores on the board

To make a score show on the screen we need to do two things: create a score integer that tracks the value itself, then create a new node type, **SKLabelNode**, that displays the value to players.

The **SKLabelNode** class is somewhat similar to **UILabel** in that it has a **text** property, a font, a position, an alignment, and so on. Plus we can use Swift's string interpolation to set the text of the label easily, and we're even going to use the property observers you learned about it project 8 to make the label update itself when the score value changes.

Declare these properties at the top of your class:

```
var scoreLabel: SKLabelNode!

var score: Int = 0 {
  didSet {
    scoreLabel.text = "Score: \(score)"
  }
}
```

We're going to use the Chalkduster font, then align the label to the right and position it on the top-right edge of the scene. Put this code into your **didMoveToView()** method, just before the end:

```
scoreLabel = SKLabelNode(fontNamed: "Chalkduster")
scoreLabel.text = "Score: 0"
scoreLabel.horizontalAlignmentMode = .Right
scoreLabel.position = CGPoint(x: 980, y: 700)
addChild(scoreLabel)
```

That places the label into the scene, and the property observer automatically updates the

label as the **score** value changes. But it's not complete yet because we don't ever modify the player's score. Fortunately, we already have places in the **collisionBetweenBall()** method where we can do exactly that, so modify the method to this:

```
func collisionBetweenBall(ball: SKNode, object: SKNode) {
    if object.name == "good" {
        destroyBall(ball)
        score += 1
    } else if object.name == "bad" {
        destroyBall(ball)
        score -= 1
    }
}
```

The **+=** and **-=** operators add or subtract one to the variable depending on whether a good or bad slot was struck. When we change the variable, the property observer will spot the change and update the label.

We have a score, so that means players have the achievement they were craving, right? Well, no. Clearly all it takes to get a number even higher than Gangnam Style's YouTube views is to sit and tap at the top of the screen directly above a green slot.

Let's add some actual challenge: we're going to let you place obstacles between the top of the scene and the slots at the bottom, so that players have to position their balls exactly correctly to bounce off things in the right ways.

To make this work, we're going to add two more properties. The first one will hold a label that says either "Edit" or "Done", and one to hold a boolean that tracks whether we're in editing mode or not. Add these two alongside the score properties from earlier:

```
var editLabel: SKLabelNode!

var editingMode: Bool = false {
```

```
    didSet {
      if editingMode {
        editLabel.text = "Done"
      } else {
        editLabel.text = "Edit"
      }
    }
  }
}
```

Then add this to **didMoveToView()** to create the edit label in the top-left corner of the scene:

```
editLabel = SKLabelNode(fontNamed: "Chalkduster")
editLabel.text = "Edit"
editLabel.position = CGPoint(x: 80, y: 700)
addChild(editLabel)
```

That's pretty much identical to creating the score label, so nothing to see here. We're using a property observer again to automatically change the editing label's text when edit mode is changed.

But what *is* new is detecting whether the user tapped the edit/done button or is trying to create a ball. To make this work, we're going to ask SpriteKit to give us a list of all the nodes at the point that was tapped, and check whether it contains our edit label. If it does, we'll flip the value of our **editingMode** boolean; if it doesn't, we want to execute the previous ball-creation code.

We're going to insert this change just after **let location =** and before **let ball =**, i.e. right here:

```
let location = touch.locationInNode(self)
// new code to go here!
let ball = SKSpriteNode(imageNamed: "ballRed")
```

Change that to be:

```
let location = touch.locationInNode(self)

let objects = nodesAtPoint(location) as [SKNode]

if objects.contains(editLabel) {
  editingMode = !editingMode
} else {
  let ball = SKSpriteNode(imageNamed: "ballRed")
  // rest of ball code
}
```

Obviously the **// rest of ball code** comment is where the rest of the ball-creating code goes, but note that you need to add the new closing brace after you've created the ball, to close the **else** block.

This uses the **!** operator that you met in project 5 to mean "set **editingMode** to be the opposite of whatever it is right now." That change will be picked up by the property observer, and the label will be updated to reflect the change.

Now that we have a boolean telling us whether we're in editing mode or not, we're going to extend **touchesBegan()** even further so that if we're in editing mode we add blocks to the screen of random sizes, and if we're not it drops a ball.

To get the structure right, this is what you want to have:

```
if objects.contains(editLabel) {
  editingMode = !editingMode
} else {
```

```
  if editingMode {
    // create a box
  } else {
    // create a ball
  }
}
```

The **// create a ball** comment is where your current ball creation code goes. The **// create a box** comment is what we're going to write in just a moment.

First, look in the code you downloaded for this project, and copy the file Helper.swift into your own project, making sure (as always) "Copy items if needed" is checked. This gives you a function to generate random colors, but I'm not going to explain how it works work because it doesn't matter – we're just going to use it.

Second, we're going to use a new property on nodes called **zRotation**. When creating the background image, we gave it a Z position, which adjusts its depth on the screen, front to back. If you imagine sticking a skewer through the Z position – i.e., going directly into your screen – and through a node, then you can imagine Z rotation: it rotates a node on the screen as if it had been skewered straight through the screen.

Replace the **// create a box** comment with this:

```
let size = CGSize(width: GKRandomDistribution(lowestValue: 16,
highestValue: 128).nextInt(), height: 16)

let box = SKSpriteNode(color: RandomColor(), size: size)

box.zRotation = RandomCGFloat(min: 0, max: 3)

box.position = location


box.physicsBody = SKPhysicsBody(rectangleOfSize: box.size)

box.physicsBody!.dynamic = false


addChild(box)
```

Before I explain a key new line of code, I want you to add this line near the top of your file, just above where it says **import UIKit**:

```
import GameplayKit
```

We've been using GameplayKit so far to handle array shuffling, but in the code above it's also being used to generate random numbers between two values: 16 and 128. So, we create a size with a height of 16 and a width between 16 and 128, then create an **SKSpriteNode** with the random size we made along with a random color, then give the new box a random rotation and place it at the location that was tapped on the screen. For a physics body, it's just a rectangle, but we need to make it non-dynamic so the boxes don't move when hit.

At this point, we almost have a game: you can tap Edit, place as many blocks as you want, then tap Done and try to score by dropping balls. It's not perfect because we don't force the Y position of new balls to be the top of the screen, but that's something you can fix yourself – how else would you learn, right?
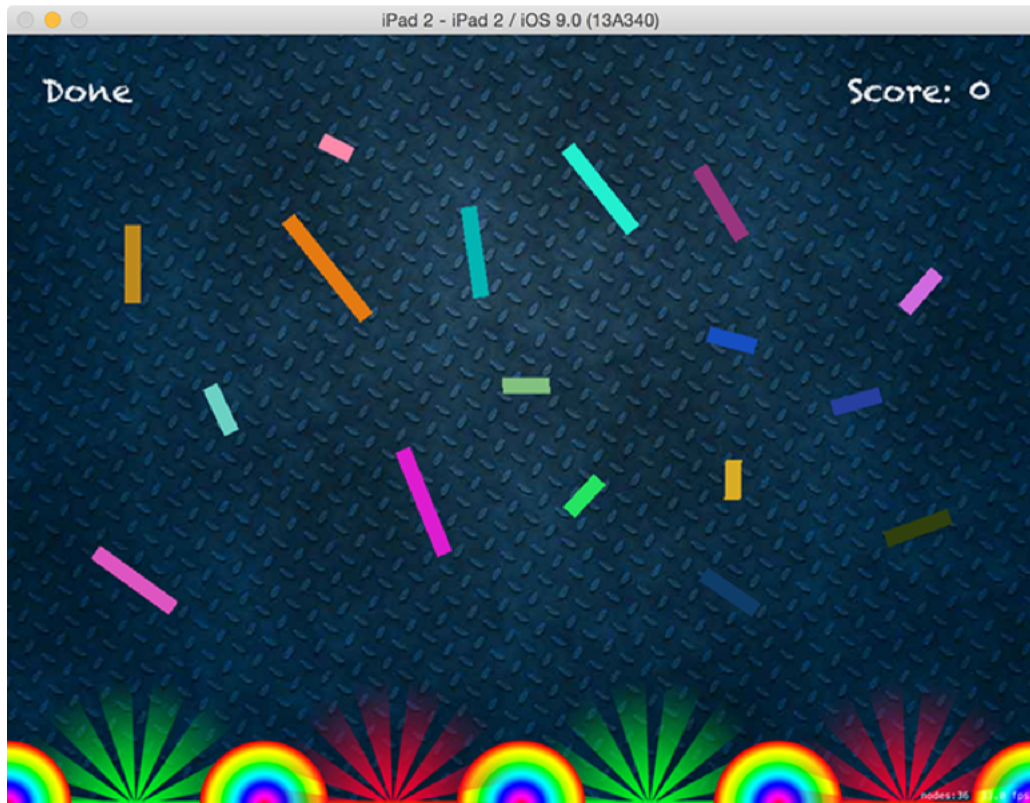
Once the edit button has been tapped, users can create as many obstacles as they want.

Special effects

Our current **destroyBall()** method does nothing much, it just removes the ball from the game. But I made it a method for a reason, and that's so that we can add some special effects now, in one place, so that however a ball gets destroyed the same special effects are used.

Perhaps unsurprisingly, it's remarkably easy to create special effects with SpriteKit. In fact, it has a built-in particle editor to help you create effects like fire, snow, rain and smoke almost entirely through a graphical editor. I already created an example particle effect for you (which you can customise soon, don't worry!) so let's take a look at the code first.

Modify your **destroyBall()** method to this:

```
func destroyBall(ball: SKNode) {
  if let fireParticles = SKEmitterNode(fileNamed: "FireParticles") {
    fireParticles.position = ball.position
    addChild(fireParticles)
  }


  ball.removeFromParent()
}
```

The **SKEmitterNode** class is new and powerful: it's designed to create high-performance particle effects in SpriteKit games, and all you need to do is provide it with the filename of the particles you designed and it will do the rest. Once we have an **SKEmitterNode** object to work with, we position it where the ball was then use **addChild()** to add it to the scene.

If you run the app now, you'll see the balls explode in a fireball when they touch a slot – a pretty darn amazing effect given how little code was written!

But the real fun is yet to come, because the code for this project is now all done and you get to play with the particle editor. In Xcode, look in the Content folder you dragged in and select the FireParticles.sks file to load the particle editor.

The particle editor is split in two: the center area shows how the current particle effect looks,

and the right pane shows one of three inspectors. Of those three inspectors, only the third is useful because that's where you'll see all the options you can use to change the way your particles look.

At the time of writing, Xcode's particle editor is a little buggy, so I suggest you change the Maximum value to 0 before beginning otherwise you might see nothing at all.

Confused by all the options? Here's what they do:

 • Particle Texture: what image to use for your particles.
 • Particles Birthrate: how fast to create new particles.
 • Particles Maximum: the maximum number of particles this emitter should create before finishing.
 • Lifetime Start: the basic value for how many seconds each particle should live for.
 • Lifetime Range: how much, plus or minus, to vary lifetime.
 • Position Range X/Y: how much to vary the creation position of particles from the emitter node's position.
 • Angle Start: which angle you want to fire particles, in degrees, where 0 is to the right and 90 is straight up.
 • Angle Range: how many degrees to randomly vary particle angle.
 • Speed Start: how fast each particle should move in its direction.
 • Speed Range: how much to randomly vary particle speed.
 • Acceleration X/Y: how much to affect particle speed over time. This can be used to simulate gravity or wind.
 • Alpha Start: how transparent particles are when created.
 • Alpha Range: how much to randomly vary particle transparency.
 • Alpha Speed: how much to change particle transparency over time. A negative value means "fade out."
 • Scale Start / Range / Speed: how big particles should be when created, how much to vary it, and how much it should change over time. A negative value means "shrink slowly."
 • Rotation Start / Range / Speed: what Z rotation particles should have, how much to vary it, and how much they should spin over time.
 • Color Blend Factor / Range / Speed: how much to color each particle, how much to vary it, and how much it should change over time.

(PS: Once you've finished editing your particles, make sure you put a maximum value back on them otherwise they'll never go away!)

It's worth adding that you can create particles from one of Xcode's built-in particle template.

Add a new file, but this time choose "Resource" under the iOS heading, then choose "SpriteKit Particle File" to see the list of options.

Wrap up

This project is done, and it's been a long one, but I hope you look at the results and think it was all worth it. Plus, you've once again learned a lot: SpriteKit, physics, blend modes, radians and **CGFloat**.

You've got the firm foundations of a real game here, but there's lots more you can do to make it even better. Here are some ideas to get you started:

 • The Content folder you copied in has other ball pictures rather than just ballRed. Generate a random number and choose ball colors randomly.
 • Right now, users can tap anywhere to have a ball created there, which makes the game too easy. Try to force the Y value of new balls so they are near the top of the screen.
 • Give players a limit of five balls, then remove obstacle boxes when they are hit. Can they clear all the pins with just five balls? You could make it so that landing on a green slot gets them an extra ball.
 • Make clicking on an obstacle box in editing mode removes it.

And if you were wondering how to get rid of the node and frames per second counts in your game, look inside the GameViewController.swift file for these two lines:

```
skView.showsFPS = true

skView.showsNodeCount = true
```

Project 12

NSUserDefaults

Learn how to save user settings and data for later use.

Setting up

This is our fourth technique project, and we're going to go back to project 10 and fix its glaring bug: all the names and faces you add to the app don't get saved, which makes the app rather pointless!

We're going to fix this using a new class called **NSUserDefaults** and a new protocol called **NSCoding**. We'll also be using the class **NSKeyedUnarchiver** that you just met in project 11, along with its counterpart: **NSKeyedArchiver**. Putting all these together, we're going to update project 10 so that it saves its **people** array whenever anything is changed, then loads when the app runs.

We're going to be modifying project 10, so if you want to preserve the old code take a copy now and call it project 12.

Reading and writing basics

You can use **NSUserDefaults** to store any basic data type for as long as the app is installed. You can write basic types such as **Bool**, **Float**, **Double**, **Int**, **String**, or **NSURL**, but you can also write more complex types such as arrays, dictionaries and **NSDate** – and even **NSData** values.

When you write data to **NSUserDefaults**, it automatically gets loaded when your app runs so that you can read it back again. This makes using it really easy, but you need to know that it's a bad idea to store lots of data in there because it will slow loading of your app. If you think your saved data would take up more than say 100KB, **NSUserDefaults** is almost certainly the wrong choice.

Before we get into modifying project 12, we're going to do a little bit of test coding first to try out what **NSUserDefaults** lets us do. You might find it useful to create a fresh Single View Application project just so you can test out the code.

To get started with **NSUserDefaults**, you create a new instance of the class like this:

```
let defaults = NSUserDefaults.standardUserDefaults()
```

Once that's done, it's easy to set a variety of values – you just need to give each one a unique key so you can reference it later. These values nearly always have no meaning outside of what you use them for, so just make sure the key names are memorable.

Here are some examples:

```
let defaults = NSUserDefaults.standardUserDefaults()
defaults.setInteger(25, forKey: "Age")
defaults.setBool(true, forKey: "UseTouchID")
defaults.setDouble(M_PI, forKey: "Pi")
```

In older versions of iOS, you needed to tell iOS when it was a good time to save the defaults data to disk, but this isn't needed (or even recommended!) any more.

After that, you should use the **setObject()** to set strings, arrays, dictionaries and dates. Now, here's a curiosity that's worth explaining briefly: in Swift, strings, arrays and dictionaries are all structs, not objects. But **NSUserDefaults** was written for **NSString** and friends – all of whom are 100% interchangeable with Swift their equivalents – which is why this code works.

Using **setObject()** is just the same as using other data types:

```
defaults.setObject("Paul Hudson", forKey: "Name")
defaults.setObject(NSDate(), forKey: "LastRun")
```

Even if you're trying to save complex types such as arrays and dictionaries, **NSUserDefaults** laps it up:

```
let array = ["Hello", "World"]
defaults.setObject(array, forKey: "SavedArray")


let dict = ["Name": "Paul", "Country": "UK"]
defaults.setObject(dict, forKey: "SavedDict")
```

That's enough about writing for now; let's take a look at reading.

When you're reading values from **NSUserDefaults** you need to check the return type carefully to ensure you know what you're getting. Here's what you need to know:

- **integerForKey()** returns an integer if the key existed, or 0 if not.
- **boolForKey()** returns a boolean if the key existed, or false if not.
- **floatForKey()** returns a float if the key existed, or 0.0 if not.
- **doubleForKey()** returns a double if the key existed, or 0.0 if not.
- **objectForKey()** returns **AnyObject?** so you need to conditionally typecast it to your data type.

Knowing the return values are important, because if you use **boolForKey()** and get back

"false", does that mean the key didn't exist, or did it perhaps exist and you just set it to be false?

It's **objectForKey()** that will cause you the most bother, because you get an optional object back. You're faced with two options, one of which isn't smart so you realistically have only one option!

Your options:

 • Use **as** to typecast your object to the data type it should be. This worked in Xcode 6.2 or earlier.
 • Use **as!** to force typecast your object to the data type it should be. This is available from Xcode 6.3 or later.
 • Use **as?** to optionally typecast your object to the type it should be.

If you use **as**/**as!** and **objectForKey()** returned **nil**, you'll get a crash, so I really don't recommend it unless you're absolutely sure. But equally, using **as?** is annoying because you then have to unwrap the optional or create a default value.

There is a solution here, and it has the catchy name of the **nil** *coalescing operator*, and it looks like **??**. This does two things at once: if the object on the left is optional and exists, it gets unwrapped into a non-optional value; if it does not exist, it uses the value on the right instead. This means we can use **objectForKey()** and **as?** to get an optional object, then use **??** to either unwrap the object or set a default value, all in one line.

For example, let's say we want to read the array we saved earlier with the key name **SavedArray**. Here's how to do that with the **nil** coalescing operator:

```
let array = defaults.objectForKey("SavedArray") as? [String] ??
[String]()
```

So, if **SavedArray** exists and is a string array, it will be placed into the **array** constant. If it doesn't exist (or if it does exist and isn't a string array), then **array** gets set to be a new string array.

This technique also works for dictionaries, but obviously you need to typecast it correctly. To read the dictionary we saved earlier, we'd use this:

```
let dict = defaults.objectForKey("SavedDict") as? [String: String] ??
[String: String]()
```

Fixing Project 10

You've just learned all the core basics of working with **NSUserDefaults**, but we're just getting started. You see, above and beyond integers, dates, strings, arrays and so on, you can also save any kind of data inside **NSUserDefaults** as long as you follow some rules.

What happens is simple: you use the **archivedDataWithRootObject()** method of **NSKeyedArchiver**, which turns an object graph into an **NSData** object, then write that to **NSUserDefaults** as if it were any other object. If you were wondering, "object graph" means "your object, plus any objects it refers to, plus any objects those objects refer to, and so on."

The rules are very simple:

1. All your data types must be one of the following: boolean, integer, float, double, string, array, dictionary, **NSDate**, or a class that fits rule 2.
2. If your data type is a class, it must conform to the **NSCoding** protocol, which is used for archiving object graphs.
3. If your data type is an array or dictionary, all the keys and values must match rule 1 or rule 2.

Many of Apple's own classes support **NSCoding**, including but not limited to: **UIColor**, **UIImage**, **UIView**, **UILabel**, **UIImageView**, **UITableView**, **SKSpriteNode** and many more. But your own classes do not, at least not by default. If we want to save the **people** array to **NSUserDefault** we'll need to conform to the **NSCoding** protocol.

The first step is to modify your **Person** class to this:

```
class Person: NSObject, NSCoding {
```

When we were working on this code in project 10, there were two outstanding questions:

 • Why do we need a class here when a struct will do just as well? (And in fact better, because structs come with a default initializer!)
 • Why do we need to inherit from **NSObject**?

It's time for the answers to become clear. You see, working with **NSCoding** requires you to use objects, or, in the case of strings, arrays and dictionaries, structs that are

interchangeable with objects. If we made the **Person** class into a struct, we couldn't use it with **NSCoding**.

The reason we inherit from **NSObject** is again because it's required to use **NSCoding** – although cunningly Swift won't mention that to you, your app will just crash.

Once you conform to the **NSCoding** protocol, you'll get compiler errors because the protocol requires you to implement two methods: a new initializer and **encodeWithCoder()**.

We need to write some more code to fix the problems, and although the code is very similar to what you've already seen in **NSUserDefaults**, it has two new things you need to know about.

First, you'll be using a new class called **NSCoder**. This is responsible for both encoding (writing) and decoding (reading) your data so that it can be used with **NSUserDefaults**.

Second, the new initializer must be declared with the **required** keyword. This means "if anyone tries to subclass this class, they are required to implement this method." An alternative to using **required** is to declare that your class can never be subclassed, known as a *final class*, in which case you don't need **required** because subclassing isn't possible. We'll be using **required** here.

Add these two methods to the **Person** class:

```
required init(coder aDecoder: NSCoder) {

  name = aDecoder.decodeObjectForKey("name") as! String

  image = aDecoder.decodeObjectForKey("image") as! String

}


func encodeWithCoder(aCoder: NSCoder) {

  aCoder.encodeObject(name, forKey: "name")

  aCoder.encodeObject(image, forKey: "image")

}
```

The initializer is used when loading objects of this class, and **encodeWithCoder()** is used when saving. The code is very similar to using **NSUserDefaults**, but I'm typecasting both values using **as** because I saved the data so I know what I'm loading.

With those changes, the **Person** class now conforms to **NSCoding**, so we can go back to ViewController.swift and add code to load and save the **people** array.

Let's start with writing, because once you understand that the reading code will make much more sense. As I said earlier, you can write **NSData** objects to **NSUserDefaults**, but we don't currently have an **NSData** object – we just have an array.

Fortunately, the **archivedDataWithRootObject()** method of **NSKeyedArchiver** turns an object graph into an **NSData** object using those **NSCoding** methods we just added to our class. Because we make changes to the array by adding people or by renaming them, let's create a single **save()** method we can use anywhere that's needed:

```
func save() {

  let savedData = NSKeyedArchiver.archivedDataWithRootObject(people)

  let defaults = NSUserDefaults.standardUserDefaults()

  defaults.setObject(savedData, forKey: "people")

}
```

So: line 1 is what converts our array into an **NSData** object, then lines 2 and 3 save that data object to **NSUserDefaults**. You now just need to call that **save()** method when we change a person's name or when we import a new picture.

You need to modify our collection view's **didSelectItemAtIndexPath** method so that you call **self.save()** just after calling **self.collectionView.reloadData()**. Both times the **self** is required because we're inside a closure. You then need to modify the image picker's **didFinishPickingMediaWithInfo** method so that it calls **save()** just before the end of the method.

And that's it – we only change the data in two places, and both now have a call to **save()**.

Finally, we need to load the array back from disk when the app runs, so add this code to

viewDidLoad():

```
let defaults = NSUserDefaults.standardUserDefaults()


if let savedPeople = defaults.objectForKey("people") as? NSData {

  people = NSKeyedUnarchiver.unarchiveObjectWithData(savedPeople) as!
[Person]

}
```

This code is effectively the **save()** method in reverse: we use the **objectForKey()** method to pull out an optional **NSData**, using if/let and **as?** to unwrap it. We then give that to the **unarchiveObjectWithData()** method of **NSKeyedUnarchiver** to convert it back to an object graph – i.e., our array of **Person** objects.

Wrap up

You *will* use **NSUserDefaults** in your projects. That isn't some sort of command, just a statement of inevitability. If you want to save any user settings, or if you want to save program settings, it's just the best place for it. And I hope you'll agree it is (continuing a trend!) easy to use and flexible, particularly when your own classes conform to **NSCoding**.

One proviso you ought to be aware of: please don't consider **NSUserDefaults** to be safe, because it isn't. If you have user information that is private, you should consider writing to the keychain instead – something we'll look at in project 28.

Project 13

Instafilter

Make a photo manipulation program using Core Image filters and a UISlider.

Setting up

In project 10 you learned how to use **UIImagePickerController** to select and import a picture from your user's photo library. In this project, we're going to add the reverse: writing images back to the photo library. But because you're here to learn as much as possible, I'm also going to introduce you to another UIKit component, **UISlider**, and also a little bit of Core Image, which is Apple's high-speed image manipulation toolkit.

The project we're going to make will let users choose a picture from their photos, then manipulate it with a series of Core Image filters. Once they are happy, they can save the processed image back to their photo library.

To get started, create a new Single View Application project in Xcode and name it Project13. You can target iPad, iPhone or Universal – whichever you feel like.

Designing the interface

Select your Main.storyboard file to open Interface Builder, then embed the view controller inside a navigation controller.

Bring up the object library, then search for "UIView" and drag a view into your controller. Give it a width of 600 and height of 430, with X:0 and Y:64. This should place it just below the navigation controller, occupying most of the screen. In the attributes inspector, give the view the background color "Dark Gray Color".

Create an image view, and place it inside the view you just created. I'd like you to indent it by 10 points on every side – i.e., width 580, height 410, X:10, Y:10. Change the image's view mode from "Scale to fill" to "Aspect Fit". Don't place any more views inside the gray view – everything else should be placed directly on the main (white) view.

That's the top part of the UI complete. For the bottom part, start by creating a label with width 70, height 20, X:28, Y:512. Give it the text "Intensity" and make it right-aligned. Now drop a slider next to it, giving it width 470, X:104, Y:508. You can't adjust the height for sliders, so leave it at the default.

Finally, place two buttons. The first button should be 120 wide and 44 high, with X:16, Y:540. Give it the title "Change Filter". The second button should be 60 wide by 44 high, with X:524, Y:540. Give it the title "Save".

In the picture below you can see how your finished layout should look.
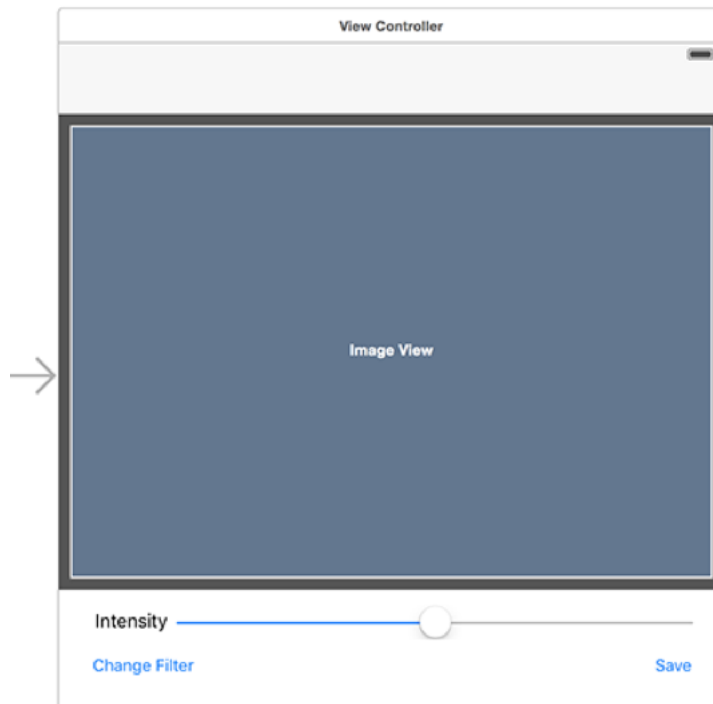
Your finished game layout should look like this.

So that's the basic layout complete, but of course we need to add Auto Layout constraints because we need it all to resize smoothly on various devices. But, you know, I'm feeling lazy – how about we make Auto Layout do the work for us this time?

Select the view controller, either by clicking on "View Controller" in the document outline, or by clicking the small yellow and white button just above the view controller on the canvas – it's the first icon of three. Now go to the Editor menu and choose Resolve Auto Layout Issues > Add Missing Constraints.
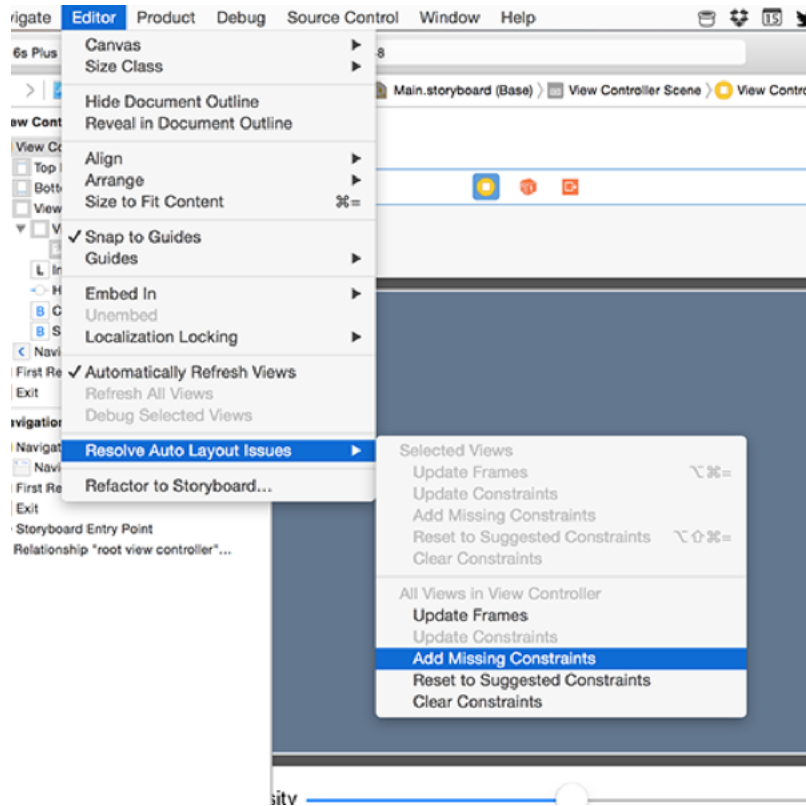
Using Add Missing Constraints can be a real time saver when using Auto Layout.

Er… that's it. Your Auto Layout is done: Xcode just added the ideal constraints everywhere so that your interface resizes perfectly. Don't believe me? Try giving the image view a red background color (temporarily!), then launching it in any device and rotating the screen. You should see everything (including the red box) be positioned and resized correctly.

Make sure you switch the image view back to having a clear background color.

That was remarkably easy, and is another example of Apple doing a lot of hard work for you. Using Xcode to make your Auto Layout rules can be a real help, but it won't be right all the time. After all, it just takes its best guess as to your intentions. It will also frequently add more constraints than strictly necessary for the job, so use it with care.

Before we leave Interface Builder, I'd like you to add an outlet for the image view and the slider, called respectively **imageView** and **intensity**. Please also create actions from the two buttons, calling methods **changeFilter()** and **save()**. You can leave these methods with no code inside them for now.

Finally, we want the user interface to update when the slider is dragged, so please create an action from the slider. It should give you the "Value Changed" event rather than Touch Up Inside, and that's what we want. Call the action's method **intensityChanged()**.

That's it for the storyboard, so bring up ViewController.swift and let's start coding…

Importing a picture

We already have two outlets at the top of our class: one for the image view and one for the slider. We need another property, in which we will store a **UIImage** containing the image that the user selected. So, add this beneath the two outlets:

```
var currentImage: UIImage!
```

Our first task will be to import a photo from the user's photo library. This is almost identical to project 10, so I'm going to explain only the important bits. If you missed out project 10, you should have paid heed to my warning not to skip projects!

First we need to add a button to the navigation bar that will allow users to import a photo from their library. Put these two lines into your **viewDidLoad()** method:

```
title = "YACIFP"

navigationItem.rightBarButtonItem =
UIBarButtonItem(barButtonSystemItem: .Add, target: self, action:
"importPicture")
```

Alright, so the first one isn't needed – it just sets the title to be YACIFP, short for "Yet Another Core Image Filters Program." (Spoiler: the App Store is full of them!) If you're feeling a bit less cynical than me, try "Instafilter" for a title instead. But what matters is the second line, because it starts the import process.

Here's the **importPicture()** method – it's almost identical to the import method from project 10, so again no explaining required:

```
func importPicture() {
  let picker = UIImagePickerController()
  picker.allowsEditing = true
  picker.delegate = self
  presentViewController(picker, animated: true, completion: nil)
```

```
}
```

You should remember that the first time you use a **UIImagePickerController** iOS will ask the user for permission to read their photo library.

Once you assign our view controller to be the image picker's delegate, you'll get warnings that we don't conform to the correct protocols. Fix that by changing the view controller's class definition to this:

```
class ViewController: UIViewController,
UIImagePickerControllerDelegate, UINavigationControllerDelegate {
```

Again, this is identical to project 10.

As before, we need to implement two methods to make the image picker useful: one for when the user pressed cancel, and one for when they selected a picture. This code is almost verbatim from project 10, so it should all be old news to you:

```
func imagePickerController(picker: UIImagePickerController,
didFinishPickingMediaWithInfo info: [String : AnyObject]) {

  var newImage: UIImage


  if let possibleImage = info["UIImagePickerControllerEditedImage"]
as? UIImage {

    newImage = possibleImage

  } else if let possibleImage =
info["UIImagePickerControllerOriginalImage"] as? UIImage {

    newImage = possibleImage

  } else {

    return

  }
```

```
    dismissViewControllerAnimated(true, completion: nil)


    currentImage = newImage
}


func imagePickerControllerDidCancel(picker: UIImagePickerController)
{
    dismissViewControllerAnimated(true, completion: nil)
}
```

There is one slight change in there, and it's where we set our **currentImage** image to be the one selected in the image picker. This is required so that we can have a copy of what was originally imported. Whenever the user changes filter, we need to put that original image back into the filter.

This has all been old code, so nothing too taxing. But now it's time for Core Image!

Applying filters

You're probably getting tired of hearing me saying this, but Core Image is yet another super-fast and super-powerful framework from Apple. It does only one thing, which is to apply filters to images that manipulate them in various ways.

One downside to Core Image is it's not very guessable, so you need to know what you're doing otherwise you'll waste a lot of time. It's also not able to take advantage of Swift's type safety, so you need to be careful when using it because the compiler won't help you as much as you're used to.

To get started, we need to add two more properties to our class, so put these underneath the **currentImage** property:

```
var context: CIContext!
var currentFilter: CIFilter!
```

The first is a Core Image context, which is the Core Image component that handles rendering. We create it here and use it throughout our app, because creating a context is computationally expensive so we don't want to keep doing it.

The second is a Core Image filter, and will store whatever filter we have activated. This filter will be given various input settings before we ask it to output a result for us to show in the image view.

We want to create both of these in **viewDidLoad()**, so put this just before the end of the method:

```
context = CIContext(options: nil)
currentFilter = CIFilter(name: "CISepiaTone")
```

That creates a default Core Image context, then creates an example filter that will apply a sepia tone effect to images. It's just for now; we'll let users change it soon enough.

To begin with, we're going to let users drag the slider up and down to add varying amounts of sepia effect to the image they select.

To do that, we need to set our **currentImage** property as the input image for the **currentFilter** Core Image filter. We're then going to call a method (as yet unwritten) called **applyProcessing()**, which will do the actual Core Image manipulation.

So, add this to the end of the **didFinishPickingMediaWithInfo** method:

```
let beginImage = CIImage(image: currentImage)

currentFilter.setValue(beginImage, forKey: kCIInputImageKey)


applyProcessing()
```

The **CIImage** data type is, for the sake of this project, just the Core Image equivalent of **UIImage**. Behind the scenes it's a bit more complicated than that, but really it doesn't matter.

As you can see, we can create a **CIImage** from a **UIImage**, and we send the result into the current Core Image Filter using the **kCIInputImageKey**. There are lots of Core Image key constants like this; at least this one is self-explanatory!

We also need to call the (still unwritten!) **applyProcessing()** method when the slider is dragged around, so modify the **intensityChanged()** method to this:

```
@IBAction func intensityChanged(sender: AnyObject) {
  applyProcessing()
}
```

With these changes, **applyProcessing()** is called as soon as the image is first imported, then whenever the slider is moved. Now it's time to write the initial version of the **applyProcessing()** method, so put this just before the end of your class:

```
func applyProcessing() {

    currentFilter.setValue(intensity.value, forKey:
kCIInputIntensityKey)


    let cgimg = context.createCGImage(currentFilter.outputImage!,
fromRect: currentFilter.outputImage!.extent)

    let processedImage = UIImage(CGImage: cgimg)


    imageView.image = processedImage

}
```

That's only four lines, none of which are terribly taxing.

The first line uses the value of our **intensity** slider to set the **kCIInputIntensityKey** value of our current Core Image filter. For sepia toning a value of 0 means "no effect" and 1 means "fully sepia."

The second line is where the hard work happens: it creates a new data type called **CGImage** from the output image of the current filter. We need to specify which part of the image we want to render, but using **currentFilter.outputImage!.extent** means "all of it." Until this method is called, no actual processing is done, so this is the one that does the real work.

The third line creates a new **UIImage** from the **CGImage**, and line four assigns that **UIImage** to our image view. Yes, I know that **UIImage**, **CGImage** and **CIImage** all sound the same, but they are different under the hood and we have no choice but to use them here.

You can now press Cmd+R to run the project as-is, then import a picture and make it sepia toned. It might be a little slow in the simulator, but I can promise you it runs brilliantly on devices - Core Image is extraordinarily fast.

Adding a sepia effect isn't very interesting, and I want to help you explore some of the other options presented by Core Image. So, we're going to make the "Change Filter" button work: it will show a **UIAlertController** with a selection of filters, and when the user selects one it will update the image.

First, here's the new **changeFilter()** method:

```
@IBAction func changeFilter(sender: AnyObject) {

  let ac = UIAlertController(title: "Choose filter", message: nil,
preferredStyle: .ActionSheet)

  ac.addAction(UIAlertAction(title: "CIBumpDistortion",
style: .Default, handler: setFilter))

  ac.addAction(UIAlertAction(title: "CIGaussianBlur",
style: .Default, handler: setFilter))

  ac.addAction(UIAlertAction(title: "CIPixellate", style: .Default,
handler: setFilter))

  ac.addAction(UIAlertAction(title: "CISepiaTone", style: .Default,
handler: setFilter))

  ac.addAction(UIAlertAction(title: "CITwirlDistortion",
style: .Default, handler: setFilter))

  ac.addAction(UIAlertAction(title: "CIUnsharpMask", style: .Default,
handler: setFilter))

  ac.addAction(UIAlertAction(title: "CIVignette", style: .Default,
handler: setFilter))

  ac.addAction(UIAlertAction(title: "Cancel", style: .Cancel,
handler: nil))

  presentViewController(ac, animated: true, completion: nil)

}
```

That's seven different Core Image filters plus one cancel button, but no new code. When tapped, each of the filter buttons will call the **setFilter()** method, which we need to make. This method should update our **currentFilter** property with the filter that was chosen, set the **kCIInputImageKey** key again (because we just changed the filter), then call **applyProcessing()**.

Each **UIAlertAction** has its title set to a different Core Image filter, and because our **setFilter()** method must accept as its only parameter the action that was tapped, we can use the action's title to create our new Core Image filter. Here's the **setFilter()** method:

```
func setFilter(action: UIAlertAction!) {
    currentFilter = CIFilter(name: action.title!)

    let beginImage = CIImage(image: currentImage)
    currentFilter.setValue(beginImage, forKey: kCIInputImageKey)

    applyProcessing()
}
```

But don't run the project yet! Our current code has a problem, and it's this line:

```
currentFilter.setValue(intensity.value, forKey: kCIInputIntensityKey)
```

That sets the intensity of the current filter. But the problem is: not all filters have an intensity setting. If you try this using the **CIBumpDistortion** filter, the app will crash because it doesn't know what to do with a setting for the key **kCIInputIntensityKey**.

All the filters and the keys they use are described fully in Apple's documentation, but for this project we're going to take a shortcut. There are four input keys we're going to manipulate across seven different filters. Sometimes the keys mean different things, and sometimes the keys don't exist, so we're going to apply only the keys that do exist with some cunning code.

Each filter has an **inputKeys** property that returns an array of all the keys it can support. We're going to use this array in conjunction with the **contains()** method to see if each of our input keys exist, and, if it does, use it. Not all of them expect a value between 0 and 1, so I sometimes multiply the slider's value to make the effect more pronounced.

Change your **applyProcessing()** method to be this:

```
func applyProcessing() {
    let inputKeys = currentFilter.inputKeys
```

```
   if inputKeys.contains(kCIInputIntensityKey)
{ currentFilter.setValue(intensity.value, forKey:
kCIInputIntensityKey) }

   if inputKeys.contains(kCIInputRadiusKey)
{ currentFilter.setValue(intensity.value * 200, forKey:
kCIInputRadiusKey) }

   if inputKeys.contains(kCIInputScaleKey)
{ currentFilter.setValue(intensity.value * 10, forKey:
kCIInputScaleKey) }

   if inputKeys.contains(kCIInputCenterKey)
{ currentFilter.setValue(CIVector(x: currentImage.size.width / 2, y:
currentImage.size.height / 2), forKey: kCIInputCenterKey) }


   let cgimg = context.createCGImage(currentFilter.outputImage!,
fromRect: currentFilter.outputImage!.extent)

   let processedImage = UIImage(CGImage: cgimg)


   self.imageView.image = processedImage

}
```

Using this method, we check each of our four keys to see whether the current filter supports it, and, if so, we set the value. The first three all use the value from our **intensity** slider in some way, which will produce some interesting results. If you wanted to improve this app later, you could perhaps add three sliders.

If you run your app now, you should be able to choose from various filters then watch them distort your image in weird and wonderful ways. Note that some of them – such as the Gaussian blur – will run very slowly in the simulator, but quickly on devices. If we wanted to do more complex processing (not least chaining filters together!) you can add configuration options to the **CIContext** to make it run even faster; another time, perhaps.

Saving to the photo library

I know it's fun to play around with Core Image filters (and you've only seen some of them!), but we have a project to finish so I want to introduce you to a new function: **UIImageWriteToSavedPhotosAlbum()**. This method does exactly what its name says: give it a **UIImage** and it will write the image to the photo album.

This method takes four parameters: the image to write, who to tell when writing has finished, what method to call, and any context. The context is just like the context value you can use with KVO, as seen in project 4, and again we're not going to use it here. The first two parameters are quite simple: we know what image we want to save (the processed one in the image view), and we also know that we want **self** (the current view controller) to be notified when writing has finished.

The third parameter is ugly. It needs to be a string that lists the method in our view controller that will be called, and it needs to be a particular format: methodName:parameterName:parameterName. We're going to use **"image:didFinishSavingWithError:contextInfo:"**, which will call a method that looks like this:

```
func image(image: UIImage, didFinishSavingWithError error:
NSErrorPointer, contextInfo:UnsafePointer<Void>) {
```

It's ugly, it's unavoidable, and it's a bit of a wart in iOS. To be fair, though, the fact that it stands out so much is testament to the fact that there are so few warts around!

Putting it all together, here's the finished **save()** method:

```
@IBAction func save(sender: AnyObject) {

  UIImageWriteToSavedPhotosAlbum(imageView.image!, self,
"image:didFinishSavingWithError:contextInfo:", nil)

}
```

From here on it's easy, because we just need to write the **didFinishSavingWithError**

method. This must show one of two messages depending on whether we get an error sent to us. The error might be, for example, that the user denied us permission to write to the photo album. This will be sent as an **NSError?** object, so if it's **nil** we know there was no error.

We first met **NSError** in project 1, but didn't actually use it. This time we need to, because if an error has occurred (i.e., the **error** parameter is not **nil**) then we need to unwrap the **NSError** object and use its **localizedDescription** property – this will tell users what the error message was in their own language.

```
func image(image: UIImage, didFinishSavingWithError error: NSError?,
contextInfo:UnsafePointer<Void>) {

  if error == nil {

    let ac = UIAlertController(title: "Saved!", message: "Your
altered image has been saved to your photos.",
preferredStyle: .Alert)

    ac.addAction(UIAlertAction(title: "OK", style: .Default, handler:
nil))

    presentViewController(ac, animated: true, completion: nil)

  } else {

    let ac = UIAlertController(title: "Save error", message:
error?.localizedDescription, preferredStyle: .Alert)

    ac.addAction(UIAlertAction(title: "OK", style: .Default, handler:
nil))

    presentViewController(ac, animated: true, completion: nil)

  }

}
```

And that's it: your app now imports pictures, manipulates them with a Core Image filter and a **UISlider**, then saves the result back to the photo library. Easy!

Wrap up

This has been the briefest possible introduction to Core Image, yet we still managed to make something useful, using **UISlider** for the first time and even writing images to the photo album! Unless you really do intend to make Yet Another Core Image Filters Program (best of luck!) your use of Core Image will mostly be about manipulating a picture in a very specific way, using a filter you have hand-crafted to look great.

If you want to try other filters, search on Google for "Core Image Filter Reference" and have a read – it will list the input keys for each of them so that you can get really fine-grained control over the filters.

If you want to spend more time working on this app, you could start by making the Change Filter button change title to show the name of the current filter. If you fancy something bigger, then you should definitely investigate having separate sliders to control each of the input keys you care about. For example, one for radius and one for intensity. If you want to tackle something small, see if you can make tapping "Save" do nothing if there is no image loaded.

Project 14

Whack-a-Penguin

Build a game using SKCropNode and a sprinkling of Grand Central Dispatch.

Setting up

It's time for another game, and we'll be using more of SpriteKit to build a whack-a-mole-style game, except with penguins because Whack-a-Penguin isn't trademarked. You're going to learn about **SKCropNode**, **SKTexture** and some more types of **SKAction**, and we'll also use **dispatch_after()** to execute closures after a delay.

Create a new SpriteKit game project in Xcode, named Project14 and targeting iPad, then delete the spaceship image and most of the example code just like you did in project 11. Now download the files for this project from **GitHub** and copy the assets from the Content folder into your Xcode project. Please also copy in the file Helper.swift; we'll be using it later.

Before we get into the code, please disable Portrait and Upside Down orientations because this game will run only in landscape mode.

All set? Open up GameScene.swift and get whacking!

Warning: When working with SpriteKit projects, I strongly recommend you use the iPad 2 simulator rather than iPad Air – you'll get much faster frame rates, making it much more suitable for testing.

Getting up and running

We already went over the basics of SpriteKit in project 11, so this time we're going to move a little faster: modify your **didMoveToView()** method so it reads this:

```
override func didMoveToView(view: SKView) {

    let background = SKSpriteNode(imageNamed: "whackBackground")

    background.position = CGPoint(x: 512, y: 384)

    background.blendMode = .Replace

    background.zPosition = -1

    addChild(background)


    gameScore = SKLabelNode(fontNamed: "Chalkduster")

    gameScore.text = "Score: 0"

    gameScore.position = CGPoint(x: 8, y: 8)

    gameScore.horizontalAlignmentMode = .Left

    gameScore.fontSize = 48

    addChild(gameScore)

}
```

For that to work properly, you'll need to add two properties to the top of the class:

```
var gameScore: SKLabelNode!

var score: Int = 0 {

    didSet {

        gameScore.text = "Score: \(score)"

    }

}
```

Blah blah property observers blah – this is old stuff to a Swift veteran like you, so I don't need to explain what that does.

If you run the "game" now you'll see a grassy background with a tree on one side. We're going to fill that with holes, and in each hole there'll be a penguin. We want each hole to do as much work itself as possible, so rather than clutter our game scene with code we're going to create a subclass of **SKNode** that will encapsulate all hole related functionality.

Add a new file, choosing iOS > Source > Cocoa Touch Class, make it a subclass of **SKNode** and name it "WhackSlot". You've already met **SKSpriteNode**, **SKLabelNode** and **SKEmitterNode**, and they all come from **SKNode**. This base class doesn't draw images like sprites or hold text like labels; it just sits in our scene at a position, holding other nodes as children.

Note: If you were wondering why we're not calling the class **WhackHole** it's because a slot is more than just a hole. It will contain a hole, yes, but it will also contain the penguin image and more.

When you create the subclass you will immediately get a compile error, because Swift claims not to know what **SKNode** is. This is easily fixed by adding the line **import SpriteKit** at the top of your file, just above the **import UIKit**.

To begin with, all we want the **WhackSlot** class to do is add a hole at its current position, so add this method to your new class:

```
func configureAtPosition(pos: CGPoint) {

  position = pos


  let sprite = SKSpriteNode(imageNamed: "whackHole")

  addChild(sprite)
}
```

You might wonder why we aren't using an initializer for this purpose, but the truth is that if you created a custom initializer you get roped into creating others because of Swift's **required init** rules. If you don't create any custom initializers (and don't have any non-

optional properties) Swift will just use the parent class's **init()** methods.

We want to create four rows of slots, with five slots in the top row, then four in the second, then five, then four. This creates quite a pleasing shape, but as we're creating lots of slots we're going to need three things:

1. An array in which we can store all our slots for referencing later.
2. A **createSlotAt()** method that handles slot creation.
3. Four loops, one for each row.

The first item is easy enough – just add this property above the existing **gameScore** definition in GameScene.swift:

```
var slots = [WhackSlot]()
```

As for number two, that's not hard either – we need to create a method that accepts a position, then creates a **WhackSlot** object, calls its **configureAtPosition()** method, then adds the slot both to the scene and to our array:

```
func createSlotAt(pos: CGPoint) {
    let slot = WhackSlot()
    slot.configureAtPosition(pos)
    addChild(slot)
    slots.append(slot)
}
```

The only moderately hard part of this task is the four loops that call **createSlotAt()** because you need to figure out what positions to use for the slots. Fortunately for you, I already did the design work, so I can tell you exactly where the slots should go! Put this just before the end of **didMoveToView()**:

```
for i in 0 ..< 5 { createSlotAt(CGPoint(x: 100 + (i * 170), y:
```

```
410)) }
for i in 0 ..< 4 { createSlotAt(CGPoint(x: 180 + (i * 170), y:
320)) }
for i in 0 ..< 5 { createSlotAt(CGPoint(x: 100 + (i * 170), y:
230)) }
for i in 0 ..< 4 { createSlotAt(CGPoint(x: 180 + (i * 170), y:
140)) }
```

Remember that higher Y values in SpriteKit place nodes towards the top of the scene, so those lines create the uppermost slots first then work downwards.

In case you've forgotten, **..<** is the half-open range operator, meaning that the first loop will count 0, 1, 2, 3, 4 then stop. The **i** is useful because we use that to calculate the X position of each slot.

So far this has all been stuff you've done before, so I tried to get through it as fast as I could. But it's now time to try something new: **SKCropNode**. This is a special kind of **SKNode** subclass that uses an image as a cropping mask: anything in the colored part will be visible, anything in the transparent part will be invisible.

By default, nodes don't crop, they just form part of a node tree. The reason we need the crop node is to hide our penguins: we need to give the impression that they are inside the holes, sliding out for the player to whack, and the easiest way to do that is just to have a crop mask shaped like the hole that makes the penguin invisible when it moves outside the mask.

The easiest way to demonstrate the need for **SKCropNode** is to give it a **nil** mask – this will effectively stop the crop node from doing anything, thus allowing you to see the trick behind our game.

In WhackSlot.swift, add a property to your class in which we'll store the penguin picture node:

```
var charNode: SKSpriteNode!
```

Now add this just before the end of the **configureAtPosition()** method:

```
let cropNode = SKCropNode()

cropNode.position = CGPoint(x: 0, y: 15)

cropNode.zPosition = 1

cropNode.maskNode = nil


charNode = SKSpriteNode(imageNamed: "penguinGood")

charNode.position = CGPoint(x: 0, y: -90)

charNode.name = "character"

cropNode.addChild(charNode)


addChild(cropNode)
```

Some parts of that are old and some are new, but all bear explaining.

First, we create a new **SKCropNode** and position it slightly higher than the slot itself. The number 15 isn't random – it's the exact number of points required to make the crop node line up perfectly with the hole graphics. We also give the crop node a **zPosition** value of 1, putting it to the front of other nodes, which stops it from appearing behind the hole.

We then do something that, right now, means nothing: we set the **maskNode** property of the crop node to be **nil**, which is the default value. It's there because we'll be changing it in just a moment.

We then create the character node, giving it the "good penguin" graphic, which is a blue color – the bad penguins are red, presumably because they are bubbling over with hellfire or something. This is placed at -90, which is way below the hole as if the penguin were properly hiding. And by "properly" you should read "bizarrely" because penguins aren't exactly known for hiding in holes in the countryside!

I hope you noticed the important thing, which is that the character node is added to the crop node, and the crop node added to the slot. This is because the crop node only crops nodes that are inside it, so we need to have a clear hierarchy: the slot has the hole and crop node as children, and the crop node has the character node as a child.

If you run the game now you'll see that every hole now has a penguin directly beneath it. This is where the penguin is hiding, "in the hole", or at least would be if we gave the crop node a mask graphic. Now is probably a good time to select the whackMask.png graphic in the project navigator – it's a red square with a curved bottom to match the rim of the hole.

Our penguins are positioned just below their holes, and they'll become invisible once added to a crop node.

Remember, with crop nodes everything with a color is visible, and everything transparent is invisible, so the whackMask.png will show all parts of the character that are above the hole. Change the **maskNode = nil** line to load the actual mask instead:

```
cropNode.maskNode = SKSpriteNode(imageNamed: "whackMask")
```

If you run the game now, you'll see the penguins are invisible. They are still there, of course, but now can't be seen.

Penguin, show thyself

We want the slots to manage showing and hiding penguins themselves as needed, which means we need to give them some properties and methods of their own. The two things a slot needs to know are "am I currently visible to be whacked by the player?" and "have I already been hit?" The former avoids players tapping on slots that are supposed to be invisible; the latter so that players can't whack a penguin more than once.

To track this data, put these two properties at the top of your **WhackSlot** class:

```
var visible = false
var isHit = false
```

Showing a penguin for the player to tap on will be handled by a new method called **show()**. This will make the character slide upwards so it becomes visible, then set **visible** to be true and **hit** to be false. The movement is going to be created by a new **SKAction**, called **moveByX(_:y:duration:)**.

This method will also decide whether the penguin is good or bad – i.e., whether the player should hit it or not. This will be done using a **RandomInt()** function that I bundled into Helper.swift for you to make random number generation easier: one-third of the time the penguin will be good; the rest of the time it will be bad.

To make it clear to the player which is which, we have two different pictures: penguinGood and penguinEvil. We can change the image inside our penguin sprite by changing its **texture** property. This takes a new class called **SKTexture**, which is to **SKSpriteNode** sort of what **UIImage** is to **UIImageView** – it holds image data, but isn't responsible for showing it.

Changing the character node's texture like this is helpful because it means we don't need to keep adding and removing nodes. Instead, we can just change the texture to match what kind of penguin this is, then change the node name to match so we can do tap detection later on.

However, all the above should only happen if the slot isn't already visible, because it could cause havoc. So, the very first thing the method needs to do is check whether **visible** is true, and if so exit.

Enough talk; here's the **show()** method:

```
func show(hideTime hideTime: Double) {
  if visible { return }

  charNode.runAction(SKAction.moveByX(0, y: 80, duration: 0.05))
  visible = true
  isHit = false

  if RandomInt(min: 0, max: 2) == 0 {
    charNode.texture = SKTexture(imageNamed: "penguinGood")
    charNode.name = "charFriend"
  } else {
    charNode.texture = SKTexture(imageNamed: "penguinEvil")
    charNode.name = "charEnemy"
  }
}
```

You may have noticed that I made the method accept a parameter called **hideTime**. This is for later, to avoid having to rewrite too much code. You may also have noticed that it's actually written as **hideTime hideTime**, because by default in Swift the first parameter to a method does not have a label.

If we hadn't used the **hideTime** twice, you could call the method like this: **show(1)**. With the duplicate label, you must use the label, like this: **show(hideTime: 1)**. Doubling the parameter name to force a label is helpful here because otherwise people won't know why they are passing a parameter.

The **show()** method is going to be triggered by the view controller on a recurring basis, managed by a property we're going to create called **popupTime**. This will start at 0.85 (create a new enemy a bit faster than once a second), but every time we create an enemy

we'll also decrease **popupTime** so that the game gets harder over time.

First, the easy bit: add this property to GameScene.swift:

```
var popupTime = 0.85
```

To jump start the process, we need to call **createEnemy()** once when the game starts, then have **createEnemy()** call itself thereafter. Clearly we don't want to start creating enemies as soon as the game starts, because the player needs a few moments to orient themselves so they have a chance.

So, in **didMoveToView()** we're going to call the (as yet unwritten) **createEnemy()** method after a delay. This requires some new Grand Central Dispatch (GCD) code: **dispatch_time()** is used to create time values for a delay, and **dispatch_after()** is used to schedule a closure to execute after the time has been reached.

The reason we need to create a GCD time value is because it thinks in nanoseconds, which isn't very intuitive for humans. One second is equal to 1,000,000,000 (one billion) nanoseconds, but helpfully Apple has a built-in constant called **NSEC_PER_SEC** to help us convert between seconds and nanoseconds. Here's how the code looks to run a closure after a delay:

```
let delay = 1.0
let time = dispatch_time(DISPATCH_TIME_NOW, Int64(delay *
Double(NSEC_PER_SEC)))
dispatch_after(time, dispatch_get_main_queue()) { [unowned self] in

  self.doStuff()

}
```

That's not easy to remember or type, so in the file Helper.swift I've included a function called **RunAfterDelay()** that does all the work for you. Call it like this:

```
RunAfterDelay(1.0) { [unowned self] in

   self.doStuff()

}
```

Now, onto the **createEnemy()** method. This will do several things:

• Decrease **popupTime** each time it's called. I'm going to multiply it by 0.991 rather than subtracting a fixed amount, otherwise the game gets far too fast.
• Shuffle the list of available slots using the GameplayKit shuffle that we've used previously.
• Make the first slot show itself, passing in the current value of **popupTime** for the method to use later.
• Generate four random numbers to see if more slots should be shown. Potentially up to five slots could be shown at once.
• Call itself again after a random delay. The delay will be between **popupTime** halved and **popupTime** doubled. For example, if **popupTime** was 2, the random number would be between 1 and 4.

There are only two new things in there. First, I'll be using the ***=** operator to multiply and assign at the same time, in the same way that **+=** meant "add and assign" in project 2. Second, I'll be using the **RandomDouble()** fuction to generate a random **Double** value, which is what **RunAfterDelay()** uses for its delay.

Here's the method to create enemies:

```
func createEnemy() {

   popupTime *= 0.991


   slots =
GKRandomSource.sharedRandom().arrayByShufflingObjectsInArray(slots)
as! [WhackSlot]

   slots[0].show(hideTime: popupTime)


   if RandomInt(min: 0, max: 12) > 4 { slots[1].show(hideTime:
popupTime) }

   if RandomInt(min: 0, max: 12) > 8 {  slots[2].show(hideTime:
```

```
popupTime) }

   if RandomInt(min: 0, max: 12) > 10 { slots[3].show(hideTime:
popupTime) }

   if RandomInt(min: 0, max: 12) > 11 { slots[4].show(hideTime:
popupTime)   }


   let minDelay = popupTime / 2.0

   let maxDelay = popupTime * 2


   RunAfterDelay(RandomDouble(min: minDelay, max: maxDelay))
{ [unowned self] in

     self.createEnemy()

   }

}
```

Make sure you add **import GameplayKit** to the top of your file in order to make the array shuffling work.

Because **createEnemy()** calls itself, all we have to do is call it once in **didMoveToView()** after a brief delay. Put this just before the end of the method:

```
RunAfterDelay(1) { [unowned self] in

   self.createEnemy()

}
```

From then on, we don't have to worry about it because **createEnemy()** will call itself.

Before we're done, we need to upgrade the **WhackSlot** class to include a **hide()** method. If you run the code now, you'll see that the penguins appear nice and randomly, but they never actually go away. We're already passing a **hideTime** parameter to the **show()** method, and we're going to use that so the slots hide themselves after they have been visible for a time.

We could of course just make the slots hide after a fixed time, but that's no fun. By using

popupTime as the input for hiding delay, we know the penguins will hide themselves more quickly over time.

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

```
func hide() {
  if !visible { return }

  charNode.runAction(SKAction.moveByX(0, y:-80, duration:0.05))
  visible = false
}
```

That just undoes the results of **show()**: the penguin moves back down the screen into its hole, then its **visible** property is set to false.

We want to trigger this method automatically after a period of time, and, through extensive testing (that is, sitting around playing) I have determined the optimal hide time to be 3.5x **popupTime**.

So, put this code at end of show():

```
RunAfterDelay(hideTime * 3.5) { [unowned self] in
  self.hide()
}
```

Go ahead and run the app, because it's really starting to come together: the penguins show randomly, sometimes by themselves and sometimes in groups, then hide after a period of being visible. But you can't hit them, which means this game is more Watch-a-Penguin than Whack-a-Penguin. Let's fix that!

Whack to win

To bring this project to a close, we still need to do two major components: letting the player tap on a penguin to score, then letting the game end after a while. Right now it never ends, so with **popupTime** getting lower and lower it means the game will become impossible after a few minutes.

We're going to add a **hit()** method to the **WhackSlot** class that will handle hiding the penguin. This needs to wait for a moment (so the player still sees what they tapped), move the penguin back down again, then set the penguin to be invisible again.

We're going to use an **SKAction** for each of those three things, which means you need to learn some new uses of the class:

• **SKAction.waitForDuration()** creates an action that waits for a period of time, measured in seconds.
• **SKAction.runBlock()** will run any code we want, provided as a closure. "Block" is Objective C's name for a Swift closure.
• **SKAction.sequence** takes an array of actions, and executes them in order. Each action won't start executing until the previous one finished.

We need to use **SKAction.runBlock()** in order to set the penguin's **visible** property to be false rather than doing it directly, because we want it to fit into the sequence. Using this technique, it will only be changed when that part of the sequence is reached.

Put this method into the **WhackSlot** class:

```
func hit() {
  isHit = true


  let delay = SKAction.waitForDuration(0.25)

  let hide = SKAction.moveByX(0, y:-80, duration:0.5)

  let notVisible = SKAction.runBlock { [unowned self] in self.visible
= false }

  charNode.runAction(SKAction.sequence([delay, hide, notVisible]))

}
```

With that new method in place, we can call it from the **touchesBegan()** method in GameScene.swift. This method needs to figure out what was tapped using the same **nodesAtPoint()** method you saw in project 11: find any touch, find out where it was tapped, then get a node array of all nodes at that point in the scene.

We then need to loop through the list of all nodes that are at that point, and see if they have the name "charFriend" or "charEnemy" and take the appropriate action. Rather than dump all the code on you at once, here's the basic outline of **touchesBegan()** to start with:

```
override func touchesBegan(touches: Set<UITouch>, withEvent event:
UIEvent?) {
    if let touch = touches.first {
      let location = touch.locationInNode(self)
      let nodes = nodesAtPoint(location)


      for node in nodes {
        if node.name == "charFriend" {
          // they shouldn't have whacked this penguin
        } else if node.name == "charEnemy" {
          // they should have whacked this one
        }
      }
    }
  }
```

Nothing complicated there – this is all stuff you know already.

What *is* new is what comes in place of those two comments. The first comment marks the code block that will be executed if the player taps a friendly penguin, which is obviously against the point of the game. When this happens, we need to call the **hit()** method to make the penguin hide itself, subtract 5 from the current score, then run an action that plays a "bad

hit" sound. All of that should only happen if the slot was visible and not hit.

The code for this block is going to do something interesting that you haven't seen before, and it looks like this:

```
let whackSlot = node.parent!.parent as WhackSlot
```

It gets the parent of the parent of the node, and typecasts it as a **WhackSlot**. This line is needed because the player has tapped the penguin sprite node, not the slot – we need to get the parent of the penguin, which is the crop node it sits inside, then get the parent of the crop node, which is the **WhackSlot** object, which is what this code does.

You're also going to meet two new pieces of code. First, the **-=** operator, which is similar to **+=** and ***=** and means "subtract and assign." So, **a -= 5** means "subtract 5 from a." The second new piece of code is SKAction's **playSoundFileNamed()** method, which plays a sound and optionally waits for the sound to finish playing before continuing – useful if you're using an action sequence.

We haven't used sound files in iOS yet, but there isn't really a whole lot to say. The three main sound file formats you'll use are MP3, M4A and CAF, with the latter being a renamed AIFF file. AIFF is a pretty terrible file format when it comes to file size, but it's much faster to load and use than MP3s and M4As, so you'll use them often.

Put this code where the **// they shouldn't have whacked this penguin** comment was:

```
let whackSlot = node.parent!.parent as! WhackSlot
if !whackSlot.visible { continue }
if whackSlot.isHit { continue }


whackSlot.hit()
score -= 5


runAction(SKAction.playSoundFileNamed("whackBad.caf",
waitForCompletion:false))
```

When the player taps a bad penguin, the code is similar. The differences are that we want to add 1 to the score (so that it takes five correct taps to offset one bad one), and run a different sound. But we're also going to set the **xScale** and **yScale** properties of our character node so the penguin visibly shrinks in the scene, as if they had been hit.

Put this code where the **// they should have whacked this one** comment was:

```
let whackSlot = node.parent!.parent as! WhackSlot
if !whackSlot.visible { continue }
if whackSlot.isHit { continue }


whackSlot.charNode.xScale = 0.85
whackSlot.charNode.yScale = 0.85


whackSlot.hit()
score += 1


runAction(SKAction.playSoundFileNamed("whack.caf",
waitForCompletion:false))
```

Since we're now potentially modifying the **xScale** and **yScale** properties of our character node, we need to reset them to 1 inside the **show()** method of the slot. Put this just before the **runAction()** call inside **show()**:

```
charNode.xScale = 1
charNode.yScale = 1
```

This game is almost done. Thanks to the property observer we put in early on the game is now perfectly playable, at least until **popupTime** gets so low that the game is effectively

unplayable.

To fix this final problem and bring the project to a close, we're going to limit the game to creating just 30 rounds of enemies. Each round is one call to **createEnemy()**, which means it might create up to five enemies at a time.

First, add this property to the top of your game scene:

```
var numRounds = 0
```

Every time **createEnemy()** is called, we're going to add 1 to the **numRounds** property. When it is greater than or equal to 30, we're going to end the game: hide all the slot, show a "Game over" sprite, then exit the method. Put this code just before the **popupTime** assignment in **createEnemy()**:

```
numRounds += 1

if numRounds >= 30 {
  for slot in slots {
    slot.hide()
  }

  let gameOver = SKSpriteNode(imageNamed: "gameOver")
  gameOver.position = CGPoint(x: 512, y: 384)
  gameOver.zPosition = 1
  addChild(gameOver)

  return
}
```

That uses a position **zPosition** so that the game over graphic is placed over other items in our game.

The game is now complete! Go ahead and play it for real and see how you do. If you're using the iOS simulator, bear in mind that it's much hard to move a mouse pointer than it is to use your fingers on a real iPad, so don't adjust the difficulty unless you're testing on a real device!

Wrap up

You have another game under your belt, and I hope your brain is already starting to bubble up ideas for things you can do to improve it. Plus, you learned more skills, not least **SKCropNode**, **SKTexture**, GCD's **dispatch_after()**, the ***=** and **-=** operators, forcing method labels, plus lots of new **SKAction** types, so it's all time well spent.

If you're looking to improve this project, you could start by adding a smoke-like particle effect to be used when the penguins are hit, and perhaps a separate mud-like effect when they go into or come out of a hole. You could also record your own voice saying "Game over!" and have it play when the game ends. Try experimenting with the difficulty and see what you come up with – is it easier or harder if the penguin show/hide animation happens at random speeds?

Project 15

Animation

Bring your interfaces to life with animation, and meet switch/case at the same time.

Setting up

As we're just before the half-way point of this series, it's time to introduce one of the most important techniques in iOS development: animation. Sadly, many people don't consider animation important at all, which makes for some thoroughly awful user interface design.

Animation – making things move, scale, and so on – of your views is not only about making things pretty, although that's certainly a large part. It's main purpose is to gives users a sense of what's changing and why, and it helps them make sense of a state change in your program. When you use a navigation controller to show a new view controller, we don't just want it to appear. Instead, we want it to slide in, making it clear that the old screen hasn't gone away, it's just to the left of where we were.

You're almost certainly tired of hearing me say this, but iOS has a ridiculously powerful animation toolkit that's also easy to use. I know, I'm a broken record, right?

Well, don't just take my word for it – let's try out some animation together so you can see exactly how it works. You're also going to meet switch/case for the first time and learn about the **CGAffineTransform** struct, both of which will serve you just as well as animations. So, create a new Single View Application project in Xcode, name it Project15 and set its target to be iPad.

Please download the files for this project from **GitHub** and copy its Content folder into your Xcode project. Finally, set the orientation to be landscape only. Animation of course works in all orientations, but it's easier to work with a fixed size for now.

Preparing for action

Open Interface Builder with Main.storyboard and place a button on there with the title "Tap". Give it width 46 and height 44, with X:277 and Y:542. We need to add some Auto Layout constraints, so select the button in the document outline and Ctrl-drag diagonally on it. The popup menu will offer "Width", "Height" and "Aspect Ratio"; please add Width and Height.

We want our button to always stay near the bottom of the view controller, so Ctrl-drag from the button to the view directly above it and choose "Bottom Space to Bottom Layout Guide." Now Ctrl-drag the same way again and choose "Center Horizontally in Container."

That's it for Auto Layout, so please switch to the assistant view so we can add an action and an outlet. Ctrl-drag from the button to your code to create an outlet for it called **tap**. Then Ctrl-drag again to create an action for the button called **tapped()**.

Every time the user taps the "Tap" button, we're going to execute a different animation. This will be accomplished by cycling through a counter, and moving an image view. To make all that work, you need to add two more properties to the class:

```
var imageView: UIImageView!
var currentAnimation = 0
```

There isn't an image view in the storyboard – we're going to create it ourself in **viewDidLoad()** using an initializer that takes a **UIImage** and makes the image view the correct size for the image.

Add this code to **viewDidLoad()**:

```
imageView = UIImageView(image: UIImage(named: "penguin"))
imageView.center = CGPoint(x: 512, y: 384)
view.addSubview(imageView)
```

That places the penguin in the middle of an iPad-sized landscape screen, ready for us to animate.

There's one more thing we're going to do before we start looking at the animations, and that's to put a little bit of code into the **tapped()** method so that we cycle through animations each time the button is tapped. Put this in there:

```
currentAnimation += 1

if currentAnimation > 7 {
    currentAnimation = 0
}
```

That will add 1 to the value of **currentAnimation** until it reaches 7, at which point it will set it back to 0.

Switch, case, animate

The **currentAnimation** property can have a value between 0 and 7, each one triggering a different animation. You might be tempted to write code like this:

```
if currentAnimation == 0 {
  anim1()
} else if currentAnimation == 1 {
  anim2()
} else if currentAnimation == 2 {
  andSoOn()
}
```

But that's not a very efficient way of checking a variable for multiple possible values, so programming languages have a different syntax for doing exactly that, known as switch/ case. Using this syntax, we could more or less rewrite the previous code like this:

```
switch currentAnimation {
case 0:
  anim1()
case 1:
  anim2()
case 3:
  andSoOn()
}
```

I say "more or less rewrite" because Swift wants to make sure your code is as safe as possible, and one of the checks it executes is that your switch/case statements are exhaustive – that every possible outcome is catered for.

As a result, you will frequently need to include a **default** case block to match any value not

explicitly catered for above, like this:

```
switch currentAnimation {
case 0:
  anim1()
case 1:
  anim2()
case 3:
  andSoOn()
default:
  break
}
```

The **break** statement exits the switch/case block, so the **default** case effectively does nothing.

Note: by default, Swift executes only the case block that matches the value you are switching on. If you want it to carry on executing the next one as well, you should use the **fallthrough** statement. If you don't know what this means, you don't want it!

We're going to create a big switch/case block inside **tapped()**, but we're going to start small and work our way up – the **default** case will handle any values we don't explicitly catch.

This switch/case statement is going to go inside a new method of the **UIView** class called **animateWithDuration()**, which is a kind of method you haven't seen before because it actually accepts two closures. The parameters we'll be using are how to long animate for, how long to pause before the animation starts, any options you want to provide, what animations to execute, and finally a closure that will execute when the animation finishes.

Because the completion closure is the final parameter to the method, we'll be using trailing closure syntax just like we did in project 5.

Update your **tapped()** method to this:

```
@IBAction func tapped(sender: AnyObject) {

  tap.hidden = true


  UIView.animateWithDuration(1, delay: 0, options: [],

    animations: { [unowned self] in

      switch self.currentAnimation {

      case 0:

        break


      default:

        break

      }

  }) { [unowned self] (finished: Bool) in

    self.tap.hidden = false

  }


  currentAnimation += 1


  if currentAnimation > 7 {

    currentAnimation = 0

  }

}
```

That won't do anything yet, which is remarkable given that it's quite a lot of code! However, it has put us in a position where we can start dabbling with animations. But first, here's a breakdown of the code:

 • When the method begins, we hide the **tap** button so that our animations don't collide; it gets unhidden in the completion closure of the animation.
 • We call **animateWithDuration()** with a duration of 1 second, no delay, and no interesting

options.

• For the **animations** closure we first do the usual **[unowned self] in** dance to avoid strong reference cycles, then enter the switch/case code.

• We switch using the value of **self.currentAnimation**. We need to use **self** to make the closure capture clear, remember. This switch/case does nothing yet, because both possible cases just call **break**.

• We use trailing closure syntax to provide our completion closure. This will be called when the animation completes, and its **finished** will be true if the animations completed fully.

• As I said, the completion closure unhides the **tap** button so it can be tapped again.

• After the **animateWithDuration()** call, we have the old code to modify and wrap **currentAnimation**.

If you run the app now and tap the button, you'll notice it doesn't actually hide and show as you might expect. This is because UIKit detects that no animation has taken place, so it calls the completion closure straight away.

Transform!

Our code now has the perfect structure in place to let us dabble with animations freely, so it's time to learn about **CGAffineTransform**. This is a structure that represents a specific kind of transform that we can apply to any **UIView** object or subclass.

Unless you're into mathematics, affine transforms can seem like a black art. But Apple does provide some great helper functions to make it easier: there are functions to scale up a view, functions to rotate, functions to move, and functions to reset back to default.

All of these functions return a **CGAffineTransform** value that you can put into a view's **transform** property to apply it. As we'll be doing this inside an animation block, the transform will automatically be animated. This illustrates one of the many powerful things of Core Animation: you tell it what you want to happen, and it calculates all the intermediary states automatically.

Let's start with something simple: when we're at **currentAnimation** value 0, we want to make the view 2x its default size. Change the switch/case code to this:

```
switch self.currentAnimation {
case 0:

  self.imageView.transform = CGAffineTransformMakeScale(2, 2)


default:

  break

}
```

That uses a new function, **CGAffineTransformMakeScale()**, which takes an X and Y scale value as its two parameters. A value of 1 means "the default size," so **2, 2** will make the view twice its normal width and height. By default, UIKit animations have an "ease in, ease out" curve, which means the movement starts slow, accelerates, then slows down again before reaching the end.

Run the app now and tap the button to watch the penguin animate from 1x to 2x its size over one second, all by setting the transform inside an animation. You can keep tapping the

button as many times more as you want, but nothing else will happen at this time. If you apply a 2x scale transform to a view that already has a 2x scale transform, nothing happens.

When iOS draws the penguin at twice its size, it automatically smooths the image so it doesn't look too jaggy.

The next case is going to be 1, and we're going to use a special existing transform called **CGAffineTransformIdentity**. This effectively clears our view of any pre-defined transform, resetting any changes that have been applied by modifying its **transform** property.

Add this to the switch/case statement after the existing case:

```
case 1:
    self.imageView.transform = CGAffineTransformIdentity
```

For the sake of clarity, your code should now read:

```
switch self.currentAnimation {

case 0:

  self.imageView.transform = CGAffineTransformMakeScale(2, 2)


case 1:

  self.imageView.transform = CGAffineTransformIdentity


default:

  break

}
```

With the second case in there, tapping the button repeatedly will first scale the penguin up, then scale it back down (resetting to defaults), then do nothing for lots of taps, then repeat the scale up/scale down. This is because our **currentAnimation** value is told to wrap (return to 0) when it's greater than 7, so the **default** case executes quite a few times.

Let's continue adding more cases: one to move the image view, then another to reset it back to the identity transform:

```
case 2:

  self.imageView.transform = CGAffineTransformMakeTranslation(-256,
-256)


case 3:

  self.imageView.transform = CGAffineTransformIdentity
```

That uses another new function, **CGAffineTransformMakeTranslation()**, which accepts X and Y values for its parameters. These values are *deltas*, or differences from the current value, meaning that the above code subtracts 256 from both the current X and Y position.

Tapping the button now will scale up then down, then move and return back to the center, all

smoothly animated by Core Animation.

We can also use **CGAffineTransform** to rotate views, using the
CGAffineTransformMakeRotation method. This accepts one parameter, which is the
amount in radians you want to rotate. There are three catches to using this function:

1. You need to provide the value as a **CGFloat**. This usually isn't a problem – if you type 1.0
in there, Swift is smart enough to make that a **CGFloat** automatically. But if you try to use
M_PI to use the value of π (mathematical pi), you'll find that's a Double and you need to
convert it explicitly: **CGFloat(M_PI)**.
2. Core Animation will always take the shortest route to make the rotation work. So, if your
object is straight and you rotate to 90 degrees (radians: half of pi, or **M_PI_2**), it will rotate
clockwise. If your object is straight and you rotate to 270 degrees (radians: **M_PI + M_PI_2**) it
will rotate counter-clockwise because it's the smallest possible animation.
3. A consequence of the second catch is that if you try to rotate 360 degrees (radians: **M_PI
* 2**), Core Animation will calculate the shortest rotation to be "just don't move, because we're
already there." The same goes for values over 360, for example if you try to rotate 540
degrees (one and a half full rotations), you'll end up with just a 180-degree rotation.

With all that in mind, here's are two more cases that show off rotation:

```
case 4:
  self.imageView.transform =
CGAffineTransformMakeRotation(CGFloat(M_PI))


case 5:
  self.imageView.transform = CGAffineTransformIdentity
```

As well as animating transforms, Core Animation can animate many of the properties of your
views. For example, it can animate the background color of the image view, or the level of
transparency. You can even change multiple things at once if you want something more
complicated to happen.

As an example, to make our view almost fade out then fade back in again while also
changing its background color, we're going to modify its transparency by setting its **alpha**

value, where 0 is invisible and 1 is fully visible, and also set its **backgroundColor** property – first to green, then to clear.

Add these two new cases:

```
case 6:
  self.imageView.alpha = 0.1
  self.imageView.backgroundColor = UIColor.greenColor()


case 7:
  self.imageView.alpha = 1
  self.imageView.backgroundColor = UIColor.clearColor()
```

That completes all possible cases, 0 to 7. But Core Animation isn't finished just yet. In fact, we've only scratched its surface in these tests, and there's much more it can do.

To give you the briefest glimpse of its power, replace this line of code:

```
UIView.animateWithDuration(1, delay: 0, options: [],
```

…with this:

```
UIView.animateWithDuration(1, delay: 0, usingSpringWithDamping: 0.5,
initialSpringVelocity: 5, options: [],
```

This changes the **animateWithDuration()** so that it uses spring animations rather than the default, ease-in-ease-out animation. I'm not even going to tell you what this does because I'm sure you're going to be impressed – press Cmd+R to run the app and tap the button for yourself. We're done!

Wrap up

Core Animation is an extraordinary toolkit, and UIKit wraps it in a simple and flexible set of methods. And because it's so simple to use, you really have no excuse for not using it. If you're moving something around conceptually (e.g., moving an email to a folder, showing a palette of paint brushes, rolling a dice, etc) then move it around *visually* too. Your users will thank you for it!

You also learned a little about switch/case as a way of evaluating multiple possible values. Although you haven't seen much of it yet, Swift's switch/case syntax is actually one of the most powerful and expressive I've ever come across, although it can bend your brain a little. In this project we were only matching simple values, but trust me: it can do so much more.

If you want to put your new-found animation skill into practice, try going back to project 8 – 7 Swifty Words – and making the letter group buttons fade out when they are tapped. We were using the **hidden** property, but you'll need to switch to **alpha** because **hidden** is either true or false, it has no animatable values between.

Project 16
JavaScript Injection

Extend Safari with a cool feature for JavaScript developers.

Setting up

Welcome to the second half of the series! From here on in, the apps you create will be looking beyond plain UIKit to explore some of the other great ways you can use Apple's tools to produce great apps. In this project you're going to create a Safari extension, which lets us embed a version of our app directly inside Safari's action menu, then manipulate Safari data in interesting ways.

What do I mean by "interesting ways"? Well, our little Safari extension is going to read in the URL and page title that the user was visiting, then show them a large text area they can type JavaScript into. When the extension is dismissed, we'll execute that JavaScript in Safari – like Mozilla's Greasemonkey extension, just for iOS.

This is the first of two projects that are *hard*. This is not because I want to torture you, but because your skills are improving and it's time to tackle bigger things. In this project, the actual amount of code you're going to be writing is quite small, because most of the code will be provided for us by Xcode. However, it's dense, and there's a lot to take in, so it might feel like slow going.

At the very least, the project will still be useful and you'll learn a lot too – not least about Safari extensions and a new class called **NSNotificationCenter**.

Let's get started: create a new Single View Application project in Xcode, name it Project16 and set it to target iPhone.

Making a shell app

Safari extensions are launched from within the Safari action menu, but they ship inside a parent app. That is, you can't ship an extension by itself – it needs have an app alongside it. Frequently the app does very little, but it must at least be present.

There are two common ways to use the app side of the extension: to show help information, or to show basic settings for the user to adjust. We're going to go with the first option, although to skip writing lots of help text we'll just be using "Hello, world!"

Open your apps Main.storyboard file and drop a **UILabel** into the view controller. Give it the text "Hello, world!" then make it large enough to fit. Now drag it so that horizontal and vertical blue alignment lines appear, meaning that it's centered. In the document outline, Ctrl-drag from the label to the view just above it, and select "Center Horizontally in Container" and "Center Vertically in Container."

When you add those two constraints, you'll probably see some orange boxes around your label – one is wholly orange, and one has a dashed line. These orange markers mean your views don't match your constraints: the solid orange lines mean "this is where you view is," and the dashed orange lines mean "this is where your view will be when your code runs."

The reason for the difference is because labels have a default size of whatever fits their current text. We drew out the size by hand, and in my case I drew it too large, so Xcode is telling me when the code runs the label will be smaller. You can fix this warning by going to the Editor menu and choosing Resolve Auto Layout Issues > Update Frames, which will make the label the size Auto Layout thinks it ought to be.

That's the entire app complete. We're not going to add any more to it here because it's really not the point; we're going to focus on the extension from here on.

Adding an extension

Extensions are miniature apps in their own right, and as such need their own space in your code. That doesn't mean you can't share code and resources between your extensions and your app, just that it's not automatic.

To get started with a fresh extension, go to the File menu and choose New > Target. When you're asked to choose a template, select iOS > Application Extension > Action Extension, then click Next. For the name just call it Extension, make sure Action Type is set to "Presents User Interface", then click Finish.
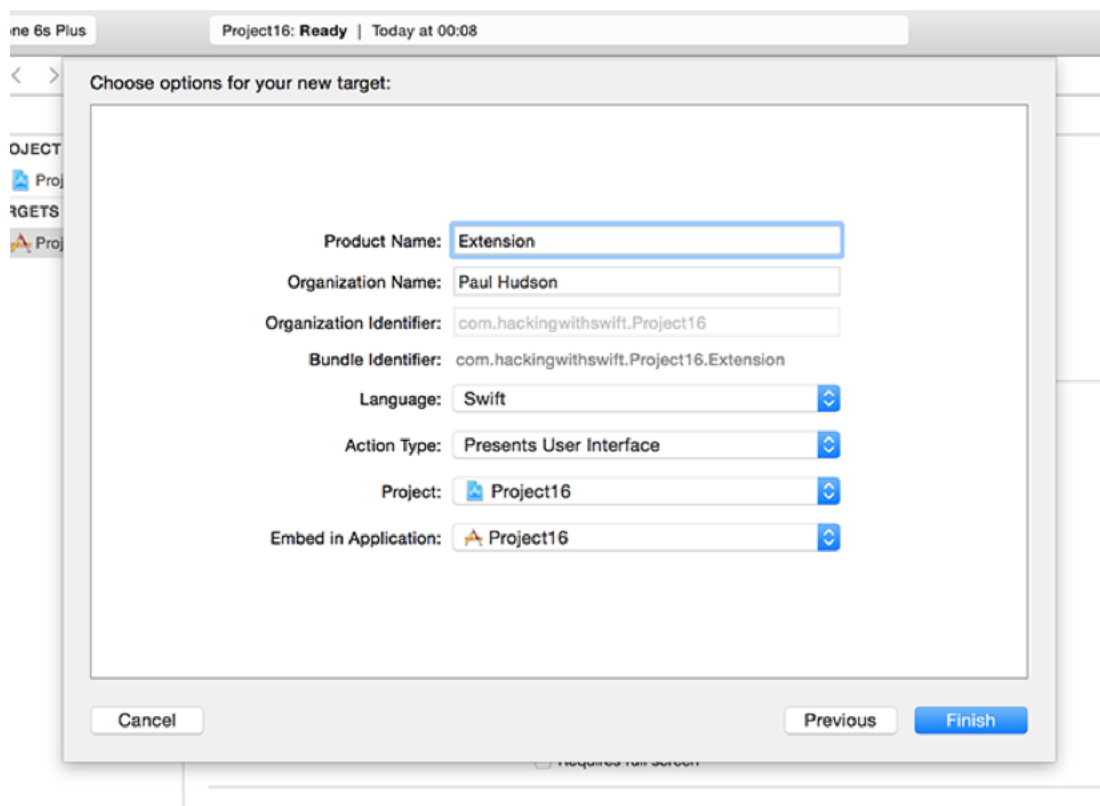
Creating a new Action Extension target effectively creates a separate chunk of source code to manage inside your project.

When you create an extension inside an app, Xcode will ask you whether you want to activate its scheme. Check the "Do not show this message again" box then click Activate. With this change, when you run your code, you'll actually launch the extension – it's perfect for our needs right now.

Once your extension has been created, it will appear in the project navigation in its own

yellow folder. You should see Project16 at the top, but look below and you'll see Extension. Open up the disclosure arrow and you'll see Xcode has given you two files: ActionViewController.swift and MainInterface.storyboard.

If you look inside ActionViewController.swift you'll see a fair amount of code, and I have some bad news for you: the code is complicated, the code is pretty much all new, and most if it is required. It's complicated because it needs to be: your extension doesn't talk to Safari and Safari doesn't talk to your extension, because it opens up security risks. Instead, iOS acts as an intermediary between Safari and the extension, passing data safely between the two.

To help make the code a little easier to understand, I want you to delete it. Go on – zap it all, leaving **viewDidLoad()** doing nothing more than calling **super.viewDidLoad()**. We're going to replace it with code that is somewhat similar, but I've removed the complicated parts to try to make it easier. You'll probably want to return to Apple's template code in your own apps!

Change your **viewDidLoad()** method to this:

```
override func viewDidLoad() {

  super.viewDidLoad()


  if let inputItem = extensionContext!.inputItems.first as?
NSExtensionItem {

    if let itemProvider = inputItem.attachments?.first as?
NSItemProvider {

      itemProvider.loadItemForTypeIdentifier(kUTTypePropertyList as
String, options: nil) { [unowned self] (dict, error) in

        // do stuff!

      }

    }

  }

}
```

Let's walk through that line by line:

• When our extension is created, its **extensionContext** lets us control how it interacts with the parent app. In the case of **inputItems** this will be an array of data the parent app is sending to our extension to use. We only care about this first item in this project, and even then it might not exist, so we conditionally typecast using if/let and **as?**.

• Our input item contains an array of attachments, which are given to us wrapped up as an **NSItemProvider**. Our code pulls out the first attachment from the first input item.

• The next line uses **loadItemForTypeIdentifier()** to ask the item provider to actually provide us with its item, but you'll notice it uses a closure so this code executes asynchronously. That is, the method will carry on executing while the item provider is busy loading and sending us its data.

• Inside our closure we first need the usual **[unowned self]** to avoid strong reference cycles, but we also need to accept two parameters: the dictionary that was given to us to by the item provider, and any error that occurred.

• With the item successfully pulled out, we can get to the interesting stuff: working with the data. We have **// do stuff!** right now, but it'll be more interesting later, I promise.

This code takes a number of shortcuts that Apple's own code doesn't, which is why it's significantly shorter. Once you've gotten to grips with this basic extension, I do recommend you go back and look at Apple's template code to see how it loops through all the items and providers to find the first image it can.

Despite all that work, you can't see the results just yet – we need to do some configuration work first, because Apple's default action extension configure is for images, not for web page content.

What do you want to get?

Inside the Extension group in the project navigator is a file called Info.plist. You have one for your app too, and in fact all apps have one. This plist (that's short for property list, remember) contains metadata about apps and extensions: what language is it, what version number is it, and so on.

For extensions, this plist also describes what data you are willing to accept and how it should be processed. Look for the key marked NSExtension and open its disclosure indicator: you should see NSExtensionAttributes, NSExtensionMainStoryboard and NSExtensionPointIdentifier. It's that first one we care about, because it modifies the way our extension behaves.

Open up the disclosure arrow for NSExtensionAttributes and you should see NSExtensionActivationRule, then String, then TRUEPREDICATE. Change String to be Dictionary, then click the small + button to the left of Dictionary, and when it asks you for a key name change "New item" to be "NSExtensionActivationSupportsWebPageWithMaxCount". You can leave the new item as a string (it doesn't really matter), but change its value to be 1 – that's the empty space just to the right of String.

Adding this value to the dictionary means that we only want to receive web pages – we aren't interested in images or other data types.

Now select the NSExtensionAttributes line itself, and click the + button that appears next to the word Dictionary. Replace "New item" with "NSExtensionJavaScriptPreprocessingFile", then give it the value "Action". This tells iOS that when our extension is called, we need to run the JavaScript preprocessing file called Action.js, which will be in our app bundle. Make sure you type "Action" and not "Action.js", because iOS will append the ".js" itself.

In the picture below you can see how your extension's property list should look. Make sure you enter the key names precisely, because there is no room for error.

Bundle OS Type code		String	XPC!
Bundle versions string, short		String	1.0
Bundle creator OS Type code		String	????
Bundle version		String	1
▼ NSExtension		Dictionary	(3 items)
▼ NSExtensionAttributes		Dictionary	(2 items)
NSExtensionJavaScriptPreprocessingFile		String	Action
▼ NSExtensionActivationRule		Dictionary	(1 item)
NSExtensionActivationSupportsWebPageWithMaxCount		String	1
NSExtensionMainStoryboard		String	MainInterface
NSExtensionPointIdentifier		String	com.apple.ui-services

This is how your extension's property list should look.

I say "will be" rather than "is" because we haven't actually created this file yet. Right-click on your extension's Info.plist file and choose New File. When you're asked what template you want, choose iOS > Other > Empty, then name it Action.js, and put this text into it:

```javascript
var Action = function() {};


Action.prototype = {


run: function(parameters) {


},


finalize: function(parameters) {


}


};


var ExtensionPreprocessingJS = new Action
```

This is a book about Swift, not a book about JavaScript, so I'm afraid I don't intend to

explain what that code does except for two things:

 • There are two functions: **run()** and **finalize()**. The first is called before your extension is run, and the other is called after.
 • Apple expects the code to be exactly like this, so you shouldn't change it other than to fill in the **run()** and **finalize()** functions.

Even now, after all this hacking around, your extension *still* isn't ready to run, and I can only apologise – I told you it was complicated!
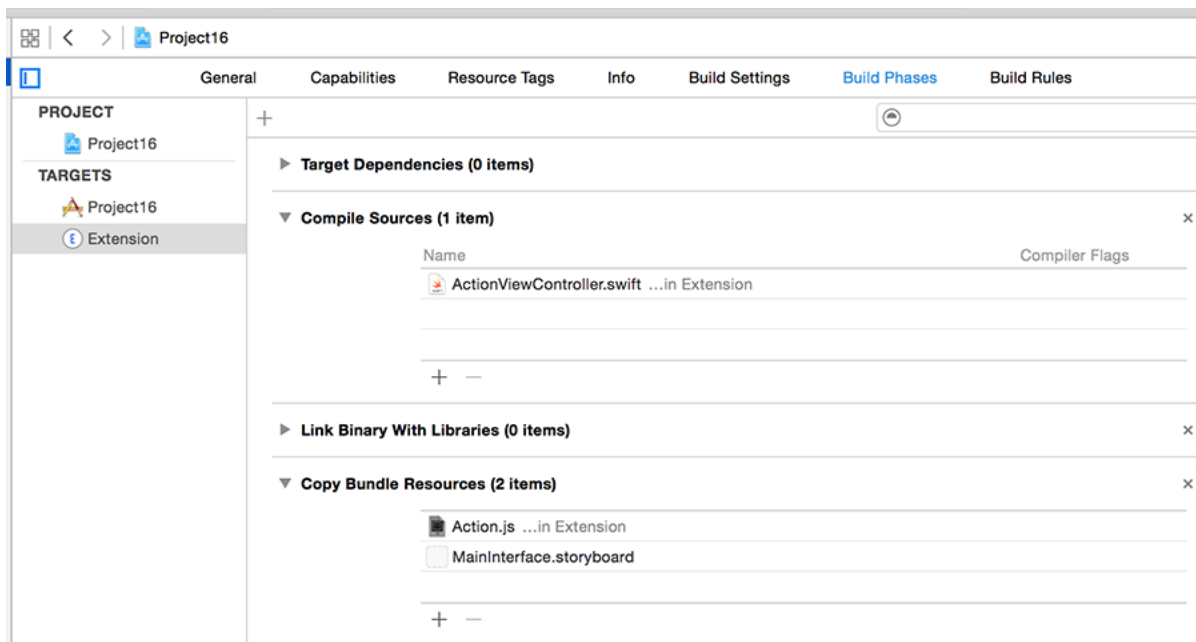
Your Action.js file needs to be in the Copy Bundle Resources build phase of your extension.

Having problems?One reader reported that Xcode had tried to compile Action.js rather than copy it into the project, which will cause problems when you try to run the extension. If you're worried that this might have happened to you, it's easy enough to check: choose your project from the Project Navigator, then choose your extension from the list of targets – it's just called Extension if you followed my instructions so far.

Now choose the Build Phases tab and open up Compile Sources and Copy Bundle Resources. If things have worked correctly you should see Action.js under Copy Bundle Resources and *not* Compile Sources. If this isn't the case, you can just drag it to move.

Establishing communication

To begin with, all we're going to do is send some data from Safari to our extension to make sure everything is set up correctly – after all, it's been quite a bit of hassle so far with nothing to show for it!

First, we're going to modify Action.js to send two pieces of data to our extension: the URL the user was visiting, and the title of the page. Go to Action.js and modify the **run()** function to this:

```
run: function(parameters) {

  parameters.completionFunction({"URL": document.URL, "title":
document.title });

},
```

JavaScript is quite a murky language, so you might be staring at that blankly. If I were to put it in plain English, what it means is "tell iOS the JavaScript has finished preprocessing, and give this data dictionary to the extension." The data that is being sent has the keys "URL" and "title", with the values being the page URL and page title.

As with the previous JavaScript, don't worry about the nitty-gritty. Tthere are many volumes of books on learning JavaScript and I don't intend to repeat them here.

Now that data is being sent from JavaScript, data will be received in Swift. In ActionViewController.swift, replace the **// do stuff!** comment with this:

```
let itemDictionary = dict as! NSDictionary

let javaScriptValues =
itemDictionary[NSExtensionJavaScriptPreprocessingResultsKey] as!
NSDictionary

print(javaScriptValues)
```

Before I explain what that code does, please run the code. I'm saying this because if you're like me then you're probably desperate to see anything working at this point, so let's at least

make sure things are working. When you press Run, wait for the list of host applications to finish loading, then select Safari and click Run.
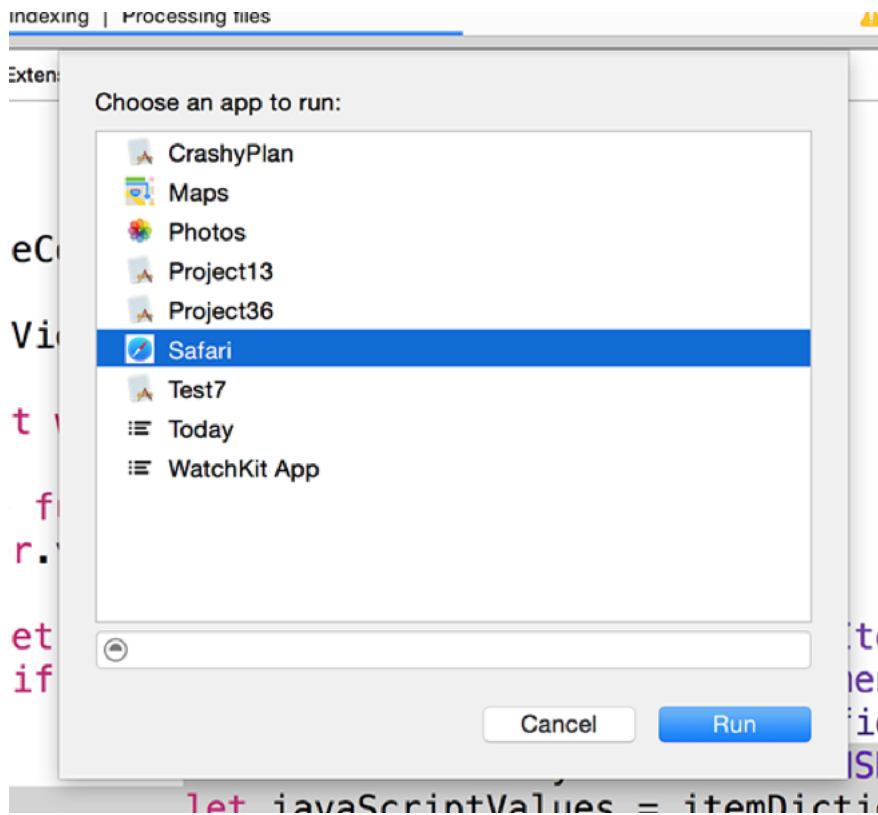
Xcode will ask you which app you should run with your extension. Please choose Safari.

When Safari loads, go to any web page, then tap the action toolbar button on the bottom – it's the box with an arrow coming out. You'll see two rows of icons: swipe to the right on the bottom row. If your extension isn't listed, click More and enable it there. Don't be surprised if you see strange errors being printed out in the Xcode debug console while you're doing this, because Apple sometimes likes to spout unhelpful warnings in their own code.

In the picture below you can see where your extension's icon should appear inside Safari's action menu. If you don't see it there, tap More.
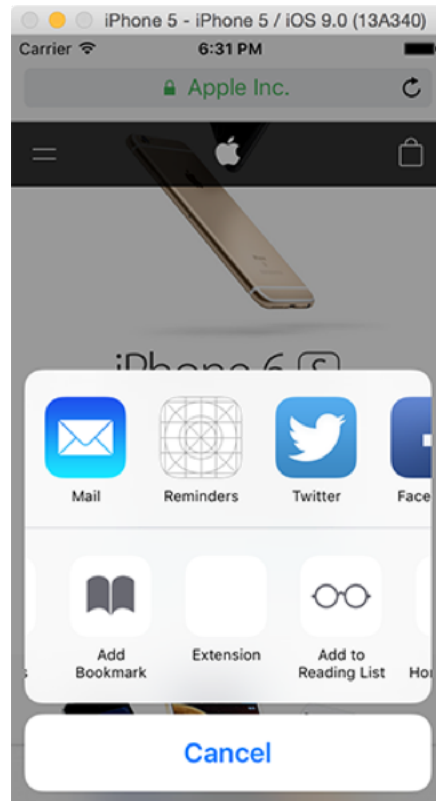
Look for your extension inside Safari's action menu.

When your app runs, you should see messages in the console at the bottom of your Xcode window. If the console isn't visible, use Shift+Cmd+C to activate it, and you should see something like this:

```
{
    URL = "http://www.apple.com/retail/code/";
    title = "Apple Retail Store - Hour of Code Workshop";
}
```

If you're seeing that, well done – your extension is working! If not, you screwed up somewhere, so check my steps again.

Let's take a look at the code. As a reminder, here it is again:

```
let itemDictionary = dict as! NSDictionary

let javaScriptValues =
itemDictionary[NSExtensionJavaScriptPreprocessingResultsKey] as!
NSDictionary

print(javaScriptValues)
```

I've previously spoken about how Swift's String type can be used interchangeably with **NSString**, and how Swift's Array type can be used interchangeably with **NSArray** – both of these are Objective C's versions of the Swift structures. Well, the Objective C version of Swift's Dictionary is called **NSDictionary**, and it's what we receive from Safari.

One of the nice things about **NSDictionary** is that you don't need to declare or even know what data types it holds. One of the nasty things about **NSDictionary** is that you don't need to declare or even know what data types it holds. Yes, it's both an advantage and a disadvantage in one. When working with extensions, however, it's definitely an advantage because we just don't care what's in there – we just want to pull out our data.

When you use **loadItemForTypeIdentifier()**, your closure will be called with the data that was received from the extension along with any error that occurred. Apple could provide other data too, so what we get is a dictionary of data that contains all the information Apple wants us to have, and we put that into **itemDictionary**.

Right now, there's nothing in that dictionary other than the data we sent from JavaScript, and that's stored in a special key called **NSExtensionJavaScriptPreprocessingResultsKey**. So, we pull that value out from the dictionary, and put it into a value called **javaScriptValues**.

We sent a dictionary of data from JavaScript, so we typecast **javaScriptValues** as an **NSDictionary** again so that we can pull out values using keys, but for now we just send the whole lot to the **print()** function, which dumps the dictionary contents to Xcode's debug console.

So, we've successfully proved that Safari is sending data to our extension; it's time to do something interesting with it!

Hacking with JavaScript

Our extension is going to let users type in JavaScript, so before we get onto more coding we're going to add a basic user interface. Open MainInterface.storyboard, then delete its UIImageView and navigation bar. Once that's done, embed the view controller in a navigation controller.

We're going to use a new UIKit component called **UITextView**. You already met **UITextField** in project 5, and it's useful for letting users enter text into a single-line text box. But if you want multiple lines of text, you need **UITextView**, so search for "textview" in the object library and drag one into your view so that it takes up all the space. Delete the template "Lorem ipsum" text that is in there.

Use Resolve Layout Issues > Add Missing Constraints to add automatic Auto Layout constraints. If you find you're having trouble with text going behind the navigation bar, try deleting the top vertical constraint and replacing it with a constraint against the top layout guide. Now use the assistant editor to create an outlet named **script** for the text view in ActionViewController.swift, and while you're there you can delete the **UIImageView** outlet that Xcode made for you.

That's everything for Interface Builder, so switch back to the standard editor, open ActionViewController.swift and add these two properties to your class:

```
var pageTitle = ""
var pageURL = ""
```

We're going to store these two because they are being transmitted by Safari. We'll use the page title to show useful text in the navigation bar, and the URL is there for you to use yourself if you make improvements.

You already saw that we're receiving the data dictionary from Safari, because we used the **print()** function to output its values. Replace the **print()** call with this:

```
self.pageTitle = javaScriptValues["title"] as! String
self.pageURL = javaScriptValues["URL"] as! String
```

```
dispatch_async(dispatch_get_main_queue()) {
    self.title = self.pageTitle
}
```

That sets our two properties from the **javaScriptValues** dictionary, typecasting them as String. It then uses **dispatch_async()** to set the view controller's **title** property on the main queue. This is needed because the closure being executed as a result of **loadItemForTypeIdentifier()** could be called on anything thread, and we don't want to change the UI unless we're on the main thread.

You might have noticed that I haven't written **[unowned self] in** for the **dispatch_async()** call, and that's because it's not needed. The closure will capture the variables it needs, which includes **self**, but we're already inside a closure that has declared **self** to be unowned, so this new closure will use that.

We can immediately make our app useful by modifying the **done()** method. It's been there all along, but I've been ignoring it because there's so much other prep to do just to get out of first gear, but it's now time to turn our eyes towards it and add some functionality.

The **done()** method was originally called as an action from the storyboard, but we deleted the navigation bar Apple put in because it's terrible. Instead, let's create a **UIBarButtonItem** in code, and make that call **done()** instead. Put this in **viewDidLoad()**:

```
navigationItem.rightBarButtonItem =
UIBarButtonItem(barButtonSystemItem: .Done, target: self, action:
"done")
```

Right now, **done()** just has one line of code, which is this:

```
self.extensionContext!.completeRequestReturningItems(self.extensionCo
ntext!.inputItems, completionHandler: nil)
```

Calling **completeRequestReturningItems()** on our extension context will cause the extension to be closed, returning back to the parent app. However, it will pass back to the parent app any items that we specify, which in the current code is the same items that were sent in.

In a Safari extension like ours, the data we return here will be passed in to the **finalize()** function in the Action.js JavaScript file, so we're going to modify the **done()** method so that it passes back the text the user entered into our text view.

To make this work, we need to:

 • Create a new **NSExtensionItem** object that will host our items.
 • Create a dictionary containing the key "customJavaScript" and the value of our script.
 • Put that dictionary into *another* dictionary with the key **NSExtensionJavaScriptFinalizeArgumentKey**.
 • Wrap the big dictionary inside an **NSItemProvider** object with the type identifier **kUTTypePropertyList**.
 • Place that **NSItemProvider** into our **NSExtensionItem** as its attachments.
 • Call **completeRequestReturningItems()**, returning our **NSExtensionItem**.

I realise that seems like far more effort than it ought to be, but it's really just the reverse of what we are doing inside **viewDidLoad()**.

With all that in mind, rewrite your **done()** method to this:

```
@IBAction func done() {

  let item = NSExtensionItem()

  let webDictionary = [NSExtensionJavaScriptFinalizeArgumentKey:
["customJavaScript": script.text]]

  let customJavaScript = NSItemProvider(item: webDictionary,
typeIdentifier: kUTTypePropertyList as String)

  item.attachments = [customJavaScript]


  extensionContext!.completeRequestReturningItems([item],
completionHandler: nil)

}
```

That's all the code required to send data back to Safari, at which point it will appear inside the **finalize()** function in Action.js. From there we can do what we like with it, but in this project the JavaScript we need to write is remarkably simple: we pull the "customJavaScript" value out of the **parameters** array, then pass it to the JavaScript **eval()** function, which executes any code it finds.

Open Action.js, and change the **finalize()** function to this:

```
finalize: function(parameters) {
  var customJavaScript = parameters["customJavaScript"];
  eval(customJavaScript);
}
```

That's it! Our user has written their code in our extension, tapped Done, and it gets executed in Safari using **eval()**. If you want to give it a try, enter the code **alert(document.title);** into the extension. When you tap Done, you'll return to Safari and see the page title in a message box.

Fixing the keyboard

Before we're done, there's a bug in our extension, and it's a bad one – or at least it's bad once you spot it. You see, when you tap to edit a text view, the iOS keyboard automatically appears so that user can start typing. But if you try typing lots, you'll notice that you can actually type underneath the keyboard because the text view hasn't adjusted its size because the keyboard appeared.

If you don't see a keyboard when you tap to edit, it probably means you have the Connect Hardware Keyboard setting turned on. Press Shift+Cmd+K to disable the hardware keyboard and use the on-screen one.

Having our view adjust to the presence of a keyboard is tricky, because there are a number of situations you need to cope with. For example, various keyboards are different heights, the user can rotate their device at will, they can connect a hardware keyboard when they need to, and there's even the QuickType bar that can be shown or hidden on demand.

In all the years I've done iOS development, I've seen at least a dozen ways of coping with keyboards, and few of them are easy. Even Apple's example solution requires fiddling around with constraints, which isn't ideal. I've tried to put together a solution that copes with all possibilities and also requires as little code as possible. If you manage to find something even simpler, do let me know!

We can ask to be told when the keyboard state changes by using a new class called **NSNotificationCenter**. Behind the scenes, iOS is constantly sending out notifications when things happen – keyboard changing, application moving to the background, as well as any custom events that applications post. We can add ourselves as an observer for certain notifications and a method we name will be called when the notification occurs, and will even be passed any useful information.

When working with the keyboard, the notifications we care about are **UIKeyboardWillHideNotification** and **UIKeyboardWillChangeFrameNotification**. The first will be sent when the keyboard has finished hiding, and the second will be shown when any keyboard state change happens – including showing and hiding, but also orientation, QuickType and more.

It might sound like we don't need **UIKeyboardWillHideNotification** if we have **UIKeyboardWillChangeFrameNotification**, but in my testing just using **UIKeyboardWillChangeFrameNotification** isn't enough to catch a hardware keyboard

being connected. Now, that's an extremely rare case, but we might as well be sure!

To register ourselves as an observer for a notification, we get a reference to the default notification center. We then use the **addObserver()** method, which takes four parameters: the object that should receive notifications (it's **self**), the method that should be called, the notification we want to receive, and the object we want to watch. We're going to pass **nil** to the last parameter, meaning "we don't care who sends the notification."

So, add this code to **viewDidLoad()**:

```
let notificationCenter = NSNotificationCenter.defaultCenter()

notificationCenter.addObserver(self, selector: "adjustForKeyboard:",
name: UIKeyboardWillHideNotification, object: nil)

notificationCenter.addObserver(self, selector: "adjustForKeyboard:",
name: UIKeyboardWillChangeFrameNotification, object: nil)
```

The **adjustForKeyboard()** method is complicated, but that's because it has quite a bit of work to do. First, it will receive a parameter that is of type **NSNotification**. This will include the name of the notification as well as an **NSDictionary** containing notification-specific information called **userInfo**.

When working with keyboards, the dictionary will contain a key called **UIKeyboardFrameEndUserInfoKey** telling us the frame of the keyboard after it has finished animating. This will be of type **NSValue**, which in turn is of type **CGRect**. The **CGRect** struct holds both a **CGPoint** and a **CGSize**, so it can be used to describe a rectangle.

One of the quirks of Objective C was that arrays and dictionaries couldn't contain structures like **CGRect**, so Apple had a special class called **NSValue** that acted as a wrapper around structures so they could be put into dictionaries and arrays. That's what's happening here: we're getting an **NSValue** object, but we know it contains a **CGRect** inside so we use its **CGRectValue()** method to pull out that value.

Once we finally pull out the correct frame of the keyboard, we need to convert the rectangle to our view's co-ordinates. This is because rotation isn't factored into the frame, so if the user is in landscape we'll have the width and height flipped – using the **convertView()** method will fix that.

The next thing we need to do in the **adjustForKeyboard()** method is to adjust the **contentInset** and **scrollIndicatorInsets** of our text view. These two essentially indent the edges of our text view so that it appears to occupy less space even though its constraints are still edge to edge in the view.

Finally, we're going to make the text view scroll so that the text entry cursor is visible. If the text view has shrunk this will now be off screen, so scrolling to find it again keeps the user experience intact.

It's not a lot of code, but it *is* complicated – par for the course on this project, it seems. Anyway, here's the method:

```
func adjustForKeyboard(notification: NSNotification) {

  let userInfo = notification.userInfo!


  let keyboardScreenEndFrame =
(userInfo[UIKeyboardFrameEndUserInfoKey] as! NSValue).CGRectValue()

  let keyboardViewEndFrame = view.convertRect(keyboardScreenEndFrame,
fromView: view.window)


  if notification.name == UIKeyboardWillHideNotification {

    script.contentInset = UIEdgeInsetsZero

  } else {

    script.contentInset = UIEdgeInsets(top: 0, left: 0, bottom:
keyboardViewEndFrame.height, right: 0)

  }


  script.scrollIndicatorInsets = script.contentInset


  let selectedRange = script.selectedRange

  script.scrollRangeToVisible(selectedRange)

}
```

As you can see, setting the inset of a text view is done using the **UIEdgeInsets** struct, which needs insets for all four edges. I'm using the text view's content inset for its **scrollIndicatorInsets** to save time.

Note there's a check in there for **UIKeyboardWillHideNotification**, and that's the workaround for hardware keyboards being connected by explicitly setting the insets to be zero.

Wrap up

I'll tell you what: *I'm* feeling tired and I didn't even have to learn anything to write this project – I can't imagine how tired you are! But please don't be too disheartened: extensions are new in iOS 8 and developers are still trying to figure out the best way to use them.

Some of the code isn't pleasant to work with, and certainly I wish iOS would just figure out text view insets automatically for keyboards, but you're through it now so your project is done. Even though this was a hard project, I did cut quite a few corners in this project to make the code as easy as possible, so again I want to encourage you to try creating another extension and see how Apple's example code is different from mine.

If you'd like to make improvements to this project, you could try combining a number of techniques together to make a pretty awesome app. You're already receiving the URL of the site the user is on, so why not use **NSUserDefaults** to save the user's JavaScript for each site? You should convert the URL to an **NSURL** object in order to use its **host** property. If you wanted to be really fancy, you could let users name their scripts, then select one to load view using a **UITableView**.

Project 17
Swifty Ninja

Learn to draw shapes in SpriteKit while making a fun and tense slicing game.

Setting up

I don't want to put you off, but this is by far the longest project in the series. It's not the most *complicated*, but it's long, coming in just short of 500 lines in total. That said, I hope it'll be worth it, because the end result is great: we're going to make a Fruit Ninja-style game, where slicing penguins is good and slicing bombs is bad. I think I must unconsciously have something against penguins…

Anyway, in this project you're going to be creating your own enums for the first time, you're going to meet **SKShapeNode** and **AVAudioPlayer**, you're going to create **SKAction** groups, you're going to create shapes with **UIBezierPath**, learn about default parameters, and more. So, it's the usual recipe: make something cool, and learn at the same time.

This is the second of two projects that are hard – not because I'm trying to set you back, just because they are more complex than the others. This project is hard because you need to write a lot of code before you can start to see results, which I personally find frustrating. I much prefer it when I can write a few lines, see the result, write a few lines more, see the result again, and so on. That isn't possible here, so I suggest you make some coffee before you begin.

Still here? OK!

Create a new SpriteKit project in Xcode, name it Project17, set its target to be iPad, then do the usual cleaning job: remove all the code from **didMoveToView()** and **touchesBegan()**, then delete the spaceship graphics from Images.xcassets. Please also download the files for this project from **GitHub**, then copy its Content folder and Helper.swift files into your Xcode project.

Please force the app to be landscape only before continuing.

Warning: When working with SpriteKit projects, I strongly recommend you use the iPad 2 simulator rather than iPad Air – you'll get much faster frame rates, making it much more suitable for testing.

Basics quick start

The only way we can get through this project with our sanity intact is by whizzing through the things you know already so I can spend more time focusing on the new bits. So, be prepared for abrupt changes of pace: fast, slow, fast, slow, as appropriate.

Open up GameScene.swift and put this into **didMoveToView()**:

```
let background = SKSpriteNode(imageNamed: "sliceBackground")
background.position = CGPoint(x: 512, y: 384)
background.blendMode = .Replace
background.zPosition = -1
addChild(background)

physicsWorld.gravity = CGVector(dx: 0, dy: -6)
physicsWorld.speed = 0.85

createScore()
createLives()
createSlices()
```

The last three are all methods we'll create in a moment, but first there are two new lines in there. The default gravity of our physics world is -0.98, which is roughly equivalent to Earth's gravity. I'm using a slightly lower value so that items stay up in the air a bit longer.

Gravity is expressed using a new data type called **CGVector**, which looks and works like a **CGPoint** except it takes "delta x" and "delta y" as its parameters. "Delta" is a fancy way of saying "difference", in this case from 0. Vectors are best visualised like an arrow that has its base always at 0,0 and its tip at the point you specify. We're specifying X:0 and Y:-6, so our vector arrow is pointing straight down.

I'm also telling the physics world to adjust its speed downwards, which causes all movement to happen at a slightly slower rate.

The first two new methods are easy and require little explanation, but you will need to add some properties to the **GameScene** class to support them:

```
var gameScore: SKLabelNode!

var score: Int = 0 {

  didSet {

    gameScore.text = "Score: \(score)"

  }

}


var livesImages = [SKSpriteNode]()

var lives = 3
```

That's all old news for you – if nothing else, this should show how far you've come! Now here are the two new methods:

```
func createScore() {

  gameScore = SKLabelNode(fontNamed: "Chalkduster")

  gameScore.text = "Score: 0"

  gameScore.horizontalAlignmentMode = .Left

  gameScore.fontSize = 48


  addChild(gameScore)


  gameScore.position = CGPoint(x: 8, y: 8)

}


func createLives() {

  for i in 0 ..< 3 {
```

```
    let spriteNode = SKSpriteNode(imageNamed: "sliceLife")
    spriteNode.position = CGPoint(x: CGFloat(834 + (i * 70)), y: 720)
    addChild(spriteNode)


    livesImages.append(spriteNode)
  }
}
```

You'll notice I'm adding the lives images to the **livesImages** array, which is done so that we can cross off lives when the player loses.

Our game interface is quite simple: score on the bottom left, lives on the top right, and a big empty area in the middle where we can smite penguins.

That leaves the **createSlices()** method, and this bit *is* new. In this game, swiping around the screen will lead a glowing trail of slice marks that fade away when you let go or keep on

moving. To make this work, we're going to do three things:

1. Track all player moves on the screen, recording an array of all their swipe points.
2. Draw two slice shapes, one in white and one in yellow to make it look like there's a hot glow.
3. Use the **zPosition** property that you met in project 11 to make sure the slices go above everything else in the game.

Drawing a shape in SpriteKit is easy thanks to a special node type called **SKShapeNode**. This lets you define any kind of shape you can draw, along with line width, stroke color and more, and it will render it to the screen. We're going to draw two lines – one for a yellow glow, and one for a white glow in the middle of the yellow glow – so we're going to need two **SKShapeNode** properties:

```
var activeSliceBG: SKShapeNode!
var activeSliceFG: SKShapeNode!
```

And here's the code for the **createSlices()** method:

```
func createSlices() {
    activeSliceBG = SKShapeNode()
    activeSliceBG.zPosition = 2


    activeSliceFG = SKShapeNode()
    activeSliceFG.zPosition = 2


    activeSliceBG.strokeColor = UIColor(red: 1, green: 0.9, blue: 0,
alpha: 1)
    activeSliceBG.lineWidth = 9


    activeSliceFG.strokeColor = UIColor.whiteColor()
    activeSliceFG.lineWidth = 5
```

```
    addChild(activeSliceBG)

    addChild(activeSliceFG)

}
```

Note that the background slice has a thicker line width than the foreground, and we have to add the background one first. I'm using Z position 2 for the slice shapes, because I'll be using Z position 1 for bombs and Z position 0 for everything else – this ensures the slice shapes are on top, then bombs, then everything else.

Shaping up for action

Like I already explained, we're going to keep an array of the user's swipe points so that we can draw a shape resembling their slicing. To make this work, we're going to need five new methods, one of which you've met already. They are: **touchesBegan()**, **touchesMoved()**, **touchesEnded()**, **touchesCancelled()** and **redrawActiveSlice()**.

You already know how **touchesBegan()** works, and the other three "touches" methods all work the same way. There's a subtle difference between **touchesEnded()** and **touchesCancelled()**: the former is called when the user stops touching the screen, and the latter is called if the system has to interrupt the touch for some reason – e.g. if a low memory warning appears. We're going to make **touchesCancelled()** just call **touchesEnded()**, to avoid duplicating code.

First things first: add this new property to your class so that we can store swipe points:

```
var activeSlicePoints = [CGPoint]()
```

We're going to tackle the three easiest methods first: **touchesMoved()**, **touchesEnded()** and **touchesCancelled()**. All the touches **touchesMoved()** method needs to do is figure out where in the scene the user touched, add that location to the slice points array, then redraw the slice shape, so that's easy enough:

```
override func touchesMoved(touches: Set<UITouch>, withEvent event:
UIEvent?) {
  guard let touch = touches.first else { return }

  let location = touch.locationInNode(self)

  activeSlicePoints.append(location)
  redrawActiveSlice()
}
```

When the user finishes touching the screen, **touchesEnded()** will be called. I'm going to make this method fade out the slice shapes over a quarter of a second because we either remove them immediately (which looks ugly) or we leave them sitting there for no reason (which destroys the effect). So, add this **touchesEnded()** method:

```
override func touchesEnded(touches: Set<UITouch>?, withEvent event:
UIEvent?) {

  activeSliceBG.runAction(SKAction.fadeOutWithDuration(0.25))

  activeSliceFG.runAction(SKAction.fadeOutWithDuration(0.25))

}
```

You haven't used the **fadeOutWithDuration()** action before, but I think it's pretty obvious what it does!

The third easy function is **touchesCancelled()**, and it's easy because we're just going to forward it on to **touchesEnded()** like this:

```
override func touchesCancelled(touches: Set<UITouch>?, withEvent
event: UIEvent?) {

  if let touches = touches {

    touchesEnded(touches, withEvent: event)

  }

}
```

So far this is all easy stuff, but we're going to look at an interesting method now: **touchesBegan()**. This needs to do several things:

1. Remove all existing points in the **activeSlicePoints** array, because we're starting fresh.
2. Get the touch location and add it to the **activeSlicePoints** array.
3. Call the (as yet unwritten) **redrawActiveSlice()** method to clear the slice shapes.
4. Remove any actions that are currently attached to the slice shapes. This will be important if they are in the middle of a **fadeOutWithDuration()** action.
5. Set both slice shapes to have an alpha value of 1 so they are fully visible.

We can convert that to code with ease – in fact, I've put numbered comments in the code below so you can match them up to the points above:

```
override func touchesBegan(touches: Set<UITouch>, withEvent event:
UIEvent?) {
  super.touchesBegan(touches, withEvent: event)

  // 1
  activeSlicePoints.removeAll(keepCapacity: true)

  // 2
  if let touch = touches.first {
    let location = touch.locationInNode(self)
    activeSlicePoints.append(location)

    // 3
    redrawActiveSlice()

    // 4
    activeSliceBG.removeAllActions()
    activeSliceFG.removeAllActions()

    // 5
    activeSliceBG.alpha = 1
    activeSliceFG.alpha = 1
  }
}
```

So, there's some challenge there but not a whole lot. Where it gets interesting is the

redrawActiveSlice() method, because this is going to use a new class called **UIBezierPath** that will be used to connect our swipe points together into a single line.

As with the previous method, let's take a look at what **redrawActiveSlice()** needs to do:

1. If we have fewer than two points in our array, we don't have enough data to draw a line so it needs to clear the shapes and exit the method.
2. If we have more than 12 slice points in our array, we need to remove the oldest ones until we have at most 12 – this stops the swipe shapes from becoming too long.
3. It needs to start its line at the position of the first swipe point, then go through each of the others drawing lines to each point.
4. Finally, it needs to update the slice shape paths so they get drawn using their designs – i.e., line width and color.

To make this work, you're going to need to know that an **SKShapeNode** object has a property called **path** which describes the shape we want to draw. When it's **nil**, there's nothing to draw; when it's set to a valid path, that gets drawn with the **SKShapeNode**'s settings. **SKShapeNode** expects you to use a data type called **CGPath**, but we can easily create that from a **UIBezierPath**.

Drawing a path using **UIBezierPath** is a cinch: we'll use its **moveToPoint()** method to position the start of our lines, then loop through our **activeSlicePoints** array and call the path's **addLineToPoint()** method for each point.

To stop the array storing more than 12 slice points, we're going to use a new loop type called a **while** loop. This loop will continue executing until its condition stops being true, so we'll just give the condition that **activeSlicePoints** has more than 12 items, then ask it to remove the first item until the condition fails.

I'm going to insert numbered comments into the code again to help you match up the goals with the code more easily:

```
func redrawActiveSlice() {
  // 1
  if activeSlicePoints.count < 2 {
    activeSliceBG.path = nil
    activeSliceFG.path = nil
```

```
    return
  }


  // 2
  while activeSlicePoints.count > 12 {
    activeSlicePoints.removeAtIndex(0)
  }


  // 3
  let path = UIBezierPath()
  path.moveToPoint(activeSlicePoints[0])


  for i in 1 ..< activeSlicePoints.count {
    path.addLineToPoint(activeSlicePoints[i])
  }


  // 4
  activeSliceBG.path = path.CGPath
  activeSliceFG.path = path.CGPath
}
```

At this point, we have something you can run: press Cmd+R to run the game, then tap and swipe around on the screen to see the slice effect – I think you'll agree that **SKShapeNode** is pretty powerful!
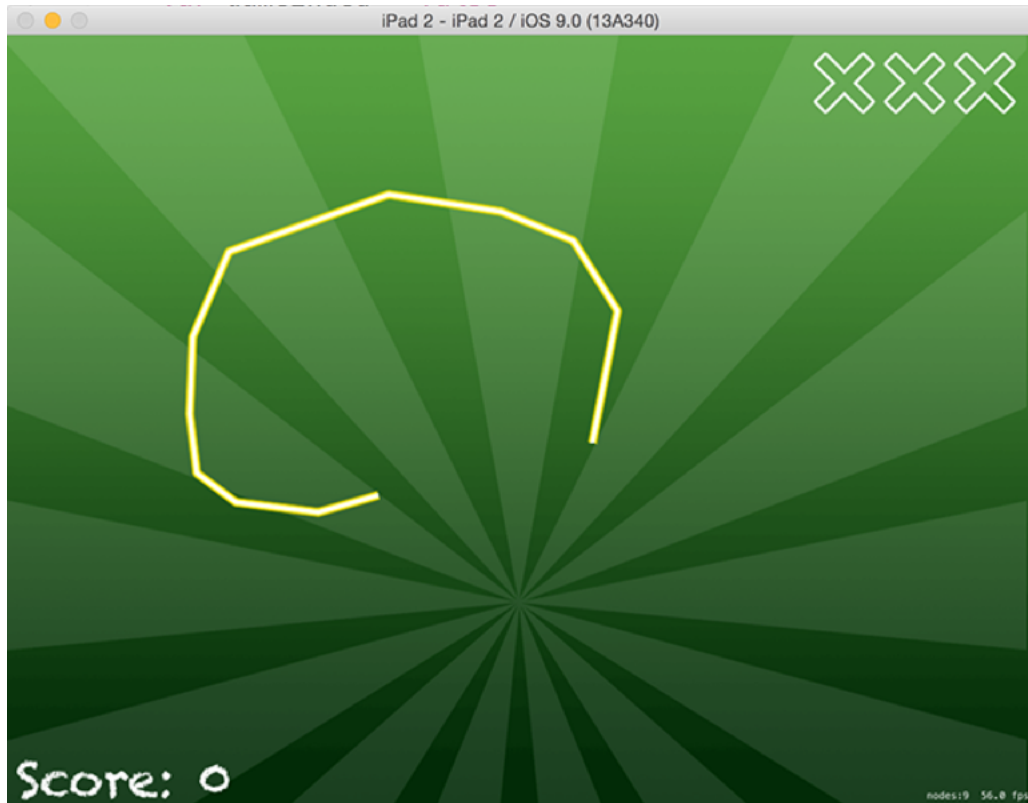
As the player swipes, their slices light up the screen in a bright yellow curve.

Before we're done with the slice effect, we're going to add one more thing: a "swoosh" sound that plays as you swipe around. You've already seen the **playSoundFileNamed()** method of **SKAction**, but we're going to use it a little differently here.

You see, if we just played a swoosh every time the player moved, there would be 100 sounds playing at any given time – one for every small movement they made. Instead, we want only one swoosh to play at once, so we're going to set to true a property called **swooshSoundActive**, make the **waitForCompletion** of our **SKAction** true, then use a completion closure for **runAction()** so that **swooshSoundActive** is set to false.

So, when the player first swipes we set **swooshSoundActive** to be true, and only when the swoosh sound has finished playing do we set it back to false again. This will allow us to ensure only one swoosh sound is playing at a time.

First, give your class this new property:

```
var swooshSoundActive = false
```

Now we need to check whether that's false when **touchesMoved()** is called, and, if it is false, call a new method called **playSwooshSound()**. Add this to code just before the end of **touchesMoved()**:

```
if !swooshSoundActive {
  playSwooshSound()
}
```

I've provided you with three different swoosh sounds, all of which are effectively the same just at varying pitches. The **playSwooshSound()** method needs to set **swooshSoundActive** to be true (so that no other swoosh sounds are played until we're ready), play one of the three sounds, then when the sound has finished set **swooshSoundActive** to be false again so that another swoosh sound can play.

By playing our sound with **waitForCompletion** set to true, SpriteKit automatically ensures the completion closure given to **runAction()** isn't called until the sound has finished, so this solution is perfect.

```
func playSwooshSound() {
  swooshSoundActive = true


  let randomNumber = RandomInt(min: 1, max: 3)
  let soundName = "swoosh\(randomNumber).caf"


  let swooshSound = SKAction.playSoundFileNamed(soundName,
waitForCompletion: true)


  runAction(swooshSound) { [unowned self] in
    self.swooshSoundActive = false
```

```
    }
}
```

Enemy or bomb?

In this section we're going to look at just one method, which should tell you immediately that this is a *complicated* method. This method is called **createEnemy()**, and is responsible for launching either a penguin or a bomb into the air for the player to swipe. That's it – that's all it does. And yet it's going to take quite a lot of code because it takes quite a lot of functionality in order to make the game complete:

1. Should this enemy be a penguin or a bomb?
2. Where should be it created on the screen?
3. What direction should it be moving in?

It should be obvious that 3) relies on 2) – if you create something on the left edge of the screen, having it move to the left would make the game impossible for players!

An additional complexity is that in the early stages of the game we sometimes want to force a bomb, and sometimes force a penguin, in order to build a smooth learning curve. For example, it wouldn't be fair to make the very first enemy a bomb, because the player would swipe it and lose immediately.

We're going to specify what kind of enemy we want using an enum. You've used enums already (not least in project 2), but you've never created one before. To make **createEnemy()** work, we need to declare a new enum that tracks what kind of enemy should be created: should we force a bomb always, should we force a bomb never, or use the default randomisation?

Add this *above* your class definition in GameScene.swift:

```
enum ForceBomb {
    case Never, Always, Default
}
```

You can now use those values in your code, for example like this:

```
if forceBomb == .Never {
```

```
    enemyType = 1

} else if forceBomb == .Always {

    enemyType = 0

}
```

OK, it's time to start looking at the **createEnemy()** method. I say "start" because we're going to look at it in three passes: the code required to create bombs, the code to position enemies and set up their physics, and the code required to do everything else. Your code probably won't run until all three parts are in place, so don't worry!

We're going to need to track enemies that are currently active in the scene, so please add this array as a property of your class:

```
var activeEnemies = [SKSpriteNode]()
```

And now let's look at the core of the **createEnemy()** method. It needs to:

1. Accept a parameter of whether we want to force a bomb, not force a bomb, or just be random.
2. Decide whether to create a bomb or a penguin (based on the parameter input) then create the correct thing.
3. Add the new enemy to the scene, and also to our **activeEnemies** array.

That's it. Not too much, I hope. To decide whether to create a bomb or a player, I'll choose a random number from 0 to 6, and consider 0 to mean "bomb". Here's the code:

```
func createEnemy(forceBomb forceBomb: ForceBomb = .Default) {

    var enemy: SKSpriteNode


    var enemyType = RandomInt(min: 0, max: 6)


    if forceBomb == .Never {
```

```
    enemyType = 1

  } else if forceBomb == .Always {

    enemyType = 0

  }


  if enemyType == 0 {

    // bomb code goes here

  } else {

    enemy = SKSpriteNode(imageNamed: "penguin")

    runAction(SKAction.playSoundFileNamed("launch.caf",
waitForCompletion: false))

    enemy.name = "enemy"

  }


  // position code goes here


  addChild(enemy)

  activeEnemies.append(enemy)

}
```

You may have spotted that the **forceBomb** parameter is specified with a default value of **.Default**, as seen at the end of project 2.

Other than that, there's nothing complicated in there, but I *have* taken out two fairly meaty chunks of code. That **// position code goes here** comment masks a lot of missing functionality that really makes the game come alive, so we're going to fill that in now.

I'm going to use numbered comments again so you can see exactly how this code matches up with what it should do. So, here is what that missing position code needs to do:

1. Give the enemy a random position off the bottom edge of the screen.
2. Create a random angular velocity, which is how fast something should spin.
3. Create a random X velocity (how far to move horizontally) that takes into account the

enemy's position.

4. Create a random Y velocity just to make things fly at different speeds.

5. Give all enemies a circular physics body where the **collisionBitMask** is set to 0 so they don't collide.

The only thing that might catch you out in the actual code is my use of magic numbers, which is what programmers call seemingly random (but actually important) numbers appearing in code. Ideally you don't want these, because it's better to make them constants with names, but then how would I be able to give you any homework? (*Evil laugh.*)

Turning those five points into code is easy enough – just replace the **// position code goes here** with this:

```
// 1
let randomPosition = CGPoint(x: RandomInt(min: 64, max: 960), y: -128)
enemy.position = randomPosition


// 2
let  randomAngularVelocity = CGFloat(RandomInt(min: -6, max: 6)) / 2.0
var randomXVelocity = 0


// 3
if randomPosition.x < 256 {
  randomXVelocity = RandomInt(min: 8, max: 15)
} else if randomPosition.x < 512 {
  randomXVelocity = RandomInt(min: 3, max: 5)
} else if randomPosition.x < 768 {
  randomXVelocity = -RandomInt(min: 3, max: 5)
} else {
  randomXVelocity = -RandomInt(min: 8, max: 15)
}
```

```
// 4

let randomYVelocity = RandomInt(min: 24, max: 32)


// 5

enemy.physicsBody = SKPhysicsBody(circleOfRadius: 64)

enemy.physicsBody!.velocity = CGVector(dx: randomXVelocity * 40, dy:
randomYVelocity * 40)

enemy.physicsBody!.angularVelocity = randomAngularVelocity

enemy.physicsBody!.collisionBitMask = 0
```

The last missing part of the **createEnemy()** method is about creating bombs, and I've left it separate because it requires some thinking. A "bomb" node in our game is actually going to be made up of three parts: the bomb image, a bomb fuse particle emitter, and a container that puts the two together so we can move and spin them around together.

The reason we need to keep the bomb image and bomb fuse separate is because tapping on a bomb is a fatal move that causes the player to lose all their lives immediately. If the fuse particle emitter were inside the bomb image, then the user could accidentally tap a stray fuse particle and lose unfairly.

As a reminder, we're going to force the Z position of bombs to be 1, which is higher than the default value of 0. This is so that bombs always appear in front of penguins, because hours of playtesting has made it clear to me that it's awful if you don't realise there's a bomb lurking behind something when you swipe it!

Creating a bomb also needs to play a fuse sound, but that has its own complexity. You've already seen that **SKAction** has a very simple way to play sounds, but it's so simple that it's not useful here because we want to be able to stop the sound and **SKAction** sounds don't let you do that. It would be confusing for the fuse sound to be playing when no bombs are visible, so we need a better solution.

That solution is called **AVAudioPlayer**, and it's not a SpriteKit class – it's available to use in your UIKit apps too if you want. We're going to have an **AVAudioPlayer** property for our class that will store a sound just for bomb fuses so that we can stop it as needed.

Let's put numbers to the tasks this chunk of code needs to perform:

1. Create a new **SKSpriteNode** that will hold the fuse and the bomb image as children, setting its Z position to be 1.
2. Create the bomb image, name it "bomb", and add it to the container.
3. If the bomb fuse sound effect is playing, stop it and destroy it.
4. Create a new bomb fuse sound effect, then play it.
5. Create a particle emitter node, position it so that it's at the end of the bomb image's fuse, and add it to the container.

That's all you need to know, so please replace the **// bomb code goes here** comment with this, watching out for my numbered comments to help you match code against meaning:

```
// 1
enemy = SKSpriteNode()

enemy.zPosition = 1

enemy.name = "bombContainer"


// 2
let bombImage = SKSpriteNode(imageNamed: "sliceBomb")

bombImage.name = "bomb"

enemy.addChild(bombImage)


// 3
if bombSoundEffect != nil {

  bombSoundEffect.stop()

  bombSoundEffect = nil

}


// 4
let path = NSBundle.mainBundle().pathForResource("sliceBombFuse.caf",
ofType:nil)!

let url = NSURL(fileURLWithPath: path)
```

```
let sound = try! AVAudioPlayer(contentsOfURL: url)

bombSoundEffect = sound

sound.play()


// 5

let emitter = SKEmitterNode(fileNamed: "sliceFuse")!

emitter.position = CGPoint(x: 76, y: 64)

enemy.addChild(emitter)
```

Note that I've used **try!** here because if we're unable to read a sound file from our app bundle then clearly something is fatally wrong.

To make that code work, you need to import the **AVFoundation** framework, so go to the top of your file and add this line of code before **import UIKit**:

```
import AVFoundation
```

You'll also need to declare the **bombSoundEffect** property, so put this just after the declaration of **swooshSoundActive**:

```
var bombSoundEffect: AVAudioPlayer!
```

After all that work, you're almost done with bombs. But there's one small bug that we can either fix now or fix when you can see it, but we might as well fix it now because your brain is thinking about all that bomb code.

The bug is this: we're using **AVAudioPlayer** so that we can stop the bomb fuse when bombs are no longer on the screen. But where do we actually stop the sound? Well, we don't yet – but we need to.

To fix the bug, we need to modify the **update()** method, which is something we haven't

touched before. This method is called every frame before it's drawn, and gives you a chance to update your game state as you want. We're going to use this method to count the number of bomb containers that exist in our game, and stop the fuse sound if the answer is 0.

Change your **update()** method to this:

```
override func update(currentTime: CFTimeInterval) {
  var bombCount = 0

  for node in activeEnemies {
    if node.name == "bombContainer" {
      bombCount += 1
      break
    }
  }

  if bombCount == 0 {
    // no bombs – stop the fuse sound!
    if bombSoundEffect != nil {
      bombSoundEffect.stop()
      bombSoundEffect = nil
    }
  }
}
```

Follow the sequence

You've come so far already, and really there isn't a lot to show for your work other than being able to draw glowing slice shapes when you move touches around the screen. But that's all about to change, because we're now about to create the interesting code – we're going to make the game actually create some enemies.

Now, you might very well be saying, "but Paul, we just wrote the enemy creating code, and I never want to see it again!" You're right (and I never want to see it again either!) but it's a bit more complicated: the **createEnemy()** method creates one enemy as required. The code we're going to write now will call **createEnemy()** in different ways so that we get varying groups of enemies.

For example, sometimes we want to create two enemies at once, sometimes we want to create four at once, and sometimes we want to create five in quick sequence. Each one of these will call **createEnemy()** in different ways.

There's a *lot* to cover here, so let's get started: add this new enum before the **ForceBomb** enum you added a few minutes ago:

```
enum SequenceType: Int {

  case OneNoBomb, One, TwoWithOneBomb, Two, Three, Four, Chain,
FastChain

}
```

That outlines the possible types of ways we can create enemy: one enemy that definitely is a bomb, one that might or might not be a bomb, two where one is a bomb and one isn't, then two/three/four random enemies, a chain of enemies, then a fast chain of enemies.

The first two will be used exclusively when the player first starts the game, to give them a gentle warm up. After that, they'll be given random sequence types from **TwoWithOneBomb** to **FastChain**.

We're going to need quite a few new properties in order to make the plan work, so please add these now:

```
var popupTime = 0.9

var sequence: [SequenceType]!

var sequencePosition = 0

var chainDelay = 3.0

var nextSequenceQueued = true
```

And here's what they do:

 • **popupTime** is the amount of time to wait between the last enemy being destroyed and a new one being created.
 • **sequence** is an array of our **SequenceType** enum that defines what enemies to create.
 • **sequencePosition** is where we are right now in the game.
 • **chainDelay** is how long to wait before creating a new enemy when the sequence type is **.Chain** or **.FastChain**. Enemy chains don't wait until the previous enemy is offscreen before creating a new one, so it's like throwing five enemies quickly but with a small delay inbetween each one.
 • **nextSequenceQueued** is used so we know when all the enemies are destroyed and we're ready to create more.

Whenever we call our new method, which is **tossEnemies()**, we're going to decrease both **popupTime** and **chainDelay** so that the game gets harder as they play. Sneakily, we're always going to increase the speed of our physics world, so that objects move rise and fall faster too.

Nearly all the **tossEnemies()** method is a large switch/case statement that looks at the **sequencePosition** property to figure out what sequence type it should so. It then calls **createEnemy()** correctly for the sequence type, passing in whether to force bomb creation or not.

The one thing that will need to be explained is the way enemy chains are created. Unlike regular sequence types, a chain is made up of several enemies with a space between them, and the game doesn't wait for an enemy to be sliced before showing the next thing in the chain.

The best thing for you to do is to put this source code into your project, and we can talk

about the chain complexities in a moment:

```
func tossEnemies() {
    popupTime *= 0.991
    chainDelay *= 0.99
    physicsWorld.speed *= 1.02


    let sequenceType = sequence[sequencePosition]


    switch sequenceType {
    case .OneNoBomb:
        createEnemy(forceBomb: .Never)


    case .One:
        createEnemy()


    case .TwoWithOneBomb:
        createEnemy(forceBomb: .Never)
        createEnemy(forceBomb: .Always)


    case .Two:
        createEnemy()
        createEnemy()


    case .Three:
        createEnemy()
        createEnemy()
        createEnemy()


    case .Four:
```

```
      createEnemy()

      createEnemy()

      createEnemy()

      createEnemy()


  case .Chain:

      createEnemy()


      RunAfterDelay(chainDelay / 5.0) { [unowned self] in
self.createEnemy() }

      RunAfterDelay(chainDelay / 5.0 * 2) { [unowned self] in
self.createEnemy() }

      RunAfterDelay(chainDelay / 5.0 * 3) { [unowned self] in
self.createEnemy() }

      RunAfterDelay(chainDelay / 5.0 * 4) { [unowned self] in
self.createEnemy() }


  case .FastChain:

      createEnemy()


      RunAfterDelay(chainDelay / 10.0) { [unowned self] in
self.createEnemy() }

      RunAfterDelay(chainDelay / 10.0 * 2) { [unowned self] in
self.createEnemy() }

      RunAfterDelay(chainDelay / 10.0 * 3) { [unowned self] in
self.createEnemy() }

      RunAfterDelay(chainDelay / 10.0 * 4) { [unowned self] in
self.createEnemy() }

  }


  sequencePosition += 1
```

```
    nextSequenceQueued = false
}
```

That looks like a massive method, I know, but in reality it's just the same thing being called in different ways. The interesting parts are the **.Chain** and **.FastChain** cases, and also I want to explain in more detail the **nextSequenceQueued** property.

Each sequence in our array creates one or more enemies, then waits for them to be destroyed before continuing. Enemy chains are different: they create five enemies with a short break inbetween, and don't wait for each one to be destroyed before continuing.

To handle these chains, we have calls to **RunAfterDelay()** with a timer value. If we assume for a moment that **chainDelay** is 10 seconds, then:

 • **chainDelay / 10.0** is 1 second.
 • **chainDelay / 10.0 * 2** is 2 seconds.
 • **chainDelay / 10.0 * 3** is three seconds.
 • **chainDelay / 10.0 * 4** is four seconds.

So, it spreads out the **createEnemy()** calls quite neatly.

The **nextSequenceQueued** property is more complicated. If it's false, it means we don't have a call to **tossEnemies()** in the pipeline waiting to execute. It gets set to true only in the gap between the previous sequence item finishing and **tossEnemies()** being called. Think of it as meaning, "I know there aren't any enemies right now, but more will come shortly."

We can make our game come to life with enemies with two more pieces of code. First, add this just before the end of **didMoveToView()**:

```
sequence =
[.OneNoBomb, .OneNoBomb, .TwoWithOneBomb, .TwoWithOneBomb, .Three, .O
ne, .Chain]


for _ in 0 ... 1000 {

  let nextSequence = SequenceType(rawValue: RandomInt(min: 2, max:
7))!
```

```
    sequence.append(nextSequence)

}


RunAfterDelay(2) { [unowned self] in

    self.tossEnemies()

}
```

That code fills the **sequence** array with seven pre-written sequences to help players warm up to how the game works, then adds 1001 (the **...** operator means "up to and including") random sequence types to fill up the game. Finally, it triggers the initial enemy toss after two seconds.

The way we generate random sequence type values is new and quite interesting. If you cast your mind back, this is how we defined the **SequenceType** enum:

```
enum SequenceType: Int {

    case OneNoBomb, One, TwoWithOneBomb, Two, Three, Four, Chain,
FastChain

}
```

Note that it says **enum SequenceType: Int**. We didn't have that for the **ForceBomb** enum – it's new here, and it means "I want this enum to be mapped to integer values," and means we can reference each of the sequence type options using so-called "raw values" from 0 to 7.

For example, to create a **TwoWithOneBomb** sequence type we could use **SequenceType(rawValue: 2)**. Swift doesn't know whether that number exists or not (we could have written 77), so it returns an optional type that you need to unwrap.

The second change we're going to make is to remove enemies from the game when they fall off the screen. This is required, because our game mechanic means that new enemies aren't created until the previous ones have been removed. The exception to this rule are enemy chains, where multiple enemies are created in a batch, but even then the game won't continue until all enemies from the chain have been removed.

We're going to modify the **update()** method so that:

1. If we have active enemies, we loop through each of them.
2. If any enemy is at or lower than Y position -140, we remove it from the game and our **activeEnemies** array.
3. If we don't have any active enemies *and* we haven't already queued the next enemy sequence, we schedule the next enemy sequence and set **nextSequenceQueued** to be true.

Put this code first in the **update()** method:

```
if activeEnemies.count > 0 {

  for node in activeEnemies {

    if node.position.y < -140 {

      node.removeFromParent()


      if let index = activeEnemies.indexOf(node) {

        activeEnemies.removeAtIndex(index)

      }

    }

  }

} else {

  if !nextSequenceQueued {

    RunAfterDelay(popupTime) { [unowned self] in

      self.tossEnemies()

    }


    nextSequenceQueued = true

  }

}
```

And now the part you've been waiting for extremely patiently: press Cmd+R to run the game, because it should now be getting close to useful!

Now that the game has bombs as well as penguins, it's almost starting to come together.

Slice to win

We need to modify **touchesMoved()** to detect when users slice penguins and bombs. The code isn't complicated, but it *is* long, so I'm going to split it into three. First, here's the structure – place this just before the end of **touchesMoved()**:

```
let nodes = nodesAtPoint(location)

for node in nodes {
  if node.name == "enemy" {
    // destroy penguin
  } else if node.name == "bomb" {
    // destroy bomb
  }
}
```

Now, let's take a look at what destroying a penguin should do. It should:

1. Create a particle effect over the penguin.
2. Clear its node name so that it can't be swiped repeatedly.
3. Disable the **dynamic** of its physics body so that it doesn't carry on falling.
4. Make the penguin scale out and fade out at the same time.
5. After making the penguin scale out and fade out, we should remove it from the scene.
6. Add one to the player's score.
7. Remove the enemy from our **activeEnemies** array.
8. Play a sound so the player knows they hit the penguin.

Replace the **// destroy penguin** with this, following along with my numbered comments:

```
// 1
let emitter = SKEmitterNode(fileNamed: "sliceHitEnemy")!
emitter.position = node.position
```

```
addChild(emitter)


// 2

node.name = ""


// 3

node.physicsBody!.dynamic = false


// 4

let scaleOut = SKAction.scaleTo(0.001, duration:0.2)

let fadeOut = SKAction.fadeOutWithDuration(0.2)

let group = SKAction.group([scaleOut, fadeOut])


// 5

let seq = SKAction.sequence([group, SKAction.removeFromParent()])

node.runAction(seq)


// 6

score += 1


// 7

let index = activeEnemies.indexOf(node as! SKSpriteNode)!

activeEnemies.removeAtIndex(index)


// 8

runAction(SKAction.playSoundFileNamed("whack.caf", waitForCompletion:
false))
```

You've now seen the two ways of collecting SpriteKit actions together: groups and sequences. An action *group* specifies that all actions inside it should execute simultaneously,

whereas an action **sequence** runs them all one at a time. In the code above we have a group inside a sequence, which is common.

If the player swipes a bomb by accident, they lose the game immediately. This uses much the same code as destroying a penguin, but with a few differences:

•The node called "bomb" is the bomb image, which is inside the bomb container. So, we need to reference the node's parent when looking up our position, changing the physics body, removing the node from the scene, and removing the node from our **activeEnemies** array..
•I'm going to create a different particle effect for bombs than for penguins.
•We end by calling the (as yet unwritten) method **endGame()**.

Replace the **// destroy bomb** comment with this:

```
let emitter = SKEmitterNode(fileNamed: "sliceHitBomb")!
emitter.position = node.parent!.position
addChild(emitter)


node.name = ""
node.parent!.physicsBody!.dynamic = false


let scaleOut = SKAction.scaleTo(0.001, duration:0.2)
let fadeOut = SKAction.fadeOutWithDuration(0.2)
let group = SKAction.group([scaleOut, fadeOut])


let seq = SKAction.sequence([group, SKAction.removeFromParent()])


node.parent!.runAction(seq)


let index = activeEnemies.indexOf(node.parent as! SKSpriteNode)!
activeEnemies.removeAtIndex(index)
```

```
runAction(SKAction.playSoundFileNamed("explosion.caf",
waitForCompletion: false))

endGame(triggeredByBomb: true)
```

Before I walk you through the **endGame()** method, we need to adjust the **update()** method a little. Right now, if a penguin or a bomb falls below -140, we remove it from the scene. We're going to modify that so that if the player misses slicing a penguin, they lose a life. We're also going to delete the node's name just in case any further checks for enemies or bombs happen – clearing the node name will avoid any problems.

In the **update()** method, replace this code:

```
if node.position.y < -140 {
  node.removeFromParent()

  if let index = activeEnemies.indexOf(node) {
    activeEnemies.removeAtIndex(index)
  }
}
```

…with this:

```
if node.position.y < -140 {
  node.removeAllActions()

  if node.name == "enemy" {
    node.name = ""
    subtractLife()

    node.removeFromParent()
```

```
        if let index = activeEnemies.indexOf(node) {
            activeEnemies.removeAtIndex(index)
        }
    } else if node.name == "bombContainer" {
        node.name = ""
        node.removeFromParent()

        if let index = activeEnemies.indexOf(node) {
            activeEnemies.removeAtIndex(index)
        }
    }
}
```

That's mostly the same, except now we call **subtractLife()** when the player lets any penguins through. So, if you miss a penguin you lose one life; if you swipe a bomb, you lose all your lives. Or at least you would if our code actually compiled, which it won't: you're missing the **subtractLife()** and **endGame()** methods!

Game over, man

You are now within reach of the end of this project, and not a moment too soon, I suspect. You'll be pleased to know that you're just two methods away from the end, and neither of them are particularly taxing.

First is the **subtractLife()** method, which is called when a penguin falls off the screen without being sliced. It needs to subtract 1 from the **lives** property that we created what seems like years ago, update the images in the **livesImages** array so that the correct number are crossed off, then end the game if the player is out of lives.

To make it a bit clearer that something bad has happened, we're also going to add playing a sound and animate the life being lost – we'll set the X and Y scale of the life being lost to 1.3, then animate it back down to 1.0.

Here's the code:

```
func subtractLife() {

  lives -= 1


  runAction(SKAction.playSoundFileNamed("wrong.caf",
waitForCompletion: false))


  var life: SKSpriteNode


  if lives == 2 {

    life = livesImages[0]

  } else if lives == 1 {

    life = livesImages[1]

  } else {

    life = livesImages[2]

    endGame(triggeredByBomb: false)

  }
```

```
    life.texture = SKTexture(imageNamed: "sliceLifeGone")


    life.xScale = 1.3

    life.yScale = 1.3

    life.runAction(SKAction.scaleTo(1, duration:0.1))
}
```

Note how I'm using **SKTexture** to modify the contents of a sprite node without having to recreate it, just like in project 14.

Finally, there's the **endGame()** method. I've made this accept a parameter that sets whether the game ended because of a bomb, and I've forced the label for it using **triggeredByBomb triggeredByBomb** because without the label it's a meaningless method name.

```
func endGame(triggeredByBomb triggeredByBomb: Bool) {
  if gameEnded {
    return
  }


  gameEnded = true
  physicsWorld.speed = 0
  userInteractionEnabled = false

  if bombSoundEffect != nil {
    bombSoundEffect.stop()
    bombSoundEffect = nil
  }


  if triggeredByBomb {
    livesImages[0].texture = SKTexture(imageNamed: "sliceLifeGone")
```

```
        livesImages[1].texture = SKTexture(imageNamed: "sliceLifeGone")

        livesImages[2].texture = SKTexture(imageNamed: "sliceLifeGone")

    }

}
```

If the game hasn't already ended, this code stops every object from moving by adjusting the speed of the physics world to be 0. It stops any bomb fuse fizzing, and sets all three lives images to have the same "life gone" graphic. Nothing surprising in there, but you do need to declare **gameEnded** as a property for your class, like this:

```
var gameEnded = false
```

Even though the game has ended, some actions can still take place. This should be banned if possible, so add these lines to the start of **tossEnemies()** and **touchesMoved()**:

```
if gameEnded {
    return
}
```

That's it, your game is done!

Wrap up

You've just finished two hard projects back to back, and regardless of how much you have learned you deserve kudos for all your patience. This project required you to follow several long steps before you could see your code run. I hope it was worth it, and I hope in retrospect that you can see why all the code was needed.

Along the way, you've learned all about **SKShapeNode**, **AVAudioPlayer**, **UIBezierPath**, custom enums, default method parameters, and more, so you're several steps closer to your goal of being an experienced Swift developer. Well done!

If you're looking to improve this project some more, and you're able to steer yourself away from the particle editor for a few minutes, why not have a go at removing the magic numbers in the **createEnemy()** method. Instead, define them as constant properties of your class, giving them useful names. You could also try adding a new and fast-moving type of enemy that awards the player bonus points if they hit it.

Project 18
iAd and Debugging

A double bill of learning teaches you how to place adverts and debug your code.

Setting up

With two hard projects under your belt, let's take a breather and try some easier stuff: iAd, Apple's advertising network. How easy is iAd? So easy that I decided this technique project was so short I needed to add something else to make it worth your time, so I added a section debugging with Xcode.

To get started with iAd, create a new Single View Application project in Xcode, named Project18 and targeting any device you like.

iAd in 10 minutes

Can iAd really be taught in ten minutes? Absolutely, and I won't even need to rush. This is because the hard work of iAd is done in two places: Interface Builder and iTunes Connect (iTC). You haven't met iTC yet, but it's the content management system for app developers that lets you create apps, set prices, upload screenshots, and more. iTC is also where you opt in to iAd, and then you get paid through your normal payment method.

So: all the enabling, all the contracts, all the payment and so on are handled through iTunes Connect. The next job is taken care of by Interface Builder: it has an iAd object in its object library, and you can just place your advert where you want it in a view controller, and constraints, and you're almost done.

To make things a little more interesting, I'm going to show you how to create an iAd advert in code. This is actually a fairly common thing to do, because many apps that show adverts also have an in-app purchase to remove those ads. Making your adverts in code means you can manage that process much more efficiently.

First, you need to import the iAd coding framework, so put this at the top, just before the **import UIKit** line:

```
import iAd
```

Now add this property so we can reference the iAd banner view throughout our class:

```
var bannerView: ADBannerView!
```

The actual act of creating an iAd for your screen is a matter of using the **ADBannerView** class. Apple's guidelines for using this class say that you should only have one visible at a time, that you should share the view across view controllers if possible, and that you should hide it if no ads are showing.

We're going to create the banner view and start it as hidden. Then, by setting ourself as the delegate, we'll get told by iAd if a banner loaded or not, at which point we can choose to show the advert.

When you create an iAd banner, you need to specify a banner type: do you want a regular banner or a medium rectangle? The former is short and wide and most commonly seen at the top of web pages; the latter is useful for placing inside content, and is what you would normally see on a web page while you're reading a story.

We're also going to need to use the **translatesAutoresizingMaskIntoConstraints** property of **UIView** subclasses so that we can add Auto Layout constraints ourselves. As long as you followed project 6, none of this code should worry you. Put this into **viewDidLoad()**:

```
bannerView = ADBannerView(adType: .Banner)

bannerView.translatesAutoresizingMaskIntoConstraints = false

bannerView.delegate = self

bannerView.hidden = true

view.addSubview(bannerView)


let viewsDictionary = ["bannerView": bannerView]

view.addConstraints(NSLayoutConstraint.constraintsWithVisualFormat("H
:|[bannerView]|", options: [], metrics: nil, views: viewsDictionary))

view.addConstraints(NSLayoutConstraint.constraintsWithVisualFormat("V
:[bannerView]|", options: [], metrics: nil, views: viewsDictionary))
```

Those constraints will force the iAd banner to fill the full width of the screen and align to its bottom.

We've made our view controller the delegate of the banner view so it will receive its load and fail messages, but you'll get a warning saying that it doesn't currently conform to the **ADBannerViewDelegate** protocol. This is easily fixed by changing your class definition to this:

```
class ViewController: UIViewController, ADBannerViewDelegate {
```

Warning: that's the last time I'll be showing you how to make a class conform to a protocol. In the future I'll tell you to conform to a protocol and you should know how to do it yourself.

The **ADBannerViewDelegate** protocol has no required methods, but does have two optional ones that we care about: **bannerViewDidLoadAd()** tells us when we received an advert, and **bannerView(_,didFailToReceiveAdWithError:)** tells us when something went wrong. The very least we need to do is show our banner when we received an advert, and hide it otherwise:

```
func bannerViewDidLoadAd(banner: ADBannerView!) {

  bannerView.hidden = false

}


func bannerView(banner: ADBannerView!, didFailToReceiveAdWithError
error: NSError!) {

  bannerView.hidden = true

}
```

If you're using a table view, a scroll view, a collection view, a text view or something else that scrolls, you should set its **contentInset** and **scrollIndicatorInset** properties (see project 16) so that it doesn't scroll under the advert when it's visible.

Otherwise, that's it. Yes, that's iAd ready in your app – it will automatically serve correctly sized adverts for each user's device, it will automatically handle users tapping on the advert to show more information, and it will automatically report usage statistics to Apple so you can get paid. The fact that it's so easy is because it's built into the core of iOS!

You can run your app now and seem some sample iAd adverts so you know it's all working. You won't see live adverts until your app is actually live on the App Store, but as long as the sample adverts work you're OK.

There's one more thing I want to discuss, only so that it saves you the hassle of trying to figure it yourself, and that's sharing a banner view across view controllers. You see, if your user is pushing and popping lots of views inside a navigation controller, and each one has an advert inside, you're going to create lots of adverts that appear for only a few seconds, which wastes your advert inventory.

Apple's solution is to create an **ADBannerView** object once then share it across all your views. You should still have only one visible at any given time, but at least this way you don't need to keep creating and destroying adverts.

All the projects we've made (and will make) all include the file AppDelegate.swift, and we looked at it very briefly in project 7. The app delegate is, as you might imagine, the delegate for all our app's notifications, but more importantly it's an object that gets created when the app starts and never destroyed until the app is terminated – it's always there.

The permanent nature of this app delegate means that we can easily use it to store app-wide information. That is, anything we need to reuse in multiple places. In complex apps it's usual to try to isolate functionality in different classes rather than put things into the app delegate, for example you might have a class dedicated to managing iAds. But for now using our app delegate is perfectly fine.

To make this solution work, you need to move the property and the **ADBannerView** creation into your app delegate – inside **application(_,didFinishLaunchingWithOptions:)** is fine. Then you would use the **viewWillAppear()** and **viewWillDisappear()** to hide or show the advert in each view controller.

To get a reference to the banner view belonging to your app delegate, you need to learn some new code. Specifically, you need to use the **UIApplication** class to get a reference to your current application, which is done using the **sharedApplication()** method. You can then read its delegate, but you do need to typecast it as the **AppDelegate** type because that's what contains your banner view property.

So, you might have something like this in **viewDidAppear()**

```
let appDelegate = UIApplication.sharedApplication().delegate as
AppDelegate

view.addSubview(appDelegate.bannerView)
```

It's not hard to do, and really it's something Apple could do automatically, but it *is* required so don't forget it!

Debugging in Xcode

Using iAd is so easy that I figured you'd want more. And you're still reading, so I guess I was right! To give you a little more bang for your buck, I'm going to walk you through some of the ways you can find problems in your code and fix them – a process known as *debugging*. Please create a new Single View Application project named Project18b and targeting iPhone.

We're going to start with the absolute easiest, which is the **print()** function. This prints a message into the Xcode debug console that can say anything you want, because users won't see it in the UI. The "scattershot" approach to bug fixing is to litter your code with calls to **print()** then follow the messages to see what's going on.

You'll meet lots of people telling you how bad this is, but the truth is it's the debugging method everyone starts with – it's easy, it's natural, and it often gives you enough information to solve your problem. Use it with Swift's string interpolation to see the contents of your variables when your app is running.

One level up from **print()** are assertions, which are debug-only checks that will force your app to crash if a specific condition isn't true. When you ship your app to the App Store, Xcode automatically disables your assertions so they have no impact on performance.

Here's a very basic example:

```
assert(1 == 1, "Maths failure!")
assert(1 == 2, "Maths failure!")
```

As you can see **assert()** takes two parameters, and they are something to check, and a message to print out of the check fails. If the check evaluates to false, your app will be forced to crash because you know it's not in a safe state, and you'll see the error message in the debug console.

The advantage to assertions is that their check code is never executed in a live app, so your users are never aware of their presence. This is different from **print()**, which would remain in your code if you shipped it, albeit mostly invisible.

Because calls to **assert()** are ignored in release builds of your app, you can do complex

checks:

```
assert(myReallySlowMethod() == false, "The slow method returned
false, which is a bad thing!")
```

Next up, let's look at breakpoints. You can put this code into your existing Project18 code if you want, into its **viewDidLoad()** method – it would certainly help you follow along if you had an actual project to type things into. These are easy to use initially, but have a lot of hidden complexity if you want to use it.

Let's start small, with a simple loop that prints numbers from 1 to 100:

```
for i in 1 ... 100 {

  print("Got number \(i)")

}
```

If we wanted to see exactly what our program state was at the time we call the **print()** function, look to the left of where you've been typing and you'll see the line number markers. Click on the line number where **print()** is, and a blue marker will appear. Click again, and the blue arrow will become faint to show that the breakpoint exists but is disabled. Click again to make it active, or right-click and choose Delete Breakpoint to remove it entirely.

No line numbers? If your Xcode isn't showing line numbers by default, I suggest you turn them on. Go to the Xcode menu and choose Preferences, then choose the Text Editing tab and make sure "Line numbers" is checked.

With that breakpoint in place, Xcode will pause execution when it's reached and show you the values of all your variables. Try running it now, and you should see your app paused, with a green marker on the line of code that is about to be executed. At the bottom of the Xcode window you should see Xcode telling you that **i** currently has a value of 1. That's because it paused as soon as this line is reached, which is the very first iteration of our loop.

From here, you can carry on execution by pressing F6, but you may need to use Fn+F6 because the function keys are often mapped to actions on Macs. This shortcut is called Step

Over and will tell Xcode to advance code execution by one line. You can walk through the loop in its entirety by pressing F6 again and again, but there's another command called Continue (Ctrl+Cmd+Y) that means "continue executing my program until you hit another breakpoint."

When your program is paused, you'll see something useful on the left of Xcode's window: a *back trace* that shows you all the threads in your program and what they are executing. So if you find a bug somewhere in method **d()**, this back trace will show you that **d()** was called by **c()**, which was called by **b()**, which in turn was called by **a()** – it effectively shows you the events leading up to your problem, which is invaluable when trying to spot bugs.

Breakpoints can do two more clever things, but for some reason both of them aren't used nearly enough. The first is that you can make breakpoints conditional, meaning that they will pause execution of your program only if the condition is matched. Right now, our breakpoint will stop execution every time our loop goes around, but what if we wanted it to stop only every 10 times?

Right-click on the breakpoint (the blue arrow marker) and choose Edit Breakpoint. In the popup that appears, set the condition value to be **i % 10 == 0** – this uses modulo, as seen in project 8. With that in place, execution will now pause only when **i** is 10, 20, 30 and so on, up to 100. You can use conditional breakpoints to execute debugger commands automatically – the "Automatically continue" checkbox is perfect for making your program continue uninterrupted while breakpoints silently trigger actions.

The second clever thing that breakpoints can do is be automatically triggered when an exception is thrown. Exceptions are errors that aren't handled, and will cause your code to crash. With breakpoints, you can say "pause execution as soon as an exception is thrown," so that you can examine your program state and see what the problem is.

To make this happen, press Cmd+7 to choose the breakpoint navigator – it's on the left of your screen, where the project navigator normally sits. Now click the + button in the bottom-left corner and choose "Add Exception Breakpoint." That's it! The next time your code hits a fatal problem, the exception breakpoint will trigger and you can take action.

The last debugging technique I want to look at is view debugging, because this is relatively new so developers aren't really using it much yet. View debugging used to be difficult to do, because you'd have a complicated view controller layout with buttons, labels, views inside views, and so on. If something wasn't showing, it was hard to know *why*.

Xcode 6 and later can help you, because it adds two clever features. The first is called Show View Frames, and it's accessible under the Debug menu by choosing View Debugging > Show View Frames. If you have a complicated view layout (project 8 is a good example!), this option will draw lines around all your views so you can see exactly where they are.

The second feature is extraordinarily cool, but I've only seen it used to good effect a couple of times. It's called Capture View Hierarchy, and when it's used your see your current view layout inside Xcode with thin gray lines around all the views. You might think this is just like Show View Frames, but it's cleverer than that!

If you click and drag inside the hierarchy display, you'll see you're actually viewing a 3D representation of your view, which means you can look behind the layers to see what else is there. The hierarchy automatically puts some depth between each of its views, so they appear to pop off the canvas as you rotate them. This debug mode is perfect for times when you know you've placed your view but for some reason can't see it – often you'll find the view is behind something else by accident.

Wrap up

I couldn't in good conscience show you iAd by itself, because it's just so darn easy to use. But on the flip side, it at least means you got a lightning intro to Xcode debugging. There are lots of options to choose from, and you will, I promise, use all of them at some point. Yes, even **print()**.

There's more to learn about debugging, such as the Step Into and Step Out commands, but realistically you need to start with what you have before you venture any further. I would much rather you mastered three of the debugging tools available to you rather than having a weak grasp of all of them.

Project 19
Capital Cities

Teach users about geography while you learn about MKMapView and annotations.

Setting up

It's time for another app project, and this time you're going to learn about **MapKit** – Apple's mapping framework that lets us drops pins, plan routes, and zoom around the world with just a few swipes.

Working with MapKit requires you to learn quite a few new classes, so I've tried to construct a project simple enough that we can focus on the mapping element. In this project you'll make an app that shows the locations of capital cities around the world, and when one of them is tapped you can bring up more information.

Create a new Single View Application project in Xcode, name it Project19 and set its target to be iPhone. Now go to Interface Builder for your view controller, and embed it inside a navigation controller. Search for "map" in the object library, drop a map view into your view controller so that it occupies the full view, then use Resolve Auto Layout Issues > Add Missing Constraints so that it stays next to each edge.

Now, run your program and… crash? Oh dear. Time for some code!

Up and running with MapKit

Select your project in the project navigator – it's the top thing, with a blue icon next to it. In the center of the Xcode window you'll see lots of options, and this is usually where you limit the orientation of your app. But this time, I want you to go up to the list of tabs and change from General to Capabilities, because we're going to ask that our app be allowed to use maps.

There are lots of capabilities you can request on this screen, but the one we're looking for right now is called simply "Maps". Find it, then change its switch from Off to On. You should be able to try running your app again, because this time will work.

In the picture below you can see the Entitlements tab showing map support being enabled.
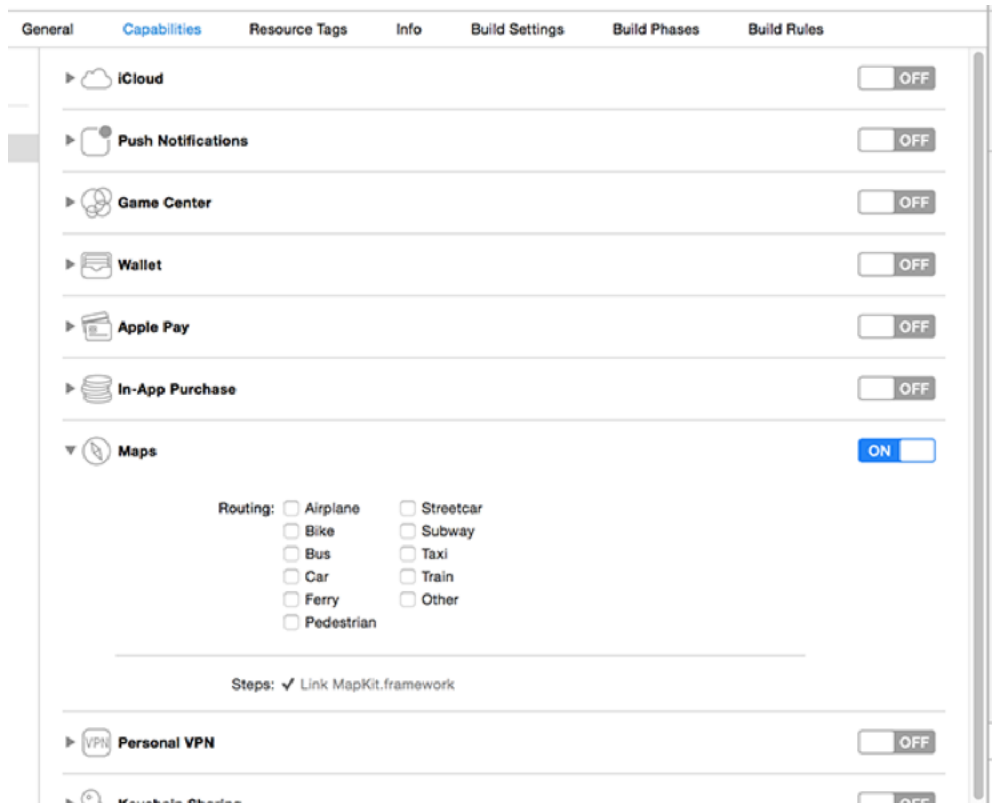
You must use the Entitlements tab to enable support for MapKit.

You'll see a default map view, and you can pan around, zoom in and out, and so on. If you were wondering, you need to hold down Alt to trigger a virtual "pinch" gesture – just click and drag as if you were moving one finger, and the other "finger" will move in the opposite

direction.

Using the assistant editor, please create an outlet for your map view called **mapView**. You should also set your view controller to be the delegate of the map view. As a reminder, you do that by Ctrl-dragging from the map view to the orange and white view controller button just above the layout area. You will also need to add **import MapKit** to ViewController.swift so it understands what **MKMapView** is.

With that done, we're going to add some annotations to our map. Annotations are objects that contain a title, a subtitle and a position. The first two are both strings, the third is a new data type called **CLLocationCoordinate2D**, which is a structure that holds a latitude and longitude for where the annotation should be placed.

Map annotations are described not as a class, but as a protocol. This is something you haven't seen before, because so far protocols have all been about methods. But if we want to conform to the **MKAnnotation** protocol, which is the one we need to adopt in order to create map annotations, it states that we *must* have a coordinate in our annotation. That makes sense, because there's no point in having an annotation on a map if we don't know where it is. The title and subtitle are optional, but we'll provide them anyway.

Create a new file and choose iOS > Source > Cocoa Touch Class. Make it a subclass of **NSObject** and name it Capital. With map annotations, you can't use structs, and you must inherit from **NSObject** because it needs to interactive with Apple's Objective C code.

Change the contents of Capital.swift to this:

```
import MapKit

import UIKit


class Capital: NSObject, MKAnnotation {

  var title: String?

  var coordinate: CLLocationCoordinate2D

  var info: String


  init(title: String, coordinate: CLLocationCoordinate2D, info:
String) {
```

```
        self.title = title

        self.coordinate = coordinate

        self.info = info

    }

}
```

There are our three properties, along with a basic initializer that just copies in the data it's given. Again, we need to use **self.** here because the parameters being passed in are the same name as our properties. I've added **import MapKit** to the file because that's where **MKAnnotation** and **CLLocationCoordinate2D** are defined.

With this custom subclass, we can create capital cities by passing in their name, coordinate and information – I'll be using the **info** property to hold one priceless (read: off-the-cuff, I sucked at geography) informational nugget about each city. You're welcome to do better!

Put these lines into the **viewDidLoad()** method of ViewController.swift:

```
let london = Capital(title: "London", coordinate:
CLLocationCoordinate2D(latitude: 51.507222, longitude: -0.1275),
info: "Home to the 2012 Summer Olympics.")

let oslo = Capital(title: "Oslo", coordinate:
CLLocationCoordinate2D(latitude: 59.95, longitude: 10.75), info:
"Founded over a thousand years ago.")

let paris = Capital(title: "Paris", coordinate:
CLLocationCoordinate2D(latitude: 48.8567, longitude: 2.3508), info:
"Often called the City of Light.")

let rome = Capital(title: "Rome", coordinate:
CLLocationCoordinate2D(latitude: 41.9, longitude: 12.5), info: "Has a
whole country inside it.")

let washington = Capital(title: "Washington DC", coordinate:
CLLocationCoordinate2D(latitude: 38.895111, longitude: -77.036667),
info: "Named after George himself.")
```

These **Capital** objects conform to the **MKAnnotation** protocol, which means we can send it to map view for display using the **addAnnotation()** method. Put this just before the end of

viewDidLoad():

```
mapView.addAnnotation(london)

mapView.addAnnotation(oslo)

mapView.addAnnotation(paris)

mapView.addAnnotation(rome)

mapView.addAnnotation(washington)
```

Alternatively, you can add multiple annotations at once using the **addAnnotations()** method. Using this, you would replace those five lines with this:

```
mapView.addAnnotations([london, oslo, paris, rome, washington])
```

That creates an array out of the annotations and sends it in one lump to the map view.

If you run your program now, you'll see pins on the map for each city, and you can tap any of them to see the city name. But where's the **info** property? To show more information, we need to customise the view used to show the annotations.

Our pins are visible on the map, but they don't do anything more than just show each city's name.

Annotations and accessory views

Every time the map needs to show an annotation, it calls a **viewForAnnotation** method on its delegate. We don't implement that method right now, so the default red pin is used with nothing special – although as you've seen it's smart enough to pull out the title for us.

Customising an annotation view is a little bit like customising a table view cell or collection view cell, because iOS automatically reuses annotation views to make best use of memory. If there isn't one available to reuse, we need to create one from scratch using the **MKPinAnnotationView** class.

Our custom annotation view is going to look a lot like the default view, with the exception that we're going to add a button that users can tap for more information. So, they tap the pin to see the city name, then tap its button to see more information. In our case, it's those fascinating facts I spent literally tens of seconds writing.

There are a couple of things you need to be careful of here. First, **viewForAnnotation** will be called for your annotations, but also Apple's. For example, if you enable tracking of the user's location then that's shown as an annotation and you don't want to try using it as a capital city. If an annotation is not one of yours, just return **nil** from the method to have Apple's default used instead.

Second, adding a button to the view isn't done using the **addTarget()** method you already saw in project 8. Instead, you just add the button and the map view will send a message to its delegate (us!) when it's tapped.

Here's a breakdown of what the method will do:

1. Define a reuse identifier. This is a string that will be used to ensure we reuse annotation views as much as possible.
2. Check whether the annotation we're creating a view for is one of our **Capital** objects.
3. Try to dequeue an annotation view from the map view's pool of unused views.
4. If it isn't able to find a reusable view, create a new one using **MKPinAnnotationView** and sets its **canShowCallout** property to be true. This triggers the popup with the city name.
5. Create a new **UIButton** using the built-in **.DetailDisclosure** type. This is a small blue "i" symbol with a circle around it.
6. If it can reuse a view, update that view to use a different annotation.
7. If the annotation isn't from a capital city, it must return **nil** so iOS uses a default view.

Put this method into your view controller, watching out for my numbered comments:

```
func mapView(mapView: MKMapView!, viewForAnnotation annotation:
MKAnnotation!) -> MKAnnotationView! {
    // 1

    let identifier = "Capital"


    // 2

    if annotation.isKindOfClass(Capital.self) {

        // 3

        var annotationView =
mapView.dequeueReusableAnnotationViewWithIdentifier(identifier)


        if annotationView == nil {

            //4

            annotationView = MKPinAnnotationView(annotation: annotation,
reuseIdentifier: identifier)

            annotationView!.canShowCallout = true


            // 5

            let btn = UIButton(type: .DetailDisclosure)

            annotationView!.rightCalloutAccessoryView = btn

        } else {

            // 6

            annotationView!.annotation = annotation

        }


        return annotationView

    }


    // 7
```

```
      return nil
}
```

You can press Cmd+R to run your app, and now if you tap on any pin you'll see a city's name as well as a button you can tap to show more information. Like I said, you don't need to use **addTarget()** to add an action to the button, because you'll automatically be told by the map view using a **calloutAccessoryControlTapped** method.

When this method is called, you'll be told what map view sent it (we only have one, so that's easy enough), what annotation view the button came from (this is useful), as well as the button that was tapped.

The annotation view contains a property called **annotation**, which will contain our **Capital** object. So, we can pull that out, typecast it as a **Capital**, then show its title and information in any way we want. The easiest for now is just to use a **UIAlertController**, so that's what we'll do.

Add this code to your view controller, just beneath the previous method:

```
func mapView(mapView: MKMapView!, annotationView view:
MKAnnotationView!, calloutAccessoryControlTapped control: UIControl!)
{
  let capital = view.annotation as! Capital

  let placeName = capital.title

  let placeInfo = capital.info


  let ac = UIAlertController(title: placeName, message: placeInfo,
preferredStyle: .Alert)

  ac.addAction(UIAlertAction(title: "OK", style: .Default, handler:
nil))

  presentViewController(ac, animated: true, completion: nil)
}
```

With that, the project is done. We have pins in place, city names being showed when the

pins are tapped, and more information popping up when requested. Perfect!
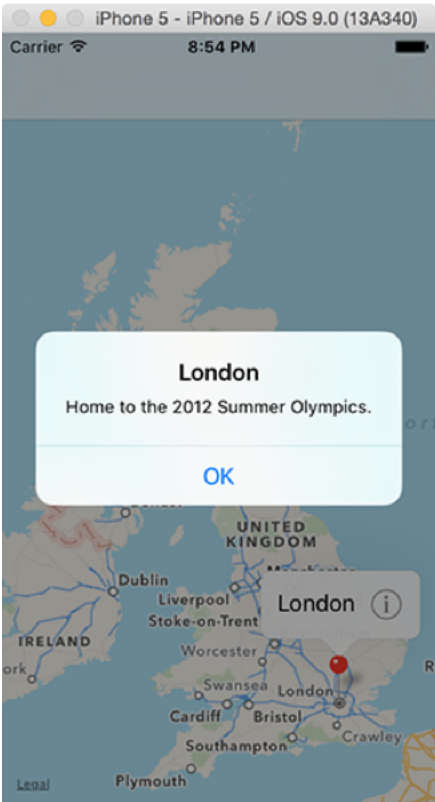
Now that we have customized the map pins, users can tap the 'i' button to see some detail about each city.

Wrap up

I tried to keep this project as simple as possible so that you can focus on the map component, because there's a lot to learn: **MKMapView**, **MKAnnotation**, **MKPinAnnotationView**, **CLLocationCoordinate2D** and so on, and all must be used before you get a finished product.

Again, we've only scratched the surface of what maps can do in iOS, but that just gives you more room to extend the app yourself! Try adding a **UIAlertController** action sheet that lets users specify how they want to view the map. There's a **mapType** property that draws the maps in different ways. For example, **.Satellite** gives a satellite view of the terrain.

If you want to try something harder, you could typecast the return value from **dequeueReusableAnnotationViewWithIdentifier()** so that it's an **MKPinAnnotationView**. This will always be the case, because that's what we're creating. But once you typecast the return value, it means you can change the **pinColor** property to either **.Red**, **.Green** or **.Purple**. With that information, try to add a "Favorite" button to cities: regular cities are red, favorites are green.

Project 20

Fireworks Night

Learn about timers and color blends while making things go bang!

Setting up

In this game project we're going to let users create fireworks displays using their fingers. They'll need to touch fireworks of the same color, then shake their device to make them explode. Shaking an iPad isn't the most pleasant user experience, but I had to find *some* way of teaching you about shake gestures!

On the topic of what you'll learn, you're going to meet **NSTimer**, you're going to use sprite color blending, and you're going to try the **followPath()** SpriteKit action.

Create a new SpriteKit project in Xcode, name it project 20, and set its target to be iPad. Force its orientation to be landscape. Now download the files for this project from **GitHub** and drag the Content folder into your Xcode project.

You should, like always with SpriteKit, delete the existing spaceship image from Images.xcassets and delete the contents of the **didMoveToView()** and **touchesBegan()** methods.

Warning: When working with SpriteKit projects, I strongly recommend you use the iPad 2 simulator rather than iPad Air – you'll get much faster frame rates, making it much more suitable for testing.

Ready... aim... fire!

To get the game up and running quickly, we're going to work on the three methods required to launch some fireworks: **didMoveToView()** (will create a timer that launches fireworks every six seconds), **createFirework()** (will create precisely one firework at a specific X/Y position) and **launchFireworks()** (will call **createFirework()**; designed to create firework spreads.)

First, the easy stuff: we need to add some properties to our class:

 • **gameTimer** will be a new class called **NSTimer**. We'll use this to call the **launchFireworks()** method every six seconds.
 • **fireworks** will be an array of **SKNode** objects. Fireworks, like the bomb fuse in project 17, will have a container node, an image node and a fuse node. This avoids accidental taps triggered by tapping on the fuse of a firework.
 • **leftEdge**, **bottomEdge** and **rightEdge** are used to define where we launch fireworks from. Each of them will be just off screen to one side.
 • **score** will track the player's score. I'm going to give you a **didSet** property observer but leave it blank for you to fill in later – you should know how to show a score label by now!

The only thing that's new in there is the **NSTimer** class, and we'll come to that in a moment. First, add these properties now:

```
var gameTimer: NSTimer!
var fireworks = [SKNode]()


let leftEdge = -22
let bottomEdge = -22
let rightEdge = 1024 + 22


var score: Int = 0 {
  didSet {
    // your code here
  }
}
```

While you're up there, please add **import GameplayKit** above **import UIKit**, because we'll be using that to generate random numbers.

To get the whole thing moving, we need to put in a background picture (the same way we've put in all the background pictures so far) and start up our **NSTimer** object. This is done using the **scheduledTimerWithTimeInterval()** method, which not only creates the timer but also starts it going.

When you create an **NSTimer** you specify five parameters: how many seconds you want the delay to be, what object should be told when the timer fires, what method should be called on that object when the timer fires, any context you want to provide, and whether the time should repeat.

In our case, we're going to have it call **launchFireworks()** every six seconds, with repeating enabled. So, replace your current **didMoveToView()** with this:

```
override func didMoveToView(view: SKView) {

  let background = SKSpriteNode(imageNamed: "background")

  background.position = CGPoint(x: 512, y: 384)

  background.blendMode = .Replace

  background.zPosition = -1

  addChild(background)


  gameTimer = NSTimer.scheduledTimerWithTimeInterval(6, target: self,
selector: "launchFireworks", userInfo: nil, repeats: true)

}
```
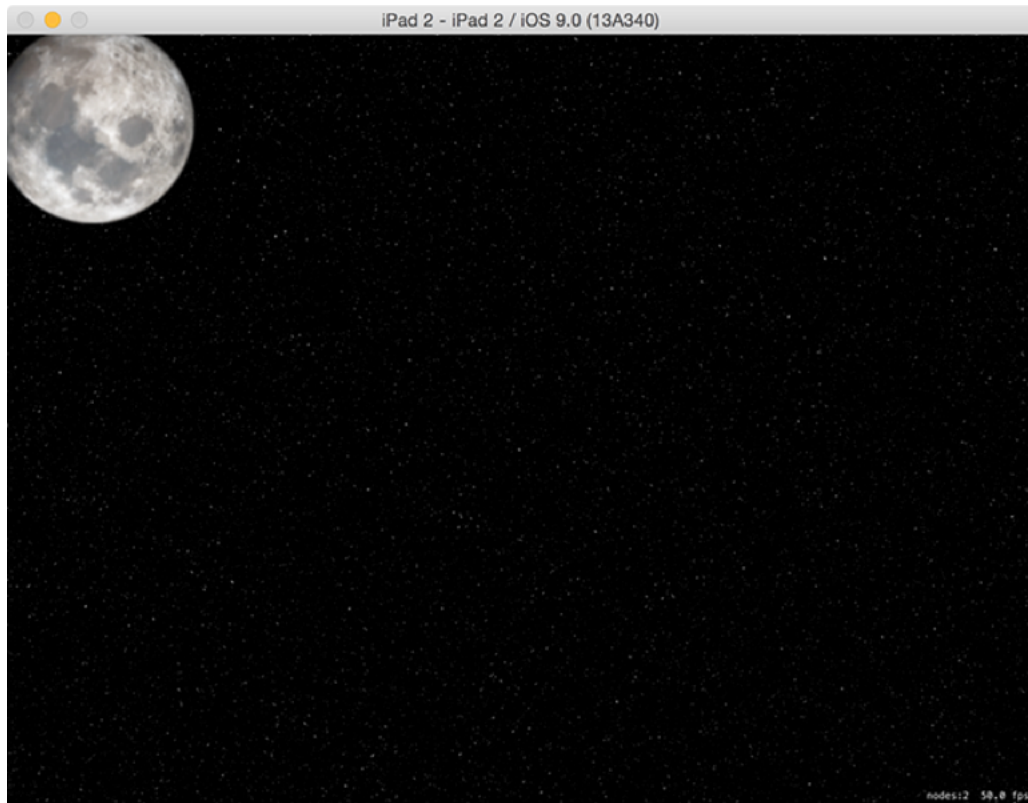
We're using a night sky as the background for this game – it is called Fireworks Night, after all!

That timer will carry on repeating until we tell it to stop, which in this project we won't – that'll be your job! Each time the timer fires, it will call **launchFireworks()**, which itself will call **createFirework()**. Initially we're going to write four types of firework "spreads" (different ways of launching fireworks) but this is something you could easily add more to later.

This system of having one method to create a single enemy and one to create multiple is identical to project 17, so hopefully it makes sense.

First, let's take a look at the **createFirework()** method. This needs to accept three parameters: the X movement speed of the firework, plus X and Y positions for creation. Inside the method there's a lot going on. It needs to:

1. Create an **SKNode** that will act as the firework container, and place it at the X/Y position that was specified.
2. Create a rocket sprite node, give it the name "firework" so we know that it's the important thing, then add it to the container node.
3. Give the firework sprite node one of three random colors: cyan, green or red. I've chosen

cyan because pure blue isn't particularly visible on a starry sky background picture.

4. Create a **UIBezierPath** that will represent the movement of the firework.

5. Tell the container node to follow that path, turning itself as needed.

6. Create particles behind the rocket to make it look like the fireworks are lit.

7. Add the firework to our **fireworks** array and also to the scene.

Here's that, just in Swift:

```swift
func createFirework(xMovement xMovement: CGFloat, x: Int, y: Int) {
  // 1
  let node = SKNode()
  node.position = CGPoint(x: x, y: y)


  // 2
  let firework = SKSpriteNode(imageNamed: "rocket")
  firework.name = "firework"
  node.addChild(firework)


  // 3
  switch GKRandomSource.sharedRandom().nextIntWithUpperBound(3) {
  case 0:
    firework.color = UIColor.cyanColor()
    firework.colorBlendFactor = 1

  case 1:
    firework.color = UIColor.greenColor()
    firework.colorBlendFactor = 1

  case 2:
    firework.color = UIColor.redColor()
    firework.colorBlendFactor = 1
```

```
    default:

      break

  }


  // 4

  let path = UIBezierPath()

  path.moveToPoint(CGPoint(x: 0, y: 0))

  path.addLineToPoint(CGPoint(x: xMovement, y: 1000))


  // 5

  let move = SKAction.followPath(path.CGPath, asOffset: true,
orientToPath: true, speed: 200)

  node.runAction(move)


  // 6

  let emitter = SKEmitterNode(fileNamed: "fuse")!

  emitter.position = CGPoint(x: 0, y: -22)

  node.addChild(emitter)


  // 7

  fireworks.append(node)

  addChild(node)

}
```

Step three is done using two new properties: **color** and **colorBlendFactor**. These two show off a simple but useful feature of SpriteKit, which is its ability to recolor your sprites dynamically with absolutely no performance cost. So, our rocket image is actually white, but by giving it **UIColor.redColor()** (red) with **colorBlendFactor** set to 1 (use the new color exclusively) it will appear red.

Step five is done using a new **SKAction** you haven't seen before: **followPath()**. This takes a CGPath as its first parameter (we'll pull this from the **UIBezierPath**) and makes the node move along that path. It doesn't have to be a straight line like we're using, any bezier path is fine.

The **followPath()** method takes three other parameters, all of which are useful. The first decides whether the path coordinates are absolute or are relative to the node's current position. If you specify **asOffset** as true, it means any coordinates in your path are adjusted to take into account the node's position.

The third parameter to **followPath()** is **orientToPath** and makes a complicated task into an easy one. When it's set to true, the node will automatically rotate itself as it moves on the path so that it's always facing down the path. Perfect for fireworks, and indeed most things! Finally, you can specify a speed to adjust how fast it moves along the path.

Now comes the **launchFireworks()** method, which will lauch fireworks five at a time in four different shapes. As a result this method is quite long because it needs to call **createFirework()** 20 times, but really it's not difficult at all.

The method will generate a random number between 0 and 3 inclusive. If it's zero, we launch the fireworks straight up; if it's one, we fire them in a fan from the center outwards; if it's two we fire them from the left edge to the right; it it's three we fire them from the right edge to the left.

Regardless of the direction of travel, the **createFirework()** call is much the same: how much should the firework move horizontally, and what should its starting X/Y coordinates be. Put this method into your project, then we'll look at it again:

```
func launchFireworks() {
  let movementAmount: CGFloat = 1800

  switch GKRandomSource.sharedRandom().nextIntWithUpperBound(4) {
  case 0:
    // fire five, straight up
    createFirework(xMovement: 0, x: 512, y: bottomEdge)
    createFirework(xMovement: 0, x: 512 - 200, y: bottomEdge)
```

```
createFirework(xMovement: 0, x: 512 - 100, y: bottomEdge)

createFirework(xMovement: 0, x: 512 + 100, y: bottomEdge)

createFirework(xMovement: 0, x: 512 + 200, y: bottomEdge)


case 1:

// fire five, in a fan

createFirework(xMovement: 0, x: 512, y: bottomEdge)

createFirework(xMovement: -200, x: 512 - 200, y: bottomEdge)

createFirework(xMovement: -100, x: 512 - 100, y: bottomEdge)

createFirework(xMovement: 100, x: 512 + 100, y: bottomEdge)

createFirework(xMovement: 200, x: 512 + 200, y: bottomEdge)


case 2:

// fire five, from the left to the right

createFirework(xMovement: movementAmount, x: leftEdge, y:
bottomEdge + 400)

createFirework(xMovement: movementAmount, x: leftEdge, y:
bottomEdge + 300)

createFirework(xMovement: movementAmount, x: leftEdge, y:
bottomEdge + 200)

createFirework(xMovement: movementAmount, x: leftEdge, y:
bottomEdge + 100)

createFirework(xMovement: movementAmount, x: leftEdge, y:
bottomEdge)


case 3:

// fire five, from the right to the left

createFirework(xMovement: -movementAmount, x: rightEdge, y:
bottomEdge + 400)

createFirework(xMovement: -movementAmount, x: rightEdge, y:
bottomEdge + 300)

createFirework(xMovement: -movementAmount, x: rightEdge, y:
bottomEdge + 200)
```

```
    createFirework(xMovement: -movementAmount, x: rightEdge, y:
bottomEdge + 100)

    createFirework(xMovement: -movementAmount, x: rightEdge, y:
bottomEdge)


  default:

    break

  }

}
```

You'll notice I made **movementAmount** into a constant. This is because I was testing various values to find one that worked best, so having it in a constant made it easy to adjust with trial and error.

As you can see in the code, each firework is fired from different X/Y positions so that you get a nice spread on the screen. For example, firing a fan creates one firework on the far left and moving to the left, one in the center left and moving to the left, one in the center moving straight up, and so on.

With that code, you're now able to run the game and see how it works – after a few seconds the first fireworks will start, then they'll continue lauches as the timer continues to fire.
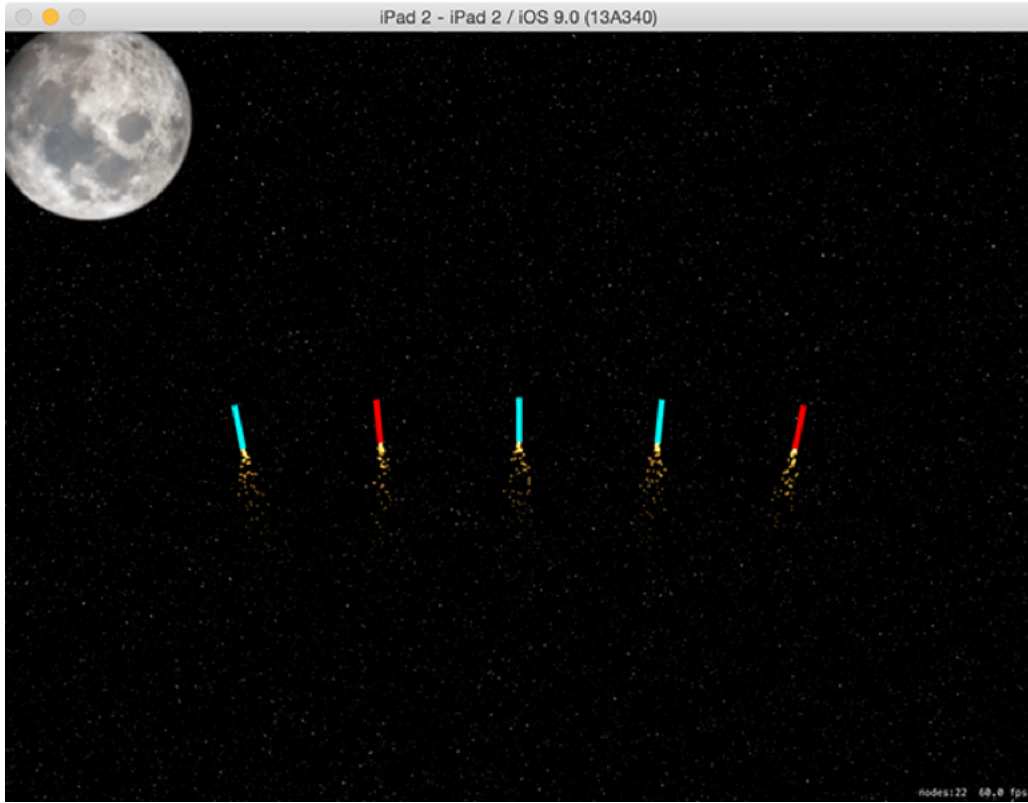
Fireworks can fire out in one of four formations, to add some challenge to the game.

Swipe to select

Now that you can see fireworks shooting across your screen, it's time to reveal the difficulty element. You see, every game needs some challenge, and in our case the challenge is to destroy fireworks in groups of the same color. We're going to make it so that players can select only one color of firework at a time, so if they choose two red then touch a green, the two red will become deselected.

So, the challenge will be to select and detonate fireworks based on their color, and as you'll see shortly we're going to heavily bias scores so that players receive many more points for larger groups.

What we're going to code now is the touch handling method, **checkForTouches()**. We're going to call this from **touchesBegan()** and **touchesMoved()** so that users can either tap to select fireworks or just swipe across the screen.

The method needs to start by figuring out where in the scene the player touches, and what nodes are at that point. It will then loop through all nodes under the point to find any with the name "firework". When it finds one, it will set its name to be "selected" rather than "firework" and change its **colorBlendFactor** value to be white. That will disable the color blending entirely, making the firework white.

Here's the **checkForTouches()** method with that functionality in there:

```
func checkForTouches(touches: Set<UITouch>) {
  guard let touch = touches.first else { return }

  let location = touch.locationInNode(self)
  let nodes = nodesAtPoint(location)

  for node in nodes {
    if node.isKindOfClass(SKSpriteNode.self) {
      let sprite = node as! SKSpriteNode

      if sprite.name == "firework" {
```

```
        sprite.name = "selected"

        sprite.colorBlendFactor = 0

      }

    }

  }

}
```

You've seen most of that previously, but that's because I missed out the logic to handle ensuring that players select only one color at a time. The above code will let them select all the fireworks, regardless of color.

So, we need to insert a second loop just before the **sprite.name = "selected"** line. When you place one loop inside another it's called an inner loop, and you need to be careful: if you have one loop that executes 100 times it's OK, and if you have another loop that executes 200 times that's OK too, but if you put one inside the other you now have 20,000 iterations of your loop and that's almost certainly *not* fine.

Here, though, we'll have maybe two or three items in our outer loop and a maximum of 10 or so in the inner, so we're quite alright.

Remember, this inner loop needs to ensure that the player can select only one firework color at a time. So if they select red then another red, both are selected. But if they then select a green, we need to deselect the first two because they are red.

So, the loop will go through every firework in our **fireworks** array, then find the firework image inside it. Remember, that array holds the container node, and each container node holds the firework image and its spark emitter. If the firework was selected *and* is a different color to the firework that was just tapped, then we'll put its name back to "firework" and put its **colorBlendFactor** back to 1 so it resumes its old color.

So, put this code just before the **sprite.name = "selected"** line:

```
for parent in fireworks {

  let firework = parent.children[0] as! SKSpriteNode
```

```
   if firework.name == "selected" && firework.color != sprite.color {

      firework.name = "firework"

      firework.colorBlendFactor = 1

   }

}
```

That's the entire method, so all we need to do is make sure it's called. To make that happen, we need to modify the existing **touchesBegan()** method and add one for **touchesMoved()** too. All they will do is send the touch information on to **checkForTouches()**, like this:

```
override func touchesBegan(touches: Set<UITouch>, withEvent event:
UIEvent?) {

   super.touchesBegan(touches, withEvent: event)

   checkForTouches(touches)

}


override func touchesMoved(touches: Set<UITouch>, withEvent event:
UIEvent?) {

   super.touchesMoved(touches, withEvent: event)

   checkForTouches(touches)

}
```

There's one more thing we need to code before moving on, and that's some additions to the **update()** method. This is because we need to handle the fireworks that the player *doesn't* destroy, and our solution is simple enough: if they get past 900 points up vertically, we consider them dead and remove them from the **fireworks** array and from the scene.

There is one curious quirk here, and it's down to how you remove items from an array. When removing items, we're going to loop through the array backwards rather than forwards. The reason for is that array items move down when you remove an item, so if you have 1, 2, 3, 4 and remove 3 then 4 moves down to become 3. If you're counting forwards, this is a problem because you just checked three and want to move on, but there's now a new 3 and possibly

no longer a 4! If you're counting backwards, you just move on to 2.

Note: I chose 900 rather than 800 to mean "off screen vertically" because it's nice to give players a little extra time when making important actions. It's possible that the top firework is at 890 and the bottom one still on screen and being manipulated, so at least this way the player has the best possible window in which to make all their selections.

Here's the new **update()** method:

```
override func update(currentTime: NSTimeInterval) {

  for (index, firework) in fireworks.enumerate().reverse() {

    if firework.position.y > 900 {

      // this uses a position high above so that rockets can explode
off screen

      fireworks.removeAtIndex(index)

      firework.removeFromParent()

    }

  }

}
```

Making things go bang

This is easily the best bit of the game, mostly because it involves even more particle systems. There are three things we need to create: a method to explode a single firework, a method to explode all the fireworks (which will call the single firework explosion method), and some code to detect and respond the device being shaken.

First, the code to explode a single firework. Put this somewhere in your game scene:

```
func explodeFirework(firework: SKNode) {

  let emitter = SKEmitterNode(fileNamed: "explode")!

  emitter.position = firework.position

  addChild(emitter)


  firework.removeFromParent()
}
```

You should be able to read that once and know exactly what it does: it creates an explosion where the firework was, then removes the firework from the game scene.

The **explodeFireworks()** method is next, and is only fractionally more complicated. It will be triggered when the user wants to set off their selected fireworks, so it needs to loop through the **fireworks** array (backwards again!), pick out any selected ones, then call **explodeFirework()** on it.

As I said earlier, the player's score needs to go up by more when they select more fireworks, so about half of the **explodeFireworks()** method is taken up with figuring out what score to give the player.

There's one small piece of extra complexity: remember, the **fireworks** array stores the firework container node, so we need to read the firework image out of its **children** array.

Enough talk – here's the code:

```swift
func explodeFireworks() {
    var numExploded = 0


    for (index, fireworkContainer) in fireworks.enumerate().reverse() {
        let firework = fireworkContainer.children[0] as! SKSpriteNode


        if firework.name == "selected" {
            // destroy this firework!
            explodeFirework(fireworkContainer)
            fireworks.removeAtIndex(index)


            numExploded += 1
        }
    }


    switch numExploded {
    case 0:
        // nothing - rubbish!
        break
    case 1:
        score += 200
    case 2:
        score += 500
    case 3:
        score += 1500
    case 4:
        score += 2500
    default:
        score += 4000
    }
```

```
}
```

As you can see, exploding five fireworks is worth 20x more points than exploding just one, hence the incentive to select groups by color!

There's one last thing to do before this game is complete, and that's to detect the device being shaken. This is easy enough to do because iOS will automatically call a method called **motionBegan()** on our game when the device is shaken. Well, it's a little more complicated than that – what actually happens is that the method gets called in GameViewController.swift, which is the **UIViewController** that hosts our SpriteKit game scene.

The default view controller doesn't know that it has a SpriteKit view, and certainly doesn't know what scene is showing, so we need to do a little typecasting. Once we have a reference to our actual game scene, we can call **explodeFireworks()**. Put this method just after the **prefersStatusBarHidden()** method in GameViewController.swift:

```
override func motionBegan(motion: UIEventSubtype, withEvent event:
UIEvent?) {
  let skView = view as! SKView
  let gameScene = skView.scene as! GameScene
  gameScene.explodeFireworks()
}
```

That's it, your game is done. Obviously you can't shake your laptop to make the iOS Simulator respond, but you can use the keyboard shortcut Ctrl+Cmd+Z to get the same result. If you're testing on your iPad, make sure you give it a good shake in order to trigger the explosions!

Wrap up

I've enjoyed making this project, so I hope you enjoyed following along. Plus you have yet more Swift coding experience under your belt, now complete with **NSTimer**, **followPath()**, color blending and, yes, even the shake gesture – although I wouldn't be surprised if you switch to having a button on the screen to make explosions easier!

There's a lot more you can do with this foundation behind you. How about adding a score label? Or perhaps adding different fireworks spread types, for example one where fireworks launch from the left and right? Or if you fancy a bigger challenge, how about making the game end after a certain number of launches? You will need to use the **invalidate()** method of **NSTimer** to stop it from repeating.

Project 21

Local Notifications

Send reminders, prompts and alerts even when your app isn't running.

Setting up

This is going to be the easiest technique project in the entire series, and I expect you're extremely relieved to hear that because it can be hard going always having to learn new things!

What you're going to learn about are local notifications, which let you send reminders to your user's lock screen to show them information when your app isn't running. If you set a reminder in your calendar, making it pop up on your lock screen at the right time is a local notification.

These aren't the same as push notifications, and in fact they are quite a different beast from a development perspective. I would love to cover push notifications here, but they require a dedicated server (or service, if you outsource) to send from and that's outside the remit of this course.

To get started, create a new Single View Application project in Xcode, name it Project21, and set it to target any device.

Scheduling notifications

Open Main.storyboard in Interface Builder and place two buttons, one above the other. The first should have the title "Register Local" and the second the title "Schedule Local". Add whatever constraints you think sensible, but ideally make them centered horizontally so they fit any device. Using the assistant editor, create an action for each: **registerLocal()** and **scheduleLocal()**. Now go back to the standard editor and switch to ViewController.swift.

Let me explain how this project needs to work. First, you can't post messages to the user's lock screen unless you have their permission. This was changed in iOS 8, but it's quite sensible – it would, after all, be awfully annoying if any app could bother you when it pleased.

So, in order to send local notifications in our app, we first need to request permission, and that's what we'll put in the **registerLocal()** method. You register your settings based on what you actually need, and that's done with a class called **UIUserNotificationSettings**. For this example we're going to request an alert (a message to show), along with a badge (for our icon) and a sound (because users just *love* those.)

Once you've created your notification settings object, it's just a matter of calling the **registerUserNotificationSettings()** method to tell iOS what you want, and it will then prompt the user if needed. If you requested access before and were denied, nothing will be shown.

Change your **registerLocal()** method to be this:

```
@IBAction func registerLocal(sender: AnyObject) {

  let notificationSettings = UIUserNotificationSettings(forTypes:
[.Alert, .Badge, .Sound], categories: nil)


UIApplication.sharedApplication().registerUserNotificationSettings(no
tificationSettings)

}
```

Helpful tip: if you want to test allowing or denying permission, just reset the simulator and run the app again to get a clean slate. Choose the iOS Simulator menu then "Reset Content and Settings" to make this happen.
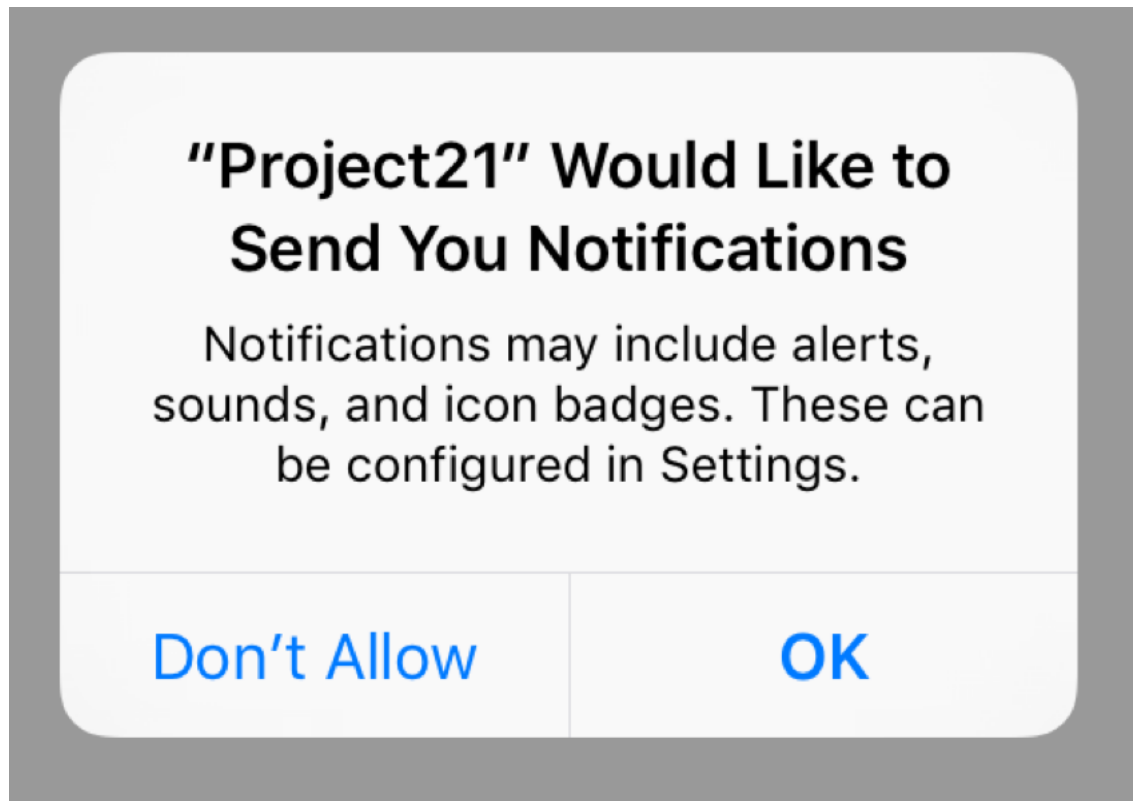
When you request permission to show notifications, iOS shows an alert like this one.

Once we have user permission, it's time to fill in the **scheduleLocal()** method. This will use the **UILocalNotification** class to configure all the data needed to schedule a notification, then call **scheduleLocalNotification()** to schedule it for delivery to the user.

We're going to use the following properties:

 • **fireDate** decides when the notification should be shown. iOS tracks this for us, so our app doesn't need to be running when it's time for the notification to be delivered.
 • **alertBody** is a string containing the text to show to users. The title of the message will automatically be your app's name.
 • **alertAction** is a string shown under your message that completes the sentence, "Slide to…" For example, if you set it be "pericombobulate", it would read "Slide to pericombobulate."
 • **soundName** we'll be using the default alert sound, but it's not hard to specify your own – just make sure you include it in the project!
 • **userInfo** is a dictionary of keys and values that you can provide. The system does nothing with these other than hand them back to you when the app launches so you can respond.

Here's the first draft of the **scheduleLocal()** method:

```
@IBAction func scheduleLocal(sender: AnyObject) {

  let notification = UILocalNotification()

  notification.fireDate = NSDate(timeIntervalSinceNow: 5)

  notification.alertBody = "Hey you! Yeah you! Swipe to unlock!"

  notification.alertAction = "be awesome!"

  notification.soundName = UILocalNotificationDefaultSoundName

  notification.userInfo = ["CustomField1": "w00t"]

UIApplication.sharedApplication().scheduleLocalNotification(notificat
ion)

}
```

I'm providing a single key/value pair to the **userInfo** property; we'll come onto that soon.

That code will trigger a notification five seconds after you click "Schedule Local", so:

• Click "Register Local" and agree to let the app show notifications.
• Click "Schedule Local".
• Before the five seconds are up, press Cmd+L to lock the iOS Simulator screen.
• Wait.

You should see the message appear after a few seconds. Note that the **alertAction** text can be quite faint.
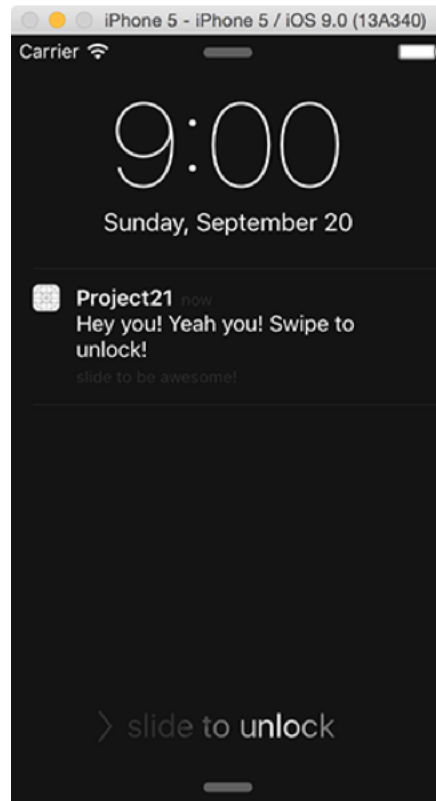
Notifications appear directly on the iOS lock screen, so try not to annoy your users!

Before I move on, our **scheduleLocal()** method has a bug: what if the user doesn't grant us permission, and we try showing a local notification? Well, nothing will happen. That's good because your app didn't crash, but it's bad because users will think your app is broken.

To fix the bug, we need to modify **scheduleLocal()** so that it checks if we have permission to show local notifications before proceeding. This is as easy as querying the return value of **currentUserNotificationSettings()** for our application, and if it's **.None** then we need to alert the user and exit the method.

Put this code at the top of the **scheduleLocal()** notification:

```
guard let settings =
UIApplication.sharedApplication().currentUserNotificationSettings()
else { return }


if settings.types == .None {
```

```
    let ac = UIAlertController(title: "Can't schedule", message:
"Either we don't have permission to schedule notifications, or we
haven't asked yet.", preferredStyle: .Alert)

    ac.addAction(UIAlertAction(title: "OK", style: .Default, handler:
nil))

    presentViewController(ac, animated: true, completion: nil)

    return

}
```

Acting on responses

There's one more thing to learn before we're done with notifications, and that's what happens to your application when it's given a notification to process. So far we've just been scheduling them, but if the user swipes on a notification to unlock their device, your app is launched and given the notification to process. Also, what happens if the user is actually inside your app while one of your notifications fires?

These are two separate cases under the hood, and both need to be addressed independently. In your AppDelegate.swift file we've already briefly looked at the **didFinishLaunchingWithOptions** method, but we didn't look at what the "options" might be. Well, if your app was lauched from a notification, this is how iOS will tell you, so let's take a closer look now.

First, this is how **launchOptions** is defined:

```
[NSObject: AnyObject]?
```

Translated, it's an optional dictionary where an NSObject (or subclass) will be the key and any object can be the value.

In order to read information about the notification, we need first to unwrap this optional so that we have a real dictionary on our ends. Then we need to look up the (*deep breath*) **UIApplicationLaunchOptionsLocalNotificationKey** key. Yes, that's absurdly long. If that exists, it will be a **UILocalNotification** object, so we need to conditionally typecast it using **as?**.

Once we have the **UILocalNotification** object, we need to see if it has a **userInfo** value, which means more optional unwrapping. And finally, if we've made it this far, we will have the same data we set for **userInfo** back when scheduling the notification, so you can do with it as you please.

Modify your **didFinishLaunchingWithOptions** method to this:

```
func application(application: UIApplication,
didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?)
```

```
-> Bool {

  if let options = launchOptions {

    if let notification =
options[UIApplicationLaunchOptionsLocalNotificationKey] as?
UILocalNotification {

      if let userInfo = notification.userInfo {

        let customField1 = userInfo["CustomField1"] as! String

        // do something neat here

      }

    }

  }


  return true

}
```

I called my data **CustomField1** but you can put whatever you want in there.

The other situation that might occur is if your app's notification fires while the app is still running. In this situation, it's obviously too late to call **didFinishLaunchingWithOptions**, so instead your app delegate will have its **didReceiveLocalNotification** method called. This doesn't exist by default, but it's easy enough to create.

As this method can only be called when a notification has definitely been received, you don't need to unwrap any launch options or look for any absurdly long dictionary keys. Instead, just check for the presence of **userInfo** and go from there. Add this method to your application delegate:

```
func application(application: UIApplication,
didReceiveLocalNotification notification: UILocalNotification) {

  if let userInfo = notification.userInfo {

    let customField1 = userInfo["CustomField1"] as! String

    print("didReceiveLocalNotification: \(customField1)")

  }
```

```
}
```

And that's it – you're able to go ahead and use local notifications all you want in your apps. But remember, if you abuse the trust your user has placed in you they will undoubtedly delete your app!

Wrap up

That was easy, right? And yet it's such a great feature to have, because now your app can talk to users even when it isn't running. You want to show a step count for how far they've walked? Local notification. You want to trigger an alert because it's their turn to play in a game? Local notification. You want to send them marketing messages to make them buy more stuff? Actually, just don't do that, you bad person.

If you're curious about push notifications then check out something like **pushwizard.com**, because you need a server somewhere that can store which devices to send to and handle delivery of the messages.

Project 22

Detect-a-Beacon

Learn to find and range iBeacons using our first project for a physical device.

Setting up

Apple introduced iBeacon technology with iOS 7, and it's helped make the Internet of Things hypefest even more stratospheric. In this project you're going to learn to detect and range beacons, which in turn means learning how to ask your user for their location. With this, you'll have all the tools required to make your own location-aware apps – just scatter a few beacons around your house!

If you don't have any iBeacons at home, that's OK because most people don't. Instead, I recommend you install the app "Locate Beacon" on your iPad or iPhone, because that comes with an iBeacon transmitter built in, making it perfect for testing. You also need an iOS device that's compatible with iBeacons, which means iPhone 4s or later, 3rd generation iPad or later, iPad Mini or later, or 5th generation iPod Touch or later. I'm afraid the iOS Simulator won't work, but you can at least follow along with the code. Please ensure you have Bluetooth enabled on your device.

If you've never pushed an app to a real device before, you need to make sure you select the device from the list of destinations. You can do this by clicking where it says "Project22" to the right of the play and stop buttons, or by going to the Product menu and choosing Destination then selecting your device. If it comes up with "ineligible" it means your device is running an older version of iOS than your project is designed for, so you may need to go to your project settings (where you configure orientation) and change Deployment Target to match.

Create a new Single View Application project in Xcode, name it Project22 and set its target to be whichever of the above devices you own.

Requesting location

It should come as no surprise that Apple considers a user's location to be private, and that means we need to ask for permission to use it. From iOS 8 onwards, how you ask for permission depends on what you're trying to do: would you like the user's location only when your app is running, or would you like a user's location even when your app isn't running?

You might think that you'd only ever want location access when your app is running. After all, what's the point in asking for information when your app isn't around to use it?

There are times you'll want both. For example, if you're creating a map app that shows users how to get from their current location to your nearest store, you'll only need their location when the app is being used. But if you're creating an app that needs to be woken up when the user reaches a location, then you'll need access even when the app isn't running – iOS monitors the user's location on your behalf and automatically starts your app as needed.

Requesting location access requires a change to your apps Info.plist file, which is the property list file we met in project 16. Depending on whether you want "always" access or just "when in use" you need to set one key or the other. Select your property list, then go to the Editor menu and choose Add Item. Now change the name of your new item to either:

 • **NSLocationAlwaysUsageDescription** if you want to have the user's location even when the app isn't running.
 • **NSLocationWhenInUseUsageDescription** if you only want the user's location when the app is running.

You should make sure the type is set to String, then in the value field enter some text to explain to users *why* you want their location. For example, "We want to help you find your nearest store." When your user is prompted to grant location access, this text will be shown alongside Apple's own descriptive message.
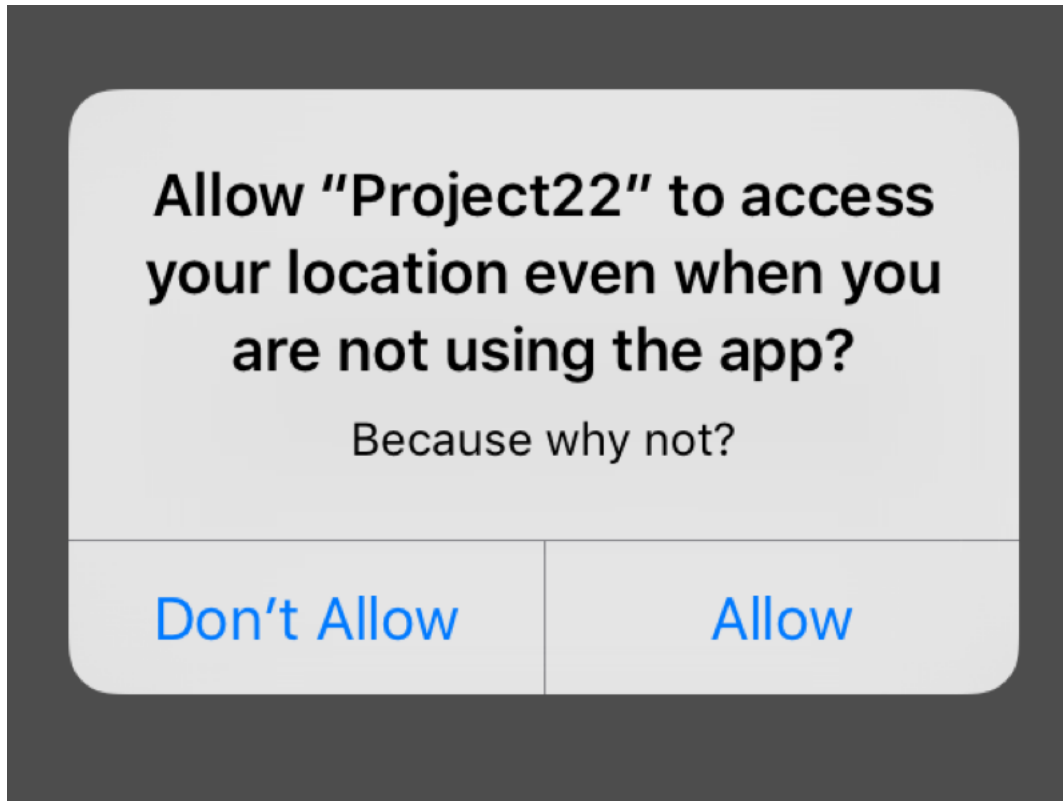
When iOS requests location access users are likely to be suspicious, so make sure you explain why you want it.

That's enough knowledge to get this app jump started, so open up Main.storyboard and place a label in there. Give it the custom font Helvetica Neue Thin size 40, then give the text "UNKNOWN". For constraints, please center it horizontally and vertically. Now create an outlet for it using the assistant editor, and name the outlet **distanceReading**.

That label will show one of four messages depending on how close we are to our test beacon, which of course might be an iPad acting as a beacon if you don't own actual hardware. Because iBeacons use very low energy levels, their range is limited and also easily interrupted; even something as simple as turning your back to the beacon weakens its signal dramatically. Based on the beacon's distance to us, we'll show either "UNKNOWN", "FAR", "NEAR" or "RIGHT HERE".

Apple restricts your ranging to these values because of the signal's low energy, but it's more than enough for most uses.

Our user interface is just one label: how close is the user to the beacon?

To complete our current step, let's make sure we have location configured correctly. This bit will work fine on the simulator, because although the simulator isn't capable of detecting iBeacons it can simulate general location information well enough.

Open up ViewController.swift and add this property to your class:

```
var locationManager: CLLocationManager!
```

This is the Core Location class that lets us configure how we want to be notified about location, and will also deliver location updates to us. As soon as you add that property you'll get errors in Xcode because it doesn't know what **CLLocationManager** is, so add this import to the top of your file:

```
import CoreLocation
```

That doesn't actually create a location manager, or even prompt the user for location permission! To do that, we first need to create the object (easy), then set ourselves as its delegate (easy, but we need to conform to the protocol), then finally we need to request authorization. We'll start by conforming to the protocol, so change your class definition to this:

```
class ViewController: UIViewController, CLLocationManagerDelegate {
```

Now modify your **viewDidLoad()** method to this:

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

  locationManager = CLLocationManager()
  locationManager.delegate = self
  locationManager.requestAlwaysAuthorization()

  view.backgroundColor = UIColor.grayColor()
}
```

Creating the object and setting the delegate are easy enough, but the **requestAlwaysAuthorization()** call is new. This is where the actual action happens: if you have already been granted location permission then things will Just Work; if you haven't, iOS will request it now. Note: if you used the "when in use" key, you should call **requestWhenInUseAuthorization()** instead.

Warning: If you did not set the correct plist key earlier, your request for location access will be ignored.

I slipped one other thing in there: I set the view's background color to be gray. As well as

changing the label's text, we'll be using color to tell users how distant the beacon is.

Requesting location authorization is a non-blocking call, which means your code will carry on executing while the user reads your location message and decides whether to grant you access to their location.

When the user has finally made their mind, you'll get told their result because we set ourselves as the delegate for our **CLLocationManager** object. The method that will be called is this one:

```swift
func locationManager(manager: CLLocationManager,
didChangeAuthorizationStatus status: CLAuthorizationStatus) {

   if status == .AuthorizedAlways {

     if
CLLocationManager.isMonitoringAvailableForClass(CLBeaconRegion.self)
{

       if CLLocationManager.isRangingAvailable() {

         // do stuff

       }

     }

   }

}
```

Put that into your view controller class somewhere, then run your app. It's important to test it before continuing, because if you've made a mistake somewhere it's hard to know unless you stop and check. The most common error is misconfiguring the plist with location privacy settings, so if you don't see a message requesting location access then check there first.

The **didChangeAuthorizationStatus** method we just added doesn't do anything because it just has a comment saying **// do stuff**. We'll fill that in with great stuff shortly, but for now look at the conditional statements wrapped around it: did we get authorized by the user? If so, is our device able to monitor iBeacons? If so, is ranging available? (Ranging is the ability to tell roughly how far something else is away from our device.)

Hunting the beacon

If everything is working, you should have received a large iOS confirmation prompt asking whether you grant the user access to their location. This message is really blunt, so users hopefully take a few moments to read it before continuing.

But that prompt is not the only way iOS helps users guard their privacy. If you went for "when in use", you'll still get location information while your app is in the background if you enable the background capability, and iOS will notify users that this is happening by making the device status bar blue and saying "YourAppName is using your location." If you went for "always", iOS will wait a few days then ask the user if they still want to grant permission, just to be fully sure.

Assuming everything went well, let's take a look at how we actually range beacons. First, we use a new class called **CLBeaconRegion**, which is used to identify a beacon uniquely. Second, we give that to our **CLLocationManager** object by calling its **startMonitoringForRegion()** and **startRangingBeaconsInRegion()** methods. Once that's done, we sit and wait. As soon as iOS has anything tell us, it will do so.

iBeacons are identified using three pieces of information: a universally unique identifier (UUID), plus a major number and a minor number. The first number is a long hexadecimal string that you can create by running the **uuidgen** in your Mac's terminal. It should identify you or your store chain uniquely.

The major number is used to subdivide within the UUID. So, if you have 10,000 stores in your supermarket chain, you would use the same UUID for them all but give each one a different major number. That major number must be between 1 and 65535, which is enough to identify every McDonalds and Starbucks outlet combined!

The minor number can (if you wish) be used to subdivide within the major number. For example, if your flagship London store has 12 floors each of which has 10 departments, you would assign each of them a different minor number.

The combination of all three identify the user's precise location:

- **UUID:** You're in a Acme Hardware Supplies store.
- **Major:** You're in the Glasgow branch.
- **Minor:** You're in the shoe department on the third floor.

If you don't need that level of detail you can skip minor or even major – it's down to you.

It's time to put this into code, so we're going to create a new method called **startScanning()** that contains the following:

```
func startScanning() {
   let uuid = NSUUID(UUIDString: "5A4BCFCE-174E-4BAC-
A814-092E77F6B7E5")!

   let beaconRegion = CLBeaconRegion(proximityUUID: uuid, major: 123,
minor: 456, identifier: "MyBeacon")

   locationManager.startMonitoringForRegion(beaconRegion)
   locationManager.startRangingBeaconsInRegion(beaconRegion)
}
```

You met **NSUUID** in project 10, but here we're converting a string into a UUID rather than generating a UUID and converting it to a string. The UUID I'm using there is one of the ones that comes built into the Locate Beacon app – look under "Apple AirLocate 5A4BCFCE" and find it there. Note that I'm scanning for specific major and minor numbers, so please enter those into your Locate Beacon app.

The **identifier** field is just a string you can set to help identify this beacon in a human-readable way. That, plus the UUID, major and minor fields, goes into the **CLBeaconRegion** class, which is used to identify and work with iBeacons. It then gets sent to our location manager, asking it to monitor for the existence of the region and also to start measuring the distance between us and the beacon.

Find the **// do stuff** comment inside the **didChangeAuthorizationStatus** method you wrote a few minutes ago, and change it to this:

```
startScanning()
```

That method should now be much clearer: we only start scanning for beacons when we have

permission and if the device is able to do so.

If you run the app now (on a real device, remember!) you'll see that it literally looks identical, as if we needn't have bothered writing any iBeacon code. But behind the scenes, detection and ranging *is* happening, we're just not doing anything with it!

This app is going to change the label text and view background color to reflect proximity to the beacon we're scanning for. This will be done in a single method, called **updateDistance()**, which will use a switch/case block and animations in order to make the transition look smooth. Let's write that method first:

```
func updateDistance(distance: CLProximity) {
  UIView.animateWithDuration(0.8) { [unowned self] in
    switch distance {
    case .Unknown:
      self.view.backgroundColor = UIColor.grayColor()
      self.distanceReading.text = "UNKNOWN"

    case .Far:
      self.view.backgroundColor = UIColor.blueColor()
      self.distanceReading.text = "FAR"

    case .Near:
      self.view.backgroundColor = UIColor.orangeColor()
      self.distanceReading.text = "NEAR"

    case .Immediate:
      self.view.backgroundColor = UIColor.redColor()
      self.distanceReading.text = "RIGHT HERE"
    }
  }
}
```

Most of that is just choosing the right color and text, but you'll notice the method accepts a **CLProximity** as its parameter. This can only be be one of our four distance values, which is why we don't need a **default** case in there – Swift can see the switch/case is complete.

With that method written, all that remains before our project is complete is to catch the ranging method from **CLLocationManager**. We'll be given the array of beacons it found for a given region, which allows for cases where there are multiple beacons transmitting the same UUID.

If we receive any beacons from this method, we'll pull out the first one and use its **proximity** property to call our **updateDistance()** method and redraw the user interface. If there aren't any beacons, we'll just use **.Unknown**, which will switch the text back to "UNKNOWN" and make the background color gray.

Here's the code:

```
func locationManager(manager: CLLocationManager, didRangeBeacons
beacons: [CLBeacon], inRegion region: CLBeaconRegion) {

  if beacons.count > 0 {

    let beacon = beacons[0]

    updateDistance(beacon.proximity)

  } else {

    updateDistance(.Unknown)

  }

}
```

With that, your code is done. Run it on a device, make sure Locate Beacon is up and transmitting, and enjoy your location-aware app!

Wrap up

Working with iBeacon locations is different from working with maps. The technology is often called *micro-location* because it can tell the difference between a few centimeters and a meter or more. Plus it works inside, which is somewhere GPS continues to be poor, and understandably.

What you've produced is designed to do ranging, but you could easily make it ignore the range data and just focus on whether a beacon is present. For example, if the beacon in your house is present (regardless of range), you could make your app show home-related tasks that you have pre-configured.

Project 23

Space Race

Dodge space debris while you learn about per-pixel collision detection.

Setting up

In this game project we'll seek to answer the question, "how fast can you make a fun game in SpriteKit?" Spoiler warning: the answer is very fast. And that's even when you ignore learning about advancing particle systems, linear and angular damping, and per-pixel collision detection.

The game we're going to produce is a very simple survival game: our player will have to pilot a spaceship safely through a field of space junk. The longer they stay alive the higher their score will be, but they need to keep moving otherwise certain death awaits!

Remarkably, we're going to make this project in just over 100 lines of code. To begin, create a new SpriteKit project in Xcode, name it Project23 and set its target to be iPad. Configure it work only in landscape, then download the files for this project and copy the Content folder into your project. Now delete the spaceship from Images.xcassets and all the code from inside **didMoveToView()** and **touchesBegan()**.

All done? Start the clock – let's see how long it takes to make this game!

Warning: When working with SpriteKit projects, I strongly recommend you use the iPad 2 simulator rather than iPad Air – you'll get much faster frame rates, making it much more suitable for testing.

Space: the final frontier

To begin with we're going to place a handful of things that are required to make our game work: a starfield (not a static background picture this time), the player image, plus a score label. Those three things will use an **SKEmitterNode**, an **SKSpriteNode** and an **SKLabelNode** respectively, so let's declare them as properties now:

```
var starfield: SKEmitterNode!
var player: SKSpriteNode!


var scoreLabel: SKLabelNode!
var score: Int = 0 {
  didSet {
    scoreLabel.text = "Score: \(score)"
  }
}
```

As per usual, we're using a property observer to update the score label as needed.

In order to get those properties set up with meaningful values, we're going to put a lot of code into **didMoveToView()** so that everything is created and positioned up front.

I'm not going to bore you by going through every line of code – three quarters of it you should know by heart at this point! – but I do want to point out a few interesting things.

First, the starfield particle emitter is positioned at X:1024 Y:384, which is the right edge of the screen and half way up. If you created particles like this normally it would look strange, because most of the screen wouldn't start with particles and they would just stream in from the right. But by using the **advanceSimulationTime()** method of the emitter we're going to ask SpriteKit to simulate 10 seconds passing in the emitter, thus updating all the particles as if they were created 10 seconds ago. This will have the effect of filling our screen with star particles.

Second, because the spaceship is an irregular shape and the objects in space are also

irregular, we're going to use per-pixel collision detection. This means collisions happen not based on rectangles and circles but based on actual pixels from one object touching actual pixels in another.

Now, SpriteKit does a really great job of optimizing this so that it looks like it's using actual pixels when in fact it just uses a very close approximation, but you should still only use it when it's needed. If something can be created as a rectangle or a circle you should do so because it's much faster.

Third, we're going to set the contact test bit mask for our player to be 1. This will match the category bit mask we will set for space debris later on, and it means that we'll be notified when the player collides with debris.

Fourth, I'm going to set the gravity of our physics world to be empty, because this is space and there isn't any gravity. Well, that's not strictly true because there is a small amount of gravity everywhere in space, but certainly nothing we can simulate effectively in this game!

Here's the new **didMoveToView()** method:

```
override func didMoveToView(view: SKView) {
  backgroundColor = UIColor.blackColor()


  starfield = SKEmitterNode(fileNamed: "Starfield")!
  starfield.position = CGPoint(x: 1024, y: 384)
  starfield.advanceSimulationTime(10)
  addChild(starfield)
  starfield.zPosition = -1


  player = SKSpriteNode(imageNamed: "player")
  player.position = CGPoint(x: 100, y: 384)
  player.physicsBody = SKPhysicsBody(texture: player.texture!, size:
player.size)
  player.physicsBody!.contactTestBitMask = 1
  addChild(player)
```

```
    scoreLabel = SKLabelNode(fontNamed: "Chalkduster")

    scoreLabel.position = CGPoint(x: 16, y: 16)

    scoreLabel.horizontalAlignmentMode = .Left

    addChild(scoreLabel)


    score = 0


    physicsWorld.gravity = CGVector(dx: 0, dy: 0)

    physicsWorld.contactDelegate = self
}
```

Did you see how easy it is to make per-pixel collision detection work? You just need to create the **SKPhysicsBody** by passing in a texture and size, and for us we just want to use the player's current texture and size. That's it!

The last line of code in that method sets our current game scene to be the contact delegate of the physics world, so you'll need to conform to the **SKPhysicsContactDelegate** protocol.
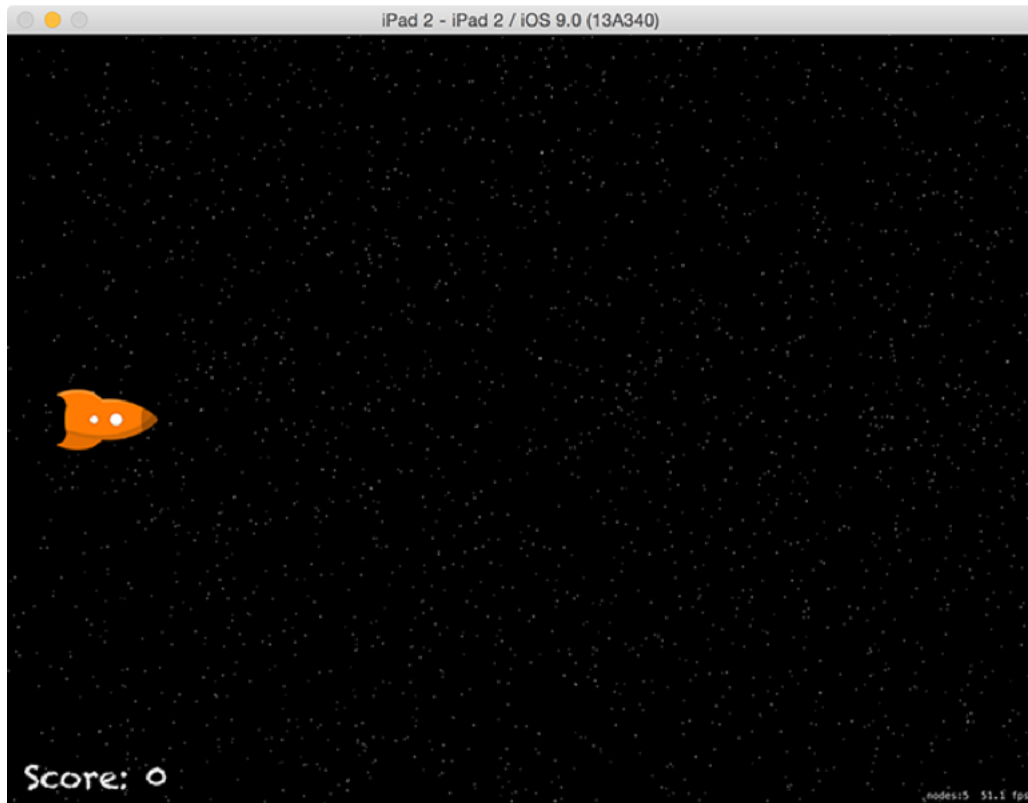
Our basic game has the user piloting a space rocket through space.

Bring on the enemies!

The point of our game is for the spaceship to survive while random "space debris" gets thrown at it. I've included three items of various shapes in this example, but you can add more easily enough. As long as the player stays alive their score ticks upwards, so clearly it's going to take some quick movement to get the highest score.

To add enemies and time to the game, we need to declare four new properties:

```
var possibleEnemies = ["ball", "hammer", "tv"]

var gameTimer: NSTimer!

var gameOver = false
```

The **possibleEnemies** array contains the names of the three images that can be used as space debris in the game: a ball, a hammer and a TV. You met **NSTimer** in project 20, and we'll be using it here to create new enemies regularly. Finally, **gameOver** is a simple boolean that will be set to true when we should stop increasing the player's score.

We need to create a new enemy on a regular basis, so the first thing to do is create a scheduled timer. I'm going to give it a timer interval of 0.35 seconds, so it will create about three enemies a second. Put this code into **didMoveToView()**:

```
gameTimer = NSTimer.scheduledTimerWithTimeInterval(0.35, target:
self, selector: "createEnemy", userInfo: nil, repeats: true)
```

Creating an enemy needs to use techniques that you've mostly seen already: it will shuffle the **possibleEnemies** array, create a sprite node using the first item in that array, position it off the right edge and with a random vertical position, then add it to the scene.

That part is old. The new part is the way we're going to create the physics body of the debris: we're going to use per-pixel collision again, tell it to collide with the player, make it move to the left at a fast speed, and give it some angular velocity. But we're also going to set to 0 its **linearDamping** and **angularDamping** properties, which means its movement and rotation will never slow down over time. Perfect for a frictionless space environment!

Here's the **createEnemy()** method in full:

```
func createEnemy() {

  possibleEnemies =
GKRandomSource.sharedRandom().arrayByShufflingObjectsInArray(possible
Enemies) as! [String]

  let randomDistribution = GKRandomDistribution(lowestValue: 50,
highestValue: 736)


  let sprite = SKSpriteNode(imageNamed: possibleEnemies[0])

  sprite.position = CGPoint(x: 1200, y: randomDistribution.nextInt())

  addChild(sprite)


  sprite.physicsBody = SKPhysicsBody(texture: sprite.texture!, size:
sprite.size)

  sprite.physicsBody?.categoryBitMask = 1

  sprite.physicsBody?.velocity = CGVector(dx: -500, dy: 0)

  sprite.physicsBody?.angularVelocity = 5

  sprite.physicsBody?.linearDamping = 0

  sprite.physicsBody?.angularDamping = 0
}
```

Make sure you add **import GameplayKit** at the top of the file so the array shuffling code works. That method uses a new way of generating random numbers that we haven't covered before: it uses the **GKRandomDistribution** class from GameplayKit to generate a random number between 50 and 736 inclusive.

Now that lots of debris will appear, we need to make sure we remove their nodes once they are invisible. In this game, that means removing nodes from the scene once they are effectively useless because they have passed the player. This will be done using a check in the **update()** method: if any node is beyond X position -300, we'll consider it dead.

The **update()** method is also a good place to make our score increment all the time. All we need to do is check whether **gameOver** is still false, and add one to the score if so. Here's the code for the **update()** method:

```
override func update(currentTime: CFTimeInterval) {
  for node in children {
    if node.position.x < -300 {
      node.removeFromParent()
    }
  }


  if !gameOver {
    score += 1
  }
}
```
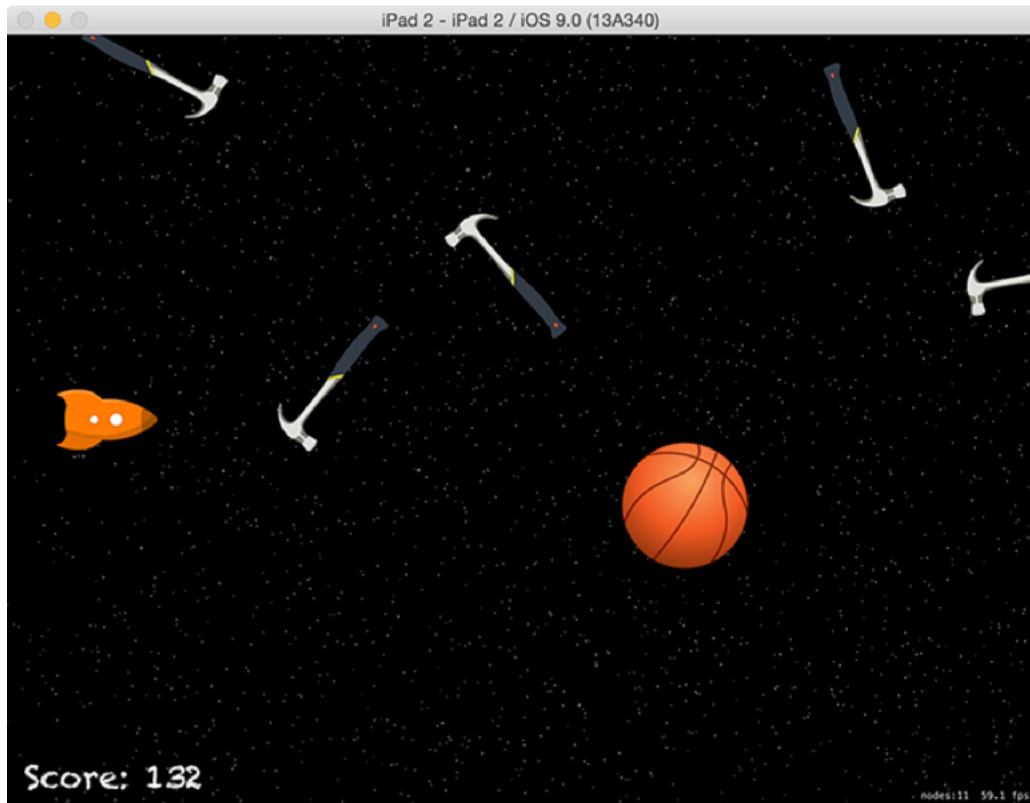
Random deadly objects are thrown at the player, who scores points simply by staying alive.

Making contact

Check your clock, because remarkably we're just two methods away from finishing this game! Predictably, the two methods are critically important: one to move the player around the screen, and one to handle collisions between the player and the space debris.

Handling player movement is as simple as implementing the **touchesMoved()** method. We will, like always, need to use the **locationInNode()** method to figure out where on the screen the user touched. But this time we're going to clamp the player's Y position, which in plain English means that we're going to stop them going above or below a certain point, keeping them firmly in the game area.

I'll be clamping the player's position so they can't overlap the score label, and I'll apply the same restriction on top so that the player has a symmetrical channel to fly through. This is a cinch to do, so here's the **touchesMoved()** method:

```
override func touchesMoved(touches: Set<UITouch>, withEvent event:
UIEvent?) {
  guard let touch = touches.first else { return }
  var location = touch.locationInNode(self)

  if location.y < 100 {
    location.y = 100
  } else if location.y > 668 {
    location.y = 668
  }

  player.position = location
}
```

Our last task is to end the game when the player hits any piece of space debris. This is all code you know already: we're going to create a particle emitter, position it where the player is (or was!), and add the explosion to the scene while removing the player. In this game we're also going to set **gameOver** to be true so that the **update()** method stops adding to their

score. Here's all the code:

```
func didBeginContact(contact: SKPhysicsContact) {
    let explosion = SKEmitterNode(fileNamed: "explosion")!
    explosion.position = player.position
    addChild(explosion)

    player.removeFromParent()

    gameOver = true
}
```

Wrap up

That's it! We just made a game in 20 minutes or so, which shows you just how fast SpriteKit is. I even showed you how per-pixel collision detection works (it's so easy!), how to advance particle systems so they start life with some history behind them, and how to adjust linear and angular damping so that objects don't slow down over time.

If you're tempted to work on this project some more, you could start by fixing a bug: if the player gets in a difficult position, they can just remove their finger from the screen then touch somewhere else to have the spaceship immediately jump there. How could you fix this? Well, one easy way is to add code for **touchesEnded()** that terminates the game if the player stops touching the screen.

If you're looking for something bigger to try, how about turn this game into a full space shooter. To do this, you need to create lasers going the opposite way, then make those lasers also collide with the space debris. In terms of controls, it wouldn't be hard to use **touchesMoved()** to move the player and **touchesBegan()** to fire lasers.

Project 24

Swift Extensions

Try your hand at improving the built-in data types of Swift.

Setting up

There's one technique I've been patiently waiting to show you since this series started, and now is the right time: extensions. Cunningly, this is not at all the same thing I already showed you in project 16, which was when we created an extension to Safari. This time we're going to create extensions for Swift – literally extending the language so it can do more things.

This isn't complicated, honest. We're not trying to make Swift do things it wasn't designed to do; in fact, I'd wager that language extensions are used in nearly all major projects that are shipping today. Why? Because they let you attach functionality to data types you didn't create. You've seen time and time again how we can add any methods we want to our own classes. Well, extensions let you do that to other classes and structs, including Apple's own.

There isn't enough in extensions to give you an intellectual challenge, so you're going to learn about extensions while using a Swift playground. We haven't used these yet because they haven't been appropriate, but now is a good chance. I'm also going to take this opportunity to outline in more detail the differences between functions and methods.

In Xcode, go to the File menu and choose New > Playground. Name it Project24, make sure iOS is selected as the platform, then choose Next and save it somewhere you can find later. Swift playgrounds are split into two columns: the left half contains your code, the right contains your output.

Adding to integers

I've been asking you to use a file called Helper.swift several times so far in this series, and it's basically a collection of interesting functions that do a handful of difficult tasks. However, that file is messy: it has function names like **RandomCGFloat()** and **RandomColor()** mixed together even though they do quite different things.

This gets confusing. It gets confusing because we don't know where these functions come from, it gets confusing because we're filling the code completion database with global functions that have similar names, and it gets confusing because these functions don't all take uniform parameters. Extensions can help us fix the first two, because it lets us move these global functions to be methods inside a class or struct.

We're going to start with an extremely simple extension so you can get a basic grip on how things work before moving on to more complicated examples.

Let's start with an extension that adds one to an integer. Yes, I realise this is essentially just **+= 1**, but we're starting simple. Put this in your playground:

```
import UIKit


var myInt = 0
```

This code will be evaluated immediately, so in the right column you'll see 0. This tells you that the **myInt** variable has the value 0. Now add this to the playground, just beneath the **import UIKit** statement:

```
extension Int {

  func plusOne() -> Int {

    return self + 1

  }

}
```

There are two things in there that are new:

1. **extension Int** tells Swift that we want to add functionality to its **Int** struct. We could have used **String**, **Array**, **UIButton** or whatever instead, but **Int** is a nice easy one to start.
2. **self + 1** is new because so far we've only used **self** to refer to a view controller or SpriteKit scene. Well, **self** in this case refers to the number that was used to call this method.

How the extension works will become clear once you use it. Put this line just below the current **myInt** line:

```
var myInt = 0
myInt.plusOne()
```

In the right column you'll now see 0 for the first line and 1 for the second, so calling **plusOne()** has returned a number one higher than the number we called it on. Note: unlike regular Swift files, playground code is executed *linearly* – i.e., top to bottom. That means you need to put the extension code above where you use it.

 The extension has been added to all integers, so you can even call it like this:

```
5.plusOne()
```

When you do that, you'll see 6 in the output column.

Our little extension adds 1 to its input number and returns it to the caller, but *doesn't* modify the original value. Try typing this:

```
var myInt = 10
myInt.plusOne()
myInt
```

Using a variable by itself tells the playground just to output its value, so in the output column you'll see 10, then 11, then 10 again. This is the original value, the return from the **plusOne()** method, and the original, unchanged value.

To push things a little further, let's modify the **plusOne()** method so that it doesn't return anything, instead modifying the instance itself – i.e., the input integer.

To make that happen, you might think we need to do something like this:

```
extension Int {
  func plusOne() {
    self += 1
  }
}
```

That removes the return value because we aren't returning anything now, and it uses the **+=** operator to add one to **self**. But this won't work, and in fact Xcode will give you a wonderfully indecipherable error message: "Cannot invoke += with an argument of type Int."

That error will undoubtedly make you think, "surely += and Ints go together like George Lucas and endless Star Wars remakes," but what Xcode really means is that it Swift doesn't let you modify **self** inside an extension by default. The reason is that we could call **plusOne()** using **5.plusOne()**, and clearly you can't modify the number 5 to mean something else.

So, Swift forces you to declare the method **mutating**, meaning that it will change its input. Change your extension to this:

```
extension Int {
  mutating func plusOne() {
    self += 1
  }
}
```

…and now the error message will go away. Once you have declared a method as being **mutating**, Swift knows it will change values so it won't let you use it with constants. For example:

```
var myInt = 10
myInt.plusOne()


let otherInt = 10
otherInt.plusOne()
```

The first integer will be modified correctly, but the second will fail because Swift won't let you modify constants.

Cleaning up the mess

I've been using the words *function* and *method* a lot so far, but haven't stopped to explain the difference. Well, there is a difference, but it's not really that important despite what many pedants might tell you. Heck, even Swift uses **func** for both functions and methods.

The main difference between a function and a method is that a method belongs to a particular class, struct or enum. Methods that belong to a class have access to all the class's data, and are useful because they help to encapsulate functionality. This means that an object owns its data, and the same object is responsible for manipulating that data using its methods. Functions, however, are global beasts: they don't belong to a particular class, so they can be called anywhere.

There is one other difference between the two in Swift, and that's how parameters are handled. In project 14 we used a method called **show()** to show enemies, and it accepted as a parameter how long to wait until hiding the enemies. As I said then, this would by default be callable like this:

```
show(1)
```

Problem is, the 1 is meaningless. As a result, Apple makes their own method names longer to try to describe what the first parameter is doing, for example they would probably have called that method **showWithHideDelay()** so that when you call it with **1** you could read the code aloud and have it make sense.

The solution we used at the time was to force a label on the parameter. In Swift, this is called an external parameter name, and it means that we had to call **show(hideTime: 1)** to make the call work.

This concept of internal parameter name (the name you use inside the method/function) and external parameter name (the name you use when calling the method/function) is where Swift's use of methods and functions can get confusing. Here are the rules:

1. By default, methods and functions have automatic external parameter names, but not for the first parameter. This means you call things like **obj.myFunc(1, age: 101, sound: "Moo", alive: false)**. The first parameter doesn't have a forced external name because of Apple's use of **veryLongDescriptiveMethodNames()**.

2. Both methods and functions can have external parameter names defined separately for the internal parameter names.
3. Both methods and functions can be forced to have external parameter names that match their internal parameter names by repeating the name.

The first and last of these behaviors are automatic: Swift forces external parameters for functions and methods (except the first one), and we already looked at the fourth one – if you remember, we ended up defining the **show()** method like this:

```
func show(hideTime hideTime: Double) {
```

Doubling the parameter name means "make this the external parameter name as well as the internal." What hasn't been covered yet is the second behavior: how to define one parameter name for external use and one for internal. Well, the good news is that it's really easy to do; the bad news is that the syntax looks a little clumsy at first because you just use two different parameter names.

As a working example, let's take the below function, **RandomInt()**:

```
func RandomInt(min: Int, max: Int) -> Int {

   if max < min { return min }

   return Int(arc4random_uniform(UInt32((max - min) + 1))) + min

}
```

That's designed to generate a random number in the same way GameplayKit does, except this code works on iOS versions earlier than 9.0 – helpful if you really need backwards compatibility.

Remember, Swift already forces users to provide the parameter name for the second and subsequent parameters, but the first one is used without a parameter name, like this:

```
RandomInt(10, max: 10)
```

If we wanted to force users to use the parameter name for **min** as well as **max**, we'd need to double the parameter name, like this:

```
func RandomInt(min min: Int, max: Int) -> Int {
  if max < min { return min }
  return Int(arc4random_uniform(UInt32((max - min) + 1))) + min
}
```

Using that, the code must now be called using **RandomInt(min: 1, max: 10)**. But if we wanted to use different external parameter names from internal parameter names, we'd need to specify them twice:

```
func RandomInt(minimum min: Int, maximum max: Int) -> Int {
  if max < min { return min }
  return Int(arc4random_uniform(UInt32((max - min) + 1))) + min
}
```

There are still only two parameters in there, but they now have two names. The first, **minimum min**, means "call me using minimum, but inside the method use min," and the latter means "call me using maximum, but inside the method use max." So, the method body hasn't changed, but it now must be called using **RandomInt(minimum: 1, maximum: 10)**.

With the parameter names cleaned up nicely, it's time to turn this into an extension so that it is neatly organised. You've already tried a basic extension, so you should be able to see that we can make **RandomInt()** an extension of **Int** like this:

```
extension Int {
  mutating func plusOne() {
    self += 1
  }
```

```
  static func random(min min: Int, max: Int) -> Int {

    if max < min { return min }

    return Int(arc4random_uniform((max - min) + 1)) + min

  }

}
```

Did you hit a mental speed bump while reading that code? I hope so, because I snuck two new things in there, both of which are important:

1. I renamed the method to be just **random()**, because methods must belong to a class, struct or enum, and this one belongs to **Int** so we know it's for integers – the "Int" part of **RandomInt()** is self-evident. Also, methods always start with a lowercase letter, hence **random()**2. I added a new keyword: **static**. This means "this method belongs to the **Int** type, not to specific instances of that type.

You're probably thinking that **static** doesn't make any sense. But look at this two lines of code and tell me which one makes more sense:

```
Int.random(min: 1, max: 10)

10.random(min: 1, max: 10)
```

The former means "call this method on the **Int** type," and the latter means "call this method on the number 5." As you might imagine, 5 is an instance of Int, and that's the difference **static** makes: do you want to call this method on the type itself, or do you want to call this on an instance of the type?

You can extend any data type using this technique. For example, if you wanted to extend **UIColor** so that it has another built-in color, you would use this:

```
extension UIColor {

  static func chartreuseColor() -> UIColor {
```

```
        return UIColor(red: 0.875, green: 1, blue: 0, alpha: 1)
    }
}
```

Extensions for brevity

Like I said earlier, it's extremely common for developers to use extensions to add functionality to things. In some ways, extensions are similar to subclasses, because we could easily subclass **UIView** and add new methods to it like this:

```
func fadeOut(duration: NSTimeInterval) {

  UIView.animateWithDuration(duration) { [unowned self] in

    self.alpha = 0

  }

}
```

So why use extensions at all? The main reason is extensibility: extensions work across all data types, and they don't conflict when you have more than one. That is, we could make a **UIView** subclass that adds **fadeOut()**, but what if we find some open source code that contains a **spinAround()** method? We would have to copy and paste it in to our subclass, or perhaps even subclass again.

With extensions you can have ten different pieces of functionality in ten different files – they can all modify **UIView** directly, and you don't need to subclass anything. A common naming scheme for naming your extension files is Type+Modifier.swift, for example Array +Shuffling.swift. That's a file we've used multiple times so far, and now you can look inside it and see that's an extension to the **Array** type.

Having extensions in different files also means you can reuse individual extensions just by copying a file into a project. That's what we've been doing with the array shuffling code: every time we've needed it, we've just copied the file into our project and immediately the **shuffle()** method is available.

If you find yourself wanting to make views fade out often, an extension is perfect for you. If you find yourself trimming whitespace off strings frequently, you'll probably get tired of using this monstrosity:

```
str =
str.stringByTrimmingCharactersInSet(NSCharacterSet.whitespaceAndNewli
```

```
neCharacterSet())
```

…so why not just make an extension like this:

```
extension String {

  mutating func trim() {

    self =
stringByTrimmingCharactersInSet(NSCharacterSet.whitespaceAndNewlineCh
aracterSet())

  }

}
```

You can extend as much as you want, although it's good practice to keep differing functionality separated into individual files. That said, "good practice" and "I'm up against a deadline" are rarely the same, so expect to see extensions named String+Additions.swift that add a collection of unrelated things to the **String** struct.

If you want to see what super-charging Swift's built-in data types can look like, look up the **ExSwift** extension catalog on GitHub at **https://github.com/pNre/ExSwift**.

Wrap up

Swift extensions are the smart way to add functionality to existing types, and you're going to meet them time and time again – and hopefully write quite a few of your own too. They aren't all-encompassing, because they don't let you add properties to a class whereas a full subclass would, but they are easy to use and easy to share so I'm sure you'll use them frequently.

In this project, you've also learned a little of how useful Swift playgrounds can be for prototyping code, because the immediate feedback you get makes it extremely easy to try things out and make quick adjustments.

Project 25

Selfie Share

Make a multipeer photo sharing app in just 150 lines of code.

Setting up

This project is going to give you some practice with collection views, the image picker and GCD, but at the same time introduce you to a new technology called the multipeer connectivity framework. This is a way to let users form impromptu connections to each other and send data, rather like BitTorrent.

The app we're going to make will show photos of your choosing in a collection view. That much is easy enough, because we did pretty much that already in project 10. But this time there's a subtle difference: when you add a photo it's going to automatically send it to any other devices you are currently connected to, and any photos they select will appear for you.

Create a new Single View Application project in Xcode, naming it Project25 and selecting any device you want. Please note: the nature of peer-to-peer apps is that you need to have at least two copies of your app running, one to send and one to receive. Because the iOS simulator only lets you run one simulated app at a time, this means you'll need to have one physical device alongside your simulator.

Importing photos again

We've used the **UIImagePickerController** class twice now: once in project 10 and again in project 13, so I hope you're already comfortable with it. We also used a collection view in project 10, but we haven't used it since so you might not be quite so familiar with it.

Open Main.storyboard in Interface Builder, then embed the default view controller inside a navigation controller. Now place a collection view so that it fills the screen and set its background to be white.

With the collection view selected, set cell size to be 145 wide and 145 high, and give all four section insets a value of 10. Click inside the prototype cell that Xcode made for you and give it the reuse identifier "ImageView". Finally, drop an image view into the cell so that it occupies all its space, and give it the tag 1000.

All the constraints in this project can be made automatically, so go to the Editor menu and choose Resolve Auto Layout Issues > Add Missing Constraints.

Before we're done with the storyboard, please make our view controller the data source and delegate of the collection view – Ctrl-drag from the collection view to the view controller icon just above the view. Now use the assistant editor to create an outlet for it called **collectionView**.

Go back to the standard editor and open up ViewController.swift, because we're finished with the user interface and can plug in the code. Note that almost all of this has been covered in other projects, so we're not going to waste much time here when there are far more interesting things around the corner!

To start, add a right bar button item that uses the system's camera icon and calls an **importPicture()** method that we'll write shortly. I'm also going to customise the title of the view controller so that it isn't empty, so here's the new **viewDidLoad()** method:

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

  title = "Selfie Share"
  navigationItem.rightBarButtonItem =
```

```
UIBarButtonItem(barButtonSystemItem: .Camera, target: self, action:
"importPicture")

}
```

Next, let's make the collection view work correctly. You need to make your view controller conform to the **UICollectionViewDataSource** and **UICollectionViewDelegate** protocols, but while you're there you might as well also add the **UINavigationControllerDelegate** and **UIImagePickerControllerDelegate** protocols too, because we need those to work with the image picker.

We will store all our apps images inside a **UIImage** array, so please add this property:

```
var images = [UIImage]()
```

We're going to use that array to know how many items are in our collection view, so you should know to write this method yourself:

```
func collectionView(collectionView: UICollectionView,
numberOfItemsInSection section: Int) -> Int {

    return images.count

}
```

Next comes the only thing out of the ordinary in all this code, which is the **cellForItemAtIndexPath** method for our collection view. To get us through this part of the project as quickly as possible, I took a shortcut: when it comes to configuring cells to look correct, we can dequeue using the identifier "ImageView" then find the **UIImageView** inside them without a property.

I asked you to set the tag of the image view to be 1000, and here's why: all **UIView** subclasses have a method called **viewWithTag()**, which searches for any views inside itself (or indeed itself) with that tag number. We can find our image view just by using this method, although I'll still use if/let and a conditional typecast to be sure.

Here's the code for **cellForItemAtIndexPath**:

```
func collectionView(collectionView: UICollectionView,
cellForItemAtIndexPath indexPath: NSIndexPath) ->
UICollectionViewCell {

  let cell =
collectionView.dequeueReusableCellWithReuseIdentifier("ImageView",
forIndexPath: indexPath)

  if let imageView = cell.viewWithTag(1000) as? UIImageView {

    imageView.image = images[indexPath.item]

  }

  return cell

}
```

That makes the collection view work just fine, but we still need three more methods in order to get our basic app ready, and these are the methods to handle the image picker. If this code isn't identical to the code we've previously written, it might as well be – check project 10 if your memory is bad!

```
func importPicture() {

  let picker = UIImagePickerController()

  picker.allowsEditing = true

  picker.delegate = self

  presentViewController(picker, animated: true, completion: nil)

}


func imagePickerController(picker: UIImagePickerController,
didFinishPickingMediaWithInfo info: [String : AnyObject]) {

  var newImage: UIImage
```

```swift
    if let possibleImage = info["UIImagePickerControllerEditedImage"]
as? UIImage {

        newImage = possibleImage

    } else if let possibleImage =
info["UIImagePickerControllerOriginalImage"] as? UIImage {

        newImage = possibleImage

    } else {

        return

    }


    dismissViewControllerAnimated(true, completion: nil)


    images.insert(newImage, atIndex: 0)

    collectionView.reloadData()

}


func imagePickerControllerDidCancel(picker: UIImagePickerController)
{

    dismissViewControllerAnimated(true, completion: nil)

}
```

Done. No more boring old code now, it's all new from here. You can run the app if you want, but there's really not much point other than being sure your code works – this is just a cut-down version of project 10 so far.

Going peer to peer

Add a left bar button item to our view controller, using the "add" system icon, and making it call a method called **showConnectionPrompt()**. We're going to make that method ask users whether they want to connect to an existing session with other people, or whether they want to create their own. Here's the code for the bar button item – put this in **viewDidLoad()**:

```
navigationItem.leftBarButtonItem =
UIBarButtonItem(barButtonSystemItem: .Add, target: self, action:
"showConnectionPrompt")
```

Asking users to clarify how they want to take an action is of course the purpose of **UIAlertController** as an action sheet, and our **showConnectionPrompt()** method is going to use one to ask users what kind of connection they want to make. Put this code into your view controller:

```
func showConnectionPrompt() {

  let ac = UIAlertController(title: "Connect to others", message:
nil, preferredStyle: .ActionSheet)

  ac.addAction(UIAlertAction(title: "Host a session",
style: .Default, handler: startHosting))

  ac.addAction(UIAlertAction(title: "Join a session",
style: .Default, handler: joinSession))

  ac.addAction(UIAlertAction(title: "Cancel", style: .Cancel,
handler: nil))

  presentViewController(ac, animated: true, completion: nil)

}
```

Now, here's where it gets trickier. Multipeer connectivity requires four new classes:

1. **MCSession** is the manager class that handles all multipeer connectivity for us.
2. **MCPeerID** identifies each user uniquely in a session.
3. **MCAdvertiserAssistant** is used when creating a session, telling others that we exist and handling invitations.

4. **MCBrowserViewController** is used when looking for sessions, showing users who is nearby and letting them join.

We're going to use all four of them in our app, but only three need to be properties – add these to your view controller:

```
var peerID: MCPeerID!

var mcSession: MCSession!

var mcAdvertiserAssistant: MCAdvertiserAssistant!
```

You'll get errors on each of those lines because Xcode doesn't recognise the classes. You can fix that by asking it to import the multipeer framework – add this line just above **import UIKit**:

```
import MultipeerConnectivity
```

Depending on what users select in our alert controller, we need to call one of two methods: **startHosting()** or **joinSession()**. Because both of these are coming from the result of a **UIAction** being tapped, both methods must accept a **UIAlertAction!** as their only parameter.

Before I show you the code to get multipeer connectivity up and running, I want to go over what they will do. Most important of all is that all multipeer services on iOS must declare a service type, which is a 15-digit string that uniquely identify your service. Those 15 digits can contain only the letters A-Z, numbers and hyphens, and it's usually preferred to include your company in there somehow.

Apple's example is, "a text chat app made by ABC company could use the service type **abc-txtchat**"; for this project I'll be using **hws-project25**.

This service type is used by both **MCAdvertiserAssistant** and **MCBrowserViewController** to make sure your users only see other users of the same app. They both also want a reference to your **MCSession** instance so they can take care of connections for you.

We're going to start by initialising our **MCSession** so that we're able to make connections.

Put this code into **viewDidLoad()**:

```
peerID = MCPeerID(displayName: UIDevice.currentDevice().name)

mcSession = MCSession(peer: peerID, securityIdentity: nil,
encryptionPreference: .Required)

mcSession.delegate = self
```

As you can see in that code, we're creating an **MCPeerID** object using the name of the current device, which will usually be something like "Paul's iPhone". That ID is then used to create the session, along with the encryption option of **.Required** to ensure that any data transferred is kept safe.

Don't worry about conforming to any extra protocols just yet; we'll do that in just a minute.

At this point, the code for **startHosting()** and **joinSession()** will look quite trivial. Here goes:

```
func startHosting(action: UIAlertAction!) {

  mcAdvertiserAssistant = MCAdvertiserAssistant(serviceType: "hws-
project25", discoveryInfo: nil, session: mcSession)

  mcAdvertiserAssistant.start()

}


func joinSession(action: UIAlertAction!) {

  let mcBrowser = MCBrowserViewController(serviceType: "hws-
project25", session: mcSession)

  mcBrowser.delegate = self

  presentViewController(mcBrowser, animated: true, completion: nil)

}
```

We're making our view controller the delegate of a second object, so that's two protocols we need to conform to in order to fix our current compile failures. Easily done: add **MCSessionDelegate** and **MCBrowserViewControllerDelegate** to your class definition…

and now there are even more errors, because we need to implement lots of new methods.

Invitation only

Merely by saying that we conform to the **MCSessionDelegate** and **MCBrowserViewControllerDelegate** protocols, your code won't build any more. This is because the two protocols combined have seven required methods that you need to implement just to be compatible.

Helpfully, for this project you can effectively ignore three of them, two more are trivial, and one further is just for diagnostic information in this project. That leaves only one method that is non-trivial and important to the program.

Let's tackle the ones we can effectively ignore. Of course, you can't *ignore* required methods, otherwise they wouldn't be required. But these methods aren't ones that do anything useful to our program, so we can just create empty methods. Remember, once you've said you conform to a protocol, Xcode's code completion is updated so you can just start typing the first few letters of a method name in order to have Xcode prompt you with a list to choose from.

Here are the three methods that we need to provide, but don't actually need any code inside them:

```
func session(session: MCSession, didReceiveStream stream:
NSInputStream, withName streamName: String, fromPeer peerID:
MCPeerID) {


}


func session(session: MCSession, didStartReceivingResourceWithName
resourceName: String, fromPeer peerID: MCPeerID, withProgress
progress: NSProgress) {


}


func session(session: MCSession, didFinishReceivingResourceWithName
resourceName: String, fromPeer peerID: MCPeerID, atURL localURL:
NSURL, withError error: NSError?) {
```

```
}
```

They are really long, so make sure you use code completion!

The two methods we're going to implement that are trivial are both for the multipeer browser: one is called when it finishes successfully, and one when the user cancels. Both methods just need to dismiss the view controller that is currently being presented, which means this is their entire code:

```
func browserViewControllerDidFinish(browserViewController:
MCBrowserViewController) {

  dismissViewControllerAnimated(true, completion: nil)

}
```

```
func browserViewControllerWasCancelled(browserViewController:
MCBrowserViewController) {

  dismissViewControllerAnimated(true, completion: nil)

}
```

Brilliant! Isn't it easy being a coder?

There are two methods left: one that is used in this project only for diagnostic information, and one that's actually useful. Let's eliminate the diagnostic method first so that we can focus on the interesting bit.

When a user connects or disconnects from our session, the method **session(_:peer:didChangeState:)** is called so you know what's changed – is someone connecting, are they now connected, or have they just disconnected? We're not going to be using this information in the project, but I do want to show you how it might be used by printing out some diagnostics. This is helpful for debugging, because it means you can look in Xcode's debug console to see these messages and know your code is working.

When this method is called, you'll be told what peer changed state, and what their new state is. There are only three possible session states: not connected, connecting, and connected.

So, we can make our app print out useful information just by using switch/case and a bit of **println()**:

```
func session(session: MCSession, peer peerID: MCPeerID,
didChangeState state: MCSessionState) {

  switch state {

  case MCSessionState.Connected:

    print("Connected: \(peerID.displayName)")


  case MCSessionState.Connecting:

    print("Connecting: \(peerID.displayName)")


  case MCSessionState.NotConnected:

    print("Not Connected: \(peerID.displayName)")

  }

}
```

That just leaves one more method that must be implemented before you're fully compliant with the protocols, but before I talk you through it you need to know how the core of this app works. It's not hard, but it is important, so listen carefully!

Right now, when we add a picture to the collection view it is shown on our screen but doesn't go anywhere. We're going to add some code to the image picker's **didFinishPickingMediaWithInfo** method so that when an image is added it also gets sent out to peers.

Sending images across a multipeer connection is remarkably easy. In project 10 you met the function **UIImageJPEGRepresentation()**, which converts a **UIImage** object into an **NSData** so it can be saved to disk. Well, **MCSession** objects have a **sendData()** method that will ensure that data gets transmitted reliably to your peers.

Once the data arrives at each peer, the method **session(_:didReceiveData:fromPeer:)** will get called with that data, at which point we can create a **UIImage** from it and add it to our

images array. There is one catch: when you receive data it might not be on the main thread, and you never manipulate user interfaces anywhere but the main thread, right? Right.

Here's the final protocol method, to catch data being received in our session:

```
func session(session: MCSession, didReceiveData data: NSData,
fromPeer peerID: MCPeerID) {

  if let image = UIImage(data: data) {

    dispatch_async(dispatch_get_main_queue()) { [unowned self] in

      self.images.insert(image, atIndex: 0)

      self.collectionView.reloadData()

    }

  }

}
```

Take note of the call to **dispatch_async()** to ensure we definitely only manipulate the user interface on the main thread!

The final piece of code to finish up this whole project is the bit that sends image data to peers. This is so easy you might not even believe me. In fact, the code is only as long as it is because there's some error checking in there.

This final code needs to:

1. Check if there are any peers to send to.
2. Convert the new image to an **NSData** object.
3. Send it to all peers, ensuring it gets delivered.
4. Show an error message if there's a problem.

Converting that into code, you get the below. Put this into your **didFinishPickingMediaWithInfo** method, just after the call to **reloadData()**:

```
// 1
if mcSession.connectedPeers.count > 0 {
```

```
    // 2

    if let imageData = UIImagePNGRepresentation(newImage) {

        // 3

        do {

            try mcSession.sendData(imageData, toPeers:
mcSession.connectedPeers, withMode: .Reliable)

        } catch let error as NSError {

            let ac = UIAlertController(title: "Send error", message:
error.localizedDescription, preferredStyle: .Alert)

            ac.addAction(UIAlertAction(title: "OK", style: .Default,
handler: nil))

            presentViewController(ac, animated: true, completion: nil)

        }

    }

}
```

Yes, the code to ensure data gets sent intact to all peers, as opposed to having some parts lost in the ether, is just to use transmission mode **.Reliable** – nothing more.

Now, that code does something new: you've seen **try!** and **try?** before, but this time I'm using plain old **try** without a question or exclamation mark. This means "try running this code, and let me know if it fails." To make this work, you need to surround your code in a **do/ catch** block as shown above. When any error is thrown in the **do** block, your program immediately jumps straight to the **catch** block where you can handle it – or in our case show a message.

Anyway, I hope you'll agree that the multipeer connectivity framework is super easy to use. The advertiser assistant takes care of telling the world that our app is looking for connections, as well as handling people who want to join. The browser controller takes care of finding all compatible sessions, and sending invitations. Our job is just to hook it all together with a nice user interface, then relax and wait for the App Store riches to come in. Sort of.

Remember: to test your project, you'll need to either run it on multiple devices, or use one device and one simulator.

Wrap up

Multipeer connectivity is something that used to be awfully hard, but in iOS it's only 150 lines of code to produce this entire app – and over half of that is code for the collection view and the image picker! The advantage it has compared to traditional data sharing over Wi-Fi is that multipeer can use an existing Wi-Fi network, or can silently create a new Wi-Fi network or even a Bluetooth network depending on what's available. All this is an implementation detail that Apple solves on your behalf.

If you'd like to take this project further, add a button that will show a table view listing the names of all devices currently connected to the session. You could also try sending text messages across the wire – there's a **dataUsingEncoding()** method for strings that converting a string to an **NSData**. Use it with the parameters **NSUTF8Encoding** and **false**.

Project 26

Marble Maze

Respond to device tilting by steering a ball around a vortex maze.

Setting up

In this game project you'll create a rolling ball game for iPad, using the accelerometer – you tip your device, and the balls rolls in that direction, hopefully avoiding holes as you go.

Along with the accelerometer, you're also going to learn how to load levels, how to have fine-grained contact bitmasks, how to reverse arrays, and how to write code that executes in the simulator but not on devices (or vice versa). So, you learn things, you make a cool game, and I get to bask in the warmth of knowing that your Swift mastery is growing more than ever.

Create a new SpriteKit project, name it Project26, and set it for iPad only. Make sure it uses a fixed **landscape right** orientation, which is more restrictive than we usually use. We can't enable landscape left because we'll be tilting the device in all directions, and it would be annoying to have the device rotate because we tipped the iPad too far!

Please delete the spaceship picture and delete the contents of the **didMoveToView()** and **touchesBegan()** methods in GameScene.swift. Now download the files for this project from **GitHub** and copy its Content folder into your project.

In this project we're going to use the accelerometer, which is not supported in the iOS Simulator. To make things easier, we're going to add some code that lets you control the game through touch – it's nowhere near as fun, but at least it can be tested in the simulator.

Warning: When working with SpriteKit projects, I strongly recommend you use the iPad 2 simulator rather than iPad Air – you'll get much faster frame rates, making it much more suitable for testing.

Loading a level

We're going to start this project by looking at the biggest method in the project, and perhaps even the entire Hacking with Swift series. It's called **loadLevel()** and is responsible for loading a level file from disk and creating SpriteKit nodes onscreen.

The method isn't long because it's complicated, it's long just because it does a lot. When you finish this project one of the suggested ways to improve the code is to split this method off into small methods, so you should pay close attention to how it works!

At the core of the method it loads a level file from disk, then splits it up by line. Each line will become one row of level data on the screen, so the method will loop over every character in the row and see what letter it is. Our game will recognise five possible options: a space will mean empty space, "x" means a wall, "v" means a vortex (deadly to players), "s" means a star (awards points), and "f" means level finish.

Using this kind of very simple level text format means that you can write your levels in a text editor, and visually see exactly how they will look in your game. You've already tackled most of the code required for the skeleton of **loadLevel()**, but there are a few things I want to highlight:

 • We'll be using the **enumerate()** function again. In case you've forgotten, this loops over an array, extracting each item and its position in the array.
 • We'll be positioning items as we go. Each square in the game world occupies a 64x64 space, so we can find its position by multiplying its row and column by 32. But: remember that SpriteKit calculates its positions from the center of objects, so we need to add 32 to the X and Y coordinates in order to make everything lines up on our screen.
 • You might also remember that SpriteKit uses an inverted Y axis to UIKit, which means for SpriteKit Y:0 is the bottom of the screen whereas for UIKit Y:0 is the top. When it comes to loading level rows, this means we need to read them in reverse so that the last row is created at the bottom of the screen and so on upwards.

Here's the initial code for **loadLevel()**:

```
func loadLevel() {
    if let levelPath = NSBundle.mainBundle().pathForResource("level1",
ofType: "txt") {
        if let levelString = try? String(contentsOfFile: levelPath,
```

```
usedEncoding: nil) {

      let lines = levelString.componentsSeparatedByString("\n")


      for (row, line) in lines.reverse().enumerate() {
        for (column, letter) in line.characters.enumerate() {
          let position = CGPoint(x: (64 * column) + 32, y: (64 * row)
+ 32)


          if letter == "x" {
            // load wall
          } else if letter == "v"  {
            // load vortex
          } else if letter == "s"  {
            // load star
          } else if letter == "f"  {
            // load finish

          }
        }
      }
    }
}
```

There are lots of comments in there where we're going to do work to load the various level components. Much of the code for these is the same: load in an image, position it, give it a physics body, then add it to the scene. But they do vary, because we want the player to be able to collide with some, we want to notified of collisions with some, and so on.

But first: we're going to be using the **categoryBitMask**, **contactTestBitMask** and **collisionBitMask** properties in their fullest for this project, because we have very precise rules that make the game work. To clarify, here's what each of them mean:

- **categoryBitMask** is a number defining the type of object this is for considering collisions.
- **collisionBitMask** is a number defining what categories of object this node should collide with,
- **contactTestBitMask** is a number defining which collisions we want to be notified about.

They all do very different things, although the distinction might seem fine before you fully understand. Category is simple enough: every node you want to reference in your collision bitmasks or your contact test bitmasks must have a category attached. If you give a node a collision bitmask but not a contact test bitmask, it means they will bounce off each other but you won't be notified. If you do the opposite (contact test but not collision) it means they won't bounce off each other but you will be told when they overlap.

By default, physics bodies have a collision bitmask that means "everything", so everything bounces off everything else. By default, they also have a contact test bitmask that means "nothing", so you'll never get told about collisions.

A bitmask is a complicated beast to explain, but what it means in practice is that you can combine values together. In our game, vortexes, stars and the finish flag all have the player set for their contact test bitmask, and the player has star *and* vortex *and* finish flag.

SpriteKit expects these three bitmasks to be described using a **UInt32**, which we met briefly in project 2. It's a particular way of storing numbers, but rather than using numbers we're going to use enums with a raw value like we did in project 17. This means we can refer to the various options using names. Put this enum definition above your class in GameScene.swift:

```
enum CollisionTypes: UInt32 {
    case Player = 1
    case Wall = 2
    case Star = 4
    case Vortex = 8
    case Finish = 16
}
```

Note that your bitmasks should start at 1 then double each time. With that, let's start replacing the comments in the **loadLevel()** method with real code. First, here's how to create

a wall – replace the **// load wall** comment with this:

```
let node = SKSpriteNode(imageNamed: "block")

node.position = position


node.physicsBody = SKPhysicsBody(rectangleOfSize: node.size)

node.physicsBody!.categoryBitMask = CollisionTypes.Wall.rawValue

node.physicsBody!.dynamic = false

addChild(node)
```

It uses rectangle physics, it's not dynamic because the walls should be fixed… this is all child's play to you now, right? In project 17 you saw how to create an enum from a number, but here we're getting the number from the enum using its **rawValue** property.

Next, replace the **// load vortex** comment with this:

```
let node = SKSpriteNode(imageNamed: "vortex")

node.name = "vortex"

node.position = position

node.runAction(SKAction.repeatActionForever(SKAction.rotateByAngle(CGFloat(M_PI), duration: 1)))

node.physicsBody = SKPhysicsBody(circleOfRadius: node.size.width / 2)

node.physicsBody!.dynamic = false


node.physicsBody!.categoryBitMask = CollisionTypes.Vortex.rawValue

node.physicsBody!.contactTestBitMask = CollisionTypes.Player.rawValue

node.physicsBody!.collisionBitMask = 0

addChild(node)
```

This is a little more interesting, because it uses **rotateByAngle()** and **repeatActionForever()**

to make each vortex rotate around and around for as long the game lasts. It also sets the
contactTestBitMask property to the value of the player's category, which means we want to
be notified when these two touch.

The code to load stars and the finish flag are almost identical and quite trivial for you at this
point, so here's the code to load stars:

```
let node = SKSpriteNode(imageNamed: "star")
node.name = "star"
node.physicsBody = SKPhysicsBody(circleOfRadius: node.size.width / 2)
node.physicsBody!.dynamic = false

node.physicsBody!.categoryBitMask = CollisionTypes.Star.rawValue
node.physicsBody!.contactTestBitMask = CollisionTypes.Player.rawValue
node.physicsBody!.collisionBitMask = 0
node.position = position
addChild(node)
```

And the code to load the finish flag:

```
let node = SKSpriteNode(imageNamed: "finish")
node.name = "finish"
node.physicsBody = SKPhysicsBody(circleOfRadius: node.size.width / 2)
node.physicsBody!.dynamic = false

node.physicsBody!.categoryBitMask = CollisionTypes.Finish.rawValue
node.physicsBody!.contactTestBitMask = CollisionTypes.Player.rawValue
node.physicsBody!.collisionBitMask = 0
node.position = position
addChild(node)
```

That completes the method. It's long, but it's quite repetitive – there are several ways it could be refactored into something neater, but that would be cheating for later!

To see the fruits of your labor, add a call to **loadLevel()** in **didMoveToView()** then run your game. Remember to use the iPad 2 simulator, because iPad Air will be slow!
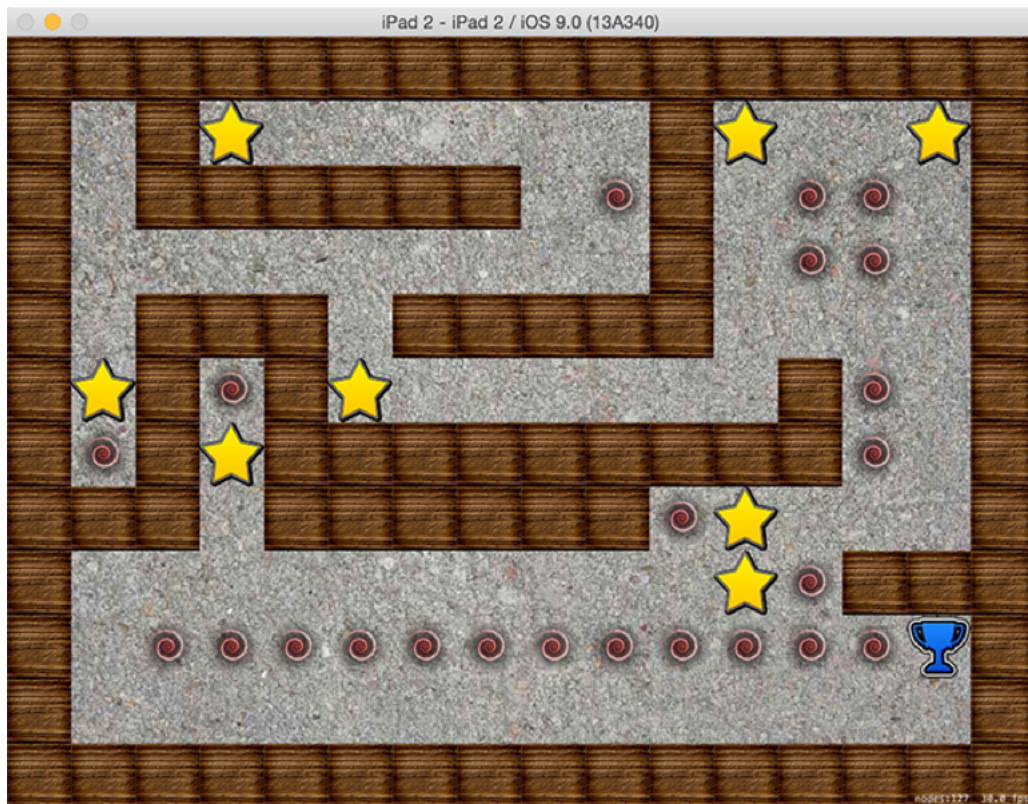
Our code has successfully parsed the level text file into something playable.

To finish off the level-loading code, we should add a background picture. You've done this many times so far, so please just go ahead and put this code into **didMoveToView()**, *before* the **loadLevel()** call:

```
let background = SKSpriteNode(imageNamed: "background.jpg")
background.position = CGPoint(x: 512, y: 384)
background.blendMode = .Replace
```

```
background.zPosition = -1

addChild(background)
```

Tilt to move

We're going to control this game using the accelerometer that comes as standard on all iPads, but it has a problem: it doesn't come as standard on any Macs, which means we either resign ourselves to testing only on devices or we put in a little hack. This course isn't calling Giving Up with Swift, so we're going to add a hack – in the simulator you'll be able to use touch, and on devices you'll have to use tilting.

To get started, add this property so we can reference the player throughout the game:

```
var player: SKSpriteNode!
```

We're going to add a dedicated **createPlayer()** method that loads the sprite, gives it circle physics, and adds it to the scene, but it's going to do three other things that are important.

First, it's going to set the physics body's **allowsRotation** property to be false. We haven't changed that so far, but it does what you might expect – when false, the body no longer rotates. This is useful here because the ball looks like a marble: it's shiny, and those reflections wouldn't rotate in real life.

Second, we're going to give the ball a **linearDamping** value of 0.5, which applies a lot of friction to its movement. The game will still be hard, but this does help a little by slowing the ball down naturally.

Finally, we'll be combining three values together to get the ball's **contactTestBitMask**: the star, the vortex and the finish.

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

```
func createPlayer() {
  player = SKSpriteNode(imageNamed: "player")
  player.position = CGPoint(x: 96, y: 672)
  player.physicsBody = SKPhysicsBody(circleOfRadius:
player.size.width / 2)
  player.physicsBody!.allowsRotation = false
```

```
    player.physicsBody!.linearDamping = 0.5


    player.physicsBody!.categoryBitMask =
CollisionTypes.Player.rawValue

    player.physicsBody!.contactTestBitMask =
CollisionTypes.Star.rawValue | CollisionTypes.Vortex.rawValue |
CollisionTypes.Finish.rawValue

    player.physicsBody!.collisionBitMask = CollisionTypes.Wall.rawValue

    addChild(player)

}
```

You can go ahead and add a call to **createPlayer()** directly after the call to **loadLevel()** inside **viewDidLoad()**. Note: you must create the player after the level, otherwise it will appear below vortexes and other level objects.

If you try running the game now, you'll see the ball drop straight down until it hits a wall, then it bounces briefly and stops. This game has players looking down on their iPad, so by default there ought to be no movement – it's only if the player tilts their iPad down that the ball should move downwards.

The ball is moving because the scene's physics world has a default gravity roughly equivalent to Earth's. We don't want that, so in **didMoveToView()** add this somewhere:

```
physicsWorld.gravity = CGVector(dx: 0, dy: 0)
```

Playing the game *now* hasn't really solved much: sure, the ball isn't moving now, but… the ball isn't moving now! This would make for a pretty terrible game on the App Store.

Before we get onto how to work with the accelerometer, we're going to put together a hack that lets you simulate the experience of moving the ball using touch. What we're going to do is catch **touchesBegan()**, **touchesMoved()**, **touchesEnded()** and **touchesCancelled()** and use them to set or unset a new property called **lastTouchPosition**. Then in the **update()** method we'll subtract that touch position from the player's position, and use it set the world's gravity.

It's a hack. And if you're happy to test on a device, you don't really need it. But if you're stuck with the iOS Simulator or are just curious, let's put in the hack. First, declare the new property:

```
var lastTouchPosition: CGPoint?
```

Now use **touchesBegan()** and **touchesMoved()** to set the value of that property using the same three lines of code, like this:

```
override func touchesBegan(touches: Set<UITouch>, withEvent event:
UIEvent?) {

  if let touch = touches.first {

    let location = touch.locationInNode(self)

    lastTouchPosition = location

  }

}


override func touchesMoved(touches: Set<UITouch>, withEvent event:
UIEvent?) {

  if let touch = touches.first {

    let location = touch.locationInNode(self)

    lastTouchPosition = location

  }

}
```

When **touchesEnded()** or **touchesCancelled()** are called, we need to set the property to be **nil** – it is optional, after all:

```
override func touchesEnded(touches: Set<UITouch>, withEvent event:
UIEvent?) {
```

```
   lastTouchPosition = nil

}


override func touchesCancelled(touches: Set<UITouch>?, withEvent
event: UIEvent?) {

   lastTouchPosition = nil

}
```

Easy, I know, but it gets (only a little!) trickier in the **update()** method. This needs to unwrap our optional property, calculate the difference between the current touch and the player's position, then use that to change the **gravity** value of the physics world. Here it is:

```
override func update(currentTime: CFTimeInterval) {

   if let currentTouch = lastTouchPosition {

      let diff = CGPoint(x: currentTouch.x - player.position.x, y:
currentTouch.y - player.position.y)

      physicsWorld.gravity = CGVector(dx: diff.x / 100, dy: diff.y /
100)

   }

}
```

This is clearly not a permanent solution, but it's good enough that you can run the app now and test it out.

Now for the new bit: working with the accelerometer. This is easy to do, which is remarkable when you think how much is happening behind the scenes.

All motion detection is done with an Apple framework called Core Motion, and most of the work is done by a class called **CMMotionManager**. Using it here won't require any special user permissions, so all we need to do is create an instance of the class and ask it to start collecting information. We can then read from that information whenever and wherever we need to, and in this project the best place is **update()**.

Add **import CoreMotion** just above the **import SpriteKit** line at the top of your game scene, then add this property:

```
var motionManager: CMMotionManager!
```

Now it's just a matter of creating the object and asking it start collecting accelerometer data. This is done using the **startAccelerometerUpdates()** method, which instructs Core Motion to start collecting accelerometer information we can read later. Put this this into **didMoveToView()**:

```
motionManager = CMMotionManager()
motionManager.startAccelerometerUpdates()
```

The last thing to do is to poll the motion manager inside our **update()** method, checking to see what the current tilt data is. But there's a complication: we already have a hack in there that lets us test in the simulator, so we want one set of code for the simulator and one set of code for devices.

Swift solves this problem by adding special compiler instructions. If the instruction evaluates to true it will compile one set of code, otherwise it will compile the other. This is particularly helpful once you realise that any code wrapped in compiler instructions that evaluate to false never get seen – it's like they never existed. So, this is a great way to include debug information or activity in the simulator that never sees the light on devices.

The compiler directives we care about are: **#if (arch(i386) || arch(x86_64))**, **#else** and **#endif**. As you can see, this is mostly the same as a standard Swift if/else block, although here you don't need braces because everything until the **#else** or **#endif** will execute.

The code to read from the accelerometer and apply its tilt data to the world gravity look like this:

```
if let accelerometerData = motionManager.accelerometerData {
  physicsWorld.gravity = CGVector(dx:
```

```
accelerometerData.acceleration.y * -50, dy:
accelerometerData.acceleration.x * 50)

}
```

The first line safely unwraps the optional accelerometer data, because there might not be any available. The second line changes the gravity of our game world so that it reflects the accelerometer data. You're welcome to adjust the speed multipliers as you please; I found a value of 50 worked well.

Note that I passed accelerometer Y to **CGVector**'s X and accelerometer X to **CGVector**'s Y. This is not a typo! Remember, your device is rotated to landscape right now, which means you also need to flip your coordinates around.

We need to put that code inside the current **update()** method, wrapped inside the new compiler directives. Here's how the method should look now:

```
override func update(currentTime: CFTimeInterval) {

#if (arch(i386) || arch(x86_64))

  if let currentTouch = lastTouchPosition {

    let diff = CGPoint(x: currentTouch.x - player.position.x, y:
currentTouch.y - player.position.y)

    physicsWorld.gravity = CGVector(dx: diff.x / 100, dy: diff.y /
100)

  }

#else

  if let accelerometerData = motionManager.accelerometerData {

    physicsWorld.gravity = CGVector(dx:
accelerometerData.acceleration.y * -50, dy:
accelerometerData.acceleration.x * 50)

  }

#endif

}
```

If you can test on a device, please do. It took only a few lines of code, but the game is now adapting beautifully to device tilting!

Contacting but not colliding

All the game is missing now is some challenge, and that's where our star and vortex level elements come in. Players will get one point for every star they collect, and lose one point every time they fall into a vortex. To track scores, we need a property to hold the score and a label to show it, so add these now:

```
var scoreLabel: SKLabelNode!

var score: Int = 0 {
  didSet {
    scoreLabel.text = "Score: \(score)"
  }
}
```

We're going to show the label in the top-left corner of the screen, so add this to **didMoveToView()**:

```
scoreLabel = SKLabelNode(fontNamed: "Chalkduster")
scoreLabel.text = "Score: 0"
scoreLabel.horizontalAlignmentMode = .Left
scoreLabel.position = CGPoint(x: 16, y: 16)
addChild(scoreLabel)
```

When a collision happens, we need to figure out whether it was the player colliding with a star, or the star colliding with a player – the same semi-philosophical problem we had in project 11. And our solution is identical too: figure out which is which, then call another method.

First, we need to make ourselves the contact delegate for the physics world, so make your class conform to **SKPhysicsContactDelegate** then add this line in **didMoveToView()**:

```
physicsWorld.contactDelegate = self
```

We already know which node is our player, which means we know which node *isn't* our player. This means our **didBeginContact()** method is easy:

```
func didBeginContact(contact: SKPhysicsContact) {
  if contact.bodyA.node == player {
    playerCollidedWithNode(contact.bodyB.node!)
  } else if contact.bodyB.node == player {
    playerCollidedWithNode(contact.bodyA.node!)
  }
}
```

There are three types of collision we care about: when the player hits a vortex they should be penalised, when the player hits a star they should score a point, and when the player hits the finish flag the next level should be loaded. I'll deal with the first two here, and you can think about the third one yourself!

When a player hits a vortex, a few things need to happen. Chief among them is that we need to stop the player controlling the ball, which will be done using a single boolean property called **gameOver**. Add this property now:

```
var gameOver = false
```

You'll need to modify your **update()** method so that it works only when **gameOver** is false, like this:

```
override func update(currentTime: CFTimeInterval) {
  if !gameOver {
```

```
#if (arch(i386) || arch(x86_64))

    if let currentTouch = lastTouchPosition {

        let diff = CGPoint(x: currentTouch.x - player.position.x, y:
currentTouch.y - player.position.y)

        physicsWorld.gravity = CGVector(dx: diff.x / 100, dy: diff.y /
100)

    }

#else

    if let accelerometerData = motionManager.accelerometerData {

        physicsWorld.gravity = CGVector(dx:
accelerometerData.acceleration.y * -50, dy:
accelerometerData.acceleration.x * 50)

    }

#endif

  }

}
```

Of course, a number of other things need to be done when a player collides with a vortex:

• We need to stop the ball from being a dynamic physics body so that it stops moving once it's sucked in.
• We need to move the ball over the vortex, to simulate it being sucked in. It will also be scaled down at the same time.
• Once the move and scale has completed, we need to remove the ball from the game.
• After all the actions complete, we need to create the player ball again and re-enable control.

We'll put that together using an **SKAction** sequence, followed by a trailing closure that will execute when the actions finish. When colliding with a star, we just remove the star node from the scene and add one to the score.

```
func playerCollidedWithNode(node: SKNode) {
  if node.name == "vortex" {

    player.physicsBody!.dynamic = false
```

```
        gameOver = true

        score -= 1


        let move = SKAction.moveTo(node.position, duration: 0.25)

        let scale = SKAction.scaleTo(0.0001, duration: 0.25)

        let remove = SKAction.removeFromParent()

        let sequence = SKAction.sequence([move, scale, remove])


        player.runAction(sequence) { [unowned self] in

            self.createPlayer()

            self.gameOver = false

        }

    } else if node.name == "star" {

        node.removeFromParent()

        score += 1

    } else if node.name == "finish" {

        // next level?

    }

}
```

That method finishes the game, so it's down to you now to try and play the whole level without falling into a vortex. What happens when you hit the finish flag? Nothing… yet.

Wrap up

There's something wonderfully tactile about using the accelerometer to affect gravity in a game, because it feels incredibly realistic even though we're not using particularly good graphics. SpriteKit is of course doing most of the hard work of collision detection, and Core Motion takes away all the complexity of working with accelerometers, so again it's our job to sew the components together to make something bigger than the sum of its parts.

There are two things you should immediately tackle if you want to continue working on this project. First, have a go at refactoring the **loadLevel()** method so that it's made up of multiple smaller methods. This will make your code easier to read and easier to maintain, at least it will do if you do a good job!

Second, when the player finally makes it to the finish marker, nothing happens. What should happen? Well, that's down to you now. You could easily design several new levels and have them progress through, but could you add things that make the new levels different – perhaps a teleport that moves the player from one point in the level to another? Add a new letter type in **loadLevel()**, add another collision type to our enum, then see what you can do. Have fun!

Project 27
Core Graphics

Draw 2D shapes using Apple's high-speed drawing framework.

Setting up

You're probably tired of me saying this: iOS is full of powerful and easy to use programming frameworks. It's true, and you've already met UIKit, SpriteKit, Core Animation, Core Motion, Core Image, Core Location, Grand Central Dispatch and more. But how would you feel if I said that we've yet to use one of the biggest, most powerful and most important frameworks of all?

Well, it's true. And in this technique project we're going to right that wrong. The framework is called Core Graphics, and it's responsible for device-independent 2D drawing – when you want to draw shapes, paths, shadows, colors or so on, you'll want Core Graphics. Being device-independent means you can draw things to the screen or draw them a PDF without having to change your code.

Create a new Single View Application project, name it Project27 and set its target to be iPad. We're going to create a Core Graphics sandbox that's similar to project 15's Core Animation sandbox – a button you can type will trigger Core Graphics drawing in different ways.

If you haven't already downloaded the files for this project, please do so now from **GitHub**, then copy the mouse picture into your project.

Creating the sandbox

Open Main.storyboard in Interface Builder, then place an image view in there that's 512x512 in size, with X:44 and Y:44. Now place a button centered beneath it, and give it the title "Redraw".

For constraints, please place a width and height constraint on the image view so that it's always 512x512, then make it center vertically and horizontally inside the view controller. The button should be centered horizontally, but place a vertical space constraint between it and the image view so that it always stays just below it.

We're going to need to reference the image view in code, so please switch to the assistant editor and create an outlet for it called **imageView**. While you're there, you should also create an action for the button being tapped, called **redrawTapped()**.

Switch back to the standard editor, and open up ViewController.swift because that's our user interface done; the rest is code!
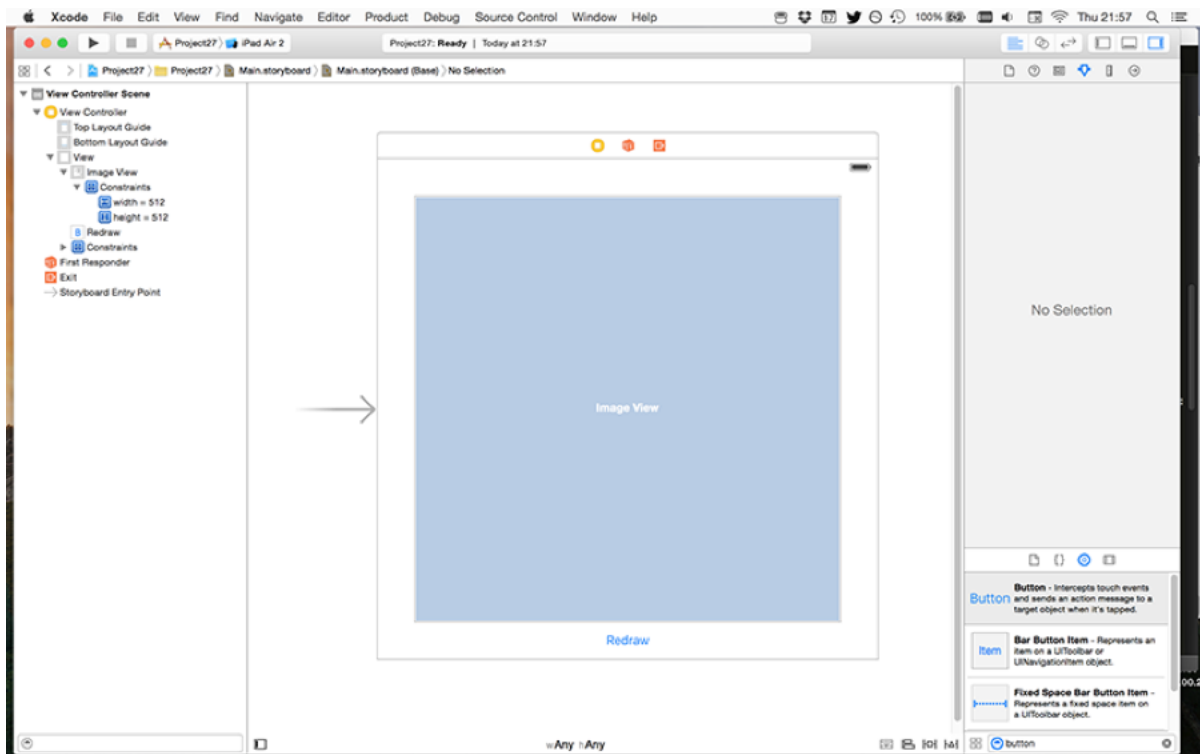
Here's our basic layout: a big UIImageView with a UIButton underneath.

We're going to use a similar code structure to project 15: a property that we increment

through, using a switch/case to call different methods each time. In addition, we need **viewDidLoad()** to call an initial method so that we start by drawing something. Start by adding this property to your view controller:

```
var currentDrawType = 0
```

And now create this empty method – we'll be filling it shortly:

```
func drawRectangle() {


}
```

As with project 15, the button we placed needs to add one to the property, wrapping it back to zero when it reaches a certain point. The property is then used to decide what method to call, although right now we're only going to have one useful case: **drawRectangle()**.

Here's the initial code for **redrawTapped()**; we'll be adding more cases over time:

```
@IBAction func redrawTapped(sender: AnyObject) {
  currentDrawType += 1

  if currentDrawType > 5 {
    currentDrawType = 0
  }

  switch currentDrawType {
  case 0:
    drawRectangle()

  default:
```

```
        break

    }

}
```

The only remaining step to make our sandbox complete is to have **viewDidLoad()** call the **drawRectangle()** method so that the screen starts by showing something. Change your **viewDidLoad()** method to this:

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

  drawRectangle()
}
```

Running the app at this point will do very little, because although your user interface works the code effectively does nothing. We're going to fix that by filling in the **drawRectangle()** method, then proceed to add more cases to the switch/case block

Drawing into a context

Carl Sagan once said, "if you wish to make an apple pie from scratch, you must first invent the universe." Filling in the **drawRectangle()** method doesn't require you to invent the universe, but it *does* require a fair amount of Core Graphics learning before you get to the actual drawing part. I will, of course, try to remove as much of it as I can so that the remaining bits are important.

The most important thing to understand is that, like Core Animation, Core Graphics sits at a lower technical level than UIKit. This means it doesn't understand classes you know like **UIColor** and **UIBezierPath**, so you either need to use their counterparts (**CGColor** and **CGPath** respectively), or use helper methods from UIKit that convert between the two.

To be fair to Apple, they have written some great helper code that seamlessly blends UIKit and Core Graphics, so a lot of the time you don't need to worry about it. Take for example creating a 256x256 image. Using **UIImage** you don't need to care whether it's retina or not, but when you're creating a Core Graphics context you need to tell it whether you want 1x, 2x or 3x scale. Helpfully, Apple also lets you specify 0 for scale, to mean "use whatever is right for the current device."

Another important thing to understand about Core Graphics is that it's written in C, not in Objective C. That doesn't mean *you* need to code in C any more than using SpriteKit means you need to write in Objective C, but you'll find it uses different data types from what you're used to. Being a C framework also means that everything is a function; there aren't any methods around here.

Thirdly, you need to understand that Core Graphics differentiates between creating a path and drawing a path. That is, you can add lines, squares and other shapes to a path as much as you want to, but none of it will do anything until you actually draw the path. Think of it like a simple state machine: you configure a set of states you want (colors, transforms, and so on), then perform actions. You can even maintain multiple states at a time by pushing and popping in order to backup and restore specific states.

Finally, you should know that Core Graphics is extremely fast: you can use it for updating drawing in real time, and you'll be very impressed. Core Graphics can work on a background thread – something that UIKit can't do – which means you can do complicated drawing without locking up your user interface.

It's time to start looking at some code, so here are the four functions you'll be using lots of

times in this project:

1. **UIGraphicsBeginImageContextWithOptions()** creates a new Core Graphics drawing context.
2. **UIGraphicsGetCurrentContext()** returns a reference to the Core Graphics context we just created.
3. **UIGraphicsGetImageFromCurrentImageContext()** converts the Core Graphics context to a **UIImage**.
4. **UIGraphicsEndImageContext()** terminates our Core Graphics context.

Now, wait a minute… all those functions start with "UI", so what makes them Core Graphics functions? Well, they *aren't* Core Graphics functions – they are UIKit functions, but they act as a gateway to and from Core Graphics for UIKit-based apps like ours. The first two are used to set up drawing and the last two are used to tear down drawing; everything between will be Core Graphics functions or UIKit methods that are designed to work with Core Graphics contexts.

Speaking of contexts, you'll notice that all four of those functions have the word "context" in there. In Core Graphics, a context is a canvas upon which we can draw, but it also stores information about how we want to draw (e.g., what should our line thickness be?) and information about the device we are drawing to. So, it's a combination of canvas and metadata all in one, and it's what you'll be using for all your drawing.

When you create a context, you get to specify how big it should be, whether it should be opaque or not, and what pixel to point scale you want. You'll then pull out a reference to the current context using **UIGraphicsGetCurrentContext()**, and do your drawing. When you're finished, you should use **UIGraphicsGetImageFromCurrentImageContext()** to get the contents of your drawing as a **UIImage**, then end drawing by calling **UIGraphicsEndImageContext()**.

Armed with this knowledge, you can write the first version of **drawRectangle()**:

```
func drawRectangle() {
    UIGraphicsBeginImageContextWithOptions(CGSize(width: 512, height:
512), false, 0)

    let context = UIGraphicsGetCurrentContext()
```

```
    // awesome drawing code



    let img = UIGraphicsGetImageFromCurrentImageContext()

    UIGraphicsEndImageContext()



    imageView.image = img

}
```

In that code, we use **UIGraphicsBeginImageContextWithOptions()** to create a context with the same size as our image view, make it not opaque by passing **false** as its second parameter, and tell it to match the scale of whatever screen we're on. So on non-retina iPads, it'll be 512x512 pixels and on retina iPads it will be 1024x1024 pixels.

Creating the context doesn't give us a reference to it, which is what the second line of the method does – we store our reference in a constant called **context**. Finally, drawing is ended by pulling out a **UIImage** then ending the Core Graphics context, at which point we can assign our new image to the image view.

That code still does nothing, at least not visibly. This is because it does all the setup and tear down of Core Graphics, but doesn't do any actual drawing. Let's fix that by making the **drawRectangle()** method actually draw a rectangle. And not just *any* rectangle – a **stroked** rectangle, which is a rectangle with a line around it.

There are a number of ways of drawing boxes in Core Graphics, but I've chosen the easiest: we'll define a **CGRect** structure that holds the bounds of our rectangle, we'll set the context's fill color to be red and its stroke color to be black, we'll set the context's line drawing width to be 10 points, then add a rectangle path to the context and draw it.

The part that might seem strange is the way we're adding a path then drawing it. This is because you can actually add multiple paths to your context before drawing, which means Core Graphics batches them all together. Your path can be as simple or as complicated as you want, you still need to set up your Core Graphics state as you want it then draw the path.

Let's take a look at the five new functions you'll need to use to draw our box:

1. **CGContextSetFillColorWithColor()** sets the fill color of our context, which is the color used on the insides of the rectangle we'll draw. It also sounds like a typo, after all what else what you want to use for a fill color other than a color? The reason for the curious name is because there's an older function called **CGContextSetFillColor()** that let you set colors in a rather arcane way. It's not recommended any more!

2. **CGContextSetStrokeColorWithColor()** sets the stroke color of our context, which is the color used on the line around the edge of the rectangle we'll draw. Yes, it's named strangely for the same reason.

3. **CGContextSetLineWidth()** adjusts the line width that will be used to stroke our rectangle. Note that the line is drawn centered on the edge of the rectangle, so a value of 10 will draw 5 points inside the rectangle and five points outside.

4. **CGContextAddRect()** adds a **CGRect** rectangle to the context's current path to be drawn.

5. **CGContextDrawPath()** draws the context's current path using the state you have configured.

All five of those take a Core Graphics context was their first parameter, then a second parameter that does the actual work. So for setting colors the second parameter is the color to set (remember how to convert **UIColor** values to **CGColor** values? I hope so!), for setting the line width it's a number in points, for adding a rectangle path it's the **CGRect** of your rectangle, and for drawing it's a special constant that says whether you want just the fill, just the stroke, or both.

Time for some code: replace **// awesome drawing code** with this:

```
let rectangle = CGRect(x: 0, y: 0, width: 512, height: 512)


CGContextSetFillColorWithColor(context, UIColor.redColor().CGColor)

CGContextSetStrokeColorWithColor(context,
UIColor.blackColor().CGColor)

CGContextSetLineWidth(context, 10)


CGContextAddRect(context, rectangle)

CGContextDrawPath(context, .FillStroke)
```

At long last, this project does something useful: when you run it, you'll see a red box with a black line around it. Again, the black line will be just five points across, because it's centered on the edge of its path and therefore is cropped. You'll see this more clearly in a moment.

Ellipses and checkerboards

Add a case to your **redrawTapped()** method to call a new method: **drawCircle()**. This will… wait for it… *draw a circle*. So, your switch/case should look like this:

```
switch currentDrawType {
case 0:
    drawRectangle()

case 1:
    drawCircle()

default:
    break
}
```

There are several ways of drawing rectangles using Core Graphics, but the method we used in **drawRectangle()** is particularly useful because in order to draw a circle we need to change just one line of code. This is because drawing circles (or indeed any elliptical shape) in Core Graphics is done by specifying its rectangular bounds.

So, where before you had:

```
CGContextAddRect(context, rectangle)
```

…you can now use this:

```
CGContextAddEllipseInRect(context, rectangle)
```

It even has the same parameters! So, the full **drawCircle()** method is this:

```
func drawCircle() {

    UIGraphicsBeginImageContextWithOptions(CGSize(width: 512, height:
512), false, 0)

    let context = UIGraphicsGetCurrentContext()

    let rectangle = CGRect(x: 0, y: 0, width: 512, height: 512)


    CGContextSetFillColorWithColor(context, UIColor.redColor().CGColor)

    CGContextSetStrokeColorWithColor(context,
UIColor.blackColor().CGColor)

    CGContextSetLineWidth(context, 10)


    CGContextAddEllipseInRect(context, rectangle)

    CGContextDrawPath(context, .FillStroke)


    let img = UIGraphicsGetImageFromCurrentImageContext()

    UIGraphicsEndImageContext()


    imageView.image = img

}
```

Run the app now and click the button once to make it draw a circle. Notice how the stroke around the edge appears to be clipped at the top, right, bottom and left edges? This is a direct result of what I was saying about stroke positioning: the stroke is centered on the edge of the path, meaning that a 10 point stroke is 5 points inside the path and 5 points outside.
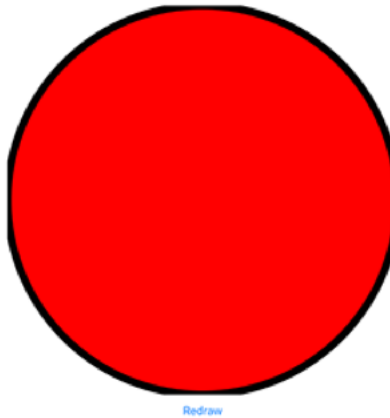
Redraw

Now that we're drawing a circle, you can see how Core Graphics is clipping our border at the edges.

The rectangle being used to define our circle is the same size as the whole context, meaning that it goes edge to edge – and thus the stroke gets clipped. To fix the problem, change the rectangle to this:

```
let rectangle = CGRect(x: 5, y: 5, width: 502, height: 502)
```

That indents the circle by 5 points on all sides, so the stroke will now look uniform around the entire shape.

A different way of drawing rectangles is just to fill them directly with your target color. Add a "case 2" to your switch/case that calls a method named **drawCheckerboard()**, and give it this code:

```
func drawCheckerboard() {

  UIGraphicsBeginImageContextWithOptions(CGSize(width: 512, height:
512), false, 0)

  let context = UIGraphicsGetCurrentContext()


  CGContextSetFillColorWithColor(context,
UIColor.blackColor().CGColor)


  for row in 0 ..< 8 {

    for col in 0 ..< 8 {

      if row % 2 == 0 {

        if col % 2 == 0 {

          CGContextFillRect(context, CGRect(x: col * 64, y: row * 64,
width: 64, height: 64))

        }

      } else {

        if col % 2 == 1 {

          CGContextFillRect(context, CGRect(x: col * 64, y: row * 64,
width: 64, height: 64))

        }

      }


    }

  }


  let img = UIGraphicsGetImageFromCurrentImageContext()

  UIGraphicsEndImageContext()


  imageView.image = img

}
```

The only piece of code in there that you won't recognise is **CGContextFillRect()**, which skips the add path / draw path work and just fills the rectangle given as its second parameter using whatever the current fill color is. You already know about ranges and modulo, so you should be able to see that this method makes every other square black, alternating between rows.

There are two things to be aware of with that code. First, we're filling every other square in black, but leaving the other squares alone. As we've specified that our image is not opaque, this means those places where we haven't filled anything will be transparent.So, if the view behind was green, you'd get a black and green checkerboard. Second, you can actually make checkerboards using a Core Image filter – check out **CICheckerboardGenerator** to see how!
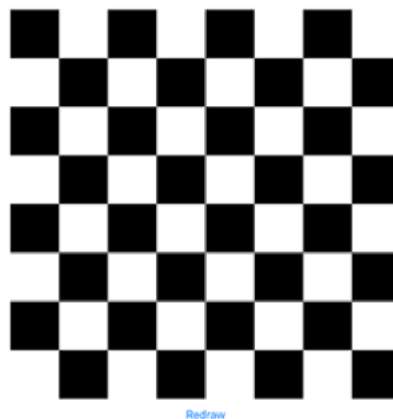
Drawing a checkerboard with Core Graphics is just a matter of drawing squares with alternating colors.

Transforms and lines

Add another case to your switch/case block, and make this one call another new method named **drawRotatedSquares()**. This is going to demonstrate how we can apply transforms to our context before drawing, and how you can stroke a path without filling it.

To make this happen, you need to know three new Core Graphics functions:

1. **CGContextTranslateCTM()** translates (moves) the current transformation matrix (CTM).
2. **CGContextRotateCTM()** rotates the current transformation matrix.
3. **CGContextStrokePath()** strokes the path with your specified line width, which is 1 if you don't set it explicitly.

The current transformation matrix (CTM) is very similar to those **CGAffineTransform** modifications we used in project 15, except its application is a little different in Core Graphics. In UIKit, you rotate drawing around the center of your view, as if a pin was stuck right through the middle. In Core Graphics, you rotate around the top-left corner, so to avoid that we're going to move the CTM half way into our image first so that we've effectively moved the rotation point.

This also means we need to draw our rotated squares so they are centered on our center: for example, setting their top and left coordinates to be -128 and their width and height to be 256.

Here's the code for the method:

```
func drawRotatedSquares() {
  UIGraphicsBeginImageContextWithOptions(CGSize(width: 512, height:
512), false, 0)

  let context = UIGraphicsGetCurrentContext()

  CGContextTranslateCTM(context, 256, 256)


  let rotations = 16

  let amount = M_PI_2 / Double(rotations)


  for _ in 0 ..< rotations {
```

```
        CGContextRotateCTM(context, CGFloat(amount))

        CGContextAddRect(context, CGRect(x: -128, y: -128, width: 256,
height: 256))

    }


    CGContextSetStrokeColorWithColor(context,
UIColor.blackColor().CGColor)

    CGContextStrokePath(context)


    let img = UIGraphicsGetImageFromCurrentImageContext()

    UIGraphicsEndImageContext()


    imageView.image = img

}
```

Run the app and look at the output: beautiful, rotated, stroked squares, with no extra calculations required. I mean, just stop for a moment and consider the math it would take to calculate the four corners of each of those rectangles. If sine and cosine are distance memories for you, be glad to have the current transformation matrix!

One thing that I should make clear: modifying the CTM is cumulative, which is what makes the above code work. That is, when you rotate the CTM, that transformation is applied on top of what was there already, rather than to a clean slate. So the code works by rotating the CTM a small amount more every time the loop goes around.

The last shape drawing I want to show you is how to draw lines, and you're going to need two new functions: **CGContextMoveToPoint()** and **CGContextAddLineToPoint()**. These are the Core Graphics equivalents to the **UIBezierPath** paths we made in project 20 to move the fireworks.

Add another case to your switch/case block, this time calling **drawLines()**. I'm going to make this translate and rotate the CTM again, although this time the rotation will always be 90 degrees. This method is going to draw boxes inside boxes, always getting smaller, like a square spiral. It's going to do this by adding a line to more or less the same point inside a loop, but each time the loop rotates the CTM so the actual point the line ends has moved

too. It will also slowly decrease the line length, causing the space between boxes to shrink like a spiral. Here's the code:

```
func drawLines() {
  UIGraphicsBeginImageContextWithOptions(CGSize(width: 512, height:
512), false, 0)

  let context = UIGraphicsGetCurrentContext()

  CGContextTranslateCTM(context, 256, 256)


  var first = true

  var length: CGFloat = 256


  for _ in 0 ..< 256 {

    CGContextRotateCTM(context, CGFloat(M_PI_2))


    if first {

      CGContextMoveToPoint(context, length, 50)

      first = false

    } else {

      CGContextAddLineToPoint(context, length, 50)

    }


    length *= 0.99

  }


  CGContextSetStrokeColorWithColor(context,
UIColor.blackColor().CGColor)

  CGContextStrokePath(context)


  let img = UIGraphicsGetImageFromCurrentImageContext()

  UIGraphicsEndImageContext()
```

```
    imageView.image = img
}
```

The end result looks like one of the hand-crafted effects from the Twilight Zone, but it shows a little of the power that transforms and lines can bring to your drawing.

Images and text

Add one final case to your switch/case statement calling a method **drawImagesAndText()**, because no discussion of Core Graphics would be useful without telling you how to draw images and text to your context.

If you have a string in Swift, how can you place it into a graphic? The answer is simpler than you think: all strings have a built-in method called **drawWithRect()** that draws the string in a rectangle you specify. Even better, you get to customise the font and size, the formatting, the line wrapping and more all with that one method.

Remarkably, the same is true of **UIImage**: any image can be drawn straight to a context, and it will even take into account the coordinate reversal of Core Graphics.

Before you're able to draw a string to the screen, you need to meet two more classes: **UIFont** and **NSMutableParagraphStyle()**. The former defines a font name and size, e.g. Helvetica Neue size 26, and the latter is used to describe paragraph formatting options, such as line height, indents and alignment.

When you draw a string to the screen, you do using a dictionary of attributes that describes all the options you want to apply. We want to apply a custom font and custom paragraph style – that bit is easy enough. But the *keys* for the dictionary are special Apple constants: **NSFontAttributeName** and **NSParagraphStyleAttributeName**.

To help make the code clearer, here's a bulleted list of all the things the method needs to do:

1. Create a context, but this time we don't need a reference to it.
2. Define a paragraph style that aligns text to the center.
3. Create an attributes dictionary containing that paragraph style, and also a font.
4. Draw a string to the screen using the attributes dictionary.
5. Load an image from the project and draw it to the context.
6. Retrieve a **UIImage** of the context's data, the end drawing.
7. Update the image view with the finished result.

Below is that the same process, now coded in Swift. As per usual, the number comments match the list above:

```
func drawImagesAndText() {
```

```swift
    // 1
    UIGraphicsBeginImageContextWithOptions(CGSize(width: 512, height:
512), false, 0)

    let context = UIGraphicsGetCurrentContext()


    // 2
    let paragraphStyle = NSMutableParagraphStyle()

    paragraphStyle.alignment = .Center


    // 3
    let attrs = [NSFontAttributeName: UIFont(name: "HelveticaNeue-
Thin", size: 36)!, NSParagraphStyleAttributeName: paragraphStyle]


    // 4
    let string = "The best-laid schemes o'\nmice an' men gang aft
agley"

    string.drawWithRect(CGRect(x: 32, y: 32, width: 448, height: 448),
options: .UsesLineFragmentOrigin, attributes: attrs, context: nil)


    // 5
    let mouse = UIImage(named: "mouse")

    mouse?.drawAtPoint(CGPoint(x: 300, y: 150))


    // 6
    let img = UIGraphicsGetImageFromCurrentImageContext()

    UIGraphicsEndImageContext()


    // 7
    imageView.image = img
}
```

That completes our project. If you found Xcode's code completion wasn't filling in **drawWithRect()** for you, try giving the string an explicit type of **NSString**, like this:

```
let string: NSString = "The best-laid schemes o'\nmice an' men gang
aft agley"
```

The code works regardless, but having code completion around does help reduce mistakes!

Strings and UIImages have built-in methods that let you draw them to a Core Graphics context.

Wrap up

I could easily have written twice as much about Core Graphics, because it's capable of some extraordinary effects.Clipping paths, gradients, blend modes and more are just a few lines of code away, so there really is no excuse not to give them a try! And if you don't give it a try yourself, don't worry: we'll be drawing with Core Graphics in project 29, so you can't avoid it!

This project has given you a sandbox where you can play around with various Core Graphics techniques easily, so I would highly encourage you to spend another hour or two tinkering with the code in your project. Use code completion to try new functions, change my values to others to see what happens, and so on. Playing with code like this can help you to discover new functionality, and will also help you remember more later. Have fun!

Project 28

Secret Swift

Save user data securely using the device keychain and
Touch ID.

Setting up

This project will introduce you to two important iOS technologies together: Touch ID and the keychain. The former is used to identify users biometrically using the fingerprint sensor on iPhones and iPads; the latter is a secure, encrypted data storage area on every device that you can read and write to.

Of course, there's little point learning about technologies without using them, so this project will have you build a secure text editor. Users can type whatever they want and have it saved, but to read it again they'll need to unlock the app using Touch ID.

You might remember in project 12 that I said **NSUserDefaults** is great for its simplicity but isn't good for private data. Well, the keychain is securely encrypted, so we can be assured that data we put there is safe.

This project is modelled on project 16, the Safari extension where users could type JavaScript. This means we'll need to use the same keyboard detection and adjustment code – if you already completed project 16, you might find it easiest to copy and paste your code as needed.

Make a new Single View Application project in Xcode, named Project28 and targeted at any device you want.

The basic text editor

Open Main.storyboard in Interface Builder, and embed the default view controller inside a navigation controller. Now place a text view (*not* a text field!) in the center and use Editor > Resolve Auto Layout Issues > Add Missing Constraints to place Auto Layout constraints.

Delete the "Lorem ipsum" text in the text view, give it a white background color if it does not already have one, then use the assistant editor to make an outlet for it called **secret**. That's us done with the storyboard for now; switch back to the standard editor and open ViewController.swift.

We need to add the same code we used in project 16 to make the text view adjust its content and scroll insets when the keyboard appears and disappears. This code is identical apart from the outlet is called **secret** now rather than **script**. First, put this into **viewDidLoad()**:

```
let notificationCenter = NSNotificationCenter.defaultCenter()

notificationCenter.addObserver(self, selector: "adjustForKeyboard:",
name: UIKeyboardWillHideNotification, object: nil)

notificationCenter.addObserver(self, selector: "adjustForKeyboard:",
name: UIKeyboardWillChangeFrameNotification, object: nil)
```

As a reminder, that asks iOS to tell us when the keyboard changes or when it hides. As a double reminder: the hide is required because we do a little hack to make the hardware keyboard toggle work correctly – see project 16 if you don't remember why this is needed.

Here's the **adjustKeyboard()** method, which again is identical apart from the outlet name to that seen in project 16:

```
func adjustForKeyboard(notification: NSNotification) {

  let userInfo = notification.userInfo!


  let keyboardScreenEndFrame =
(userInfo[UIKeyboardFrameEndUserInfoKey] as! NSValue).CGRectValue()
  let keyboardViewEndFrame = view.convertRect(keyboardScreenEndFrame,
```

```
    fromView: view.window)


    if notification.name == UIKeyboardWillHideNotification {

        secret.contentInset = UIEdgeInsetsZero

    } else {

        secret.contentInset = UIEdgeInsets(top: 0, left: 0, bottom:
keyboardViewEndFrame.height, right: 0)

    }


    secret.scrollIndicatorInsets = secret.contentInset


    let selectedRange = secret.selectedRange

    secret.scrollRangeToVisible(selectedRange)

}
```

None of that is new, so you're probably bored by now. Not to worry: we're going to fix up our storyboard in preparation for authentication, so re-open Main.storyboard.

Place a button over the text view, give it the title "Authenticate" and dimensions 100 wide by 44 high. For constraints, give it fixed width and height constraints, then make it align horizontally and vertically with its superview. Now use the assistant editor to create an action for it called **authenticateUser()**.

Before you leave Interface Builder, you need to do something we haven't done yet, which is to move views backwards and forwards relative to each other. When the user has authenticated, we need to show the text box while making sure the button is no longer visible, and the easiest way to do that is to place the button behind the text view so that when the text is visible it covers up the button.

To move layers around, go to the document outline then click and drag. In our case, we want to move the text view behind the authenticate button, so you should click on the button drag it so that it's just above the text view. When you do this the button will disappear on the canvas, but that's OK – it's still there, and we can still use it.

The last thing to do is ensure the text view starts life hidden, so select it in Interface Builder, choose the attributes inspector, and check the Hidden box – it's near the bottom, not far below Tag. That's our layout complete!

Writing somewhere safe

When the app first runs, users should see a totally innocuous screen, with nothing secret visible. But we also don't want secret information to be visible when the user leaves the app for a moment then comes back, or if they double-tap the home button to multitask – doing so might mean that the app is left unlocked, which is the last thing we want.

To make this work, let's start by giving the view controller a totally innocuous title that absolutely won't make anyone wonder what's going on. Put this into **viewDidLoad()**:

```
title = "Nothing to see here"
```

Next we're going to create two new methods: **unlockSecretMessage()** to load the message into the text view, and **saveSecretMessage()**. But before we do that, I want to introduce you to a helpful class called **KeychainWrapper**, which we'll be using to read and write keychain values.

This class was not made by Apple; instead, it's open source software released under the MIT license, which means we can use it in our own projects as long as the copyright message remains intact. This class is needed because working with the keychain is *complicated* – far harder than anything we have done so far. So instead of using it directly, we'll be using this wrapper class that makes the keychain work like **NSUserDefaults**.

If you haven't already downloaded this project's files from **GitHub**, please do so now. In there you'll find the file KeychainWrapper.swift; please copy that into your Xcode project to make the class available.

The first of our two new methods, **unlockSecretMessage()**, needs to show the text view, then load the keychain's text into it. Loading strings from the keychain using **KeychainWrapper** is as simple as using its **stringForKey()** method, but the result is optional so you should unwrap it once you know there's a value there.

Here it is:

```
func unlockSecretMessage() {
```

```
    secret.hidden = false

    title = "Secret stuff!"


    if let text = KeychainWrapper.stringForKey("SecretMessage") {

        secret.text = text

    }

}
```

The second of our two new methods, **saveSecretMessage()**, needs to write the text view's text to the keychain, then make it hidden. This is done using the **setString(_:forKey:)** method of **KeychainWrapper**, so it's just as easy as reading. Note that we should only execute this code if the text view is visible, otherwise if a save happens before the app is unlocked then it will overwrite the saved text!

Here's the code:

```
func saveSecretMessage() {

    if !secret.hidden {

        KeychainWrapper.setString(secret.text, forKey: "SecretMessage")

        secret.resignFirstResponder()

        secret.hidden = true

        title = "Nothing to see here"

    }

}
```

I slipped a new method in there: **resignFirstResponder()**. This is used to tell a view that has input focus that it should give up that focus. Or, in Plain English, to tell our text view that we're finished editing it, so the keyboard can be hidden. This is important because having a keyboard present might arouse suspicion – as if our rather obvious navigation bar title hadn't done enough already…

Now, there are still two questions remaining: how should users trigger a save when they are

ready, and how do we ensure that as soon as the user starts to leave the app we make their data safe? For the first problem, consider this: how often do you see a save button in iOS? Hardly ever, I expect!

It turns out that one answer solves both problems: if we automatically save when the user leaves the app then the user need never worry about saving because it's done for them, and our save method above automatically hides the text when it's called so the app becomes safe as soon as any action is taken to leave it.

We're already using **NSNotificationCenter** to watch for the keyboard appearing and disappearing, and we can watch for another notification that will tell us when the application will stop being active – i.e., when our app has been backgrounded or the user has switched to multitasking mode. This notification is called **UIApplicationWillResignActiveNotification**, and you should make us an observer for it in **viewDidLoad()** like this:

```
notificationCenter.addObserver(self, selector: "saveSecretMessage",
name: UIApplicationWillResignActiveNotification, object: nil)
```

That calls our **saveSecretMessage()** directly when the notification comes in, which means the app automatically saves any text and hides it when the app is moving to a background state.

The last thing to do before the app is actually useful is to make tapping the button call **unlockSecretMessage()**, like this:

```
@IBAction func authenticateUser(sender: AnyObject) {

  unlockSecretMessage()

}
```

It's not actually secure at this point (other than saving its data in the iOS keychain!), but by saving and loading its text it is at least useful.

Touch to activate

This part of the project requires a Touch ID-capable device, although I'll be showing you a hack to make it work on the simulator by bypassing checks at two points.

Touch ID is part of the Local Authentication framework, and our code needs to three things:

1. Check whether the device is capable of supporting biometric authentication.
2. If so, request that the Touch ID begin a check now, giving it a string containing the reason why we're asking.
3. If we get success back from Touch ID it means this is the device's owner so we can unlock the app, otherwise we show a failure message.

One caveat that you must be careful of: when we're told whether Touch ID was successful or not, it might not be on the main thread. This means we need to use **dispatch_async()** to make sure we execute any user interface code on the main thread.

The job of task 1 is done by the **canEvaluatePolicy()** method of the **LAContext** class, requesting the security policy type **.DeviceOwnerAuthenticationWithBiometrics**. The job of task 2 is done by the **evaluatePolicy()** of that same class, using the same policy type, but it accepts a trailing closure telling us the result of the policy evaluation: was it successful, and if not what was the reason?

Like I said, all this is provided by the Local Authentication framework, so the first thing we need to do is import that framework. Add this above **import UIKit**:

```
import LocalAuthentication
```

And now here's the new code for the **authenticateUser()** method. We already walked through what it does, so this shouldn't be too surprising:

```
@IBAction func authenticateUser(sender: AnyObject) {
  let context = LAContext()
  var error: NSError?
```

```
  if
context.canEvaluatePolicy(.DeviceOwnerAuthenticationWithBiometrics,
error: &error) {

    let reason = "Identify yourself!"


    context.evaluatePolicy(.DeviceOwnerAuthenticationWithBiometrics,
localizedReason: reason) {

      [unowned self] (success: Bool, authenticationError: NSError?) -
> Void in


      dispatch_async(dispatch_get_main_queue()) {

        if success {

          self.unlockSecretMessage()

        } else {

          // error

        }

      }

    }

  } else {

    // no Touch ID

  }

}
```

A couple of reminders: we need **[unowned self]** inside the first closure but not the second because it's already unowned by that point. You also need to use **self.** inside the closure to make capturing clear. Finally, you must provide a reason why you want Touch ID to be used, so you might want to replace mine ("Identify yourself!") with something a little more descriptive.

That's enough to get basic Touch ID working, but there are three error cases you need to catch:

1. The device does not have Touch ID capability or configured. This is true for all iPads

before iPad Air 2 and iPad Mini 3, and all iPhones before the iPhone 5s.
2. The user failed Touch ID authentication. This might be because their print wasn't scanning for whatever reason, but also if the system has to cancel scanning for some reason.
3. The user requested to enter their password instead. Touch ID includes this option on the screen as a fallback for users who are unable to use fingerprint scanning at this time.

The first two are a matter of showing alerts to the user, but the third requires a specific check: if the user requested fallback mode (i.e. password) then what Touch ID really wants us to do is have the user enter a password. But that's not the user's iTunes password or even a PIN on their phone. Instead, it's a password that we need to have set up beforehand ourselves, then validate too.

That's a little outside the remit of this project, so if a user asks to use a password instead we'll deny them with a specific error message. Replace the **// error** with this:

```
if let error = authenticationError {

  if error.code == LAError.UserFallback.rawValue {

    let ac = UIAlertController(title: "Passcode? Ha!", message: "It's Touch ID or nothing – sorry!", preferredStyle: .Alert)

    ac.addAction(UIAlertAction(title: "OK", style: .Default, handler: nil))

    self.presentViewController(ac, animated: true, completion: nil)

    return

  }

}


let ac = UIAlertController(title: "Authentication failed", message: "Your fingerprint could not be verified; please try again.", preferredStyle: .Alert)

ac.addAction(UIAlertAction(title: "OK", style: .Default, handler: nil))

self.presentViewController(ac, animated: true, completion: nil)
```

That code unwraps the optional error we received back Touch ID and checks to see whether

it equals **LAError.UserFallback.rawValue**, which is the number behind the "fallback" enum. If so, it means the user asked to enter a password, in which case we send them a derisory message and exit the method.

If the code continues to execute, it means the error wasn't a fallback request, so we show a generic error message to the user.

Lastly, we need to show an error if Touch ID just isn't available, so replace the **// no Touch ID** comment with this:

```
let ac = UIAlertController(title: "Touch ID not available", message:
"Your device is not configured for Touch ID.",
preferredStyle: .Alert)

ac.addAction(UIAlertAction(title: "OK", style: .Default, handler:
nil))

self.presentViewController(ac, animated: true, completion: nil)
```

That completes the Touch ID code, but before we're done I want to add one more thing before the project is complete: if you don't have a Touch ID-capable device and you want to make sure the code works OK, you'll need to change a couple of lines of code. Specifically, you want to change these two lines:

```
if
context.canEvaluatePolicy(.DeviceOwnerAuthenticationWithBiometrics,
error: &error) {


if success {
```

…to read this both times:

```
if true {
```

That condition will always evaluate to true, so it will allow you to test the app inside the simulator.

Wrap up

This was the last app project in the series, and I hope you didn't find it too challenging – you know a huge amount about iOS 8 and Swift now, so much of what remains is just practice. The great thing about Touch ID is that you don't get any access to fingerprints or other secure information. Instead, the system does all the biometric authentication for you, which keeps both your app and users safe.

If you want to take this project further, try using the **#if** compiler directives from project 26 to make the simulator **if true** hack be in the code at the same time as the real Touch ID code. If you're looking for something harder, try creating a password system for your app so that the Touch ID fallback is more useful. You'll need to use the **addTextFieldWithConfigurationHandler()** from project 5, and I suggest you save the password in the keychain!

Project 29
Exploding Monkeys

Remake a classic DOS game and learn about destructible terrain and scene transitions.

Setting up

This is the last game project in Hacking with Swift Level 1, so I wanted to make it special: you're going to build a clone of a game that's almost 25 years old. This game is hugely recognisable to people over the age of 30 because it was the classic way to waste time during computer classes at school.

The game? It's called Gorillas, and it first shipped with an old text-based operating system called MS-DOS 5.0 way back in 1991. Of course, we're going to re-make it using SpriteKit, but at the same time you're going to learn some new things: how to make colors from hues, texture atlases, scene transitions, mixing UIKit with SpriteKit, and destructible terrain. You'll also get a recap on the Core Graphics techniques from project 27.

"Destructible terrain" means "terrain that can be destroyed," which is a key part of Gorillas. If you've never played it before, you won't know that it pits two players against each other, both standing on high-rise buildings and both flinging exploding bananas at each other using physics. Realistic, right? Well, no, but it's certainly fun!

Make a new SpriteKit project named Project29 and targeted at iPad. Please make it use landscape orientation, then delete the existing spaceship image from Images.xcassets and the code from inside the **didMoveToView()** and **touchesBegan()** methods. You should download the files for this project from **GitHub**, but for now please only copy the file Helper.swift into your project.

Warning: When working with SpriteKit projects, I strongly recommend you use the iPad 2 simulator rather than iPad Air – you'll get much faster frame rates, making it much more suitable for testing.

Building the environment

We're going to start by making the game environment, which means building the night-time, high-rise skyscraper scene that forms the backdrop for the game. We're going to do most of this with an **SKSpriteNode** subclass for buildings that sets up physics, draws the building graphic, and ultimately handles the building being hit by stray bananas. Are you ready to flex your Core Graphics muscle a little?

Add a new file, choosing iOS > Source > Cocoa Touch Class, name it "BuildingNode" and make it a subclass of **SKSpriteNode**. Open the new file for editing, and add **import SpriteKit** just above the UIKit import.

Initially, this class needs to have three methods:

1. **setup()** will do the basic work required to make this thing a building: setting its name, texture, and physics.
2. **configurePhysics()** will set up per-pixel physics for the sprite's current texture.
3. **drawBuilding()** will do the Core Graphics rendering of a building, and return it as a **UIImage**.

In amongst those three points was one small thing that you may have missed: "the sprite's *current* texture." This tells you that the texture will change as bits get blown off by those exploding bananas. To make this work, we're going to keep a copy of the building's texture as a **UIImage** so that we can modify it later.

Add this code to your class – it's a property followed by two methods:

```
var currentImage: UIImage!

func setup() {
  name = "building"

  currentImage = drawBuilding(size)
  texture = SKTexture(image: currentImage)

  configurePhysics()
```

```
}

func configurePhysics() {
  physicsBody = SKPhysicsBody(texture: texture, size: size)
  physicsBody!.dynamic = false
  physicsBody!.categoryBitMask = CollisionTypes.Building.rawValue
  physicsBody!.contactTestBitMask = CollisionTypes.Banana.rawValue
}
```

This is using the same "don't override the initializer" hack from project 14, because quite frankly if I wanted to explain to you how and why Swift's initialization system worked I'd probably have to add another whole book to this series! Instead, we'll be creating the sprites as red-colored blocks of the right size, then drawing buildings into them.

As you can see in that code, it calls a **drawBuilding()** method that returns a **UIImage**, which then gets saved into the property and converted into a texture. It also calls **configurePhysics()** rather than putting the code straight into its method. Both of these two methods are separate because they will be called every time the building is hit, so we'll be using them in two different places.

That code uses two bitmasks that we haven't made yet. This is identical to project 26, except now we need only three categories: buildings, bananas and players. In the case of buildings, the only thing they'll collide with is a banana, which triggers our explosion. So, go back to GameScene.swift and add this enum just above the **GameScene** class definition:

```
enum CollisionTypes: UInt32 {
  case Banana = 1
  case Building = 2
  case Player = 4
}
```

That was the easy bit: you already know about bitmasks, per-pixel physics, textures and so

on. The next method is **drawBuilding()** and it's going to get harder because we're going to use Core Graphics. You *did* read project 27, right? If so, this will be a cinch.

This method needs to:

1. Create a new Core Graphics context the size of our building.
2. Fill it with a rectangle that's one of three colors.
3. Draw windows all over the building in one of two colors: there's either a light on (yellow) or not (gray).
4. Pull out the result as a **UIImage** and return it for use elsewhere.

There's nothing complicated in there, but just to keep you on your toes I'm going to introduce a new way to create colors: hue, saturation and brightness, or HSB. Using this method of creating colors you specify values between 0 and 1 to control how saturated a color is (from 0 = gray to 1 = pure color) and brightness (from 0 = black to 1 = maximum brightness), and 0 to 1 for hue.

"Hue" is a value from 0 to 1 also, but it represents a position on a color wheel, like using a color picker on your Mac. Hues 0 and 1 both represent red, with all other colors lying in between.

Now, programmers often look at HSB and think it's much clumsier than straight RGB, but there are reasons for both. The helpful thing about HSB is that if you keep the saturation and brightness constant, changing the hue value will cycle through all possible colors – it's an easy way to generate matching pastel colors, for example.

Here's the code for **drawBuilding()**, with numbered comments lining up to the list above:

```
func drawBuilding(size: CGSize) -> UIImage {
    // 1
    UIGraphicsBeginImageContextWithOptions(size, false, 0)
    let context = UIGraphicsGetCurrentContext()


    // 2
    let rectangle = CGRect(x: 0, y: 0, width: size.width, height:
size.height)
```

```swift
    var color: UIColor

    switch GKRandomSource.sharedRandom().nextIntWithUpperBound(3) {
    case 0:
        color = UIColor(hue: 0.502, saturation: 0.98, brightness: 0.67,
alpha: 1)
    case 1:
        color = UIColor(hue: 0.999, saturation: 0.99, brightness: 0.67,
alpha: 1)
    default:
        color = UIColor(hue: 0, saturation: 0, brightness: 0.67, alpha:
1)
    }

    CGContextSetFillColorWithColor(context, color.CGColor)
    CGContextAddRect(context, rectangle)
    CGContextDrawPath(context, .Fill)

    // 3
    let lightOnColor = UIColor(hue: 0.190, saturation: 0.67,
brightness: 0.99, alpha: 1)
    let lightOffColor = UIColor(hue: 0, saturation: 0, brightness:
0.34, alpha: 1)

    for var row: CGFloat = 10; row < size.height - 10; row += 40 {
        for var col: CGFloat = 10; col < size.width - 10; col += 40 {
            if RandomInt(min: 0, max: 1) == 0 {
                CGContextSetFillColorWithColor(context, lightOnColor.CGColor)
            } else {
                CGContextSetFillColorWithColor(context,
lightOffColor.CGColor)
            }
```

```
        CGContextFillRect(context, CGRect(x: col, y: row, width: 15,
height: 20))
    }

  }


  // 4

  let img = UIGraphicsGetImageFromCurrentImageContext()

  UIGraphicsEndImageContext()


  return img

}
```

The only thing new in there – and it's so tiny you probably didn't even notice – is my use of **.Fill** rather than **.Stroke** to draw the rectangles. Using what you learned in project 27, can you think of another way of doing this? (Hint: have a look at the way the windows are drawn!)

That's the **BuildingNode** class finished for now; we'll return to it later to add a method that will be called whenever it gets hit by a banana.

Go back to GameScene.swift because we have a small amount of work to do in order to use these new building nodes to build the night sky scene.

First, add a property that will store an array of buildings. We'll be using this to figure out where to place players later on:

```
var buildings = [BuildingNode]()
```

At this point, the **didMoveToView()** method needs to do only two things: give the scene a dark blue color to represent the night sky, then call a method called **createBuildings()** that will create the buildings. Here it is:

```
override func didMoveToView(view: SKView) {

  backgroundColor = UIColor(hue: 0.669, saturation: 0.99, brightness:
0.67, alpha: 1)


  createBuildings()

}
```

All those HSB values aren't an accident, by the way – I've chosen them so they look similar to the original design.

The **createBuildings()** method is the important one here, and calling it will finish our background scene. It needs to move horizontally across the screen, filling space with buildings of various sizes until it hits the far edge of the screen. I'm going to make it start at -15 rather than the left edge so that the buildings look like they keep on going past the screen's edge. I'm also going to leave a 2-point gap between the buildings to distinguish their edges slightly more.

Each building needs to be a random size. For the height, it can be anything between 300 and 600 points high; for the width, I want to make sure it divides evenly into 40 so that our window-drawing code is simple, so we'll generate a random number between 2 and 4 then multiply that by 40 to give us buildings that are 80, 120 or 160 points wide.

As I said earlier, we'll be creating each building node with a solid red color to begin with, then drawing over it with the building texture once it's generated. Remember: SpriteKit positions nodes based on their center, so we need to do a little division of width and height to place these buildings correctly. Here's the **createBuildings()** method – please put this directly beneath **didMoveToView()**:

```
func createBuildings() {

  var currentX: CGFloat = -15


  while currentX < 1024 {

    let size = CGSize(width: RandomInt(min: 2, max: 4) * 40, height:
RandomInt(min: 300, max: 600))

    currentX += size.width + 2
```

```
    let building = BuildingNode(color: UIColor.redColor(), size:
size)

    building.position = CGPoint(x: currentX - (size.width / 2), y:
size.height / 2)

    building.setup()

    addChild(building)


    buildings.append(building)
  }

}
```

Make sure you select iPad 2 and not iPad Air from the list of simulator options, then press Play to see the results of your hard work – a random set of buildings will be generated each time you run the game. Well done!

Because we draw the buildings in code, our game level is different every time it runs.

Mixing UIKit and SpriteKit

We've been mixing UIKit and SpriteKit ever since our first SpriteKit project, way back in project 11. Don't believe me? Look inside **GameViewController.swift** and you'll see a plain old **UIViewController** do all the work of loading and showing our **GameScene** code. There's a Main.storyboard file containing that view controller, and if you go to the identity inspector (Alt+Cmd+3) you'll see it has **SKView** set for its custom class – that's the view holding our scene.

This UIKit setup existed all along, but so far we've been ignoring it. No more: we're going to add some controls to that view so that players can fire bananas. The way the game works, each player gets to enter an angle and a velocity for their throw. We'll be recreating this with a **UISlider** for both of these numbers, along with a **UILabel** so players can see exactly what numbers they chose. We'll also add a "Launch" button that makes the magic happen.

Now, think about this for a moment: our game view controller needs to house and manage the user interface, and the game scene needs to manage everything inside the game. But they also need to talk to each other: the view controller needs to tell the game scene to fire a banana when the launch button is clicked, and the game scene needs to tell the view controller when a player's turn has finished so that another banana can be launched again.

This two-way communication could be done using **NSNotificationCenter**, but it's not very pleasant: we know the sender and receiver, and we know exactly what kind of data they will send and receive, so the easiest solution here is to give the view controller a property that holds the game scene, and give the game scene a property that holds the view controller.

In the very first project, I explained that outlet properties should be declared weak "because the object has been placed inside a view, so the view owns it." That's true, but using **weak** to declare properties is a bit more generalised: it means "I want to hold a reference to this, but I don't own it so I don't care if the reference goes away."

When we discussed closures in project 5, I explained that you needed to make **self** either **unowned** or **weak** so that you avoided strong reference cycles – where a view controller owns a closure and the closure owns the view controller so that neither of them ever get destroyed. Well, with our game scene and game view controller have the same problem: if they both own each other using a property, we have a problem.

The solution is to make one of them have a weak reference to the other: either the game controller owns the game scene strongly, or the game scene owns the game controller

strongly, but not both. As it so happens, the game controller already strongly owns the game scene, albeit indirectly: it owns the **SKView** inside itself, and the view owns the game scene. So, it's owned, we just don't have a reference to it.

So, our solution is straightforward: add a strong reference to the game scene inside the view controller, and add a weak reference to the view controller from the game scene. Add this property to the game scene:

```
weak var viewController: GameViewController!
```

Now add this property to the game view controller:

```
var currentGame: GameScene!
```

Like I said, the game controller already owns the game scene, but it's a pain to get to. Adding that property means we have direct access to the game scene whenever we need it. To set the property, put this into the **viewDidLoad()** method of the game view controller, just after the call to **presentScene()**:

```
currentGame = scene
scene.viewController = self
```

The first line sets the property to the initial game scene so that we can start using it. The second line makes sure that the reverse is true so that the scene knows about the view controller too.

Now to design the user interface: this needs two sliders, each with two labels, plus a launch button and one more label that will show whose turn it is. When you open Main.storyboard you'll see that it's a square, as per usual. In project 1 I said this square layout was a result of Auto Layout, which isn't *strictly* true: it's actually a result of size classes, which are a way of defining Auto Layout constraints that are restricted to specific categories of device.

The square layout is helpful because there are no square iOS devices, so you're always designing in a very abstract sense. But it's going to make our job here rather difficult, so for once we're going to disable Size Classes and design specifically for iPad-sized screens. Click inside the view controller then go to the file inspector (Alt+Cmd+1). About half way down you'll see a checkbox for "Use Size Classes". Please deselect that, then choose iPad when prompted.

You'll now see a black view the size of a portrait iPad screen. Our project is landscape only, so select the view controller (not the view) by clicking the orange and white icon above the canvas, then go to the attributes inspector and change orientation from "Inferred" to "Landscape". That's it: we now have an iPad-sized screen to work on. You might need to scroll around a bit depending on the resolution of your Mac's display!

Drop two sliders into your layout, both 300 points wide. The first should be at X:20, the second should be at X:480, and both should be at Y:20. Now place two labels in there, both 120 points wide. The first should be at X:325, the second should be at X:785, and both should be at Y:24 – this is slightly lower than the sliders so that everything is centered neatly.

For the launch button, place a button at X:910 Y:13, with width 100 and height 44; for the "which player is it?" button, place a label at X:370 Y:64 with width 285 and height 35.

That's the basic layout, but to make it all perfect we need a few tweaks. Using the attributes inspector, change the left-hand slider so that it has a maximum value of 90 and a current value of 45, then change the right-hand slider so that it has a maximum value of 250 and a current value of 125.

Make sure all three of your labels have their text color set to white, then give the bottom one the text "<<< PLAYER ONE". Select the button then give it a system bold font of size 22, a title of "LAUNCH" and a red text color.

That's the layout all done, but we also need lots of outlets: using the assistant editor, create these outlets:

 • For the left slider: **angleSlider** • For the left label: **angleLabel** • For the right slider: **velocitySlider** • For the right label: **velocityLabel** • For the launch button: **launchButton** • For the player number: **playerNumber**
You'll also need to create actions from the left slider, the right slider and the button: **angleChanged()**, **velocityChanged()** and **launch()** respectively.

That's all the layout done, so we're finished with Interface Builder and you can open up GameViewController.swift.

We need to fill in three methods (**angleChanged()**, **velocityChanged()** and **launch()**), write one new method, then make two small changes to **viewDidLoad()**.

The action methods for our two sliders are both simple: they update the correct label with the slider's current value. A **UISlider** always stores its values as a **Float**, but we only care about the integer value of that float so we're going to convert the values to **Int**s then use string interpolation to update the labels. Here's the code for both these methods:

Unleash the bananas!

It's time to get down to the nitty-gritty business of writing code: we need to create our players then fill in the **launch()** method so that the fun can begin.

We're going to start with the easy bit, which is creating players. This needs to do two things:

1. Create a player sprite and name it "player1".
2. Create a physics body for the player that collides with bananas, and set it to not be dynamic.
3. Position the player at the top of the second building in the array. (This is why we needed to keep an array of the buildings.)
4. Add the player to the scene.
5. Repeat all the above for player 2, except they should be on the second to last building.

The player physics body can be made using a circle, because the sprite used (which is the same for both players) is more or less round. We used the second building for player 1 and the second to last for player 2 so that they aren't at the very edges of the screen. Positioning them at the top is just a matter of adding the building's height to the player's height and dividing by two, then adding that to the building's Y co-ordinate. SpriteKit measures from the center of nodes, remember!

Before we look at the code, you'll need to create some properties to hold both players, plus the banana and which player is currently in control:

```
var player1: SKSpriteNode!

var player2: SKSpriteNode!

var banana: SKSpriteNode!


var currentPlayer = 1
```

Now here's the code for **createPlayers()** – please put this in GameScene.swift:

```
func createPlayers() {
```

```
player1 = SKSpriteNode(imageNamed: "player")

player1.name = "player1"

player1.physicsBody = SKPhysicsBody(circleOfRadius:
player1.size.width / 2)

player1.physicsBody!.categoryBitMask =
CollisionTypes.Player.rawValue

player1.physicsBody!.collisionBitMask =
CollisionTypes.Banana.rawValue

player1.physicsBody!.contactTestBitMask =
CollisionTypes.Banana.rawValue

player1.physicsBody!.dynamic = false


let player1Building = buildings[1]

player1.position = CGPoint(x: player1Building.position.x, y:
player1Building.position.y + ((player1Building.size.height +
player1.size.height) / 2))

addChild(player1)


player2 = SKSpriteNode(imageNamed: "player")

player2.name = "player2"

player2.physicsBody = SKPhysicsBody(circleOfRadius:
player2.size.width / 2)

player2.physicsBody!.categoryBitMask =
CollisionTypes.Player.rawValue

player2.physicsBody!.collisionBitMask =
CollisionTypes.Banana.rawValue

player2.physicsBody!.contactTestBitMask =
CollisionTypes.Banana.rawValue

player2.physicsBody!.dynamic = false


let player2Building = buildings[buildings.count - 2]

player2.position = CGPoint(x: player2Building.position.x, y:
player2Building.position.y + ((player2Building.size.height +
player2.size.height) / 2))

addChild(player2)
```

```
}
```

Now, one thing we haven't done yet is actually add in the images to be used inside the game, and the reason for that is because we're going to use a special technique called texture atlases. SpriteKit doesn't use them by default, which is why we haven't used them yet – there are bigger things to worry about! But this game is perfect for texture atlases, so we're going to use them now.

A texture atlas is a folder with the extension **.atlas**. In Finder, go into your project directory (where your .swift files are), then create a new folder called **assets.atlas**. Now go to where you downloaded the files for this project, and drag all the image assets from there into your assets.atlas directory. Finally, drag your assets.atlas directory into your Xcode project so that it gets added to the build.

Warning: you should only put the image assets into your atlas, i.e. the .png files. The two .sks files – hitBuilding.sks and hitPlayer.sks – should be dragged into your project directly, not into the texture atlas.

How is that different from adding the pictures individually? To find out, change your launch destination to be "iOS Device" rather than "iPad 2" then press Cmd+B to build your game without running it. At the very bottom of the project navigator, you'll see a yellow folder called Products. Open that up to reveal Project29.app, then right-click on that and choose "Show in Finder".

This is the actual app that Xcode built from your project, along with some intermediary stuff alongside. Your app will be selected already, so right click on it and choose Show Package Contents to see what's inside. You should see a variety of things, including a directory called assets.atlasc, which is our compiled assets folder. Inside *that* you'll see just three files: your textures, your textures at 2x resolution, and a plist file describing them.

What SpriteKit has done is take all the images for our project and sewn them into a single image file. This means it can draw lots of images without having to load and unload textures – it effectively just crops the big image as needed. SpriteKit automatically generates these atlases for us, even rotating sprites to make them fit if needed. And the best bit: just like using Images.xcassets, you don't need to change your code to make them work; just load sprites the same way you've always done.
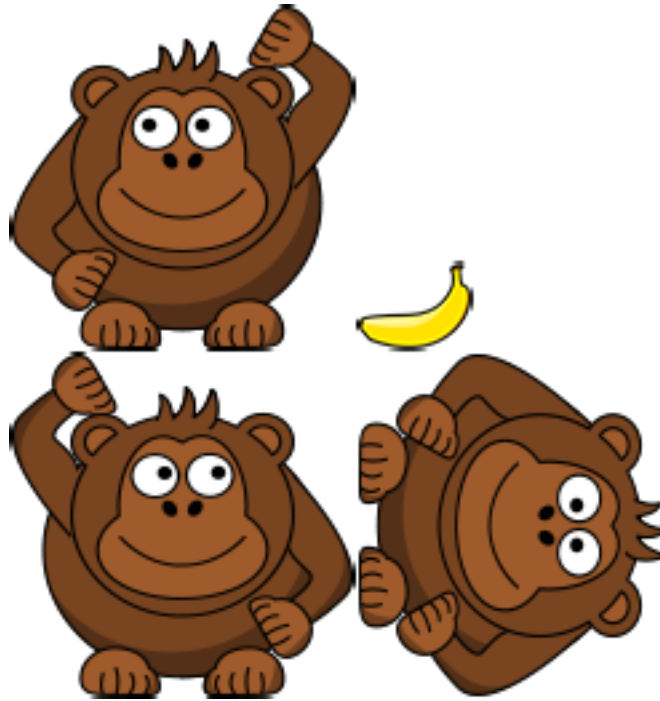
SpriteKit's texture atlases are more resource-efficient, so they load faster and take up less space.

(Note: observant readers may notice that the player images are in fact monkeys not gorillas. This is largely down to me not being able to find a public domain gorilla picture that was good enough, and I figured penguins deserved a break.)

With the **createPlayers()** method in place, all you need to is call it inside **didMoveToView()**, just after the **createBuildings()** line:

```
override func didMoveToView(view: SKView) {

  backgroundColor = UIColor(hue: 0.669, saturation: 0.99, brightness:
0.67, alpha: 1)


  createBuildings()

  createPlayers()

}
```

It's now time to flesh out the **launch()** method.   This is a complicated method because it needs to do quite a few things:

1. Figure out how hard to throw the banana. We accept a velocity parameter, but I'll be dividing that by 10. You can adjust this based on your own playtesting.
2. Convert the input angle to radians. Most people don't think in radians, so the input will come in as degrees that we will convert to radians.
3. If somehow there's a banana already, we'll remove it then create a new one using circle physics.
4. If player 1 was throwing the banana, we position it up and to the left of the player and give it some spin.
5. Animate player 1 throwing their arm up then putting it down again.
6. Make the banana move in the correct direction.
7. If player 2 was throwing the banana, we position it up and to the right, apply the opposite spin, then make it move in the correct direction.

There are few things you need to know before we translate that long list into Swift. First, converting degrees to radians is done with a fixed formula that we will put into a method called **deg2rad()**:

```
func deg2rad(degrees: Int) -> Double {

  return Double(degrees) * M_PI / 180.0

}
```

Second, SpriteKit uses a number of optimisations to help its physics simulation work at high speed. These optimisations don't work well with small, fast-moving objects, and our banana is just such a thing. To be sure everything works as intended, we're going to enable the **usesPreciseCollisionDetection** property for the banana's physics body. This works slower, but it's fine for occasional use.

Third, I said we needed to make the banana move in "the correct direction" without really explaining how we get to that. This isn't a trigonometry book, so here's the answer as briefly as possible: if we calculate the cosine of our angle in radians it will tell us how much horizontal momentum to apply, and if we calculate the sine of our angle in radians it will tell us how much vertical momentum to apply.

Once that momentum is calculated, we multiply it by the velocity we calculated (or negative velocity in the case of being player 2, because we want to throw to the left), and turn it into a **CGVector**. Remember, a vector is like an arrow where its base is at 0,0 (our current position) and tip at the point we specify, so this effectively points an arrow in the direction the banana should move.

To make the banana actually move, we use the **applyImpulse()** method of its physics body, which accepts a **CGVector** as its only parameter and gives it a physical push in that direction.

Time for the code; so you don't have to flick around so much while reading, here's a repeat list of what this method will do, with numbers matching comments in the code:

1. Figure out how hard to throw the banana. We accept a velocity parameter, but I'll be dividing that by 10. You can adjust this based on your own playtesting.
2. Convert the input angle to radians. Most people don't think in radians, so the input will come in as degrees that we will convert to radians.
3. If somehow there's a banana already, we'll remove it then create a new one using circle physics.
4. If player 1 was throwing the banana, we position it up and to the left of the player and give it some spin.
5. Animate player 1 throwing their arm up then putting it down again.
6. Make the banana move in the correct direction.
7. If player 2 was throwing the banana, we position it up and to the right, apply the opposite spin, then make it move in the correct direction.

And here's the code:

```
func launch(angle angle: Int, velocity: Int) {
    // 1
    let speed = Double(velocity) / 10.0


    // 2
    let radians = deg2rad(angle)


    // 3
```

```swift
    if banana != nil {

      banana.removeFromParent()

      banana = nil

    }


    banana = SKSpriteNode(imageNamed: "banana")

    banana.name = "banana"

    banana.physicsBody = SKPhysicsBody(circleOfRadius:
banana.size.width / 2)

    banana.physicsBody!.categoryBitMask =
CollisionTypes.Banana.rawValue

    banana.physicsBody!.collisionBitMask =
CollisionTypes.Building.rawValue | CollisionTypes.Player.rawValue

    banana.physicsBody!.contactTestBitMask =
CollisionTypes.Building.rawValue | CollisionTypes.Player.rawValue

    banana.physicsBody!.usesPreciseCollisionDetection = true

    addChild(banana)


    if currentPlayer == 1 {

      // 4

      banana.position = CGPoint(x: player1.position.x - 30, y:
player1.position.y + 40)

      banana.physicsBody!.angularVelocity = -20


      // 5

      let raiseArm = SKAction.setTexture(SKTexture(imageNamed:
"player1Throw"))

      let lowerArm = SKAction.setTexture(SKTexture(imageNamed:
"player"))

      let pause = SKAction.waitForDuration(0.15)

      let sequence = SKAction.sequence([raiseArm, pause, lowerArm])

      player1.runAction(sequence)
```

```
    // 6

    let impulse = CGVector(dx: cos(radians) * speed, dy: sin(radians)
* speed)

    banana.physicsBody?.applyImpulse(impulse)

  } else {

    // 7

    banana.position = CGPoint(x: player2.position.x + 30, y:
player2.position.y + 40)

    banana.physicsBody!.angularVelocity = 20


    let raiseArm = SKAction.setTexture(SKTexture(imageNamed:
"player2Throw"))

    let lowerArm = SKAction.setTexture(SKTexture(imageNamed:
"player"))

    let pause = SKAction.waitForDuration(0.15)

    let sequence = SKAction.sequence([raiseArm, pause, lowerArm])

    player2.runAction(sequence)


    let impulse = CGVector(dx: cos(radians) * -speed, dy:
sin(radians) * speed)

    banana.physicsBody?.applyImpulse(impulse)

  }

}
```

With that code, the game is starting to come together. Sure, the bananas don't actually explode, and player 2 never actually gets a shot, but all in good time…

Destructible terrain

It's time for the most challenging part of our project, but as per usual I've tried to keep things as simple as possible because the fun is in getting results not in learning algorithms. We're going to add collision detection to our code so that players can carve chunks out of the buildings or, better, blow up their opponents.

You will, as always, need to assign **self** to be the delegate of your scene's physics world so that you can get notified of collisions. So, put this in **didMoveToView()**:

```
physicsWorld.contactDelegate = self
```

Make sure you modify your class definition to say that you conform to the **SKPhysicsContactDelegate** protocol.

When it comes to implementing the **didBeginContact()** method, there are various possible contacts we need to consider: banana hit building, building hit banana (remember the philosophy?), banana hit player1, player1 hit banana, banana hit player2 and player2 hit banana. This is a lot to check, so we're going to eliminate half of them by eliminating whether "banana hit building" or "building hit banana". Take another look at our category bitmasks:

```
enum CollisionTypes: UInt32 {
  case Banana = 1
  case Building = 2
  case Player = 4
}
```

They are ordered numerically and alphabetically, so what we're going to do is create two new variables of type **SKPhysicsBody** and assign one object from the collision to each: the first physics body will contain the lowest number, and the second the highest. So, if we get banana (collision type 1) and building (collision type 2) we'll put banana in body 1 and building in body 2, but if we get building (2) and banana (1) then we'll still put banana in body 1 and building in body 2.

Once we have eliminated half the checks, we're going to optionally unwrap both the bodies. They are optional because they might be **nil**, and this is highly likely in our project. The reason it's likely is because we might get "banana hit building" and "building hit banana" one after the other, but when either of these happens we'll destroy the banana so the second one will definitely be **nil**.

If the banana hit a player, we're going to call a new method named **destroyPlayer()**. If the banana hit a building, we'll call a different new method named **bananaHitBuilding()**, but we'll also pass in the contact point. This value tells us where on the screen the impact actually happened, and it's important because we're going to destroy the building at that point.

That's all you need to know, so here's the code for **didBeginContact()**:

```
func didBeginContact(contact: SKPhysicsContact) {
  var firstBody: SKPhysicsBody
  var secondBody: SKPhysicsBody

  if contact.bodyA.categoryBitMask < contact.bodyB.categoryBitMask {
    firstBody = contact.bodyA
    secondBody = contact.bodyB
  } else {
    firstBody = contact.bodyB
    secondBody = contact.bodyA
  }

  if let firstNode = firstBody.node {
    if let secondNode = secondBody.node {
      if firstNode.name == "banana" && secondNode.name ==
"building" {
        bananaHitBuilding(secondNode as! BuildingNode, atPoint:
contact.contactPoint)
      }
```

```
        if firstNode.name == "banana" && secondNode.name == "player1" {

            destroyPlayer(player1)

        }


        if firstNode.name == "banana" && secondNode.name == "player2" {

            destroyPlayer(player2)

        }

    }

  }

}
```

If a banana hits a player, it means they have lost the game: we need to create an explosion (yay, particles!), remove the destroyed player and the banana from the scene, then… what? Well, so far we've just left it there – we haven't looked at how to make games restart.

There are a number of things you could do: take players to a results screen, take them to a menu screen, and so on. In our case, we're going to reload the level so they can carry on playing. We could just delete all the buildings and generate it all from scratch, but that would be passing up a great opportunity to learn something new!

SpriteKit has a super-stylish and built-in way of letting you transition between scenes. This means you can have one scene for your menu, one for your options, one for your game, and so on, then transition between them as if they were view controllers in a navigation controller.

To transition from one scene to another, you first create the scene, then create a transition using the list available from **SKTransition**, then finally use the **presentScene()** method of our scene's view, passing in the new scene and the transition you created. For example, this will cross-fade in a new scene over 2 seconds:

```
let newGame = GameScene(size: self.size)

let transition = SKTransition.crossFadeWithDuration(2)

self.view?.presentScene(newGame, transition: transition)
```

In the **destroyPlayer()** method we're going to execute the scene transition after two seconds so that players have a chance to see who won and, let's face it, laugh at the losing player. But when we create the new game scene we also need to do something very important: we need to update the view controller's **currentGame** property and set the new scene's **viewController** property so they can talk to each other once the change has happened.

We also need to call the **changePlayer()** method when a player is destroyed. We haven't written this method yet, but it transfers control of the game to the other player, then calls the **activatePlayerNumber()** method on the game view controller so that the game controls are re-shown. Calling this method here ensures that the player who lost gets the first turn in the new game.

First, here's the code for **destroyPlayer()**:

```
func destroyPlayer(player: SKSpriteNode) {
  let explosion = SKEmitterNode(fileNamed: "hitPlayer")!
  explosion.position = player.position
  addChild(explosion)

  player.removeFromParent()
  banana?.removeFromParent()

  RunAfterDelay(2) { [unowned self] in
    let newGame = GameScene(size: self.size)
    newGame.viewController = self.viewController
    self.viewController.currentGame = newGame

    self.changePlayer()
    newGame.currentPlayer = self.currentPlayer

    let transition = SKTransition.doorwayWithDuration(1.5)
```

```
        self.view?.presentScene(newGame, transition: transition)



    }

}
```

Important: after calling **changePlayer()**, we must set the new game's **currentPlayer** property to our own **currentPlayer** property, so that whoever died gets the first shot.

The **changePlayer()** method is trivial, so here it is:

```
func changePlayer() {

  if currentPlayer == 1 {

    currentPlayer = 2

  } else {

    currentPlayer = 1

  }


  viewController.activatePlayerNumber(currentPlayer)

}
```

Now it's time for the real work. How do we allow our exploding bananas to create holes in buildings? Surprisingly, it's not that hard. I'm going to split it into two parts: a **bananaHitBuilding()** game scene method that handles creating the explosion, deleting the banana and changing players, and a **hitAtPoint()** building node method that handles damaging the building. The first one is easy, so put this into the game scene:

```
func bananaHitBuilding(building: BuildingNode, atPoint contactPoint:
CGPoint) {

  let buildingLocation = convertPoint(contactPoint, toNode: building)

  building.hitAtPoint(buildingLocation)
```

```
let explosion = SKEmitterNode(fileNamed: "hitBuilding")!

explosion.position = contactPoint

addChild(explosion)


banana.name = ""

banana?.removeFromParent()

banana = nil


changePlayer()
}
```

The only new thing in there is the call to **convertPoint()**, which asks the game scene to convert the collision contact point into the coordinates relative to the building node. That is, if the building node was at X:200 and the collision was at X:250, this would return X:50, because it was 50 points into the building node.

If you're curious why I use **banana.name = ""**, it's to fix a small but annoying bug: if a banana just so happens to hit two buildings at the same time, then it will explode twice and thus call **changePlayer()** twice – effectively giving the player another throw. By clearing the banana's name here, the second collision won't happen because our **didBeginContact()** method won't see the banana as being a banana any more – it's name is gone.

And now for the part where we handle destroying chunks of the building. With your current knowledge of Core Graphics, this is something you can do by learning only one new thing: blend modes. When you draw anything to a Core Graphics context, you can set how it should be drawn. For example, should it be be drawn normally, or should it add to what's there to create a combination?

Core Graphics has quite a few blend modes that might look similar, but we're going to use one called **.Clear**, which means "delete whatever is there already." When combined with the fact that we already have a property called **currentImage** you might be able to see how our destructible terrain technique will work!

Put simply, when we create the building we save its **UIImage** to a property of the **BuildingNode** class. When we want to destroy part of the building, we draw that image into

a new context, draw an ellipse using **.Clear** to blast a hole, then save that back to our **currentImage** property and update our sprite's texture.

Here's a full break down of what the method needs to do:

1. Figure out where the building was hit. Remember: SpriteKit's positions things from the center and Core Graphics from the bottom left!
2. Create a new Core Graphics context the size of our current sprite.
3. Draw our current building image into the context. This will be the full building to begin with, but it will change when hit.
4. Create an ellipse at the collision point. The exact co-ordinates will be 32 points up and to the left of the collision, then 64x64 in size - an ellipse centered on the impact point.
5. Set the blend mode **.Clear** then draw the ellipse, literally cutting an ellipse out of our image.
6. Convert the contents of the Core Graphics context back to a **UIImage**, which is saved in the **currentImage** property for next time we're hit, and used to update our building texture.
7. Call **configurePhysics()** again so that SpriteKit will recalculate the per-pixel physics for our damaged building.

Here's that in code – put this method into the **BuildingNode** class:

```
func hitAtPoint(point: CGPoint) {

  let convertedPoint = CGPoint(x: point.x + size.width / 2.0, y:
abs(point.y - (size.height / 2.0)))


  UIGraphicsBeginImageContextWithOptions(size, false, 0)

  let context = UIGraphicsGetCurrentContext()


  currentImage.drawAtPoint(CGPoint(x: 0, y: 0))


  CGContextAddEllipseInRect(context, CGRect(x: convertedPoint.x - 32,
y: convertedPoint.y - 32, width: 64, height: 64))

  CGContextSetBlendMode(context, .Clear)

  CGContextDrawPath(context, .Fill)
```

```
    let img = UIGraphicsGetImageFromCurrentImageContext()
    UIGraphicsEndImageContext()


    texture = SKTexture(image: img)
    currentImage = img


    configurePhysics()
}
```

That's it for destructible terrain! There's one curious quirk of SpriteKit's physics implementation: if you slice a building in two with lots of bananas, only one half will respond to physics because it won't put two (now separate) physics bodies into one. Fortunately, the chances of that happening are pretty slim unless you're an appalling shot!

Wrap up

This was our final Level 1 game project, and I'd like to think I kept the best for last: lots to learn about mixing UIKit and SpriteKit, texture atlases, scene transitions, and of course destructible terrain. If you've followed the series correctly you've now made seven SpriteKit games of varying complexity, so I hope you have all the knowledge you need to get out there and make your own.

If you want to extend this project, you might want to consider starting with the art for a change: I've made it look relatively similar to the original DOS game, but let's face it that will only appear to fans of retro gaming nowadays! If you're looking to change the code, see if you can make the game track scores across scenes so that players know who is winning.

If you're looking for something harder, make it best of 5: whoever reaches a score of 3 first wins, showing a "you win!" screen of your choosing. And for a real challenge, try to modify the collision detection so that exploding bananas damage all buildings the explosion would have touched rather than just the building the banana touched.

One last thing: when you're making amateur games, don't let yourself get bogged down in boring stuff too much – you're doing it for fun, and if it isn't fun to make it's probably not much fun to play.

Project 30

Instruments

Become a bug detective and track down lost memory, slow drawing and more.

Setting up

This is the last technique project and indeed the last project in the series, so it's going to be quite different. You see, I've already written all the code for you and I'm giving you a working app. Sure it has a few bugs here and there, but it's not *too* bad. Well, OK: perhaps it's full of bugs, and perhaps this whole project is about showing you how to find and fix those bugs!

We're going to be using a tool called Instruments. It ships as part of Xcode, and is responsible for profiling your app. "Profiling" is the term used when we monitor performance, memory usage and other information of an app, with the aim of improving efficiency. I'm not going to make you a master of Instruments, but I can at least show you how it helps you find problems with your code. Plus, along the way you'll learn a few extra bits about how iOS works, including shadows, image caching, cell reuse and more.

Please note: it's recommended you use physical devices when profiling your apps because their performance characteristics are very different from those of the iOS simulator. It's not required, just strongly recommended. If you're going to choose a device, I recommend choosing the least powerful device you support. All our apps are written for iOS 8, so that means iPhone 4s and onwards and iPad 2 and onwards. If you have one available, I generally find iPad 3 the slowest Pad for testing purposes.

You should download the source code for this project from **GitHub** then modify it as we go. When you open the project, change the the build destination to be your iOS device if you have one handy, otherwise use the iPhone 6 simulator.

What are we working with?

This is a really simple app, albeit quite a broken on. Obviously the breakages are deliberate, but they are all things I have seen in real, shipping code. The app shows a table view containing images from Greek mythology. When one of the rows is tapped, a detail view controller appears, showing the image at full size.

Every time you tap on the big image, it adds one to a count of how many times that image was tapped, and that count is shown in the original view controller. All the images in the app are deliberately blurry – I use them in my Mythology app, so although I've kept them at their original size I have dramatically lowered the quality. Sorry!

Anyway, that's all the app does. I haven't even used a storyboard, because I want all the problems to be visible (and fixable!) in Swift code.

And how bad are the problems? Well, if you run the app on a device, you'll see that it crashes after viewing just a few pictures. Regardless of whether you're running on a device or a simulator, press Cmd+6 to bring up the debug gauges on the left of the Xcode window if you don't see them already – it will say "CPU", "Memory", "Disk" and "Network".

Depending on whether you're using the iOS simulator or a device, you might see memory in the debug gauges start low but then build up as you use the app more and more, never going back down after you load an image. Soon the app will crash because it runs out of memory, which is of course A Very Bad Thing.

You might also notice that in the iOS Simulator (running with the full power of your Mac behind it!) table scrolling is still slow. What gives? This is a really broken app, and with so few lines of code!

What can Instruments tell us?

Press Cmd+I to run your app using Instruments, and after a few seconds a window will appear offering you a variety of profiling templates. Please select Time Profiler then click Choose. When the new window appears, click the red circle in the top-left corner to start recording of the app.

Your app will launch on your device (or in the simulator) and Instruments will begin monitoring its usage in realtime. You'll see a spike in Instruments' readings to begin with, which reflect the huge amount of work any app does just to start up. We don't care about that for now, we're more interested in the workload of the app once it's running.

So, scroll around a bit, tap an image, go back, scroll around some more, tap another image, and so on. Aim to get about 10 seconds or so of real app usage, then press the Stop button, which is where the record button was earlier.

The top half of your Instruments window is showing readings from your app; the bottom half is showing details on those readings. By default, the detail view shows everything in the app run that was captured, but I want you to click and drag in the top window to select part of the readings when you tapped on an image. All being well (or as well as can be expected in this broken code!) you should see the readings noticeably spike.

When you select an area of the readings like this, the detail view now shows information just on the part you chose. You should see that the vast majority of the time that was selected was spent on the main thread, which means we're not taking much advantage of having multiple CPU cores.

Immediately to the left of "Main thread" in the detail view is a disclosure arrow. You can click that to open up all the top-level calls maid on the Main Thread, which will just be "Start", which in turn has its own calls under its own arrow. You can if you want hold down Alt and click on these arrows, which causes all the children (and their children's children) to be opened up, but that gets messy!

Instead, there are two options. First, you should have a right-hand detail pane with three options: record settings, display settings and extended detail (accessible through Cmd+1, Cmd+2 and Cmd+3). Select "Main thread" in the detail view then press Cmd+3 to choose the extended detail view: this will automatically show you the "heaviest" stack trace, which is the code that took the most time to run.

In the picture below you can see the bottom half of Instruments after running a time profile. The pane on the right is showing the heaviest stack trace, and on the left you can see all the threads that were executing as well as what they were doing.
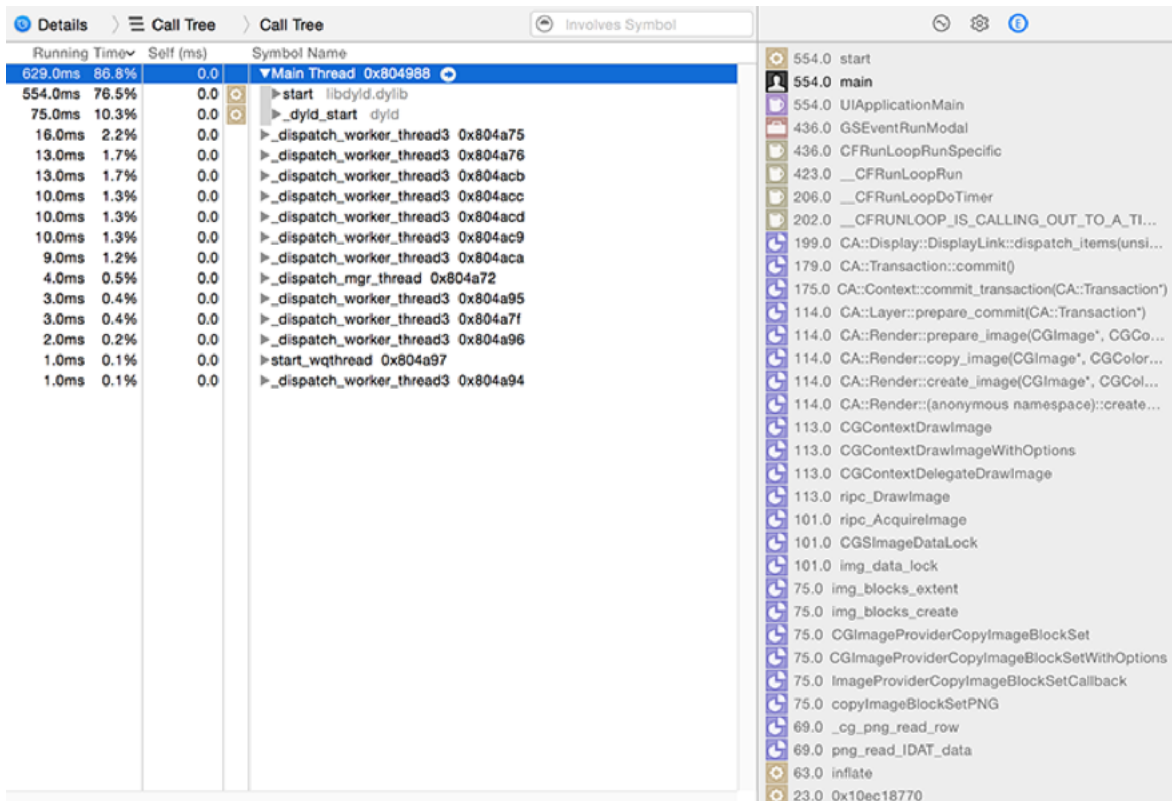
Instruments can show you the heaviest stack trace in your app, which gives you your first optimization target.

If you scroll down to the bottom, you'll see it's probably all around either **copyImageBlockSetPNG** or **copyImageBlockSetAppleJPEG** depending on whether you spent more of your time flicking through the table view or going into the detail controller.

Now, we didn't write that code, so you won't find it anywhere in the project. But we didn't *need* to write it, because some of our own code triggered it inside Apple's frameworks. This heaviest stack trace might not always be your problem; sometimes it's just something that's always going to be slow. That said, it's always a good place to start!

The second option is inside the display settings view (Cmd+2) and is called Invert Call Tree. This will show you the heaviest code first, with a disclosure arrow revealing what called it, and so on. This is the same view as before, just reversed.

There is a third option, but it's of mixed help. It's called "Hide system libraries" and is near to the "Invert Call Tree" checkbox. This eliminates all time being used by Apple's frameworks so you can focus on your own code. This is great if you have lots of complicated code that could be slow, but it doesn't help at all if your app is slow because you're using the system libraries poorly!

Quit Instruments and return to Xcode, then press Cmd+I again to re-run Instruments. This time I'd like you to choose the Core Animation instrument then press record to begin. The Core Animation instrument at the top will show a few FPS (frames per second) to begin with, then settle at 0 – that doesn't mean your app is running at zero frames per second, just that it's not doing anything right now.

Scroll around the table view a little and watch how the Core Animation instrument responds. Ideally you'll get a constant 60fps, but that's only likely on the very latest generations of iPhones – iPhone 5 and 5s will struggle to come close, and iPhone 4 has no chance.

Under the display settings for the CA instrument (Cmd+2) you'll see three very helpful options:

 • **Color Blended Layers** shows views that are opaque in green and translucent in red. If there are multiple transparent views inside each other, you'll see more and more red.
 • **Color Offscreen-Rendered Yellow** shows views that require an extra drawing pass in yellow. Some special drawing work must be drawn individually off screen then drawn again onto the screen, which means a lot more work.
 • **Color Hits Green and Misses Red** tells you how well image caching is working. We haven't used image caching yet, but it basically means "do all this complicated work once, then reuse the result." If you get it wrong, you cache the result then throw it away very often.

Broadly speaking, you want "Color Blended Layers" to show as little red as possible, "Color Offscreen-Rendered Yellow" to show no yellow, and "Color Hits Green and Misses Red" to either show no color or show green.

Try all three of these options on both the table view and the detail view to see how things look.

(Fun tip: you can turn these settings on then unplug a device – it's a great way to prank friends!)

Fixing the bugs

It's time for us to use instruments to spot and fix some problems. **Important:** when making performance changes you should change only one thing at a time, then re-test to make sure your change helped. If you changed two or more things and performance got better, which one worked? Or, if performance got worse, perhaps one thing worked and one didn't!

Let's begin with the table view: you should have seen nearly all the table view turn yellow when Color Offscreen-Rendered Yellow was selected. This is because each of the pictures has a shadow effect behind them that's being done on the **CALayer** belonging to the image.

You can find the code for this in SelectionViewController.swift, inside the **cellForRowAtIndexPath** method:

```
// give the images a nice shadow to make them look a bit more
dramatic
cell.imageView!.layer.shadowColor = UIColor.blackColor().CGColor

cell.imageView!.layer.shadowOpacity = 1

cell.imageView!.layer.shadowRadius = 10
```

iOS lets you add a basic shadow to any of its views, and it's a simple way to make something stand out on the screen. But it's not fast: it literally scans the pixels in the image to figure out what's transparent, then uses that information to draw the shadow correctly. When you hit a performance problem, you either drop the code that triggers the problem or you make it run faster.

In our case, we'll assume the designer insists the drop shadow is gorgeous (they are wrong!) so we need to make the code faster. Fortunately, if we give iOS a little more information it can do substantially less work. Calculating the shadow is hard, because iOS needs to see what's transparent. But we know nothing is transparent; our images are solid rectangles!

We can tell iOS not to automatically calculate the shadow path for our images by giving it the exact shadow path to use. The easiest way to do this is to create a new **UIBezierPath** that describes our images (a rectangle with width 320 and height 240), then convert it to a **CGPath** because **CALayer** doesn't understand what **UIBezierPath** is.

Here's the updated code:

```
// give the images a nice shadow to make them look a bit more
dramatic

cell.imageView!.layer.shadowColor = UIColor.blackColor().CGColor

cell.imageView!.layer.shadowOpacity = 1

cell.imageView!.layer.shadowRadius = 10

cell.imageView!.layer.shadowPath = UIBezierPath(rect: CGRect(x: 0, y:
0, width: 320, height: 240)).CGPath
```

When you run that, you'll still see the same shadows everywhere, but the yellow color is gone. You can turn off Color Offscreen-Rendered Yellow now and quit Instruments.

Back in Xcode, press Cmd+I to launch a fresh instance of Instruments, and this time I want you to choose the Allocations instrument. This tells you how many objects you're creating and what happens to them. Press record, then scroll around the table view a few times to get a complete picture of the app running. At the very least, you should go all the way down to the bottom and back up five or six times.

What you'll see is a huge collection of information being shown – lots of "malloc", lots of "CFString", lots of "__NSArrayI" and more. Stuff we just don't care about right now, because most of the code we have is user interface work. Fortunately, there's a search box just above the detail pane – it should say "Instrument Detail" but if you type "UI" in there it will only show information that has "UI" somewhere in there, which just happens to be all of Apple's user interface libraries!

In the picture below you can see how filtering for "UI" inside Instruments shows only data that has "UI" in its name somewhere, which primarily restricts the view to things that come from Apple's UIKit libraries.

Use Instruments's filtering system to show only items that matter to you.

Once you filter by "UI" you'll see see **UIImage**, **UIImageView**, **UITableViewCell** and more. The allocations instrument will tell you how many of these objects are persistent (created and still exist) and how many are transient (created and since destroyed). Notice how just swiping around has created a large number of transient **UIImageView** and **UITableViewCell** objects?

This is happening because each time the app needs to show a cell, it creates it then creates all the subviews inside it – namely an image view and a label. This is a common coding mistake to make when you're not using storyboards and prototype cells, and it's guaranteed to put a speed bump in your apps.

If you look inside SelectionViewController's **cellForRowAtIndexPath** method you'll see this line:

```
let cell = UITableViewCell(style: .Default, reuseIdentifier: "Cell")
```

That's the only place where table view cells are being created, so clearly it's the culprit because it creates a new cell every time the table view asks for one. This has been slow since the very first days of iOS development, and Apple has always had a solution: ask the table view to dequeue a cell, and if you get **nil** back then create a cell yourself.

This is different from when we were using prototype cells with a storyboard. With storyboards, if you dequeue a prototype cell then iOS automatically handles creating them as needed.

If you're creating table view cells in code, you have two options to fix this intense allocation of views. First, you could rewrite the above line to be this:

```
var cell = tableView.dequeueReusableCellWithIdentifier("Cell")

if cell == nil {
  cell = UITableViewCell(style: .Default, reuseIdentifier: "Cell")
}
```

That dequeues a cell, but if it gets **nil** back then we create one. Note the force-unwrapped optional at the end of the first line, meaning "this might be **nil**, but trust us: it's safe." And we know it's safe because we explicitly catch the **nil** scenario just afterwards.

The other solution you could use is to register a class with the table view for the reuse identifier "Cell". Using this method you could leave the original line of code as-is, but add this to **viewDidLoad()**:

```
tableView.registerClass(UITableViewCell.self, forCellReuseIdentifier: "Cell")
```

With that line of code, you will never get **nil** when dequeuing a cell with the identifier "Cell". As with prototype cells, if there isn't one to dequeue a new cell will be created automatically.

The second solution is substantially newer than the first and can really help cut down the amount of code you need. But it has two drawbacks: with the first solution you can specify different kinds of cell styles than just **.Default**, not least the **.Subtitle** option we used in project 7; also, with the first solution you explicitly know when a cell has just been created, so it's easy to force any one-off work into the **if cell == nil {** block.

Regardless of which solution you chose (you'll use both in your production code, I expect), you should be able to run the allocations instrument again and see far fewer table view cell allocations. With this small change, iOS will just reuse cells as they are needed, which makes your code run faster and operate more efficiently.

Now, why does the app crash when you go the detail view controller enough times? There are two answers to this question, one code related and one not. For the first answer, look up one of the images in Finder, such as hephaestus.jpg. What do you notice about it?

Hopefully you spotted that it was 6620x4140 pixels in size, which is ten times more pixels than even the iPhone 6 Plus screen has. Our designer forgot to scale them down! The end result is that we're trying to show far, far more detail than is visible, which is putting crazy pressure on memory and the graphics unit. The solution here: resize those images down to a sensible size – preferably the smallest they need to be in order to fit the design correctly.

But there's something else subtle here, and it's something we haven't covered yet so this is the perfect time. When you create a **UIImage** using **UIImage(named:)** iOS loads the image and puts it into an image cache for reuse later. This is sometimes helpful, particularly if you know the image will be used again. But if you know it's unlikely to be reused or if it's quite large, then don't bother putting it into the cache – it will just add memory pressure to your app and probably flush out other more useful images!

If you look in the **viewDidLoad()** method of **ImageViewController** you'll see this line of code:

```
imageView.image = UIImage(named: image)
```

How likely is it that users will go back and forward to the same image again and again? Not likely at all, so we can skip the image cache by creating our images using the **UIImage(contentsOfFile:)** method. This method isn't as friendly as **UIImage(named:)** because you need to specify the exact path to an image rather than just its filename in your app bundle, but you already know how to use **pathForResource()** so it's not so hard:

```
let path = NSBundle.mainBundle().pathForResource(image, ofType: nil)!
imageView.image = UIImage(contentsOfFile: path)
```

Let's take a look at one more problem, this time quite subtle. Loading the images was slow because they were so big, and iOS was caching them unnecessarily. But **UIImage**'s cache is intelligent: if it senses memory pressure, it automatically clears itself to make room for other stuff. So why does our app run out of memory?

To find out, profile the app using Instruments and select the allocations instrument again. This time filter on "imageviewcontroller" and to begin with you'll see nothing because the app starts on the table view. But if you tap into a detail view then go back, you'll see one is created *and remains persistent* – it hasn't been destroyed. Which means the image it's showing also hasn't been destroyed, hence the massive memory usage.

What's causing the image view controller to never be destroyed? If you read through SelectionViewController.swift and ImageViewController.swift you might spot these two things:

1. The selection view controller has a **viewControllers** array that claims to be a cache of the detail view controllers. This cache is never actually used, and even if it were used it really isn't needed.
2. The image view controller has a property **var owner: SelectionViewController!** – that makes it a strong reference to the view controller that created it.

The first problem is easily fixed: just delete the **viewControllers** array and any code that uses it, because it's just not needed. The second problem smells like a strong reference cycle, so you should probably change it to this:

```
weak var owner: SelectionViewController!
```

Run Instruments again and you'll see that the problem is… still there?! That's right: those two were either red herrings or weren't enough to solve the problem, because something far more sneaky is happening.

The view controllers aren't destroyed because of this line of code in ImageViewController.swift:

```
self.animTimer = NSTimer.scheduledTimerWithTimeInterval(5, target:
self, selector: "animateImage", userInfo: nil, repeats: true)
```

That timer does a hacky animation on the image, and it could easily be replaced with better animations as done inside project 15. But even so, why does that cause the image view controllers to never leak?

The reason is that when you specify a target for your timer (what object should be told when the timer is up), the timer holds a strong reference to it so that it's definitely there when the timer is up. We're using **self** for the target, which means our view controller owns the timer strongly and the timer owns the view controller strongly, so we have a strong reference cycle.

There are two solutions here: rewrite the code using smarter animations, or destroy the timer when it's no longer needed, thus breaking the cycle. The second option is easier, because it avoids having to write too much new code. In fact, all we need to do is detect when the image view controller is about to disappear and stop the timer. We'll do this in **viewWillDisappear()**:

```
override func viewWillDisappear(animated: Bool) {

  super.viewWillDisappear(animated)

  self.animTimer.invalidate()

}
```

Calling **invalidate()** on a timer stops it immediately, which also forces it to release its strong reference on the view controller it belongs to, thus breaking the strong reference cycle. If you profile again, you'll see all the ImageViewController objects are now transient, and the app should no longer crash.

Wrap up

Hold up your right hand and repeat after me: "I will never ship an app without running it through Instruments first." It doesn't take long, it's not difficult, and I promise it will pay off – your user interfaces will be smoother, your code will run faster, and you'll avoid wasting memory, all using a tool that's completely free and you already have installed.

I have, predictably, only touched briefly on the many features of Instruments here, but I hope I've inspired you to learn more. Instruments can tell you exactly what each CPU core is doing at any given time, it can tell you when every object was created and when it was destroyed along with what code triggered it, and it can even simulate user interface interactions to help you stress test your apps!

This is the end of the last Level 1 project in the series. I hope you've learned a lot, I hope you've made a lot, and I hope you've had a lot of fun along the way. I've worked hard to delete difficult parts, cut corner cases and take out technicalities, so all being well you've found this series productive.

But – and this is a big but – this is not the end of your Swift learning. There's so much more you can do with Swift and iOS that I haven't come close to so far, and they are every bit as exciting as what you've learned so far. To be blunt, I have given you the first 90% of learning iOS development, so it's time for you to step out into the big world and learn the second 90% by yourself.

What's more, there's a third 90%, and a fourth 90%, and so on – the magic of programming is that it's always evolving, so just as you think you're starting to know everything even more stuff comes along to surprise you all over again. Don't let that feeling of joy escape you: program for fun, and you'll never work a day in your life.

If you'd like to continue your learning on Hacking with Swift – and I hope you do! – then you'll be pleased to know you can continue right onto Level 2 projects. These are different from Level 1 because they no longer follow a sequence – now that you've finished all the Level 1 projects, you can tackle the Level 2 projects in any order you want.

Have fun, and happy Swifting!

Project 31

Multibrowser

Get started with UIStackView and see just how easy iPad multitasking is in iOS 9.

Setting up

This is the first chapter of Hacking with Swift Level 2. Chapters 1 to 30 all formed a structured tutorial series teaching you Swift, UIKit, SpriteKit, Auto Layout, MapKit, iBeacons, Core Graphics, Core Image and more all from scratch, using incremental learning and real projects to make the experience fun and productive. That's Level 1, and you should have completed it all. From here on it's Level 2, and projects are designed to be standalone: as long as you have finished projects 1 to 30 already, you can complete any of Level 2 in any order you want.

If you have not already completed projects 1 to 30 of Hacking with Swift, please don't complain if you don't understand a concept! Click here to return to the homepage and browse the list of Swift projects, then work through them in order.

Now for the important stuff: what are we going to build? Well, I want to use this first Level 2 project to show you how to use two great features in iOS 9: **UIStackView** and iPad multitasking. Both of these are stand out technologies in iOS 9, and, remarkably, both are so easy to adopt that we can make this entire project in about 20 minutes. We're also going to touch on Size Classes briefly for the first time, so there's a lot to learn.

The project itself is called Multibrowser, and it shows one or more web views that the user can simultaneously browse. So, you could have one pane with live sports results, one pane with the latest news, and another on Reddit – just like Safari tabs, except they are all visible at the same time. I'll be using **UIWebView** for this project rather than **WKWebView**, but it's easy enough to change in your own project if you want to.

Please go ahead and create a new project in Xcode 7, choosing the Single View Application template. Name it Project 31, choose Swift for your language, and iPad for the device. We're using iPad here because multitasking is only available as an iPad feature.

UIStackView by example

iOS 9 did not introduce many new user interface components, but it did bring us **UIStackView** and that's more than enough – once you understand what it does and how it works, you'll be able to use it in your own apps in just a few minutes, because it's extraordinarily simple.

It's very rare I say this, but Android does have some features that are enviable, and one of them is called **LinearLayout**. When you add views to a **LinearLayout**, they automatically stack up vertically one above the other, or horizontally, side by side. You don't have to worry about sizing them to fit correctly because they automatically fill the space, and you don't have to worry about moving other things around when you remove a view. Well, that's exactly what **UIStackView** gives us: a flexible, Auto Layout-powered view container that makes it even easier to build complex user interfaces.

As an example, lets say you want users to fill in a short form: you have a label saying "Name" then a **UITextField** to the right of it; beneath that you have another label saying "Address" and a **UITextView** beneath it; below that you have a label saying "Opt-in to marketing" and to the right of that a switch; and so on. This isn't uncommon, and to be fair it's not exactly hard to make this in Auto Layout, but you do still need to do a lot grunt work for not much benefit.

The problems usually occur if you want to make changes later: what if there's no longer enough space to show your Name label and its **UITextField** side by side? Previously this would happen if your app was running on iPhone, but with the new multitasking system in iPad it can happen if your user activates Slide Over or uses your app in Split View.

If you missed the WWDC15 keynote:

• **Slide Over** is when your app literally sits over the screen on the right edge, with the original application remaining full screen but dimmed. This is supported on iPad Air, Air 2, Mini and Mini 2.
• **Split View** is activated if the user drags the Slide Over divider to the left slightly, and it causes your app to be pinned to the screen edge while the original application is resized to take up less space. This is supported only on iPad Air 2.

In both these scenarios, your app now has much less space to work with, so your label and text field won't sit well side by side. Fortunately, **UIStackView** can fix this problem: you can tell it to place items side by side when your app has lots of space, or placed vertically when

space is restricted. So, your app will look great on iPhone, iPad, in Slide Over and in Split View, all with a single layout.

What's more, **UIStackViews** can be nested, meaning that you can have stack views inside stack views to create a flexible grid-like layout in no time at all.

In our app, we're going to have a **UIStackView** take up nearly all the screen, and it will host multiple web views inside it. Our interface will also need a **UITextField** in there so users can enter a URL to visit. We'll use Auto Layout to pin these views in place and resize to fill the screen, but that's all.

So, open Main.storyboard in Interface Builder, then embed the existing view controller inside a navigation controller and move it across so you can see it fully. Using the object library, drag a Text Field anywhere into your view, then drag a Horizontal Stack View directly below it. Don't worry about size and position: all that matters is that the text field is above the stack view.
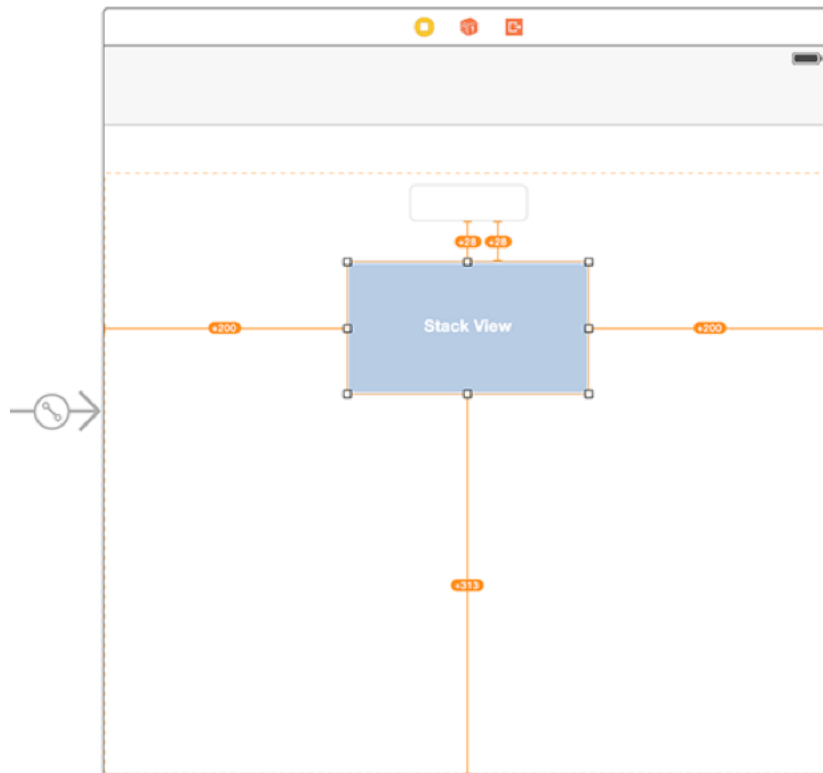
The beginnings of our layout: just place a text field above a horizontal stack view.

With the text field selected, click to bring up the Pin menu. If you've forgotten where this is, it's in the set of buttons at the bottom of the Interface Builder window, to the right "w Any h Any" and looks like a square with two vertical lines either side.

Using the Pin menu, deselect Constrain to Margins, then type 5 in each of the four text boxes. This will color the four pin lines as solid red rather than dashed lines, and you can then click "Add 4 Constraints" at the bottom of the Pin menu to save your changes. If you see "Add 3 Constraints" you might need to click your mouse in a different text box to have Xcode recognize your change.

When you click "Add 4 Constraints", four Auto Layout warnings will appear telling you that your view isn't placed correctly. Ignore them for a moment. Select your stack view, then open up the Pin menu for that. Deselect Constrain to Margins again, then enter the number 5 for the top value and 0 for the left, right and bottom values. Again, click "Add 4 Constraints".

You'll see even more Auto Layout warnings, and your view still looks like a mess. We're going to fix that now: click the Resolve Auto Layout Issues button (one to the right of the Pin menu button), and choose the Update Frames option from under where it says "All Views in View Controller". Boom! Your view will re-layout to match our exact requirements: the text field along the top, and a stack view below, both occupying all the screen. Easy, huh?
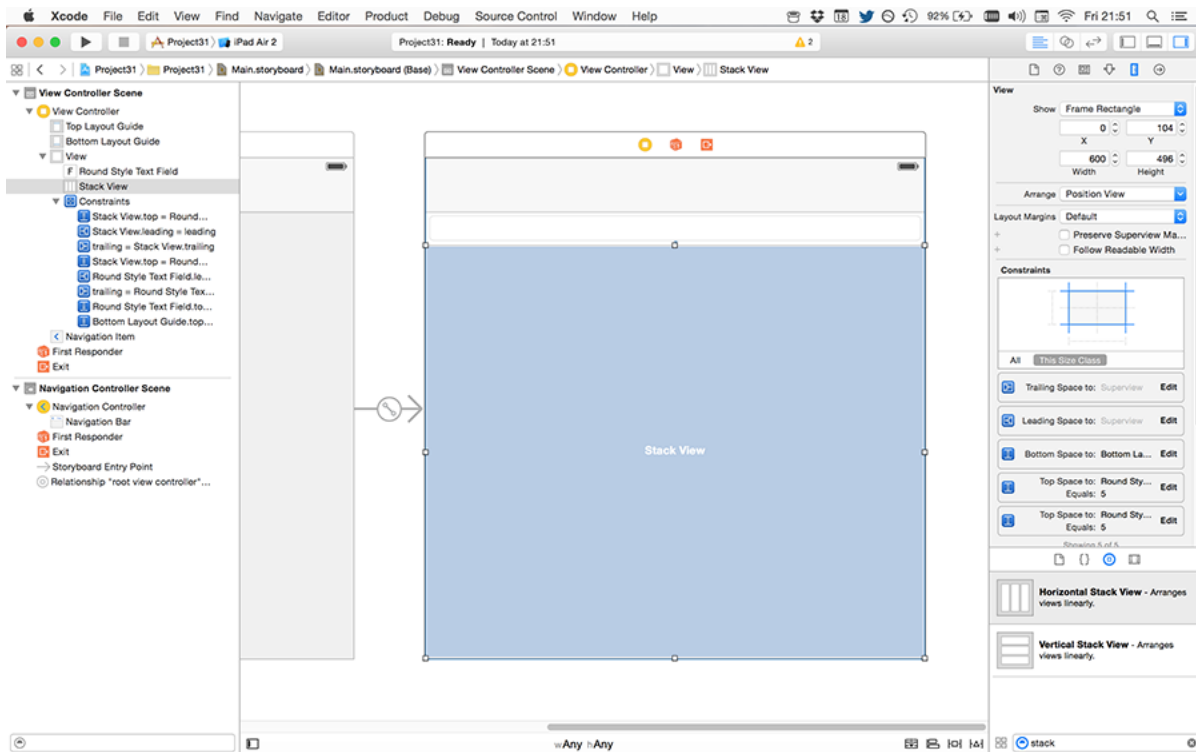
Once Auto Layout rules have been added, the text field and stack view automatically jump into place.

Before we're done with Interface Builder, we're going to make two small changes to the stack view, so make sure it's selected then open the Attributes inspector (Alt+Command+4). From the list of attributes, please change Distribution to Fill Equally, then for Spacing enter 5.

There are a few options for the Distribution attribute, and it's worth covering what they do briefly. If our stack view had four subviews in there, then:

• **Fill** will leave three of them their natural size, and make the fourth one take up the most space. It uses Auto Layout's content hugging priority to decide which one to stretch.
• **Fill Equally** will make each subview the same size so they fill all the space available to the stack view.
• **Fill Proportionally** uses the intrinsic content size of each subview to resize them by an equal amount. So view 1 was designed to have twice as much height as views 2, 3 and 4, that ratio will remain when they are resized – all the subviews get proportionally bigger or smaller.
• **Equal Spacing** does not resize the subviews, and instead resizes the spacing between the subviews to fill the space.
• **Equal Centering** is the most complicated, but for many people also the most aesthetically pleasing. It attempts to ensure the centers of each subview are equally spaced. This might mean that the right edge of view 1 is only 10 points from the left edge of view 2, while the right edge of view 2 is 50 points from the left edge of view 3, but what matters is that the centers of view 1, 2, 3 and 4 are all identically spaced.
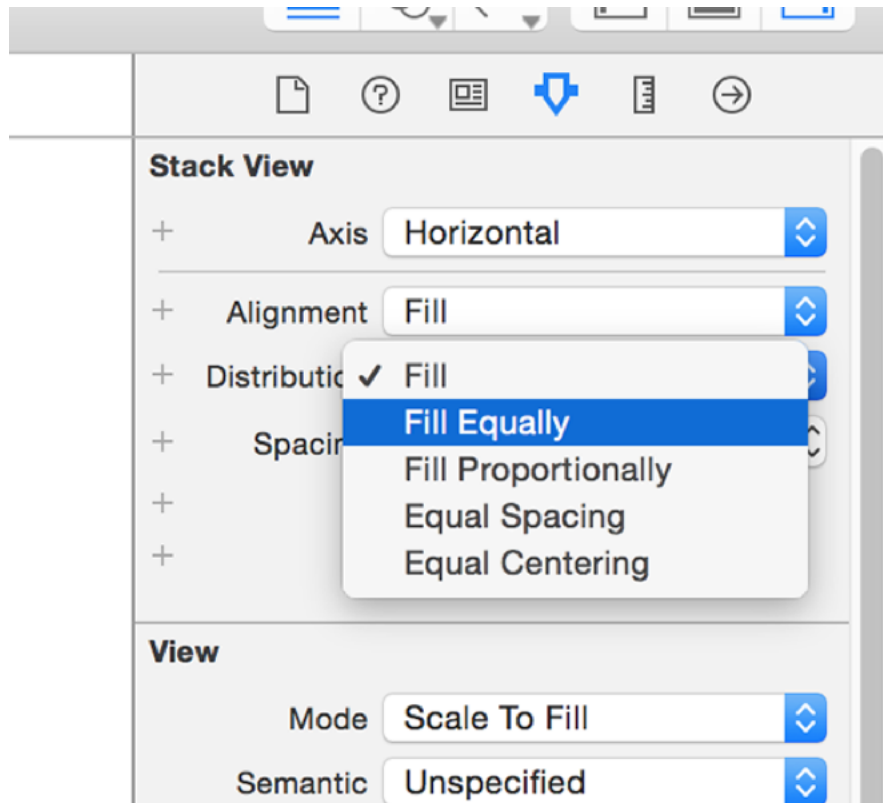
Change your stack view's distribution setting to be Equal to have its subviews take up an equal amount of space.

As for the Spacing attribute, this just determines how much margin to place between items in the stack view. We've set it to 5 here so there's a nice gap between our web views.

The last thing to do is create some connections, so hit Alt+Cmd+Return to go to the Assistant Editor, or click the overlapping circles icon in the top-right of Xcode's window. Now create IBOutlets for the text field and stack view, called **addressBar** and **stackView** respectively. Please also set the view controller to be the delegate of the text field by Ctrl +dragging from the text field to the gold and white View Controller icon in the Document Outline.

We're done with Interface Builder, so press Cmd+Return to return to the Standard Editor, then open ViewController.swift for editing. Time to write some code! And I hope you're ready for just how easy this is going to be…

UIStackView in Swift 2

With our storyboard designed, it's time to write the code. As you know, our plan is to produce an app where the user can have multiple web views visible at one time, stacked together and usable in their own right. We have one address bar, so the user will need to tap a web view to select it, then enter a URL to visit.

To make this interface work, we need two buttons in our navigation bar: one to add a new web view, and one to delete whichever one the user doesn't want any more. We're also going to use the title space in the navigation bar to show the page title of whichever web view is currently active.

So, modify your **viewDidLoad()** method to this:

```
override func viewDidLoad() {

  super.viewDidLoad()


  setDefaultTitle()


  let add = UIBarButtonItem(barButtonSystemItem: .Add, target: self,
action: "addWebView")
  let delete = UIBarButtonItem(barButtonSystemItem: .Trash, target:
self, action: "deleteWebView")
  navigationItem.rightBarButtonItems = [delete, add]

}
```

That uses three new methods we haven't created yet, but only one will create a compile error: **setDefaultTitle()**. This is a fairly simple method in this project, but you're welcome to extend it later to add more interesting information for users prompting them to get started. Put this method directly beneath **viewDidLoad()**:

```
func setDefaultTitle() {
  title = "Multibrowser"
```

```
}
```

The two bar buttons will crash if you try tapping them, but that's OK – we'll fix one now and the other later.

First up, the important stuff: adding a new **UIWebView** to our **UIStackView**. This is done using a method on the stack view called **addArrangedSubview()** and *not* **addSubview()**. That's worth repeating, because it's extremely important: you do not call **addSubview()** on **UIStackView**. The stack view has its own subviews that it manages invisibly to you. Instead, you add to its arranged subviews array, and the stack view will arrange them as needed.

So, our first draft of **addWebView()** is pretty easy: we create a new **UIWebView**, set our view controller to be the web view's delegate, add it to the stack view, then point it at an example URL to get things started.

Here's the code – put this into ViewController.swift, just below **setDefaultTitle()**:

```
func addWebView() {
  let webView = UIWebView()
  webView.delegate = self

  stackView.addArrangedSubview(webView)

  let url = NSURL(string: "https://www.hackingwithswift.com")!
  webView.loadRequest(NSURLRequest(URL: url))
}
```

You can't assign **self** to **webView.delegate** without conforming to the **UIWebViewDelegate** delegate, so please add that. While you're there, you should also add **UITextFieldDelegate** and **UIGestureRecognizerDelegate** – we'll be using these later. So, your view controller's class should start like this:

```
class ViewController: UIViewController, UIWebViewDelegate,
UITextFieldDelegate, UIGestureRecognizerDelegate {
```

Notice that we don't need to give the web view a frame or any Auto Layout constraints – that's all handled for us by **UIStackView**. Remember, iOS apps can only load HTTPS websites by default, and you need to enable App Transport Security exceptions if you want to load non-secure websites. If you want to learn how to do that, **see my guide to App Transport Security**.

Anyway, you can run this project now and try clicking + a couple of times to add new web views, and you'll see them stack up beautifully. Stack views are amazing, right? If you find that the web views aren't appearing correctly, make sure you have set the Distribution attribute of the stack view to be Fill Equally.
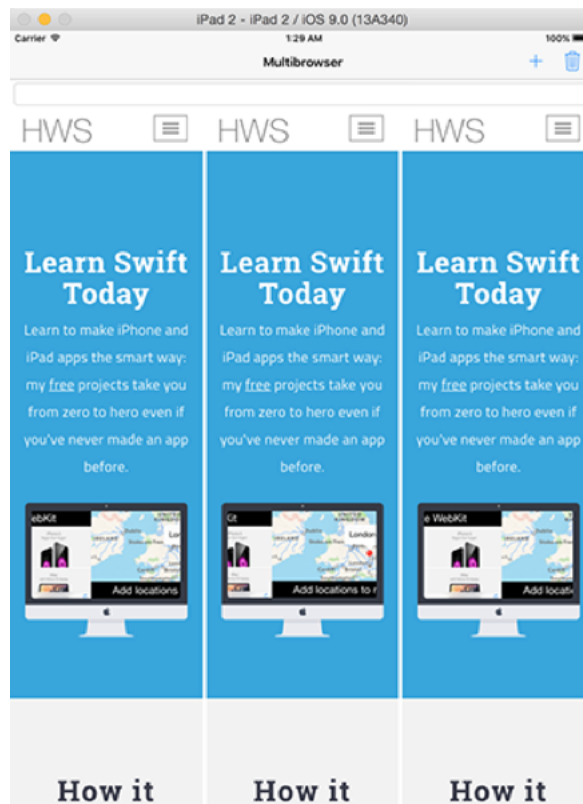
Our app so far: users can add multiple web views, and the UIStackView automatically fits them in equally.

Now that you can see why stack views are perfectly suited to our project, you may notice a major flaw in the plan: how does the user control each web view? And how do they know

which one is currently being controlled?

We're going to fix both these problems at once using something brilliant in its simplicitly: we're going to let users tap on a web view to activate it, then highlight the selected web view in blue so the user knows what's in control. When a web view is activated we also want to show its page title in the navigation bar, and if the user enters a new URL in the address bar it will be loaded inside the active web view.

We're going to draw a blue line around the selected web view so readers can clearly see their current status. To make things easier, we'll draw a blue line around every one of the web views, but because the default line width is 0 it won't be visible until we say so. I'm going to put the code to select web views inside a method so that it can be called when we create a new web view (so that each new web view starts life active) and also when a web view is tapped.

As for handling taps, we'll do that by adding a **UITapGestureRecognizer** to each web view as it's created. This has one minor complication, but it's easily fixed: **UIWebView** already has a pile of gesture recognizers attached to it, and it will catch and consume any taps before our own gesture recognizer. The fix is easy, though, and it's just a matter of telling iOS we want our recognizer and the built-in ones to work at the same time.

So, add this code to the end of **addWebView()**:

```
webView.layer.borderColor = UIColor.blueColor().CGColor

selectWebView(webView)


let recognizer = UITapGestureRecognizer(target: self, action:
"webViewTapped:")

recognizer.delegate = self

webView.addGestureRecognizer(recognizer)
```

We haven't written **selectWebView()** yet, but before we do I just want to recap its job. This method will get called whenever we want to activate a web view, meaning that we want it to be the one used to navigate to any URL the user requests, and we also want it to be highlighted so the user knows which view is in control.

We're going to track the active web view inside a property called **activeWebView**, so add this now:

```
weak var activeWebView: UIWebView?
```

It's **weak** because it might go away at any time if the user deletes it.

With that property created, the **selectWebView()** method is straightforward: it needs to loop through the array of web views belonging to the stack view, updating each of them to have a zero-width border line, then set the newly selected one to have a border width of three points. Here's the code – place it below **addWebView()**:

```
func selectWebView(webView: UIWebView) {
  for view in stackView.arrangedSubviews {
    view.layer.borderWidth = 0
  }

  activeWebView = webView
  webView.layer.borderWidth = 3
}
```

There are two more things to do before our app starts to become useful: we need to implement the **webViewTapped()** method so that our tap gesture recognizers start working, then we need to detect when users have entered a new URL so we can navigate to it.

First up, here's the **webViewTapped()** method that gets called by the tap gesture recognizers when they are triggered:

```
func webViewTapped(recognizer: UITapGestureRecognizer) {
  if let selectedWebView = recognizer.view as? UIWebView {
```

```
    selectWebView(selectedWebView)
  }
}
```

Like I said, you need to tell iOS we want these gesture recognizers to trigger alongside the recognizers built into the **UIWebView**, so add this too:

```
func gestureRecognizer(gestureRecognizer: UIGestureRecognizer,
shouldRecognizeSimultaneouslyWithGestureRecognizer
otherGestureRecognizer: UIGestureRecognizer) -> Bool {

  return true

}
```

We already set our view controller to be the delegate of the **UITapGestureRecognizers** we create for the web views, which means that new method will automatically tell iOS to trigger all gesture recognizers at the same time.

Finally, at least for this chapter, we need to detect when the user enters a new URL in the address bar. We already set this view controller to be the delegate of the address bar, so we'll get sent the **textFieldShouldReturn()** delegate method when the user presses Return on their iPad keyboard. We then need to make sure we have an active web view and that there's a URL to navigate to, and make it happen. We're also going to call **resignFirstResponder()** on the text field so that the keyboard hides.

Put this into your code, below **selectWebView()**:

```
func textFieldShouldReturn(textField: UITextField) -> Bool {
  if let webView = activeWebView, address = addressBar.text {
    if let url = NSURL(string: address) {
      webView.loadRequest(NSURLRequest(URL: url))
    }
  }
```

```
    textField.resignFirstResponder()

    return true

}
```

Notice that there are a few **if/lets** in there to make sure all the data is unwrapped safely, and particularly important is the URL: if you try to enter a URL without https:// iOS will reject it. That's something you can fix later!

At this point your project should compile, although we still haven't added any code to the delete navigation button so don't tap it just yet. You can, though, click + a few times to add some web views, then select one and enter a URL to navigate. Remember: iOS 9 is a bit buggy at this stage, but I tried http://www.reddit.com and http://www.imgur.com and both worked for me. Things are coming together!

Removing views from a UIStackView

That was a long chapter, and I hope you learned a lot. But you deserve a break, so I have some good news: it's trivial to remove views from a UIStackView. Heck, at its simplest it's just a matter of telling **removeArrangedSubview()** which view to remove then removing that view from its superview – the others are automatically resized and re-arranged to fill the space.

In this particular project, we need to do a little more:

• We want the delete button to work only if there's a web view selected.
• We want to find the location of the active web view inside the stack view, then remove it.
• If there are now no more web views, we want to call **setDefaultTitle()** to reset the user interface.
• We need to find whatever web view immediately follows the one that was removed.
• We then make that the new selected web view, highlighting it in blue.

We already pointed the delete button at a method called **deleteWebView()**, so all you need to do is plug this in. I've added comments to make sure it's all clear:

```
func deleteWebView() {
  // safely unwrap our webview
  if let webView = activeWebView {
    if let index = stackView.arrangedSubviews.indexOf(webView) {
      // we found the current webview in the stack view! Remove it
from the stack view
      stackView.removeArrangedSubview(webView)

      // now remove it from the view hierarchy – this is important!
      webView.removeFromSuperview()

      if stackView.arrangedSubviews.count == 0 {
        // go back to our default UI
        setDefaultTitle()
```

```
    } else {
        // convert the Index value into an integer
        var currentIndex = Int(index)


        // if that was the last web view in the stack, go back one
        if currentIndex == stackView.arrangedSubviews.count {
            currentIndex = stackView.arrangedSubviews.count - 1
        }


        // find the web view at the new index and select it
        if let newSelectedWebView =
stackView.arrangedSubviews[currentIndex] as? UIWebView {
            selectWebView(newSelectedWebView)
        }
    }
  }
}
```

So, although the act of removing a view from a **UIStackView** is just a matter of calling
removeArrangedSubview() and **removeFromSuperview()**, we need to do a little more to
make sure the user interface updates correctly.

You might be wondering why **removeFromSuperview()** is required when we're already
calling **removeArrangedSubview()**. The reason is that you can remove something from a
stack view's arranged subview list then re-add it later, without having to recreate it each time
– it was hidden, not destroyed. We don't want a memory leak, so we want to remove deleted
web views entirely. If you find your memory usage ballooning, you probably forgot this step!

The last thing we're going to do is talk about the new iPad multitasking features in iOS 9, and
add a few user interface clean ups to make the project complete…

iPad multitasking in iOS 9

Multitasking is an all-new feature in iOS 9, and I can guarantee you it's going to be adopted pretty much everywhere. Now is the time to get started upgrading your apps so that you're ready, otherwise you'll have your users complaining that they can't multitask with your app!

You'll be pleased to know that supporting multitasking is easy. In fact, it's so easy that our current app *already supports it*. Don't believe me? Try it out now: launch your app, rotate the simulator to landscape (Cmd + left or right cursor key), then drag from the right edge of the screen.

The first time you do this, you'll see a list of various apps to choose from. Please choose Calendar for now. When you do this, iOS will activate Slide Over, which means your app still owns the full screen, but it's dimmed as the Calendar app has focus in the right part of the screen. On the left edge of Calendar you'll see a thin white line, which is the divider – drag that a little to the left and you'll see the whole interface change as iOS switches from Slide Over to Split View.

Now, the reason I asked you to change the simulator to landscape mode is because Split View actually has two snap points. The first, which you probably triggered this time, has your original app taking up about 2/3rds of the screen on the left and Calendar taking up the remainder on the right. The second, which you can get to by dragging the divider into the center of the screen, has both apps taking up half the screen each. If you're in portrait orientation you have only one mode, which is about 60/40.

So, our app already supports multitasking pretty well, although we'll make it better in a moment. First, though: what if you're upgrading existing apps? Well, you might not have such an easy ride, but if your code is modern you're probably still OK. To make multitasking work, you need to:

1. Have a launch XIB. This is the same thing that enables iPhone 6 support with iOS 8, so you might already have one. If not, add a new file, choose User Interface, then Launch Screen. Then, in your plist, add a key for "Launch screen interface file base name" and point it to the name of your launch XIB, without the ".xib" extension. For example, if your launch screen is called LaunchScreen.xib, give this key the value of "LaunchScreen".
2. Make sure your app is configured to support all orientations. This may already be configured this way, but if not make sure you choose all orientations now. As you might imagine, selectively choosing only some orientations would cause havoc with multitasking!
3. Use Auto Layout everywhere. If your app pre-dates Auto Layout or if you found it

intimidating at first, you might still be struggling along with autoresizing masks. Now is the time to change: the various multitasking sizes make Auto Layout a no-brainer.
4. Use adaptive UI wherever needed. Adaptive layout is Apple's term for technologies like Size Classes and Dynamic Type, the former of which is a huge advantage when working with multitasking. Size Classes let you make your interface adjust to two major screen sizes, compact and regular, which previously were great for working with iPhone and iPad, but are now also used for iPad multitasking.

Even though this particular project works with multitasking by default, it doesn't have any adaptive user interface built in. As a result, if we use multitasking the *other* way – i.e., if it's our app that is the one occupying 1/3rd of the screen while some other app has the remainder – then it looks terrible: our vertically stacked web views end up being so thin that they are unusable.

The solution is simple: we're going to tell the stack view to arrange its web views horizontally when we have lots of space, and vertically when we don't. This is done using the **traitCollectionDidChange()** method, which gets called automatically when our app's size class has changed. We can then query which size class we now have, and adapt our user interface.

There is one complication, and that's understanding size classes. There are two axes for size classes, namely horizontal and vertical, and each of them has two sizes, Compact and Regular. No matter what orientation or multitasking setup, the vertical size class is always regular on iPad. For the other possibilities, here are the key rules:

 • An iPad app running by itself in portrait or landscape has a regular horizontal size classes.
 • In landscape where the apps are split 50/50, both are running in a compact horizontal size class.
 • In landscape where the apps are split 70/30, the app on the left is a regular horizontal size class and the app on the right is compact.
 • In portrait where the apps are split 60/40, both are running in a compact horizontal size class.

We're going to use this information so that we detect when the size class has changed and update our stack view appropriately. When we have a regular horizontal size class we'll use horizontal stacking, and when we have a compact size class we'll use vertical stacking. Here's the code:

```
override func traitCollectionDidChange(previousTraitCollection:
UITraitCollection?) {

  if traitCollection.horizontalSizeClass == .Compact {

    stackView.axis = .Vertical

  } else {

    stackView.axis = .Horizontal

  }

}
```

The project is technically finished at this point, but we're going to do two more things just to make it a bit more polished. First, we're going to create a method that updates the navigation bar to show the page title from the active web view when it changes. This will use the **stringByEvaluatingJavaScriptFromString()** method to execute **document.title** and pull out the page's title. This compares very poorly to **WKWebView's title** property, so you're welcome to try converting your app to use that instead!

Here's that method:

```
func updateUIUsingWebView(webView: UIWebView) {

  title =
webView.stringByEvaluatingJavaScriptFromString("document.title")

  addressBar.text = webView.request?.URL?.absoluteString ?? ""

}
```

Second, we need to call that method in the two places it's needed: whenever we select a web view, and whenever the web view changes page. The former is just a matter of adding this line just before the end of **selectWebView()**:

```
updateUIUsingWebView(webView)
```

The latter is a matter of implementing the **webViewDidFinishLoad()** method, which we can

receive because we configured our view controller to be the delegate of each of the web views. So, put this code somewhere in ViewController.swift:

```
func webViewDidFinishLoad(webView: UIWebView) {
  if webView == activeWebView {
    updateUIUsingWebView(webView)
  }
}
```

As you can see, we only update the user interface to reflect a page's title if it comes from the active web view, otherwise it would be confusing. That's it – the project is done!

Wrap up

With **UIStackView** in place that's another UIKit component under your belt – good job! The addition of multitasking and size classes helps make the app much more polished, and I think you've got the seeds of a great app here.

If you want to try extending this project, the sensible place to start is in the URL entry: if users don't type "http://" before their web site addresses the app fails, which isn't very helpful. Another smart place to improve the app is inside the **setDefaultTitle()** method: it just writes Multibrowser in the navigation bar while leaving a large white space in the center – hardly intuitive, and it wouldn't be hard to add a placeholder label in there telling users what to do.

I do intend to do more iOS 9 projects in the future, so please let me know what you'd like to see: I'm **@twostraws on Twitter** and I'd love to hear from you. In the meantime, why not read my **guide to the new features in iOS 9** and my other **guide to the new features in Swift 2.0**. Have fun!

Project 32

SwiftSearcher

Add your app's content to iOS Spotlight and take
advantage of the new Safari integration from iOS 9.

Setting up

I hope you're all set for a massive Level 2 project, because this one is going to cram a lot in. In this project, you're going to make an app for this tutorial series, Hacking with Swift – very meta, I know. The app is going to list all the projects and let users choose which ones they favorite, which by itself sounds like the kind of thing we might have done in project 7 or so.

So home come it's a Level 2 project? Simple: when users favorite a project, we're going to store that in Core Spotlight so they can find it later. And when they view a project, we'll use the new **SFSafariViewController** class for them to read. If those two new features from iOS 9 weren't enough for you, I'm going to throw in a little bit of **UITableViewCell** automatic sizing, a little bit of Dynamic Type and even some **NSAttributedString** to handle formatted string drawing.

Are you feeling it now? I hope so. But if all this talk of Level 2 is confusing you, that probably means you haven't completed projects 1 to 30 already, and should **click here to browse the projects list**. Those first 30 projects comprise Hacking with Swift Level 1 and you really need to have completed them before tackling projects 31 onwards.

Still here? Great! Make sure you have Xcode 7 installed, because we'll be using iOS 9 in this tutorial. If you missed my summaries of **what's new in Swift 2.0** and **new features in iOS 9 for developers** click one of those two links.

Please go ahead and create a new project in Xcode 7, choosing the Master-Detail Application template. Name it Project32, choose Swift for your language, and iPhone for the device.

Automatically resizing UITableViewCells with Dynamic Type and NSAttributedString

We're going to make a UITableView with formatted text that matches a user's preferred size, and where every cell automatically resizes to fit its contents. What's more, it's going to be so easy that you'll barely notice – Apple really has polished this technology, so you get an incredible amount of power for free.

First, we need to make two small changes to the default storyboard, so please open Main.storyboard in Interface Builder. Using the Document Outline, select and delete "Detail Scene" and "Navigation Controller Scene". Now select and delete DetailViewController.swift into the Project Navigator pane on the left of your Xcode window.

What you just did was delete the built-in segue from master controller to detail controller, so you can expect a few errors. This might seem a strange thing to do given that we want to list Hacking with Swift projects and let users choose one – a perfect example of a master/detail layout – but actually things are a bit more complicated here. You see, we're going to be using the new **SFSafariViewController** class, which is *not* part of UIKit and so not part of Interface Builder. Showing one of these is something that's best done in pure Swift, which is why we removed stuff from the storyboard.

Now get out your shears, choose MasterViewController.swift, and delete the following:

 • **var detailViewController: DetailViewController? = nil** • In **viewDidLoad()** everything but the call to **super.viewDidLoad()** • All of **viewWillAppear()** • All of **insertNewObject()** • All of **prepareForSegue()** • From **tableView(_:cellForRowAtIndexPath:)** the two lines **let object = objects[indexPath.row] as! NSDate** and **cell.textLabel!.text = object.description** • All of **tableView(_:canEditRowAtIndexPath:)** • The contents of **tableView(_:commitEditingStyle:forRowAtIndexPath:)** but not the method itself

I don't know about you, but I rather like deleting source code – it's cathartic, and gives us a nice, clean foundation to work from!

If you've followed those instructions, your MasterViewController.swift file should look like this:

```swift
import UIKit


class MasterViewController: UITableViewController {

  var objects = [AnyObject]()


  override func viewDidLoad() {

    super.viewDidLoad()

  }


  override func didReceiveMemoryWarning() {

    super.didReceiveMemoryWarning()

    // Dispose of any resources that can be recreated.

  }


  // MARK: - Table View


  override func numberOfSectionsInTableView(tableView: UITableView) -
> Int {

    return 1

  }


  override func tableView(tableView: UITableView,
numberOfRowsInSection section: Int) -> Int {

    return objects.count

  }


  override func tableView(tableView: UITableView,
cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {

    let cell = tableView.dequeueReusableCellWithIdentifier("Cell",
forIndexPath: indexPath)

    return cell

  }
```

```
  override func tableView(tableView: UITableView, commitEditingStyle
editingStyle: UITableViewCellEditingStyle, forRowAtIndexPath
indexPath: NSIndexPath) {

  }

}
```

What we're going to do now is replace the **objects** array with a new array called **projects**.
There are lots of pieces of metadata we could store about Hacking with Swift projects, but
the goal here isn't to teach you about custom subclasses – we already did that in the tutorial
on NSUserDefaults and NSCoding. So, we're going to take an epic shortcut here so we can
spend more time focusing on the new stuff: the projects array will hold an array of String
arrays.

Arrays within arrays aren't complicated, but I'm just going to clarify in case some peope
didn't quite understand. Each project will be stored as an array of two elements: the project
name and its subtitle. We'll then store an array of those to contain all the projects. At the end
of this project we'll return to this as homework to test your skills, but for the purpose of this
project it's perfectly fine.

So, remove the **objects** property and replace it with this:

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

When you do this, you'll get an error in **numberOfRowsInSection** because we removed the
objects array, so just change it to use **projects** instead:

```
override func tableView(tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {

  return projects.count

}
```

And now let's fill up that new property with the first eight projects from Hacking with Swift. I would do more, but there's no point – you get the idea, and it just takes up more space on your screen! Put this at the end of your **viewDidLoad()** method:

```
projects.append(["Project 1: Storm Viewer", "Constants and variables,
UITableView, UIImageView, NSFileManager, storyboards"])

projects.append(["Project 2: Guess the Flag", "@2x and @3x images,
asset catalogs, integers, doubles, floats, operators (+= and -=),
UIButton, enums, CALayer, UIColor, random numbers, actions, string
interpolation, UIAlertController"])

projects.append(["Project 3: Social Media", "UIBarButtonItem,
UIActivityViewController, the Social framework, NSURL"])

projects.append(["Project 4: Easy Browser", "loadView(), WKWebView,
delegation, classes and structs, NSURLRequest, UIToolbar,
UIProgressView., key-value observing"])

projects.append(["Project 5: Word Scramble", "NSString, closures,
method return values, booleans, NSRange"])

projects.append(["Project 6: Auto Layout", "Get to grips with Auto
Layout using practical examples and code"])

projects.append(["Project 7: Whitehouse Petitions", "JSON, NSData,
UITabBarController"])

projects.append(["Project 8: 7 Swifty Words", "addTarget(),
enumerate(), countElements(), find(), join(), property observers,
range operators."])
```

So far, so easy. The next step is hardly challenging either: let's show each project's title and subtitle in the table view cells. We're going to do this in a very basic way at first, but it's enough to get you started. We need to modify **cellForRowAtIndexPath** so that it shows each project's title and subtitle using Swift's string interpolation. Here's the new code:

```
override func tableView(tableView: UITableView, cellForRowAtIndexPath
indexPath: NSIndexPath) -> UITableViewCell {

  let cell = tableView.dequeueReusableCellWithIdentifier("Cell",
forIndexPath: indexPath)


  let project = projects[indexPath.row]
```

```
cell.textLabel?.text = "\(project[0]): \(project[1])"


return cell
}
```

Your project is now good to run, but what you'll see is deeply unsatisfying: each of the table view cells shows only one line of text, so our summaries get truncated after only a few letters, making them rather pointless.
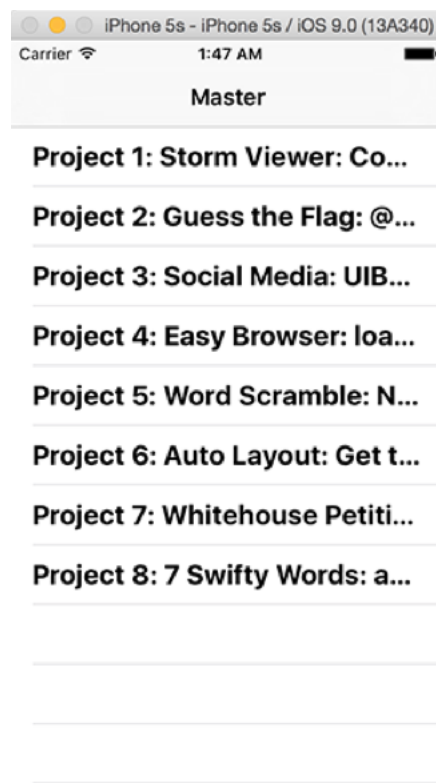
The first version of our app shows the project name and title on the same line, so you can't see very much.

Fortunately, iOS 8 introduced a technology that developers had been requesting for a long time: automatic sizing of UITableViewCells based on their contents. Even better, this technology is already baked right into our project!

To make our project titles and subtitles fully visible, we just need to tell the UITableViewCell

that its label should show more than one line. Go back to Main.storyboard, look for the "Master" scene, and click on the table view cell where it says Title. You may need to click multiple times because the label is several levels deep – you'll know when you have the correct thing selected because the Attributes inspector will say Label at the top.

Once the label is selected, look for the Lines property – it will be 1 by default, but you should change that to 0, which means "as many lines as it takes to fit the text."

And now for the hard part: telling the table view to automatically figure out the size for every cell based on its text. Just kidding: this bit is laughably simple, because Auto Layout does all the hard work for you.

Go back to MasterViewController.swift and add these two new methods:

```swift
override func tableView(tableView: UITableView,
heightForRowAtIndexPath indexPath: NSIndexPath) -> CGFloat {

    return UITableViewAutomaticDimension

}


override func tableView(tableView: UITableView,
estimatedHeightForRowAtIndexPath indexPath: NSIndexPath) -> CGFloat {

    return UITableViewAutomaticDimension

}
```

That's it. No, really: go ahead and run your app now and you'll see every cell now correctly fits its content. And if you rotate between portrait and landscape you'll even see everything resize smoothly – it really couldn't be simpler.
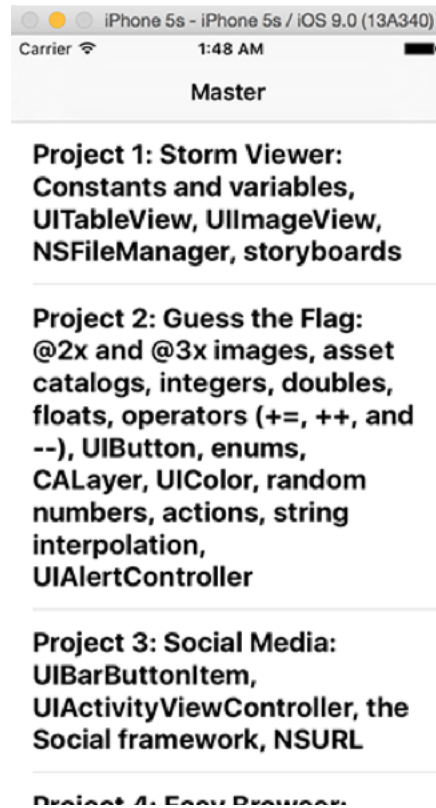
The second version of our app now wraps text across multiple lines, so at least you can see what the text says.

But this has exposed another problem: the app looks terrible! All that bold text doesn't help users see what's important and what's not, so it's terrible user interface design. We're going to fix this with a technology that was introduced way back in iOS 6: **NSAttributedString**. This is a way of adding formatting such as fonts, colors and alignment to text, and can even be used to add hyperlinks if you want them.

In our case, we're going to make the project titles big and their subtitles small. We could do this by creating a **UIFont** at various sizes, but a much smarter (and user friendly!) way is used to a technology called Dynamic Type. This was introduced in iOS 7 as a way to let users control font size across all applications to match their preferences.

Apple pre-defined a set of fonts for use with Dynamic Type, all highly optimized for readability, and all responsive to a user's settings. To use them, all you need to do is use the **preferredFontForTextStyle()** method of **UIFont** and tell it what style you want. In our case we're going to use **UIFontTextStyleHeadline** for the project title and **UIFontTextStyleSubheadline** for the project subtitle.

Remarkably enough, that's all you need to handle Dynamic Type in this project, so we can turn to look at **NSAttributedString**. Like I said, this class is designed to show text with formatting, and you can use it all across iOS to show formatted labels, buttons, navigation bar titles, and more. You create an attributed string by giving it a plain text string plus a dictionary of the attributes you want to set. If you want finer-grained control you can provide specific ranges for formatting, e.g. "bold and underline the first 10 characters, then underline everything else."

For this project, our use of attributed strings isn't complicated: we're going to create one set of formatting attributes for the title and another for the subtitle, create an attributed string out of both of them, then merge them together and return. To make things easier to read for the user, we're going to separate the title and subtitle with a line break, which looks a lot nicer.

To keep our code easy to understand, I put the attributed string work into a new method called **makeAttributedString()**. Here's the code:

```
func makeAttributedString(title title: String, subtitle: String) ->
NSAttributedString {

  let titleAttributes = [NSFontAttributeName:
UIFont.preferredFontForTextStyle(UIFontTextStyleHeadline),
NSForegroundColorAttributeName: UIColor.purpleColor()]

  let subtitleAttributes = [NSFontAttributeName:
UIFont.preferredFontForTextStyle(UIFontTextStyleSubheadline)]


  let titleString = NSMutableAttributedString(string: "\(title)\n",
attributes: titleAttributes)

  let subtitleString = NSAttributedString(string: subtitle,
attributes: subtitleAttributes)


  titleString.appendAttributedString(subtitleString)


  return titleString

}
```

There are some new things in there, so let's go over them quickly:

 • **NSFontAttributeName** is the dictionary key that specifies what font the attributed text should use. This should be provided with a **UIFont** as its value, and like I said already we're using **preferredFontForTextStyle()** so we can take advantage of Dynamic Type.
 • **NSForegroundColorAttributeName** is the dictionary key that specifies what text color to use. This isn't needed in this project, but I figured it would be boring to have only attribute!
 • Both attributed strings are created by providing a plain text string and the matching dictionary of attributes. The title string is created as a **NSMutableAttributedString** because we append the subtitle to the title to make one attributed string that can be returned.

The last piece of the puzzle is to use the return value from **makeAttributedString()** inside **cellForRowAtIndexPath** so that our interface looks better. This is just a matter of setting the **attributedText** property rather than the **text** property of the cell's text label, like this:

```
override func tableView(tableView: UITableView, cellForRowAtIndexPath
indexPath: NSIndexPath) -> UITableViewCell {

  let cell = tableView.dequeueReusableCellWithIdentifier("Cell",
forIndexPath: indexPath)


  let project = projects[indexPath.row]

  cell.textLabel?.attributedText = makeAttributedString(title:
project[0], subtitle: project[1])


  return cell

}
```

That's it! Go ahead and run the app now and admire your purple headlines!
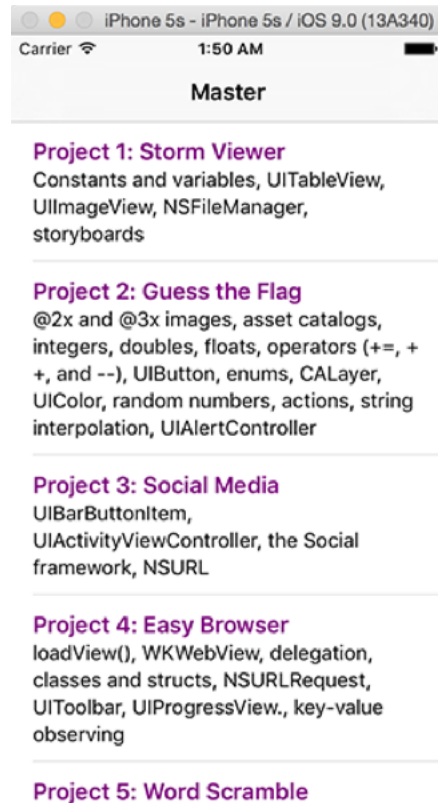
Using multiple lines of text and an attributed string makes our user interface look a lot nicer.

How to use SFSafariViewController to browse a web page

You just learned about automatic cell resizing, **NSAttributedString** and Dynamic Type, so you deserve a pat on the back. But there's more: I want to introduce you to one of the new view controllers in iOS 9, and you'll be pleased to know that, once again, Apple has made it startlingly simple to use.

When a user taps on one of our table rows, we want to show the Hacking with Swift project that matches their selection. In Ye Olden Days we would do this either with **UIWebView** or **WKWebView**, adding our own user interface to handle navigation. But this had a few problems: everyone's user interface was different, features such as cookies and Auto Fill were unavailable for security reasons, and inevitably users looked for an "Open in Safari" button because that was what they trusted.

Apple fixed all these problems in iOS 9 using a new class called **SFSafariViewController**, which effectively embeds all of Safari inside your app using an opaque view controller. That is, you can't style it, you can't interact with it, and you certainly can't pull any private data out of it, and as a result **SFSafariViewController** can take advantage of the user's secure web data in ways that **UIWebView** and **WKWebView** never could.

What's more, Apple builds the new iOS 9 power features right into **SFSafariViewController**, so you get things like content blocking free of charge – and users get consistent features, consistent UI, and consistent security. Everybody wins!

SFSafariViewController is not part of UIKit, so you need to import a new framework to use it. Add this to the existing **import UIKit** line at the top of MasterViewController.swift:

```
import SafariServices
```

We're going to create a method that accepts an integer and shows the matching tutorial. All the Hacking with Swift tutorials are numbered from 1 upwards, so we can match that up to our **projects** array (which is zero-based) just by adding 1. We'll convert that to an **NSURL** then pass that to a new **SFSafariViewController** to show to the user.

When working with **SFSafariViewController** there are two things you need to know. First,

you can either create it just with a URL or with a URL and the instruction to use reader mode if available. "Reader mode" is Apple's name for a text-only view of web pages. This doesn't work on Hacking with Swift, but I'm including it here so you can see how it works.

Second, the **SFSafariViewController** is dismissed when a user taps a Done button in its user interface. This calls a **safariViewControllerDidFinish()** method on the delegate of the **SFSafariViewController**, which you can use to run any code to handle picking up where the user left off. We won't be using it here, but if you want it in your own projects make sure you conform to the **SFSafariViewControllerDelegate** protocol.

Bringing all that together, let's write some code. Go ahead and add this new method somewhere in the class:

```
func showTutorial(which: Int) {

  if let url = NSURL(string: "https://www.hackingwithswift.com/read/\
(which + 1)") {

    let vc = SFSafariViewController(URL: url,
entersReaderIfAvailable: true)

    presentViewController(vc, animated: true, completion: nil)

  }

}
```

You can see how easy it is to control reader mode – just set the **entersReaderIfAvailable** flag to be true or false as needed.

There's only one more thing to do to finish this stage of the project: when any table row is tapped, we need to call that new **showTutorial()** method and pass in the index path of the row so the correct tutorial can be shown. This is as simple as adding a **didSelectRowAtIndexPath** method like this:

```
override func tableView(tableView: UITableView,
didSelectRowAtIndexPath indexPath: NSIndexPath) {

  showTutorial(indexPath.row)

}
```

That's it for the new **SFSafariViewController** – easy, huh?

How to add Core Spotlight to index your app content

One of the most important additions in iOS 9 is the ability for apps to communicate bidirectionally with Spotlight, the iOS system-wide search feature. What this means is that iOS 9 apps can ask for their content to be shown in the Spotlight search results and, if the user taps one of those search results, the app gets launched and told what was tapped so it can load the right content.

In this project, we're going to have users favorite the Hacking with Swift projects that most interest them. When they do that, we'll store the project title and subtitle in Spotlight so they can search for things like "wk" and find the **WKWebView** tutorial that is project 4.

We're going to tackle this problem in three stages: updating the user interface to reflect saved favorites, adding and removing items from Core Spotlight, then responding to deep links when our app is launched from a search result in Spotlight.

First up: creating a user interface that lets user favorite and unfavorite projects, and saving those choices. There are various ways of doing this, but I've chosen the simplest: we're going to set the table to be in editing mode, use the "Insert" and "Delete" icons to let users select their favorites, and use a checkmark accessory type to show which projects are already favorited.

Behind the scenes we'll also need an array of **Ints** that tracks which project numbers are currently favorited, and that will be saved to **NSUserDefaults** whenever a change is made.

That's all we have to do in theory, but in practise there are two catches:

 • When a table is in editing mode, you can't tap the cells any more. Given that this is our way of reading the projects, that's a big problem! Fortunately, we can just set the **allowsSelectionDuringEditing** property to true to fix this.
 • When a table is in editing mode, you can't just set the **accessoryType** because that isn't shown. Instead, you need to set **editingAccessoryType**, which functions the same but is visible while editing.

Let's do the easy stuff first: add this property to your class:

```
var favorites = [Int]()
```

We need to load that from **NSUserDefaults** if it exists there already, which means using **if/let** to conditionally unwrap the result of **objectForKey()** as an **Int** array. Put this just before the end of **viewDidLoad()**:

```
let defaults = NSUserDefaults.standardUserDefaults()

if let savedFavorites = defaults.objectForKey("favorites") as? [Int]
{

    favorites = savedFavorites

}
```

To make the list of favorites work in our user interface, we need to add two more lines to **viewDidLoad()**. Like I said already, these set the table view to be in editing mode, and tell it to let users tap on rows to select them. Add these lines now:

```
tableView.editing = true

tableView.allowsSelectionDuringEditing = true
```

Next we need to update **cellForRowAtIndexPath** so that cells show a checkmark if they exist in the **favorites** array, or nothing otherwise. This is done just by using the **contains()** method of the **favorites** array, like this:

```
if favorites.contains(indexPath.row) {

    cell.editingAccessoryType = .Checkmark

} else {

    cell.editingAccessoryType = .None

}
```

You should put that inside **cellForRowAtIndexPath**, just before the **return cell** line.

Now it's just a matter of telling the table view that some rows should have the "insert" icon and others the "delete" icon. To do that, you just need to implement the **editingStyleForRowAtIndexPath** method and check whether the item in question is in the **favorites** array. Put this into your class:

```
override func tableView(tableView: UITableView,
editingStyleForRowAtIndexPath indexPath: NSIndexPath) ->
UITableViewCellEditingStyle {

  if favorites.contains(indexPath.row) {

    return .Delete

  } else {

    return .Insert

  }

}
```

If you run the app now all the rows will have a green + symbol to their left and no checkmark on the right, because no projects have been marked as a favorite. If you click the + nothing will happen, because we haven't told the app what to do in that situation. To make this work, we need to handle the **tableView(_:commitEditingStyle:forRowAtIndexPath:)** method, checking whether the user is trying to insert or delete their favorite.

If the user is adding a favorite, we're going to call a method called **indexItem()** that we'll write in a moment. We'll also add it to the **favorites** array, save it to **NSUserDefaults** then reload the table to reflect the change. If they are deleting a favorite, we do pretty much the opposite: call **deindexItem()** (also not yet written), remove it from the favorites array, save that array and reload the table.

There's one small catch here, which is that removing an item from an array requires you to know its position in the array. We don't know the position of a project in the **favorites** array because they can add any projects they want – the array could contain 5, 2, 4, for example. We'll solve this by using the **indexOf()** method to find the position of a project number in the **favorites** array, then use that index to remove it.

Here's new code for the **commitEditingStyle** method that's currently empty, plus stubs for **indexItem()** and **deindexItem()**:

```
override func tableView(tableView: UITableView, commitEditingStyle
editingStyle: UITableViewCellEditingStyle, forRowAtIndexPath
indexPath: NSIndexPath) {

  if editingStyle == .Insert {

    favorites.append(indexPath.row)

    indexItem(indexPath.row)

  } else {

    if let index = favorites.indexOf(indexPath.row) {

      favorites.removeAtIndex(index)

      deindexItem(indexPath.row)

    }

  }


  let defaults = NSUserDefaults.standardUserDefaults()

  defaults.setObject(favorites, forKey: "favorites")


  tableView.reloadRowsAtIndexPaths([indexPath],
withRowAnimation: .None)

}


func indexItem(which: Int) {

}


func deindexItem(which: Int) {

}
```

You should replace your existing **commitEditingStyle** method with that new one, but the other two are new.

OK, that's our first stage complete: the user user interface now updates to reflect saved favorites. You could give it a try now if you really wanted, but I suggest you don't to avoid confusing yourself later on. The next stage is adding and removing items from Core Spotlight, which means filling out those **indexItem()** and **deindexItem()** methods that get called when favorites are added and deleted.

Using Core Spotlight means importing two extra frameworks: CoreSpotlight and MobileCoreServices. The former does all the heavy lifting of indexing items; the latter is just there to identify what type of data we want to store. So, import these two now:

```
import CoreSpotlight
import MobileCoreServices
```

Now for the new stuff: **indexItem()** accepts an **Int** identifying which project has been favorited. It needs to look inside the **projects** array to find that project, then create a **CSSearchableItemAttributeSet** object from it. This attribute set can store lots of information for search, including a title, description and image, as well as use-specific information such as dates (for events), camera focal length and flash setting (for photos), latitude and longitude (for places), and much more.

Regardless of what you choose, you wrap up the attribute set inside a **CSSearchableItem** object, which contains a unique identifier and a domain identifier. The former must identify the item absolutely uniquely inside your app, but the latter is a way to group items together. Grouping items is how you get to say "delete all indexed items from group X" if you choose to, but in our case we'll just use "com.hackingwithswift" because we don't need grouping. As for the unique identifier, we can use the project number.

To index an item, you need to call **indexSearchableItems()** on the default searchable index of **CSSearchableIndex**, passing in an array of **CSSearchableItem** objects. This method runs asynchronously, so we're going to use a trailing closure to be told whether the indexing was successful or not.

Here's the code:

```
func indexItem(which: Int) {

  let project = projects[which]


  let attributeSet = CSSearchableItemAttributeSet(itemContentType:
kUTTypeText as String)

  attributeSet.title = project[0]

  attributeSet.contentDescription = project[1]


  let item = CSSearchableItem(uniqueIdentifier: "\(which)",
domainIdentifier: "com.hackingwithswift", attributeSet: attributeSet)


CSSearchableIndex.defaultSearchableIndex().indexSearchableItems([item
]) { (error: NSError?) -> Void in

    if let error = error {

      print("Indexing error: \(error.localizedDescription)")

    } else {

      print("Search item successfully indexed!")

    }

  }

}
```

The only thing in there that I haven't explained is **kUTTypeText as String**, which tells iOS we want to store text in our indexed record.

By default, content you index has an expiration date of one month after you add it. This is probably OK for most purposes (although you do need to make sure you re-index items when your app runs in case they have expired!), but you can change the expiration date if you want. It's not something that can easily be tested, but this kind of code probably works to make your items never expire:

```
let item = CSSearchableItem(uniqueIdentifier: "\(which)",
domainIdentifier: "com.hackingwithswift", attributeSet: attributeSet)

item.expirationDate = NSDate.distantFuture()
```

The last thing we need to do is fill in the **deindexItem()** method, which is very similar to the **indexItem()** in that it receives an **Int**, calls a method on the default searchable index of **CSSearchableIndex**, then has a trailing closure to handle error reporting. Here's the code:

```
func deindexItem(which: Int) {

CSSearchableIndex.defaultSearchableIndex().deleteSearchableItemsWithI
dentifiers(["\(which)"]) { (error: NSError?) -> Void in

    if let error = error {

      print("Deindexing error: \(error.localizedDescription)")

    } else {

      print("Search item successfully removed!")

    }

  }
}
```

With that, the second stage of our Core Spotlight integration is complete: adding and removing items works! That just leaves the final stage, which is responding to deep links when our app is launched from a search result in Spotlight.

Now that we are indexing our content in Spotlight, users can search for our projects and tap on results. This will launch our app and pass in the unique identifier of the item that was tapped, and it's down to the app to do something with it. This is all done using in an **AppDelegate.swift** method called **application(_:continueUserActivity:restorationHandler:)**, with the important part being what's given to us as the **continueUserActivity** parameter.

This app delegate method is called when the application has finished launching and it's time to launch the activity requested by the user. If the user activity has the type **CSSearchableItemActionType** it means we're being launched as a result of a Spotlight search, so we need to unwrap the value of the **CSSearchableItemActivityIdentifier** that was passed in – that's the unique identifier of the indexed item that was tapped. In this project, that's the project number.

Once we know which project caused the app to be launched, we need to do a little view controller dance. This isn't particularly pretty, but if you look at the **didFinishLaunchingWithOptions** method (also in AppDelegate.swift) you'll see the code is identical – it's not pretty, but at least it's consistent!

This view controller dance is a matter of typecasting the window's root view controller to a **UISplitViewController**, finding the navigation controller it contains, then finding the **MasterViewController** inside *that*. In short, it lets the app delegate find our **MasterViewController** object, and once that's done we're able to call **showTutorial()** directly. If you remember, this takes the project number to launch, and because we're also using project numbers as our unique identifier it makes our code easy.

Put this into AppDelegate.swift:

```
func application(application: UIApplication, continueUserActivity
userActivity: NSUserActivity, restorationHandler: ([AnyObject]?) ->
Void) -> Bool {

  if userActivity.activityType == CSSearchableItemActionType {

    if let uniqueIdentifier = userActivity.userInfo?
[CSSearchableItemActivityIdentifier] as? String {

      let splitViewController = self.window!.rootViewController as!
UISplitViewController

      let navigationController =
splitViewController.viewControllers[splitViewController.viewControlle
rs.count-1] as! UINavigationController


      if let masterVC = navigationController.topViewController as?
MasterViewController {

        masterVC.showTutorial(Int(uniqueIdentifier)!)

      }

    }

  }


  return true
```

```
}
```

You'll get an error at first, but that's easily fixed by adding **import CoreSpotlight** to the top of the file.

That's the third and final stage complete, which means the project is also complete. Run it now, and try clicking the + button next to Project 4. Now press Shift+Cmd+H to return to the home screen in the simulator and swipe to the left until you reach the Spotlight search tab. You should be able to type "uit" into the search box to have it find the reference to **UIToolbar** in project 4's description.

Now, before you go off indexing all sorts of information, be warned: Apple has said that iOS will automatically monitor how frequently users interact with your search results, and if you consistently serve up unhelpful results because you indexed your data badly then your results may stop appearing. Index only what's important!

Wrap up

This project covered a huge amount, including Core Spotlight, **SFSafariViewController**, **NSAttributedString**, automatically sized table view cells, and also Dynamic Type. Plus, you have another project complete, and you're now able to customize it to fit your needs – as nice as a Hacking with Swift browser is, I'm sure you have better ideas!

If you want to work on this project some more, a great place to start is to convert the **projects** array to contain objects of a custom subclass rather than just an array. Not only is it safer coding, but it's also more extensible – you might want to add images or other data, and our array stops being so simple when you add more to it! You should follow much the same technique as taught in project 12 to handle loading and saving.

I'd also recommend you investigate some of the many other formatting options you can use with **NSAttributedString**. Right-click on **NSFontAttributeName** and choose Jump to Definition to see a list, and just try things out! You'll see that Apple has put comments next to each key so you can see what kind of data to provide.

There's one more thing, which is the user changing their Dynamic Type size. This won't happen very often, but if it happens while your app is running you'll receive the **UIContentSizeCategoryDidChangeNotification** if you subscribe to it using **NSNotificationCenter**. This is your chance to refresh your user interface so that fonts are drawn at the new size.

Project 33

What's that Whistle?

Build a crowd-sourced song recognition app using Apple's free platform as a service: CloudKit.

Setting up

As I write these initial words, I already know this is going to be one of the most expansive and useful Hacking with Swift tutorials to date. We're going to be using CloudKit to load and save user data, we'll read from the microphone using **AVAudioRecorder**, we'll add **UIStackView** and **NSAttributedString** for great layout, and we're going to tie in push messaging for dynamic updates.

This tutorial is going to show you CloudKit to a depth you won't see much elsewhere. Yes, we're going to be loading and saving text data, but we're also going to be loading and saving binary data, registering for updates, and delivering push messages – in short, we're going to be covering the majority of CloudKit in one project, and you'll learn a huge amount along the way. We're even going to be covering the simple and advanced methods of working with CloudKit, because it gives a much better user experience.

As you know, my usual plan is to pick which technologies I want to teach, then strap a real project around it. In this case, the project is called "What's that Whistle?" It's an app where users can whistle or hum into their microphone, and upload it to iCloud. Other users can then download whistles and try to identify what song it's from. This is a genuinely useful app: think how often you know how a song goes but just can't remember its name, and boom: this app is for you. Of course, you could also turn it into a game – who can guess the song first? It's down to you.

To make things more exciting, we'll let users choose which music genres they specialize in, and we'll deliver them a push message whenever a new whistle comes in for that genre. They can then swipe to unlock and launch the app, and start posting suggestions for the song name.

CloudKit is something that was announced with iOS 8, but it had some remarkably low limits that made it so unappealing I decided not to write a tutorial about it. Back then, each app got just 25MB per day of data transfer, growing at 0.5MB – yes, half a megabyte – per user per day, which was absurdly low. But with iOS 9 Apple announced they were raising the limits so that every app now gets 2GB per month plus a further 50MB per user per month. Even better, database transfer is now judged on requests per second rather than an arbitrary data cap.

If you're not aware of what CloudKit does, I'll be going into much more detail later. For now, the least you need to know is that CloudKit lets you send and retrieve data for your app, effectively providing a server back-end for your app to talk to. Even better, unless you go

over Apple's new generous limits, it's completely free.

This is a Hacking with Swift Level 2 project, which means I'm expecting you to have completed **projects 1 to 30** already. Please don't complain if you find something unexplained, because it was almost certainly covered earlier. It's also going to built using Swift 2 and iOS 9, so please ensure you have Xcode 7 installed before continuing.

Important warning: this project requires an active Apple developer program account, because CloudKit requires iCloud developer access. Although everything except push messaging works great in the simulator, you'll find CloudKit responds much faster on devices so you might find it easier to work from a device the entire time.

Please go ahead and create a new project in Xcode 7, choosing the Single View Application template. Name it Project 33, choose Swift for your language, and iPhone for the device. Now strap yourself in, because this is going to be awesome…

Recording from the microphone with AVAudioRecorder

We're going to start off this project easily enough by looking at **AVAudioRecorder**: the iOS way of recording audio from the microphone. You might be tempted to skip past this so you can focus on the CloudKit parts, but please don't – I didn't put audio recording in here just for fun! Instead, it's used to demonstrate how to store binary assets (i.e., data files) inside CloudKit, so it's an integral part of the project.

The built-in Xcode template will have given you one empty view controller inside Main.storyboard, plus a ViewController.swift file. We're going to be doing almost all the user interface in code for this project, so you can almost ignore the storyboard entirely – in fact, all you need to do is select the view controller that was created and embed it inside a navigation controller. Now go to ViewController.swift and add these lines to **viewDidLoad()**:

```
title = "What's that Whistle?"

navigationItem.rightBarButtonItem =
UIBarButtonItem(barButtonSystemItem: .Add, target: self, action:
"addWhistle")

navigationItem.backBarButtonItem = UIBarButtonItem(title: "Home",
style: .Plain, target: nil, action: nil)
```

That gives us a button to tap to add a whistle to the app, then customizes the title of the navigation bar's back button to say "Home" rather than "What's that Whistle?" which is a bit too long. We're going to return to this view controller much later to add a table view and other bits, but for now let's write the very simple **addWhistle()** method:

```
func addWhistle() {
  let vc = RecordWhistleViewController()
  navigationController?.pushViewController(vc, animated: true)
}
```

So, that creates a new object of type **RecordWhistleViewController** (not yet written), and

pushes it onto the view controller stack. That's it for ViewController.swift for now – trust me, we'll be adding a *lot* more code to it later!

Add a new file to your project, choosing Cocoa Touch Class. Make it a subclass of **UIViewController** and name it **RecordWhistleViewController**. Make sure "Also create XIB file" is not selected and that "Swift" is chosen for your language, then click Next to save it.

To begin with, we're going to place a **UIStackView** into the view, along with a button saying "Tap to record". You might think the stack view isn't necessary, but once we've got the recording working we're going to add a playback button and have **UIStackView** animate it all nicely for us, so it is definitely needed.

First things first: let's create the stack view, then use Auto Layout to pin it horizontally and center it vertically. This way, the stack view will automatically grow as more views are added to it, which can be animated to look extra slick. If you haven't already read my previous UIStackView tutorial you might want to refer to that now.

Add this property to the new **RecordWhistleViewController** class:

```
var stackView: UIStackView!
```

Like I said, we're going to be doing all the view layout in code to make it easier to follow. **UIStackView** takes care of all the layout of its subviews, so all we need to do is position and size the stack view correctly. Put this into your class to load the view:

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

  view.backgroundColor = UIColor.grayColor()

  stackView = UIStackView()
  stackView.spacing = 30
  stackView.translatesAutoresizingMaskIntoConstraints = false
```

```
    stackView.distribution = UIStackViewDistribution.FillEqually

    stackView.alignment = UIStackViewAlignment.Center

    stackView.axis = .Vertical

    view.addSubview(stackView)
```

```
view.addConstraints(NSLayoutConstraint.constraintsWithVisualFormat("H
:|[stackView]|", options: NSLayoutFormatOptions.AlignAllCenterX,
metrics: nil, views: ["stackView": stackView]))

   view.addConstraint(NSLayoutConstraint(item: stackView,
attribute: .CenterY, relatedBy: .Equal, toItem: view,
attribute: .CenterY, multiplier: 1, constant:0))

}
```

You'll notice I use Visual Format Language (VFL) for horizontal pinning, then a regular Auto Layout constraint for the vertical centering – that's not something you can easily express in VFL, which is why it's done in straight code. The stack view is set up to have its subviews size themselves equally with a 30-point space between them, which looked pretty good in my testing.

I've covered all that in previous projects, but now for the new stuff: using **AVAudioRecorder** to record audio from the microphone. If you're using the simulator this will automatically use your Mac's built-in microphone, so you can test either on device or in the simulator.

Recording audio in iOS 9 uses two classes: **AVAudioSession** and **AVAudioRecorder**. **AVAudioSession** is there to enable and track sound recording as a whole, and **AVAudioRecorder** is there to track one individual recording. That is, the session is the bit that ensures we are able to record, the recorder is the bit that actual pulls data from the microphone and writes it to disk.

Use of the microphone is restricted by access controls, as you might imagine, so when we use **AVAudioSession** to request access it will automatically display a warning to the user, prompting them to confirm the access. Once we have access, we can record as much data as we want at whatever quality we want, and iOS will do most of the work.

To make the interface user friendly we're going to allow users to re-record their whistle as many times as it takes, so we'll need a button to handle that. We're also going to change the

background color of the view to either red or green, to show the recording or not recording state in a more visually obvious way.

First up, we need to import the **AVFoundation** framework for this class, so please add this import now:

```
import AVFoundation
```

We're also going to track three new properties: the record button, the recording session, and the **AVAudioRecorder** itself. Please add these three properties to your class:

```
var recordButton: UIButton!

var recordingSession: AVAudioSession!
var whistleRecorder: AVAudioRecorder!
```

Now for the complicated part: setting up the recording environment. To do this, we're going to get hold of the built-in system audio session and ask for play and record privileges – record so we can grab audio from the microphone, and play so users can previous what was recorded. We'll use the **requestRecordPermission()** method of the audio session to ask the user whether we can record or not, and give that a trailing closure to execute when the user makes a choice.

If the user grants us access to the microphone, we'll execute a new method called **loadRecordingUI()**, otherwise we'll call **loadFailUI()**. Both of these need to be pushed onto the main thread because the callback from **requestRecordPermission()** can happen on any thread. I'll provide you with method stubs for **loadRecordingUI()** and **loadFailUI()** for now, but we'll add to them shortly. Replace your current **viewDidLoad()** with this:

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

```swift
        title = "Record your whistle"

    navigationItem.backBarButtonItem = UIBarButtonItem(title: "Record",
style: .Plain, target: nil, action: nil)


    recordingSession = AVAudioSession.sharedInstance()


    do {
        try
recordingSession.setCategory(AVAudioSessionCategoryPlayAndRecord)

        try recordingSession.setActive(true)

        recordingSession.requestRecordPermission() { [unowned self]
(allowed: Bool) -> Void in

            dispatch_async(dispatch_get_main_queue()) {

                if allowed {

                    self.loadRecordingUI()

                } else {

                    self.loadFailUI()

                }

            }

        }

    } catch {

        self.loadFailUI()

    }

}


func loadRecordingUI() {

}


func loadFailUI() {

}
```

If you haven't met **the new try/catch/do/throw error handling syntax in Swift 2**, you should really click that link now. Short version: if we fail to request access to the microphone, or if we request access and are denied, we load the fail UI, otherwise we load the recording UI.

To help the interface adapt to each user's preferences, we're going to use Dynamic Type to control our fonts. This means users can adjust the font size in the Settings app and have it reflected in our app. We don't need to worry about sizing up or positioning the buttons – just by using **UIFontTextStyleTitle1** and **UIFontTextStyleHeadline** we give the labels enough information to size themselves, and the stack view will do the rest.

Here's some new code for **loadRecordingUI()** and **loadFailUI()**; replace the previous stubs with this:

```
func loadRecordingUI() {

  recordButton = UIButton()

  recordButton.translatesAutoresizingMaskIntoConstraints = false

  recordButton.setTitle("Tap to Record", forState: .Normal)

  recordButton.titleLabel?.font =
UIFont.preferredFontForTextStyle(UIFontTextStyleTitle1)

  recordButton.addTarget(self, action: "recordTapped",
forControlEvents: .TouchUpInside)

  stackView.addArrangedSubview(recordButton)

}


func loadFailUI() {

  let failLabel = UILabel()

  failLabel.font =
UIFont.preferredFontForTextStyle(UIFontTextStyleHeadline)

  failLabel.text = "Recording failed: please ensure the app has
access to your microphone."

  failLabel.numberOfLines = 0


  stackView.addArrangedSubview(failLabel)
```

```
}
```

As you should know by now, setting **numberOfLines** to 0 means "wrap over as many lines as you need." Also, and this is important, you should also be aware that you never add a subview to a **UIStackView** directly. Instead, you use its **addArrangedSubview()** method, which is what triggers the layout work.

Tapping that button will trigger a method that we haven't written yet: **recordTapped()**. But before we get onto that, you need to know a little about how recording works:

• You need to tell iOS where to save the recording. This is done when you create the **AVAudioRecorder** object because iOS streams the audio to the file as it goes so it can write large files.
• Before recording begins, you need to decide on a format, bit rate, channel number and quality. We'll be using Apple's AAC format because it gets the most quality for the lowest bitrate. For bitrate we'll use 12,000Hz, which, when combined with the High AAC quality, sounds good in my testing. We'll specify 1 for the number of channels, because iPhones only have mono input.
• If you set your view controller as the delegate of a recording, you'll be told when recording stops and whether it finished successfully or not.
• Recording won't stop if your app goes into the background briefly. Instead, it's things like a call coming in that might make it stop unexpectedly.

As our app is simple, we don't need a complicated method to figure out where to save our whistle audio. In project 10 I gave you a simple helper method called **getDocumentsDirectory()**, which returns the path to a writeable directory owned by your app. This is a great place to save the audio, so we'll take that and append "whistle.m4a" for our filename. Put these two new methods into your code:

```
class func getDocumentsDirectory() -> NSString {

  let paths =
NSSearchPathForDirectoriesInDomains(.DocumentDirectory, .UserDomainMa
sk, true) as [String]

  let documentsDirectory = paths[0]

  return documentsDirectory

}
```

```
class func getWhistleURL() -> NSURL {

  let audioFilename =
getDocumentsDirectory().stringByAppendingPathComponent("whistle.m4a")

  let audioURL = NSURL(fileURLWithPath: audioFilename)


  return audioURL

}
```

Note that both of those methods have the **class** keyword at the beginning, which means you call them on the class not on instances of the class. This is important, because it means we can find the whistle URL from anywhere in our app rather than typing it in everywhere.

When we want to start recording, the app needs to do a few things:

1. Make the view have a red background color so the user knows they are in recording mode.
2. Change the title of the record button to say "Tap to Stop".
3. Use the **getWhistleURL()** method we just wrote to find where to save the whistle.
4. Create a settings dictionary describing the format, sample rate, channels and quality.
5. Create an **AVAudioRecorder** object pointing at our whistle URL, set ourselves as the delegate, then call its **record()** method.

Before I show you the code for that, there are two other important things to know. First, when working in the simulator I usually like to print out the URL to a file using **print()**, because it means I can look at it in Finder and be sure it's working correctly. Second, creating an **AVAudioRecorder** can throw an error, so we need to wrap it in a **do/try/catch** block.

That's it – here's the code for **startRecording()**, with numbers added to match the list above:

```
func startRecording() {
  // 1
  view.backgroundColor = UIColor(red: 0.6, green: 0, blue: 0, alpha:
1)
```

```
  // 2

  recordButton.setTitle("Tap to Stop", forState: .Normal)


  // 3

  let audioURL = RecordWhistleViewController.getWhistleURL()

  print(audioURL.absoluteString)


  // 4

  let settings = [

    AVFormatIDKey: Int(kAudioFormatMPEG4AAC),

    AVSampleRateKey: 12000.0,

    AVNumberOfChannelsKey: 1 as NSNumber,

    AVEncoderAudioQualityKey: AVAudioQuality.High.rawValue

  ]


  do {

    // 5

    whistleRecorder = try AVAudioRecorder(URL: audioURL, settings:
settings)

    whistleRecorder.delegate = self

    whistleRecorder.record()

  } catch {

    finishRecording(success: false)

  }

}
```

Note: as soon as you try to set **self** for **whistleRecorder.delegate**, you'll need to conform to the **AVAudioRecorderDelegate** protocol, like this:

```
class RecordWhistleViewController: UIViewController,
```

```
AVAudioRecorderDelegate {
```

Once recording has started, we naturally want to stop it at some point. For that, we're going to create a **finishRecord()** method, which will take one boolean parameter saying whether the recording was successful or not. It will make the view's background color green to show that recording is finished, then it will destroy the **AVAudioRecorder** object.

The special part of this method lies in whether the recording was a success or not. If it was a success, we're going to set the record button's title to be "Tap to Re-record", but then show a new right bar button item in the navigation bar, letting users progress to the next stage of submission. So, they can submit what they have, or re-record as often as they want. If the record wasn't a success, we'll put the button's title back to being "Tap to Record" then show an error message.

Here's the new method:

```
func finishRecording(success success: Bool) {
  view.backgroundColor = UIColor(red: 0, green: 0.6, blue: 0, alpha:
1)


  whistleRecorder.stop()
  whistleRecorder = nil


  if success {
    recordButton.setTitle("Tap to Re-record", forState: .Normal)

    navigationItem.rightBarButtonItem = UIBarButtonItem(title:
"Next", style: .Plain, target: self, action: "nextTapped")
  } else {
    recordButton.setTitle("Tap to Record", forState: .Normal)


    let ac = UIAlertController(title: "Record failed", message:
"There was a problem recording your whistle; please try again.",
preferredStyle: .Alert)
    ac.addAction(UIAlertAction(title: "OK", style: .Default, handler:
```

```
nil))

    presentViewController(ac, animated: true, completion: nil)

  }

}
```

So, that's the code to start and stop recording – amazingly you're most of the way there! Our one button will be used to trigger both events, so we need to write **recordTapped()**. All this will do is call **startRecording()** or **finishRecording()** depending on the current state of the app. And that's it! Here's the code:

```
func recordTapped() {

  if whistleRecorder == nil {

    startRecording()

  } else {

    finishRecording(success: true)

  }

}
```

The last thing to do before we're done with recording is to catch the scenario where recording ends with a problem. We already set our view controller to be the delegate of our **AVAudioRecorder** object, so we'll get sent a **audioRecorderDidFinishRecording()** message when recording finished. If the recording wasn't a success, we'll call **finishRecording()** so it can clean up and restore the view to its pre-recording state.

Here's the code:

```
func audioRecorderDidFinishRecording(recorder: AVAudioRecorder,
successfully flag: Bool) {

  if !flag {

    finishRecording(success: false)

  }
```

```
}
```

At this point your code will run, and I encourage you to try it in the iOS Simulator so you can see that it's working – if you kept my **print()** call in there, you can open that folder in Finder and see the finished m4a if everything has gone well.
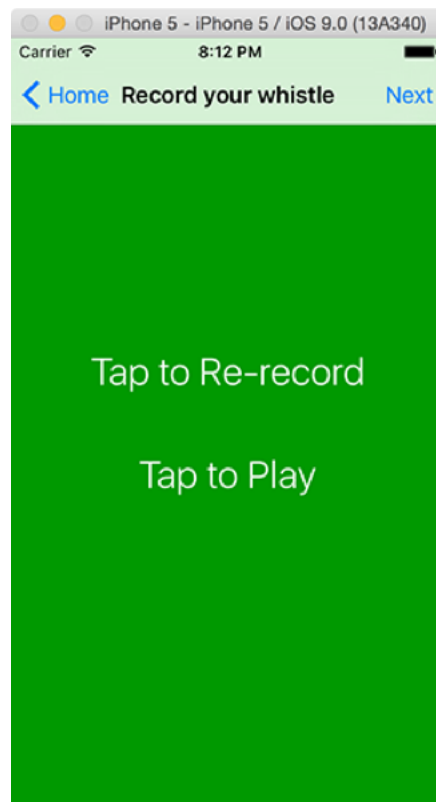
Adding our buttons into a UIStackView means we can animate them appearing and disappearing without much work.

Note: if you're a less experienced OS X user, you might not know how to navigate to a folder like the one the iOS Simulator uses, because it's hidden by default. For example, you'll get something like this: **file:///Users/twostraws/Library/Developer/CoreSimulator/Devices/ E470B24D-5C0C-455F-9726-DC1EAF30D5A4/data/Containers/Data/Application/ D5E4C08C-2B1E-40BC-8EBE-97F136D0AFC0/Documents/whistle.m4a** – which hardly trips off the tongue!

The easiest thing to do is copy that to a clipboard, open a Finder window, press Shift+Cmd +G, and paste it into the box. Now delete the "file://" from the start so that it reads "/Users/

yourusername/.....", and "whistle.m4a" from the end, then press Return.

Animating UIStackView subview layout

Before we get onto the CloudKit part of this tutorial, we're going to add a bit more to our user interface. Specifically, we're going to add a "Tap to Play" button into the stack view, and have it animate so that it slides out when recording has finished. This is the work of only a few minutes thanks to **UIStackView**, and I'm sure you'll agree the results look marvellous.

While we're finishing up the user interface, we're going to quickly add a couple more simple view controllers to let the user attach some metadata to their whistle: they'll be able to select what genre it is, then enter some free text with any comments – something like "I definitely remember hearing it in the early 90s" to help listeners narrow the scope a little.

First, the play button. Add this new property:

```
var playButton: UIButton!
```

Now create it by placing this just before the end of **loadRecordingUI()**:

```
playButton = UIButton()

playButton.translatesAutoresizingMaskIntoConstraints = false

playButton.setTitle("Tap to Play", forState: .Normal)

playButton.hidden = true

playButton.alpha = 0

playButton.titleLabel?.font =
UIFont.preferredFontForTextStyle(UIFontTextStyleTitle1)

playButton.addTarget(self, action: "playTapped",
forControlEvents: .TouchUpInside)

stackView.addArrangedSubview(playButton)
```

That's almost identical to the code for creating the record button, except the play button is set to hidden and alpha 0. Normally you need only one of these, but with stack views it's a little different: a view that is not hidden but has an alpha of 0 appears hidden (i.e., the user can't see it) but still occupies space in the stack view. By setting the button to be hidden and

have alpha 0, we're saying "don't show it to the user, and don't let it take up any space in the stack view."

We want to show and hide that play button when needed, meaning that we show it when recording finished successfully and hide it if the user taps to re-record. To solve the first of those, put this code into the **finishRecording()** method, just before setting the right bar button item:

```
if playButton.hidden {
  UIView.animateWithDuration(0.35) { [unowned self] in
    self.playButton.hidden = false
    self.playButton.alpha = 1
  }
}
```

To solve the second, put this into **recordTapped()**, just after the call to **startRecording()**:

```
if !playButton.hidden {
  UIView.animateWithDuration(0.35) { [unowned self] in
    self.playButton.hidden = true
    self.playButton.alpha = 0
  }
}
```

The **hidden** property of any **UIView** subclass is a simple boolean, meaning that it's either true or false: a view is either hidden or it's not. As a result, if we had put this code anywhere else it would be meaningless to try to animate it, because there are no intermediate steps between "visible" and "invisible" to animate. But with **UIStackView** it has a meaning, and that meaning is brilliant: the stack view will animate the play button being shown, making it slide out neatly. Changing the alpha at the same time is the perfect finishing touch.

When we created the play button we attached a method called **playTapped()** to it, which isn't written yet. But now that you've seen how to use **AVAudioRecorder**, the code to play using **AVAudioPlayer** should be second nature. Just in case you don't fancy writing the code for yourself, I'll walk you through the steps.

First, create a new property to hold the audio player:

```
var whistlePlayer: AVAudioPlayer!
```

Now, add a **playTapped()** method using the code below. This grabs the shared whistle URL, creates an **AVAudioPlayer** inside a **do/try/catch** block, and makes it play. If there's an error loading the sound it shows an alert message to the user. Easy, right?

```
func playTapped() {
  let audioURL = RecordWhistleViewController.getWhistleURL()

  do {
    whistlePlayer = try AVAudioPlayer(contentsOfURL: audioURL)
    whistlePlayer.play()
  } catch {
    let ac = UIAlertController(title: "Playback failed", message:
"There was a problem playing your whistle; please try re-recording.",
preferredStyle: .Alert)
    ac.addAction(UIAlertAction(title: "OK", style: .Default, handler:
nil))
    presentViewController(ac, animated: true, completion: nil)
  }
}
```

If you run the app now I think you'll agree it looks good, particularly as the play button slides out in the stack view. Being able to hear what you recorded is of course a nice touch!

Once the user has a recording they are happy with, we're going to ask them to choose which genre they think it belongs to, and add any comments. Both of these are very simple view controllers that we can make in just a few minutes. As I made clear earlier, you should already have finished projects 1 to 30 of Hacking with Swift before getting this far, so I'm going to assume you know everything that was taught there and will explain only things that are new or interesting!

Add a new file to your project, choosing Cocoa Touch Class. Make it a subclass of **UITableViewController** and name it **SelectGenreViewController**. Open the file for editing, and give it this property:

```
static var genres = ["Unknown", "Blues", "Classical", "Electronic",
"Jazz", "Metal", "Pop", "Reggae", "RnB", "Rock", "Soul"]
```

This is marked as static so that we can use it in lots of other places – it's a shared list of all the music categories we want to work with. I added "Unknown" in there for people like me who struggle to tell the difference between some music types!

In this class's **viewDidLoad()** method we're going to give it a title, configure the back button to take up less space, then register a cell for re-use. All old stuff:

```
override func viewDidLoad() {

  super.viewDidLoad()


  title = "Select genre"

  navigationItem.backBarButtonItem = UIBarButtonItem(title: "Genre",
style: .Plain, target: nil, action: nil)

  tableView.registerClass(UITableViewCell.self,
forCellReuseIdentifier: "Cell")

}
```

For handling the content of the table view, it's all code you've seen in previous projects, but I want to point out three things:

1. When referencing the **genres** array we need to use **SelectGenreViewController.genres** because the array belongs to the class, not to our instance of the class.

2. When reading the text of the cell that was tapped, we're going to use the nil coalescing operator. The nil coalescing operator was covered in project 12, and in this situation it guarantees we have a genre.

3. When the user has selected a genre, we're going to create an instance of the class **AddCommentsViewController**, store that genre there, then push it onto our navigation stack.

That's it – here are the methods for handling the table view data source and delegate:

```
override func numberOfSectionsInTableView(tableView: UITableView) ->
Int {

   return 1

}


override func tableView(tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {

   return SelectGenreViewController.genres.count

}


override func tableView(tableView: UITableView, cellForRowAtIndexPath
indexPath: NSIndexPath) -> UITableViewCell {

   let cell = tableView.dequeueReusableCellWithIdentifier("Cell",
forIndexPath: indexPath)

   cell.textLabel?.text =
SelectGenreViewController.genres[indexPath.row]

   cell.accessoryType = .DisclosureIndicator

   return cell

}


override func tableView(tableView: UITableView,
didSelectRowAtIndexPath indexPath: NSIndexPath) {
```

```
if let cell = tableView.cellForRowAtIndexPath(indexPath) {

    let genre = cell.textLabel?.text ??
SelectGenreViewController.genres[0]

    let vc = AddCommentsViewController()

    vc.genre = genre

    navigationController?.pushViewController(vc, animated: true)

  }

}
```

That completes the class – I've deliberately kept it simple because this tutorial is about CloudKit rather than tables! You can now return to RecordWhistleViewController.swift and add this method to the end:

```
func nextTapped() {

  let vc = SelectGenreViewController()

  navigationController?.pushViewController(vc, animated: true)

}
```

There's one more easy class to add before we get onto CloudKit, and that's **AddCommentsViewController**. This will show a full-screen **UITextView** for the user to type any extra comments into. Create it now, making it a subclass of **UIViewController**, then select it for editing.

We're going to give this new class three properties: one to hold the genre that gets passed in from **SelectGenreViewController**, one to hold a reference to the **UITextView**, and one to hold a placeholder string. That last property will be used to solve a long-standing **UITextView** annoyance: unlike **UITextField**, you cannot give a **UITextView** a placeholder string, which is a piece of text telling users what to type in there. We'll replicate this behaviour by putting a default string into the text view and removing it when the user taps it.

So, add these three properties to **AddCommentsViewController**:

```
var genre: String!


var comments: UITextView!

let placeholder = "If you have any additional comments that might
help identify your tune, enter them here."
```

We're going to override the **loadView()** method of this class, using it to create a new **UITextView** that is pinned to all edges using Auto Layout. The only vaguely interesting thing here is that we'll use Dynamic Type to make the font size adjustable for the user. Here's the code:

```
override func loadView() {

    super.loadView()


    comments = UITextView()

    comments.translatesAutoresizingMaskIntoConstraints = false

    comments.delegate = self

    comments.font =
UIFont.preferredFontForTextStyle(UIFontTextStyleBody)

    view.addSubview(comments)



view.addConstraints(NSLayoutConstraint.constraintsWithVisualFormat("H
:|[comments]|", options: .AlignAllCenterX, metrics: nil, views:
["comments": comments]))

view.addConstraints(NSLayoutConstraint.constraintsWithVisualFormat("V
:|[comments]|", options: .AlignAllCenterX, metrics: nil, views:
["comments": comments]))

}
```

As per usual, assigning the view controller to be a delegate of something requires conforming to a protocol. In this case, it means conforming to **UITextViewDelegate**, so please add that

now.

The absolute least we need to do to make this class work is to fill in the **viewDidLoad()** method with a title for the view controller and a right bar button item to let the user proceed with their submission, then to write a **submitTapped()** method that gets triggered when the button is tapped.

Submitting will use another new class that we'll define shortly, called **SubmitViewController**, and will pass in the genre we got from **SelectGenreViewController** and the user's comments if there are any. If they kept the placeholder intact, we'll send an empty string on. Here's the code:

```
override func viewDidLoad() {

  super.viewDidLoad()


  title = "Comments"
  navigationItem.rightBarButtonItem = UIBarButtonItem(title:
"Submit", style: .Plain, target: self, action: "submitTapped")
  comments.text = placeholder

}


func submitTapped() {
  let vc = SubmitViewController()
  vc.genre = genre


  if comments.text == placeholder {
    vc.comments = ""
  } else {
    vc.comments = comments.text
  }


  navigationController?.pushViewController(vc, animated: true)
```

```
}
```

We could easily leave it there and get onto to the CloudKit work, but there's one small tweak we can make to improve the whole experience. As this view controller is the delegate for the **comments** text view, iOS will send us the **textViewDidBeginEditing()** message when the user starts editing it. We can then compare the text view's current text against the placeholder, and clear it if they match. Here's that code:

```
func textViewDidBeginEditing(textView: UITextView) {

  if textView.text == placeholder {

    textView.text = ""

  }

}
```

That's it: it's time for CloudKit.

Writing to iCloud with CloudKit: CKRecord and CKAsset

We still have an error in our code, because we haven't created the **SubmitViewController** class yet. This is where CloudKit comes into play, because this view controller has only one job: to show the user that iCloud submission is happening until it completes, at which point we'll show a "Done" button.

To make the view a little more interesting, we're going to use another **UIStackView** to arrange a text label and an activity spinner to keep the user informed. We're also going to hide the back button so the user can't escape until iCloud finishes, successfully or otherwise.

The last two view controllers have been collecting and passing on metadata, specifically a genre and user comments. We'll need to add these as properties for **SubmitViewController**, along with properties for the stack view, the label and the activity spinner. So, create a new class called **SubmitViewController**, make it a subclass of **UIViewController**, then add these properties:

```
var genre: String!
var comments: String!


var stackView: UIStackView!
var status: UILabel!
var spinner: UIActivityIndicatorView!
```

As for the **loadView()** method, this is very similar to what we did with **RecordWhistleViewController**: we'll use a stack view that is pinned to the left and right edges then centered vertically. We'll add to this a **UILabel** to show the send status, and a **UIActivityIndicatorView** to show the user that something is happening, but otherwise there's nothing surprising in this code:

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

```
   view.backgroundColor = UIColor.grayColor()


   stackView = UIStackView()

   stackView.spacing = 10

   stackView.translatesAutoresizingMaskIntoConstraints = false

   stackView.distribution = UIStackViewDistribution.FillEqually

   stackView.alignment = UIStackViewAlignment.Center

   stackView.axis = .Vertical

   view.addSubview(stackView)



view.addConstraints(NSLayoutConstraint.constraintsWithVisualFormat("H
:|[stackView]|", options: NSLayoutFormatOptions.AlignAllCenterX,
metrics: nil, views: ["stackView": stackView]))

   view.addConstraint(NSLayoutConstraint(item: stackView,
attribute: .CenterY, relatedBy: .Equal, toItem: view,
attribute: .CenterY, multiplier: 1, constant:0))



   status = UILabel()

   status.translatesAutoresizingMaskIntoConstraints = false

   status.text = "Submitting…"

   status.textColor = UIColor.whiteColor()

   status.font =
UIFont.preferredFontForTextStyle(UIFontTextStyleTitle1)

   status.numberOfLines = 0

   status.textAlignment = .Center


   spinner =
UIActivityIndicatorView(activityIndicatorStyle: .WhiteLarge)

   spinner.translatesAutoresizingMaskIntoConstraints = false

   spinner.hidesWhenStopped = true

   spinner.startAnimating()
```

```
    stackView.addArrangedSubview(status)
    stackView.addArrangedSubview(spinner)
}
```

As I said already, we're going to hide the navigation bar's back button so the user can't back out of the view controller until submission has finished. Avoiding multiple submissions is a whole other discussion that I'll save for another day, so this is the easiest way out. Modify the existing **viewDidLoad()** method to this:

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

    title = "You're all set!"
    navigationItem.hidesBackButton = true
}
```

There are just three more methods to write before we get onto the real meat of this tutorial: CloudKit. The first is **viewDidAppear()**, which we'll use to start the submission process. The second is a stub for **doSubmission()**, which is empty for now but we'll fill with CloudKit goodness shortly. The last one is **doneTapped()**, which will be called from a bar button item, which in turn will be created when the submission finishes.

Add this code:

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

    doSubmission()
}
```

```
func doSubmission() {


}


func doneTapped() {
  navigationController?.popToRootViewControllerAnimated(true)
}
```

I don't think we've used **popToRootViewControllerAnimated()** before, but it's not a difficult method: calling it pops off all the view controllers on a navigation controller's stack, returning us to the original view controller - in our case, that's the "What's that Whistle?" screen with the + button.

OK, time for CloudKit: add this import to SubmitViewController.swift:

```
import CloudKit
```

Once that's done, a whole range of new classes become available for our Swift code, but our app can't use CloudKit just yet. This is because access to iCloud is restricted by Apple, so you need to state that your app wants permission to use iCloud. To do that, click the blue Project33 icon in the Project Navigator pane, then choose your Project33 target. Select the Capabilities tab, then turn iCloud to ON and make sure the CloudKit box is checked.
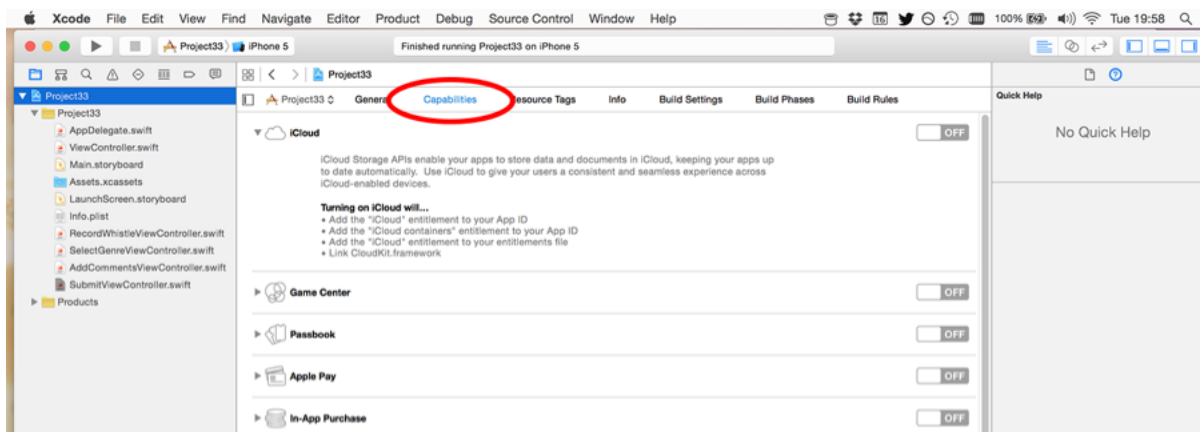
By default the iCloud entitlement is turned off for your project.
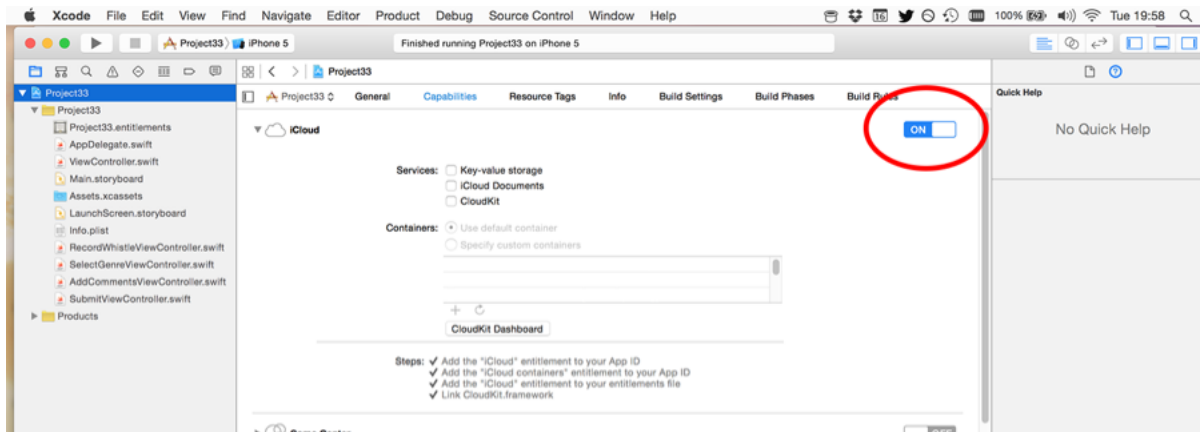
Enabling the iCloud entitlement requires a valid Apple developer program account.
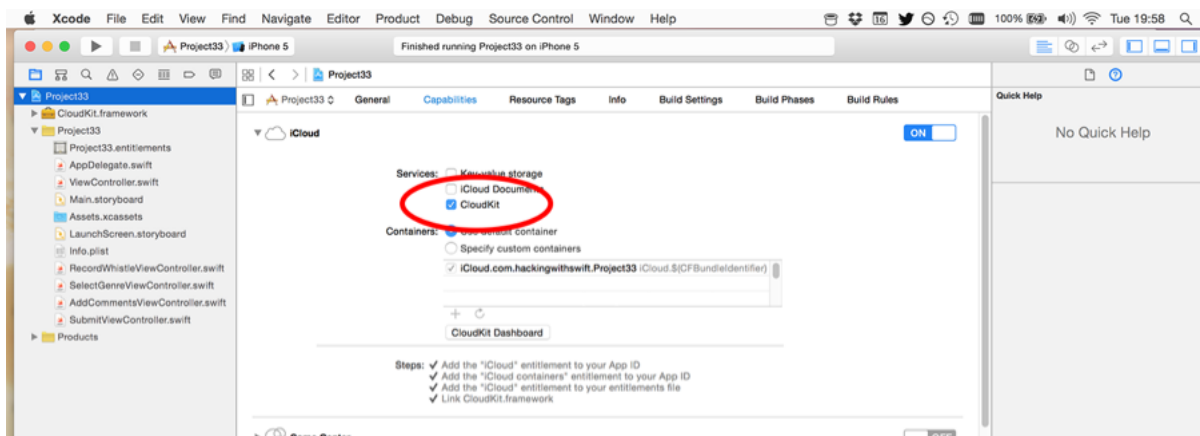
Make sure you select the checkbox marked CloudKit inside the iCloud entitlement.

When you do this, Apple will create an iCloud container for you called something like iCloud.com.hackingwithswift.Project33. Now: on your device (or in the simulator) you should make sure you are logged into iCloud and have iCloud Drive enabled. That's it: you're all set to use iCloud!

At this point in the project, here's what you need to know about iCloud and CloudKit:

 • You create **CKRecord** objects to contain keys and values. They are like a dictionary, just with some extra smarts built in.
 • You create **CKAsset** objects to hold binary blobs like our audio recording. You can attach these to a **CKRecord** just like any other value.

• Each app has its own CloudKit container (**CKContainer**), and each container has two databases (**CKDatabase**) called the public and the private database.

• The private database is for storing private user data. Any thing you upload there gets taken out of that user's iCloud quota. The public database is for storing data anyone can read. Any thing you upload there gets taken out of your CloudKit quota.

• When you write data to CloudKit it automatically figures out how to store it based on all the keys and values you provide, and their data types. You can change this later if you want.

• All CloudKit calls are asynchronous, so you provide completion blocks to be executed when the call finishes. This will tell you what went wrong if anything, but the block can be called on any thread so be careful!

Because CloudKit automatically figures out how to store your data, it means we can go ahead and start sending it whistles and whistle meta data and have it stored – no back-end configuration required.

To make things easier to explain, I'm going to split the **doSubmission()** method in two: a part that creates the record to send to iCloud, and a part that handles the result.

The first part is straightforward, because like I said the **CKRecord** class looks and works much like a dictionary: you set any key to any (valid!) value, and it does the rest. By "valid" I mean things that you can normally store in a dictionary: strings, numbers, arrays, dates and so forth. You can even store **CLLocations** for doing map-based queries – it's surprisingly simple!

You can also store assets inside **CKRecord** objects, which is exactly what we're going to do: the genre and comments are both simple string keys, but the whistle audio itself needs to be uploaded as a **CKAsset** before it's attached. This isn't hard to do, because there's a constructor method for **CKAsset** that takes a file URL just like we get back from **RecordWhistleViewController.getWhistleURL()**.

One last thing before I show you the code: each **CKRecord** has a record type, which is a string. This is a name you provide and has meaning only to you, but identifies the particular type of data you're trying to save. We're working with whistles, so we'll use the record type "Whistles". Make sure you type it correctly, because it needs to match when you write and read.

That's it – here's the first part of the **doSubmission()** method:

```
let whistleRecord = CKRecord(recordType: "Whistles")

whistleRecord["genre"] = genre

whistleRecord["comments"] = comments


let audioURL = RecordWhistleViewController.getWhistleURL()

let whistleAsset = CKAsset(fileURL: audioURL)

whistleRecord["audio"] = whistleAsset
```

The second part of the method really isn't hard at all, but it does come with a few important notices. We're going to be using the **saveRecord()** method of the CloudKit public database, which sends a **CKRecord** off to iCloud and tell us how it went. The result of this method is handed to us in a trailing closure that can be called on any thread, so the first thing you'll see is that I bounce the code back to the main thread so we can manipulate the user interface.

You'll notice that every CloudKit send method has a **NSError** being passed in to the closure, reporting whether there was a problem. It is really important that you don't ignore this: mobile networks vary in strength so actions might fail at any time, plus iCloud itself is a monstrously huge beast where a dozen things could go wrong behind the scenes, leading to your code failing. But that's OK, because you're going to catch the errors and do something sensible, right? Right.

The last important notice is that we'll be setting a property on **ViewController** to be true. This property is called **dirty** and it doesn't exist just yet, so expect an error.

All set? Here's the second part of **doSubmission()**:

```
CKContainer.defaultContainer().publicCloudDatabase.saveRecord(whistle
Record) { [unowned self] (record, error) -> Void in

  dispatch_async(dispatch_get_main_queue()) {

    if error == nil {

      self.view.backgroundColor = UIColor(red: 0, green: 0.6, blue:
0, alpha: 1)

      self.status.text = "Done!"

      self.spinner.stopAnimating()
```

```
        ViewController.dirty = true

    } else {

        self.status.text = "Error: \(error!.localizedDescription)"

        self.spinner.stopAnimating()

    }


    self.navigationItem.rightBarButtonItem = UIBarButtonItem(title:
"Done", style: .Plain, target: self, action: "doneTapped")

    }

}
```

As you can see, regardless of whether the operation succeeds or fails we show a "Done" button so the user can escape the screen. This calls the **doneTapped()** method we already wrote. Also, the **dirty** property belongs to **ViewController**: again, it's a *static* property so we can set it on the whole class rather than trying to find the correct instance of the class.

To silence the final warning, and to make your code build and run, add that static property to ViewController.swift:

```
static var dirty = true
```

Please go ahead and build your code now, and try submitting a whistle to iCloud. If you see an error at the the end asking for an authenticated account, make sure device/simulator has an iCloud account logged in, with iCloud Drive enabled, then try again.

Before we go any further, it's time for a tangent…

A hands-on guide to the CloudKit dashboard

If you already finished Hacking with Swift projects 1 to 30 (and you did, right?) you'll know I hate tangents. I'm here to teach you something cool, and I prefer to do that using as little waffle as possible – and tangents are apt to create the Perfect Storm for waffle. But in this case it's important, so please bear with me.

If you haven't already done so, you need to run your app, record a whistle, and tap Submit now. All being well it will work first time (if not you probably missed something!), but how do you know it's worked? I mean really be *sure* that's it worked? And what do you do if you want to change a data type because you made a mistake, or perhaps even delete the whole thing and start again?

Apple has a solution for this, and it's called the CloudKit Dashboard. Now that you have submitted your first record to iCloud, you can launch **https://icloud.developer.apple.com/dashboard** in your web browser and look behind the iCloud curtain as it were. The CloudKit dashboard shows you exactly what data your app is storing, who can access it, and how much of your free quota you're using.

So, just briefly, it's time for a tangent: I want to explain a few things about CloudKit Dashboard, because it's important. Yes, it *is* important – later code won't run unless you read my instructions, so please don't skip ahead.

When you log into CloudKit Dashboard, you may need to select your project in the top-left corner of the window. The default view you'll see is Schema > Record Types, and you'll see two Record Types already there: Users and Whistles. The first of those was created automatically for you by Apple, and tracks anonymised user IDs for your app. The second of these was created by you just a few minutes ago: as soon as you called **saveRecord()** CloudKit transformed your record into a database in iCloud, and added your test whistle there.

If you select the Whistles record type you'll see that CloudKit has identified that the comments and genre fields are both strings, and the audio field is an asset. You'll also see a line saying "Metadata Indexes" with the number 1 and an arrow below it. Please click that now to reveal some fields that CloudKit has created for you: ID, Created By, Date Created, Date Modified and Modified By. These are great for searching, but you can't sort usig these fields by default.
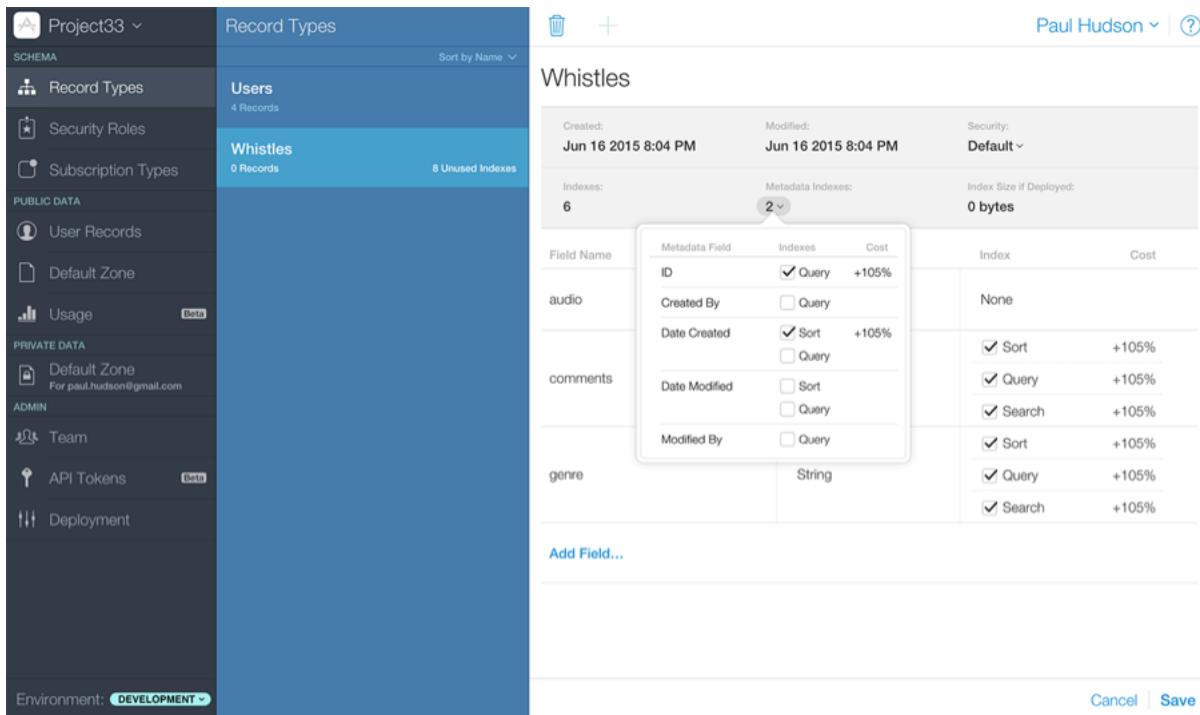
Apple's default CloudKit fields aren't enabled for searching and sorting unless you ask.

We're going to want to sort by the Date Created field later on, so please click Sort next to Date Created. While you're there, please also check Query next to ID so that we can query all records easily. With that done, click the Save button in the bottom-right of your browser window to commit that change.

That was the critical stuff needed to continue this tutorial, but there are a few other niceties while you're here:

 • Any of your rows can be deleted by hovering over them and clicking the X on the right-hand side. System rows like Created By cannot be deleted.
 • You can add fields by clicking the Add Field button at the end of your own fields, then giving it a name and type.
 • You can browse all the data that has been uploaded by clicking Default Zone in the left-hand menu and choosing a schema name.
 • When browsing individual records, you'll see links to download or remove the assets attached to the record.
 • You'll also see a trash icon above the record, which is what you click when you want to delete it.

So, the CloudKit Dashboard is basicaly a miniature CMS that lets you peek into your data and confirm everything is working OK. But it does one more thing, which is to provide usage data for your app, which is important because CloudKit is free only if you stay below certain usage limits.

To see how much of your quota you're using, click Usage now from the left-hand menu. You'll see a scrolling list of charts that show you how many users you have, how many requests per second they've made, how much storage data transfer you're using for assets, and how much database storage you're using. CloudKit shows you a solid line to represent how much you've actually used, then a dashed line to show its projections about how much you're likely to use if current trends continue.
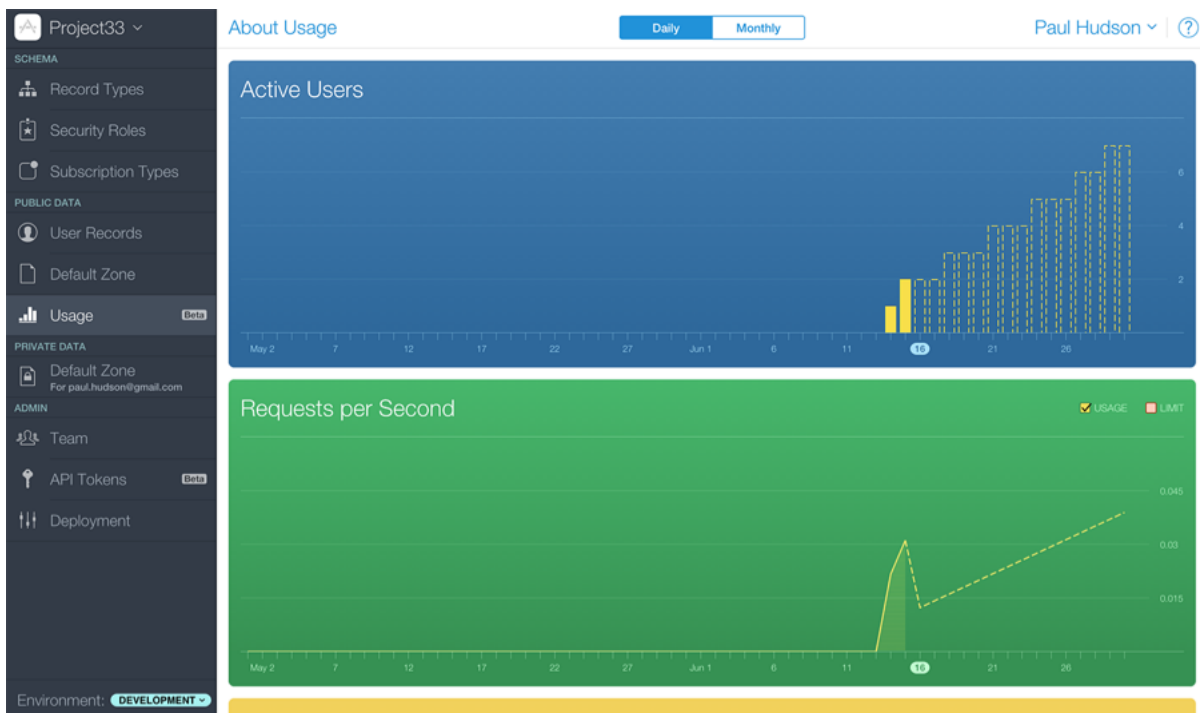
CloudKit Dashboard shows you your current and projected usage, helping you stay within Apple's free quota.

Note that all quota directly depends on the number of users you have – as you add more users, Apple adds more quota. So, the first graph directly affects all the others.

What these charts don't show is how your usage maps against your quota, and there's a good reason for that: as soon as you add in your quota, your usage becomes so tiny that you won't be able to see it! Don't believe me? Try it now: at the top right of each chart are check

boxes saying either "Usage" and "Quota", or "Usage" and "Limit" – select the one that is currently unchecked, and you'll see what I mean.

So, that's CloudKit Dashboard: it's the perfect debugging tool because it shows you exactly what content is being stored. If the data you see is bad it means your writing code is bad. If the data there is good but your app isn't showing correctly, it means your reading code is bad. Simple!

Tangent over. Back to the code!

Reading from iCloud with CloudKit: CKQueryOperation and NSPredicate

So far our app takes a recording from the microphone using **AVAudioRecorder** and sends it off to iCloud for storage. You should have just seen that data in the CloudKit Dashboard, so the next step is to write the code that pulls recordings back down to the device.

This is where things get a little bit more complicated, but only a little. You see, there are two ways of writing to CloudKit: a core API and a convenience API. The core API exposes every possible behavior of the system, offering effectively unlimited functionality to do what you want. The convenience API takes a subset of those features and simplifies them, making it easier to learn and use but less powerful.

When we wrote records two chapters ago we used the convenience API, but when it comes to reading we're going to use the core API. This isn't because I enjoy torturing you, there is a legitimate reason: when you read data using the convenience API it automatically downloads all the data for each record. Often that's helpful, because it means you have everything you need to show a record's data. But in our case that would mean downloading the audio for every record every time we loaded our data – and that's a huge waste of resources.

Remember, CloudKit gives you a basic quota of about 64MB per day of asset transfer, and you need to be careful not to waste it. One of the features offered by the core API that is absent from the convenience API is the ability to selectively download records. In our case, that means we want the genre and user comments, but not the audio – we'll fetch that separately, as needed.

Go ahead and select ViewController.swift for editing. We're going to be using the CloudKit framework, so please add this import:

```
import CloudKit
```

This view controller is a **UIViewController** subclass as opposed to a **UITableViewController** subclass, which is intentional as you'll see later. But it does mean we need to create and configure a **UITableView**, then pin it to the edges of our view. Except this time there's a small twist: we're going to use the built-in **topLayoutGuide** property to make the table view pin to the edge of our navigation bar.

First, add this property:

```
var tableView: UITableView!
```

Now put in this new **loadView()** method:

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

  view.backgroundColor = UIColor.whiteColor()

  tableView = UITableView()
  tableView.translatesAutoresizingMaskIntoConstraints = false
  tableView.dataSource = self
  tableView.delegate = self
  view.addSubview(tableView)


view.addConstraints(NSLayoutConstraint.constraintsWithVisualFormat("H
:|[tableView]|", options: .AlignAllCenterX, metrics: nil, views:
["tableView": tableView]))

view.addConstraints(NSLayoutConstraint.constraintsWithVisualFormat("V
:|[guide][tableView]|", options: .AlignAllCenterX, metrics: nil,
views: ["guide": topLayoutGuide, "tableView": tableView]))

}
```

As you will already know, setting the **dataSource** and **delegate** properties above requires that we conform to the **UITableViewDataSource** and **UITableViewDelegate** protocols.

Our table is going to show a series of whistles to users, letting them see at a glance the

genre and user comments before choosing which whistle to listen to. To make this work we're going to create a new class called **Whistle** that will store those two fields, but also an **NSURL** for where the audio is stored when it's downloaded, and the CloudKit record ID that identifies the whistle in iCloud so we can work with it.

In Swift there is usually a discussion as to whether a class or a struct is the right approach when considering data types, but here we have no choice as you'll see later.

Create a new file, choose Cocoa Touch Class, name it Whistle, then make it subclass from NSObject. It doesn't need much code in there, but it does need to use the CloudKit framework. Change the contents of Whistle.swift to this:

```
import CloudKit
import UIKit


class Whistle: NSObject {
  var recordID: CKRecordID!
  var genre: String!
  var comments: String!
  var audio: NSURL!
}
```

Back in ViewController.swift, we need a property that will store an array of Whistle objects so that we can show them in our table view. This is as simple as adding the following property:

```
var whistles = [Whistle]()
```

Every time the view is shown we're going to refresh our data from iCloud using a **loadWhistles()** method. That one does all the complicated CloudKit work so we're going to leave it to last, but we can at least write **viewDidAppear()** and put in a stub for **loadWhistles()**. The **viewWillAppear()** method is going to clear the table view's selection if it has one, then it will use the **dirty** flag we made earlier to call **loadWhistles()** only if it's

needed.

Here's the code:

```swift
override func viewWillAppear(animated: Bool) {
  super.viewWillAppear(animated)

  if let indexPath = tableView.indexPathForSelectedRow {
    tableView.deselectRowAtIndexPath(indexPath, animated: true)
  }

  if ViewController.dirty {
    loadWhistles()
  }
}


func loadWhistles() {
}
```

For the table view, we're going to use some techniques first seen in project 32, which was my Core Spotlight and SFSafariViewController tutorial. Specifically, we're going to use **NSAttributedString** to show text neatly formatted, then use automatic **UITableViewCell** sizing to make each cell fit its contents.

Let's start by using an almost identical **makeAttributedString()** method here, except this version automatically removes user comments if there aren't any:

```swift
func makeAttributedString(title title: String, subtitle: String) ->
NSAttributedString {
  let titleAttributes = [NSFontAttributeName:
UIFont.preferredFontForTextStyle(UIFontTextStyleHeadline),
NSForegroundColorAttributeName: UIColor.purpleColor()]
```

```
  let subtitleAttributes = [NSFontAttributeName:
UIFont.preferredFontForTextStyle(UIFontTextStyleSubheadline)]


  let titleString = NSMutableAttributedString(string: "\(title)",
attributes: titleAttributes)


  if subtitle.characters.count > 0 {

    let subtitleString = NSAttributedString(string: "\n\(subtitle)",
attributes: subtitleAttributes)

    titleString.appendAttributedString(subtitleString)

  }


  return titleString

}
```

Just like in project 32, that uses Dynamic Type to ensure user font choices are respected. Putting that into each table view cell is identical to project 32, except here we're modifying the **numberOfLines** property by hand because we don't have prototype cells to work with:

```
func tableView(tableView: UITableView, cellForRowAtIndexPath
indexPath: NSIndexPath) -> UITableViewCell {

  let cell = tableView.dequeueReusableCellWithIdentifier("Cell",
forIndexPath: indexPath)

  cell.accessoryType = .DisclosureIndicator

  cell.textLabel?.attributedText =  makeAttributedString(title:
whistles[indexPath.row].genre, subtitle:
whistles[indexPath.row].comments)

  cell.textLabel?.numberOfLines = 0

  return cell

}
```

To make that code work, you'll need to register the "Cell" re-use identifier in **viewDidLoad()**,

like this:

```
tableView.registerClass(UITableViewCell.self, forCellReuseIdentifier:
"Cell")
```

And now we need four very simple methods to tell iOS how many rows and sections we need, plus how high they need to be. We'll be using **UITableViewAutomaticDimension** for the height, so all these four are simple:

```
func numberOfSectionsInTableView(tableView: UITableView) -> Int {

  return 1

}


func tableView(tableView: UITableView, numberOfRowsInSection section:
Int) -> Int {

  return self.whistles.count

}


func tableView(tableView: UITableView,
estimatedHeightForRowAtIndexPath indexPath: NSIndexPath) -> CGFloat {

  return UITableViewAutomaticDimension

}


func tableView(tableView: UITableView, heightForRowAtIndexPath
indexPath: NSIndexPath) -> CGFloat {

  return UITableViewAutomaticDimension

}
```

I know – this is all a bit easy now, right? Well, the next part isn't, because the next part is where CloudKit comes in. To make this work you're going to need to meet a few new classes:

• **NSPredicate** describes a filter that we'll use to decide which results to show.

• **NSSortDescriptor** tels CloudKit which field we want to sort on, and whether we want it ascending or descending.

• **CKQuery** combines a predicate and sort descriptors with the name of the record type we want to query. That will be "Whistles" for us, if you remember.

• **CKQueryOperation** is the work horse of CloudKit data fetching, executing a query and returning results.

What complicates **CKQueryOperation** – and at the same time makes it so incredibly powerful – is that is has two separate closures attached to it. One streams records to you as they are downloaded, and one is called when all the records have been downloaded. To handle this, we're going to create a new array that will hold the whistles as they are parsed, and use it inside both closures.

As I said already, one of the advantages of this core API is that we can request only the record keys we want, but it also lets us specify how many results we want to receive from iCloud. Putting all this together, we can write the first part of **loadWhistles()**:

```swift
func loadWhistles() {
  let pred = NSPredicate(value: true)
  let sort = NSSortDescriptor(key: "creationDate", ascending: false)
  let query = CKQuery(recordType: "Whistles", predicate: pred)
  query.sortDescriptors = [sort]

  let operation = CKQueryOperation(query: query)
  operation.desiredKeys = ["genre", "comments"]
  operation.resultsLimit = 50

  var newWhistles = [Whistle]()
```

Our use of **NSPredicate** is trivial right now: we just say "all records that match true," which means "all records." Notice how we set the **desiredKeys** property to be an array of the record keys we want – that's what makes this API so useful.

The next part of the method is going to set a **recordFetchedBlock** closure on our **CKQueryOperation** object. This will be given a one **CKRecord** value for every record that gets downloaded, and we'll convert that into a **Whistle** object. This means pulling out the record ID for the **recordID** property, then reading the **genre** and **comments** values of the dictionary. Both those two values must be converted to strings, because by default they come out as the data type **CKRecordValue?**.

Here's the next part of **loadWhistles()**:

```
operation.recordFetchedBlock = { (record) in
  let whistle = Whistle()
  whistle.recordID = record.recordID
  whistle.genre = record["genre"] as! String
  whistle.comments = record["comments"] as! String
  newWhistles.append(whistle)
}
```

At this point, the last part isn't too hard: we're going to set a **queryCompletionBlock** closure for the query operation. This will be called by CloudKit when all records have been downloaded, and will be given two parameters: a query cursor and an error if there was one. The query cursor is useful if you want to implement paging, because you can use that query cursor to have CloudKit show the next 50 rows, then the next 50 rows, and so on.

We won't be using the cursor here, but we do want to know whether there was any error. Additionally, error or not, we're going to be doing user interface work and this closure might be run on any thread, so we need to push all the work onto the main thread.

And what is that work? Well, if there was no error we're going to overwrite our current **whistles** array with the **newWhistles** array that was built up from downloaded records. We also need to clear the **dirty** flag so we know the update was fetch, then reload the table view. If there was an error, we'll show a **UIAlertController** with a meaningful message to help you debug.

Here's the next part of **loadWhistles()**:

```
operation.queryCompletionBlock = { [unowned self] (cursor, error) in

  dispatch_async(dispatch_get_main_queue()) {

    if error == nil {

      ViewController.dirty = false

      self.whistles = newWhistles

      self.tableView.reloadData()

    } else {

      let ac = UIAlertController(title: "Fetch failed", message:
"There was a problem fetching the list of whistles; please try again:
\(error!.localizedDescription)", preferredStyle: .Alert)

      ac.addAction(UIAlertAction(title: "OK", style: .Default,
handler: nil))

      self.presentViewController(ac, animated: true, completion: nil)

    }

  }

}
```

The last part of the method is the easiest: now that we've created a query, added it to a
CKQueryOperation, then configured its two closures to handle downloading data, it's just a
matter of asking CloudKit to run it. Put this at the end of the method, and you're done:

```
CKContainer.defaultContainer().publicCloudDatabase.addOperation(opera
tion)
```

At this point you're code is in a working state, so you should be able to run the app now and
see the whistle you submitted earlier listed in the table. If not, you should see an error that
gives you an idea of what went wrong. Some things to check:

• Did you see your data in the CloudKit Dashboard?
• Did you name your record type "Whistles" when writing and reading?
• For the Metadata Indexes, did you select Query next to ID, and Sort next to Date Created?

- Is your device definitely online?

Working with CloudKit records: CKReference, fetchRecordWithID, and saveRecord

I promised this was going to be a thorough CloudKit tutorial, and I'm going to keep that promise over the next two chapters, starting here: we're going to learn about references and records, as well as the **fetchRecordWithID()** convenience API.

So far, our app records whistles using **AVAudioRecorder**, submits it to CloudKit, then shows all whistles into a table view. The next step is to let users tap a whistle that interests them so they can see more information, and in our case that will show the user's comments, any suggestions submitted by other users, and a Listen button that downloads the whistle.

The valuable thing about this screen is that it gives me a chance to show you the **CKReference** class, which is used to link records together. Specifically, we're going to build what's called a one-to-many relationship: one whistle can have many suggestions attached to it. Using **CKReference** let us query to find all suggestions for a specific whistle, but it has another brilliant advantage known as *cascade deletes*: if we delete a whistle from our database, iCloud will automatically delete any suggestions that belong to it.

Now, an important warning: as each whistle holds multiple suggestions, and each suggestion is just going to be a string saying something like "I think this is the theme tune from Star Wars," you might be tempted to think "ah, that means our whistle should have an array of strings attached to its record." If you try that, it'll work, and it'll work great – in testing. But when it comes to shipping apps, this approach hits a core problem: conflicts.

A conflict occurs when CloudKit receives two sets of different information, and it's something that record arrays are particularly prone to. You see, if I get the record and it has no suggestions, I might write "that's the Star Wars theme tune." But before I hit Submit, you also download the record, see that it has no suggestions, and write "That's totally the theme tune to a big movie, but I can't remember which one," then hit Submit straight away. In iCloud, that record is now updated to have your (quite useless!) suggestion, so when I submit mine there's a conflict: I'm telling CloudKit the record has one suggestion (mine) and CloudKit thinks it already had one suggestion (yours), so it isn't sure what to do.

Conflict resolution isn't something CloudKit handles for you, because the correct answer depends on your app. In this case, the correct answer is to merge both the arrays, but really

the whole premise is bad – using arrays to reference child objects like this is a terrible idea. This method of referencing is known as forward references, and as you can see it's error-prone. A much better solution are back references, which are where our Whistle record doesn't keep track of its suggestions; instead, the suggestions all know which whistle own it. So, the references go from the child back to the parent, rather than from the parent forward to its child.

Enough theory – time for action. Create a new **UITableViewController** subclass called **ResultsViewController**. This will need to import AVFoundation so we can listen to whistles, and also CloudKit so we can download whistle audio and any user suggestions. So, add these imports now:

```
import AVFoundation
import CloudKit
```

The view controller will need three extra properties: a **Whistle** object that will pass in whichever whistle object was selected in the main view controller, an array of strings for the suggestions (these are *not* stored in the whistle record, remember!), and an **AVAudioPlayer** object that will be used to play the downloaded whistle. Add these now:

```
var whistle: Whistle!
var suggestions = [String]()


var whistlePlayer: AVAudioPlayer!
```

Now let's talk about user interface. This is a **UITableViewController** subclass, because we have structured data that fits neatly into a table view. It's going to have two sections: one for showing the user's comments in big text, and one for showing user suggestions.

We're going to use **titleForHeaderInSection** to provide a title for the second section so that users can see what it's supposed to do. More importantly, the second section is going to have as many rows as there are suggestions, with one extra: a row that says "Add suggestion" so that users can tap that and suggest their own matches for the whistle. That

last row will be the only one that responds to taps, so we'll set the **selectionStyle** of the other cells to be **.None**.

All the cells in this table view wil have their **numberOfLines** property set to 0 so that lines wrap, and we'll use **UITableViewAutomaticDimension** yet again to have table cells figure out their height as needed. That explains all the code, but remember: watch out for the call to **max()**, which ensures the second section always has room for the "Add suggestion" row. Put this code into ResultsViewController.swift:

```swift
override func numberOfSectionsInTableView(tableView: UITableView) ->
Int {

  return 2

}


override func tableView(tableView: UITableView,
titleForHeaderInSection section: Int) -> String? {

  if section == 1 {

    return "Suggested songs"

  }


  return nil

}


override func tableView(tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {

  if section == 0 {

    return 1

  } else {

    return max(1, suggestions.count + 1)

  }

}
```

```swift
override func tableView(tableView: UITableView, cellForRowAtIndexPath
indexPath: NSIndexPath) -> UITableViewCell {

  let cell = tableView.dequeueReusableCellWithIdentifier("Cell",
forIndexPath: indexPath)

  cell.selectionStyle = .None

  cell.textLabel?.numberOfLines = 0


  if indexPath.section == 0 {

    // the user's comments about this whistle

    cell.textLabel?.font =
UIFont.preferredFontForTextStyle(UIFontTextStyleTitle1)


    if whistle.comments.characters.count == 0 {

      cell.textLabel?.text = "Comments: None"

    } else {

      cell.textLabel?.text = whistle.comments

    }

  } else {

    cell.textLabel?.font =
UIFont.preferredFontForTextStyle(UIFontTextStyleBody)


    if indexPath.row == suggestions.count {

      // this is our extra row

      cell.textLabel?.text = "Add suggestion"

      cell.selectionStyle = .Gray

    } else {

      cell.textLabel?.text = suggestions[indexPath.row]

    }

  }


  return cell

}
```

```
override func tableView(tableView: UITableView,
estimatedHeightForRowAtIndexPath indexPath: NSIndexPath) -> CGFloat {

    return UITableViewAutomaticDimension

}


override func tableView(tableView: UITableView,
heightForRowAtIndexPath indexPath: NSIndexPath) -> CGFloat {

    return UITableViewAutomaticDimension

}
```

At this point your iOS career, every line of that should be second nature – I'm only repeating it here to help jog your memory. The real work happens when a user taps on the "Add suggestion" table view cell. This code needs to show a **UIAlertController** with a text field prompting the user to enter their suggestion. This code is a bit clumsy: if you haven't already ready my addTextFieldWithConfigurationHandler tutorial that was inside project 5, that's a good place to start.

To summarise, here's what we're going to do:

 • We're going to hook into the **didSelectRowAtIndexPath** method of our table view, which will be triggered when any row is tapped.
 • If the row that was tapped was not the last row in the second section (the "Add suggestion" row) we'll exit the method.
 • We'll create a **UIAlertController** in the style **.Alert**, then add a text field to it.
 • We'll add a Submit button to the alert that, when tapped, will submit the suggestion if the text field has any text.
 • Because we configure the text field in one closure and submit it in another, we need to create it outside of both – just like in project 5.
 • As an added touch, we're going to deselect the row that was tapped, making it highlighted only temporarily.

Here's the code:

```
override func tableView(tableView: UITableView,
```

```
didSelectRowAtIndexPath indexPath: NSIndexPath) {

  guard indexPath.section == 1 && indexPath.row == suggestions.count
else { return }


  tableView.deselectRowAtIndexPath(indexPath, animated: true)


  let ac = UIAlertController(title: "Suggest a song…", message: nil,
preferredStyle: .Alert)

  var suggestion: UITextField!


  ac.addTextFieldWithConfigurationHandler { (textField) -> Void in

    suggestion = textField

    textField.autocorrectionType = .Yes

  }


  ac.addAction(UIAlertAction(title: "Submit", style: .Default)
{ (action) -> Void in

    if suggestion.text?.characters.count > 0 {

      self.addSuggestion(suggestion.text!)

    }

  })


  ac.addAction(UIAlertAction(title: "Cancel", style: .Cancel,
handler: nil))

  presentViewController(ac, animated: true, completion: nil)

}
```

Don't worry that **self.addSuggestion(suggestion.text!)** will error at this point – we haven't written that yet. As I mentioned in my **what's new in Swift 2** article, Swift 2 measures strings using **characters.count** – a huge improvement, in my opinion.

It's time for some CloudKit action again, and this time we're going to be using the

CKReference class to link a user's suggestion to the whistle they were reading about. When you create a **CKReference** you need to provide it two things: a record ID to link to, and a behavior to trigger when that linked record is deleted. We already have the record ID to link to because we're storing it in the **whistle** property, and for the action to trigger we'll use **.DeleteSelf** – when the parent whistle is deleted, delete the child suggestions too.

CKReferences, like **CKAssets**, can be placed directly into a **CKRecord**, which means the first part of **addSuggestion()** is easy:

```
func addSuggestion(suggest: String) {

  let whistleRecord = CKRecord(recordType: "Suggestions")

  let reference = CKReference(recordID: whistle.recordID,
action: .DeleteSelf)

  whistleRecord["text"] = suggest

  whistleRecord["owningWhistle"] = reference


  // more code to come!

}
```

Note that I'm using the name "Suggestions" as the record type for our user suggestions, and **owningWhistle** as the key for that reference value.

The second part of **addSuggestion()** isn't much more difficult, because we'll use **saveRecord()** to post that new record back to iCloud, then check for errors.

Remember: CloudKit tells us when the save completes by executing our code as a closure, and that could be running on any thread. We want to either reload the table view or show a message depending on whether there was an error, but regardless this work needs to be pushed to the main thread as it involves user interface changes.

Here's the second part of **addSuggestion()** – put this where the **more code to come!** comment is:

```
CKContainer.defaultContainer().publicCloudDatabase.saveRecord(whistle
```

```
Record) { [unowned self] (record, error) -> Void in

  dispatch_async(dispatch_get_main_queue()) {

    if error == nil {

      self.suggestions.append(suggest)

      self.tableView.reloadData()

    } else {

      let ac = UIAlertController(title: "Error", message: "There was
a problem submitting your suggestion: \
(error!.localizedDescription)", preferredStyle: .Alert)

      ac.addAction(UIAlertAction(title: "OK", style: .Default,
handler: nil))

      self.presentViewController(ac, animated: true, completion: nil)

    }

  }

}
```

Note that I append the user's new suggestion to the existing **suggestions** array so they see it has been posted successfully.

There are two more tasks to do before this view controller is complete. First, when the view is loaded, we need to fetch the existing list of user suggestions and show them in the table. Second, we need to let users download and listen to each whistle so they can try to guess what it is.

To download all suggestions that belong to a particular whistle we need to create another **CKReference**, just like before. We can then pass that into an **NSPredicate** that will check for suggestions where **owningWhistle** matches that predicate. This time we're going to sort by **creationDate** ascending so that oldest suggestions appear first, but otherwise this isn't tricky – here's the first part of the new **viewDidLoad()** method:

```
override func viewDidLoad() {

  super.viewDidLoad()
```

```
   title = "Genre: \(whistle.genre)"

   navigationItem.rightBarButtonItem = UIBarButtonItem(title:
"Download", style: .Plain, target: self, action: "downloadTapped")


   tableView.registerClass(UITableViewCell.self,
forCellReuseIdentifier: "Cell")


   let reference = CKReference(recordID: whistle.recordID, action:
CKReferenceAction.DeleteSelf)

   let pred = NSPredicate(format: "owningWhistle == %@", reference)

   let sort = NSSortDescriptor(key: "creationDate", ascending: true)

   let query = CKQuery(recordType: "Suggestions", predicate: pred)

   query.sortDescriptors = [sort]


   // more code to come!
```

When it comes to running this query, we can aren't going to take the same approach from the last chapter: **CKQueryOperation** isn't needed here because we want all the fields, which means we can use the much easier convenience API: **performQuery()**. Tell this method what query to run and where it should be run (or nil for the default), and it will return back either results or an error.

The remainder of **viewDidLoad()** is easy thanks to this convenience API, although I have cheated a bit by calling out to an as-yet unwritten **parseResults()** method. Here it is:

```
CKContainer.defaultContainer().publicCloudDatabase.performQuery(query
, inZoneWithID: nil) { [unowned self] (results, error) -> Void in

   if error == nil {

     if let results = results {

       self.parseResults(results)

     }

   } else {

     print(error!.localizedDescription)
```

```
    }
}
```

If that fails to fetch the suggestions, it prints a message to the Xcode log – see if you can have a go at making it a bit smarter.

The last step in handling suggestions is to write that **parseResults** method. This gets called once the record results array has been unwrapped, so we know we'll definitely get a list of records through. It's then just a matter of looping through that array, pulling out the **text** property of each record, and adding it to our **suggestions** string array. To make things safer on multiple threads, we'll actually use an intermediate array called **newSuggestions** – it's never smart to modify data in a background thread that is being used on the main thread.

Here's the **parseResults()** method:

```
func parseResults(records: [CKRecord]) {
  var newSuggestions = [String]()

  for record in records {
    newSuggestions.append(record["text"] as! String)
  }

  dispatch_async(dispatch_get_main_queue()) {
    self.suggestions = newSuggestions
    self.tableView.reloadData()
  }
}
```

The final task for this view controller is to let users download and listen to whistles from other users. We already set up a right bar button item named "Download" in **viewDidLoad()**, but we haven't yet written the **downloadTapped()** method it will call.

This new method needs to:

1. Replace the button with a spinner so the user knows the data is being fetched.
2. Ask CloudKit to pull down the full record for the whistle, including the audio.
3. If it successfully gets audio for the whistle, attach it to the **Whistle** object of this view controller.
4. Create a new right bar button item that says "Listen" and will call **listenTapped()**.
5. If something goes wrong, show a meaningful error message and put the Download button back.

Fetching whole records is done through a simple CloudKit convenience API: **fetchRecordWithID()**. Once that fetches the complete whistle record, we can pull out the **CKAsset** and read its **fileURL** property to know where CloudKit downloaded it to. Please note: this download is just a cache – CloudKit will automatically remove downloaded files at a later date.

Remember, all user interface work needs to be pushed onto the main thread, and you should be careful to handle your CloudKit errors properly. I put a comment in this code that you should replace with an error of your choosing – don't forget!

Here's the **downloadTapped()** method:

```
func downloadTapped() {
  let spinner = UIActivityIndicatorView(activityIndicatorStyle:
UIActivityIndicatorViewStyle.Gray)
  spinner.tintColor = UIColor.blackColor()
  spinner.startAnimating()
  navigationItem.rightBarButtonItem = UIBarButtonItem(customView:
spinner)


CKContainer.defaultContainer().publicCloudDatabase.fetchRecordWithID(
whistle.recordID) { [unowned self] (record, error) -> Void in

    if error == nil {
      if let record = record {
```

```
        if let asset = record["audio"] as? CKAsset {

            self.whistle.audio = asset.fileURL


            dispatch_async(dispatch_get_main_queue()) {

                self.navigationItem.rightBarButtonItem =
UIBarButtonItem(title: "Listen", style: .Plain, target: self, action:
"listenTapped")

            }

        }

    } else {

        dispatch_async(dispatch_get_main_queue()) {

            // meaningful error message here!

            self.navigationItem.rightBarButtonItem =
UIBarButtonItem(title: "Download", style: .Plain, target: self,
action: "downloadTapped")

        }

    }

  }

}
```

There's only one more thing to do before this view controller is complete, and that's to write the **listenTapped()** method. This is almost identical to the "Tap to Play" button we already used in **RecordWhistleViewController**, so I'm not going to explain what it does here:

```
func listenTapped() {

  do {

    whistlePlayer = try AVAudioPlayer(contentsOfURL: whistle.audio)

    whistlePlayer.play()

  } catch {

    let ac = UIAlertController(title: "Playback failed", message:
"There was a problem playing your whistle; please try re-recording.",
```

```
preferredStyle: .Alert)

    ac.addAction(UIAlertAction(title: "OK", style: .Default, handler:
nil))

    presentViewController(ac, animated: true, completion: nil)

  }

}
```

That's **ResultsViewController** complete. All you need to do now is go back to ViewController.swift and tell it to show a new **ResultsViewController** when any whistle is tapped, passing in the **Whistle** object so it knows what to show:

```
func tableView(tableView: UITableView, didSelectRowAtIndexPath
indexPath: NSIndexPath) {

  let vc = ResultsViewController()

  vc.whistle = whistles[indexPath.row]

  navigationController?.pushViewController(vc, animated: true)

}
```

Go ahead and run the app now, then submit a suggestion for your whistle. Once that's done, go to the CloudKit Dashboard to make sure the record type was created as expected (i.e., that everything works!), then check the Metadata Indexes boxes next to Query for ID and Sort for Date Created, just like you did for Whistles.

Delivering notifications with CloudKit push messages: CKSubscription and saveSubscription

You're probably feeling very tired at this point: this has been a long tutorial and you've had to learn a lot. Fortunately, this last chapter is a bonus – you don't need to read this to have a great CloudKit app, but I do add some neat CloudKit technologies here that make the whole experience better.

So far, the app lets users record a whistle using **AVAudioRecorder**, send it off to CloudKit, download whistles others have posted, then write suggestions for what song they think it is. What we're going to add now is the ability for users to register themselves as experts for particular genres - they can say "I know all about jazz and blues music." When they do that, we'll automatically tell them when a new whistle has been posted in one of those categories, and they can jump right into the app to have a go at identifying it.

We're going to do this with one of the most important technologies in iOS, called push notifications. These are alerts that are delivered straight to the lock screens of users whenever something interesting happens, and the app usually isn't running at the time. Push is so important to iOS that it comes built into CloudKit, and it's done so elegantly that this tutorial will be over in no time. In short: don't worry, the end is in sight!

Create a new **UITableViewController** subclass called **MyGenresViewController**. Add a CloudKit import to it, then add this property:

```
var myGenres: [String]!
```

We'll use that to track the list of genres the user considers themself an expert on.

Before we starting adding code to this new view controller, we need to make two small changes in ViewController.swift. First, add this in **viewDidLoad()**:

```
navigationItem.leftBarButtonItem = UIBarButtonItem(title: "Genres",
style: .Plain, target: self, action: "selectGenre")
```

Second, add the following method that allows users to choose genres:

```
func selectGenre() {
  let vc = MyGenresViewController()
  navigationController?.pushViewController(vc, animated: true)
}
```

Now, back to MyGenresViewController.swift. We can split this code into two parts: all the bits that handle users selecting their experts genres and saving that list, and the CloudKit part that tells iCloud we want to be notified when a new whistle has been published that might be of interest to this user.

Letting the user choose which genres interest them is easy: we're going to use **NSUserDefaults** to save an array of their expert genres. If there isn't one already saved, we'll create an empty array. We're also going to use a right bar button item with the title "Save" that handles the CloudKit synchronization. More on that later; for now, here's the **viewDidLoad()** method that handles loading saved genres:

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

  let defaults = NSUserDefaults.standardUserDefaults()
  if let savedGenres = defaults.objectForKey("myGenres") as? [String]
{
    myGenres = savedGenres
  } else {
    myGenres = [String]()
  }

  title = "Notify me about…"
```

```
   navigationItem.rightBarButtonItem = UIBarButtonItem(title: "Save",
style: .Plain, target: self, action: "saveTapped")

   tableView.registerClass(UITableViewCell.self,
forCellReuseIdentifier: "Cell")

}
```

When it comes to how many sections and rows we have, we're going to specify 1 and the value of **SelectGenreViewController.genres.count** – that static property contains all the genres used in the app, so re-using it here is perfect:

```
override func numberOfSectionsInTableView(tableView: UITableView) ->
Int {

   return 1

}


override func tableView(tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {

   return SelectGenreViewController.genres.count

}
```

Now for the interesting part: we want users to be able to tap on rows that they like, and have iOS show a checkmark next to them. Once a genre is added to the **myGenres** array, we can check whether it's in there using the **contains()** of that array – if the array contains the genre, we put a check next to it.

Here's the **cellForRowAtIndexPath** method that does just that:

```
override func tableView(tableView: UITableView, cellForRowAtIndexPath
indexPath: NSIndexPath) -> UITableViewCell {

   let cell = tableView.dequeueReusableCellWithIdentifier("Cell",
forIndexPath: indexPath)


   let genre = SelectGenreViewController.genres[indexPath.row]
```

```
  cell.textLabel?.text = genre


  if myGenres.contains(genre) {

    cell.accessoryType = .Checkmark

  } else {

    cell.accessoryType = .None

  }


  return cell

}
```

The other part of this approach is catching **didSelectRowAtIndexPath** so that we check and unchecks rows as they are tapped, adding and removing them from the **myGenres** array as necessary. Adding things to an array is as simple as calling the **append()** method on the array, but removing takes a little more hassle: we need to use **indexOf()** to find the location of an item in the array, and if that returns a value then we use **removeAtindex()** to remove the item from the array at that index.

As a finishing touch, we'll call **deselectRowAtIndexPath()** to deselect the selected row, so it highlights only briefly. That's all there is to it – here's the **didSelectRowAtIndexPath** method:

```
override func tableView(tableView: UITableView,
didSelectRowAtIndexPath indexPath: NSIndexPath) {

  if let cell = tableView.cellForRowAtIndexPath(indexPath) {

    let selectedGenre =
SelectGenreViewController.genres[indexPath.row]


    if cell.accessoryType == .None {

      cell.accessoryType = .Checkmark

      myGenres.append(selectedGenre)

    } else {

      cell.accessoryType = .None
```

```
        if let index = myGenres.indexOf(selectedGenre) {

            myGenres.removeAtIndex(index)

        }

    }

}


    tableView.deselectRowAtIndexPath(indexPath, animated: false)

}
```

And now for the challenging part: when users click Save, we want to write their list of genres to **NSUserDefaults**, then send them all to iCloud. Then, when new whistles arrive that match a user's selected genres, we want to notify them with a push message.

Thanks to what was I'm sure many months of effort from Apple engineers, registering for push messages is a breeze thanks to a class called **CKSubscription**. This lets you configure a query to run on the iCloud servers, and as soon as that query matches something it will automatically trigger a push message. In our case, that query will be "when anyone publishes a whistle in a genre we care about."

But first: we need to flush out any existing subscriptions so that we don't get duplicate errors. In the interests of keeping it brief, the easiest way to do this is by calling **fetchAllSubscriptionsWithCompletionHandler()** to fetch all **CKSubscriptions**, then passing each of them into **deleteSubscriptionWithID()**. As always, it's up to you to do useful error handling – I've done it enough for you already, so you're most of the way there.

Note: the two parameters you get from **fetchAllSubscriptionsWithCompletionHandler()** are an optional array of subscriptions and an error if one occurred. You need to unwrap the optional array, because the user might not have any subscriptions. Here's an initial version of **saveTapped()**:

```
func saveTapped() {
  let defaults = NSUserDefaults.standardUserDefaults()
```

```swift
    defaults.setObject(myGenres, forKey: "myGenres")


    let database = CKContainer.defaultContainer().publicCloudDatabase


    database.fetchAllSubscriptionsWithCompletionHandler() { [unowned
self] (subscriptions, error) -> Void in
      if error == nil {
        if let subscriptions = subscriptions {
          for subscription in subscriptions {

database.deleteSubscriptionWithID(subscription.subscriptionID,
completionHandler: { (str, error) -> Void in
              if error != nil {
                // do your error handling here!
                print(error!.localizedDescription)
              }
            })
          }


          // more code to come!
        }
      } else {
        // do your error handling here!
        print(error!.localizedDescription)
      }
    }
}
```

Again, please do put in some user-facing error messages telling users what's going on – they won't see the Xcode log messages, and it's important to keep them informed.

There's just one more thing to do before this entire project is done, and that's registering

subscriptions with iCloud and handling the result. You tell it what condition to match and what message to send, then call the **saveSubscription()** method, and you're done – iCloud takes care of the rest. Normally you'd need to opt into push messages and create a push certificate, but again Xcode and iCloud have taken away all this work when you enabled CloudKit what feels like long ago.

To make this work, we're going to loop through each string in the **myGenres** array, create a predicate that searches for it, then use that to create a **CKSubscription** using the option **.FiresOnRecordCreation**. That means we want this subscription to be informed when any record is created that matches our genre predicate.

If you want to attach a visible message to a push notification, you need to create a **CKNotificationInfo** object and set its **alertBody** property. If you want an invisible push (one that launches your app in the background silently) you should set **shouldSendContentAvailable** to be true instead.

We're going to set a notification message that contains the genre that changed by using Swift's string interpolation. We're also going to use the value **UILocalNotificationDefaultSoundName** for the **soundName** property, which will trigger the default iOS tri-tone sound when the message arrives. With that done, we can call **saveSubscription()** to send it off to iCloud, then handle any error messages that come back.

That's it – here's the code to put in place of the **more code to come!** comment:

```
for genre in self.myGenres {
    let predicate = NSPredicate(format:"genre = %@", genre)

    let subscription = CKSubscription(recordType: "Whistles",
predicate: predicate, options: .FiresOnRecordCreation)


    let notification = CKNotificationInfo()

    notification.alertBody = "There's a new whistle in the \(genre)
genre."

    notification.soundName = UILocalNotificationDefaultSoundName


    subscription.notificationInfo = notification
```

```
    database.saveSubscription(subscription) { (result, error) -> Void
in

        if error != nil {

            print(error!.localizedDescription)

        }

    }

}
```

If you run the app now you'll be able to save your genre preferences and send them off to iCloud, but you won't get any push messages through just yet. That's because although iCloud is now totally configured to send push messages when interesting things happened, the user hasn't opted in to receive them. As you might imagine, Apple doesn't want to send push messages to users who haven't explicitly opted in to receive them, so we need to ask for push message permission.
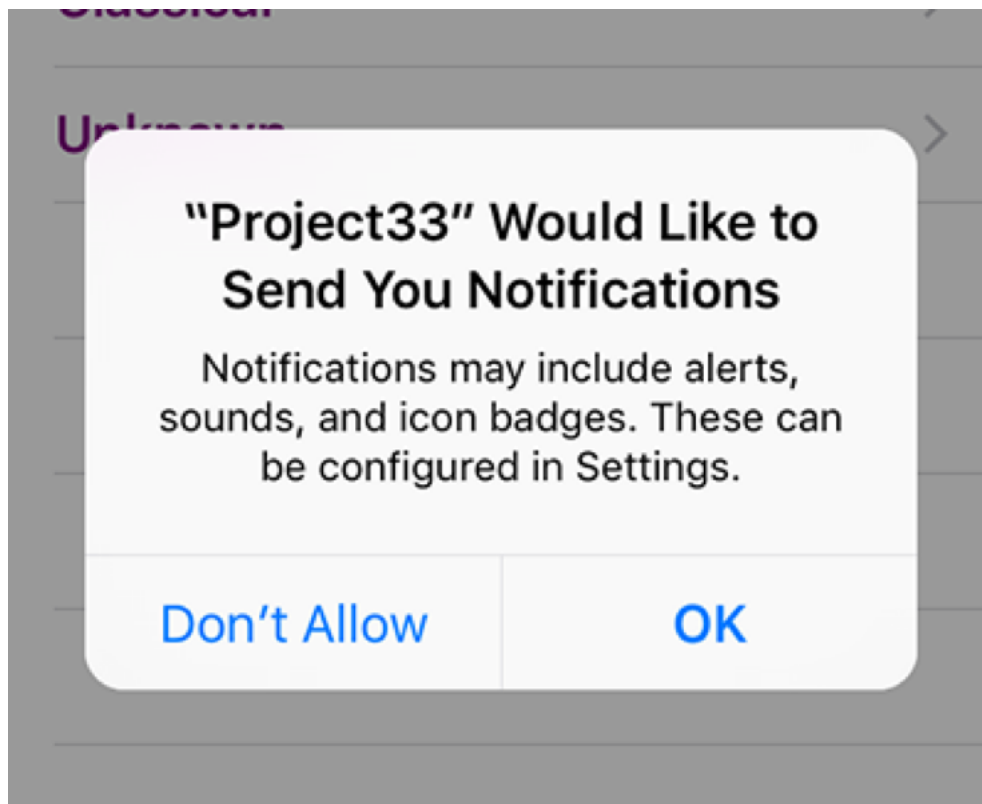
When you request permission to show notifications (local or remote) Apple will display a screen

Go to AppDelegate.swift and put this code into the **didFinishLaunchingWithOptions** method:

```
let notificationSettings = UIUserNotificationSettings(forTypes:
[.Alert, .Sound], categories: nil)

UIApplication.sharedApplication().registerUserNotificationSettings(no
tificationSettings)

UIApplication.sharedApplication().registerForRemoteNotifications()
```

The first line sets up a **UIUserNotificationSettings** object that says we want to show an alert and play a sound. The second line is what asks the user "do you want to let this app send you push messages?" The third line is what connects your phone to iCloud via the Apple Push Notification Service, or APNS – it registers your unique device token with iCloud so that you can be notified when something happens.

For the sake of completion, you could optionally also catch the **didReceiveRemoteNotification** message sent to your app delegate, which is called if a push message arrives while the app is running. Something like this ought to do the trick:

```
func application(application: UIApplication,
didReceiveRemoteNotification userInfo: [NSObject : AnyObject]) {

  if let pushInfo = userInfo as? [String: NSObject] {

    let notification =
CKNotification(fromRemoteNotificationDictionary: pushInfo)


    let ac = UIAlertController(title: "What's that Whistle?",
message: notification.alertBody, preferredStyle: .Alert)

    ac.addAction(UIAlertAction(title: "OK", style: .Default, handler:
nil))


    if let nc = window?.rootViewController as? UINavigationController
{
```

```
        if let vc = nc.visibleViewController {
            vc.presentViewController(ac, animated: true, completion: nil)
        }
    }
}
```

Make sure you add the usual CloudKit import to your app delegate in order for that to compile.

That's it. No, really – this epic project is now done. Go ahead and run the app on a real iOS device, register for certain genres, then quit the app – unplug your phone, lock it, and put it to one side. Now launch the app in the simulator and create a new whistle in one of the genre you just marked, and you'll get your push.

Wrap up

This was an epic tutorial: epic in length, epic in breadth, and I hope you'll agree epic in what we've accomplished. You've built another real app, you've learned about **AVAudioRecorder**, **CKQuery**, **CKRecord**, **CKAsset**, **CKQueryOperation**, **CKSubscription**, **NSPredicate**, **NSSortDescriptor** and more, while also having some bonus practise working with **UIStackView**, **UITableView** and **NSAttributedString**.

So yes, the tutorial was long, but even though you're tired I'd like to think you're pleased with the end result. Take a break, perhaps even a couple of days, then come back and have a think about how you could improve this project. It's so big there are lots of possibilities, not least:

 • **ViewController** is a subclass of **UIViewController** rather than **UITableViewController**. Make it show something useful while fetching the iCloud records rather than just a blank table.
 • If the iCloud fetch fails, we tell the user. How about adding a "Retry" button to the user interface?
 • We made the **Whistle** class inherit from **NSObject**. Can you make it conform to the **NSCoding** protocol? You might find my guide to NSCoding and NSUserDefaults in Swift useful.
 • Fix the **AddCommentsViewController** class so that it correctly adjusts the text view when the keyboard appears. I already showed you how to do this in my tutorial on adjusting the keyboard uising NSNotificationCenter.
 • Stop people from posting too many line breaks in their comments, or at least trim the comments when shown in the main table view.

Of course, the other thing you could do is perhaps the most important of all: go back through all your code and make sure you handle CloudKit errors gracefully. Seriously, put your hand in the air and repeat after me: I promise to show meaningful iCloud errors to my users. Now, I know you didn't actually do that, but you really ought to at least mean it. As Apple has said, handling errors is the difference between working apps and non-working apps, and you don't want a non-working app, do you?

Project 34

Four in a Row

Let iOS take over the AI in your games using new iOS 9 GameplayKit features.

Setting up

One of the most powerful features Apple introduced in iOS 9 is called GameplayKit. It's a library designed to handle non-drawing game functionality such as artificial intelligence, path finding and randomness, and it is pretty dazzling in its scope, so I was really looking forward to write a tutorial about it.

In this project, we're going to create a Four-in-a-Row (4IR) game, and I'm going to be honest with you: I've cheated a bit. You see, Apple already released some sample code for a 4IR game based on GameplayKit, and it works pretty well. Why, then, am I choosing to write a tutorial based on it? Well, for some reason known only to Apple, the source code for the project isn't up to their usual standard. Not only is it in Objective C, but it includes C functions like **memcpy()**, it uses **CABasicAnimation** and **CAShapeLayer** when regular **UIView** functionality would do, and includes a 90-line method to detect wins that can be replaced with code a third that size and easier to understand.

Frustrating things further, this sample code is what's used to document GameplayKit, so you're kind of stuck trying to learn about a very large new technology while studying an unfriendly project, or reading the documentation… that's about the same unfriendly project. I wanted to produce a project that was easier to understand and easier to learn, then produce a tutorial that explained how it all worked.

So, I took the Objective C code and rewrote it in Swift. I then simplified the structure to make it more useful for learners, renamed some methods to make more sense, then cleaned up the user interface. Where it wasn't too strange I have tried to keep Apple's original structure, so if you choose to check out their original source code you won't be too lost – look for FourInARow in the Apple sample code. I accept any and all blame for bugs introduced in the Swift conversion process: if you spot any problems, **let me know on Twitter @twostraws** and I'll get them fixed.

You might well say, "well, if you didn't like the 4IR game, how about Apple's DemoBots sample code? That's really cool, and it uses GameplayKit!" Yes, it does use GameplayKit. But it's also made up of 84 Swift files, 6,952 images, 14 SKS files for effects and scenes, and a custom shader. Cool: yes. Easy to learn from: not really. By all means download it, but this tutorial is aimed at people just getting started with GameplayKit.

Please note: this is a Hacking with Swift Level 2 project, which means I expect you to have finished projects 1-30 first. If you have not done so, please **click here now**, then return here when you've levelled up. They are all free, so there's nothing holding you back.

So, please go ahead and create a new project in Xcode 7, choosing the Single View Application template. Name it Project 34, choose Swift for your language, and iPad for the device. When it's created, please lock the app orientation to landscape.

Creating the interface with UIStackView

Once you start using UIStackView it's hard to stop, so naturally I wanted it in this project even though we're making a game. The nature of 4IR games is that you have rows and columns, and we're going to create a **UIStackView** to host the columns. If I didn't want to animate the chips falling into the board we'd be using stack views for the columns too!

Open Main.storyboard in Interface Builder, then embed the existing view controller inside a navigation controller. Now select the navigation controller you just created, find its Navigation Bar in the document outline, then deselect Translucent in the attributes inspector – we don't want our game going behind the navigation bar, after all.

From the object library drag a Horizontal Stack View into your view controller, then resize it to fill the entire view up to the bottom of the navigation bar. When that's done, go to Editor > Resolve Auto Layout Issues > Reset to Suggested Constraints to add in the Auto Layout constraints required to keep it filling the view.

Now drag seven buttons into the stack view. By default, the stack view will show the first one large and all the others as small as possible, but you should select the stack view, go to the attributes inspector, then change Distribution to Fill Equally. While you're there, give Spacing a value of 2.

These seven buttons will be used to store the columns in our game, so to make them stand out I'd like you to select them all and give them all a white background color. Button background color is quite a way down the attributes inspector, so you will need to scroll to find it. Now clear the text for all the buttons, making them just large white spaces that respond to taps.

Now I need you to select each of the buttons in order and give them increasing Tag values. The one on the far left can keep its Tag of 0, but the second one should have a Tag of 1, then 2, 3, 4, 5 and 6. This will be used to identify which button was tapped later on.

To make the buttons stand out as columns, select the view itself (you might need to use the document outline view for this) then give it a gray background color. Don't try to give the stack view a background color – it's doesn't actually do any drawing, so your background color will be ignored.
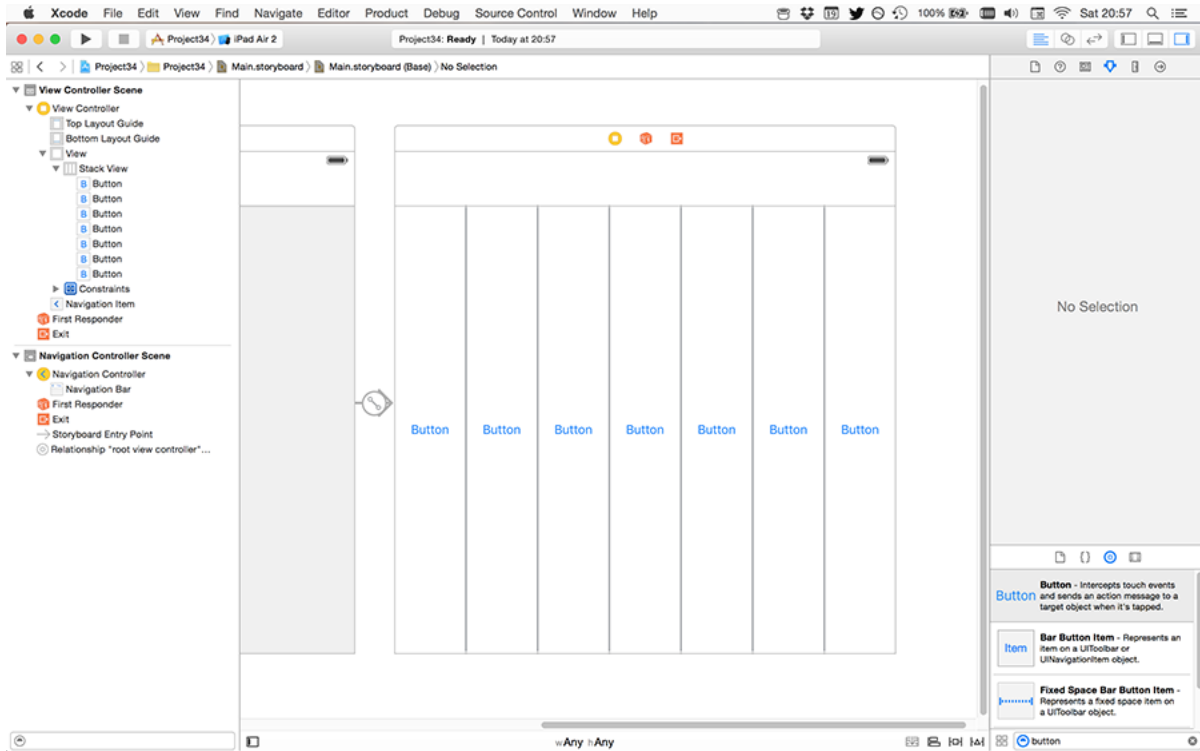
Your user interface should have seven buttons inside a stack view, placed over a gray background.

There are two more things to do in Interface Builder before we can get on with some code. The first is to create IBOutlets for those buttons so we can control them from code, but rather than create individual outlets we're going to use something called an **IBOutletCollection**. These are just IBOutlet arrays that work like normal Swift arrays, except in IB you connect multiple outlets to the same thing.

To create an outlet collection, switch to the assistant editor by pressing Alt+Cmd+Return, then Ctrl+drag from the first button on the left from IB into your source code, just before the **viewDidLoad()** line. When you release the action/outlet menu will appear, and I'd like you to choose Outlet Collection. Note: if you see only Outlet and Outlet Collection in the list (i.e., Action is missing) it means you probably selected the stack view rather than a button, so try again!

Name the outlet collection **columnButtons** and click Connect. Now Ctrl+drag from the other six buttons, but this time connect them to the same **@IBOutlet** that was just created – you need to hover over the **var columnButtons: [UIButton]!** part in order for this to work.

```
 4 //
 5 //    Created by Hudzilla on 19/09/2015.
 6 [Connection  Outlet Collection  ⌃]  Paul Hudson. All rights
 7 Object  ◯ View Controller
 8 Name  columnButtons
 9 Type  UIButton  ⌄
10 [Cancel]                    [Connect]
11 class ViewController: UIViewController {
12
13     override func viewDidLoad() {
14         super.viewDidLoad()
15         // Do any additional setup after loa
```
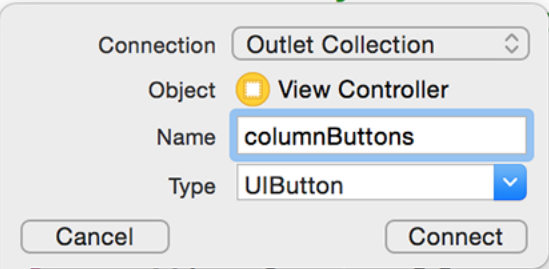
Creating an IBOutletCollection in Interface Builder puts all your outlets into a single array.

The second change to make is to hook up an IBAction for those buttons. We set each of them to have their own Tag because we're going to use the same action for seven buttons – the tag will be used to figure out which button was tapped.

So, Ctrl+drag from the first button into some free space below **didReceiveMemoryWarning()**, then create an action called **makeMove**. Make sure you change the Type value to be **UIButton** rather than **AnyObject** – this will be used later. Now Ctrl+drag from the other six buttons onto that **makeMove()** method, and we're done with Interface Builder.

Preparing for basic play

We're going to put together the absolute basics required to represent a 4IR game on iOS. Because of the way GameplayKit works, it's especially important to keep a good separation between your model (the state of the game) and your view (how things look). Over the course of this project we're going to produce three different data models: one for the game board (stores the complete game state), one for players (for storing their color and name), and one for a "move" – used when it comes to building an AI.

We're going to start with the game board now, so add a new file to the project: choose Cocoa Touch Class and click Next, name it Board and make it subclass from **NSObject**, then click Next and Create.

To begin with, this **Board** class is just going to have two properties: one for tracking the width of the board, and one to track the height. We'll be adding more later, but for now add these two:

```
static var width = 7
static var height = 6
```

Now go back to ViewController.swift. Like I said, there's an important distinction between the model and the view. That **Board** class will hold our model, which means it will store where all the chips are and who is winning. This view controller will store its own array of where the chips are, but it does this so it can draw the view correctly.

A "chip" in this case is a 4IR piece, either red or black. We'll be using **UIViews** for this purpose, setting a high corner radius so they look like circles – it's a simple trick, but effective. To store this, we'll need an array of arrays. That is, we'll need an array to store each column, and another array to hold all those column arrays. We're also going to add a property to store a **Board** object, using that class we just created. So, add these two properties to ViewController.swift:

```
var placedChips = [[UIView]]()
var board: Board!
```

When the game is first run, we need to populate that **placedChips** property with empty arrays. When a game is started, we'll wipe those arrays (removing any views that were added), and also create a new game board to track progress. This is all done in two methods: **viewDidLoad()** to do the initial set up, and **resetBoard()** to create the game board variable and clear out any views from the previous game. Add these two now:

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

  for _ in 0 ..< Board.width {
    placedChips.append([UIView]())
  }

  resetBoard()
}


func resetBoard() {
  board = Board()

  for i in 0 ..< placedChips.count {
    for chip in placedChips[i] {
      chip.removeFromSuperview()
    }

    placedChips[i].removeAll(keepCapacity: true)
  }
}
```

There won't be any views to clear out in the first run of that method, but we'll be adding more to the **resetBoard()** method later so it makes sense to call it during the first run.

Now for some complicated coding: we need to give the gameboard enough logic to keep track of chips. We need to make the **Board** class do five new things:

1. Report what chip is at a specific row and column.
2. Set a particular slot to contain a chip, i.e. "make row 4 column 3 a red chip."
3. Determine whether the player can make a move in a column.
4. Find the next empty slot in a column, which is where a chip would land if it were dropped in there.
5. Add a chip to a column at the next available space.

On top of that, we also need to create the default array of slots, which will be filled with blanks to begin with.

First up, though, we need to define a new enum that will be used to store the current state of each slot. A slot can either contain a red chip, a black chip, or no chip, so we're going to encapsulate that in an enum called **ChipColor**. Put this enum in Board.swift, just before the class definition:

```
enum ChipColor: Int {
    case None = 0
    case Red
    case Black
}
```

Note that I have given it a raw type of Int, then given the first value a specific number: 0. When you do this, Swift will assign the following values an auto-incremented number, which means Red will be 1 and Black will be 2. This will be important later!

With that enum defined, we can create an array of **ChipColors**, and initialize it when the class is instantiated. Add this property and method to the **Board** class:

```
var slots = [ChipColor]()
```

```
override init() {

  for _ in 0 ..< Board.width * Board.height {

    slots.append(.None)

  }


  super.init()

}
```

This array has only one dimension, which means it's a regular array rather than an array of arrays. One-dimensional arrays are less easy to work with but significantly faster, which is important because we'll be using this array *a lot* in this project.

The custom **init()** method pre-fills the array with **.None**, which means all the slots have no chip in them by default, as you would expect.

Now that we have a **slots** array, we can create the two most frequently used methods in this game: **chipInColumn()** and **setChip()**. The first is used to read the chip color of a specific slot, and the second is used to set the chip color of a specific slot. As **slots** is a one-dimensional array, you need to do a small amount of maths to find the correct row/column: you multiply the column number by the height of the board, then add the row.

Add these two methods to the **Board** class:

```
func chipInColumn(column: Int, row: Int) -> ChipColor {

  return slots[row + column * Board.height]

}
```

```
func setChip(chip: ChipColor, inColumn column: NSInteger, row:
NSInteger) {

  slots[row + column * Board.height] = chip;

}
```

Those two methods will be used extensively to check what moves are valid, so it's good to keep them as small as possible.

Next up: determining whether a player can place a chip in a column. To make this work, we're going to use a helper method called **nextEmptySlotInColumn()**, which will return the first row number that contains no chips in a specific column. With that helper method in place, we can check whether a player can move in a column just by checking to see if there is an empty slot there.

The **nextEmptySlotInColumn()** helper method works by counting up in a column, from 0 up to the height of the board. For every slot, it calls **chipInColumn()** to see what chip color is there already, and if it gets back **.None** it means that row is good to use. If it gets to the end of the board without finding a **.None** it will return **nil** – this column has no free slots.

Here's the code:

```
func nextEmptySlotInColumn(column: Int) -> Int? {

  for row in 0 ..< Board.height {

    if chipInColumn(column, row: row) == .None {

      return row

    }

  }


  return nil;
}
```

As promised, figuring out whether a player can play a particular column is now easy: we just call **nextEmptySlotInColumn()** and check whether it returns **nil** or not, like this:

```
func canMoveInColumn(column: Int) -> Bool {

  return nextEmptySlotInColumn(column) != nil
}
```

The last method we need to add to our model at this time is **addChip()**, which blends two of our above methods: find the next available slot in a column using **nextEmptySlotInColumn()**, and if the result is not **nil** then use **setChip()** to change that slot's color. Here it is:

```
func addChip(chip: ChipColor, inColumn column: Int) {

  if let row = nextEmptySlotInColumn(column) {

    setChip(chip, inColumn:column, row:row)

  }

}
```

Annoyingly, all that code doesn't have any visual impact on our game: this is all model stuff, which is the behind-the-scenes representation how the game works. We have three other methods to write in ViewController.swift in order to update our view to match the model! I realise this seems like unnecessary duplication, but as you'll see later it's important to keep your GameplayKit classes as light as possible.

The first of our new methods is called **addChipAtColumn()** and it matches the board's **addChip()** method. Adding a chip in the view takes more than just three lines of code, though: it needs to calculate the size of a chip, create a **UIView** with the correct background color, position it correctly inside the board, then add it to the **placedChips** array.

To make doubly certain the movie is safe, we're only going to add a chip if the row is set correctly – i.e., if we aren't trying to add a chip below the existing row height. We're also going to animate the **transform** property of the chip view so that it starts off the top of the screen and slides in. Here's the code to put into the **ViewController** class – note that you'll get an error for the time being:

```
func addChipAtColumn(column: Int, row: Int, color: UIColor) {

  let button = columnButtons[column]

  let size = min(button.frame.size.width, button.frame.size.height /
6)

  let rect = CGRect(x: 0, y: 0, width: size, height: size)
```

```
   if (placedChips[column].count < row + 1) {

     let newChip = UIView()

     newChip.frame = rect

     newChip.userInteractionEnabled = false

     newChip.backgroundColor = color

     newChip.layer.cornerRadius = size / 2

     newChip.center = positionForChipAtColumn(column, row: row)

     newChip.transform = CGAffineTransformMakeTranslation(0, -800)

     view.addSubview(newChip)


     UIView.animateWithDuration(0.5, delay: 0, options: .CurveEaseIn,
animations: { () -> Void in

       newChip.transform = CGAffineTransformIdentity

       }, completion: nil)


     placedChips[column].append(newChip)

   }

}
```

There are three other small things in that code I want to pick out as interesting. First, user interaction is disabled on the view so that tapping on a chip is ignored and the tap is pass through to its column button. Second, the corner radius of the chip is set to **size / 2**, which will make it a circle. Finally, I've used the **.CurveEaseIn** animation speed so that the chip starts dropping slowly and picks up pace.

The error in the code is that it calls a method we haven't defined yet, called **positionForChipAtColumn()**. We'll call this method with a row and a column, and it will return the **CGPoint** where the chip should be placed. This uses six lines of code that make sense once they have been explained, but might make you draw a blank at first. So, here's how it works:

1. It pulls out the **UIButton** that represents the correct column.

2. It sets the chip size to be either the width of the column button, or the height of the column button divided by six (for six full rows) – whichever is the lowest.

3. It uses **CGRectGetMidX()** to get the horizontal center of the column button, used for the X position of the chip.

4. It uses **CGRectGetMaxY** to get the bottom of the column button, then subtracts half the chip size because we're working with the center of the chip.

5. It then multiplies the row by the size of each chip to figure out how far to offset the new chip, and subtracts that from the Y position calculated in 4.

6. Finally, it creates a **CGPoint** return value by putting together the X offset calculated in step 3 with the Y offset calculated in step 5.

It's not a graceful piece of code, I'll give you that, but once you understand what it's doing it's innocent enough.

Here's the code for the missing method:

```
func positionForChipAtColumn(column: Int, row: Int) -> CGPoint {

  let button = columnButtons[column]

  let size = min(button.frame.size.width, button.frame.size.height /
6)

  let xOffset = CGRectGetMidX(button.frame)

  var yOffset = CGRectGetMaxY(button.frame) - size / 2

  yOffset -= size * CGFloat(row)

  return CGPointMake(xOffset, yOffset)

}
```

That just leaves one final method before our code springs into life, and this is just a matter of filling in **makeMove()**. This uses the tag of the button that was tapped to figure out which column the player wants to use. We then use that column as the input for **nextEmptySlotInColumn()** to figure out which row to play, then call **addChip()** on the board model and **addChipAtColumn()** to create the chip's **UIView**. Here's the code:

```
@IBAction func makeMove(sender: UIButton) {
    let column = sender.tag

    if let row = board.nextEmptySlotInColumn(column) {
        board.addChip(.Red, inColumn: column)
        addChipAtColumn(column, row: row, color: .redColor())
    }
}
```

At this point the code works, although it's a pretty dull game: there's only one player, so you can click in any column you like again and again, adding more and more chips until the board is full. The board does, however, respect the game logic: you can't add more than six chips in a column, and the chips stack up neatly.
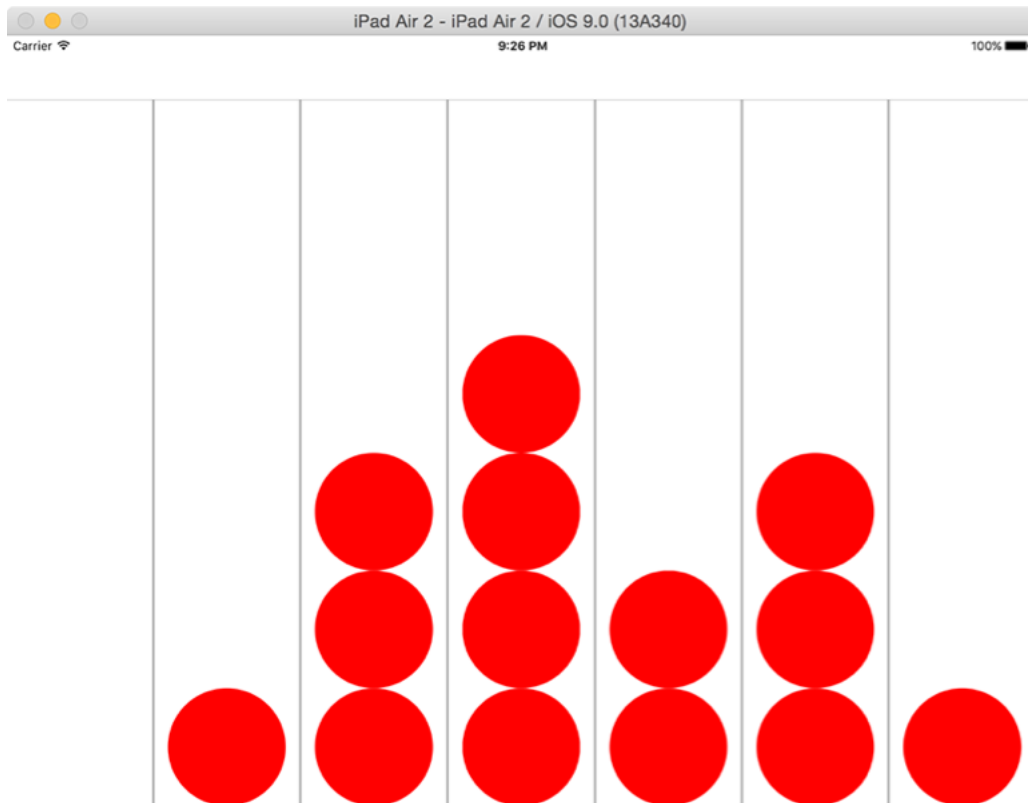
Our four in a row game so far: there's only one player, but at least the game logic works.

Adding in players: GKGameModelPlayer

It's time to take our first step into GameplayKit, although at first this step will be small. Right now our game is single-player, and all chips that get dropped are red chips. We're going to upgrade this so that there are two players by creating a new **Player** class that stores a player's chip type, their name, their color, and a special GameplayKit value called **playerId** – this is just a number that identifies every player uniquely.

On top of that, we're also going to create a static property for players, which means it's a property that belongs to the class and thus can be called from anywhere. This will be an **allPlayers** array holding both **Player** objects for easy reference

So, please create a new Cocoa Touch Class in your project. Name it Player, then make it inherit from **NSObject**. Let's go ahead and create the properties up front. As a reminder, we need:

- The chip color of the player, either **.Red**, **.Black**, or **.None**.
- The drawing color the player, set to a **UIColor**.
- The name of the player, which will be either "Red" or "Black".
- A GameplayKit **playerId** property, which we'll just set to the raw value of their chip type. (We set this enum up as an integer, remember?)
- A static array of two players, red and black.

Add these properties now:

```
var chip: ChipColor

var color: UIColor

var name: String

var playerId: Int


static var allPlayers = [Player(chip: .Red), Player(chip: .Black)]
```

Now for what is going to be a huge anti-climax: let's bring in GameplayKit by adding this import:

```
import GameplayKit
```

Now make your **Player** class conform to the **GKGameModelPlayer** protocol, like this:

```
class Player: NSObject, GKGameModelPlayer {
```

Xcode will be flagging up errors all over your code, but none of them are a result of GameplayKit – in fact, those two changes are all it takes to make GameplayKit work with our player data. Instead, the errors are because we've declared four properties non-optional and haven't given them any values, so we need to create a custom initializer.

This initializer will accept one parameter, which is the chip color to use for each player. From that we can set the player ID (as the raw value of the chip type enum), the color (either red or black **UIColor**), and the player name (either "Red" or "Black"). It's pretty straightforward really – here's the code:

```
init(chip: ChipColor) {
  self.chip = chip
  self.playerId = chip.rawValue

  if chip == .Red {
    color = .redColor()
    name = "Red"
  } else {
    color = .blackColor()
    name = "Black"
  }

  super.init()
}
```

We're going to add one more thing to this class before we're done, which is a small computed property that returns the opponent for a specific player. If the player is red, it returns the black player from the **allPlayers** array, and if the player is black, it returns the red player. Here's the code:

```
var opponent: Player {

  if chip == .Red {

    return Player.allPlayers[1]

  } else {

    return Player.allPlayers[0]

  }

}
```

That's the **Player** class finished: we won't be adding any more to it in this project. But after that code, nothing has really changed because we're not actually using those players.

To take the next step in our game, we're going to start using the new **Player** class so that we have two players in the game, and we're also going to update the user interface to mark whose turn it is.

Updating the user interface requires two methods in the **Board** class: one to determine if the board is full of pieces, and one to determine if a particular player has won. With these two we can show either "Red/Black Wins!" or "Draw!" in the user interface, but for now we're just going to return false from these methods – we'll put the real code in later.

Put these two into your **Board** class:

```
func isFull() -> Bool {

  return false

}
```

```
func isWinForPlayer(player: Player) -> Bool {

  return false

}
```

Now onto the real work: we're going to create two new methods called **continueGame()** and **updateUI()**, both in the **ViewController** class. The first will get called after every move, and will end the game with an alert if needed, otherwise it will switch players. The second is responsible for updating the title of the view controller to show whose turn it is, although later on we'll also be making it kick off AI work.

The code in **updateUI()** is trivial, so let's get it out of the way. Open up ViewController.swift and add this method:

```
func updateUI() {

  title = "\(board.currentPlayer.name)'s Turn"

}
```

We aren't tracking the current player just yet so you'll get an error at first, but we'll fix it in a moment.

The **continueGame()** method is longer, but isn't really very complicated. To help you along I'll break it down into numbered steps in the code:

1. We create a **gameOverTitle** optional string set to nil.
2. If the game is over or the board is full, **gameOverTitle** is updated to include the relevant status message.
3. If **gameOverTitle** is not nil (i.e., the game is won or drawn), show an alert controller that resets the board when dismissed.
4. Otherwise, change the current player of the game, then call **updateUI()** to set the navigation bar title.

Here's the code, with the number comments matching the list above:

```swift
func continueGame() {
  // 1
  var gameOverTitle: String? = nil


  // 2
  if board.isWinForPlayer(board.currentPlayer) {
    gameOverTitle = "\(board.currentPlayer.name) Wins!"
  } else if board.isFull() {
    gameOverTitle = "Draw!"
  }


  // 3
  if gameOverTitle != nil {
    let alert = UIAlertController(title: gameOverTitle, message: nil,
preferredStyle: .Alert)
    let alertAction = UIAlertAction(title: "Play Again",
style: .Default) { [unowned self] (action) in
      self.resetBoard()
    }

    alert.addAction(alertAction)
    presentViewController(alert, animated: true, completion: nil)

    return
  }


  // 4
  board.currentPlayer = board.currentPlayer.opponent
  updateUI()
}
```

The final steps are actually pretty straightforward. First we need to create the **currentPlayer** property in the **Board** class so that it silences the Xcode errors:

```
var currentPlayer: Player
```

That's a non-optional value, so by fixing the previous errors we introduce a new one: **currentPlayer** must be given a value inside the board's initializer. Add this at the start of the **init()** method in Board.swift:

```
currentPlayer = Player.allPlayers[0]
```

We're almost done with this chapter, and in fact we need only make a couple more changes for our game to work with two players. First, we need to update the **makeMove()** method so that it drops in chips of the correct color rather than always using red. This is easy now that we have the **board.currentPlayer** property:

```
@IBAction func makeMove(sender: UIButton) {

  let column = sender.tag


  if let row = board.nextEmptySlotInColumn(column) {

    board.addChip(board.currentPlayer.chip, inColumn: column)

    addChipAtColumn(column, row: row, color:
board.currentPlayer.color)

    continueGame()

  }

}
```

Note that I snuck in a call to **continueGame()** so that control automatically flips between players after each move.

Finally, we need to set the initial view controller title when the game is started or reset, which means modifying the **resetBoard()** method so that it calls **updateUI()**. Modify the start of the method to this:

```
func resetBoard() {
  board = Board()
  updateUI()
```

If you run your app now, you'll see things are starting to come together: you can tap on a column to play a chip, then control hands over to the other player and the user interface updates. The game doesn't end, though, because it has no idea whether one player has made four in a row, or whether it's a draw. Let's tackle that now…
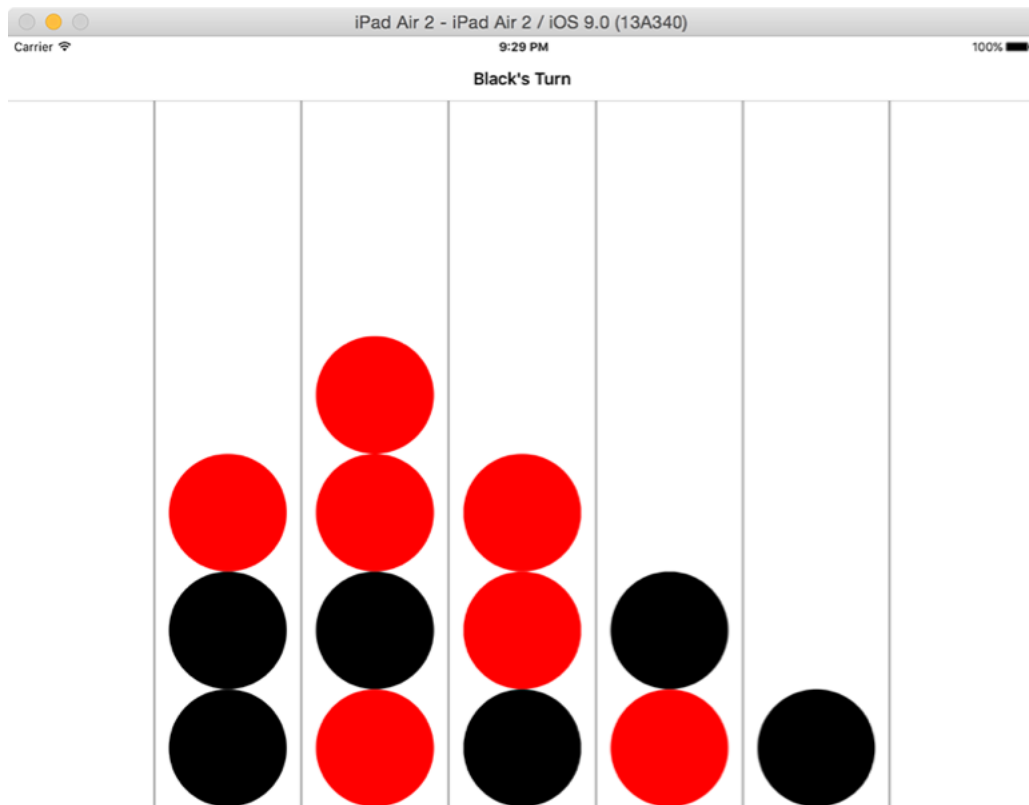
Our four in a row game has two players now, taking alternating turns.

Detecting wins and draws in Four in a Row

Now it's time to make our game an actual game – i.e., something a player can win. Four in a Row is what's called a zero-sum game, which means for one player to win the other must lose. This in turn means it's very easy to determine a winner: as soon as either player manages to place four chips in a row in any direcion, they win. As for detecting a draw, that's just a matter of checking to see if no more moves are available.

Of the two, detecting a draw is far easier, so let's write that first. We already put a stub for **isFull()** into the **Board** class, but we can fill that out now: it will return false if any column passes the **canMoveInColumn()** test, otherwise it will return false. Here's the updated method for Board.swift:

```
func isFull() -> Bool {

  for column in 0 ..< Board.width {

    if canMoveInColumn(column) {

      return false

    }

  }


  return true;

}
```

Now for the more challenging method: how to detect when a player has won? In Apple's original code for this, they took a brute force approach with four different methods for detecting wins: left to right, up to down, and two types of diagonal. It's probably very efficient code, but it's unpleasant to read and understand, so I've ditched their code and replaced it with something substantially shorter and easier to understand.

My solution involves two methods: **isWinForPlayer()** and **squaresMatchChip()**, and we'll start with the second one first.

For a player to win, they must have four chips of the same color lined up in a row anywhere

on the board. The **squaresMatchChip()** has the job of being given a square on the board and checking for one possible win type. It will accept five parameters:

 • The chip color to check.
 • The row and column of the initial chip.
 • The X and Y movement of our check.

That last one is the tricky part, so let me explain further. To detect a horizontal win, we'll call this method with an X movement of 1 and a Y movement of 0. The method can then check a slot, move along by X:1 and Y:0 (i.e., one place to the right), check a second slot, move along by X:1 and Y:0, check a third slot, then move along by X:1 and Y:0 and check the final slot. If all four have matched the player's chip color, they win.

The advantage to using this technique is that it can check for all other win types. For example, checking a vertical win means passing an X movement of 0 and a Y movement of 1, and checking a diagonal win means passing an X movement of 1 and a Y movement of 1. Remember, though, that diagonal wins go both up and down, so we need to have a second diagonal check with a Y movement of -1.

To make the **squaresMatchChip()** method safe to call from any slot on the board, we'll make it return false if it will try to check outside the bounds of the board. For example, if it starts in the bottom left and tries to search for a downward win, we'll bail out immediately. The method will also return false as soon as it has failed to detect a win for a particular movement, because there's no point checking slots 3 and 4 if slot 2 doesn't match the player's chip color.

That's everything you need to know, so here's the code for **squaresMatchChip()** – put this into the **Board** class:

```
func squaresMatchChip(chip: ChipColor, row: Int, col: Int, moveX:
Int, moveY: Int) -> Bool {
  // bail out early if we can't win from here

  if row + (moveY * 3) < 0 { return false }

  if row + (moveY * 3) >= Board.height { return false }

  if col + (moveX * 3) < 0 { return false }

  if col + (moveX * 3) >= Board.width { return false }
```

```
  // still here? Check every square

  if chipInColumn(col, row: row) != chip { return false }

  if chipInColumn(col + moveX, row: row + moveY) != chip { return
false }

  if chipInColumn(col + (moveX * 2), row: row + (moveY * 2)) != chip
{ return false }

  if chipInColumn(col + (moveX * 3), row: row + (moveY * 3)) != chip
{ return false }


  return true

}
```

That just leaves one final task before our game starts being useful: we need to fill in the **isWinForPlayer()** method so that it loops over every slot in the board, calling **squaresMatchChip()** four times for each slot: once for horizontal wins, once for vertical wins, and once for both kinds of diagonal wins. As soon as any of those calls returns true for any slot, the whole method returns true. If the loop finishes with no win, the method will return false so that play continues.

Here's the updated **isWinForPlayer()** method:

```
func isWinForPlayer(player: GKGameModelPlayer) -> Bool {
  let chip = (player as! Player).chip


  for row in 0 ..< Board.height {
    for col in 0 ..< Board.width {

      if squaresMatchChip(chip, row: row, col: col, moveX: 1, moveY:
0) {

        return true

      } else if squaresMatchChip(chip, row: row, col: col, moveX: 0,
moveY: 1) {

        return true
```

```
        } else if squaresMatchChip(chip, row: row, col: col, moveX: 1,
moveY: 1) {

            return true

        } else if squaresMatchChip(chip, row: row, col: col, moveX: 1,
moveY: -1) {

            return true

        }

    }

  }


  return false

}
```

At this point, you have a complete two-player Four in a Row game on your hands. If it weren't for GameplayKit, we'd be done here. But you want to add an AI opponent, don't you? Sure you do. So go ahead and run your app briefly, marvel at your coding prowess, then get back to Xcode: this is where the difficulty ramps up!

How GameplayKit AI works: GKGameModel, GKGameModelPlayer and GKGameModelUpdate

Amongst the many features introduced in GameplayKit, one of the most immediately useful is its ability to provide artificial intelligence that can evaluate a situation and make smart choices. We're going to be using it in our Four in a Row game to provide a meaningful opponent, but first it's essential that you understand how GameplayKit tackles the AI problem because it directly affects the code we'll write.

GameplayKit has three protocols we need to implement in various parts of our model:

 • **GKGameModel** is used to represent the state of play, which means it needs to know where all the game pieces are, who the players are, what happens after each move is made, and what the score for a player is given any state.
 • **GKGameModelPlayer** is used to represent one player in the game. This protocol is so simple we already implemented it: all you need to do is make sure your player class has a **playerId** integer. It's used to identify a player uniquely inside the AI.
 • **GKGameModelUpdate** is used to represent one possible move in the game. For us, that means storing a column number to represent a piece being played there. This protocol requires that you also store a **value** integer, which is used to rank all possible results by quality to help GameplayKit make a good choice.

We have a sensible match for the first two in our **Board** and **Player** classes, but we have nothing suitable for **GKGameModelUpdate** so let's create that now. Like I said, this needs to track only how "good" a move is, where each move is represented by a column number to play.

This is easy to do, so please go ahead and create a new Cocoa Touch class in your project. Name it Move, and make it subclass from NSObject. Now replace its source code with this:

```
import GameplayKit
import UIKit


class Move: NSObject, GKGameModelUpdate {
```

```
    var value: Int = 0

    var column: Int


    init(column: Int) {

      self.column = column

    }

}
```

That's it: the default for **value** is 0, and we create a **Move** object by passing in the column it represents. We're done with that class, and I already said we were finished with the **Player** class, which means we can focus our mental energies on what remains: **Board**.

GameplayKit's artificial intelligence works through brute force: it tries every possible move, then tries every possible follow-on move, then every possible follow-on follow-on move, etc. This runs up combinations extremely quickly, particularly when you consider that there are 4,531,985,219,092 unique positions for all the pieces on the board! So, you will inevitably limit the depth of the search to provide just enough intelligence to be interesting.

Now, this bit is really important, so read carefully. When you ask GameplayKit to find a move, it will examine all possible moves. To begin with, that is every column, because they all have space for moves in them. It then takes a copy of the game, and makes a virtual move on that copy. It then takes a copy of the game, and makes a different virtual move, and so on until until all initial first moves have been made.

Next, it starts to re-use its copies to save on memory: it will take one of those copies and apply a game state to it, which means it will reset the board so that it matches the position after one of its virtual moves. It will then rinse and repeat: it will examine all possible moves, and make one. It does this for all moves, and does so recursively until it has created a tree of all possible moves and outcomes, or at least as many as you ask it to scan.

Each time the AI has made a move, it will ask us what the player score is. For some games this will be as simple as returning a score variable, but for our 4IR game it's a bit trickier because there is no score, only a win or a loss. The original Apple source code provides a simple heuristic for this, and I've kept it here because it's quite fun – the AI can sometimes make dumb mistakes, or sometimes play like a genius, which makes the game interesting!

If you were wondering, a *heuristic* is the computer science term for a guesstimate – it's a function that tries to solve a problem quickly by taking shortcuts. For us, that means we'll tell the AI the player's score is 1000 if a move wins the game, -1000 if a move loses the game, or 0 otherwise.

All this information is important because I hope now you can see why we separate the game model from the game view – why we have a **slots** array inside the game board and a **placedChips** array inside the view controller. If you're still not sure, try to imagine how many moves the AI needs to simulate in order to decide what to do – our board has seven columns, so:

 • The player goes first, and all seven columns are valid.
 • The AI calculates its first move, which could be any of those seven columns. (7 moves in total.)
 • The AI then calculates what the player might do, but the player's move depends on the previous AI move so it has to calculate one player move for every possible AI move. (49 more moves; 56 in total.)
 • The AI then calculates what its second move might look like, which of course depends on the players first and second moves, and the AI's first move. So, for every one of those 49 moves, it has to calculate 7 more. (343 moves; 399 in total.)

…and so on. Eventually one column will become full so the multiplications will decrease, but you're still talking many thousands of copies of the board. Now imagine if the **Board** class kept track of all the **UIViews** used to draw the chips – suddenly we'd be copying far more than intended, and doing it 5000 times!

So: if a couple of chapters ago you were thinking I was wasting your time by forcing you to separate your model from your view, I hope you can now see why. AI is slow enough without doing a huge stack of extra work for no reason!

That's enough theory, it's time for some code. If you remember nothing else, remember this: to simulate a move, GameplayKit takes copies of our board state, finds all possible moves that can happen, and applies them all on different copies.

Implementing GKGameModel: gameModelUpdatesForPlayer() and applyGameModelUpdate()

Now you understand how GameplayKit approaches AI, it's time for some action. Open up Board.swift, then add this import:

```
import GameplayKit
```

You can now make your **Board** class conform to the **GKGameModel** protocol like this:

```
class Board: NSObject, GKGameModel {
```

As soon as you do that, your beautiful project will stop compiling and you'll see two errors: the **Board** class does not conform to **NSCopying** or **GKGameModel**. We covered **NSCoding** in **the original Hacking with Swift** series, but not **NSCopying**, so let's start there.

As you'll no doubt remember(!), **NSCoding** is used to encoding and decode objects so that they can be archived, such as when you're writing to **NSUserDefaults**. This is great for when you want to save or distribute your data, but it's not very efficient if you just want to copy it, and that's where **NSCopying** comes in: it's a protocol that lets iOS take a copy of your object in memory, with the copy being identical but separate to the original. As you saw in the last chapter, GameplayKit will be taking a lot of copies of our game board, so we definitely need to conform to **NSCopying**.

Implementing **NSCopying** is as simple as adding one new method, called **copyWithZone()**. The "zone" part is an optimization hangover from many years ago, and has been ignored for years. In our particular case, we're going to take a little shortcut by merging two things together: taking a copy of the game board and applying a game state.

If you remember, GameplayKit takes multiple copies of our board so that it can evaluate various moves. It then re-uses those copies by setting their game state, which is where

GameplayKit resets the board so that it matches the position after one of its moves. To remove some code duplication, we're going to make **copyWithZone()** call the method used to apply a board state. That is, **copyWithZone()** will make an empty **Board** object then call a new **setGameModel()** method to actually copy across the slot data and set the active player.

This is helpful because **setGameModel()** is part of the **GKGameModel** protocol, so we needed to implement it anyway. This method needs to accept a **GKGameModel** object as its only parameter, but of course we know that's a **Board** object so we'll do an optional downcast before copying across the properties.

Here's the code – add this to the **Board** class:

```
func copyWithZone(zone: NSZone) -> AnyObject {

  let copy = Board()

  copy.setGameModel(self)

  return copy

}


func setGameModel(gameModel: GKGameModel) {

  if let board = gameModel as? Board {

    slots = board.slots

    currentPlayer = board.currentPlayer

  }

}
```

Next, GameplayKit will ask us to tell it all the possible moves that can be made, if any. This will be called on a copy of our game board that may already have had virtual moves applied to it, but that's OK because the copy has its own **slots** array that we can read from to find where moves are possible.

Because we're conforming to the **GKGameModel** protocol, this method needs to have a precise name, accept a precise parameter, and return a precise data type. Specifically, it needs to be called **gameModelUpdatesForPlayer()**, it needs to accept a

GKGameModelPlayer object, and return a **GKGameModelUpdate** object. In our game, the last two map to the **Player** and **Move** classes, both of which conform to those protocols.

We've already written several methods that make this code surprisingly easy: if **isWinForPlayer()** is true either for the player or their opponent we return nil, and we call **canMoveInColumn()** for every column to see if the AI can move in each column. If so, we create a new **Move** object to represent that column, and add it to an array of possible moves.

To make sure you understand all the code, here it is broken down:

1. We optionally downcast our **GKGameModelPlayer** parameter into a **Player** object.
2. If the player or their opponent has won, return **nil** to signal no moves are available.
3. Otherwise, create a new array that will hold **Move** objects.
4. Loop through every column in the board, asking whether the player can move in that column.
5. If so, create a new **Move** object for that column, and add it to the array.
6. Finally, return the array to tell the AI all the possible moves it can make.

Here's the code, with the number comments matching the list above:

```
func gameModelUpdatesForPlayer(player: GKGameModelPlayer) ->
[GKGameModelUpdate]? {
  // 1
  if let playerObject = player as? Player {
    // 2
    if isWinForPlayer(playerObject) ||
isWinForPlayer(playerObject.opponent) {
      return nil
    }


    // 3
    var moves = [Move]()


    // 4
```

```
    for column in 0 ..< Board.width {

      if canMoveInColumn(column) {

        // 5

        moves.append(Move(column: column))

      }

    }


    // 6

    return moves;

  }


  return nil

}
```

The next step for the AI is to try all of those moves. GameplayKit will execute a method called **applyGameModelUpdate()** once for every move, and again this will get called on a copy of our game board that reflects the current state of play after its virtual moves. This method needs to accept a **GKGameModelUpdate** object as a parameter (that's a **Move** for us), then apply that move to its copy of the board.

Again, we've already written the methods required to make this happen. Our **Move** class contains a column number that represent's an AI move, so we just need to downcast the **GKGameModelUpdate** to a **Move**, call **addChip()** for that move, then change players. Here's the code:

```
func applyGameModelUpdate(gameModelUpdate: GKGameModelUpdate) {

  if let move = gameModelUpdate as? Move {

    addChip(currentPlayer.chip, inColumn: move.column)

    currentPlayer = currentPlayer.opponent

  }

}
```

Once GameplayKit has made a move, it will want to know whether the move is good or not. Obviously this varies from game to game, so Apple's implementation is simple: it will ask us to provide a player score after each virtual move has been made, and that score affects the way GameplayKit ranks each move.

The method name this time is **scoreForPlayer()**, and we'll get passed a **GKGameModelPlayer** object that we need to evaluate. This is a **Player** object in our game, so we'll optionally downcast it. Now, as I said already our game doesn't have a meaningful score that can be passed back as this method's return value, so we'll use a very lazy heuristic: if the player has won we'll return 1000, if their opponent has won we'll return -1000, otherwise we'll return 0.

Here's the code:

```
func scoreForPlayer(player: GKGameModelPlayer) -> Int {
  if let playerObject = player as? Player {
    if isWinForPlayer(playerObject) {
      return 1000
    } else if isWinForPlayer(playerObject.opponent) {
      return -1000
    }
  }


  return 0
}
```

There are only two further changes required to make our **Board** class conform fully to the **GKGameModel** protocol, both of which are easy and just do typecasting. You see, GameplayKit wants to see these two properties:

```
var players: [GKGameModelPlayer]?
```

```
var activePlayer: GKGameModelPlayer?
```

We don't have these right now, because we use our custom subclasses of **NSObject**. Rather than duplicate data, we're going to use computed properties to just return what we have – Swift will then correctly treat them as **GKGameModelPlayer** types. So, rather than adding those two lines of code above, use this code instead:

```
var players: [GKGameModelPlayer]? {

    return Player.allPlayers

}


var activePlayer: GKGameModelPlayer? {

    return currentPlayer

}
```

That's it: the **Board** class now conforms fully to the **GKGameModel** protocol, the **Player** class conforms fully to the **GKGameModelPlayer** protocol, and the **Move** class conforms fully to the **GKGameModelUpdate** protocol – we're finished with all these classes and those protocols, which means we can get onto the next task: configuring the AI player.

Creating a GameplayKit AI using GKMinmaxStrategist

If you've made it this far then you have built a Four in a Row game where two players can place chips in the game slots and either win or draw, and you've also prepared your model data to be run through the new GameplayKit AI routines. But we haven't created the AI just yet: we've just added some methods to our game models to enable an AI to make choices.

In this final step, we're going to use a new class called **GKMinmaxStrategist**, which is a gameplay strategy that tries to MINimize losses while MAXimizing gains – hence the name minmax, or minimax. When you create a **GKMinmaxStrategist** you tell it how many moves it should look ahead, and also what it should do to break ties, i.e. if it has two or more moves that are equally good.

Once you've created the strategist object, you need to provide it a game model to examine (that's our **Board** class), then ask it either to make the best move or make a random good move. If you ask for the best move, you'll get given back a **GKGameModelUpdate** object (that's a **Move** in our game) that represents the best move. If you ask for a random good move you'll need to tell it how many it should consider good (i.e., pick one from the top 5), and you'll get back a random **GKGameModelUpdate** from that list of good moves.

Now, one thing to be aware of up front: running AI takes a long time, particularly if you have a high look ahead depth. As a result, you should run the AI on a background thread so that your user interface doesn't lock up, and only push work back to the main thread when you have a move ready to make.

Let's go ahead and implement **GKMinmaxStrategist** now. Open ViewController.swift in your editor, then import GameplayKit. Now add this property to the **ViewController** class:

```
var strategist: GKMinmaxStrategist!
```

One strategist is capable of handling more than one game (i.e., if the player restarts the game) just by changing its game model, so we only need to create one **GKMinmaxStrategist** object. As it's needed straight away, we might as well put this into **viewDidLoad()** – anywhere before the call to **resetBoard()** is fine:

```
strategist = GKMinmaxStrategist()
strategist.maxLookAheadDepth = 7
strategist.randomSource = nil
```

Having a **maxLookAheadDepth** of 7 is a significant amount of work, because of those look aheads is one move being made by the player or AI – and each of those moves can be in any of seven columns. If you intend to alter this number upwards, be prepared for exponentially slower processing.

The **randomSource** property of **GKMinmaxStrategist** is there as a tie-breaker: if two moves result in the same advantage for the AI, which should it take? Setting it to **nil** as above means "just return the first best move," but if you wanted to have the AI take a random best move then you could try something like this:

```
strategist.randomSource = GKARC4RandomSource()
```

Now that the strategist is created, it wants to be fed some data. This is done by setting its **gameModel** property to an object that conforms to the **GKGameModel** protocol – which by now you should immediately recognize as our **Board** class. So, whenever we reset the board, we need to feed the new board into the strategist so it stands ready to look for moves.

We've done all the hard work to prepare for this, so all you need to do is change the start of your **resetBoard()** method to the following:

```
func resetBoard() {
  board = Board()
  strategist.gameModel = board


  updateUI()
```

At this point, the AI understands the state of play, and stands ready to look for good moves. With **GKMinmaxStrategist** this is done using the **bestMoveForPlayer()** method, which accepts a **GKModelPlayer** as its parameter and returns a **GKModelUpdate** for the best move if it finds one.

Remember, however, that AI can take a long time to consider all options depending on the look ahead depth you specify, so we're going to wrap this call up in a new method: **columnForAIMove()**. This will return an optional integer: either the best column for a move, or nil to mean "no move found." We'll call this on a background thread so it can take as long as it needs.

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

```
func columnForAIMove() -> Int? {

  if let aiMove = strategist.bestMoveForPlayer(board.currentPlayer)
as? Move {

    return aiMove.column

  }


  return nil

}
```

Once the AI has found a good move, we want to run that that move on the main thread, because it will involve user interface changes. I've wrapped this up in another new method called **makeAIMoveInColumn()**: this takes the column to move on, then makes it happen. This method will find the next available slot for the selected coumn, then use **addChip()** to make the move on the model, and **addChipAtColumn()** to make the move in the view.

Once the AI move has been made, we'll call **continueGame()** to check for a win or draw, then flip turns so the player is in control.

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

```
func makeAIMoveInColumn(column: Int) {
```

```
    if let row = board.nextEmptySlotInColumn(column) {

        board.addChip(board.currentPlayer.chip, inColumn: column)

        addChipAtColumn(column, row:row, color:
board.currentPlayer.color)


        continueGame()

    }

}
```

At this point our game is almost finished, but we still need to call those methods on the appropriate threads. All this will be done in one big method called **startAIMove()**, which is going to do a number of things:

1. Dispatch some work to the background thread.
2. Get the current time, then run **columnForAIMove()**.
3. Get the time again, compare the difference, and subtract that value from 1 second to form a delay value.
4. Run **makeAIMoveInColumn()** after that delay, to execute the move.

The delay is there so that the AI always waits at least one second before making its move, otherwise it might confuse the user. If the AI takes half a second to find a move, we subtract that from our one second minimum to wait for a further half a second, equalling one second in total from before starting the AI to executing the move.

Here's the first draft of **startAIMove()**:

```
func startAIMove() {


dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFA
ULT, 0)) { [unowned self] in

    let strategistTime = CFAbsoluteTimeGetCurrent()

    let column = self.columnForAIMove()!

    let delta = CFAbsoluteTimeGetCurrent() - strategistTime
```

```
    let aiTimeCeiling = 1.0

    let delay = min(aiTimeCeiling - delta, aiTimeCeiling)


    self.runAfterDelay(delay) {

      self.makeAIMoveInColumn(column)

    }

  }

}
```

Note that I use a helper method called **runAfterDelay()** because my puny brain refuses to remember the syntax for **dispatch_time()**. If you're like me, here's the helper method:

```
func runAfterDelay(delay: NSTimeInterval, block: dispatch_block_t) {

  let time = dispatch_time(DISPATCH_TIME_NOW, Int64(delay *
Double(NSEC_PER_SEC)))

  dispatch_after(time, dispatch_get_main_queue(), block)

}
```

Now only one thing more is required to finish the game: we need to call **startAIMove()** when it's black's turn. Change your **updateUI()** method to this:

```
func updateUI() {

  title = "\(board.currentPlayer.name)'s Turn"


  if board.currentPlayer.chip == .Black {

    startAIMove()

  }

}
```

Now, the game works, and you could even ship it today if you really wanted, but before you hit Run I'd like to suggest two tiny changes that make the whole experience better.

First, what happens if a user starts tapping buttons while the AI is "thinking"? Well, the answer is "bad things" – our game lets them play as black, and gets confused very quickly. So, our first change will be to disable all the column buttons when the AI's move starts, then re-enable them when it's finished.

Second, if the AI takes a long time, how does the user know the app hasn't locked up? There's no indication the AI is thinking, but it's easy enough to add by showing a custom **UIBarButtonItem** containing a **UIActivityIndicatorView**. It's not much, but it's enough to show the app is alive and well.

We're going to make both of these changes at once. All the AI code lives in **startAIMove()** so we can disable the column buttons and show the thinking spinner in there too. Add these lines to the start of **startAIMove()**:

```
columnButtons.forEach { $0.enabled = false }


let spinner = UIActivityIndicatorView(activityIndicatorStyle: .Gray)

spinner.startAnimating()


navigationItem.leftBarButtonItem = UIBarButtonItem(customView:
spinner)
```

If you haven't seen **forEach** before, it's a way of quickly looping through an array, executing some code on every item in that array. In our case, the **$0** means "each button in the loop", and in this way all the buttons get disabled.

Once the AI has finished their move, **makeAIMoveInColumn()** will be called on the main thread, and that's our chance to undo these changes: we need to re-enable the column buttons, then destroy the thinking spinner. Add these two lines of code at the start of **makeAIMoveInColumn()**:

```
columnButtons.forEach { $0.enabled = true }
navigationItem.leftBarButtonItem = nil
```

These tiny changes stop users from accidentally screwing things up, and also stop them from worrying your app has got stuck in a loop somewhere. It's polish, yes, but polish is frequently what separates good games from great games.

That's it: the app is done! You can run it now and see how quickly you can beat the AI. It ought not to be too hard – our heuristic isn't very good, so sometimes the AI will miss obvious moves, just like a real player.

Wrap up

I don't know about you, but I certainly enjoyed this tutorial – not only does it involve cool new iOS 9 stuff, but it's always fun to watch an artificial intelligence "think" its way through a problem and come to a solution. Plus, I got the chance to sneak in more **UIStackView** action, which is always a good thing!

If you're looking to extend this project, the first thing you're likely to target is the heuristic function. As I said at the beginning, this project is based on some less-than-perfect Apple sample code, which I went on to rewrite in Swift, then refactor to make it easier to understand. The heuristic code is what makes the AI smart, but it doesn't take into account how many moves it takes for a win to happen, and so it performs fairly poorly.

As for other improvements, you've seen how this game could work in one- or two-player modes, so you could easily add a user interface to let the player select what kind of game they wanted. Then, by adjusting the level of look ahead, you could implement Easy, Medium and Hard computer opponents.

For a much easier improvement to make, you could switch out our **UIView** chips for **UIImageViews**, then draw your own red and black chip graphics. There isn't much coding required to make this happen, but let's face it: you've just written a mountain of code, so you probably deserve a break!

Anyway, that's it for this project. Once again you've made a useful, real-world project that is now your own to extend in whichever direction you want. As a heavy user of iOS apps, I'm particularly looking forward to seeing how apps (not games!) will use AI – can it recommend songs with some real intelligence, for example? Have fun!

Project 35

Random Numbers

The new iOS 9 GameplayKit lets you generate random numbers in ways you soon won't be able to live without.

Setting up

If you already read project 34 you'll know how much I love GameplayKit in iOS 9. In that project we used a new class called **GKMinmaxStrategist** to produce an AI that can win at Four in a Row games by looking ahead many moves in advance, but the truth is that we only scratched the surface of what GameplayKit can do.

In this project, the first Hacking with Swift Level 2 technique project, we're going to look at another aspect of GameplayKit that is hugely exciting: randomisation. This will, I'm certain, strike you as a strange topic to choose: surely randomisation is a solved problem – what makes it interesting enough to warrant discussion, never mind to dedicate a whole technique project?

It's true that generating random data – or at least the pseudo-random that most of us consider good enough – is old news, but the GameplayKit implementation goes a step further: Apple thought specifically about random needs for games, and has built a randomisation system that I promise you're going to love, and going to use even when you're not making games.

Don't believe me? Fire up Xcode, create a new playground, and let's begin!

Generating random numbers in iOS 8 and earlier

There were lots of ways you can generate random numbers before iOS 9 came along, but none were both easy and good. For example, you could use the **rand()** function like this:

```
print(rand())
print(rand())
print(rand())
print(rand())
```

That calls **rand()** four times. Place those four lines of code into your playground, and you'll see four values in the results pane of Xcode. Now, using my incredible psychic powers, I shall predict what those four random numbers are. Ready? Here goes: 16807, 282475249, 1622650073 and 984943658.

Was I right? Probably. This is because **rand()** isn't very random by default. Random number generators are called pseudo-random for a reason: given a known starting state, you can usually predict what will come next. This has advantages for things like network games, because you just need to synchronise the starting state between clients and in theory they can both generate the same "random" numbers.

If you want to seed the random number generator with an (effectively) random state, you would use the **srand()** function first, like this:

```
srand(UInt32(time(nil)))
print(rand())
print(rand())
print(rand())
print(rand())
```

That uses the current time of the day to seed the random number generator, and because

you're running that code now rather than when I wrote it, it means I can no longer predict what output you'll get. Note that you call **srand()** only once, regardless of how many times you call **rand()**.

But, that solution has a problem: the **rand()** function is rubbish when it comes to generating randomness, and no serious programmer has used it for a long time. Fortunately, there's a solution called **arc4random()**, which uses a more algorithmically sound number generator that can also generate larger random numbers than **rand()**. What's more, **arc4random()** doesn't need to be seeded, so you can use it straight away:

```
print(arc4random())
print(arc4random())
print(arc4random())
print(arc4random())
```

No seed, no problem – I can't easily predict what your playground will output.

The **arc4random()** function generates numbers between 0 and 4,294,967,295, giving a range of 2 to the power of 32 - 1 – i.e., a lot. But if you wanted to generate a random number within a specific range – say up to 5 – there were two ways of doing it: the way most people use, and The Proper Way. Let's look at them both, because you'll encounter them both in real code.

First, the widely used but problematic way of generating random numbers in a range:

```
print(arc4random() % 6)
```

That uses modulus to ensure that the result from **arc4random()** falls within a specific range. We already covered modulus in project 8 so skip back there if you need a refresher. Note that we need to specify 6 because the values range from 0 to 5 inclusive.

This method is very common, but also problematic because it produces something called **modulo bias**, which is a fairly small but not insignificant problem that causes some numbers

to be generated more frequently than others. You might know it as **the Pigeonhole Principle** if you prefer slightly catchier names!

Second, The Proper Way of generating random numbers in a range:

```
print(arc4random_uniform(6))
```

Yes, the ARC4 family of functions comes with a built-in way of generating random numbers in a range. No, it's not new. No, I don't know why it's not used by everyone – the world's a funny place, huh?

Anyway, using **arc4random_uniform()** we can generate a range of numbers that don't have a modulo bias, don't require seeding, and are suitably random for all but cryptographic purposes.

But it's not perfect, because its range is 0 up to the maximum you specify – what if you want a number between 10 and 20, or 100 and 500? Then you need to write something thoroughly ugly indeed:

```
func RandomInt(min min: Int, max: Int) -> Int {
  if max < min { return min }
  return Int(arc4random_uniform(UInt32((max - min) + 1))) + min
}
```

That figures out the difference between the high and low ends of your range, uses that to calculate a random number, then re-adds the low end to get the full range. Because **arc4random_uniform()** works only with non-negative integers (**UInt32**) it has to do some typecasting to make the process seamless, and all those extra parentheses probably give you a headache.

Does it work? Yes, absolutely. Could you remember it if you closed this window now? No chance.

But I have some good news for you: all this complexity gets wiped away with iOS 9 thanks to

GameplayKit. Not only does it do everything above better, but it has ridiculously simple syntax that you're going to get hooked on.

Before I continue, it's worth making it doubly clear that GameplayKit is available only with iOS 9 or later, so I'm afraid all the techniques above are going to be around for quite a few years yet – I didn't write all that for nothing!

Generating random numbers with GameplayKit: GKRandomSource

Let's look at the most basic way of generating random numbers using GameplayKit, which is the **GKRandomSource** class and its **sharedRandom()** method.

A random source is a provider of an unfiltered stream of random numbers as you need them. As you'll see soon, GameplayKit has various options for your stream, but for now we're going to look at the simplest one: **sharedRandom()**.

Using **sharedRandom()** for a random number source returns the systems built-in random source that's used for a variety of other tasks, which means you can be pretty sure it's in a truly random state by the time it gets to you. It does, however, mean that it's useless for synchronizing network games, because everyone's device is in a different state.

To produce a truly random number you'd use the **nextInt()** method like this:

```
print(GKRandomSource.sharedRandom().nextInt())
```

That produces a number between -2,147,483,648 and 2,147,483,647 – yes, that's a negative number, which means it's not a drop-in replacement for **arc4random()**. Plus, even with GameplayKit's great new logic, Apple includes a warning that it's not guaranteed to be random for very specific situations (harkening back to **rand()**, no less), so for both these reasons it's not likely you'll want to use **nextInt()** much.

As an alternative, try using the **nextIntWithUpperBound()** method, which works identically to **arc4random()**:

```
print(GKRandomSource.sharedRandom().nextIntWithUpperBound(6))
```

That will return a random number from 0 to 5 using the system's built-in random number generator.

As well as **nextInt()** and **nextIntWithUpperBound()** are **nextBool()** for generating a random

true/false value and **nextUniform()** for generating a random floating-point number between 0 and 1. Both of these are implemented using **nextIntWithUpperBound()** so they output properly random numbers.

Choosing a random number source: GKARC4RandomSource and other GameplayKit options

Using the system's built-in random number source is exactly what you want when you just need something simple. But the system's random number generator is not deterministic, which means you can't predict what numbers it will output because it always starts in a different state – and that makes it useless for synchronizing network games.

Using the system's random number source is also useless to avoid cheating. If someone is playing a brilliant strategy game you made and loses a battle because a dice roll didn't go their way, they could quickly quit the app, relaunch, and try the battle again hoping that the random roll would go differently for them.

GameplayKit offers three custom sources of random numbers, all of which are deterministic, and all of which can be serialized – i.e., written out to disk using something like **NSCoding** that we looked at in project 12. This means network play can be synchronized and cheaters are unable to force their way around your game – a win all around!

The reason GameplayKit has three sources of random numbers is simple: generating random numbers is hard, so you get to choose whether you want something simple and fast, hard and slow, or somewhere in the middle. That is, if you know the result of your random number doesn't matter that much and you're going to need thousands quickly, you can use the faster-but-less-random option. Alternatively, if you need one random number but it's got to be as random as they come, you can use the more intensive algorithm. In short, **you pays your money and you takes your choice.**The three options are:

 • **GKLinearCongruentialRandomSource**: has high performance but the lowest randomness
 • **GKMersenneTwisterRandomSource**: has high randomness but the lowest performance
 • **GKARC4RandomSource**: has good performance and good randomness – in the words of Apple, "it's going to be your Goldilocks random source."

Honestly, the performance difference between the three of these is all but insignificant unless you're generating vast quantities of random numbers.

So, to generate a random number between 0 and 19 using an ARC4 random source that you can save to disk, you'd use this:

```
let source = GKARC4RandomSource()
source.nextIntWithUpperBound(20)
```

If you really want the maximum possible randomness for your app or game, try the Mersenne Twister source instead:

```
let source = GKMersenneTwisterRandomSource()
source.nextIntWithUpperBound(20)
```

As you can see, once you've created the random source the method calls on it are identical – all you've done is change the underlying random number generator.

Before continuing, you should know that Apple recommends you force flush its ARC4 random number generator before using it for anything important, otherwise it will generate sequences that can be guessed to begin with. Apple suggests dropping at least the first 769 values, so I suspect most coders will round up to the nearest pleasing value: 1024. To drop values, use this code:

```
source.dropValuesWithCount(1024)
```

Regardless of which source you choose, Apple goes to great lengths to point out that none of them are recommended for cryptography purposes. Apps, yes, games, yes, but not cryptography – sorry!

Shaping GameplayKit random numbers: GKRandomDistribution, GKShuffledDistribution and GKGaussianDistribution

Random sources are interesting enough, but chances are you're wondering why you've spent the last 20 minutes reading about GameplayKit and have yet to see anything interesting. Well, here's where good becomes great: GameplayKit lets you shape the random sources in various interesting ways using *random distributions*.

Let's start off with something simple: rolling a six-sided dice. This is effectively identical to generating a random number between 1 and 6, which before meant having to call our old friend:

```
func RandomInt(min min: Int, max: Int) -> Int {

  if max < min { return min }

  return Int(arc4random_uniform(UInt32((max - min) + 1))) + min

}
```

That's all gone in GameplayKit, because they have built six-sided dice right into their API. No, really. Try this:

```
let d6 = GKRandomDistribution.d6()

d6.nextInt()
```

Boom: you'll get a random number between 1 and 6. Want a 20-sided die? Great:

```
let d20 = GKRandomDistribution.d20()

d20.nextInt()
```

Oh, you want a 11,539-sided die? Done:

```
let crazy = GKRandomDistribution(lowestValue: 1, highestValue: 11539)
crazy.nextInt()
```

And right there you can immediately see how the concept of dice is just there to frame the random number generation: give it a lowest value of 100 and a highest of 200, and you'll get a number between those two, inclusive.

If you intend to specify lowest and highest values, you need to be careful if you intend also to use **nextIntWithUpperBound()** – code like this will crash your app:

```
let distribution = GKRandomDistribution(lowestValue: 10, highestValue: 20)
print(distribution.nextIntWithUpperBound(9))
```

When you create a random distribution in this way, iOS automatically creates a random source for you using an unspecified algorithm. If you want one particular random source, there are special constructors for you:

```
let rand = GKMersenneTwisterRandomSource()
let distribution = GKRandomDistribution(randomSource: rand, lowestValue: 10, highestValue: 20)
print(distribution.nextInt())
```

So, GameplayKit makes it easy to generate truly random numbers within a specific range – no more hand-coded functions required. That alone is a huge win, but what if I said GameplayKit could also generate random numbers that *weren't* truly random?

Yes, I know that sounds pointless, but stick with me. If you roll a six-sided die twice, there's a

one in six chance you'll get the same number twice in a row. Unless you're working with fixed dice, this is true randomness. But true randomness is sometimes unpleasant, because it's possible to roll a six five times in a row or even 50 times in a row. Sure, it's not *likely*, but it's possible, and players inevitably curse their "unlucky streak" and may even leave angry App Store reviews.

But there's more: what if you want a spread of numbers, but really you want them to naturally cluster towards a middle ground? This would mean a tank in your strategy game normally fights like an average tank, but occasionally will do something spectacularly brave – or spectacularly stupid.

GameplayKit has solutions for both of these situations, and it's so simple you'll want to start using it straight away.

You just saw me using **GKRandomDistribution** to shape a random source, either created automatically or specified in its constructor. GameplayKit provides two other distributions: **GKShuffledDistribution** ensures that sequences repeat less frequently, and **GKGaussianDistribution** ensures that your results naturally form a bell curve where results near to the mean average occur more frequently.

Let's look at **GKShuffledDistribution** first. This is an anti-clustering distribution, which means it shapes the distribution of random numbers so that you are less likely to get repeats. This means it will go through every possible number before you see a repeat, which makes for a truly perfect distribution of numbers.

For example, the code below generates the numbers 1 to 6 in a random order:

```
let distribution = GKShuffledDistribution.d6()

print(distribution.nextInt())

print(distribution.nextInt())

print(distribution.nextInt())

print(distribution.nextInt())

print(distribution.nextInt())

print(distribution.nextInt())
```

To be clear, that code literally will generate the number 1 once, the number 2 once, etc, up to 6, but the order is random. This makes **GKShuffleDistribution** a so-called "fair distribution" because every number will appear an equal number of times. You are, in theory at least, guaranteed that the first roll and the second roll will be different.

Obviously fair random has its own downside, which is that if a player rolls a six they can be know for sure they won't get a six in their next five rolls. This means you need to use it with caution: there's no point having random numbers if they are predictably random. It is, however, worth adding that **GKShuffledDistribution** is still random over time – the actual order of numbers can't be predicted.

The other distribution option is **GKGaussianDistribution** (tip: "Gauss" rhymes with "House"), which causes the random numbers to bias towards the mean average of the range. So if your range is from 0 to 20, you'll get more numbers like 10, 11 and 12, fewer numbers like 8, 9, 13 and 14, and decreasing amounts of any numbers outside that.

To give you an idea of the bell curve this distribution generates, I created 100,000 random numbers between 1 and 20, and only 228 were the number 1. That's way below the statistical average of 5000, particularly when you realise that the number 11 got rolled 12,488 times – more than 50 times as often.

You can see the actual output of my test below, visualised on a chart:
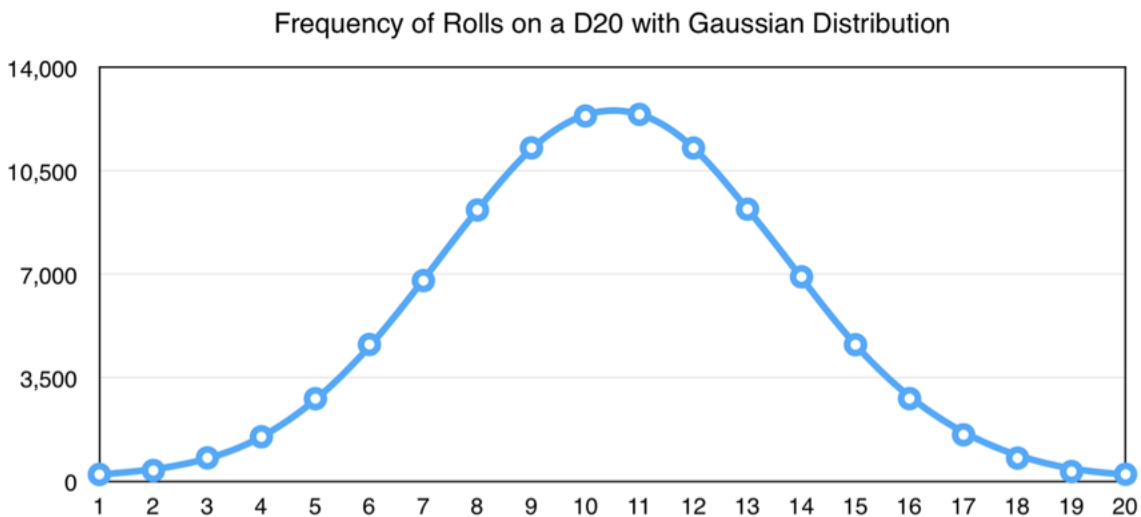
Frequency of Rolls on a D20 with GKGaussianDistribution

GKGaussianDistribution is perfect for when you want random things to happen, but you also want to steer that randomness so that has a degree of averageness to it.

Shuffling an array with GameplayKit: arrayByShufflingObjectsInArray()

Many Swift game projects use this Fisher-Yates array shuffle algorithm implemented in Swift by Nate Cook:

```
extension Array {

  mutating func shuffle() {

    for i in 0..<(count - 1) {

      let j = Int(arc4random_uniform(UInt32(count - i))) + i

      swap(&self[i], &self[j])

    }

  }

}
```

With GameplayKit there's a specific method you can call that does a similar thing: **arrayByShufflingObjectsInArray()**. I say "similar thing" rather than "identical thing" because the GameplayKit returns a new array rather than modifying the original, whereas Nate's version shuffles in place. We used this method a lot in the original Hacking with Swift projects, so hopefully you're familiar with it by now.

For example, if you wanted to blithely ignore the inevitable legalities and set up a lottery in your neighborhood, you could create an array containing the numbers 1 to 49, randomise its order, then pick the first six balls:

```
let lotteryBalls = [Int](1...49)

let shuffledBalls =
GKRandomSource.sharedRandom().arrayByShufflingObjectsInArray(lotteryB
alls)

print(shuffledBalls[0])

print(shuffledBalls[1])

print(shuffledBalls[2])
```

```
print(shuffledBalls[3])

print(shuffledBalls[4])

print(shuffledBalls[5])
```

Note that I'm using the default system randomisation because determinism is exactly what you *don't* want in a lottery. Actually, forget it: if you're going to ignore the law and set up your own lottery, you might as well fix it so you win, right?

One of the advantages of GameplayKit's randomisation is that it is truly deterministic, even across devices. This means as long as you tell it where to start, it will produce the same series of random numbers in the future. This is perfect for our evil lottery plan, and it gives me the chance to show you one last thing: seeding GameplayKit's random sources.

When we created our random seeds earlier, we just used this:

```
let source = GKMersenneTwisterRandomSource()
```

That creates a new Mersenne Twister random source with a random starting point. But if you want to force a starting point – either because you want to win your lottery or because you want players in a network game to be synchronised – you can create your random source with a specific *seed*, which is a fixed starting point.

When you use a seed value, your random number generator becomes predictable - you're back to my "psychic powers" when I knew what numbers you would get out of **rand()**. But that's OK, because you can generate the seeds using a separate random number generator, so you're guaranteed uniqueness.

Here's our lottery example rewritten using a fixed seed value of 1001:

```
let lotteryBalls = [Int](1...49)

let shuffledBalls = GKMersenneTwisterRandomSource(seed:
1001).arrayByShufflingObjectsInArray(lotteryBalls)

print(shuffledBalls[0])
```

```
print(shuffledBalls[1])

print(shuffledBalls[2])

print(shuffledBalls[3])

print(shuffledBalls[4])

print(shuffledBalls[5])
```

If you run that code now you'll see that the balls are shuffled identically every time. It's a random order, but *predictably* random if you know what I mean!

Wrap up

I realise technique projects can be a little dry, but I hope you can see some real advantages to using GameplayKit randomisation over other solutions. Not only does it offer a wider range of functionality (shuffle and Gaussian distributions are awesome!) but it makes your code much simpler, and also has the guarantee of being provably random.

Of course, if you're stuck supporting prior versions of iOS, you'll need to mix and match GameplayKit randomisation with calls to **arc4random_uniform()** and the like. Make sure you take advantage of the **#available syntax in Swift 2** so you can use the new APIs when they are available.

It bears repeating that this is only a small slice of what GameplayKit offers. If you haven't already read tutorial 34, you should check it out now - it's a tutorial for GKMinmaxStrategist from GameplayKit that shows how to create an AI for Four in a Row.

Project 36

Crashy Plane

Ever wanted to make a Flappy Bird clone? Now you can do it in under an hour thanks to SpriteKit.

Setting up

This is the first game project in Hacking with Swift level 2, and we're going to produce a Flappy Bird clone called Crashy Plane. You might be forgiven for thinking, "do we really need another tutorial on how to make Flappy Bird?" And you're right, because there are quite a few tutorials out that already. But this one is different: we'll be using Swift 2, we'll be using iOS 9.0, and you're going to get some gorgeous assets you can use in your own games.

So, it's Yet Another Flappy Bird tutorial, but I'm going to take the opportunity to teach you some Swift 2 and iOS 9 along the way. Before you start, please **download the assets for this project** so you can follow along. If you haven't played Flappy Bird before, the concept is simple: tap the screen to keep your bird flying, and don't touch the floor of any pipes. In our game it'll be a plane with mountains as obstacles, but the idea is the same.

The assets you download are all licensed under CC0 / public domain, which means you can use them however you want without attribution. If you want to attribute the original authors, see the README.txt file in the zip. The game art comes from a designer called Kenney, who offers a huge selection of public domain game assets in return for a donation – if you're serious about making games you should definitely **visit his home page**.

All set? Great! Launch Xcode and create a new project from the game template. Choose Swift for your language, SpriteKit for the game technology, and iPhone for device. Name it Project36 and click Next then Finish. Before we go any further, please lock your game's orientation to be portrait.

Creating a player: ResizeFill vs AspectFill

The first thing we're going to do in this game is clear out what's there and get our player on the screen so we can be sure everything is working. This is an iPhone game which means we need to be able to handle various device sizes: iPhone 4s, iPhone 5, iPhone 6 and iPhone 6 Plus all need to be catered for.

In your project, select the Assets.xcassets asset catalog. You'll see AppIcon and Spaceship already in there, and as usual I'd like you to delete Spaceship because it's not used in our game. In the assets you downloaded, look in the GFX folder and drag all the files from there into your asset catalog – you'll see I've provided 1x, 2x and 3x versions of each piece of art, which means you could expand this into an iPad game later on if you wanted.

Now right-click on your project group in the Project Navigator pane – that's not the blue "Project36" at the top, but the yellow "Project36" directly beneath it. Choose New Group, then name it "Content" and hit Enter. Copy into there the remaining assets you downloaded – coin.wav, explosion.wav, music.m4a, PlayerExplosion.sks and spark.png.

That's all the assets configured for this game, so let's look at the code: open up GameScene.swift then remove everything from **didMoveToView()**, **touchesBegan()** and **update()**, but leave the methods themselves intact because we'll be using them later. That means your class should look like this:

```
class GameScene: SKScene {

  override func didMoveToView(view: SKView) {


  }


  override func touchesBegan(touches: Set<UITouch>, withEvent event:
UIEvent?) {


  }


  override func update(currentTime: CFTimeInterval) {
```

```
    }
}
```

To make sure our game is working OK, we're going to start by creating the player. I've provided three different player sprites for you that make it look like the propeller is spinning around. We're going to need to reference the player throughout the game, so create a property for it now by adding this to your **GameScene** class:

```
var player: SKSpriteNode!
```

By the time it's finished, our game is going to need to create lots of different elements, so rather than clutter up **didMoveToView()** to handle everything we're going to use a common programming methodology called Composed Methods, which essentially just means "make each method do one small thing, then combine them together as needed." In our case, that means we're going to have lots of methods to create things in our game, then call them individually in **didMoveToView()**.

For creating the player, we're going to create a sprite node out of the first frame in the player animation, "player-1". We're then going to position the player most of the way up the screen and most of the way to the left – this gives them enough time to respond when the game starts. To get the propeller animation we're going to use the **animateWithTextures()** SpriteKit action, cycling through each frame every 0.01 seconds. That's actually faster than our game draws, so it essentially means "as fast as possible."

Here's the code - put this into GameScene.swift, below **update()**:

```
func createPlayer() {
    let playerTexture = SKTexture(imageNamed: "player-1")
    player = SKSpriteNode(texture: playerTexture)
    player.zPosition = 10
    player.position = CGPoint(x: frame.width / 6, y: frame.height *
0.75)
```

```
    addChild(player)


    let frame2 = SKTexture(imageNamed: "player-2")

    let frame3 = SKTexture(imageNamed: "player-3")

    let animation = SKAction.animateWithTextures([playerTexture,
frame2, frame3, frame2], timePerFrame: 0.01)

    let runForever = SKAction.repeatActionForever(animation)


    player.runAction(runForever)
}
```

Now add this to **didMoveToView()**:

```
createPlayer()
```

Press Play in Xcode to build and run your app, and… no player? That's right: you won't see the player. If you look back at the code you'll see it says "place the player 1/6th of the way across the screen and 3/4s of the way up," but still there's no player. The reason for this becomes clear if you open GameScene.sks (not GameScene.swift!) in Xcode. If you look in the Attributes inspector you'll see your game scene is configured to be 1024x768 in size, which isn't helpful given that we're creating an iPhone game rather than an iPad game.

The easy fix here is to tell SpriteKit to resize the game scene to fit the actual view it's running inside. Open up GameViewController.swift and look for this line:

```
scene.scaleMode = .AspectFill
```

That means "scale the scene so that it fits the view, allowing to be cropped if needed." This is causing the left and right edges of the game scene to get lost. To fix the problem, change the line to this:

```
scene.scaleMode = .ResizeFill
```

That now means "just make the game scene the same size as the view it is inside." Press Play now and you should see the plane flying away in the top left corner – a good start!

Sky, background and ground: parallax scrolling with SpriteKit

There are multiple depth levels in Flappy Bird, and when they come together and scroll at various speeds you get a surprisingly nice depth effect. We'll be controlling the depth of our graphics by setting the **zPosition** property of sprites, starting with the sky: this is just two colored blocks that sit right at the very back of the game.

You can create **SKSpriteNodes** with nothing more than a color and a size, and that's what we'll use here. I've sampled the sky colors based on the other graphics that we'll be adding soon – the top and bottom parts of the sky are very, *very* similar, but just different enough to be visible.

To make things easier for my brain, I'm going to be setting the **anchorPoint** property of the sprite nodes. This means they calculate their positions differently from the default, which might not sound easy at all, but trust me: it is! By default, nodes have the anchor point X0.5, Y0.5, which means they calculate their position from their horizontal and vertical center. We'll be modifying that to be X0.5, Y1 so that they measure from their center top instead – it makes it easier to position because one part of the sky will take up 67% of the screen and the other part will take up 33%.

Here's the **createSky()** method; add this just below **createPlayer()**:

```
func createSky() {

  let topSky = SKSpriteNode(color: UIColor(hue: 0.55, saturation:
0.14, brightness: 0.97, alpha: 1), size: CGSize(width: frame.width,
height: frame.height * 0.67))

  topSky.anchorPoint = CGPoint(x: 0.5, y: 1)


  let bottomSky = SKSpriteNode(color: UIColor(hue: 0.55, saturation:
0.16, brightness: 0.96, alpha: 1), size: CGSize(width: frame.width,
height: frame.height * 0.33))

  topSky.anchorPoint = CGPoint(x: 0.5, y: 1)


  topSky.position = CGPoint(x: CGRectGetMidX(frame), y:
frame.size.height)
```

```
    bottomSky.position = CGPoint(x: CGRectGetMidX(frame), y:
bottomSky.frame.height / 2)


    addChild(topSky)

    addChild(bottomSky)


    bottomSky.zPosition = -40

    topSky.zPosition = -40

}
```

Add a call to **createSky()** inside **didMoveToView()** then press Play again – you should see some sky behind the player's plane now. Can you spot that it's two different colors?

Next up is the background. In the assets for this game it's a set of distant mountains and clouds with a faint blue color, but we can't just add this to the game using a sprite node. The reason is simple: while the sky is just two fixed (and very similar!) colors, the background mountains need to scroll.

Making the mountains scroll is easy enough, but what's harder is ensuring the mountains don't just scroll off the screen and leave nothing behind. What we really want to happen is to have mountains scroll to the left forever, looping infinitely. We're going to accomplish this with a little cheat: we're going to create two sets of mountains, both moving left. When one moves off the screen completely we're going to move it way over to the other side of the screen so that it can carry on moving. With two sets of mountains in place, this means there'll be a seamless, never-ending mountain range in the background.

First up, add this method to your class. It creates two sprite nodes from the background texture, positioning them side by side in your scene:

```
func createBackground() {

  let backgroundTexture = SKTexture(imageNamed: "background")


  for i in 0 ... 1 {

    let background = SKSpriteNode(texture: backgroundTexture)
```

```
        background.zPosition = -30

        background.anchorPoint = CGPointZero

        background.position = CGPoint(x: (backgroundTexture.size().width
* CGFloat(i)) - CGFloat(1 * i), y: 100)

        addChild(background)

    }

}
```

You'll notice that I set the anchor point to the value **CGPointZero**, which makes the background texture position itself from the left edge. This is helpful because it means we know exactly when each mountain is fully off the screen, because its X position will be equal to 0 minus its width. I also set the **zPosition** properties to be -30, which places them in front of the sky.

We're using **backgroundTexture.size().width * CGFloat(i)) - CGFloat(1 * i)** to calculate the X position of each mountain, which might look hard but really it isn't. This is inside a loop that counts from 0 to 1, so the first time the loop goes around X will be 0, and the second time the loop goes around X will be the width of the texture minus 1 to avoid any tiny little gaps in the mountains.

Add a call to **createBackground()** to **didMoveToView()** and hit Play to make sure it's working - you should see a static mountain range. There are two of them there, but you can't see the other one because it's way off screen. To bring that to life we need to make the mountains move: first to the left over 20 seconds, then way back over to the right over 0 seconds, i.e. immediately.

To make that work, add this code to **createBackground()** just after **addChild()**:

```
let moveLeft = SKAction.moveByX(-backgroundTexture.size().width, y:
0, duration: 20)

let moveReset = SKAction.moveByX(backgroundTexture.size().width, y:
0, duration: 0)

let moveLoop = SKAction.sequence([moveLeft, moveReset])

let moveForever = SKAction.repeatActionForever(moveLoop)
```

```
background.runAction(moveForever)
```

So, each mountain will move to the left a distance equal to its width, then jump back another distance equal to its width. This repeats in a sequence forever, so the mountains loop indefinitely – try running the app to see how it looks!

Placing the plane on a simple moving background gives the feeling of movement without much work.

Next up we're going to create the ground. This needs to have a Z position of -10 (not -20; you'll see why later!) and have very similar movement logic to the mountains. That is, we need to create two lots of the ground texture and have it move back and forth in a loop to create an infinite scrolling landscape.

This time, however, is a little different: we can't adjust the anchor point of the sprite because it causes problems with physics, so we need to do some maths juggling. We also need to make the ground move much faster than the mountains so you get a neat parallax scrolling effect.

Here's the code for the **createGround()** method:

```
func createGround() {

    let groundTexture = SKTexture(imageNamed: "ground")


    for i in 0 ... 1 {

        let ground = SKSpriteNode(texture: groundTexture)

        ground.zPosition = -10

        ground.position = CGPoint(x: (groundTexture.size().width / 2.0 +
(groundTexture.size().width * CGFloat(i))), y:
groundTexture.size().height / 2)


        addChild(ground)


        let moveLeft = SKAction.moveByX(-groundTexture.size().width, y:
0, duration: 5)

        let moveReset = SKAction.moveByX(groundTexture.size().width, y:
0, duration: 0)

        let moveLoop = SKAction.sequence([moveLeft, moveReset])

        let moveForever = SKAction.repeatActionForever(moveLoop)


        ground.runAction(moveForever)

    }

}
```

Remember to add a call to **createGround()** inside **didMoveToView()**, then press Play to see how things look. You should see the player's plane animating, the sky, some moving mountains, plus a faster-moving ground. It's not a game yet because there aren't any controls, but I hope you can see things coming together!

Making the ground and background move at different speeds creates a parallax movement effect, mimicing depth.

Creating collisions and making random numbers with GameplayKit

Part of the infuriating nature of Flappy Bird was that there were all sorts of collisions that could instantly kill you. In our game, these are massive rocks that will come out from the top and bottom of the screen – and if a player hits any rock, or the ground, they are history.

The player's job is to fly their plane safely through the rocks that come along. The difficult part is that the gap between rocks varies in position, and can be high, low or in the middle of the screen, so the player needs quick reactions to score anything over a few points.

We're going to make a **createRocks()** method in just a moment, but first here's what it needs to do:

1. Create top and bottom rock sprites. They are both the same graphic, but we're going to rotate the top one and flip it horizontally so that the two rocks form a spiky death for the player.
2. Create a third sprite that is a large red rectangle. This will be positioned just after the rocks and will be used to track when the player has passed through the rocks safely – if they touch that red rectangle, they should score a point. (Don't worry, we'll make it invisible later!)
3. Use the new **GKRandomDistribution** class in iOS 9's GameplayKit to generate a random number in a range. This will be used to determine where the safe gap in the rocks should be.
4. Position the rocks just off the right edge of the screen, then animate them across to the left edge. When they are safely off the left edge, remove them from the game.

Here's the **createRocks()** method, with numbered comments matching the numbers above:

```
func createRocks() {
    // 1
    let rockTexture = SKTexture(imageNamed: "rock")

    let topRock = SKSpriteNode(texture: rockTexture)
    topRock.zRotation = CGFloat(M_PI)
    topRock.xScale = -1.0
```

```swift
    let bottomRock = SKSpriteNode(texture: rockTexture)


    topRock.zPosition = -20

    bottomRock.zPosition = -20



    // 2
    let rockCollision = SKSpriteNode(color: UIColor.redColor(), size:
CGSize(width: 32, height: frame.height))

    rockCollision.name = "scoreDetect"


    addChild(topRock)

    addChild(bottomRock)

    addChild(rockCollision)



    // 3
    let xPosition = frame.width + topRock.frame.width


    let max = Int(frame.height / 3)

    let rand = GKRandomDistribution(lowestValue: -100, highestValue:
max)

    let yPosition = CGFloat(rand.nextInt())


    // this next value affects the width of the gap between rocks

    // make it smaller to make your game harder – if you're feeling
evil!

    let rockDistance: CGFloat = 70



    // 4
    topRock.position = CGPoint(x: xPosition, y: yPosition +
topRock.size.height + rockDistance)
```

```
  bottomRock.position = CGPoint(x: xPosition, y: yPosition -
rockDistance)

  rockCollision.position = CGPoint(x: xPosition +
(rockCollision.size.width * 2), y: CGRectGetMidY(frame))


  let endPosition = frame.width + (topRock.frame.width * 2)


  let moveAction = SKAction.moveByX(-endPosition, y: 0, duration:
5.8)

  let moveSequence = SKAction.sequence([moveAction,
SKAction.removeFromParent()])

  topRock.runAction(moveSequence)

  bottomRock.runAction(moveSequence)

  rockCollision.runAction(moveSequence)

}
```

You'll get an error at first because you don't have GameplayKit in your app. Add this to the top of GameScene.swift:

```
import GameplayKit
```

For more information on **GKRandomDistribution** you should read tutorial 35, which covers the new GameplayKit randomization in detail. If you haven't seen it before, the **xScale** property lets you stretch sprites horizontally. Using -1.0 as the value is what causes the flip effect - it stretches the sprite by -100%, inverting it. I'm also using -20 for the **zPosition** because we want the rocks to appear behind the ground sprites to keep the illusion intact.

You'll notice we're adding the movement action to the top rock, the bottom rock and the collision sprite. If you wanted, you could create an extra **SKNode** that contains all three rock sections, then animate *that*, but it gives you the same result. You might also have noticed the curious duration I set: 5.8. I chose this through trial and error because the rocks move a different distance to the ground and yet need to move at about the same speed – a duration of 5.8 comes close enough.

Now, we're *not* going to add a call to **createRocks()** in **didMoveToView()**. You see, if we did that it would create just one pair of rocks, which isn't what we want. Instead, we want rocks to be created every few seconds continuously until the player dies, which means we need a second method: **initRocks()**. Add this now:

```
func initRocks() {
  let create = SKAction.runBlock { [unowned self] in
    self.createRocks()
  }


  let wait = SKAction.waitForDuration(3)
  let sequence = SKAction.sequence([create, wait])
  let repeatForever = SKAction.repeatActionForever(sequence)


  runAction(repeatForever)
}
```

That new method calls **createRocks()**, waits three seconds, calls **createRocks()** again, waits again, and so on, forever. Add a call to **initRocks()** to your **didMoveToView()** method, and if you run the app you'll really start to see things looking good: rocks should appear at random heights, with the red scoring box straight after them. Yes, you can't crash into them yet, but we'll get there soon!

The last thing we're going to do in this chapter is add a score. As you should have completed the earlier game projects in **Hacking with Swift** already, the only surprising thing here is that I'm not going to use Chalkduster as my font – and only then because it's not easy to read against moving rocks and mountains!

Add these two properties to your class. One is to hold the score as an integer, and the other is to draw the score to the screen using a **SKLabelNode**:

```
var scoreLabel: SKLabelNode!
```

```
var score = 0 {

  didSet {

    scoreLabel.text = "SCORE: \(score)"

  }

}
```

As per usual, we use a property observer to update the label whenever the score changes. Now add this **createScore()** method to your class:

```
func createScore() {

  scoreLabel = SKLabelNode(fontNamed: "Optima-ExtraBlack")

  scoreLabel.fontSize = 24


  scoreLabel.position = CGPointMake(CGRectGetMaxX(frame) - 20,
CGRectGetMaxY(frame) - 40)

  scoreLabel.horizontalAlignmentMode = .Right

  scoreLabel.text = "SCORE: 0"

  scoreLabel.fontColor = UIColor.blackColor()


  addChild(scoreLabel)

}
```

That positions the score in the top-right corner of the screen, safely away from the player so as not to be too annoying.

Add a call to **createScore()** to **didMoveToView()** and you're done. Now all that's left is the important stuff: the game play!
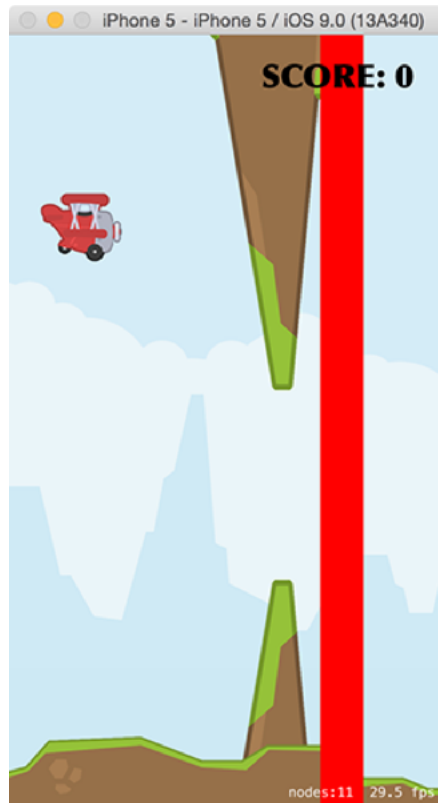
Adding a red rectangle just after each set of rocks gives us something our player can collide against, but make sure you hide it later!

Pixel-perfect physics in SpriteKit, plus explosions and more

Everything in our game is configured to look good, but it's not actually playable yet. Surprisingly, you're now only about 10 minutes away from a fully working game, because as soon as we add in a few physics calls the game is good to go.

As you might imagine, Flappy Bird is a game where physics really matters. The player's plane has physics, the rocks have physics, the ground has physics, and there's also gravity pulling the player inevitably downwards towards their doom. So, we need to make sure we are told when collisions happen, which means we need to conform to the **SKPhysicsContactDelegate** protocol. Change your **GameScene** class's definition to this:

```
class GameScene: SKScene, SKPhysicsContactDelegate {
```

Now in your **didMoveToView()** method you want to make the SpriteKit physics world report collisions to the game scene so they can be acted upon. We're also going to use this opportunity to adjust the gravity of the physics world – you can set this to any value you want, but be warned: the game is hard enough without massive amounts of gravity!

Add these two lines to **didMoveToView()**:

```
physicsWorld.gravity = CGVectorMake(0.0, -5.0)
physicsWorld.contactDelegate = self
```

So: physics. Let's start by adding physics to the player. To make things fair, we're going to use pixel-perfect collision detection to maximize the player's chance of survival, and SpriteKit makes this really easy to do. In your **createPlayer()** method, just after the call to **addChild()**, add this:

```
player.physicsBody = SKPhysicsBody(texture: playerTexture, size:
playerTexture.size())
```

```
player.physicsBody!.contactTestBitMask =
player.physicsBody!.collisionBitMask

player.physicsBody?.dynamic = true


// player.physicsBody?.collisionBitMask = 0
```

Those four lines of code pack in a lot of functionality, and might not make sense right away so let me break it down:

1. The first line sets up pixel-perfect physics using the sprite of the plane. This sprite animates, but the difference is so tiny it won't matter.
2. The second line makes SpriteKit tell us whenever the player collides with anything. This is wasteful in some games, but here the player dies if they touch anything so it's the right thing to do.
3. The third line makes the plane respond to physics. This is the default, but I'm including it here because we'll change it later.
4. The last line makes the plane bounce off nothing, or at least it would do if it weren't commented out. I've made it commented out just for a moment so you can see it's working – I'll tell you when to remove the comment.

You might think lines 2 and 4 contradict each other, but they don't and they are both needed. SpriteKit distinguishes between contact (two things touched) and collision (two things should bounce off each other in the physics world). We want our plane to notify us if it touches anything – any rock, the score counter red rectangles, or the ground. But we *don't* want it to bounce off them, because we don't want the player to lose any momentum when they touch the hidden score counters.

Don't bother running the game just yet, because all you'll see is the player falling off the screen! To make things interesting we need to make some more changes first.

In the **createGround()** method, just before the call to **addChild()**, add this:

```
ground.physicsBody = SKPhysicsBody(texture: ground.texture!, size:
ground.texture!.size())

ground.physicsBody?.dynamic = false
```

That sets up pixel-perfect collision for the ground sprites, but makes them non-dynamic – that is, they will respond to physics in the game so that the plane hits the ground, but they won't get moved by the physics. Without this line the ground would drop off the screen thanks to gravity.

We can start to approach a playable game by making just two more changes. First, add these two lines to **touchesBegan()**:

```
player.physicsBody?.velocity = CGVectorMake(0, 0)
player.physicsBody?.applyImpulse(CGVectorMake(0, 20))
```

The second line means "give the player a push upwards every time the player taps the screen." The first line is there to make the physics a bit more realistic and it effectively neutralizes any existing upward velocity the player has before applying the new movement. Without that, the player could tap multiple times quickly and apply a huge upwards force to the plane, sending them miles off the top of the screen. With that line, the plane behaves much more like the "dodo" plane in the game *Grand Theft Auto: Vice City* – each upward thrust adds only a tiny bit of lift.

The second change is to make the player's movement more dramatic. It's going to take 1/1000th of the player's upward velocity (a tiny amount) and turn that into rotation. This means that when the player is moving upwards the plane tilts up a little, and when they player is falling the plane tilts down. It's a simple effect, but it really highlights the player's impending doom!

To make the effect nicer we'll add it as a **rotateToAngle()** action over a tenth of a second. This smooths out the rotation a little, but because it's happening more slowly than the game's frame rate it effectively means the rotation animation is always happening.

Add these two lines to **update()** now:

```
let value = player.physicsBody!.velocity.dy * 0.001
let rotate = SKAction.rotateToAngle(value, duration: 0.1)
```

```
player.runAction(rotate)
```

If you run your game now you'll see it's almost playable: the player falls towards the ground, and tapping keeps them in flight just a little bit longer. You can't collide with the rocks, but you *can* collide with the ground because of that commented line that modified **collisionBitMask**. I made it commented because you should be able to fly your play around then crash into the ground in various interesting ways – it's the best (read: most fun!) way to make sure your physics are configured correctly.

Please uncomment that line of code now so that the player can no longer bounce off the ground. Now for the interesting part: adding physics to the rocks. This is going to use pixel-perfect collisions for the rocks themselves, and rectangle physics for the red scoring rectangle. All three of them need to have their **dynamic** property set to **false** so your rocks don't fall of the screen.

So, we're going to make three changes, all in the **createRocks()** method. The first is just after the **let topRock =** line – add these two lines of code:

```
topRock.physicsBody = SKPhysicsBody(texture: rockTexture, size:
rockTexture.size())

topRock.physicsBody?.dynamic = false
```

The second change is just after the **let bottomRock =** line – add these two lines of code:

```
bottomRock.physicsBody = SKPhysicsBody(texture: rockTexture, size:
rockTexture.size())

bottomRock.physicsBody?.dynamic = false
```

Finally, add these two lines of code just after the **let rockCollision =** line:

```
rockCollision.physicsBody = SKPhysicsBody(rectangleOfSize:
rockCollision.size)
```

```
rockCollision.physicsBody?.dynamic = false
```

Again, we're using pixel-perfect collision for the rocks and simple rectangle physics for the score collision rectangle. It's worth me saying that per-pixel collision detection is substantially slower than rectangle- and circle-based detection, but in our simple game it's perfectly OK.

Before you run the game again, I'd like you to make one more change. Go to GameViewController.swift and you'll see these two lines of code:

```
skView.showsFPS = true
skView.showsNodeCount = true
```

We've just added quite a lot of physics to our game, and physics can be annoying to debug because it's invisible. Or at least it's invisible *by default* – SpriteKit can actually draw faint blue lines around all our game physics, which really helps make sure everything is configured correctly. Add these new line below the previous two:

```
skView.showsPhysics = true
```

If you run the game now and look closely you should be able to see the blue physics lines all around the rocks, ground and even the player. It's such a small thing, but trust me: it's a real time saver!

We still have one more thing to do before our game starts to be playable, and that's to add collisions between the player's plane and pretty much everything else in the game. We already configured the player to report back whenever it touches anything else that has physics, so we now need to implement the **didBeginContact()** method and take appropriate action.

First: what happens when the player touches a red score rectangle? Well, we gave those rectangles a specific name – "scoreDetect" – which means we can check to see whether the collision involved a node named "scoreDetect" and, if so, it means the player passed through

the rocks. When that happens we're going to remove the score rectangle from the game (so they can't somehow score double points by accident), play the "coin.wav" sound effect, and increment the score by one.

Here's the code – add this method to your **GameScene** class, just below **update()**:

```
func didBeginContact(contact: SKPhysicsContact) {
  if contact.bodyA.node?.name == "scoreDetect" ||
contact.bodyB.node?.name == "scoreDetect" {
    if contact.bodyA.node == player {
      contact.bodyB.node?.removeFromParent()
    } else {
      contact.bodyA.node?.removeFromParent()
    }


    let sound = SKAction.playSoundFileNamed("coin.wav",
waitForCompletion: false)
    runAction(sound)


    score += 1


    return
  }
}
```

There are four important things to note in that code:

1. It checks to see whether the contact's **bodyA** or **bodyB** property was a score detection rectangle. This is because we don't know whether the player collided with the rectangle or the rectangle collided with the player. That might sound weirdly philosophical, but trust me: it matters.
2. When you first play a sound in the simulator, expect your game to pause for half a second while the sound engine is initialized. This doesn't happen on devices, but it does make this

game extremely hard – at least until we fix it in the next chapter.

3. Adding one to the **score** property triggers the **didSet** property observer we created earlier, which means the score label will be updated.

4. I added a **return** line to the end because if the player collides with anything else we want to destroy them. This just means, "you hit something safe; don't continue in this method."

And now for the really interesting bit: making the player die when they touch any rock or the ground. Because the player's physics are configured to report back contact with absolutely everything, and because we just made **didBeginContact()** exit if the player touches a scoring rectangle, we can be sure that any code coming after our previous additions will only be executed if the player hit a rock or the ground.

When this happens, we want the player to die and the game to end. So, if the collision is between the player and anything else, we're going to create a smoky particle effect using the PlayerExplosion.sks asset you copied in at the beginning, play "explosion.wav", remove the player from the game, then change the game's **speed** property to be 0.

Add this code just before the end of **didBeginContact()**:

```
if contact.bodyA.node == player || contact.bodyB.node == player {

  if let explosion = SKEmitterNode(fileNamed: "PlayerExplosion") {

    explosion.position = player.position

    addChild(explosion)

  }


  let sound = SKAction.playSoundFileNamed("explosion.wav",
waitForCompletion: false)

  runAction(sound)


  player.removeFromParent()

  speed = 0

}
```

All that is old except for the last line: the **speed** property. All SpriteKit nodes can have actions attached to them, and by default they all run in real time – that is, one second in an action is equal to one second on a real clock. This **speed** property is a time multiplier that lets you adjust how fast actions attached to a node should run. It's 1.0 by default (real time), but you could make it 2.0 to make actions happen twice as fast. That is, "fade out over 5 seconds" would actually become "fade out over 2.5 seconds."

We're adjusting the **speed** property to 0 for our game scene, which in turns get inherited by all children – i.e., everything in the game. This has the effect of halting all those move actions we added to make parallax scrolling work, effectively ending the game.

If you run the game now you'll see it's basically done: you can tap to fly high, stop tapping to fall, fly through rocks to score points, or crash into something else to die in an explosion. We could very easily stop here, but I'm going to go a bit further and add some extra polish. Partly because polish is fun, but mostly because it gives me a chance to introduce you to another new feature in iOS 9…

Background music with SKAudioNode, an intro, plus game over

To make this a finished game – or at least as finished as it can be before getting into tiny minutiae – we're going to make four more changes: we're going to add background music, show an intro screen, show a game over screen, and let the player try again when they die. None of these things are difficult, but it's a chance to polish your skills while polishing the game so hopefully you won't skip this out!

First up: background music. New in iOS 9 is a SpriteKit class called **SKAudioNode**, which adds several key new features to audio in SpriteKit, such as the ability to pan your audio left and right. For our purposes, however, **SKAudioNode** is good because it lets us stop the audio whenever we want. Remarkably, annoyingly, this was not part of SpriteKit for iOS 7 and 8, so you had to resort to using something like **AVAudioPlayer** for any sounds you wanted to stop prematurely.

One of the neat features of **SKAudioNode** is that it loops its audio by default. This makes it perfect for background music: we create the music, add it directly to the game scene as a child, and it plays our background music forever. It also has the happy side effect of starting the iOS Simulator's sound system as soon as the game begins, which means you won't have your game freeze the first time the player touches a red scoring rectangle.

Add a property for the background music now:

```
var backgroundMusic: SKAudioNode!
```

Then add this to **didMoveToView()**:

```
backgroundMusic = SKAudioNode(fileNamed: "music.m4a")
addChild(backgroundMusic)
```

Note: if you value your sanity, you'll probably want to run your game now to make sure the music works (yes, that code is all it takes!) then comment out those two lines so you don't

have to listen to the music on repeat for the rest of the time you work on the game.

And yes, that's all it takes to add looping background music – hurray for **SKAudioNode**!

The next change we're going to make is to add an intro screen when the game starts. I'm just going to make mine show the game's logo – "Crashy Plane" – over the game screen, with the player's plane flying in the background. When the player taps the first time, the game will begin. In a few minutes we're going to add a game over screen too, which means we have three possible game states: showing the logo, playing the game, and dead. We'll represent that with a dedicated enum, so add this just before the start of your **GameScene** class – i.e., just after the **import** lines:

```
enum GameState {
    case ShowingLogo
    case Playing
    case Dead
}
```

We need to create three more properties to make all this work: one to hold the logo sprite node, one to hold the game over sprite node, and one to keep track of the current game state. The game state will be **ShowingLogo** by default, which means the game won't start until the player is ready. Add these properties now:

```
var logo: SKSpriteNode!
var gameOver: SKSpriteNode!

var gameState = GameState.ShowingLogo
```

Creating the logo and game over sprite nodes is nothing special: they are just simple pictures, and we can use the **CGRectGetMidX()** and **CGRectGetMidY()** functions to position them at the center of our game scene. As you might imagine, we need to set the **alpha** property of the game over sprite to be 0 to begin with otherwise it would be quite

confusing!

Here's the **createLogos()** method:

```
func createLogos() {
    logo = SKSpriteNode(imageNamed: "logo")
    logo.position = CGPoint(x: CGRectGetMidX(frame), y:
CGRectGetMidY(frame))
    addChild(logo)


    gameOver = SKSpriteNode(imageNamed: "gameover")
    gameOver.position = CGPoint(x: CGRectGetMidX(frame), y:
CGRectGetMidY(frame))
    gameOver.alpha = 0
    addChild(gameOver)
}
```

That's not enough to make the game start in menu mode, though. First, add a call to **createLogos()** inside **didMoveToView()**. While you're there, *delete* the call to **initRocks()** because it's no longer needed – we don't want to start creating rocks before the game begins. Finally, do you remember this line of code in the **createPlayer()** method?

```
player.physicsBody?.dynamic = true
```

When I was explaining what it did, I said it "makes the plane respond to physics. This is the default, but I'm including it here because we'll change it later." Well, now it's time to change it: by changing that **true** to be **false** the player will stop responding to physics. It will still have physics attached to it ready to be used, but it won't actually do anything.

This is perfect for our game, because we want everything set up ready to go, but we *don't* want the player to start moving until we're ready. So, change that line to this:

```
player.physicsBody?.dynamic = false
```

If you run the game now you'll see it looks pretty good: the player no longer moves (even when you tap the screen) and the logo floats over the game nicely. Now we're going to rewrite **touchesBegan()** so that it distinguishes between a touch when in **ShowingLogo** mode and a touch when in **Playing** mode.

The code for touches while playing hasn't changed, so I'm not going to discuss it further, but the code for **ShowingLogo** mode is new. This needs to change the game state to be **Playing** (so that further touches move the plane), make the logo fade out and get removed from the game, wait a tiny amount, then activate the player. It also needs to call **initRocks()** so that rocks start being created at random intervals.

Because the **GameState** enum has three possible cases, and Swift likes all your switch/case statements to be exhaustive, we're going to add an empty case for **Dead** that we'll fill in shortly. Here's the new code for **touchesBegan()**:

```
override func touchesBegan(touches: Set<UITouch>, withEvent event:
UIEvent?) {
  switch gameState {
  case .ShowingLogo:
    gameState = .Playing


    let fadeOut = SKAction.fadeOutWithDuration(0.5)
    let remove = SKAction.removeFromParent()
    let wait = SKAction.waitForDuration(0.5)
    let activatePlayer = SKAction.runBlock { [unowned self] in
      self.player.physicsBody?.dynamic = true
      self.initRocks()
    }


    let sequence = SKAction.sequence([fadeOut, wait, activatePlayer,
```

```
remove])

    logo.runAction(sequence)


  case .Playing:

    player.physicsBody?.velocity = CGVectorMake(0, 0)

    player.physicsBody?.applyImpulse(CGVectorMake(0, 20))


  case .Dead:

    break

  }

}
```

You might think that removing the logo from the game is going to cause problems when we add the ability for the player to start again, but don't worry – it will all make sense soon!

Adding a simple introduction screen helps players prepare for the game, and also gives you a chance to remind them of your awesome brand name. Or not.

So, that's our game start sequence in place: all that's needed now is to end the game when the player dies. This is going to do three things: change the **alpha** of the game over sprite to be 1 (fully visible), change the game state to be **Dead** so we can respond to touches differently, and stop the background music to give an extra little sense of loss.

In **didBeginContact()**, add these three lines of code just before the call to **player.removeFromParent()**:

```
gameOver.alpha = 1

gameState = .Dead

backgroundMusic.runAction(SKAction.stop())
```

Once the game state is **Dead** the player's taps stop doing anything, which is lucky because the player is dead! However, what we really want is for player taps to start the game afresh, and the easiest way to do that is to present a whole new **GameScene** scene. This causes the whole game to be reset: a new score, a new player, a new logo sprite, no more rocks, etc, and it's significantly easier than trying to reset everything by hand.

Right now in **touchesBegan()** there is a simple **break** line for the **Dead** game state. Change it to this:

```
let scene = GameScene(fileNamed: "GameScene")!

scene.scaleMode = .ResizeFill

let transition =
SKTransition.moveInWithDirection(SKTransitionDirection.Right,
duration: 1)

self.view?.presentScene(scene, transition: transition)
```

That creates a fresh **GameScene** scene, applies the same **ResizeFill** tweak we made way back at the beginning of this tutorial to compensate for the iPad-sized scene, then makes it

transition in with a simple animation.

But wait! Before you run the game – and I'm sure you're eager – there is one tiny further tweak to make. You see, we have an **update()** method that adjusts the rotation of the player every frame, but we also don't create the player until **didMoveToView()** is called. If the update method is called first (and it is!) then Swift will try to adjust the rotation of a nil property because the player hasn't been created yet, which will make your game crash.

The solution is simple, thanks to the **guard** keyword – just add this line to the start of the **update()** method:

```
guard player != nil else { return }
```

Translated, that single line means "ensure that player is not nil, otherwise exit the method."

That's it! The game is done. I hope you agree it looks good, although I can't take any credit for that – it's the **marvellous art of Kenney** that should take all the credit, and I do encourage you again to check out his complete pack of public domain game assets.

As final touches, you should set the score rectangles to have the color **UIColor.clearColor()** so they are invisible, then go to GameViewController.swift and turn off **showsFPS**, **showsNodeCount** and **showsPhysics**.

Wrap up

This wasn't a complicated project, but I hope it was a satisfying one. The addition of new techniques like **GKRandomDistribution**, **SKAudioNode** and **guard** should have made it more interesting even for more experienced coders, and it was fun adding the extra bit of polish at the end to make the whole game feel more complete.

If you want to take this project further, you could start by having different kinds of obstacles – the repeating rocks do get a bit tiresome after a while! You could also make the game difficulty ramp up ever so slowly, either by decreasing the gap between the rocks or by increasing the world gravity. To make the game much more challenging, how about introducing a secondary scoring mechanism: perhaps the player could get extra points if they fly through hoops in between the rocks? If you fancy a bigger challenge, how about making it a universal game, i.e. support both iPad and iPhone.

Project 37

Psychic Tester

Are you psychic? Of course not. But what if we could use our coding skills to make a game to fool your friends into thinking otherwise?

Setting up

Are you psychic? Of course not. But what if we could use our coding skills to make a game to fool your friends into thinking otherwise – while also learning some new techniques along the way?

In this project we're going to build a simple game that recreates the classic **Zener test** for extrasensory perception. Our game will show the user eight cards face down, and users need to tap the card that has a star on its flip side. Casual players will get it right 1 in 8 times, but you'll get it right every time. Magic!

Well, no. We'll be cheating, naturally, but even in this cheating I'm going to find new things to teach you. First, we're going to build a tiny watchOS app that silently taps your wrist when your finger moves over the star card. Then we're going to add 3D Touch support so that pressing hard on any card will automaticaly make it the star. Whichever technique you use is going to be enough to baffle your friends, although I hope you use your powers for good!

At the same time we'll also be learning about **CAEmitterLayer**, **CAGradientLayer**, **@IBDesignable** and **@IBInspectable**, as well as how to create a 3D card flip effect using the **transitionWithView()** method.

I've left the Apple Watch and 3D Touch code until the end of the project, so at the very least you'll be able to work through the majority of the tutorial without needing special hardware. That being said, we'll be using one of Xcode's built-in iOS/watchOS templates to make the end result easier to reach.

Are you ready to take your first step into the Twilight Zone? Go ahead and launch Xcode, then create a new project. When Xcode asks you which template you want, please select watchOS then iOS App with WatchKit App. Set the target to be iPhone, the language to be Swift, then deselect Include Notification Scene. Finally, name it Project37 and click Next.

Once the project is created, please set it to support landscape left and right only; no portrait this time. You will also need to copy the image assets for this project into your project's asset catalog – you can **find them all on GitHub**. You'll also find a Content directory in that download; please add that to your project too, because it contains some music for later on.

Laying out the cards: addChildViewController()

The first step in our project will be to lay eight cards out on the screen so that the user can tap on one. We'll be doing most of this in code, but there is a small amount of storyboard work required.

Open up Main.storyboard in Interface Builder, then draw out a large **UIView** inside the view controller that was made by the template. Set its size to be 480x320 using explicit width and height constraints, then make it centered inside its parent view. This explicit sizing makes it easy to support the full range of iPhones: we'll place the cards inside this container at exact positions, and the container will be moved depending on the size of the device.

We'll be placing something behind this view later on, so please set its background color to be Clear Color. Then, using the Assistant Editor, please make an outlet connection between your new view and the ViewController class, naming it **cardContainer**.

That's it: we're done with Interface Builder for now; the rest of this chapter will all be done using code.

In the screenshot below you can see how your interface should look – note that I've temporarily colored my inner view gray so you can see it more clearly!
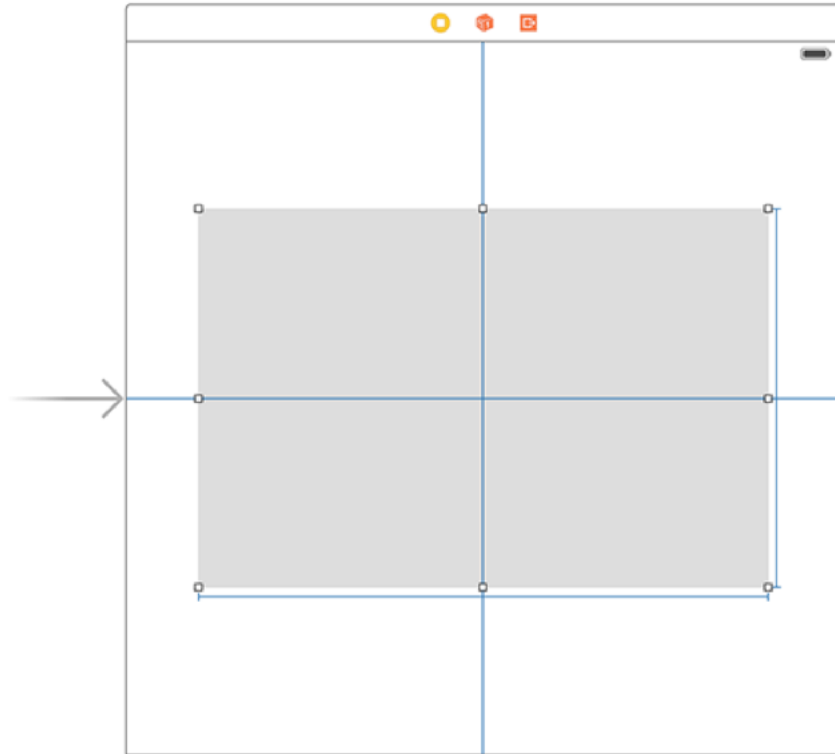
Our very basic interface just consists of one view, but yours should be set to 'clear color' at this point.

To help isolate functionality, we'll be creating a special **UIViewController** subclass to handle each card. We'll then add this view controller to an array of all card view controllers, and add them all to our main view controller so the player can see them.

So, add a new file to the project and choose iOS > Source > Cocoa Touch Class then click Next. For "Subclass of" please enter **UIViewController** and for the class name enter **CardViewController**.

To make things nice and easy, each card view controller will contain two child image views: one for its card back and one for its card front. We're also going to give them two extra properties: one to mark whether the card is correct (i.e. a star) and one to hold a weak reference to the main **ViewController** class so we can send messages back.

Add these four properties to the **CardViewController** class now:

```
weak var delegate: ViewController!


var front: UIImageView!

var back: UIImageView!


var isCorrect = false
```

The basic Xcode class has a **viewDidLoad()** stub provided for us, but it doesn't do anything interesting. We need to upgrade this method to do the following:

1. Give the view a precise size of 100x140.
2. Add two image views, one for the card back and one for the front.
3. Set the front image view to be hidden by default.
4. Set the back image view to have an alpha value of 0 by default, fading up to 1 with an animation.

That last point isn't strictly needed, but it does make the whole application look nicer.

None of the code to accomplish this is difficult, but I'm going to draw on a particular feature of **UIImageView** to make things easier. The feature is this: if you create an image view using a **UIImage**, the image view gets set to the size of that image automatically. This is perfect, because all our card images are sized at 100x140, so our view and its card contents will all line up.

Here's the new **viewDidLoad()** method for the **CardViewController** class:

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


  view.bounds = CGRectMake(0, 0, 100, 140)

  front = UIImageView(image: UIImage(named: "cardBack"))

  back = UIImageView(image: UIImage(named: "cardBack"))
```

```
    view.addSubview(front)

    view.addSubview(back)


    front.hidden = true

    back.alpha = 0


    UIView.animateWithDuration(0.2) {

        self.back.alpha = 1

    }

}
```

You'll note that I'm using the "cardBack" for both the front and the back image views. This is just for sizing purposes: the actual front image will be assigned later. Helpfully, **UIImage** shares image data across image views very efficiently, so there's no extra cost to this approach.

That's all the code required to make each card work, but it doesn't actually display them in the view. To do that, we need to return to ViewController.swift and create a new property to hold all the card view controllers, plus a new method to create them all. This creation method will also be responsible for clearing any existing cards, so that with a single method call we can wipe the slate clean and start again.

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

```
var allCards = [CardViewController]()
```

While you're up there, please add an import for **GameplayKit** because we'll be using it's array shuffling method.

Now for the new method, **loadCards()**. This needs to assemble an array of positions where cards can go (I've made some rough estimates, but you're welcome to be more precise in your own code if you want to), load the various Zener card shapes (one for each of the eight cards), then create one card view controller for each position.

So far, so easy. But this time there is going to be one extra step, because I want to introduce to you the concept of *view controller containment*. When you place one view controller inside another, it can cause problems with system events (think appearing, disappearing, rotating, etc.) because iOS wasn't originally designed to have multiple view controllers visible at the same time.

View controller containment is a simple solution where you use the methods **addChildViewController()** and **didMoveToParentViewController()** to place one view controller inside another. It's extremely easy to do, and it means iOS can keep track of all the view controllers correctly, so it is very much recommended.

Please add the new method below to your **ViewController** class. I've added comments just in case you're not sure what everything does:

```
func loadCards() {
  // create an array of card positions
  let positions = [
    CGPoint(x: 75, y: 85),
    CGPoint(x: 185, y: 85),
    CGPoint(x: 295, y: 85),
    CGPoint(x: 405, y: 85),
    CGPoint(x: 75, y: 235),
    CGPoint(x: 185, y: 235),
    CGPoint(x: 295, y: 235),
    CGPoint(x: 405, y: 235)
  ]

  // load and unwrap our Zener card images
  let circle = UIImage(named: "cardCircle")!
  let cross = UIImage(named: "cardCross")!
  let lines = UIImage(named: "cardLines")!
  let square = UIImage(named: "cardSquare")!
```

```
    let star = UIImage(named: "cardStar")!


  // create an array of the images, one for each card, then shuffle
it
  var images = [circle, circle, cross, cross, lines, lines, square,
star]
  images =
GKRandomSource.sharedRandom().arrayByShufflingObjectsInArray(images)
as! [UIImage]


  for (index, position) in positions.enumerate() {
    // loop over each card position and create a new card view
controller
    let card = CardViewController()
    card.delegate = self


    // use view controller containment and also add the card's view
to our cardContainer view
    addChildViewController(card)
    cardContainer.addSubview(card.view)
    card.didMoveToParentViewController(self)


    // position the card appropriately, then give it an image from
our array
    card.view.center = position
    card.front.image = images[index]


    // if we just gave the new card the star image, mark this as the
correct answer
    if card.front.image == star {
      card.isCorrect = true
    }
```

```
    // add the new card view controller to our array for easier
tracking

    allCards.append(card)

  }

}
```

Now you just need to add a call to that method inside **viewDidLoad()** like this:

```
override func viewDidLoad() {

  super.viewDidLoad()


  loadCards()

}
```

There's one last thing we need to add before we're done with card loading: we need to make **loadCards()** remove any existing cards. This will allow us to call **loadCards()** repeatedly and have it do the right thing.

Please add this code at the start of **loadCards()**:

```
for card in allCards {

  card.view.removeFromSuperview()

  card.removeFromParentViewController()

}


allCards.removeAll(keepCapacity: true)
```

That just acts as an "undo" for the rest of the method: it loops through all the card view controllers we stored in the **allCards** array, removes the view then removes the view controller containment, then clears the whole array.

Go ahead and run your project now, and you should see eight cards neatly lined up in two rows like the screenshot below – albeit without the gray background. They don't do anything yet, but it's a good start.
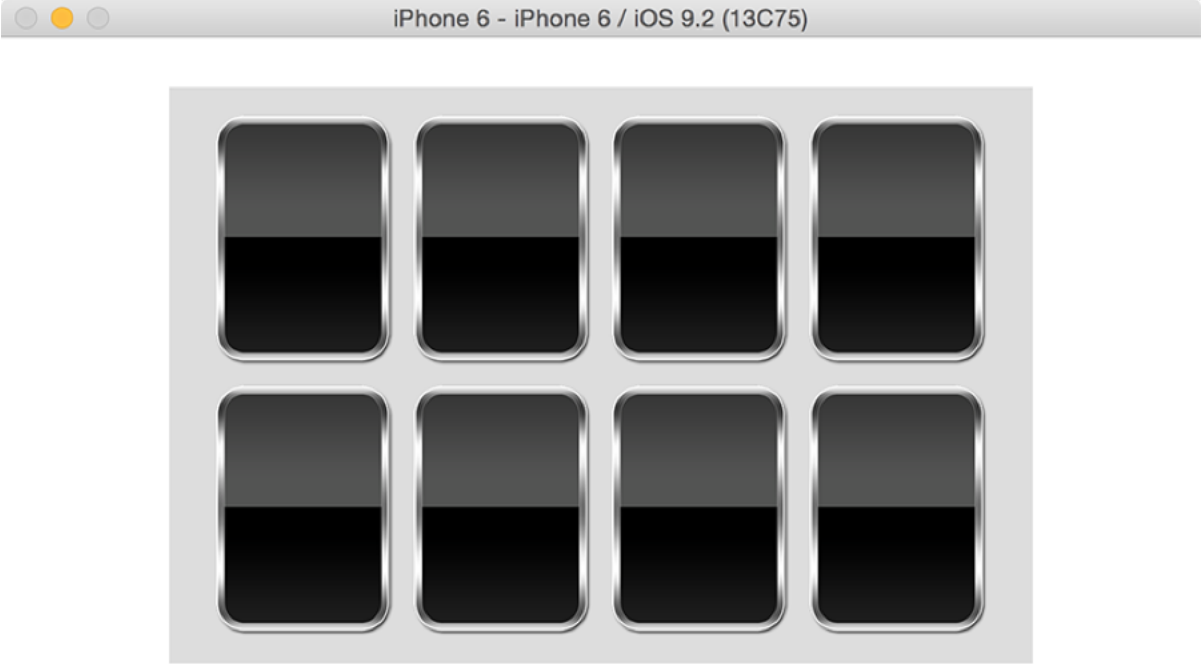
Our code places two rows of four cards into the cardContainer view.

Animating a 3D flip effect using transitionWithView()

There's a reason I've made you put the card functionality into a separate view controller, and it's because we're going to be adding some functionality to cards to handle them being flipped over. iOS makes this kind of animation really easy, but it's done in a slightly different way to our previous animations.

To handle tap detection we're going to use a **UITapGestureRecognizer** rather than something like **touchesBegan**. This will make more sense later on, but the TL;DR version is that part of the hoax effect will be you running your finger over the cards using your powers to "feel" for the star – something like **touchesBegan()** will just cause problems.

So, please add this gesture recognizer to the end of **viewDidLoad()** in the **CardViewController** class:

```
let tap = UITapGestureRecognizer(target: self, action: "cardTapped")
back.userInteractionEnabled = true
back.addGestureRecognizer(tap)
```

We haven't written the **cardTapped()** method yet, but it's trivial because all it will do is pass the message on to the **ViewController** class to handle. This is important: we need each card to decide if it was tapped, but we need to pass control onto the **ViewController** class to act upon the tap, otherwise it's possible users might tap two cards at the same time and cause problems.

So, the **cardTapped()** method in the card view controller is simple:

```
func cardTapped() {
  delegate.cardTapped(self)
}
```

Of course, that just pushes all the work to the **ViewController** class, where things get more complicated. The **cardTapped()** method there needs to:

 • Ensure that only one card can be tapped at any time
 • Loop through all the cards in the **allCards** array.
 • When it finds the card that was tapped, animate it to flip over then fade away.
 • For all other cards, animate them fading away.
 • Reset the game after two seconds so that more cards appear.

We'll be doing the animation using methods inside **CardViewController**, and resetting the game is done just by calling **loadCards()**, so that's all straightforward. But what's the best way to ensure that only one card can be chosen by the player?

It turns out this is pretty easy: as soon as the user taps any card, we're going to disable user interaction for our main view. We can then check that property inside the **cardTapped()** method using the **guard** keyword, then set it back to true inside **loadCards()**.

To make things slightly more interesting, I want to introduce you to the **performSelector()** method family. These exist on objects that inherit from **NSObject**, which is both our view controllers, and allow us to call a method after a delay or in the background really easily.

Let's take this step by step. First, here's the **cardTapped()** method for the **ViewController** class:

```
func cardTapped(tapped: CardViewController) {
  guard view.userInteractionEnabled == true else { return }
  view.userInteractionEnabled = false


  for card in allCards {
    if card == tapped {
      card.wasTapped()
      card.performSelector("wasntTapped", withObject: nil,
afterDelay: 1)
    } else {
      card.wasntTapped()
```

```
    }

  }


  performSelector("loadCards", withObject: nil, afterDelay: 2)

}
```

You can see that calls **wasTapped()** and **wasntTapped()** methods in the card view controllers, each of which will perform some animation – we'll get onto that in a moment. Using the **afterDelay** variant of **performSelector()** will cause **wasntTapped()** to be called after 1 second, and **loadCards()** to be called after 2 seconds.

For now, focus on the first two lines of that method: that's what stops users tapping two cards at once. By disabling the user interaction (and also checking that it was enabled beforehand) we can be sure the user gets to make only one choice. But we do need to re-enable user interaction when we're done, otherwise our app will be useless.

So, add this line somewhere into the **loadCards()** method:

```
view.userInteractionEnabled = true
```

Now all we need to do is write the **wasTapped()** and **wasntTapped()** methods of the card view controller. We'll do **wasntTapped()** first because it uses code you already know, so re-open CardViewController.swift and add this:

```
func wasntTapped() {

  UIView.animateWithDuration(0.7) {

    self.view.transform = CGAffineTransformMakeScale(0.00001,
0.00001)

    self.view.alpha = 0

  }

}
```

That tells the card to zoom down and fade away over 0.7 seconds. Things are more interesting in the **wasTapped()** method because it needs to animate a 3D flip effect from the card back to the card front. But if you were imagining this was going to be hard, you're wrong: this flip effect has been around since the earliest days of iOS, so Apple made it extremely easy.

Here is the **wasTapped()** method in its entirety:

```
func wasTapped() {

  UIView.transitionWithView(view, duration: 0.7, options:
[.TransitionFlipFromRight], animations: { [unowned self] in

    self.back.hidden = true

    self.front.hidden = false

  }, completion: nil)
}
```

As you can see, all the work is done by the **transitionWithView()** method. This takes a view to operate on as its first parameter, and all the animations you perform need to be done on subviews of this container view. We pass **.TransitionFlipFromRight** to create the flip effect, but you should try using the code completion to explore other options.

Inside the animations block, we just adjust the **hidden** properties of the front and back image views, but in the context of **.TransitionFlipFromRight** that will cause iOS to animate this change as a flip – it really is that simple.

That's it! Run the project now and you'll find you can tap on any card to flip it over – a neat effect with hardly any code. Thanks, iOS!

Adding a CAGradientLayer with IBDesignable and IBInspectable

Magic is really the art of misdirection: making people focus their attention on one thing in order to stop them focusing on something else. In our case, we don't want users to suspect that your Apple Watch is helping you find the star, so we're going to overload them with misdirection so that they suspect everything else before they think of your watch.

The first thing we're going to do is add a background to our view. This is going to be a simple gradient, but we're going to make the gradient change color slowly between red and blue. This has no impact on your ability to find the star, but if it makes your friends suspect that the trick is to tap a card when the background is red then it has fulfilled its job of misdirection.

Making gradients in iOS isn't hard thanks to a special **CALayer** subclass called **CAGradientLayer**. That being said, working with layers directly isn't pleasant, because they can't take part in things like Auto Layout and they can't be used inside Interface Builder.

So, I'm going to teach you how to wrap a gradient inside a **UIView**, while also adding the benefits of letting you control the gradient right from within Interface Builder. What's more, you'll be amazed at how easy it is.

Add a new Cocoa Touch subclass to your project. Make it a subclass of **UIView** then name it **GradientView**. We need this class to have a top and bottom color for our gradient, but we also want those values to be visible (and editable) inside Interface Builder. This is done using two new keywords: **@IBDesignable** and **@IBInspectable**.

The first of those, **@IBDesignable**, means that Xcode should build the class and make it draw inside Interface Builder whenever changes are made. This means any custom drawing you do will be reflected inside Interface Builder, just like it would when your app runs for real.

The second new keyword, **@IBInspectable**, exposes a property from your class as an editable value inside Interface Builder. Xcode knows how to handle various data types in meaningful ways, so strings will have an editable text box, booleans will have a checkbox, and colors will have a color selection palette.

Other than defining properties for the top and bottom colors of the gradient, the **GradientView** class needs to do only two other things to be complete: when iOS asks it what kind of layer to use for drawing it should return **CAGradientLayer**, and when iOS tells

the view to layout its subviews it should apply the colors to the gradient.

Using this approach means the entire class is just 13 lines of code, *including* whitespace and closing braces. Here's the code for the **GradientView** class:

```
@IBDesignable class GradientView: UIView {
  @IBInspectable var topColor: UIColor = UIColor.whiteColor()
  @IBInspectable var bottomColor: UIColor = UIColor.blackColor()

  override class func layerClass() -> AnyClass {
    return CAGradientLayer.self
  }

  override func layoutSubviews() {
    (layer as! CAGradientLayer).colors = [topColor.CGColor,
bottomColor.CGColor]
  }
}
```

With that new class in place, it's time to return to Interface Builder and add it to our layout. To do this, draw out another **UIView**, but make sure it stretches from edge to edge this time. Use the Pin button to add Auto Layout rules so the new view stays edge to edge no matter what size the main view becomes, then finally go to the Editor menu and choose Arrange > Send To Back to make sure the new view sits behind the card container.

We want this new view to be a **GradientView**, which is done by changing its class. Press Alt +Cmd+3 to bring up the identify inspector on the right, then look at the very top for a dropdown list of classes you can use for the new view. Look in there for "GradientView", and you'll see "Designables: Updating" appear.

After a few seconds, you should see a white to black gradient appear in Interface Builder, which shows the default colors we set. But we made those colors inspectable, so if you press Alt+Cmd+4 to go to the Attributes Inspector you should see "Top Color" and "Bottom Color" ready for you to choose – yes, Xcode has correctly converted **topColor** into "Top

Color" thanks to our property naming convention.

We'll be applying red and blue colors separately to the gradient, so please set "Top Color" to be "Dark Gray Color", and "Bottom Color" to be "Black Color". Finally, set the alpha value for the gradient view to be 0.9, so a little bit of the background view shows through.

Before we're done with Interface Builder (for real this time!) please use the Assistant Editor to create an outlet for this new gradient view called **gradientView**. We don't need this just now, but it's important in the next chapter.

If everything is correct, your interface should look like the screenshot below. As before, I've colored my container view so you can see it, but yours should have Clear Color for its background color.
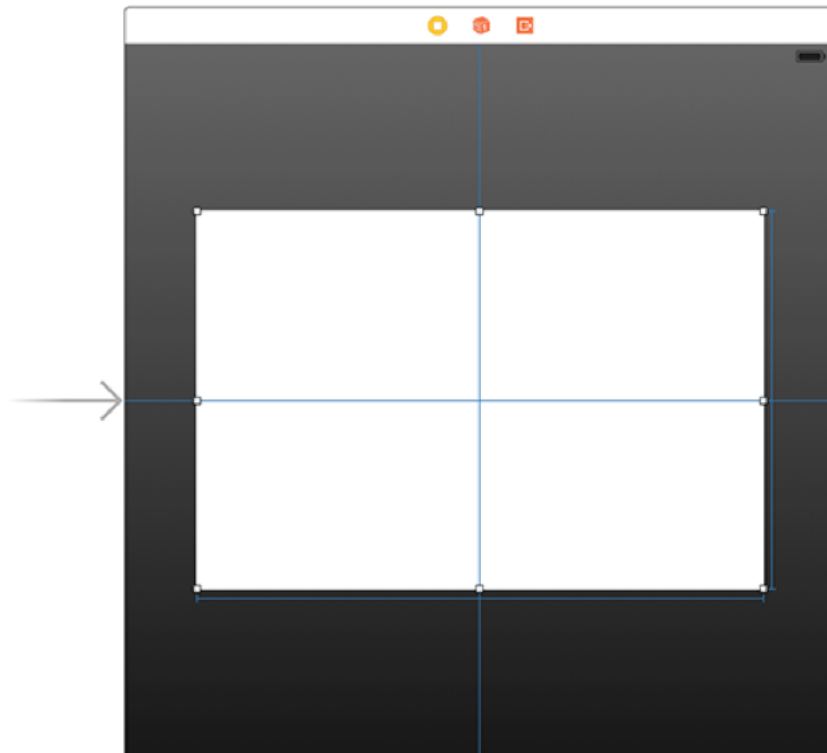
CAGradientLayer plus IBDesignable make great Interface Builder buddies.

With all those interface changes in place, we can animate the background color of the main view in just a handful of lines of code. To make this work, we'll be using three animation options: **.AllowUserInteraction** (so the user can tap cards), **.Autoreverse** to make the view

go back to its original color, and **.Repeat** to make the animation loop back and forward forever.

Place this animation code somewhere in **viewDidLoad()**:

```
view.backgroundColor = UIColor.redColor()


UIView.animateWithDuration(20, delay: 0, options:
[.AllowUserInteraction, .Autoreverse, .Repeat], animations: {

  self.view.backgroundColor = UIColor.blueColor()

}, completion: nil)
```

Note that we need to give the view an initial red color to make the animation smooth, but you can put that in Interface Builder if you prefer.

Creating a particle system using CAEmitterLayer

Let's take our misdirection up a notch by adding some falling, spinning stars behind the cards. Again, these do nothing other than misdirect your friends while also giving me a chance to squeeze some new learning into you.

We first met particle systems **in project 11** when we covered **SKEmitterNode**. That's a fast and easy way to create particle systems in SpriteKit, but we're not in SpriteKit now so we need an alternative.

Fortunately, iOS has one, and in fact it even predates **SKEmitterNode**: **CAEmitterLayer**. From its name you should already be able to tell that it's a subclass of **CALayer**, which in turn means you need to use **CGColor** rather than **UIColor** and **CGImage** rather than **UIImage**. However, I should add that **CAEmitterLayer** isn't quite a beautifully polished as **SKEmitterNode** – it has no WYSIWYG editor, for example, so you need to do everything in code.

Each **CAEmitterLayer** defines the position, shape, size and rendering mode of a particle system, but it doesn't actually define any particles – that's handled by a separate class, called **CAEmitterCell**. You can create as many emitter cells as you want, then assign them to your emitter layer to have them all fire from the same position.

There are *lots* of properties you can set on emitter cells, and without a WYSIWYG editor you're basically stuck reading the documentation to find them all. To give you a jump start, I'm going to use quite a few to make our particle system:

- **birthRate** sets how many particles to create every second.
- **lifetime** sets how long each particle should live, in seconds.
- **velocity** sets the base movement speed for each particle.
- **velocityRange** sets how much velocity variation there can be.
- **emissionLongitude** sets the direction particles are fired.
- **spinRange** sets how much spin variation there can be between particles.
- **scale** sets how large particles should be, where 1.0 is full size.
- **scaleRange** sets how much size variation there can be between particles.
- **color** sets the color to be applied to each particle.
- **alphaSpeed** sets how fast particles should be faded out (or in) over their lifetime.
- **contents** assigns a **CGImage** to be used as the

Broadly speaking, each property has "Speed" and "Range" counterparts, where "speed" dictates how much the value changes over time, and "range" dictates how much variation there is in the initial value. So, **scale** also has **scaleSpeed** and **scaleRange** alongside it.

Alongside those cell properties, we're also going to give the particle system some basic properties: we want to position it at the horizontal center of our view and just off the top, we want it to be shaped like a line so that particles are created across the width of the view, we want it to be as wide as the view but only one point high, and, as a bonus, we want it to use additive rendering so that overlapping particles get brighter.

Now that you know how it all works, please add this method to the **ViewController** class:

```
func createParticles() {
  let particleEmitter = CAEmitterLayer()


  particleEmitter.emitterPosition =
CGPointMake(view.frame.size.width / 2.0, -50)
  particleEmitter.emitterShape = kCAEmitterLayerLine;
  particleEmitter.emitterSize = CGSizeMake(view.frame.size.width, 1);
  particleEmitter.renderMode = kCAEmitterLayerAdditive;


  let cell = CAEmitterCell()
  cell.birthRate = 2
  cell.lifetime = 5.0
  cell.velocity = 100
  cell.velocityRange = 50
  cell.emissionLongitude = CGFloat(M_PI)
  cell.spinRange = 5
  cell.scale = 0.5
  cell.scaleRange = 0.25
  cell.color = UIColor(white: 1, alpha: 0.1).CGColor
  cell.alphaSpeed = -0.025
```

```
    cell.contents = UIImage(named: "particle")?.CGImage
    particleEmitter.emitterCells = [cell]


    gradientView.layer.addSublayer(particleEmitter)
}
```

Note that I'm adding the particle emitter as a sublayer of the **gradientView** view. This is important, because it ensures the stars always go behind the cards. You will also need to a call to **createParticles()** to the view controller's **viewDidLoad()** method, just before the call to **loadCards()**.

Go ahead and run the project now and I think you'll find the effect quite pleasing – it's subtle, yes, but again it's just enough to distract users into thinking maybe, just maybe, the position of the stars tells you where the green star card is.

With a color-shifting background gradient and falling stars your users will have no idea what's going on.

Wiggling cards and background music with AVAudioPlayer

The last part of our misdirection is going to be truly evil. That being said, it is entirely optional because I won't be teaching any vital new techniques here – I just enjoy screwing with my friends' heads!

We're going to add two more simple distractions to our app. First, we're going to make random cards move ever so slightly on the screen. The movement has to be small so that people catch it in the corner of their eye, but then aren't 100% sure anything actually happened. Second, we're going to add some background music to make people wonder whether there's something in the sound effects that tells you where the star is.

Making the cards move just a bit is easy thanks to the method **performSelector(_:withObject:afterDelay:)** that I introduced earlier. We're going to write a new method that scales a card so that it's a mere 1% larger than normal, before dropping it down again. To make things more interesting, we want this animation to happen only occasionally, so the users aren't sure when it will happen again.

This misdirection is clever because human eyes are extremely sensitive to motion at the edges of vision, so your eye notices a card moves and jumps to it, but of course by then our animation has stopped so your user isn't sure whether anything happened. If you want to create a more pronounced effect, just increase the transform scale that gets applied.

Open CardViewController.swift and add this new method:

```
func wiggle() {
  if GKRandomSource.sharedRandom().nextIntWithUpperBound(4) == 1 {
    UIView.animateWithDuration(0.2, delay: 0,
options: .AllowUserInteraction, animations: {
      self.back.transform = CGAffineTransformMakeScale(1.01, 1.01);
    }) { _ in
      self.back.transform = CGAffineTransformIdentity;
    }
```

```
        performSelector("wiggle", withObject: nil, afterDelay: 8)
    } else {
        performSelector("wiggle", withObject: nil, afterDelay: 2)
    }
}
```

As always, make sure you add **import GameplayKit** to be able to use **GKRandomSource**.

There are two things of interest in that new method. First, I've used the **.AllowUserInteraction** animation option so that users can tap a card even when it's animating. Second, the method calls itself so that the wiggle animation happens repeatedly, but, in a particularly evil twist, the delay is much longer after a card already moved. This means if someone's eye jumps to a card when they think it moved, they'll have to stare at it for a full eight seconds before it moves again.

Once the **wiggle()** method has been called once it will carry on calling itself, so we just need to make that initial call to get things moving. To do that, add this code to the end of **viewDidLoad()** for the card view controller:

```
performSelector("wiggle", withObject: nil, afterDelay: 1)
```

The very last piece of misdirection is an easy one: making some music play. Some mystic-sounding music was in the Content folder you should have downloaded from GitHub in the first chapter, and is a piece of music called "Phantom from Space" by Kevin MacLeod. It's licensed under Creative Commons Attribution 3.0 – see **this link** for more information.

You should already have added the Content folder to your project, so all that's left is to use it. This is done with four small changes in ViewController.swift, starting with this import to the top:

```
import AVFoundation
```

Now create a property to hold our music audio:

```
var music: AVAudioPlayer!
```

Next we need to create a **playMusic()** method that loads in the music and plays it. This is almost identical to code we've covered before, but there is a small change because we need the music to loop. This is done by setting the audio player's **numberOfLoops** property to any negative number, such as -1. Here's the new method, again for ViewController.swift:

```
func playMusic() {

  if let musicURL =
NSBundle.mainBundle().URLForResource("PhantomFromSpace",
withExtension: "mp3") {

    if let audioPlayer = try? AVAudioPlayer(contentsOfURL: musicURL)
{

      music = audioPlayer

      music.numberOfLoops = -1

      music.play()

    }

  }

}
```

The fourth and final change is just to call that new **playMusic()** method from within the view controller's **viewDidLoad()** method. So, add this to the end:

```
playMusic()
```

That completes our misdirections: we've added a shifting color gradient, we've added falling stars, we've made the cards move, and now we've added music too. With so many distractions in place hopefully your friends won't be able to guess the trick.

Speaking of tricks, that's our very next job: how to fix the app so you always guess correctly!

How to measure touch strength using 3D Touch

3D Touch is a new technology that was first trialled in Apple Watch as Force Touch, but introduced fully inside the iPhone 6s. In iOS it's responsible for multiple interesting technologies: peek and pop (to preview and jump into view controllers), application shortcuts (menus on the home screen for common actions) and also pressure-sensitive taps for **UITouch**.

All three of these are surprisingly simple to do, but in this project we're going to use only the last one, and I think you'll be impressed by how easy it is. This project is about producing a hoax, and we're going to make it so that if you press hard on any card it will automatically become a star. This allows you to be able to "guess" correctly even without an Apple Watch around, because any card is the right answer as long as you press correctly.

To accomplish this, we're going to use two new properties of **UITouch**: **force** and **maximumPossibleForce**. The first tells us how strongly the user is pressing for the current touch, and the second tells us the maximum recognizable strength for the current touch. For our purposes, we just need to make sure the two match: if the user is pressing as hard as the screen can recognize, we'll enable our cheat.

The cheat itself is really simple, because we just need to change the image on the front of the card and set its **isCorrect** property to be true.

There is one small problem here, but it's trivial to fix: devices older than the iPhone 6s devices don't support 3D Touch, and even 3D Touch devices can have the feature disabled on user request. So, we need to add a simple check to ensure 3D Touch is available and enabled on our current device.

That's how it all needs to work in theory, but now for the implementation. To keep things as straightforward as possible, we're going to add all this work to **touchesMoved()** in ViewController.swift, which will get called every time the user's finger moves on the screen. Inside this method, we'll find where the user's touch was, then loop through all the cards to find which one (if any) they are over. Then, if they are over a card and are pressing hard enough, we'll enable the cheat.

Add this method to ViewController.swift now:

```swift
override func touchesMoved(touches: Set<UITouch>, withEvent event:
UIEvent?) {
    super.touchesMoved(touches, withEvent: event)

    if let touch = touches.first {
        let location = touch.locationInView(cardContainer)

        for card in allCards {
            if CGRectContainsPoint(card.view.frame, location) {
                if view.traitCollection.forceTouchCapability == .Available {
                    if touch.force == touch.maximumPossibleForce {
                        card.front.image = UIImage(named: "cardStar")
                        card.isCorrect = true
                    }
                }
            }
        }
    }
}
```

That contains three pieces of code that we haven't looked at before. The first two are tiny but important, so I want to cover them briefly before moving on. The first is **locationInView()**, which is the UIKit version of the **locationInNode()** method we've used in SpriteKit several times. The second is **CGRectContainsPoint()**, which returns true if, well, a point is inside a rectangle. I told you it was tiny, but it's definitely important: our point is the location of the current touch, and our rectangle is the frame of each card. So, this function returns true if the user's finger is over a particular card.

The third piece of new code is the check whether 3D Touch is available, although as you can see the check is actually for "force touch" being available – presumably because Apple's marketing department got involved after development had completed! This is done by reading the current trait collection for the view and checking whether its

forceTouchCapability is set to **.Available**.

That's all the code it takes to enable our first cheat, but I'm afraid that you can test it only if you have a 3D Touch-capable device – Xcode's iOS simulator does not support 3D Touch, so either you test with a real device or just take my word for it!

Note: in case you were wondering, that code will indeed run every time the user moves their finger, but like I said earlier "**UIImage** shares image data across image views very efficiently, so there's no extra cost to this approach." The same is true here: this code will run very quickly.

Communicating between iOS and watchOS: WCSession

It's time for something new, and something I've held back from covering in Hacking with Swift because only a small proportion of people have an Apple Watch. So, I'm covering it here only briefly, and only at the very end of the project so that if you don't have an Apple Watch you can just skip on past.

Still here? OK: we're going to upgrade our project so that when your finger moves over the correct card your Apple Watch will gently tap your wrist. The haptic vibration of Apple Watches is so marvellously subtle that no one will have any idea what's happening – the effect is very impressive!

I have good news and bad news. First the good news: for our purposes, communicating between Apple Watch and iOS could not be any easier – it take us maybe five minutes in total to complete the code. Now the bad news: even when the settings are adjusted, your Watch will go to sleep after 70 seconds of inactivity, so it's down you to make sure the app stays awake.

That bad news will make more sense once you're using the finished product, so without further ado let's crack on with development. In ViewController.swift add this new import:

```
import WatchConnectivity
```

As you might imagine, the WatchConnectivity framework is responsible for connectivity between iOS apps and watchOS apps, and we'll be using it to send messages between our phone and a Watch. The messages are dictionaries of any data you want, so you can send strings, numbers, arrays and more – it's up to you.

In order to work with a session, we need to check whether it's supported on our current phone, then activate it. Put this code into the **viewDidLoad()** method of ViewController.swift:

```
if (WCSession.isSupported()) {

  let session = WCSession.defaultSession()
```

```
    session.delegate = self
    session.activateSession()
}
```

You'll get an error because the **ViewController** class doesn't conform to the **WCSessionDelegate** protocol, but that's easily fixed:

```
class ViewController: UIViewController, WCSessionDelegate {
```

Sending a message from a phone to a watch is trivial, like I said, but there is one small piece of complexity: if we want the watch to buzz every time it receives a message (spoiler: that's exactly what we want), we need a way to rate limit those messages. That is, we don't want to send 100 messages a second when the user is touching the right card, because it would make your watch go nuts.

To solve this problem, we're going to add a new property that tracks when the last watch message was sent. This way, we can avoid sending a message to the watch if there was one sent very recently – i.e., less than about half a second ago.

Add this property to the class:

```
var lastMessage: CFAbsoluteTime = 0
```

If you were wondering, **CFAbsoluteTime** is actually just a **Double** behind the scenes. We can get the current time using a function called **CFAbsoluteTimeGetCurrent()**, which returns the number of seconds that have passed since midnight on January 1st 2001. Yes, that's a rather random date, but it doesn't matter: all we care about is the time since our previous call.

Sending a message from the app to the watch is done in two parts. First, we need to check whether the watch is reachable, which in practice means "is our Apple Watch app running and in the foreground?" Second, we need to use the **sendMessage()** method of **WCSession** to send a dictionary of data. It doesn't matter what data we send, because in our app *any*

data will be interpreted as "please buzz."

Keeping in mind the need to rate limit these calls, here's a new method for the **ViewController** class:

```
func sendWatchMessage() {
  let currentTime = CFAbsoluteTimeGetCurrent()


  // if less than half a second has passed, bail out
  if lastMessage + 0.5 > currentTime {

    return

  }


  // send a message to the watch if it's reachable
  if (WCSession.defaultSession().reachable) {

    // this is a meaningless message, but it's enough for our
purposes

    let message = ["Message": "Hello"]

    WCSession.defaultSession().sendMessage(message, replyHandler:
nil, errorHandler: nil)

  }


  // update our rate limiting property
  lastMessage = CFAbsoluteTimeGetCurrent()

}
```

With that new method in place we can call it inside **touchesMoved()** by adding this code near the end of the **CGRectContainsPoint()** condition:

```
if card.isCorrect {

  sendWatchMessage()
```

```
}
```

Just in case you're not sure where I mean, here's how the complete **touchesMoved()**
method should look:

```
override func touchesMoved(touches: Set<UITouch>, withEvent event:
UIEvent?) {
  super.touchesMoved(touches, withEvent: event)

  if let touch = touches.first {
    let location = touch.locationInView(cardContainer)

    for card in allCards {
      if CGRectContainsPoint(card.view.frame, location) {
        if view.traitCollection.forceTouchCapability == .Available {
          if touch.force == touch.maximumPossibleForce {
            card.front.image = UIImage(named: "cardStar")
            card.isCorrect = true
          }
        }

        // here's the new code!
        if card.isCorrect {
          sendWatchMessage()
        }
      }
    }
  }
}
```

And that's it. Yes, that's all the code it takes to send data from our iOS app to an Apple Watch. Of course, the app won't do anything yet because sending data isn't enough: we need to write code to *receive* it and do something interesting.

Designing a simple watchOS app to receive data

So far we have a fully working iOS app that shows cards players can flip over, has a cheat in place for 3D Touch users so you can always guess correctly, and communication happening from iOS to watchOS. The next step is to write a simple watchOS app that is able to receive that data and make the device buzz gently.

We started our project with an Xcode watchOS template, so all this time you will have seen two watchOS folders in your Xcode project: WatchKit App and WatchKit Extension. Yes, cunningly they are two separate things. The *extension* contains all the code that gets run, and the *app* contains the user interface. Both run on the Apple Watch as of watchOS 2.0, but in watchOS 1.0 the extension used to run on your iPhone.

The first thing we're going to do is design a very simple interface using WatchKit, which is the watchOS equivalent of UIKit. This interface is going to contain only a label and a button, telling users to check their phone for instructions. We haven't written those instructions yet, but all in good time…

Look inside the WatchKit App folder for Interface.storyboard, and open that in Interface Builder. Using the Object Library, just like on iOS, drag a label then a button into the small black space of our app's user interface. You will see that WatchKit automatically stacks its views vertically so the interface doesn't get too cluttered.

Select the label, set its Lines property to be 0 so that it spans as many lines as necessary, then align its text center and give it the following content: "Please read the instructions on your phone before continuing." Now select the button and give it the text "I'm Ready". Finally, select both the label and button then change their Vertical alignment property to be Center.

All being well, your WatchKit interface should look like the screenshot below. Don't worry that the views go right to the edge – the Watch's bezel blends seamlessly with the edge of the screen in its apps, so it will look fine on devices.
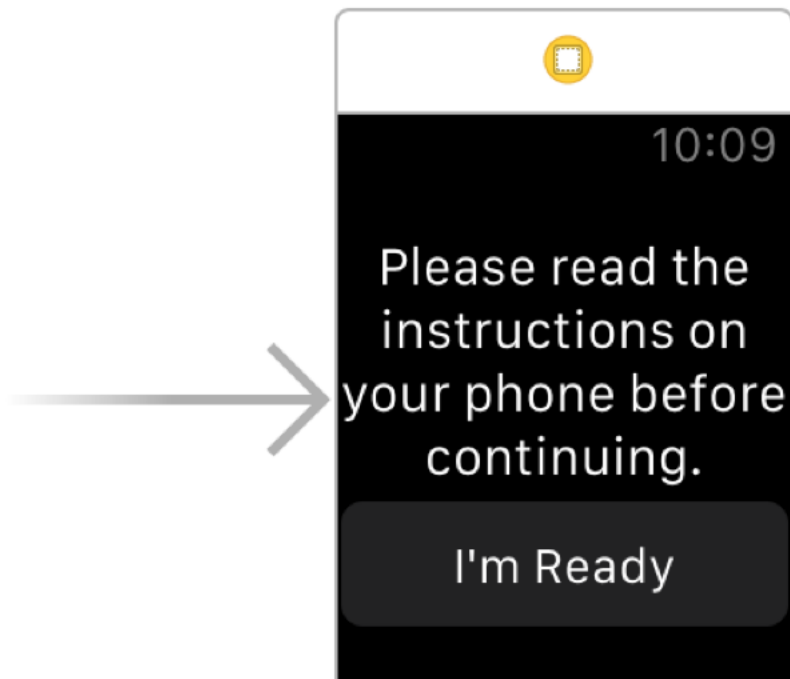
WatchKit views have some similarities with UIKIt, but arguably more differences.

That's it: that's our entire interface. Before we continue with any further coding, we need to create outlets for the label and button by using the Assistant Editor and Ctrl-dragging. Name the label **welcomeText** and the button **hideButton** – you'll notice these have the types **WKInterfaceLabel** and **WKInterfaceButton** because we're in WatchKit now, not UIKIt.

Note: When using Interface Builder with WatchKit, my Xcode's Assistant Editor seems to enjoy showing the wrong files for editing. If this happens to you, click where it says "Automatic" in the center of the Assistant Editor, then choose Top Level Objects > InterfaceController.swift.

Finally, create an action for when the button is tapped, again by Ctrl-dragging in the Assistant Editor. Name this **hideWelcomeText()**. We're done with Interface Builder now, so please go back to the standard editor and open the InterfaceController.swift file from the WatchKit Extension.

The first thing we're going to do is identical to the code from iOS: set ourselves up as the delegate for the **WCSession** and activate it. So, start by adding this import:

```
import WatchConnectivity
```

Now add this to the **willActivate()** method – for our purposes, that's the equivalent of **viewDidLoad()** in the iOS app:

```
if (WCSession.isSupported()) {
    let session = WCSession.defaultSession()
    session.delegate = self
    session.activateSession()
}
```

The code is literally identical to iOS – I told you this was going to be easy!

As before, you'll get an error when you try to assign the delegate to the Watch's view controller, so you'll need to tell iOS you conform to the **WCSessionDelegate** protocol like this:

```
class InterfaceController: WKInterfaceController, WCSessionDelegate {
```

With that, we're almost done with WatchOS. In fact, we just need to do two more things, starting with implementing the **hideWelcomeText()** method. All this needs to do is hide the label and the button we created so that the watch's screen is blank apart from the time in the corner – we don't want any obvious UI in there that might alert people.

Hiding things in WatchKit is almost the same as iOS, so update the **hideWelcomeText()** to this:

```
@IBAction func hideWelcomeText() {
    welcomeText.setHidden(true)
```

```
    hideButton.setHidden(true)

}
```

Note that you need to use **setHidden()** rather than just changing a **hidden** property as you would in UIKit.

The last thing we need to do for our watchOS app is to make the device tap your wrist when it receives a message from iOS. To do this, we just need to implement the **didReceiveMessage** method for the **WCSession** so that it plays a haptic effect.

There are quite a few effects to choose from, but by far the most subtle is **WKHapticType.Click**, which is so subtle that you can't help but marvel at the engineering of the Apple Watch. Add this code just beneath **hideWelcomeText()**:

```
func session(session: WCSession, didReceiveMessage message: [String :
AnyObject]) {

   WKInterfaceDevice().playHaptic(.Click)

}
```

So, whenever the watch receives any message from the phone, it will tap your wrist. Perfect! But… we're not done yet. You see, we need to show some instructions on the iOS app so that everything functions correctly.

You see, not only does the Apple Watch go to sleep extremely quickly, but it also likes making noise to accompany haptic effects, which would rather spoil our hoax! So, to finish up we're going to add an alert to the iOS app reminding you to check your Apple Watch configuration every time it launches.

So, head back to ViewController.swift in your iOS app, then add this new method:

```
override func viewDidAppear(animated: Bool) {

   super.viewDidAppear(animated)
```

```
    let instructions = "Please ensure your Apple Watch is configured
correctly. On your iPhone, launch Apple's 'Watch' configuration app
then choose General > Wake Screen. On that screen, please disable
Wake Screen On Wrist Raise, then select Wake For 70 Seconds. On your
Apple Watch, please swipe up on your watch face and enable Silent
Mode. You're done!"

    let ac = UIAlertController(title: "Adjust your settings", message:
instructions, preferredStyle: .Alert)

    ac.addAction(UIAlertAction(title: "I'm Ready", style: .Default,
handler: nil))

    presentViewController(ac, animated: true, completion: nil)

}
```

That shows instructions to users every time the app runs. Note that you need to put it inside **viewDidAppear()** rather than **viewDidLoad()** because it presents an alert view controller.

Wrap up

The app is finished, but really your work is about to begin: I've given you all the code you need, but it's down to you to provide some meaningful patter to convince your friends! See "Polishing your patter" below for some example patter to get you started, but first give the app a quick try to make sure it all works.

To get started, run the app on your phone, then run it on your Watch. Note that installing apps and launching apps are both quite slow on Apple Watch, so make sure you prepare ahead of time. Once the Watch app is running, remember that it will go to sleep in 70 seconds unless you stop it, and when the Watch sleeps you won't get any haptic taps. The easiest thing to do is very gently rotate the Digital Crown every 30 seconds or so, just to be sure.

So, the app is complete and you've learned all about **CAGradientLayer**, **CAEmitterLayer**, card flip effects, **performSelector()**, 3D Touch and more - good job! If you want to try taking the app further, try implementing the **sessionWatchStateDidChange()** method in ViewController.swift to detect when the Watch goes to sleep – if you make your phone play a brief but innocuous sound, it would alert you to wake your watch.

If you're looking for something more advanced, try adding a hidden button to the Watch user interface that enables "always win mode" – i.e., every card that gets tapped will be the star. Your patter can then be, "I promise it's not a trick, in fact I can even transfer my psychic power to you!" and watch as your friend suddenly finds the star every time.

Polishing your patter

Patter is verbal misdirection: what you say to your friends to confuse them while they are trying to figure out the trick. At the very least, you should explain to your friend how the cards work: there are eight cards with different shapes on, but only one has a green star. Let your friend try finding the card by hand, to prove the set up is real. Lucky people will guess the star correctly; very lucky people will guess twice in a row; but it's almost impossible that someone will get the answer right three times in a row - that's where you come in!

The easiest way to get started with the trick is by putting your finger down over one card, then dragging it slowly over the other cards. Note: don't tap the cards, because that will turn one of them over. Instead, slide your finger over them as if you're feeling for – *ahem* –

psychic vibrations.

When you've done the trick correctly once or twice, your doubting friend will almost certainly think that there's some secret signal on the screen that is alerting you to the correct card, or perhaps you're performing some sort of gesture on the screen that triggers the star. In this situation, up the ante: tell them you can do the trick without even seeing the screen. In fact, you can do it without even *touching* the screen - they can do all the touching for you.

This works in just the same way, except now your friend is the one stroking their finger across the screen. Just wait for your Watch to tap your wrist, then say something like "go back to that card - I felt something there..."

Remember, misdirection is key. So, don't tap the right answer as soon as your phone vibrates. Say something like "hmm... this card feels really warm... let me try some others first." Or go to one of the cards nearby and say "this card feels warm, but not as warm as the previous one..."

Finally, once you've fooled everyone and had your fun, let them in on the joke - after all, if you can't have fun, why bother?

Project 38
GitHub Commits

Get started with Core Data by building an app to fetch and store GitHub commits for Swift.

Setting up

In this project you'll learn how to use Core Data while building an app that fetches GitHub commit data for the Swift project. Core Data is Apple's object graph and persistence framework, which is a fancy way of saying it reads, writes and queries collections of related objects while also being able to save them to disk.

Core Data is undoubtedly useful, which is why about 500,000 apps in the App Store build on top of it. But it's also rather complicated to learn, which is why I left it so late in this tutorial series – despite my repeated attempts to simplify the topic, this tutorial is still going to be the hardest one in the whole of Hacking with Swift, and indeed I might have missed it out altogether were it not by far the most requested topic from readers!

So, strap in, because you're going to learn a lot: we'll be covering Core Data, which will encompass **NSFetchRequest**, **NSManagedObject**, **NSPredicate**, **NSSortDescriptor**, and **NSFetchedResultsController**. We'll also touch on **NSDateFormatter** for the first time, which is Apple's way of converting **NSDate** objects into human-readable formats.

As always, I like to teach new topics while giving you a real-world project to work with, and in our case we're going to be using the GitHub API to fetch information about Apple's open-source Swift project. The GitHub API is simple, fast, and outputs JSON, but most importantly it's public. Be warned, though: you get to make only 60 requests an hour without an API key, so while you're testing your app make sure you don't refresh too often!

If you weren't sure, a "Git commit" is a set of changes a developer made to source code that is stored in a source control repository. For example, if you spot a bug in the Swift compiler and contribute your changes, those changes will form one commit. And before I get emails from random internet pedants, *yes I know* things are more complicated than that, but it's more than enough of an explanation for the purposes of this tutorial.

Before we start, it's important I reiterate that Core Data can be a bit overwhelming at first. It has a lot of unique terminology, and it needs a lot of boilerplate just to get you up and running. If you find yourself struggling to understand it all, that's perfectly normal – it's not you, it's just Core Data.

Over the next four chapters, we will implement the four pieces of Core Data boilerplate. We're going to use Xcode's built-in Master-Detail Application template, but we're not *not* going to have it generate Core Data code for us – that particular template is confusing enough as it is, and the Core Data variety is practically impenetrable.

So, please go ahead and create a new Master-Detail Application project named Project38. Select Swift for your language, select Universal for your device type, then make sure you *uncheck* Core Data otherwise the rest of this tutorial will be very confusing indeed.

Do not, under any circumstances, enable Core Data for the Master-Detail Application template for this tutorial.

We need to parse the JSON coming from GitHub's API, and the easiest way to do that is with SwiftyJSON. If you haven't already downloaded the content for this project, **please get it from GitHub now**. You'll see SwiftyJSON is there, so please drag that into your project now.

There are just two more things to do before our project is set up. First, open Main.storyboard in Interface Builder and find the table view controller on the top right of the storyboard. Select its only prototype cell, then open the attributes inspector and change Style from Basic to Subtitle.
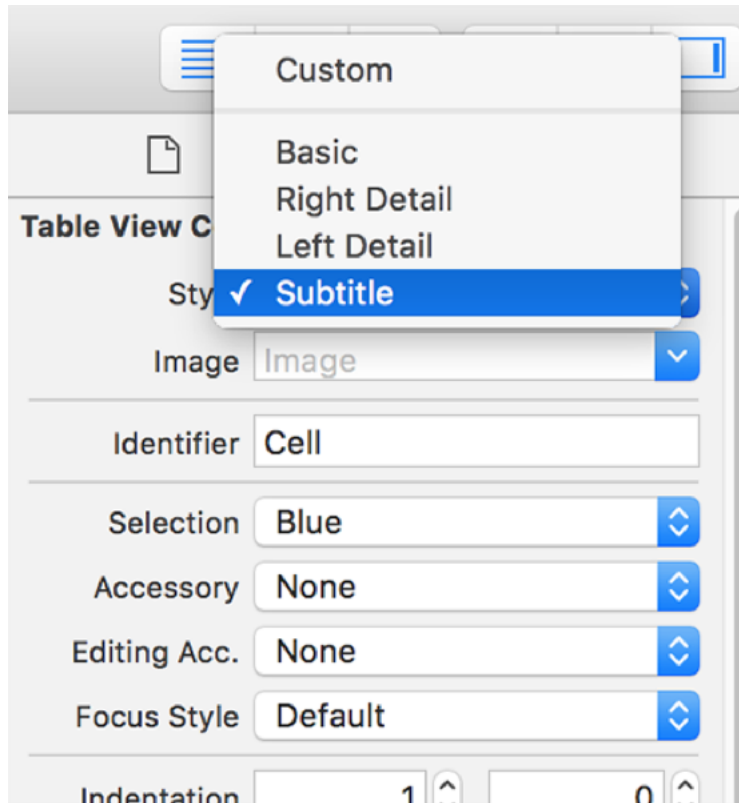
Change the table view cell style to be Subtitle.

Now scroll directly down to find the detail view controller with its single label. Select that label, and change its number of lines property to be 0 - it's not perfect, but it's enough to show everything is working.

Finally, open MasterViewController.swift for editing. Please find and delete the entire **insertNewObject()** method. Now scroll up to the **viewDidLoad()** method and delete these two lines of code:

```
let addButton = UIBarButtonItem(barButtonSystemItem: .Add, target:
self, action: "insertNewObject:")

self.navigationItem.rightBarButtonItem = addButton
```

That's it: the project is cleaned up and ready for Core Data. Remember, there are four steps to implementing Core Data in your app, so let's start with the very first step: designing a Core Data model.

Designing a Core Data model

A data model is a description of the data you want Core Data to store, and is a bit like creating a class in Swift: you define entities (like classes) and give them attributes (like properties). But Core Data takes it a step further by allowing you to describe how its entities relate to other entities, as well as adding rules for validation and uniqueness.

We're going to create a data model for our app that will store a list of all the GitHub commits for the Swift library. Take a look at the raw GitHub JSON now by loading this URL in a web browser: **https://api.github.com/repos/apple/swift/commits?per_page=100**. You'll see that each commit has a "sha" identifier, committer details, a message describing what changed, and a lot more. In our initial data model, we're going to track the "date", "message", "sha", and "url" fields, but you're welcome to add more if you want to.

To create a data model, choose File > New > File and select iOS > Core Data > Data Model. Name it Project38, then make sure the "Group" option near the bottom of the screen has a yellow folder to it rather than a blue project icon.

This will create a new file called Project38.xcdatamodeld, and when you select that you'll see a new editing display: the Data Model Editor. At the bottom you'll see a button with the title "Add Entity": please click that now.
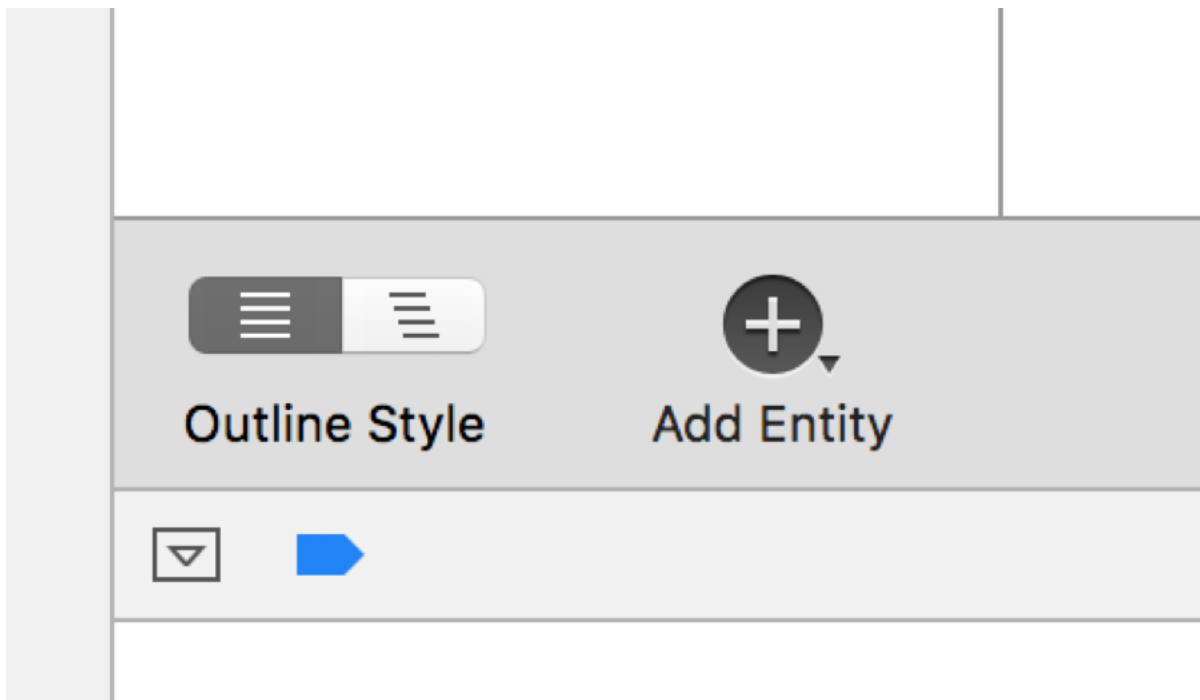
A Core Data "entity" is like a Swift class in that it is just a description of what an object is going to look like. By default, new entities are called "Entity", but you can change that in the Data Model inspector in the right-hand pane of Xcode – press Alt+Cmd+3 if it's not already visible. With your new entity selected, you should see a field named "Name", so please change "Entity" to be "Commit".
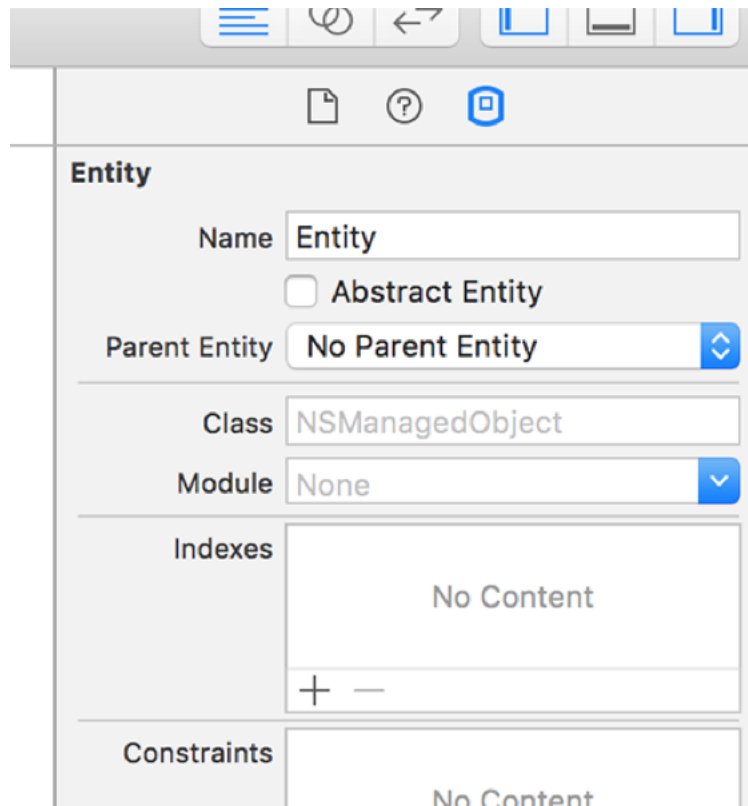
New entities have the default name Entity, but obviously it's best that you change it to something more meaningful.

To the right of the Add Entity button is another button, Add Attribute. Click that four times now to add four attributes, then name them "date", "message", "sha" and "url". These attributes are just like properties on a Swift class, including the need to have a data type. You'll see they each have "Undefined" for their type right now, but that's easily changed: set them all to have the String data type, except for "date", which should be Date.

The final change we're going to make is to mark each of these four property as non-optional. Click "date" then hold down Shift and click "url" to select all four attributes, then look in the

Data Model inspector for the Optional checkbox and deselect it. **Note:** the Data Model inspector can be a bit buggy sometimes – if you find it's completely blank, you might need to try selecting one of the other files in your project and/or deselecting then re-selecting your entity to make things work.
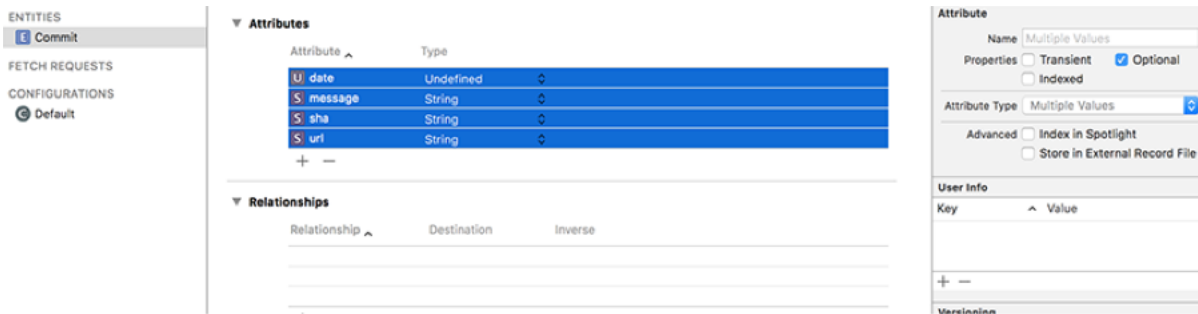
Select all four attributes and disable the Optional checkbox in the Data Model inspector.

Now, you might be forgiven for thinking, "at last! All that time spent mastering Swift optionals is paying off – I know what this checkbox does!" But I have some bad news for you. Or, more specifically, Core Data has some bad news for you: this Optional checkbox has nothing at all to do with Swift optionals, it just determines whether the objects that Core Data stores are required to have a value or not.

That's the first step of Core Data completed: the app now knows what kind of data we want to store. We'll be coming back to add to our model later, but first it's time for step two: adding the base Core Data functionality to our app so we can load the model we just defined and save any changes we make.

Adding Core Data to our project: NSPersistentStoreCoordinator

A Core Data model defines what your data should look like, but it doesn't actually store it anywhere. To make our app work, we need to create a persistent store where Core Data can read and write its data, then create what's called a "managed object context" where those objects can be manipulated by us in memory.

In this second step we're going to write code to load the model we just defined, load a peristent store where saved objects can be stored, and also create a managed object context where our objects will live while they are active. Any changes we make to objects won't be saved until we explicitly request it, and it's significantly faster to manipulate objects inside your managed object context as much as you need to before saving rather than saving after every change.

When data is saved, it's nearly always written out to an SQLite database. There are other options, but take my word for it: almost everyone uses SQLite. SQLite is a very small, very fast, and very portable database engine, and what Core Data does is provide a wrapper around it: when you read, write and query a managed object context, Core Data translates that into Structured Query Language (SQL) for SQLite to parse.

If you were wondering, SQL is pronounced Ess Cue Ell, but many people pronounce it "sequel." The pronunciation of SQLite is more varied, but when I met its author I asked him how *he* pronounces it, so I feel fairly safe that the definitive answer is this: you pronounce SQLite as Ess-Cue-Ell-ite, as if it were a mineral like Kryponite or Carbonite depending on your preferred movie.

To get started, open MasterViewController.swift and add an import for Core Data:

```
import CoreData
```

To set up the basic Core Data system, we're going to create a new method called **startCoreData()** that will do the following:

1. Load our data model we just created from the application bundle and create a **NSManagedObjectModel** object from it.

2. Create an **NSPersistentStoreCoordinator** object, which is responsible for reading from and writing to disk.

3. Set up an **NSURL** pointing to the database on disk where our actual saved objects live. This will be an SQLite database named Project38.sqlite.

4. Load that database into the **NSPersistentStoreCoordinator** so it knows where we want it to save. If it doesn't exist, it will be created automatically

5. Create an **NSManagedObjectContext** and point it at the persistent store coordinator.

Below is all that turned into Swift, along with comments marking the relevant numbers above - please add this new method to the **MasterViewController** class:

```swift
func startCoreData() {
  // 1

  let modelURL = NSBundle.mainBundle().URLForResource("Project38",
withExtension: "momd")!

  let managedObjectModel = NSManagedObjectModel(contentsOfURL:
modelURL)!


  // 2

  let coordinator = NSPersistentStoreCoordinator(managedObjectModel:
managedObjectModel)


  // 3

  let url =
getDocumentsDirectory().URLByAppendingPathComponent("Project38.sqlite
")


  do {
    // 4

    try coordinator.addPersistentStoreWithType(NSSQLiteStoreType,
configuration: nil, URL: url, options: nil)


    // 5

    managedObjectContext =
```

```
NSManagedObjectContext(concurrencyType: .MainQueueConcurrencyType)

    managedObjectContext.persistentStoreCoordinator = coordinator

  } catch {

    print("Failed to initialize the application's saved data")

    return

  }

}


// this is our usual helper function to find the user's documents
directory

func getDocumentsDirectory() -> NSURL {

  let urls =
NSFileManager.defaultManager().URLsForDirectory(.DocumentDirectory,
inDomains: .UserDomainMask)

  return urls[0]

}
```

Note that I've wrapped the call to **addPersistentStoreWithType()** inside a do/catch block because it throws an exception when it fails. My project prints a message to the Xcode console, but in production you should of course show some meaningful error to your user.

To make that code work, you'll need to add a property for **managedObjectContext**, which is where we store the context once it is loaded. Add this property to the class now:

```
var managedObjectContext: NSManagedObjectContext!
```

So: the model is in your application bundle because you define it while creating your app, the persistent store is in your application's documents directory because it gets modified as you use it, and the managed object context lives in memory, and is where you manipulate your objects before saving them.

We need to call our new method as soon as our app launches, so add this just before the end of **viewDidLoad()**:

```
startCoreData()
```

The last thing to do is add a way to save the managed object context to the persistent store, i.e. to make sure the changes we've made in memory are written to disk so they can be used later. To make this work we'll use the **hasChanges** property of the managed object context to determine whether a save is needed, and its **save()** method to do the actual writing.

Add this new method just after the previous two:

```
func saveContext() {

  if managedObjectContext.hasChanges {

    do {

      try managedObjectContext.save()

    } catch {

      print("An error occurred while saving: \(error)")

    }

  }

}
```

We'll be calling that whenever we've made changes that should be saved to disk.

At this point, our app has a working data model as well as code to load that model and prepare a persistent store for reading and writing. That means step two is done and we're on to step three: creating objects inside Core Data and fetching data from GitHub.

Creating an NSManagedObject subclass with Xcode

In our app, Core Data is responsible for reading data from a persistent store (the SQLite database) and making it available for us to use as objects. After changing those objects, we can save them back to the persistent store, which is when Core Data converts them back from objects to database records.

All this is done using a special data type called **NSManagedObject**. This is a Core Data subclass of **NSObject** that uses a unique keyword, **@NSManaged**, to provide lots of functionality for you. For example, you already saw the **hasChanges** property of a managed object context – that automatically gets set to true when you make changes to your objects, because Core Data tracks when you change properties that are marked **@NSManaged**.

Behind the scenes, **@NSManaged** effectively means "extra code will automatically be provided when the program runs." It's a bit like functionality injection: when you say "this property is **@NSManaged**" then Core Data will add getters and setters to it when the app runs so that it handles things like change tracking.

If this sounds complicated, relax: Xcode can do quite a bit of work for us. It's not perfect, as you'll see shortly, but it's certainly a head start. So, it's time for step three: creating objects in Core Data so that we can fetch and store data from GitHub.

Select the data model (Project38.xcdatamodeld) and choose Editor > Create NSManagedObject Subclass from the menu. Make sure your data model is selected then click Next. Make sure the Commit entity is checked then click Next again.
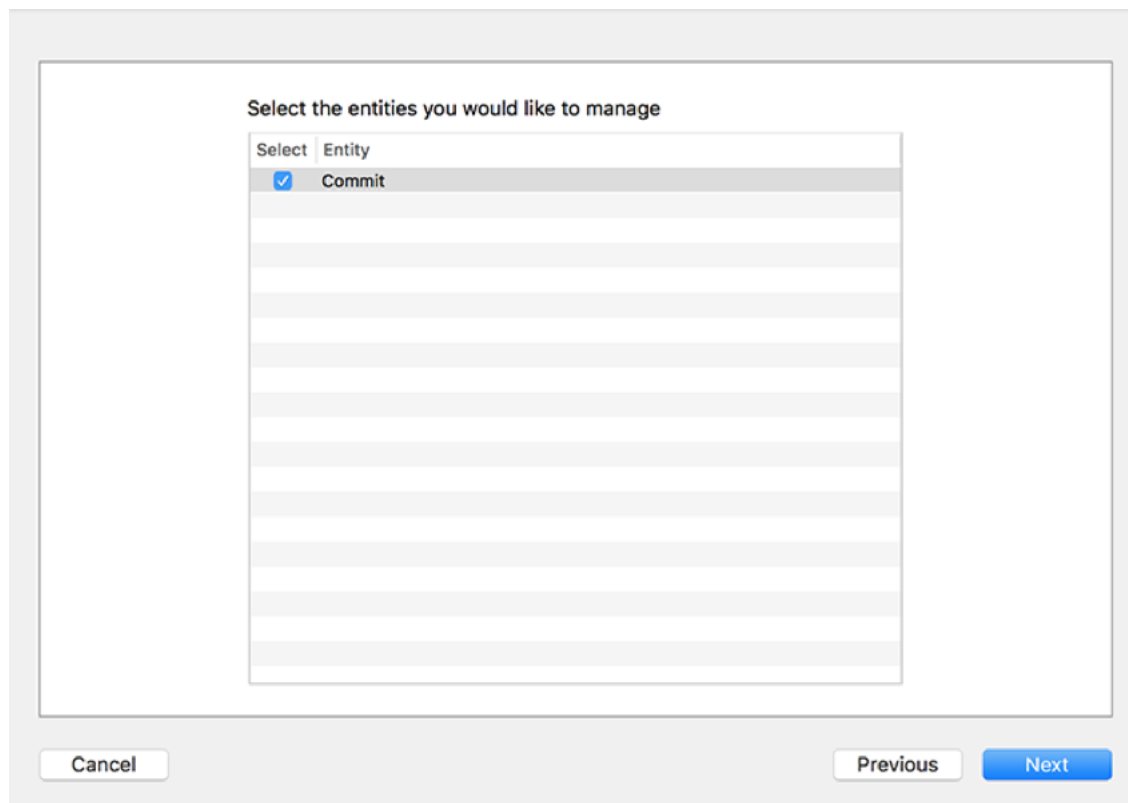
Xcode will ask you which entities it should turn into subclasses. Right now we have only one, Commit, so please choose it.

Finally, look next to Group and make sure you see a yellow folder next to "Project 38" rather than a blue project icon, and click Create.
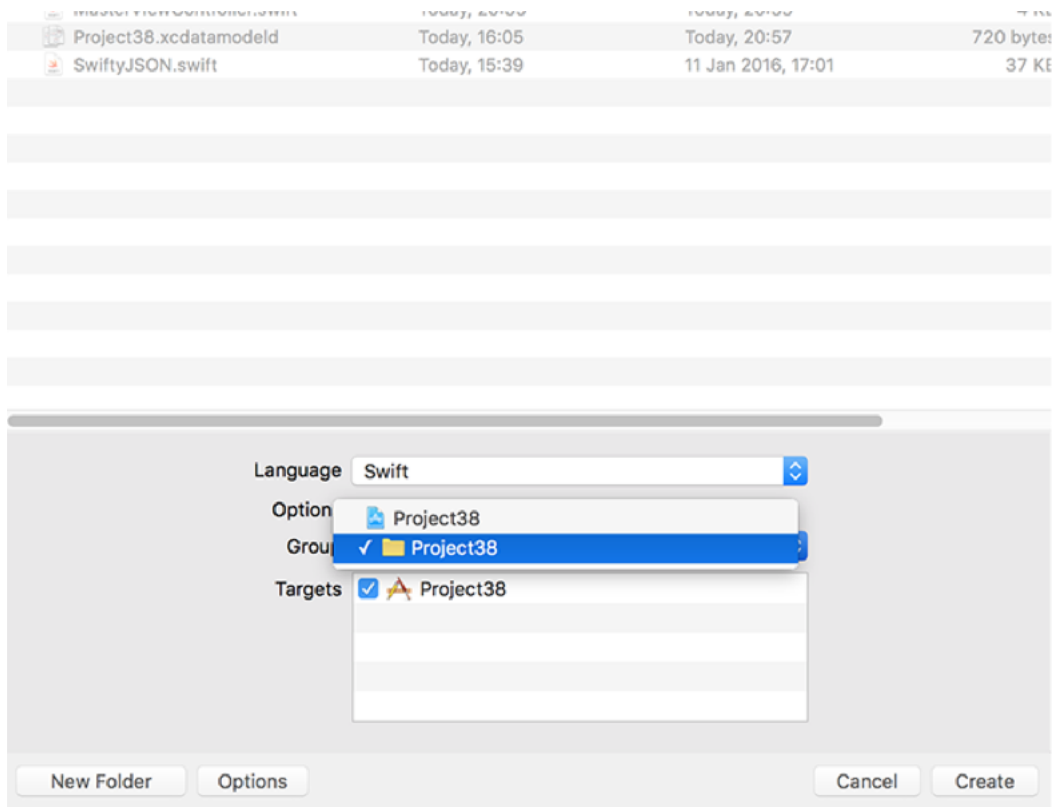
Xcode dearly loves trying to make you create Core Data models in the wrong place, regardless of how often you change this option. Make sure you select the yellow icon not the blue one!

When this process completes, two new files are created: Commit.swift and Commit +CoreDataProperties.swift. If you examine them you'll see the first one is empty, whereas the second one looks something like this:

```
import Foundation
import CoreData


extension Commit {

    @NSManaged var date: NSDate?

    @NSManaged var message: String?

    @NSManaged var sha: String?

    @NSManaged var url: String?

}
```

First, there's that **@NSManaged** keyword I mentioned to you. Second, notice that the code says **extension Commit** rather than **class Commit**, which is Xcode being clever: Commit.swift is an empty class that you can fill with your own functionality, and Commit +CoreDataProperties.swift is an *extension* to that class where Core Data writes *its* properties. This means if you ever add attributes to the Commit entity and regenerate the **NSManagedObject** subclass, Xcode will overwrite only Commit+CoreDataProperties.swift, leaving your own changes in Commit.swift untouched.

Like I said, this Core Data code generation is a head start, but not perfect: how come all those properties are marked as optional, i.e. **NSDate?** rather than **NSDate**, when we explicitly told Core Data that they were not optional when making the model? Well, remember that Swift's optionals and Core Data's optionals aren't the same, so Xcode's generated code isn't perfect.

Working with optionals when they aren't needed adds an extra layer of annoyance, so I want you go ahead and remove all the question marks from Commit+CoreDataProperties.swift. When you're finished, it should look like this:

```
import Foundation
import CoreData


extension Commit {
  @NSManaged var date: NSDate
  @NSManaged var message: String
  @NSManaged var sha: String
  @NSManaged var url: String
}
```

Warning: if you recreate the subclass using the Create NSManaged Subclass menu option, these changes will be lost and you will need to remove the optionality again. We'll be doing exactly this later on.

Now that we have Core Data objects defined, we can start to write our very first useful Core Data code: we can fetch some data from GitHub and convert it into our **Commit** objects. To make things easier to follow, I want to split this up into smaller steps: fetching the JSON, and converting the JSON into Core Data objects.

First, fetching the JSON. This needs to be a background operation because network requests are slow and we don't want the user interface to freeze up when data is loading This operation needs to go to the GitHub URL, **https://api.github.com/repos/apple/swift/commits?per_page=100** and convert the result into a SwiftyJSON object ready for conversion.

To push all this into the background, we're going to use **performSelectorInBackground()** to call **fetchCommits()** – a method we haven't written yet. Put this just before the end of **viewDidLoad()**:

```
performSelectorInBackground("fetchCommits", withObject: nil)
```

What the new **fetchCommits()** method will do is very similar to what we did back in project 7: download the URL into an **NSData** object then pass it to SwiftyJSON to convert into an array of objects. In project 10 we extended this to use **dispatch_async()** so that once the JSON was ready to be used we did the important work on the main thread.

We're not going to process the JSON just yet, but we can do everything else: download the data, create a SwiftyJSON object from it, then go back to the main thread to loop over the array of GitHub commits and save the managed object context when we're done. To make things easier to debug, I've added a **print()** statement so you can see how many commits were received from GitHub each time.

Here's our first draft of the **fetchCommits()** method:

```
func fetchCommits() {
  if let data = NSData(contentsOfURL: NSURL(string: "https://
api.github.com/repos/apple/swift/commits?per_page=100")!) {
    let jsonCommits = JSON(data: data)
    let jsonCommitArray = jsonCommits.arrayValue
```

```
    print("Received \(jsonCommitArray.count) new commits.")

    dispatch_async(dispatch_get_main_queue()) { [unowned self] in
      for jsonCommit in jsonCommitArray {
        // more code to go here!
      }

      self.saveContext()
    }
  }
}
```

There's nothing too surprising there – in fact right now it won't even do anything, because **saveContext()** will detect no Core Data changes have happened, so the **save()** call won't happen.

The second of our smaller steps is to replace **// more code to go here!** with, well, *actual code.* Here's the revised version, with a few extra lines either side so you can see where it should go:

```
dispatch_async(dispatch_get_main_queue()) { [unowned self] in
  for jsonCommit in jsonCommitArray {
    // the following three lines are new
    if let commit =
NSEntityDescription.insertNewObjectForEntityForName("Commit",
inManagedObjectContext: self.managedObjectContext) as? Commit {
      self.configureCommit(commit, usingJSON: jsonCommit)
    }
  }

  self.saveContext()
```

```
}
```

So, there are three new lines of code, of which one is just a closing brace by itself. Of course, it's so short only because I've cheated a bit by calling another method that we haven't written yet, **configureCommit()**, so to make your code build add this for now:

```
func configureCommit(commit: Commit, usingJSON json: JSON) {

}
```

Now let's take a look at the two new lines of code above. First is **NSEntityDescription.insertNewObjectForEntityForName()**, which reads the entity description for Commit from the managed object context. This method creates a new object with the type **NSManagedObject**, so we use if/let and **as? Commit** to safely downcast it to be a **Commit** object. Putting all that together, the code reads "try to create a new **Commit** object in our managed object context, and give it back to me if things went OK."

If we get a valid **Commit** object back, we pass it onto the **configureCommit()** method, along with the JSON data for the matching commit. That **Commit** object is our **NSManagedObject** subclass, so it has all sorts of magic behind the scenes, but to our Swift code is just a normal object with properties we can read and write. This would make the **configureCommit()** method straightforward if it were not for dates.

Yes, dates. Not the sweet fruity kind, but the **NSDate** kind. Make sure you have the GitHub API URL open in a web browser window so you can see exactly what it returns, and you'll notice that dates are sent back like "2016-01-26T19:46:18Z". That format is known as ISO-8601 format, we need to parse that into an **NSDate** in order to put it inside our **Commit** object.

To convert "2016-01-26T19:46:18Z" into an **NSDate** we're going to use a new class called **NSDateFormatter**. This is designed to convert text to **NSDate** and vice versa – all you need to do is describe the textual date format using special syntax For ISO-8601 dates, that syntax is "yyyy-MM-dd'T'HH:mm:ss'Z'": a four-digit year, a two-digit month, a two-digit day, the letter T, then hours, minutes, seconds, and the letter Z.

We're going to use this date format again later in the tutorial, so add this property to the

MasterViewController class:

```
let dateFormatISO8601 = "yyyy-MM-dd'T'HH:mm:ss'Z'"
```

Before I show you the new **configureCommit()** method, there's one more thing you need to know: getting an **NSDate** out of a string might fail, for example if the date format doesn't match the one we set. In this case, we'll get **nil** back, which isn't much good for our app, so I'm going to use the nil coalescing operator to use a new **NSDate** instance if the date failed to parse.

Here's the new **configureCommit()** method:

```
func configureCommit(commit: Commit, usingJSON json: JSON) {

  commit.sha = json["sha"].stringValue

  commit.message = json["commit"]["message"].stringValue

  commit.url = json["html_url"].stringValue


  let formatter = NSDateFormatter()

  formatter.timeZone = NSTimeZone(name: "UTC")

  formatter.dateFormat = dateFormatISO8601

  commit.date = formatter.dateFromString(json["commit"]["committer"]
["date"].stringValue) ?? NSDate()

}
```

I love how easy SwiftyJSON makes JSON parsing! If you've forgotten, it automatically ensures a safe value gets returned even if the data is missing or broken. For example, **json["commit"]["message"].stringValue** will either return the commit message as a string or an empty string, regardless of what the JSON contains. So if "commit" or "message" don't exist, or if they do exist but actually contains an integer for some reason, we'll get back an empty string – it makes JSON parsing extremely safe while being easy to read and write.

That completes step three of our Core Data code: we now create lots of objects when we

download data from GitHub, and the finishing collection gets saved back to SQLite. That just leaves one final step before we have the full complement of fundamental Core Data code: we need to be able to load and use all those **Commit** objects we just saved!

Loading Core Data objects using NSFetchRequest and NSSortDescriptor

This is where Core Data starts to become interesting and perhaps – gasp! – even fun. Yes, I know it's taken quite a lot of work to get this far, but I *did* warn you, remember?

Step four is where we finally get to put to use all three previous steps by showing data to users. After the huge amount of work you've put in, particularly in the previous step, I'm sure you'll be grateful to see everything pay off at last!

Xcode's Master-Detail Application template gave us this array to use in the table view:

```
var objects = [AnyObject]()
```

In our project, we know we're using **Commit** objects rather than **AnyObject**, so please change that line to this:

```
var objects = [Commit]()
```

As soon as you do that, your code will break in two places because the template expects **NSDate** objects to be in that array. Don't worry, this is easily fixed. Start by going to **prepareForSegue()** and finding this line:

```
let object = objects[indexPath.row] as! NSDate
```

To fix that line, just remove the **as! NSDate** part, like this:

```
let object = objects[indexPath.row]
```

Now scroll down to find **cellForRowAtIndexPath**, and in particular these two lines:

```
let object = objects[indexPath.row] as! NSDate
cell.textLabel!.text = object.description
```

Change them to this:

```
let object = objects[indexPath.row]
cell.textLabel!.text = object.message
cell.detailTextLabel!.text = object.date.description
```

That will show a snippet of each commit message in the table view cell titles, and a basic formatted date in the subtitles.

With that change made, we need to write one new method in order to make our entire app spring into life. But before we jump into the code, you need to learn about one of the most important classes in Core Data: **NSFetchRequest**. This is the class that performs a query on your data, and returns a list of objects that match.

We're going to use **NSFetchRequest** in a really basic form for now, then add more functionality later. In this first version, we're going to ask it to give us an array of all **Commit** objects that we have created, sorted by date descending so that the newest commits come first.

The way fetch requests work is very simple: you tell it what entity you want to query, give it an array of data describing how you want the sorting to happen, then pass it to **executeFetchRequest()** on your managed object context. If the fetch request worked then you'll get back an array of objects matching the query; if not, an exception will be thrown that you need to catch.

The sorting is done through a special data type called **NSSortDescriptor**, which is a trivial wrapper around the name of what you want to sort (in our case "date"), then a boolean setting whether the sort should be ascending (oldest first for dates) or descending (newest first). You pass an array of these, so you can say "sort by date descending, then by message

ascending," for example.

OK, time for some code, and I hope you'll be pleasantly surprised by how easy it is:

```
func loadSavedData() {
  let fetch = NSFetchRequest(entityName: "Commit")
  let sort = NSSortDescriptor(key: "date", ascending: false)
  fetch.sortDescriptors = [sort]


  do {
    if let commits = try
managedObjectContext.executeFetchRequest(fetch) as? [Commit] {
      print("Got \(commits.count) commits")
      objects = commits
      tableView.reloadData()
    }
  } catch {
    print("Fetch failed")
  }
}
```

So, that creates the **NSFetchRequest**, gives it a sort descriptor to arrange the newest commits first, then uses the **executeFetchRequest()** method to fetch the actual objects. That method returns an array of **NSManagedObjects**, but the code safely downcasts that to be **Commit**. Once that's done, it's just a matter of assigning the resulting array to the **objects** property used in the Xcode template and calling **reloadData()** on the table – everything else is handled for us by the Xcode template.

To make the app work, we need to call this new **loadSavedData()** method in two places. First, add a call just after **startCoreData()** in the **viewDidLoad()** method. Second, add a call to the **fechCommits()** method, just after where we have **self.saveContext()**. You will, of course, need to use **self.fetchCommits()** in that instance.

Again, I've written a simple **print()** statement when errors occur, but in your own production apps you will need to show something useful to your user.

Good news: that completes all four basic bootstrapping steps for Core Data. We have defined our model, loaded the data store and managed object context, fetched some example data and saved it, and loaded the resulting objects. You should now be able to run your project and see it all working!
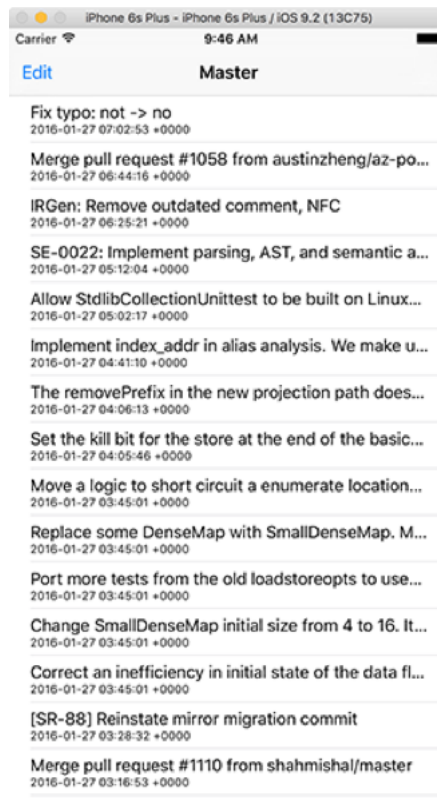
Finally our app is working: we have a scrolling table view listing commits to the Swift repository on GitHub.

Getting a crash? If your code crashes in the **cellForRowAtIndexPath** method, make sure you changed the table view's cell type to be Subtitle rather than Basic.

How to make a Core Data attribute unique using constraints

After such a huge amount of work getting Core Data up and running, you'll probably run your app a few times to enjoy it all working. But it's not perfect, I'm afraid: first, you'll see GitHub commits get duplicated each time the app runs, and second you'll notice that tapping on a commit shows the Detail screen with not much helpful information.

We'll be fixing that second problem later, but for now let's focus on the first problem: duplicate GitHub commits. In fact, you probably have triplicate or quadruplicate by now, because each time you run the app the same commits are fetched and added to Core Data, so you end up with the same data being repeated time and time again.

No one wants repeated data, so we're going to fix this problem. And for once I'm pleased to say that Core Data makes this very easy thanks to a simple technology called "unique constraints." All we need to do is find some data that is guaranteed to uniquely identify a commit, tell Core Data that is a unique identifier, and it will make sure objects with that same value don't get repeated.

Even better, Core Data can intelligently merge updates to objects in situations where this is possible. It's not going to happen with us, but imagine a situation where a commit author could retrospectively change their commit message from "I fxed a bug with Swift" to "I fixed a bug with Swift." As long as the unique identifier didn't change, Core Data could recognize this was an update on the original commit, and merge the change intelligently.

Inserting data uniquely used to be harder before iOS 9, because you needed to run an **NSFetchRequest** to see if an object matching your identifier existed, then create it or update it as needed. As of iOS 9, this is done automatically: you just need to tell Core Data what your unique attribute is.

In this app, we have the perfect unique attribute just waiting to be used: every commit has a "sha" attribute that is a long string of letters and numbers that identify that commit uniquely. SHA stands for "secure hash algorithm", and it's used in many places to generate unique identifiers from content.

A "hash" is a little bit like like one-way, truncated encryption: one piece of input like "Hello world" will always generate the same hash, but if you change it to be "Hello World" – just capitalising a single letter – you get a completely different hash. It's "truncated" because no

matter how much content you give it as input, the "sha" will always be 40 letters. It's "one way" because you can't somehow reverse the hash to discover the original content, which is where hashes are different to encryption: an encrypted message can be decrypted to its original content, whereas a hashed message cannot be "dehashed" back to its original.

Hashes are frequently used as a checksum to verify that a file or data is correct: if you download a 10GB file and want to be sure it's exactly what the sender created, you can just compare your hash with theirs. Because hashes are truncated to a specific size, it is technically possible for two pieces of very different content to generate the same hash, known as a "collision", but this is extremely rare.

Enough theory. Please go ahead and run your app a few times to make sure there are a good number of duplicates so you can see the problem in action. We added some **print()** statements in there for debugging purposes, so you'll see a message like this:

```
Got 500 commits
Received 100 new commits.
Got 600 commits
```

Select the data model (Project38.xcdatamodeld) and make sure the Commit entity is selected rather than one of its attributes. If you look in the Data Model inspector you'll see a field marked "Constraints" – click the + button at the bottom of that field. A new row will appear saying "comma,separated,properties". Click on that, hit Enter to make it editable, then type "sha" and hit Enter again to save the changes.
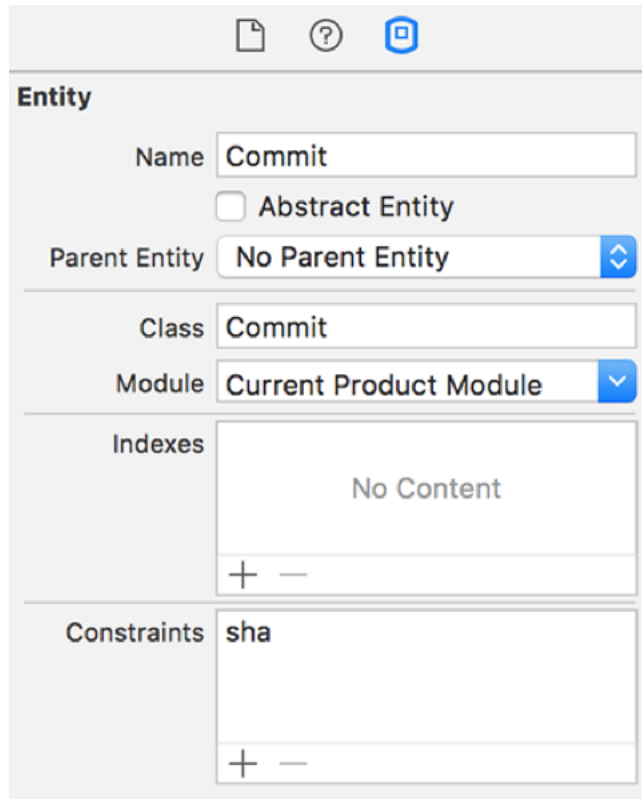
Just adding the word 'sha' to the list of constraints is enough to tell Core Data we need it to be unique.

Now for the important part, because in my testing Xcode didn't behave quite as it ought to. First, go to the the iOS simulator, then choose the Simulator menu and choose Reset Content And Settings. Now quit Xcode, and relaunch it. Re-open your project, hold down the Alt key, and go to Product > Clean Build Folder from the menu and click Clean when prompted.

What you just did was completely reset the state of your project and the iOS Simulator. The reason this is required is two-fold: first, because you just made an important change to your model, which is generally a bad idea unless you know what you're doing; second, because Xcode frequently fails to register changes to unique constraints unless you explicitly save, so by asking you to restart Xcode and clean your project I can feel fairly confident that everyone will be in a working state.

Before you run your project again, I want you to make one tiny code change. In your **startCoreData()** method, add this line to where the managed object context is configured:

```
managedObjectContext.mergePolicy =
NSMergeByPropertyObjectTrumpMergePolicy
```

It should look like this:

```
managedObjectContext =
NSManagedObjectContext(concurrencyType: .MainQueueConcurrencyType)

managedObjectContext.persistentStoreCoordinator = coordinator

managedObjectContext.mergePolicy =
NSMergeByPropertyObjectTrumpMergePolicy
```

This instructs Core Data to allow updates to objects: if an object exists in its data store with message A, and an object with the same unique constraint ("sha" attribute) exists in memory with message B, the in-memory version "trumps" (overwrites) the data store version.

Go ahead and run your project a few times now and you'll see this message in the Xcode log:

```
Got 100 commits

Received 100 new commits.

Got 100 commits
```

As you can see, 100 commits were loaded from the persistent store, 100 "new" commits were pulled in from GitHub, and after Core Data resolved unique attributes there were still only 100 commits in the persistent store. Perfect! If you run your project again after a few hours, the numbers will start to go up slowly as new commits appear on GitHub – Swift is a live project, after all!

Note: in a couple of chapters I'll be introducing you to something called **NSFetchedResultsController**. Using attribute constraints can cause problems with **NSFetchedResultsController**, but in this tutorial we're always doing a full save and load of our objects because it's an easy way to avoid problems later. Don't worry about it for now – I'll mention it again at the appropriate time.

Examples of using NSPredicate to filter NSFetchRequest

Predicates are one of the most powerful features of Core Data, but they are actually useful in lots of other places too so if you master them here you'll learn a whole new skill that can be used elsewhere. For example, if you already completed project 33 you'll have seen how predicates let us find iCloud objects by reference.

Put simply, a predicate is a filter: you specify the criteria you want to match, and Core Data will ensure that only matching objects get returned. The best way to learn about predicates is by example, so I've created three examples below that demonstrate various different filters. We'll be adding a fourth one in the next chapter once you've learned a bit more.

First, add this new property to the **MasterViewController** class:

```
var commitPredicate: NSPredicate?
```

I've made that an optional **NSPredicate** because that's exactly what our fetch request takes: either a valid predicate that specifies a filter, or **nil** to mean "no filter."

Find your **loadSavedData()** method and add this line just below where the **sortDescriptors** property is set:

```
fetch.predicate = commitPredicate
```

With that property in place, all we need to do is set it to whatever predicate we want before calling **loadSavedData()** again to refresh the list of objects. The easiest way to do this is by adding a new method called **changeFilter()**, which we'll use to show an action sheet for the user to choose from. First we need to add a button to the navigation bar that will call this method, so put this code into **viewDidLoad()**:

```
navigationItem.rightBarButtonItem = UIBarButtonItem(title: "Filter",
style: .Plain, target: self, action: "changeFilter")
```

And here's an initial version of that new method for you to add to your view controller:

```
func changeFilter() {
  let ac = UIAlertController(title: "Filter commits…", message: nil,
preferredStyle: .ActionSheet)


  // 1

  // 2

  // 3

  // 4



  ac.addAction(UIAlertAction(title: "Cancel", style: .Cancel,
handler: nil))

  presentViewController(ac, animated: true, completion: nil)

}
```

We'll be replacing the four comments one by one as you learn about predicates.

Let's start with something easy: matching an exact string. If we wanted to find commits with the message "I fixed a bug in Swift" – the kind of commit message that is frowned upon because it's not very descriptive! – you would write a predicate like this:

```
commitPredicate = NSPredicate(format: "message == 'I fixed a bug in
Swift'")
```

That means "make sure the message attribute is equal to this exact string." Typing an exact string like that is OK because you know what you're doing, but please don't ever use string interpolation to inject user values into a predicate. If you want to filter using a variable, use this syntax instead:

```
let filter = "I fixed a bug in Swift"
commitPredicate = NSPredicate(format: "message == %@", filter)
```

The **%@** will be instantly recognizable to anyone who has used Objective-C before, and it means "place the contents of a variable here, whatever data type it is." In our case, the value of **filter** will go in there, and will do so safely regardless of its value.

Like I said, "I fixed a bug in Swift" isn't the kind of commit message you'll see in your data, so == isn't really a helpful operator for our app. So let's write a real predicate that will be useful: put this in place of the **// 1** comment in the **changeFilter()** method:

```
ac.addAction(UIAlertAction(title: "Show only fixes", style: .Default,
handler: { [unowned self] _ in

  self.commitPredicate = NSPredicate(format: "message CONTAINS[c]
'fix'");

  self.loadSavedData()

}))
```

The **CONTAINS[c]** part is an operator, just like ==, except it's much more useful for our app. The **CONTAINS** part will ensure this predicate matches only objects that contain a string somewhere in their message – in our case, that's the text "fix". The **[c]** part is predicate-speak for "case-insensitive", which means it will match "FIX", "Fix", "fix" and so on. Note that we need to use **self.** twice inside the closure to make capturing explicit.

Another useful string operator is **BEGINSWITH**, which works just like **CONTAINS** except the matching text must be at the start of a string. To make this second example more exciting, I'm also going to introduce the **NOT** keyword, which flips the match around: this action below will match only objects that *don't* begin with 'Merge pull request'. Put this in place of the **// 2** comment:

```
ac.addAction(UIAlertAction(title: "Ignore Pull Requests",
style: .Default, handler: { [unowned self] _ in
```

```
    self.commitPredicate = NSPredicate(format: "NOT message BEGINSWITH
'Merge pull request'");

    self.loadSavedData()

}))
```

For a third and final predicate, let's try filtering on the "date" attribute. This is the **NSDate**
data type, and Core Data is smart enough to let us compare that date to any other date
inside a predicate. In this example, which should go in place of the **// 3** comment, we're
going to request only commits that took place 43,200 seconds ago, which is equivalent to
half a day:

```
ac.addAction(UIAlertAction(title: "Show only recent",
style: .Default, handler: { [unowned self] _ in

    let twelveHoursAgo = NSDate().dateByAddingTimeInterval(-43200)

    self.commitPredicate = NSPredicate(format: "date > %@",
twelveHoursAgo);

    self.loadSavedData()

}))
```

Again, the magic **%@** will work with Core Data to ensure the **NSDate** we created is used
correctly in the query.

For the final comment, **// 4**, we're just going to set **commitPredicate** to be **nil** so that all
commits are shown again:

```
ac.addAction(UIAlertAction(title: "Show all commits",
style: .Default, handler: { [unowned self] _ in

    self.commitPredicate = nil

    self.loadSavedData()

}))
```

That's it! **NSPredicate** uses syntax that is new to you so you might find it a bit daunting at first, but it really isn't very hard once you have a few examples to work from, and it does offer a huge amount of power to your apps.

Adding Core Data entity relationships: lightweight vs heavyweight migration

It's time to take your Core Data skills up a notch: we're going to add a second entity called Author, and link that entity to our existing Commit entity. This will allow us to attach an author to every commit, but also to find all commits that belong to a specific author.

Open the data model (Project38.xcdatamodeld) for editing, then click the Add Entity button. Name the entity Author, then give it two attributes: "name" and "email". Please make both strings, and make sure both are not marked as optional. This time we're also going to make one further change: select the "name" attribute and check the box marked "Indexed".
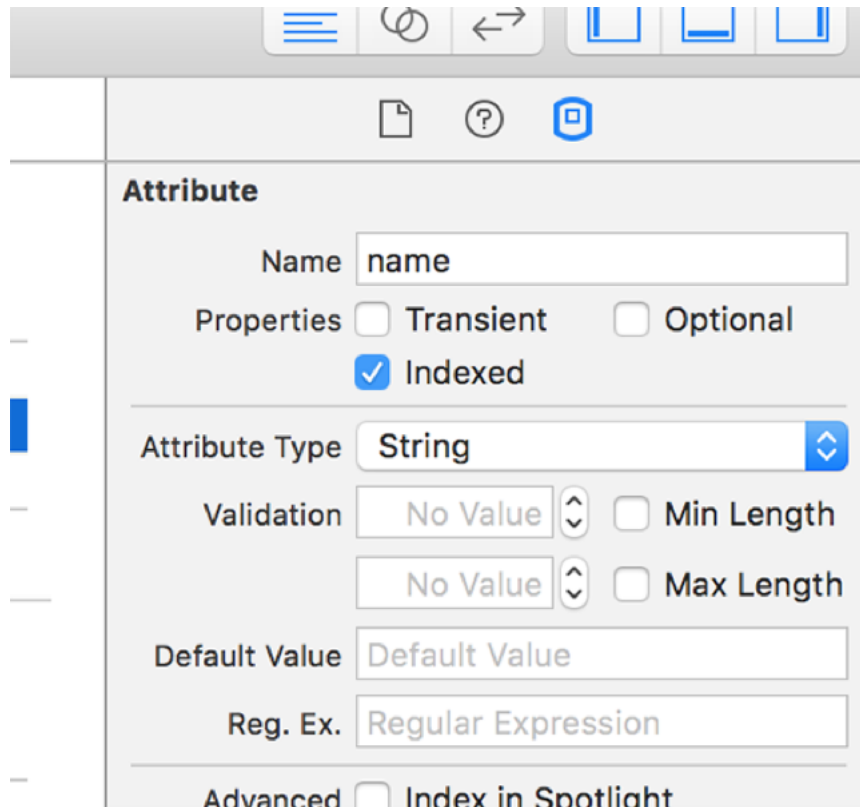
You can index as many attributes as you want, but don't go overboard: it takes time to create and update indexes, so you should index only the attributes you really need.

An indexed attribute is one that is optimized for fast searching. There is a cost to creating and maintaining each index, which means you need to choose carefully which attributes should be index. But when you find a particular fetch request is happening slowly, chances are it's because you need to index an attribute.

We want every Author to have a list of commits that belong to them, and every Commit to have the Author that created it. In Core Data, this is represented using relationships, which are a bit like calculated attributes except Core Data adds extra functionality to handle the situation when part of a relationship gets deleted.

With the Author entity selected, click the + button under the Relationships section – it's just below the Attributes section. Name the new relationship "commits" and choose "commit" for its destination. In the Data Model inspector, change Type to be "To Many", which tells Core Data that each author has many Commits attached to it.
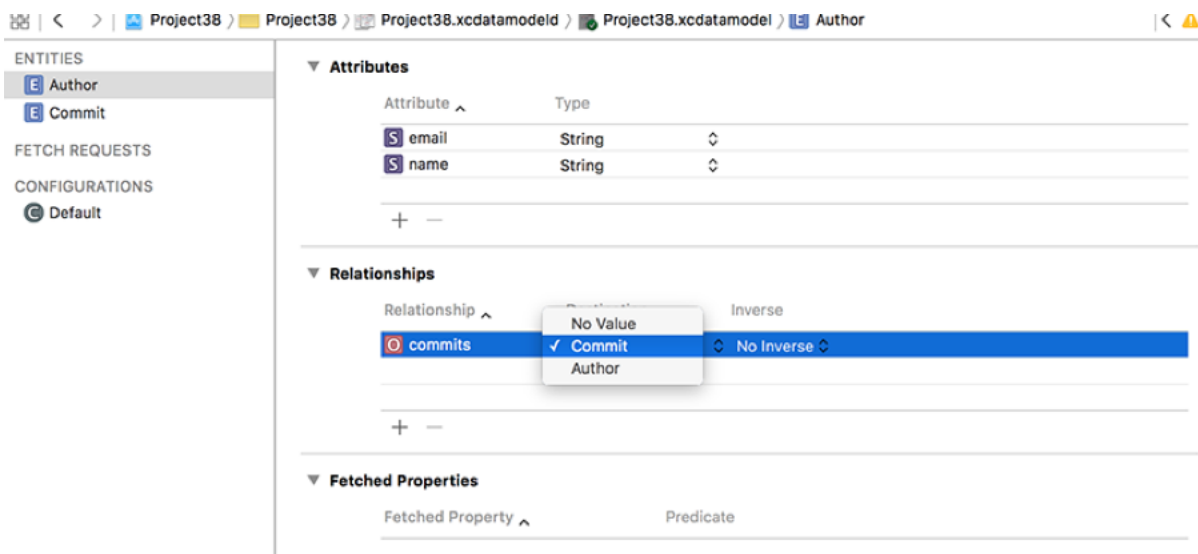
All relationships between objects should be managed inside your data model, because it allows Core Data to understand exactly what data you're storing and how it will be used.

Now choose the Commit entity we created earlier and add a relationship named "author". Choose Author for the destination then change "No Inverse" to be "commits". In the Data Model inspector, change Type to be "To One", because each commit has exactly one author).

That's it for our model changes, but if you run the app now it will either do nothing or crash, both of which are bad. If your app does nothing, it means Xcode didn't spot the model change, so please repeat the procedure from before: quit Xcode, relaunch, then hold down Alt and choose Product > Clean Build Folder. If your app crashed – great! Well, not *great*, but it was at least supposed to do that.

We just changed our data model, and Core Data doesn't know how to handle that – it considers any variation in its data model an unwelcome surprise, so either we need to tell

Core Data how to handle the changed model or we need to tell it to figure out the differences itself.

These two options are called "heavyweight migrations" and "lightweight migrations." The latter is usually preferable, and is what we'll be doing here, but it's only possible when your changes are small enough that Core Data can perform the conversion correctly. We added a new "authors" relationship, so if we tell Core Data to perform a lightweight migration it will simply set that value to be empty.

To tell Core Data we want to perform a lightweight migration, we need to set two properties when configuring our persistent store coordinator in the **startCoreData()** method. First, **NSMigratePersistentStoresAutomaticallyOption** needs to be **true**, which tells Core Data to upgrade its SQLite database when the model changes. Second, **NSInferMappingModelAutomaticallyOption** also needs to be **true**, which tells Core Data to figure out the differences when the model changes, and apply sensible defaults if possible.

To use these two options, find this line in the **startCoreData()** method:

```
try coordinator.addPersistentStoreWithType(NSSQLiteStoreType,
configuration: nil, URL: url, options: nil)
```

Now replace it with this:

```
try coordinator.addPersistentStoreWithType(NSSQLiteStoreType,
configuration: nil, URL: url, options:
[NSMigratePersistentStoresAutomaticallyOption: true,
NSInferMappingModelAutomaticallyOption: true])
```

That will stop the app from crashing, which is always a good thing. But it doesn't actually use our new Author entity, and to do that we first need to do something rather tedious: we need to re-use the NSManagedObject generator, which, if you remember, also means having to remove property optionals by hand.

So, go back to the data model, and choose Editor > Create NSManagedObject Subclass again. This time I want you to choose both Author and Commit, but don't forget to change

Group from the blue project icon to the yellow folder icon – Xcode does love to keep resetting that particular option.
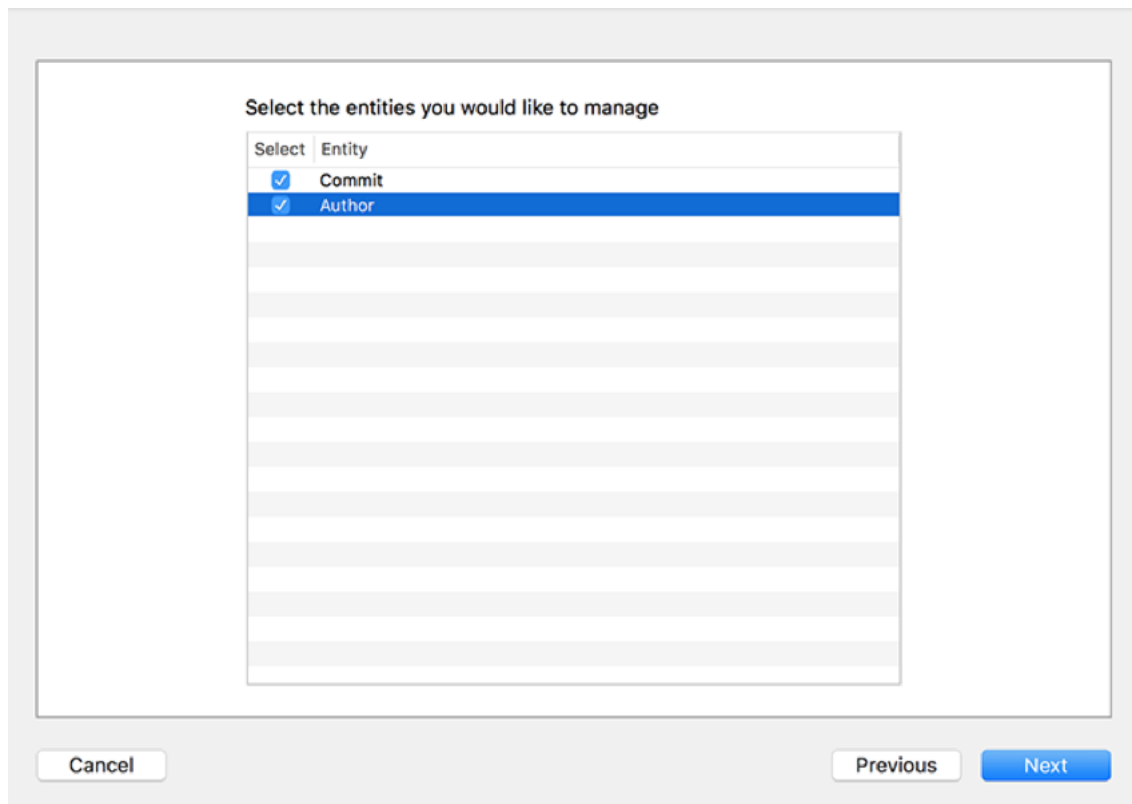
Run the Create NSManagedObject Subclass procedure again, but this time make sure you select both entities.

Once the files are generated you'll now have four files: two each for Author and Commit. Go ahead and remove optionality from the properties in Author+CoreDataProperties.swift and Commit+CoreDataProperties.Swift. While you're there, watch out for a strange corner case that might sometimes happen: in my testing sometimes Commit sometimes was created with **@NSManaged var author: NSManagedObject?**. If this happens to you, either recreate the classes once more or just change it to **@NSManaged var author: Author?** – then make it non optional, of course.

In order to attach authors to commits, I want to show you how to look for a specific named author, or create it if they don't exist already. Remember, we made the "name" attribute indexed, which makes it lightning fast for search. This needs to set up and execute a new **NSFetchRequest** (using an == **NSPredicate** to match the name), then use the result if there is one. If no matching author is found, we'll use **insertNewObjectForEntityForName()** to

create and configure a new author, and use that instead.

Put this new code just before the end of the **configureCommit()** method:

```
var commitAuthor: Author!


// see if this author exists already

let authorFetchRequest = NSFetchRequest(entityName: "Author")

authorFetchRequest.predicate = NSPredicate(format: "name == %@",
json["commit"]["committer"]["name"].stringValue)


if let authors = try?
managedObjectContext.executeFetchRequest(authorFetchRequest) as!
[Author] {

  if authors.count > 0 {

    // we have this author already

    commitAuthor = authors[0]

  }

}


if commitAuthor == nil {

  // we didn't find a saved author - create a new one!

  if let author =
NSEntityDescription.insertNewObjectForEntityForName("Author",
inManagedObjectContext: managedObjectContext) as? Author {

    author.name = json["commit"]["committer"]["name"].stringValue

    author.email = json["commit"]["committer"]["email"].stringValue

    commitAuthor = author

  }

}


// use the author, either saved or new
```

```
commit.author = commitAuthor
```

You'll note that I used **try?** for **executeFetchRequest()** this time, because we don't really care if the request failed: it will still fall through and get caught by the **if commitAuthor == nil** check later on.

To show that this worked, change your **cellForRowAtIndexPath** method so that the detail text label contains the author name as well as the commit date, like this:

```
cell.detailTextLabel!.text = "By \(object.author.name) on \
(object.date.description)"
```

You should be able to run the app now and see the author name appear after a moment, as Core Data merges the new data with the old. If you hit problems, don't be afraid to use "Reset Content and Settings" again in the simulator – that usually cures most Core Data hiccups.

We can also show that the inverse relationship works. Open DetailViewController.swift and change the **detailItem** property from **AnyObject?** to **Commit?**. Now change its **configureView()** method to this:

```
func configureView() {
  // Update the user interface for the detail item.
  if let detail = self.detailItem {
    if let label = self.detailDescriptionLabel {
      label.text = detail.message

      navigationItem.rightBarButtonItem = UIBarButtonItem(title:
"Commit 1/\(detail.author.commits.count)", style: .Plain, target:
self, action: "showAuthorCommits")
    }
  }
}
```

That will make a tappable button in the top-right corner showing how many other commits we have stored from this author. We haven't written a **showAuthorCommits()** method yet, but don't worry: that will be your homework later on!

Now that every commit has an author attached to it, I want to add one last filter to our **changeFilter()** method to show you just how clever **NSPredicate** is. Add this just before the "Show all commits" action:

```
ac.addAction(UIAlertAction(title: "Show only Durian commits",
style: .Default, handler: { [unowned self] _ in

  self.commitPredicate = NSPredicate(format: "author.name == 'Joe
Groff'");

  self.loadSavedData()

}))
```

There are three things that bear explaining in that code:

 • By using **author.name** the predicate will perform two steps: it will find the "author" relation for our commit, then look up the "name" attribute of the matching object.
 • Joe is one of Apple's Swift engineers. Although it's fairly likely you'll see commits by him, it can't be guaranteed – I'm pretty sure that Apple give him a couple of days vacation each year. Maybe.
 • Durian is a fruit that's very popular in south-east Asia, particularly Malaysia, Singapore and Thailand. Although most locals are big fans, the majority of foreigners find that it really, really stinks, so I'm sure there's some psychological reason why Joe Groff chose it for his website: **duriansoftware.com**.

Run your app now and the new filter should work. Remember, it might not return any objects, depending on just how many commits Joe has done recently. No pressure, Joe! In those changes, I also modified the detail view controller so that it shows the commit message in full, or at least as full as it can given the limited space – this app is starting to come together!

How to delete a Core Data object

The Xcode template we've been using comes with support for deleting objects, but it only removes them from the **objects** array directly – nothing is actually changed in our Core Data set up. So if you "delete" a commit, it will be there when you relaunch the app rather than gone for good.

Fortunately, this is really easy to change: our managed object context has a **deleteObject()** method that will delete any object regardless of its type or location in the object graph. Once an object has been deleted from the context, we can then call **saveContext()** to write that change back to the persistent store so that the change is permanent.

All this is easy to do by adding three new lines of code to the **commitEditingStyle** method. Here's the new method:

```
override func tableView(tableView: UITableView, commitEditingStyle
editingStyle: UITableViewCellEditingStyle, forRowAtIndexPath
indexPath: NSIndexPath) {

   if editingStyle == .Delete {

      let commit = objects[indexPath.row]

      managedObjectContext.deleteObject(commit)

      objects.removeAtIndex(indexPath.row)

      tableView.deleteRowsAtIndexPaths([indexPath],
withRowAnimation: .Fade)


      saveContext()

   } else if editingStyle == .Insert {

      // Create a new instance of the appropriate class, insert it into
the array, and add a new row to the table view.

   }

}
```

So, it 1) pulls out the **Commit** object that the user selected to delete, 2) removes it from the managed object context, 3) removes it from the **objects** array, 4) deletes it from the table

view, then 5) saves the context. Lines 3 and 4 were there in the original template, so it's only lines 1, 2, and 5 that are new. Remember: you must call **saveContext()** whenever you want your changes to persist.

Try running the app now, and either swiping to delete rows or using the Edit button in the top left. As you'll see you can delete as many commits as you want, and everything seems to work great. Now try running the app once again, and you'll get a nasty shock: the deleted commits reappear! What's going on?

Well, if you think about it, the app is doing exactly what we told it to do: every time it runs it re-fetches the list of commits from GitHub, and merges it with the commits in its data store. This means any commits we try to delete just get redownloaded again – they really are being deleted, but then they get recreated as soon as the app is relaunched.

This problem is not a hard one to fix, and it gives me a chance to show you another part of **NSFetchRequest**: the **fetchLimit** property. This tells Core Data how many items you want it to return. What we're going to do is find the newest commit in our data store, then use the date from that to ask GitHub to provide only newer commits.

First, go to the **fetchCommits()** method and modify the start of it to this:

```
func fetchCommits() {
  let newestCommitDate = getNewestCommitDate()


  if let data = NSData(contentsOfURL: NSURL(string: "https://
api.github.com/repos/apple/swift/commits?per_page=100&since=\
(newestCommitDate)")!) {
    let jsonCommits = JSON(data: data)
```

We'll be adding the **getNewestCommitDate()** method shortly, but what it will return is a date formatted as an ISO-8601 string. This date will be set to one second after our most recent commit, and we can send that to the GitHub API using its "since" parameter to receive back only newer commits.

Here is the **getNewestCommitDate()** method – only three pieces of it are new, and I'll explain them momentarily.

```
func getNewestCommitDate() -> String {

  let formatter = NSDateFormatter()

  formatter.timeZone = NSTimeZone(name: "UTC")

  formatter.dateFormat = dateFormatISO8601


  let newestCommitFetchRequest = NSFetchRequest(entityName: "Commit")

  let sort = NSSortDescriptor(key: "date", ascending: false)

  newestCommitFetchRequest.sortDescriptors = [sort]

  newestCommitFetchRequest.fetchLimit = 1


  if let commits = try?
managedObjectContext.executeFetchRequest(newestCommitFetchRequest)
as! [Commit] {

    if commits.count > 0 {

      return
formatter.stringFromDate(commits[0].date.dateByAddingTimeInterval(1))

    }

  }


  return formatter.stringFromDate(NSDate(timeIntervalSince1970: 0))

}
```

The first of the new pieces of code is the **fetchLimit** property for the fetch request. As you might imagine, it's always more efficient to fetch as few objects as needed, so if you can set a fetch limit you should do so. Second, the **stringFromDate()** method is the inverse of the **dateFromString()** method we used when parsing the commit JSON. We use the same date format that was defined earlier, because GitHub's "since" parameter is specified in an identical way. Finally, **dateByAddingTimeInterval()** is used to add one second to the time from the previous commit, otherwise GitHub will return the newest commit again.

If no valid date is found, the method returns a date from the 1st of January 1970, which will

reproduce the same behavior we had before introducing this date change.

This solution is a good start, but it has a small flaw – see if you can spot it! If not, don't worry: I'll be setting it as homework for you. Regardless, it gave me the chance to show you the **fetchLimit** property, and you know how much I love squeezing new knowledge in…

Optimizing Core Data Performance using NSFetchedResultsController

You've already seen how Core Data takes a huge amount of work away from you, which is great because it means you can focus on writing the interesting parts of your app rather than data management. But, while our current project certainly works, it's not going to scale well. To find out why open the Commit.swift file and modify it to this:

```
class Commit: NSManagedObject {

  override init(entity: NSEntityDescription,
insertIntoManagedObjectContext context: NSManagedObjectContext?) {

    super.init(entity: entity, insertIntoManagedObjectContext:
context)

    print("Init called!")

  }

}
```

When you run the program now you'll see "Init called!" in the Xcode log at least a hundred times - once for every **Commit** object that gets pulled out in our **loadSavedData()** method. So what if there are 1000 objects? Or 10,000? Clearly it's inefficient to create a new object for everything in our object graph just to load the app, particularly when our table view can only show a handful at a time.

Core Data has a brilliant solution to this problem, and it's called **NSFetchedResultsController**. It takes over our existing **NSFetchRequest** to load data, replaces our **objects** array with its own storage, and even works to ensure the user interface stays in sync with changes to the data by controlling the way objects are inserted and deleted.

No tutorial on Core Data would be complete without teaching **NSFetchedResultsController**, so that's the last thing we'll be doing in this project. I left it until the end because, although it's very clever and certainly very efficient, **NSFetchedResultsController** is entirely optional: if you're happy with the project as it is, you're welcome to skip over this last chapter.

First, add a new property for the fetched results controller:

```
var fetchedResultsController: NSFetchedResultsController!
```

We now need to rewrite our **loadSavedData()** method so that the existing **NSFetchRequest** is wrapped inside a **NSFetchedResultsController**. We want to create that fetched results controller only once, but retain the ability to change the predicate when the method is called again.

Before I show you the code, there are three new things to learn. First, we're going to be using the **fetchBatchSize** property of our fetch request so that only 20 objects are loaded at a time. Second, we'll be setting the master view controller as the delegate for the fetched results controller – you'll see why soon. Third, we need to use the **performFetch()** method on our fetched results controller to make it load its data.

Here's the revised **loadSavedData()** method:

```
func loadSavedData() {

  if fetchedResultsController == nil {

    let fetch = NSFetchRequest(entityName: "Commit")

    let sort = NSSortDescriptor(key: "date", ascending: false)

    fetch.sortDescriptors = [sort]

    fetch.fetchBatchSize = 20


    fetchedResultsController =
NSFetchedResultsController(fetchRequest: fetch, managedObjectContext:
managedObjectContext, sectionNameKeyPath: nil, cacheName: nil)

    fetchedResultsController.delegate = self

  }


  fetchedResultsController.fetchRequest.predicate = commitPredicate


  do {
```

```
    try fetchedResultsController.performFetch()

    tableView.reloadData()

  } catch {

    print("Fetch failed")

  }

}
```

Because we're setting **delegate**, you'll also need to make **MasterViewController** conform to the **NSFetchedResultsControllerDelegate** protocol, like this:

```
class MasterViewController: UITableViewController,
NSFetchedResultsControllerDelegate {
```

That was the easy part. When you use **NSFetchedResultsController**, you need to use it everywhere: that means it tells you how many sections and rows you have, it keeps track of all the objects, and it is the single source of truth when it comes to inserting or deleting objects.

You can get an idea of what work needs to be done by deleting the **objects** property: we don't need it any more, because the fetched results controller stores our results. Immediately you'll see five errors appear wherever that property was being touched, and we need to rewrite all those instances to use the fetched results controller.

First, the **prepareForSegue()** method. This line will be flagged as an error:

```
let object = objects[indexPath.row]
```

Please replace it with this:

```
let object = fetchedResultsController.objectAtIndexPath(indexPath)
as! Commit
```

You'll note that the fetched results controller uses index paths (i.e., sections as well as rows) rather than just a flat array. We also need to typecast its contents, because it returns **AnyObject** by default.

Second, replace the **numberOfSectionsInTableView()** and **numberOfRowsInSection** methods with these two new implementations:

```
override func numberOfSectionsInTableView(tableView: UITableView) ->
Int {

  return fetchedResultsController.sections?.count ?? 0

}


override func tableView(tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {

  let sectionInfo = fetchedResultsController.sections![section]

  return sectionInfo.numberOfObjects

}
```

As you can see, we can read the **sections** array, each of which contains an array of **NSFetchedResultsSectionInfo** objects describing the items in that section. For now, we're just going to use that for the number of objects in the section.

Third, find this line inside the **cellForRowAtIndexPath** method:

```
let object = objects[indexPath.row]
```

And replace it with this instead:

```
let object =
self.fetchedResultsController.objectAtIndexPath(indexPath) as! Commit
```

The final two errors are in the **commitEditingStyle** method, where deleting items happens. And this is where things get more complicated: we can't just delete items from the fetched results controller, and neither can we use **deleteRowsAtIndexPaths()** on the table view. Instead, Core Data is much more clever: we just delete the object from the managed object context directly.

You see, when we created our **NSFetchedResultsController**, we hooked it up to our existing managed object context, and we also made our current view controller its delegate. So when the managed object context detects an object being deleted, it will inform our fetched results controller, which will in turn automatically notify our view controller if needed.

So, to delete objects using fetched results controllers you need to rewrite the **commitEditingStyle** method to this:

```
override func tableView(tableView: UITableView, commitEditingStyle
editingStyle: UITableViewCellEditingStyle, forRowAtIndexPath
indexPath: NSIndexPath) {

  if editingStyle == .Delete {

    let commit =
fetchedResultsController.objectAtIndexPath(indexPath) as! Commit

    managedObjectContext.deleteObject(commit)

    saveContext()

  } else if editingStyle == .Insert {

    // Create a new instance of the appropriate class, insert it into
the array, and add a new row to the table view.

  }

}
```

As you can see, that code pulls the object to delete out from the fetched results controller, deletes it, then saves the changes – we no longer touch the table view here.

That being said, we do need to add one new method that gets called by the fetched results controller when an object changes. We'll get told the index path of the object that got

changed, and all we need to do is pass that on to the **deleteRowsAtIndexPaths()** method of our table view:

```
func controller(controller: NSFetchedResultsController,
didChangeObject anObject: AnyObject, atIndexPath indexPath:
NSIndexPath?, forChangeType type: NSFetchedResultsChangeType,
newIndexPath: NSIndexPath?) {

  switch type {

  case .Delete:

    tableView.deleteRowsAtIndexPaths([indexPath!],
withRowAnimation: .Automatic)


  default:

    break

  }

}
```

Now, you might wonder why this approach is an improvement – haven't we just basically written the same code only in a different place? Well, no. That new delegate method we just wrote could be called from anywhere: if we delete an object in any other way, for example in the detail view, that method will now automatically get called and the table will update. In short, it means our data is driving our user interface, rather than our user interface trying to control our data.

Previously, I said "Using attribute constraints can cause problems with **NSFetchedResultsController**, but in this tutorial we're always doing a full save and load of our objects because it's an easy way to avoid problems later." It's time for me to explain the problem: attribute constraints are only enforced as unique when a save happens, which means if you're inserting data then an NSFetchedResultsController may contain duplicates until a save takes place. This won't happen for us because I've made the project perform a save before a load to make things easier, but it's something you need to watch out for in your own code.

If you run the app now, you'll see "Init called!" appears far less frequently because the fetched results controller lazy loads its data – a significant performance optimization.

Before we're done with **NSFetchedResultsController**, I want to show you one more piece of its magic. You've seen how it has sections as well as rows, right? Well, try changing its constructor in **loadSavedData()** to be this:

```
fetchedResultsController = NSFetchedResultsController(fetchRequest:
fetch, managedObjectContext: managedObjectContext,
sectionNameKeyPath: "author.name", cacheName: nil)
```

The only change there is that I've provided a value for **sectionNameKeyPath** rather than **nil**. Now try adding this new method to **MasterViewController**:

```
override func tableView(tableView: UITableView,
titleForHeaderInSection section: Int) -> String? {

    return fetchedResultsController.sections![section].name

}
```

If you run the app now, you'll see the table view has sections as well as rows, with each section marked out with the author of those commits. So, **NSFetchedResultsController** is not only faster, but it even adds powerful functionality with one line of code – what's not to like?

Janek Spaderna

Move logic into one place by mergi...
By Janek Spaderna on 2016-01-25 19:43:23 +0...

Jordan Rose

[ClangImporter] Wait until a bridgi...
By Jordan Rose on 2016-01-27 22:55:16 +0000

[docs] LibraryEvolution: Public me...
By Jordan Rose on 2016-01-27 01:45:30 +0000

[docs] LibraryEvolution: To remov...
By Jordan Rose on 2016-01-27 01:45:29 +0000

Merge pull request #987 from ddu...
By Jordan Rose on 2016-01-26 05:01:12 +0000

[docs] LibraryEvolution: Eliminate t...
By Jordan Rose on 2016-01-26 00:43:19 +0000

[docs] LibraryEvolution: Move "Ver...
By Jordan Rose on 2016-01-26 00:43:19 +0000

[docs] LibraryEvolution: Weaken "f

Merge pull request #1107 from ahoppen/master

[Sema] Fix SR-590 Passing two parameters to a function that takes one...

The final app, running in landscape on an iPhone 6 Plus – the section names are inserted automatically by NSFetchedResultsController.

Wrap up

Core Data is *complicated*. It really is. So if you've made it this far, you deserve a pat on the back because you've learned a huge amount in this one project – good job! You've learned about model design, relationships, predicates, sort descriptors, persistent stores, managed object contexts, fetch requests, fetched results controllers, indexes, hashes and more, and I hope you're pleased with the final project.

But I'm afraid I have some bad news: even with everything you've learned, there's still a lot more to go if you really want to master Core Data. Migrating data models, multiple managed object contexts, delete rules, and thread safety should be top of that list, but at the very least I hope I've managed to give you a firm foundation on the technology – and perhaps even get you a bit excited about what it can do for you!

At this point, you know enough about Core Data that you should be able to start a new project with the Master-Detail Application template, but this time *enabling* the Core Data option for the template. You'll see all the code Apple generates for you, and hopefully you'll see how it's similar to our own code in this project.

Before you're done, I have two small tips for you. First, go to the Product menu and choose Scheme > Edit Scheme. In the window that appears, choose your Run target and select the Arguments tab. Now click + and enter the text **-com.apple.CoreData.SQLDebug 1**. Once that's done, running your app will print debug SQL into the Xcode log pane, allowing you to see what Core Data is up to behind the scenes.
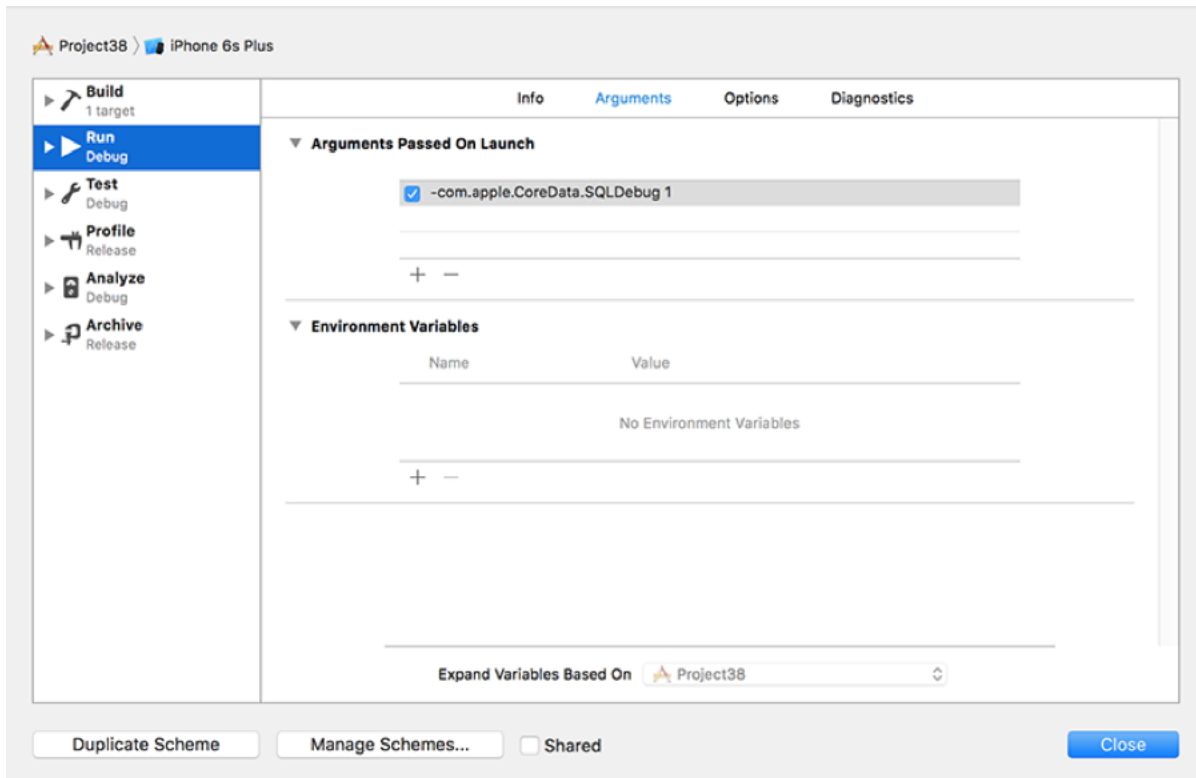
Enable the SQLDebug option for Core Data if you want to see how much work Core Data is doing on your behalf. Pay particular attention to the timings it prints out – this can help you identify and resolve slower parts of your code.

Warning: when this option is enabled, you will see scary language like "fault fulfilled from database." No, there isn't really a fault in your code – it's Core Data's way of saying that the objects it lazy loaded need to be loaded for real, so it's going back to the database to read them. You'll get into this more when you explore Core Data further, but for now relax: it's just a poor choice of phrasing from Apple.

Second, if you make changes to your model, watch out for the message "The model used to open the store is incompatible with the one used to create the store." If you intend to turn this into a production app, you need to check your managed object context after **startCoreData()** has run to see whether everything was started correctly, and handle things gracefully on failure. While you're testing, don't be afraid to reset the iOS simulator as often as you need to in order to clean out old models.

If you want to take the app further, here are some suggestions for homework:

• **Fun:** Now that we have section names in our table, using a sort descriptor for date

descending makes things look out of order. Try to add a second sort descriptor so that the fetch request sorts on author name ascending first, then date descending second.

 • **Tricky:** Use the "url" attribute of Commit in the detail view controller to show the GitHub page in a **WKWebView** rather than just a label.

 • **Taxing:** Rewrite the **getNewestCommitDate()** method so that it uses **NSUserDefaults** rather than a fetch request in order to fix the bug in the current implementation. (Didn't spot the bug? If users delete the most recent commit message, we also lose our most recent date!)

 • **Mayhem:** Complete the **showAuthorCommits()** method in the detail view controller. This should show a new table view controller listing other commits by the same author. To make your life easier, you might want to try going to the Author entity, choosing its "commits" relationship, then checking the "Ordered" box and recreating the **NSManagedObject** subclass. Don't forget to remove the optionality from the properties that get generated!

Project 39

Unit testing with XCTest

Learn how to write unit tests and user interface tests using Xcode's built-in testing framework.

Setting up

Although this is a technique project, it's a technique project with a difference because you're going to make an app from scratch. It's not a complicated app, don't worry: it will load a large file containing the text of all of Shakespeare's comedies, then have a table view showing how often each word is used.

Easy, right? Right. But here's why it's a technique project: while building this app you'll be learning all about XCTest, which is Xcode's testing framework. Although this isn't a tutorial on test-driven development, I will at least walk you through the concepts and apply them with you. Even better, I'll also be introducing you to some functional programming using **filter()**, so there's a lot to learn.

Please note: as I write this, Xcode 7.3 and Swift 2.2 are still in late beta. To avoid problems, there is one piece of code where I've given specific instructions for 2.2 and 2.1 so that everyone can follow along.

All set? Let's do this! Launch Xcode, then create a new Single View Application named Project39. For device please choose Universal, then check both Include Unit Tests and Include UI Tests. Click Create, then save the project somewhere safe.
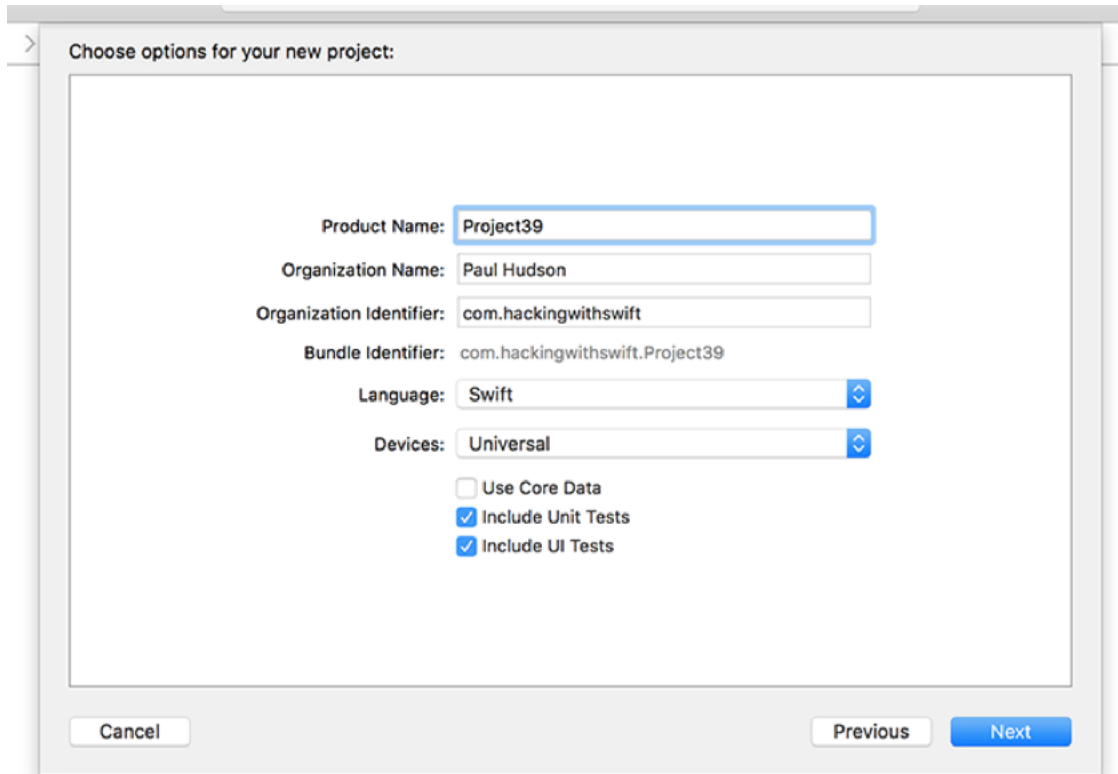
This project draws upon a text file containing the comedies of Shakespeare. **The Hacking with Swift GitHub repository** contains all the resources for these projects, and you'll find **plays.txt** inside the **project39-files** folder. Please drag that into your Xcode project before continuing.

We started with a Single View Application template, but we want to have a table view controller instead so we need to make a few changes:

1. Open ViewController.swift, then change **class ViewController: UIViewController {** to **class ViewController: UITableViewController {**2. Open Main.storyboard and delete its view controller. If you have trouble deleting it, try selecting it using the Document Outline.
3. Look in the Object library for a table view controller, then drag that out to where the previous view controller was.
4. Select the new table view controller, then change its class to be ViewController. This is done in the Identity inspector (Alt+Cmd+3) by setting the Class field.
5. In the Attributes inspector (Alt+Cmd+4) please check the box marked "Is Initial View Controller".

6. Go to the Editor menu and choose Embed In > Navigation Controller.

7. Select the prototype cell of your table view and change its style to be "Right Detail" and its reuse identifier to be "Cell".

8. If you want to, give your navigation bar a title, but this isn't required. Just double-click in the navigation bar space at the top of your table view controller and type some text.
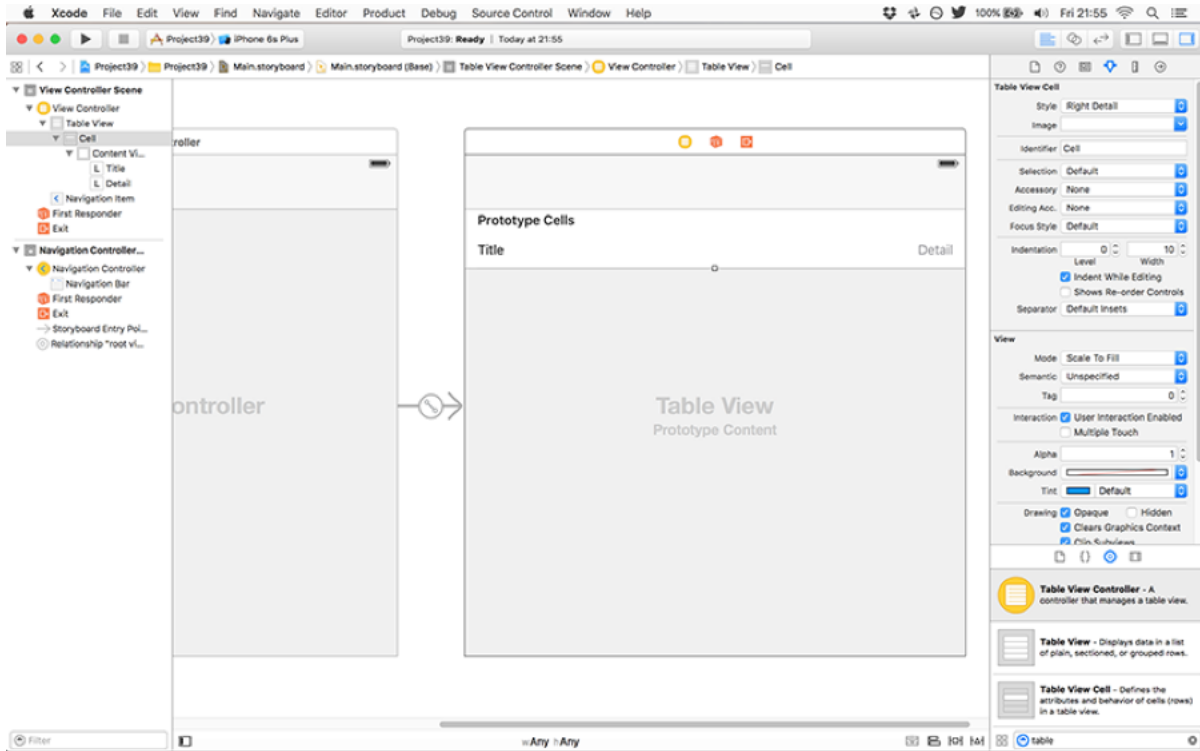
Once you've made all those changes your user interface should look like this.

At this point in your coding career, that should all come as second nature. You're welcome to try running the app now, but I'm afraid there isn't much to see yet – in fact, it should be almost blank!

Creating our first unit test using XCTest

At the core of test-driven development lies the concept that you should begin by writing a test that succeeds only when your code works the way you want it to. It might seem strange at first, but it makes a lot of sense: your new test will fail at first because your program doesn't do the right thing, then it's your job to write just enough code (but no more!) to make that test pass.

We're going to follow this approach here, but we need to do a little bit of setup work first so that we're able to write a failing test. So: go to the File menu and choose New > File. From the list of options, choose iOS > Source > Swift File. Click Next, then name it PlayData. We'll be using this to store all the words in the plays.

The goal right now is to write just enough code for us to return to testing, so for now just put this text into the file:

```
class PlayData {

  var allWords = [String]()

}
```

That's it: there's a class called **PlayData**, and we've given it a property called **allWords** that will hold an array of strings. We're not going to fill that array with data yet, though: first we need to write a test to check that **allWords** has been populated with the words from the plays.txt file. For now, just to make sure you understand how an XCTest works, we're going to write a test that checks **allWords** has exactly 0 strings inside it.

Look in the "Project39Tests" for Project39Tests.swift and open it for editing. You'll see it contains four methods: **setUp()**, **tearDown()**, as well as two example test methods, all wrapped up in a class that inherits from **XCTestCase**. Please delete the two test methods, so your file is left like this:

```
import XCTest

@testable import Project39
```

```
class Project39Tests: XCTestCase {

    override func setUp() {

        super.setUp()

        // Put setup code here. This method is called before the
invocation of each test method in the class.

    }


    override func tearDown() {

        // Put teardown code here. This method is called after the
invocation of each test method in the class.

        super.tearDown()

    }

}
```

We're going to write a very basic test that checks **allWords** has 0 items inside. Please add this method just below **tearDown()**:

```
func testAllWordsLoaded() {

  let playData = PlayData()

  XCTAssertEqual(playData.allWords.count, 0, "allWords must be 0")

}
```

If we include the method name and the closing brace, that's only four lines of code, none of which look that difficult. However, it masks quite a lot of functionality, so I want to walk through exactly what it does and why.

First, the method has a very specific name: it starts with "test" all in lowercase, it accepts no parameters and returns nothing. When you create a method like this inside an **XCTestCase** subclass, Xcode automatically considers it to be a test that should run on your code. When Xcode recognizes a test, you'll see an empty gray diamond appear in the left-hand gutter, next to the line numbers. If you hover over that – but don't click it just yet! – it will turn into a play button, which will run the test.

```
 11
◇12  class Project39Tests: XCTestCase {
 13      override func setUp() {
 14          super.setUp()
 15          // Put setup code here. This method is called before the invoca
 16      }
 17
 18      override func tearDown() {
 19          // Put teardown code here. This method is called after the invo
 20          super.tearDown()
 21      }
 22
◇23      func testAllWordsLoaded() {
 24          let playData = PlayData()
 25          XCTAssertEqual(playData.allWords.count, 0, "allWords must be 0"
 26      }
 27  }
 28
```

Xcode expects you to use a precise naming convention for your test methods, and if you follow these guidelines you should see empty gray diamonds next to your tests.

The first line of our **testAllWordsLoaded()** method does nothing surprising: it just creates a new **PlayData** object so we have something to test with. The second line is new, though, and uses a function called **XCTAssertEqual()**. This checks that its first parameter (**playData.allWords.count**) equals its second parameter (**0**). If it doesn't, the test will fail and print the message given in parameter three ("allWords must be 0").

XCTAssertEqual() lies at the center of XCTest: if all the calls to **XCTAssertEqual()** in a test return true, the test is considered a pass, otherwise it will fail. There are other assert functions you can use (**XCTAssertGreaterThan()**, **XCTAssertNotNil()**, etc), but you can do almost everything with **XCTAssertEqual()** just by using the correct parameters.

Now that you understand how the test works, hover over the gray diamond next to the test and click its play button. Xcode will run this single test, which means launching the app in the iOS Simulator and verifying that the **allWords** array contains 0 items. Because we haven't written any loading code yet, this test will succeed and the diamond will turn into a green checkmark. You'll also see a green checkmark next to the class name at the top, which means that all tests in this class passed last time they were run.

```
11
●12 class Project39Tests: XCTestCase {
13       override func setUp() {
14           super.setUp()
15           // Put setup code here. This method is called before the invoca
16       }
17
18       override func tearDown() {
19           // Put teardown code here. This method is called after the invo
20           super.tearDown()
21       }
22
●23       func testAllWordsLoaded() {
24           let playData = PlayData()
25           XCTAssertEqual(playData.allWords.count, 0, "allWords must be 0"
26       }
27 }
28
```

When tests have run successfully you'll be rewarded with green checkmarks. Pat yourself on the back!

Loading our data and splitting up words: filter()

The next step in our project is to get our app working a little bit, which means writing a method that loads the input text and splits it up into words.

Sticking with TDD for now, this means we first need to write a test that fails before updating our code to fix it.

Right now we're checking **allWords** contains 0 items, which needs to change: there are in fact 384,001 words in the input text (based on the character splitting criteria we'll get to shortly), so please update your **testAllWordsLoaded()** method to this:
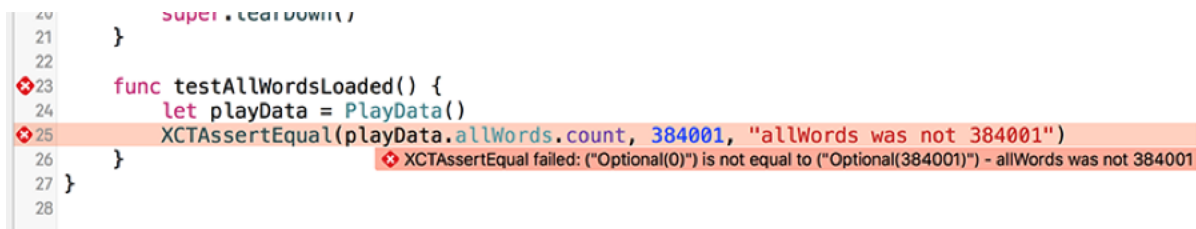
```
XCTAssertEqual(playData.allWords.count, 384001, "allWords was not
384001")
```

This number, 384,001, is of course entirely arbitrary because it depends on the input data. But that's not the point: we need to tell XCTest what "correct" looks like, because it has no way of knowing what constitutes a pass or a fail unless we give it specific criteria.

If you click the green checkmark next to the test now, it will be run again and will fail this time because we haven't written the loading code – XCTest expects **allWords** to contain 384,001 strings, but it contains 0. This is good, honest!

Our tests are now failing, which is a core part of test-driven development: write tests that fail, then write the code that makes those tests pass.

Let's put testing to one side for now and fill in some of our program – we'll be back with the testing soon enough, don't worry.

Add this to PlayData.swift:

```
init() {

  if let path = NSBundle.mainBundle().pathForResource("plays",
ofType: "txt") {

    if let plays = try? String(contentsOfFile: path, usedEncoding:
nil) {

      allWords =
plays.componentsSeparatedByCharactersInSet(NSCharacterSet.alphanumeri
cCharacterSet().invertedSet)

    }

  }

}
```

The only new line there is the one that sets **allWords**, which uses two new things at once. Previously we used **componentsSeparatedByString()** to convert a string into an array, but this time we're using **componentsSeparatedByCharactersInSet()** instead. This new method splits a string by any number of characters rather than a single string, which is important because we want to split on periods, question marks, exclamation marks, quote marks and more.

There are a number of ways of specifying character sets, including a helpful **NSCharacterSet(charactersInString:)** initializer that we could have used to specify the full list of characters we want to break on. But the simplest approach is to split on anything that isn't a letter or number, which can be achieved by inverting the alphanumeric character set as seen in the code.

That's all it takes to load enough data for us to move on with, but we can't tell that it works unless we also update the user interface to show our data. So, open ViewController.swift and add this property:

```
var playData = PlayData()
```

That gives the **ViewController** class its own **PlayData** object to work with. That one line also

creates the object immediately, which in turn will call the **init()** method we just wrote to load the word data – not bad for a single line of code!

All that's left in this step is to add the basic table view code to show some cells. Please add these two methods to **ViewController**:

```
override func tableView(tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {

  return playData.allWords.count

}


override func tableView(tableView: UITableView, cellForRowAtIndexPath
indexPath: NSIndexPath) -> UITableViewCell {

  let cell = tableView.dequeueReusableCellWithIdentifier("Cell",
forIndexPath: indexPath)


  let word = playData.allWords[indexPath.row]

  cell.textLabel!.text = word

  return cell

}
```

There is nothing new there, so I hope it posed no challenge for you.

Press Cmd+R to run the app and you'll see a long table full of words – it works! It's a long way from perfect, though: it's breaking on apostrophes (so ALL and S are on different lines), there are blank lines, there are duplicate words (the I after ACT and SCENE is repeated), and the detail text label just says "Detail" again and again.
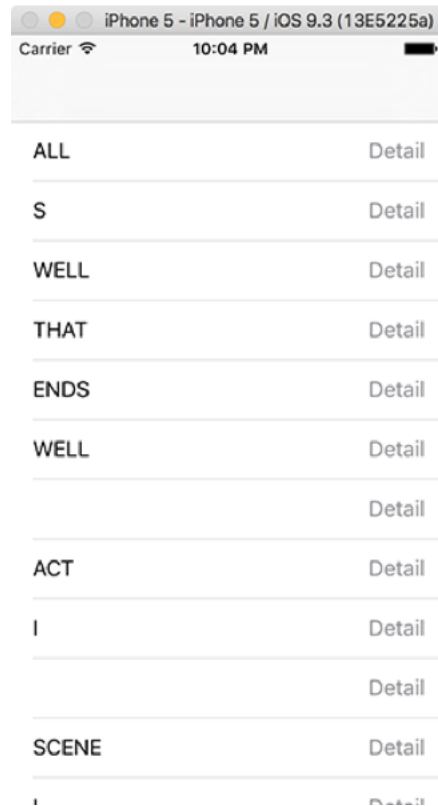
The results so far: word splitting works fine, but all those empty lines aren't great, and the detail text label needs to be fixed.

We're going to fix all those except the first one, which is a text issue rather than a coding issue. In fact, we need to fix the second one straight away because my calculation of 384,001 words excludes empty strings – we need to modify our **init()** method so that empty strings are removed if we want our test to pass.

As promised, I'm going to use this project to teach you a little bit of functional programming, in particular the **filter()** method. This creates a new array from an existing one, selecting from it only items that match a function you provide. Stick with me for a moment, because this is important: a function that accepts a function as a parameter, like **filter()**, is called a *higher-order function*, and allows you to write extremely concise, expressive code that is efficient to run.

Let's take a look at **filter()** now. Please add this to **init()**, just after the call to **componentsSeparatedByCharactersInSet()**:

```
allWords = allWords.filter { $0 != "" }
```

That one line is all it takes to remove empty lines from the **allWords** array. However, this syntax can look like line noise if you're new to Swift, so I want to deconstruct what it does by first rewriting that code in a way you're more familiar with. I don't want you to put any of this into your code – this is just to help you understand what's going on.

Here is that one line written out more verbosely:

```
allWords = allWords.filter({ (testString: String) -> Bool in
  if testString != "" {
    return true
  } else {
    return false;
  }
})
```

In that form, you can see that **filter()** is a method that takes a single parameter, which is a closure. That closure must accept a string, named **testString**, and return a **Bool**. The code then checks whether **testString** is empty or not, and returns either true or false.

Swift lives up to its name not only in that Swift code executes quickly, but it's also quick to write. So, there are a few shortcuts it offers to help reduce that long code down in size. For example, all that **(testString: String) -> Bool** definition isn't really needed: Swift can see that **filter()** wants a closure that accepts a string and returns true or false, so we don't need to repeat ourselves. So, let's take it out:

```
allWords = allWords.filter({ testString in
  if testString != "" {
    return true
  } else {
    return false;
```

```
    }
})
```

Next, we can collapse that **if/else** block into one line of code: **return testString != ""**.

When Swift runs **testString != ""** it will either find that statement to be true (yes, **testString** is not empty) or false (no, **testString** is empty), and pass that straight to **return**. So, this will return true if **testString** has any text, which is exactly what we want.

With that change, here's the code now:

```
allWords = allWords.filter({ testString in
  return testString != ""
})
```

Moving on, we can take advantage of Swift's trailing closure syntax, because **filter()**'s only parameter is a closure. If you remember, that means the parentheses aren't needed. So, the code can become this:

```
allWords = allWords.filter { testString in
  return testString != ""
}
```

Next, if your closure has only one expression – which ours does – and the closure must return a value, Swift lets you omit the **return** keyword entirely. This is because it knows the closure must return a value, and it can see you're only providing one line of code, so that must be the one that returns something. So, you can write this:

```
allWords = allWords.filter { testString in
  testString != ""
```

```
}
```

And now for the bit that usually confuses people: shorthand parameter names. When you use a closure like this, Swift automatically creates anonymous parameter names that start with a dollar sign then have a number: **$0**, **$1**, **$2**, **$3** and so on. This unique naming really helps them stand out, so if you see them in code you immediately know they are shorthand parameter names – you literally cannot use names like this yourself, so they only have one meaning.

Swift gives you one of these shorthand parameters for every parameter that your closure accepts. In this case, our filter closure accepts exactly one parameter, which is **testString**. If we want to use Swift's shorthand parameter names instead, we don't need **testString** any more because **testString** and **$0**, so that whole **testString in** part can go away:

```
allWords = allWords.filter {

  $0 != ""

}
```

Now all that's left is to put that on a single line, and we're done:

```
allWords = allWords.filter { $0 != "" }
```

Now, even though I've explained the thinning process that goes from long code to tiny code, you might still look at that one line and find it confusing. That's OK: you should write code however you want to write code. But in this instance, I hope you'll agree that having one simple, clutter-free line of code is easier to read, understand, and maintain than five or six lines.

Now that empty lines are being stripped out, you should be able to return back to **testAllWordsLoaded()** and have it pass.
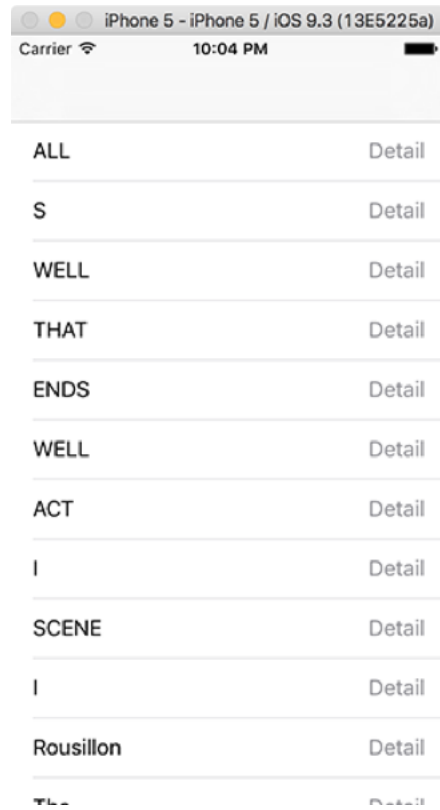
With blank lines removed this app is almost starting to look useful!

Counting unique strings in an array

Running the **allWords** array through **filter()** solved the problem of empty lines in the array, but there are still two more problems we need to address: duplicate words, and that pesky "Detail" text in the detail text label.

Well, we're going to fix two of them right now, at the same time. And, for the first time in this series, I'm going to have you write some bad code. Trust me: this will all become clear shortly, and it will be corrected.

Our app is going to show the number of times each word is used inside Shakespeare's comedies. To do that, we need to calculate how often each word appears, so we're going to add a new property to **PlayData** to store that calculation. Please add this now:

```
var wordCounts = [String: Int]()
```

That dictionary will hold a string as its key (e.g. "yonder") and a number as its value (e.g. 14), so that we can check the frequency of any word whenever we need to.

Our **init()** method already splits all the text up into words, but we need to add some new code to add the counting. This is fairly straightforward to write: loop through every word in the **allWords** array, add one to its **wordCounts** count if we have it, or set its count to 1 if we don't have it.

Modify the **init()** method in PlayData.swift so that this code appears after the call to **filter()**:

```
for word in allWords {
  if wordCounts[word] == nil {
    wordCounts[word] = 1
  } else {
    wordCounts[word]! += 1
  }
}
```

Note that I used a force unwrap with the **+=** operator, because at that point we know there is a value to modify.

Once that loop completes, **allWords** will contain every word that is used in the plays, as well as its frequency. Because we're using words as the dictionary keys, each word can appear only once in the dictionary. This means we can instantly remove duplicates from **allWords** by creating a new array from the keys of **wordCounts**. Add this code just after the previous loop:

```
allWords = Array(wordCounts.keys)
```

Our app has taken a leap towards its end goal, but we also just broke our test: now that we show each word only once, our **testAllWordsLoaded()** test will fail because there are substantially fewer strings in the **allWords**. So, please go to Project39Tests.swift and amend it one last time:

```
XCTAssertEqual(playData.allWords.count, 18440, "allWords was not
18440")
```

That's the first of two problems down: every word now appears only once in the table, and you can run the app now to verify that.
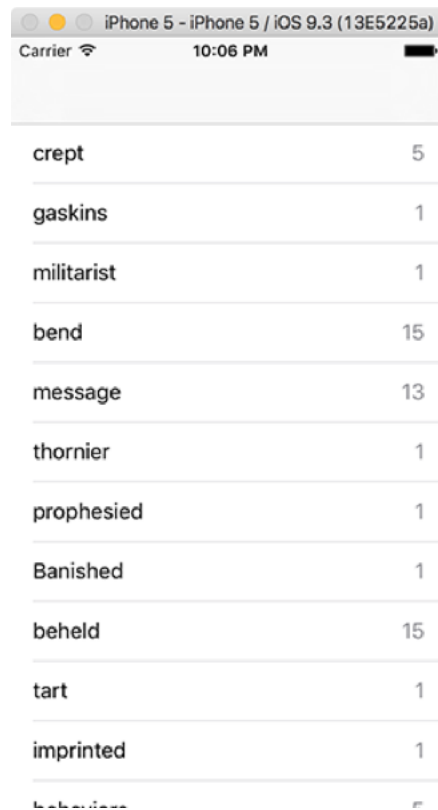
The last problem is to fix the detail text label so that it says how many times a word is used rather than just "Detail". With our new **wordCounts** dictionary we can fix this in just one line of code in ViewController.swift – add this line to **cellForRowAtIndexPath** just before the return line:

```
cell.detailTextLabel!.text = "\(playData.wordCounts[word]!)"
```

If you run the app now you'll see every word now has its count next to it – that wasn't so hard, was it?

Using our word counting code, we can now show frequency next to every word in the plays.

Before we're done, let's add another test to make sure our word counting code doesn't break in the future. Have a look through the table to find some words that interest you, and note down their frequencies. I chose "home" (174 times), "fun" (4 times), and "mortal" (41 times), but you're welcome to choose any words that interest you. Switch to Project39Tests.swift and add a new test:

```swift
func testWordCountsAreCorrect() {
  let playData = PlayData()

  XCTAssertEqual(playData.wordCounts["home"], 174, "Home does not appear 174 times")

  XCTAssertEqual(playData.wordCounts["fun"], 4, "Fun does not appear 4 times")

  XCTAssertEqual(playData.wordCounts["mortal"], 41, "Mortal does not appear 41 times")
```

```
}
```

That test should pass, which is great. But more importantly it provides a fail-safe for future work: in the next chapter we're going to rewrite our word counting code, and this test will ensure we don't break anything while we work.

measureBlock(): How to optimize our slow code and adjust the baseline

You might have noticed there's a pre-written method called **setup()** in our unit tests, which contains this comment: "Put setup code here. This method is called before the invocation of each test method in the class." Why, then, do we have **let playData = PlayData()** in both our tests – couldn't that go into **setup()** to avoid repetition?

Well, no, and you're about to see why. You will probably have noticed that our new word frequency code has slowed down our app quite a bit. Even when running in the iOS Simulator, using the full power of your Mac, this code now takes about two seconds to run – try to imagine how much slower it would be on a real device!

Of course, this is all a clever ruse to teach you more things, and here I want to teach you how to use XCTest to check performance. Our new word counting code is slow, but the only reliable way to ensure it gets faster when we make changes is to create a new test that times how long it takes for our **PlayData** object to be created. This is why we can't create it inside the **setup()** method: we need to create it as part of a measurement in this next test, as you'll see.

XCTest makes performance testing extraordinarily easy: you give it a closure to run, and it will execute that code 10 times in a row. You'll then get a report back of how long the call took on average, what the standard deviation was (how much variance there was between runs), and even how fast each of those 10 runs performed if you want the details.

Let's write a performance test now – please add this to Project39Tests.swift:

```
func testWordsLoadQuickly() {

  measureBlock {

    _ = PlayData()

  }

}
```
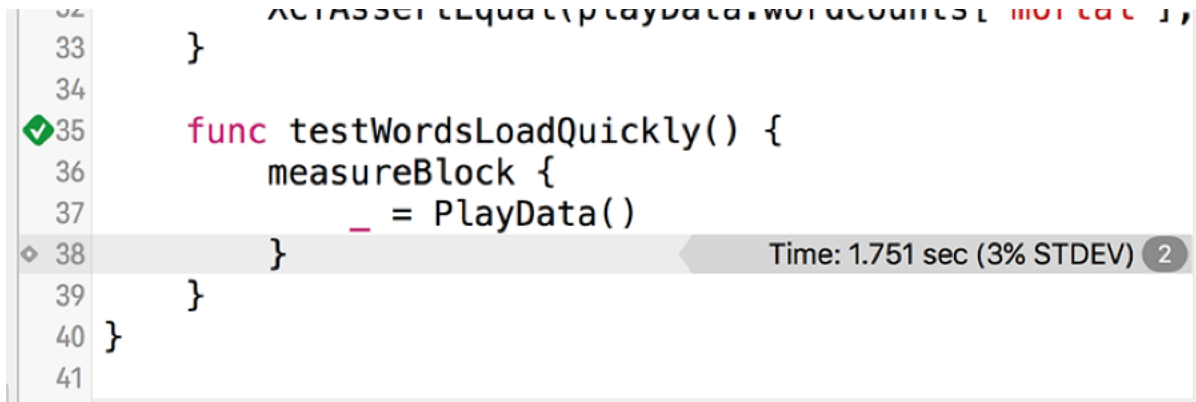
Were you expecting something more complicated? I told you it was easy and I meant it! That tiny amount of code is all it takes: assigning a new **PlayData** object to _ will load the file, split

it up by lines and count the unique words, so our test couldn't be any simpler.

Click the diamond in the gutter to run this performance test now, but be warned: it will take a little while because that closure is run 10 times. For me, each run took about 2 seconds, so the whole thing took about 20 seconds. If you have a slower computer, you might need to wait for a minute or two.

Once the test finishes you'll see a green arrow to show it succeeded, but that doesn't mean much right now because the test can't ever fail. But you'll also see something new: a gray line that tells you the results. On the right of the line it will say something like "Time: 2.060 sec (3% STDDEV)" so you can see the mean average time it took to run the code and also the standard deviation.

Performance tests offer different information to regular tests. As you can see here, you'll see the mean average time as well as the standard deviation, which helps you identify performance hotspots.

On the left of the line, in the gutter next to the line number, is a small gray diamond – clicking that will show you pop up information about all 10 runs. Also in that pop up is an important button that I want you to click now: Set Baseline. That marks your previous test run as the baseline against which future test runs should be compared to see whether performance has improved or worsened.
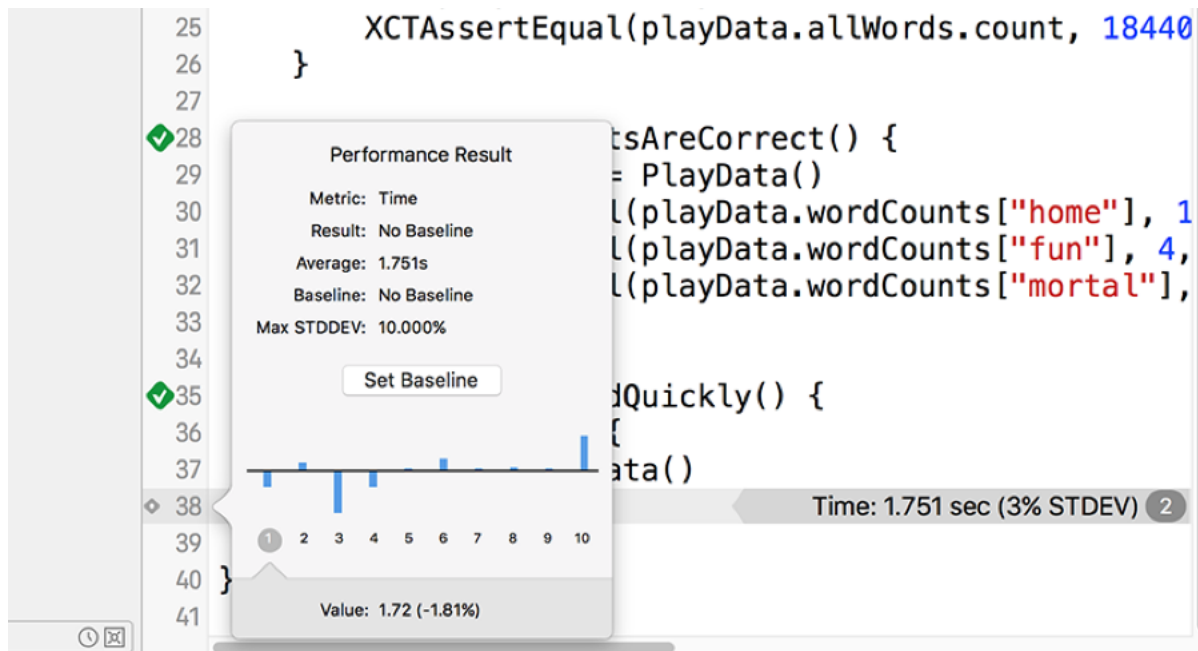
You can see detailed performance data, and set a baseline, by clicking the tiny gray diamond on the left of the standard deviation line.

Now that you've set a baseline, run the test again and wait for it to complete. When it finishes, you should see another green checkmark and you might also see a slight natural variance in the performance results. But now the checkmark *means* something: it means your code executed within a reasonable variance of the baseline, which in Xcode's eyes means that the deviation from the baseline was under 10%. If your code becomes a great deal slower, Xcode will warn you because it means something major has changed.

I promised earlier that I was going to make you write some bad code, and that it was going to be fixed. I hope you can see why: even though our loading code is very simple, it takes an extremely long time to load and we need to make it faster. In doing so you'll also get to see how XCTest helps you identify major performance changes and adjust your baseline as needed.

As a reminder, here's the slow code:

```
for word in allWords {
  if wordCounts[word] == nil {
    wordCounts[word] = 1
```

```
    } else {

        wordCounts[word]! += 1

    }

}


allWords = Array(wordCounts.keys)
```

You might wonder how we could optimize something so trivial, but it turns out we can take out most of the code while also making it run significantly faster. This is possible thanks to one of my favorite iOS classes: **NSCountedSet**. This is a set data type, which means that items can only be added once. But it's a *specialized* set object: it keeps track of how many times items you tried to add and remove each item, which means it can handle de-duplicating our words while storing how often they are used. Did I also mention it's fast?

To use **NSCountedSet** we need to make a few changes. First, change the wordCounts property of PlayData to this:

```
var wordCounts: NSCountedSet!
```

Now remove all the slow code (the nine lines I showed you above) and put these two in their place:

```
wordCounts = NSCountedSet(array: allWords)
allWords = wordCounts.allObjects as! [String]
```

The first line creates a counted set from all the words, which immediately de-duplicates and counts them all. The second line updates the **allWords** array to be the words from the counted set, thus ensuring they are unique.

Two more changes. First, in the **cellForRowAtIndexPath** method of ViewController.swift we need to use the **countForObject()** method to find out how often a word was used:

```
cell.detailTextLabel!.text = "\
(playData.wordCounts.countForObject(word))"
```

Then we need to make the same change in the **testWordCountsAreCorrect()** method of Project39Tests.swift:

```
func testWordCountsAreCorrect() {

  let playData = PlayData()

  XCTAssertEqual(playData.wordCounts.countForObject("home"), 174,
"Home does not appear 174 times")

  XCTAssertEqual(playData.wordCounts.countForObject("fun"), 4, "Fun
does not appear 174 times")

  XCTAssertEqual(playData.wordCounts.countForObject("mortal"), 41,
"Mortal does not appear 41 times")

}
```

That's it: click the green checkmark next to **testWordsLoadQuickly()** to re-run the performance test, and you'll see it now runs two or even three times faster. Xcode won't mark this test as a failure, though: Xcode only considers a test to be failed if it performs at least 10% *slower* than the baseline.

Our new code is significantly faster, and works just as well, so we're going to update our measurement baseline so it's used in the future. To do that, click the small gray diamond in the gutter at the end of **testWordsLoadQuickly()**, then click its Edit button. An Accept button will appear, which you should click to transfer the latest result to the baseline, then finally click Save.

If you were wondering, Xcode does two neat things with these benchmarks. First, they are checked into source control, which means they are shared with other team members. Second, they are stored against specific device configurations, which means Xcode won't warn when you your iPhone 4s performs slower than an iPhone 6s.

Before we're done with this chapter, there's one more thing I want to do: sort the array so that the most frequent words appear at the top of the table. This can be done with the **sort()**

method, which takes a closure describing how objects should be sorted. Swift will call this closure with a pair of words, and the closure should return true if the first word is sorted before the second. Using all the shorthand techniques you learned earlier, this means returning true if **$0** should be sorted before **$1**.

Replace these two lines:

```
wordCounts = NSCountedSet(array: allWords)
allWords = wordCounts.allObjects as! [String]
```

…with these:
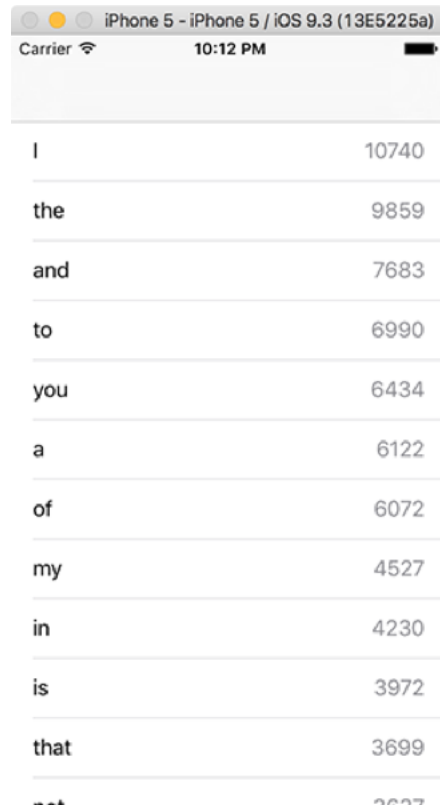
```
wordCounts = NSCountedSet(array: allWords)
let sorted = wordCounts.allObjects.sort
{ wordCounts.countForObject($0) > wordCounts.countForObject($1) }
allWords = sorted as! [String]
```

Remember, that closure needs to accept two strings (**$0** and **$1**) and needs to return true if the first string comes before the second. We call **countForObject()** on each of those strings, so this code will return true ("sort before") if the count for **$0** is higher than the code for **$1** – perfect.

If you run the app now you'll see the most frequently used words appear at the top – good job! Note, though, that running this sort takes a little time, so make sure you update the baseline for **testWordsLoadQuickly()** to reflect that change.

The table view is now finished: it shows all words, with no duplicates, and sorted by frequency descending.

Filtering using functions as parameters

We're going to add the ability for users to filter the word list in one of two ways: by showing only words that occur at or greater than a certain frequency, or by showing words that contain a specific string. This will work by giving **PlayData** a new array, **filteredWords**, that will store all words that matches the user's filter. This will also be used for the table view's data source.

As before, we're going to be writing the tests first so we can be sure the code we right is correct, but first we must create some skeleton code in **PlayData** that the test will work. Start by adding this **filteredWords** property to **PlayData**:

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

Now add this empty method, just below the existing **init()** method:

```
func applyUserFilter(input: String) {

}
```

That's just enough functionality for us to start writing tests: an **applyUserFilter()** method that accepts a single string parameter, such as "home" or "100". What it needs to do is decide whether that parameter contains a number ("100") or not ("home"), then either show words with that frequency or words that match that substring.

I've done some number crunching for you, and have found that 495 words appear at least 100 times, whereas only one word appears more than 10,000 times. I've also found that "test" appears 56 times, "Swift" appears 7 times, and "Objective-C" doesn't appear once – conclusive proof, I think, that Shakespeare prefers Swift.

Using these numbers, as well as some more, we can write the following test in Project39Tests.swift:

```
func testUserFilterWorks() {
```

```swift
    let playData = PlayData()

    playData.applyUserFilter("100")
    XCTAssertEqual(playData.filteredWords.count, 495)

    playData.applyUserFilter("1000")
    XCTAssertEqual(playData.filteredWords.count, 55)

    playData.applyUserFilter("10000")
    XCTAssertEqual(playData.filteredWords.count, 1)

    playData.applyUserFilter("test")
    XCTAssertEqual(playData.filteredWords.count, 56)

    playData.applyUserFilter("swift")
    XCTAssertEqual(playData.filteredWords.count, 7)

    playData.applyUserFilter("objective-c")
    XCTAssertEqual(playData.filteredWords.count, 0)
}
```

I haven't included any messages to print when the tests fail, but I'm sure you can fill those in yourself!

Finding the numbers for this wasn't hard, so you're welcome to try it yourself once you've written the real **applyUserFilter()** later on: just pass anything you want into **applyUserFilter()** then assert that it's equal to 0. When you run the test, Xcode will check all the assertions you've made, and tell you what the actual answer was. You can then update your number with Xcode's number, and you're done.

If you run that new test now it will fail – after all, **filteredWords** is never actually being set in the **PlayData** class, so it will always contain 0. This is a feature of test-driven development:

write tests that fail, then write just enough code to make those tests pass. For us, that means filling in **applyUserFilter()** so that it does something useful.

To make this test pass is surprisingly easy, although I'm going to make your life more difficult by squeezing some extra knowledge into you.

Let's start with identifying what the user is trying to do. They will enter a string into a **UIAlertController**, which could be "100", "556", "dog" or even "objective-c". Our code needs to decide whether the string they entered was an integer (in which case it is used to filter by frequency) or not (in which case it's used to filter by substring).

Swift has a built-in way to find out whether a string contains an integer, because it comes with a special **Int** failable initializer that accepts a string. A failable initializer is just like that **init()** method we wrote for **PlayData**, but instead of **init()** it's **init?()** because it can fail – it can return **nil**. In this situation, we'll get **nil** back if Swift was unable to convert the string we gave it into an integer.

Using this approach, we can begin to fill in **applyUserFilter()**:

```
func applyUserFilter(input: String) {
  if let userNumber = Int(input) {
    // we got a number!
  } else {
    // we got a string!
  }
}
```

You've already seen how to use **filter()** and the **countForObject()** method of **NSCountedSet**, plus we used **rangeForString()** way back in project 4, so you should know everything you need to be able to write some filtering code to replace those two comments.

If you're not sure, have a think for a moment. My solution is below:

```
if let userNumber = Int(input) {
```

```
  filteredWords = allWords.filter
{ self.wordCounts.countForObject($0) >= userNumber }

} else {

  filteredWords = allWords.filter { $0.rangeOfString(input,
options: .CaseInsensitiveSearch) != nil }

}
```

The first filter creates an array out of words with a count great or equal to the number the user entered, which is used when their text input was parsed as an integer. The second filter creates an array out of words that contain the user's text as a substring, which is used when their text input was not a number.

But I already said I want to squeeze some more knowledge into you, and in this case I want to extend our app so that rather than apply a filter directly, **applyUserFilter()** just calls a different method, **applyFilter()**, telling it what the filter function should be. This will allow you to add your own filters later on from inside ViewController.swift, without having to manipulate the contents of the **PlayData** object directly.

To make this work, we're going to create a new method called **applyFilter()**, which will accept a function as its only parameter. This function needs to accept a single string parameter, and return true or false depending on whether that string should be included in the **filteredWords** array. That's the exact format required by the **filter()** method, so we can just pass it straight in.

Accepting a function as a parameter has syntax that can hurt your eyes at first, but the important thing to remember is that Swift considers functions to be a data type, just like strings, integers and others. This means they have a parameter name, just like strings and other data types.

First, here's what the **applyFilter()** method would look like if our filter was a regular string:

```
func applyFilter(filter: String) { }
```

Now, I'll modify that so that the **filter** parameter is actually a function that accepts a string and returns a boolean:

```
func applyFilter(filter: (String) -> Bool) { }
```

Let's break that down. First, the parameter is still called **filter**, which means that's how we can refer to it inside **applyFilter()**. Then we have **(String)**, which means this parameter is a function that accepts a single string parameter. Finally, we have **-> Bool**, which means the function should return a boolean.

It's possible to have as many of these as you want, so we could have written a method that accepts three filters if we wanted to:

```
func applyFilter(filter1: (String) -> Bool, filter2: (Int) -> String,
filter3: (Double)) { }
```

In that code, **filter2** must be a function that accepts an integer parameter and returns a string, and **filter3** must be a function that accepts a double and returns nothing. We don't need anything that complicated here, but I hope you can see the syntax isn't that scary once you're used to it!

Here's the definition of applyFilter() again:

```
func applyFilter(filter: (String) -> Bool) { }
```

It accepts a single parameter, which must be a function that takes a string and returns a boolean. This is exactly what **filter()** wants, so we can just pass that parameter straight on. Here's the final code for **applyFilter()**:

```
func applyFilter(filter: (String) -> Bool) {
  filteredWords = allWords.filter(filter)
}
```

With that method written, we can now update **applyUserFilter()** so that it calls **applyFilter()** rather than modifying **filteredWords** directly, like this:

```
func applyUserFilter(input: String) {

  if let userNumber = Int(input) {

    applyFilter { self.wordCounts.countForObject($0) >= userNumber }

  } else {

    applyFilter { $0.rangeOfString(input,
options: .CaseInsensitiveSearch) != nil }

  }

}
```

Does this code work? Well, there's only one way to find out: re-run the **testUserFilterWorks()** test and see what it returns. This test was failing before because we weren't even modifying **filteredWords**, but hopefully now all six assertions will evaluate to true, and the test will pass.

Having two methods rather than one might seem pointless to you, but it's actually smart forward-thinking. Modifying the **filteredWords** property in only one place means that if we add more code to **applyFilter()** later on, it will immediately be used everywhere the method is called. If we had modified **filteredWords** directly, we'd need to remember all the places it was changed and copy-paste code there every time a change was made.

This two-method approach also gives us encapsulation, which means that functionality is encapsulated inside an object rather than exposed for others to manipulate. If you want to adjust filters directly from ViewController.swift – which is a perfectly valid thing to want to do – you really wouldn't want to change the **filteredWords** property directly. Instead, it's much nicer to call a method, and trust that **PlayData** will do the right thing.

There is a catch with this approach: what's stopping you from (unwisely!) trying to change the **filteredWords** property from ViewController.swift? The answer is "nothing" - you could put something like this in **viewDidLoad()** if you really wanted to:

```
playData.filteredWords = ["Neener!"]
```

Doing that would unpick all the work we did to avoid accessing **filteredWords**. Fortunately, Swift comes to the rescue: we can specify that everyone can *read* from the **filteredWords** property, but only the **PlayData** class can *write* to it. This restores our safety, and forces everyone to use the **applyUserFilter()** and **applyFilter()** methods.

To make this change, adjust the **filteredWords** property in **PlayData** to this:

```
private(set) var filteredWords = [String]()
```

That marks the setter of **filteredWords** – the code that handles writing – as private, which means only code inside PlayData.swift can use it. The getter – the code that handles reading – is unaffected.

You might think I engineered all this just to teach you even more Swift, but I couldn't possibly comment…

Updating the user interface with filtering

We've written the tests to prove that filtering works, but those filters don't do anything in the user interface just yet. We're going to make a few small changes so that our view controller uses **filteredWords** rather than **allWords**, then we'll add an alert controller so that users can enter a filter by hand.

First: using **filteredWords**. This is done in three changes, two of which are trivial: open ViewController.swift, and replace the two instances of **allWords** with **filteredWords**. If you run the app now you'll see no text in the table view, which is where we need to make the third change.

By default, **filteredWords** contains nothing, which is why the table is empty. It's only when a filter is applied that words are added, so our fix is just a matter of adding one line of code immediately before the end of **init()** in **PlayData**:

```
applyUserFilter("swift")
```

That will run an initial filter looking for the word "swift". If you want to show all words when the app first runs, use this code instead:

```
filteredWords = allWords
```

For now, though, please stick with **applyUserFilter("swift")** – the reason for this will become clear soon.

Now onto the interesting stuff: letting the user enter a filter value. This needs to show a **UIAlertController** with a text field and two buttons, Filter and Cancel. When the user taps Filter, whatever they entered in the text field needs to be sent to the **applyUserFilter()** method of **PlayData**, and the table reloaded to reflect their changes.

We've covered how to do all this before, but I'll give you a few reminders once you've read the code. Please add this new method to ViewController.swift:

```
func searchTapped() {

  let ac = UIAlertController(title: "Filter…", message: nil,
preferredStyle: .Alert)

  ac.addTextFieldWithConfigurationHandler(nil)


  ac.addAction(UIAlertAction(title: "Filter", style: .Default,
handler: { [unowned self] _ in

    let userInput = ac.textFields?[0].text ?? "0"

    self.playData.applyUserFilter(userInput)

    self.tableView.reloadData()

  }))


  ac.addAction(UIAlertAction(title: "Cancel", style: .Cancel,
handler: nil))


  presentViewController(ac, animated: true, completion: nil)

}
```

We haven't used **addTextFieldWithConfigurationHandler()** for a while, but here our use is simple because we don't need any special configuration for the text field – just calling it with **nil** will add a text field with no special options.

The one line your brain might have stalled on is this one:

```
let userInput = ac.textFields?[0].text ?? "0"
```

That contains two optionals: the **textFields** property might be **nil** (i.e., if there aren't any text fields) and even if we pull out the first text field from the array it might not have any text. Rather than try to fight our way through the maze of optionals, this code takes an easier approach: if either of the two optionals return **nil**, the nil coalescing operator (**??**) kicks in, and ensures that "0" is returned instead. This means **userInput** will always be a **String** and not a **String?**: it will either be something the user entered, or "0".

To finish up the user interface, we need to add a right bar button item to the navigation bar that will trigger the **searchTapped()** method. If you haven't used Swift 2.2 before, the syntax for this is a little different than what you're used to: add this line to **viewDidLoad()**:

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

In Swift 2.1 or earlier, **#selector(searchTapped)** would have been written as just **"searchTapped"**, but the Swift 2.2 approach gives us compile-time safety – if the **searchTapped()** method didn't exist, Xcode would refuse to build the app.

With that new line of code in place, please run the app now and try using the search button in the top-right corner. Thanks to us having a clear separation of our data model and our view controller, doing all the user interface work was pretty quick!
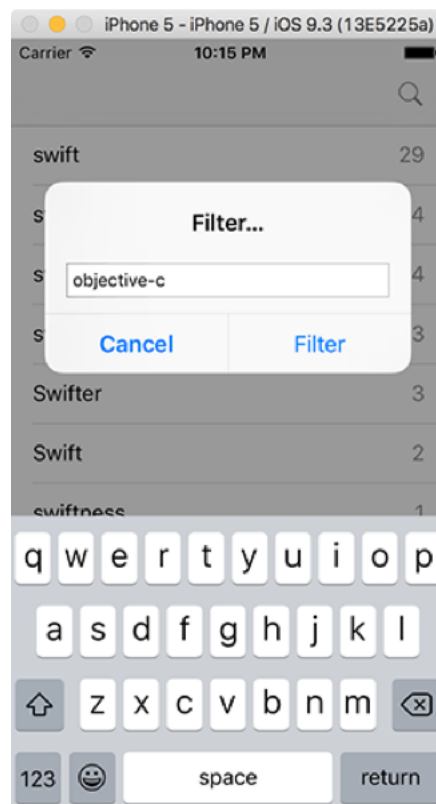
Users can filter either by substring or frequency from a single text field.

User interface testing with XCTest

There's one more trick XCTest has up its sleeve, and if you're not already impressed then I think this will finally win you over.

We've written tests for our data model: are the words loaded correctly? Are the word counts correct? Did the words get counted in an appropriate time? Does our user filtering work? These are all useful tests, and help make sure our code works now and in the future when further changes are made.

But none of those tests tell us whether our user interface is working as expected, so it's possible that the app could still be broken even though our model is in perfect condition.

XCTest has a solution, and it's a beautiful one: integrated user interface tests that manipulate your app as if there were a real user in control. XCTest is smart enough to understand how the system works, so it will automatically wait for things like animations to complete before trying to check your assertions.

When we created our project we added both unit tests (Project39Tests.swift) and also UI tests (Project39UITests.swift), and we'll be working with the latter now so please open Project39UITests.swift for editing.

You'll see Xcode has written **setup()** and **tearDown()** methods again, although this time the **setup()** method actually has some code in to get things started. You'll also see a **testExample()** method, but please just delete that – we'll be writing our own.

We're going to start with a very simple test: when the view controller loads, does it show the correct number of words? If you remember, our app applies an immediate filter for the word "swift", which appears 7 times in Shakespeare's comedies. So, to test that our initial app state is correct, we need to write this test:

```
func testInitialStateIsCorrect() {

  let table = XCUIApplication().tables

  XCTAssertEqual(table.cells.count, 7, "There should be 7 rows
initially")

}
```

Please go ahead and run that test now to make sure it works correctly.

OK, so how does that code work? There are only two lines, but it masks a whole lot of complexity:

 • Calling **XCUIApplication()** gets you access to the test harness for your running application. This lets you query its user interface and perform actions as if you were a user.
 • **.tables** will return an **XCUIElementQuery**, which in our situation would point to our table.
 • If there were more than one table visible, this would point to an array of tables, and we'd need to query them further to narrow it down to one table before acting on it. Trying to manipulate a query that points to more than one thing will crash your test.
 • This is the really mind-bending bit: the results of queries aren't fixed. So in our code the **let table** will point to a single table, but if the app adds two more tables for whatever reason, that **table** constant will now point to three tables – trying to manipulate it will crash your test.
 • Xcode uses the iOS accessibility system to navigate around these user interface tests. This is good because it means any application that is accessibility aware is ready for UI testing, but also because it encourages developers to add accessibility to their apps – which makes the world a better place for everyone. However, it's *bad* because the accessibility system has to read things from the screen rather than making API calls.
 • In our app, we start by applying a filter for the word "swift", which was for a reason: without this filter there are 18,000+ rows in our table, and the accessibility system seems to try to scan them all to perform our test. This is so slow that it simply wouldn't work, which is why the initial filter is applied.

So: user interface testing might look simple, but it's actually surprisingly hard. Fortunately, Xcode has a smart, simple and *almost* magical solution: test recording. Let's try it now – please create this empty method in your user interface tests:
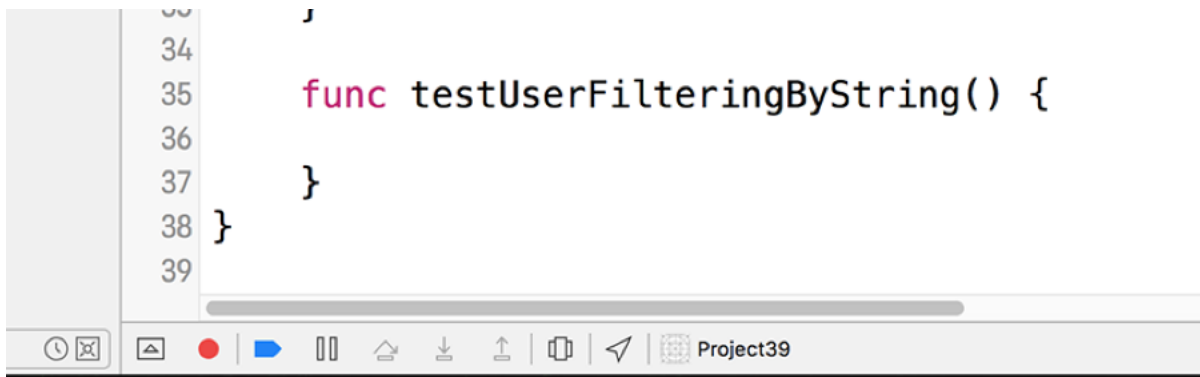
```
func testUserFilteringByString() {


}
```

Click inside that method, so that if you were to type you would be typing inside the method. Now click the red circle button just below your editing window: this will build and launch your app, but place it in record mode. It's a small button, but you can see it in the screenshot

below.

Create an empty method, click inside it, then click the small red circle that you can see in this screenshot.

When your app is recording a UI test, any taps, swipes, or other actions you perform in the app will automatically be converted to code inside your test – Xcode will literally write your test for you. However, there are three catches: first, it will usually write some fairly unpleasant code, and certainly rarely writes what a trained developer would do; second, it still doesn't know what a pass or fail looks like, so you need to add your own assertions at the end; third, sometimes it won't even write valid code.

So, it's a long way from perfect, but it does at least give you something to start with that you can easily rewrite. Your app is already in record mode, because you clicked the red circle to start it off. I want you to switch to the simulator, click the Search button, then use the on-screen keyboard to type "test". Finally, click the Filter button to dismiss the alert view, then go back to Xcode and click the red circle button again to end recording.

Note: Xcode does not like working with the iOS Simulator's hardware keyboard option – you need to use the on-screen keyboard.
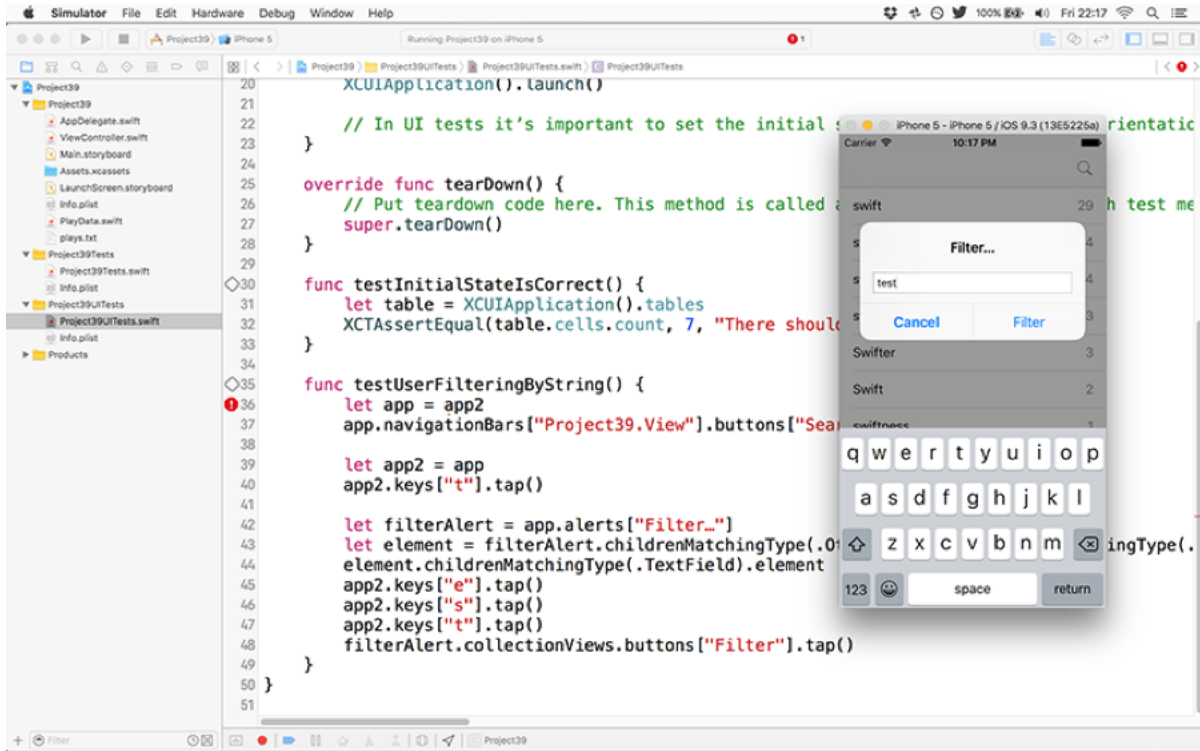
Try to position your Xcode window so you can see Xcode write its code while you work in the simulator.

Xcode will have written some code for you, but it might well be useless. Here's what I got:

```
func testUserFilteringByString() {
  let app = app2
  app.navigationBars["Project39.View"].buttons["Search"].tap()


  let app2 = app
  app2.keys["t"].tap()


  let filterAlert = app.alerts["Filter…"]
  let element =
filterAlert.childrenMatchingType(.Other).element.childrenMatchingType
(.Other).elementBoundByIndex(1).childrenMatchingType(.Other).element.
childrenMatchingType(.Other).element.collectionViews.childrenMatching
Type(.Other).element.childrenMatchingType(.Other).element.childrenMat
```

```
chingType(.Other).element.childrenMatchingType(.Other).element.childr
enMatchingType(.Other).elementBoundByIndex(1)

    element.childrenMatchingType(.TextField).element

    app2.keys["e"].tap()

    app2.keys["s"].tap()

    app2.keys["t"].tap()

    filterAlert.collectionViews.buttons["Filter"].tap()

}
```

That code is poor: it won't compile because the very first line contains a reference to an undefined "app2", the lines **let app = app2** and **let app2 = app** seem deeply confused, and don't get me started on the whole **let element = filterAlert** line. Worse, even if you fix the bugs, this test will work maybe only 1 in 10 times, and there's really no point adding tests that provide false positives.

However, like I said Xcode's attempt does give us something to build on:

- We'll need to add **let app = XCUIApplication()** at the start, because Xcode didn't.
- **app.navigationBars["Project39.View"].buttons["Search"].tap()** can be simplified to **app.buttons["Search"].tap()** because there's only one search button in view.
- **let filterAlert = app.alerts["Filter…"]** can be simplified to **let filterAlert = app.alerts**, again because there's only one of them.
- The massive long line that starts with **let element = filterAlert.childrenMatchingType** and the following one, can both be collapsed down into **let textField = filterAlert.textFields.element**, which is significantly shorter!
- All those **app2.keys["t"].tap()** won't work, but can be replaced with **textField.typeText("test")** to type everything at once.
- **filterAlert.collectionViews.buttons["Filter"].tap()** can be simplified to **filterAlert.buttons["Filter"].tap()** because we don't need to go via the collection view.
- To finish things off, we need to tell Xcode what a passing test looks like, so we'll add **XCTAssertEqual(app.tables.cells.count, 56, "There should be 56 words matching 'test'")** to match the "test" string that was entered.

With all that in mind, here's a vastly improved version of the test:

```
func testUserFilteringString() {
    let app = XCUIApplication()
    app.buttons["Search"].tap()

    let filterAlert = app.alerts
    let textField = filterAlert.textFields.element
    textField.typeText("test")

    filterAlert.buttons["Filter"].tap()

    XCTAssertEqual(app.tables.cells.count, 56, "There should be 56
words matching 'test'")
}
```

Now, you might wonder why it's worth using Xcode's UI recording system only to rewrite literally everything it produced. And I'll be honest: once you understand how Xcode UI testing works you'll probably write all your tests by hand, and you'll even add things like comments and shared code to make them a valuable part of your project. However, Xcode's UI recordings *are* useful when you're learning because it becomes significantly easier to get started when Xcode gives you something basic to work with.

Go ahead and run the test now, and you should see Xcode behaving like a real user. All being well, the assertion will prove true, and you'll get a green checkmark for your hard work – well done!

Wrap up

This has been the longest technique project by a long way, but I think you've learned a lot about the importance of good unit testing. We also managed to cover some functional programming techniques, discussed private setters, used functions as parameters, and even tried **NSCountedSet**, so I hope you're happy with the result!

Now that the full suite of tests are complete, you have three options ahead of you. First, you can run all the tests in a file by clicking the diamond (or green checkmark) next to the name of the test class. So, in Project39UITests.swift look in the gutter next to **class Project39UITests: XCTestCase** and click that. Second, you can run all the tests in your entire project by pressing Cmd+U. Give it a few moments to run, then you'll get a complete report.

Third, and most impressively, you can run all your tests on Xcode Server, which is the beginning of continuous integration: every time you commit a code change to source control, Xcode Server can pull down those changes, build the app, and run the full suite of tests. If you're working in a team, Xcode can produce a visual display of tests that are passing and failing, which is either motivational or depressing depending on your workplace!

If you'd like to take this app further, you should concentrate on testing. Can you write a test that verifies there are 55 table rows when the user filters by words that appear 1000 or more times? Can you write a test that ensures something sensible happens if the cancel button is pressed? Can you write a performance test to ensure **applyUserFilter()** doesn't get any slower?

There's also a bug in the code that you ought to be able to fix easily: if the user enters nothing into the filter text box, **applyUserFilter()** gets called with an empty string as its parameter and no results will be shown. It's down to you to think up a better solution: is it better to pretend Cancel was tapped instead? Or perhaps consider an empty string to mean "show all words"? It's your project now, so please choose what you think best.

Keep in mind that testing is only part of a sensible code review process. Yes, Xcode Server can check out your code and automatically validate that tests pass, but it is not a substitute for actual human interaction – looking through each other's code, and providing constructive criticism. Taking the time to read someone else's code, then encouraging and supporting them as they improve it, is a key developer skill – remember, code review is where mistakes get rubbed *out*, not rubbed *in*.

Appendix

The Swift Knowledge Base

Quick tips, tricks, and practical examples to help you do more with Cocoa Touch.

Arrays

How to count objects in a set using NSCountedSet
Availability: iOS 2.0 or later.

One of my favorite little gems in Cocoa Touch is called **NSCountedSet**, and it's the most efficient way to count objects in a set. Here's the code:

```
let set = NSCountedSet()
set.addObject("Bob")
set.addObject("Charlotte")
set.addObject("John")
set.addObject("Bob")
set.addObject("James")
set.addObject("Sophie")
set.addObject("Bob")


print(set.countForObject("Bob"))
```

Now to how it works: a *set* is like an array, except each item can appear only once. If we used an array in the above code, it would contain seven objects: three Bobs, one Charlotte, one John, one James and one Sophie. If we used a Swift set in the above code, it would contain four objects: one Bob, one Charlotte, one John, one James and one Sophie – the set ensures each item appears only once.

Now for the twist: **NSCountedSet** works similar to a set insofar as each object can appear only once, but it keeps track of how many times each item was added and removed. So, our counted set will have four objects in (like it would if it were a regular set), but **NSCountedSet** has a **countForObject()** method that will report back that Bob was added three times.

How to enumerate items in an array

Availability: iOS 7.0 or later.

There several ways to loop through an array in Swift, but using the **enumerate()** method is one of my favorites because it iterates overy each of the items while also telling you the items's position in the array.

Here's an example:

```
let array = ["Apples", "Peaches", "Plums"]

for (index, item) in array.enumerate() {
  print("Found \(item) at position \(index)")
}
```

That will print "Found Apples at position 0", "Found Peaches at position 1", and "Found Plums at position 2".

How to find an item in an array using indexOf()

Availability: iOS 7.0 or later.

The **indexOf()** method tells you the index of an element in an array if it exists, or returns nil otherwise. Because it's an optional value, you need to unwrap it carefully or at least check the result, like this:

```
let array = ["Apples", "Peaches", "Plums"]

if let index = array.indexOf("Peaches") {
  print("Found peaches at index \(index)")
}
```

How to join an array of strings into a single string

Availability: iOS 7.0 or later.

If you have an array of strings and want to merge all the elements together into a single string, it's just one line of code in Swift – albeit one you can't guess very easily!

For example, this joins array elements with a comma:

```
let array = ["Andrew", "Ben", "John", "Paul", "Peter", "Laura"]
let joined = array.joinWithSeparator(", ")
```

The result is that **joined** is set to "Andrew, Ben, John, Paul, Peter, Laura".

How to loop through an array in reverse

Availability: iOS 7.0 or later.

If you want to read through an array in reverse, you should use the **reverse()** method. You can use this as part of the regular fast enumeration technique if you want, which would give you code like this:

```
let array = ["Apples", "Peaches", "Plums"]

for item in array.reverse() {
  print("Found \(item)")
}
```

You can also reverse an enumerated array just by appending the method call, like this:

```
let array = ["Apples", "Peaches", "Plums"]
```

```
for (index, item) in array.reverse().enumerate() {

  print("Found \(item) at position \(index)")

}
```

Note that whether you call **reverse()** then **enumerate** or vice versa matters! In the above code, enumerate will count upwards, but if you use **array.enumerate().reverse()** it will count backwards.

How to loop through items in an array
Availability: iOS 7.0 or later.

Swift offers a selection of ways to loop through an array, but the easiest and fastest is known as fast enumeration and looks like this:

```
let array = ["Apples", "Peaches", "Plums"]

for item in array {

  print("Found \(item)")

}
```

That will print "Found Apples", "Found Peaches" then "Found Plums" to the Xcode console. Each time the loop goes around, one item is read from the array and placed into the constant **item** – and note that *is* a constant, so don't try to change it.

How to shuffle an array in iOS 8 and below
Availability: iOS 2.0 or later.

Nate Cook wrote a simple **shuffle()** extension to arrays that implements the Fisher-Yates shuffle algorithm in Swift. I use it a lot, or at least did until GameplayKit came along in iOS

9.0 – it has its own shuffle algorithm, and so is preferable.

If you're stuck on iOS 8.0 or below, here's the code:

```
extension Array {

    mutating func shuffle() {

        for i in 0 ..< (count - 1) {

            let j = Int(arc4random_uniform(UInt32(count - i))) + i

            swap(&self[i], &self[j])

        }

    }

}
```

For more information see Hacking with Swift tutorial 35.

How to shuffle an array in iOS 9 using GameplayKit
Availability: iOS 9.0 or later.

iOS 9.0 has a built-in way to shuffle arrays thanks to GameplayKit, and it's a simple one-liner. Here's an example of creating an array of lottery balls and picking six random ones:

```
let lotteryBalls = [Int](1...49)

let shuffledBalls =
GKRandomSource.sharedRandom().arrayByShufflingObjectsInArray(lotteryB
alls)

print(shuffledBalls[0])

print(shuffledBalls[1])

print(shuffledBalls[2])

print(shuffledBalls[3])

print(shuffledBalls[4])
```

```
print(shuffledBalls[5])
```

That uses the system's built-in random number generator, which means it's guaranteed to be in a random state by the time it gets to you.

For more information see Hacking with Swift tutorial 35.

How to sort an array using sort()
Availability: iOS 7.0 or later.

All arrays have a built-in **sort()** method that can be used to sort the array. If the array is simple you can just call it directly, like this:

```
var names = ["Jemima", "Peter", "David", "Kelly", "Isabella"]
names.sort()
```

If you have a custom struct or class and want to sort them arbitrarily, you should call **sort()** using a trailing closure that sorts on a field you specify. Here's an example using an array of custom structs that sorts on a particular property:

```
struct MyCustomStruct {
  var someSortableField: String
}

var customArray = [
  MyCustomStruct(someSortableField: "Jemima"),
  MyCustomStruct(someSortableField: "Peter"),
  MyCustomStruct(someSortableField: "David"),
  MyCustomStruct(someSortableField: "Kelly"),
  MyCustomStruct(someSortableField: "Isabella")
```

```
]
```

```
let sortedArray = customArray.sort { (element1, element2) -> Bool in
    return element1.someSortableField < element2.someSortableField
}
```

If you want to sort the array in place rather than assign it to another value, use **sortInPlace()** like this:

```
customArray.sortInPlace { (element1, element2) -> Bool in
    return element1.someSortableField < element2.someSortableField
}
```

How to tell if an array contains an object

Availability: iOS 7.0 or later.

It's easy to find out whether an array contains a specific value, because Swift has a **contains()** method that returns true or false depending on whether that item is found. For example:

```
let array = ["Apples", "Peaches", "Plums"]

if array.contains("Apples") {
    print("We've got apples!")
} else {
    print("No apples here – sorry!")
}
```

That example array does indeed contain "Apples" so that code will print "We've got apples!" to the Xcode console.

CALayer

How to add a border outline color to a UIView

Availability: iOS 3.2 or later.

Any **UIView** subclass has a built-in way to draw a border around it using its underlying **CALayer**. For example, to draw a 10-point black border around a view, you'd use this:

```
yourView.layer.borderWidth = 10
yourView.layer.borderColor = UIColor.blackColor().CGColor
```

Note that you need to use the **CGColor** property of your **UIColor** in order for this to work. Adding a border to a view even works if you also round the corners of your view – it's very flexible!

How to create keyframe animations using CAKeyframeAnimation

Availability: iOS 2.0 or later.

Keyframe animations offer extraordinary power for developers because they let you set multiple values and have iOS animate between them over times you specify. There are three components: a key path (the property to animate), an array of values (the value you want to use for that property), and an array of key times (when that value should be used for the property).

The number of key times needs to match the number of values, because each value is applied in order when its key time is reached. In the example code below, a view will be moved down 300 points then back to its starting point over 2 seconds. It's important that you understand the key times and duration are separate: the key times should be between 0 and 1, where 0 means "the start of the animation" and 1 means "the end of the animation."

```
let animation = CAKeyframeAnimation()
animation.keyPath = "position.y"
animation.values = [0, 300, 0]
animation.keyTimes = [0, 0.5, 1]
animation.duration = 2
animation.additive = true


vw.layer.addAnimation(animation, forKey: "move")
```

Because the animation is marked as additive, it means that 300 is relative to its starting position.

We can use key frame animations to create a simple shake effect that moves a view left and right across a brief animation. This will use additive animations again because we want to specify relative values (move to the left and right a bit) rather than absolute values:

```
func shakeView(vw: UIView) {
  let animation = CAKeyframeAnimation()
  animation.keyPath = "position.x"
  animation.values = [0, 10, -10, 10, -5, 5, -5, 0 ]
  animation.keyTimes = [0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1]
  animation.duration = 0.4
  animation.additive = true
```

```
    vw.layer.addAnimation(animation, forKey: "shake")

}
```

How to draw color gradients using CAGradientLayer
Availability: iOS 3.0 or later.

I love **CAGradientLayer** because it takes just four lines of code to use, and yet looks great because it quickly and accurately draws smooth color gradients use Core Graphics. Here's a basic example:

```
let layer = CAGradientLayer()

layer.frame = CGRect(x: 64, y: 64, width: 160, height: 160)

layer.colors = [UIColor.redColor().CGColor,
UIColor.blackColor().CGColor]

view.layer.addSublayer(layer)
```

Note that you need to fil in an array of **colors** that will be used to draw the gradient. You can provide more than one if you want to, at which point you will also need to fill in the **locations** array to tell **CAGradientLayer** where each color starts and stops. Note that you need to specify your colors as **CGColor** and not **UIColor**.

If you want to make your gradient work in a different direction, you should set the **startPoint** and **endPoint** properties. These are both **CGPoints** where the X and Y values are between 0 and 1, where 0 is one edge and 1 is the opposite edge. The default start point is X 0.5, Y 0.0 and the default end point is X 0.5, Y 1.0, which means both points are in the center of the layer, but it starts at the top and ends at the bottom.

You might be interested to know that **CAGradientLayer** happily works with translucent colors, meaning that you can make a gradient that fades out.

How to draw shapes using CAShapeLayer
Availability: iOS 3.0 or later.

There are lots of **CALayer** subclasses out there, but **CAShapeLayer** is one of my favorites: it provides hardware-accelerated drawing of all sorts of 2D shapes, and includes extra functionality such as fill and stroke colors, line caps, patterns and more.

To get you started, this uses **UIBezierPath** to create a rounded rectangle, which is then colored red using **CAShaperLayer**. Remember, **CALayer** sits underneath UIKit, so you need to use **CGColor** rather than **UIColor**.

```
let layer = CAShapeLayer()

layer.path = UIBezierPath(roundedRect: CGRect(x: 64, y: 64, width:
160, height: 160), cornerRadius: 50).CGPath

layer.fillColor = UIColor.redColor().CGColor

view.layer.addSublayer(layer)
```

How to emit particles using CAEmitterLayer

Availability: iOS 5.0 or later.

Believe it or not, iOS has a built-in particle system that works great in all UIKit apps and is immensely customisable. To get started you need to create a **CAEmitterLayer** object and tell it how to create particles: where it should create them, how big the emitter should be, and what types of particles should exist.

The "type of particles" part is handled by **CAEmitterCell**, which covers details like how fast to create, how long they should live, whether they should spin and/or fade out, what texture to use, and more. You can add as many **CAEmitterCells** to a **CAEmitterLayer** as you need.

Here's some example code to get you started. This creates particles of three different colors, all falling and spinning down from the top of the screen. The image "particle_confetti" is just a small white triangle that I drew by hand – you should replace that with something more interesting.

```
func createParticles() {
```

```swift
    let particleEmitter = CAEmitterLayer()

    particleEmitter.emitterPosition = CGPoint(x: view.center.x, y: -96)
    particleEmitter.emitterShape = kCAEmitterLayerLine
    particleEmitter.emitterSize = CGSize(width: view.frame.size.width,
height: 1)

    let red = makeEmitterCellWithColor(UIColor.redColor())
    let green = makeEmitterCellWithColor(UIColor.greenColor())
    let blue = makeEmitterCellWithColor(UIColor.blueColor())

    particleEmitter.emitterCells = [red, green, blue]

    view.layer.addSublayer(particleEmitter)
}

func makeEmitterCellWithColor(color: UIColor) -> CAEmitterCell {
    let cell = CAEmitterCell()
    cell.birthRate = 3
    cell.lifetime = 7.0
    cell.lifetimeRange = 0
    cell.color = color.CGColor
    cell.velocity = 200
    cell.velocityRange = 50
    cell.emissionLongitude = CGFloat(M_PI)
    cell.emissionRange = CGFloat(M_PI_4)
    cell.spin = 2
    cell.spinRange = 3
    cell.scaleRange = 0.5
    cell.scaleSpeed = -0.05
```

```
    cell.contents = UIImage(named: "particle_confetti")?.CGImage

    return cell

}
```

How to round the corners of a UIView

Availability: iOS 3.2 or later.

All **UIView** subclasses have the ability to round their corners thanks to their underlying **CALayer** – that's the bit that handles the actual drawing of your views. To round the corners of a view, use this code:

```
yourView.layer.cornerRadius = 10
```

The number you specify is how far the rounding should go, measured in points. This means that if you have a view that's 128x128 points wide and give it a **cornerRadius** property of 64, it will look like a circle.

Note that some types of view don't have **clipsToBounds** enabled by default, which means their corners will not round until you enable this property.

Core Graphics

How to calculate the Manhattan distance between two CGPoints

Availability: iOS 2.0 or later.

Manhattan distance is the distance between two integer points when you are unable to move diagonally. It's named "Manhattan distance" because of the grid-like layout of New York:

whether you go four streets up then five streets across, or five streets across then four streets up, or you zig zag to and fro, the actual end distance is identical because you're just moving across a grid.

If you want to calculate Manhattan distance in your own code, just drop in this function:

```
func CGPointManhattanDistance(from from: CGPoint, to: CGPoint) ->
CGFloat {
  return (abs(from.x - to.x) + abs(from.y - to.y));
}
```

How to calculate the distance between two CGPoints

Availability: iOS 2.0 or later.

You can calculate the distance between two **CGPoints** by using Pythagoras's theorem, but be warned: calculating square roots is not fast, so if possible you want to avoid it. More on that in a moment, but first here's the code you need:

```
func CGPointDistanceSquared(from from: CGPoint, to: CGPoint) ->
CGFloat {
  return (from.x - to.x) * (from.x - to.x) + (from.y - to.y) *
(from.y - to.y);
}


func CGPointDistance(from from: CGPoint, to: CGPoint) -> CGFloat {
  return sqrt(CGPointDistanceSquared(from: from, to: to));
}
```

Note that there are two functions: one for returning the distance between two points, and one for returning the distance squared between two points. The latter one doesn't use a square root, which makes it substantially faster. This means if you want to check "did the

user tap within a 10-point radius of this position?" it's faster to square that 10 (to make 100) then use **CGPointDistanceSquared()** instead.

How to compare two CGRects with CGRectEqualToRect()
Availability: iOS 2.0 or later.

You could compare two **CGRect** values by evaluating their X, Y, width and height values, but there's a much faster way: **CGRectEqualToRect()**. This takes two rects as its only two parameters and returns true if they are the same, or false otherwise.

Here's an example:

```
let rect1 = CGRect(x: 64, y: 64, width: 128, height: 128)

let rect2 = CGRect(x: 256, y: 256, width: 128, height: 128)


if CGRectEqualToRect(rect1, rect2) {

  // rects equal!

} else {

  // rects not equal

}
```


How to draw a circle using Core Graphics: CGContextAddEllipseInRect
Availability: iOS 4.0 or later.

Core Graphics is able to draw circles and ellipses with just a few lines of code, although there is some set up to do first. The example code below creates a 512x512 circle with a red fill and a black border:

```
UIGraphicsBeginImageContextWithOptions(CGSize(width: 512, height:
512), false, 0)
```

```
let context = UIGraphicsGetCurrentContext()

let rectangle = CGRect(x: 0, y: 0, width: 512, height: 512)


CGContextSetFillColorWithColor(context, UIColor.redColor().CGColor)

CGContextSetStrokeColorWithColor(context,
UIColor.blackColor().CGColor)

CGContextSetLineWidth(context, 10)


CGContextAddEllipseInRect(context, rectangle)

CGContextDrawPath(context, .FillStroke)


let img = UIGraphicsGetImageFromCurrentImageContext()

UIGraphicsEndImageContext()
```

Please note: although a 10-point border is specified, Core Graphics draws borders half-way inside and half-way outside the path you create, so if you want to see the whole border (rather than have it cropped) you either need to draw a smaller shape or create a bigger context.

For more information see Hacking with Swift tutorial 27.

How to draw a square using Core Graphics: CGContextAddRect()

Availability: iOS 4.0 or later.

You can draw a square (or indeed any size of rectangle) using the **CGContextAddRect()** Core Graphics function. There's a little bit of set up work required, such as creating a context big enough to hold the square and setting up colors, but the code below does everything you need:

```
UIGraphicsBeginImageContextWithOptions(CGSize(width: 512, height:
512), false, 0)
```

```swift
let context = UIGraphicsGetCurrentContext()

let rectangle = CGRect(x: 0, y: 0, width: 512, height: 512)


CGContextSetFillColorWithColor(context, UIColor.redColor().CGColor)

CGContextSetStrokeColorWithColor(context,
UIColor.blackColor().CGColor)

CGContextSetLineWidth(context, 10)


CGContextAddRect(context, rectangle)

CGContextDrawPath(context, .FillStroke)


let img = UIGraphicsGetImageFromCurrentImageContext()

UIGraphicsEndImageContext()
```

Important note: when setting a line width using **CGContextSetLineWidth()**, the center of the border is the edge of your path. This means our board will be 5 points inside the rectangle and 5 points outside, because we have a total border width of 10 points. Because the square's path is the same size as the context, this means the outside part of the border will be clipped.

For more information see Hacking with Swift tutorial 27.

How to draw a text string using Core Graphics
Availability: iOS 4.0 or later.

To draw text in Core Graphics is trivial because every Swift string has a built-in **drawWithRect()** method that takes an array of attributes and a position and size. There is, like always, some Core Graphics set up work to do, but this next code snippet is a complete example you can re-use easily:

```swift
UIGraphicsBeginImageContextWithOptions(CGSize(width: 512, height:
512), false, 0)
```

```
let paragraphStyle = NSMutableParagraphStyle()

paragraphStyle.alignment = .Center


let attrs = [NSFontAttributeName: UIFont(name: "HelveticaNeue-Thin",
size: 36)!, NSParagraphStyleAttributeName: paragraphStyle]


let string = "How much wood would a woodchuck\nchuck if a woodchuck
would chuck wood?"

string.drawWithRect(CGRect(x: 32, y: 32, width: 448, height: 448),
options: .UsesLineFragmentOrigin, attributes: attrs, context: nil)


let img = UIGraphicsGetImageFromCurrentImageContext()

UIGraphicsEndImageContext()
```

For more information see Hacking with Swift tutorial 27.

How to draw lines in Core Graphics: CGContextMoveToPoint() and CGContextAddLineToPoint()

Availability: iOS 4.0 or later.

You can draw lines in Core Graphics using **CGContextMoveToPoint()** and
CGContextAddLineToPoint(). The first function moves the Core Graphics path to a **CGPoint**
of your choosing, and the second function moves the path to a new point while also adding a
line. Once you add in the required code to set up a context and choose a color, you can draw
a triangle with this code:

```
UIGraphicsBeginImageContextWithOptions(CGSize(width: 500, height:
500), false, 0)

let context = UIGraphicsGetCurrentContext()


CGContextMoveToPoint(context, 50, 450)
```

```
CGContextAddLineToPoint(context, 250, 50)

CGContextAddLineToPoint(context, 450, 450)

CGContextAddLineToPoint(context, 50, 450)


CGContextSetStrokeColorWithColor(context,
UIColor.blackColor().CGColor)

CGContextStrokePath(context)


let img = UIGraphicsGetImageFromCurrentImageContext()

UIGraphicsEndImageContext()
```

Once you've mastered drawing basic lines, you can create neat effects by rotating the context as you draw, like this:

```
UIGraphicsBeginImageContextWithOptions(CGSize(width: 512, height:
512), false, 0)

let context = UIGraphicsGetCurrentContext()

CGContextTranslateCTM(context, 256, 256)


var first = true

var length: CGFloat = 256


for i in 0 ..< 256 {

    CGContextRotateCTM(context, CGFloat(M_PI_2))


    if first {

        CGContextMoveToPoint(context, length, 50)

        first = false

    } else {

        CGContextAddLineToPoint(context, length, 50)
```

```
    }


    length *= 0.99

}


CGContextSetStrokeColorWithColor(context,
UIColor.blackColor().CGColor)

CGContextStrokePath(context)


let img = UIGraphicsGetImageFromCurrentImageContext()

UIGraphicsEndImageContext()
```

For more information see Hacking with Swift tutorial 27.

How to find the rotation from a CGAffineTransform

Availability: iOS 2.0 or later.

A **CGAffineTransform** value combines scale, translation and rotation all at once, but if you just want to know its rotation value is then use this code:

```
func rotationFromTransform(transform: CGAffineTransform) -> Double {

    return atan2(Double(transform.b), Double(transform.a));

}
```


How to find the scale from a CGAffineTransform

Availability: iOS 2.0 or later.

If you have a **CGAffineTransform** and want to know what its scale component is –
regardless of whether it has been rotated or translated – use this code:

```
func scaleFromTransform(transform: CGAffineTransform) -> Double {

  return sqrt(Double(transform.a * transform.a + transform.c *
transform.c));

}
```

How to find the translation from a CGAffineTransform

Availability: iOS 2.0 or later.

You can pull out the translation from a **CGAffineTransform** by using the function below.
Feed it a transform and it will return you a **CGPoint**:

```
func translationFromTransform(transform: CGAffineTransform) ->
CGPoint {

  return CGPointMake(transform.tx, transform.ty);

}
```

How to render a PDF to an image

Availability: iOS 3.0 or later.

iOS has built-in APIs for drawing PDFs, which means it's relatively straight forward to render
a PDF to an image. I say "relatively" because there's still some boilerplate you need to worry
about: figuring out the document size, filling the background in a solid color to avoid
transparency, and flipping the rendering so that the PDF draws the right way up.

To make things easy for you, here's a pre-made method you can use that takes a URL to a
PDF and returns either a rendered image or nil if it failed. To call it you should pull out the
URL to a resource in your bundle or another local PDF file.

```
func drawPDFfromURL(url: NSURL) -> UIImage? {
```

```
    guard let document = CGPDFDocumentCreateWithURL(url) else { return
nil }
    guard let page = CGPDFDocumentGetPage(document, 1) else { return
nil }


    let pageRect = CGPDFPageGetBoxRect(page, .MediaBox)


    UIGraphicsBeginImageContextWithOptions(pageRect.size, true, 0)
    let context = UIGraphicsGetCurrentContext()


    CGContextSetFillColorWithColor(context,
UIColor.whiteColor().CGColor)
    CGContextFillRect(context,pageRect)


    CGContextTranslateCTM(context, 0.0, pageRect.size.height);
    CGContextScaleCTM(context, 1.0, -1.0);


    CGContextDrawPDFPage(context, page);
    let img = UIGraphicsGetImageFromCurrentImageContext()


    UIGraphicsEndImageContext()


    return img
}
```

How to use Core Graphics blend modes to draw a UIImage differently

Availability: iOS 2.0 or later.

If you're rendering images using Core Graphics you should definitely try out some of the
alternate blend modes that are available. If you've ever used Photoshop's blend modes

these will be familiar: screen, luminosity, multiply and so on – these are all available right in Core Graphics.

To give you an idea what's possible, here's some code that takes two UIImages and draws them into one single image. The first image is drawn using normal rendering, and the second using **.Luminosity**.

```
if let img = UIImage(named: "example"), img2 = UIImage(named:
"example2") {
  let rect = CGRect(x: 0, y: 0, width: img.size.width, height:
img.size.height)

  UIGraphicsBeginImageContextWithOptions(img.size, true, 0)
  let context = UIGraphicsGetCurrentContext()

  // fill the background with white so that translucent colors get
lighter
  CGContextSetFillColorWithColor(context,
UIColor.whiteColor().CGColor)
  CGContextFillRect(context, rect)

  img.drawInRect(rect, blendMode: .Normal, alpha: 1)
  img2.drawInRect(rect, blendMode: .Luminosity, alpha: 1)

  // grab the finished image and return it
  let result = UIGraphicsGetImageFromCurrentImageContext()
  UIGraphicsEndImageContext()
}
```

How that looks depends on the source images you used – try drawing them the other way around to see what difference it makes, or try using **.Multiply** rather than **.Luminosity**.

If you're looking for a more advanced example, this function accepts an image and returns

the same image with a rainbow effect to it. This is done by drawing six colored strips onto an image, then overlaying the original image using the blend mode **.Luminosity** along with a slight alpha.

```
func addRainbowToImage(img: UIImage) -> UIImage {
    // create a CGRect representing the full size of our input iamge
    let rect = CGRect(x: 0, y: 0, width: img.size.width, height:
img.size.height)


    // figure out the height of one section (there are six)
    let sectionHeight = img.size.height / 6


    // set up the colors – these are based on my trial and error
    let red = UIColor(red: 1, green: 0.5, blue: 0.5, alpha:
0.8).CGColor
    let orange = UIColor(red: 1, green: 0.7, blue: 0.35, alpha:
0.8).CGColor
    let yellow = UIColor(red: 1, green: 0.85, blue: 0.1, alpha:
0.65).CGColor
    let green = UIColor(red: 0, green: 0.7, blue: 0.2, alpha:
0.5).CGColor
    let blue = UIColor(red: 0, green: 0.35, blue: 0.7, alpha:
0.5).CGColor
    let purple = UIColor(red: 0.3, green: 0, blue: 0.5, alpha:
0.6).CGColor
    let colors = [red, orange, yellow, green, blue, purple]


    // set up our drawing context
    UIGraphicsBeginImageContextWithOptions(img.size, true, 0)
    let context = UIGraphicsGetCurrentContext()


    // fill the background with white so that translucent colors get
lighter
```

```
  CGContextSetFillColorWithColor(context,
UIColor.whiteColor().CGColor)

  CGContextFillRect(context, rect)


  // loop through all six colors

  for i in 0 ..< 6 {

    let color = colors[i]


    // figure out the rect for this section

    let rect = CGRect(x: 0, y: CGFloat(i) * sectionHeight, width:
rect.width, height: sectionHeight)


    // draw it onto the context at the right place

    CGContextSetFillColorWithColor(context, color)

    CGContextFillRect(context, rect)

  }


  // now draw our input image over using Luminosity mode, with a
little bit of alpha to make it fainter

  img.drawInRect(rect, blendMode: .Luminosity, alpha: 0.6)


  // grab the finished image and return it

  let img = UIGraphicsGetImageFromCurrentImageContext()

  UIGraphicsEndImageContext()


  return img

}
```

Games

How to add physics to an SKSpriteNode
Availability: iOS 7.0 or later.

SpriteKit comes with a modified version of the Box2D physics framework, and it's wrapped up a lot of complicated physics mathematics into just one or two lines of code. For example, we can create a square, red sprite and give it rectangular physics like this:

```
let box = SKSpriteNode(color: UIColor.redColor(), size: CGSize(width: 64, height: 64))

box.physicsBody = SKPhysicsBody(rectangleOfSize: CGSize(width: 64, height: 64))
```

That rectangle will wrap perfectly around the box's color, so it will bounce and rotate as it collides with other objects in your scene.

If you want to create circular physics to simulate balls, this is done using the **circleOfRadius** constructor:

```
let ball = SKSpriteNode(imageNamed: "ballRed")

ball.physicsBody = SKPhysicsBody(circleOfRadius: ball.size.width / 2.0)
```

For more information see Hacking with Swift tutorial 11.

How to add pixel-perfect physics to an SKSpriteNode
Availability: iOS 7.0 or later.

Pixel-perfect physics is just one line of code in SpriteKit. Don't believe me? Here you go:

```
player = SKSpriteNode(imageNamed: "player")

player.position = CGPoint(x: 100, y: 384)

player.physicsBody = SKPhysicsBody(texture: player.texture, size:
player.size)
```

That last line is the one that does the magic: SpriteKit will use the alpha values of your sprite
(i.e., the transparent pixels) to figure out which parts should be part of a collision.

As you might imagine, pixel-perfect collision detection is significantly slower than using
rectangles or circles, so you should use it carefully.

For more information see Hacking with Swift tutorial 23.

How to color an SKSpriteNode using colorBlendFactor
Availability: iOS 7.0 or later.

One powerful and under-used feature of SpriteKit is its ability to recolor **SKSpriteNodes**
dynamically. This has almost zero performance impact, which makes it perfect for having
multiple-colored enemies or players. It can also be animated, meaning that you could for
example make your player flash white briefly when they've been hit by a bad guy.

To tint a sprite cyan, use this code:

```
firework.color = UIColor.cyanColor()

firework.colorBlendFactor = 1
```

If you want to animate the sprite coloring, you'd use this:

```
let action = SKAction.colorizeWithColor(UIColor.redColor(),
colorBlendFactor: 1, duration: 1)
```

For more information see Hacking with Swift tutorial 20.

How to create 3D audio sound using SKAudioNode
Availability: iOS 9.0 or later.

3D audio is a feature where a sound is dynamically altered so that listeners think it comes from a particular location. Obviously they are looking at a flat 2D screen ahead of them, but using some clever mathematics iOS can make sounds "feel" like they are behind you, or at a more basic level adjust the panning so that sounds come from the left or right of the user's audio device.

As of iOS 9.0, you get these features for free: all you need to do is create an **SKAudioNode** for your sound and set its **positional** property to be **true**. That's it – iOS will automatically use the position of the node to adjust the way its audio sounds, and it even adjusts the audio as you move it around.

To give you a working example, this creates an audio node from a file called music.m4a (you'll need to provide that), then makes the audio move left and right forever. If you listen to this using headphones (which is the only effective way for 3D sound to work on iOS devices) you'll really hear a pronounced panning effect.

```
override func didMoveToView(view: SKView) {
  let music = SKAudioNode(fileNamed: "music.m4a")
  addChild(music)

  music.positional = true
  music.position = CGPoint(x: -1024, y: 0)

  let moveForward = SKAction.moveToX(1024, duration: 2)
  let moveBack = SKAction.moveToX(-1024, duration: 2)
  let sequence = SKAction.sequence([moveForward, moveBack])
  let repeatForever = SKAction.repeatActionForever(sequence)
```

```
    music.runAction(repeatForever)
}
```

How to create a SpriteKit texture atlas in Xcode

Availability: iOS 7.0 or later.

A SpriteKit texture atlas is actually just a folder with the extension .atlas, but it's more efficient than loading textures individually because multiple images are stored in a single file and thus can be loaded faster. Even better, you don't need to worry about how they are placed or even orientation – you just use them as normal, and SpriteKit does the rest.

In Finder, go into your project directory (where your .swift files are), then create a new folder called assets.atlas. Now go to where you have your SpriteKit assets stored and drag them from there into your assets.atlas directory. Finally, drag your assets.atlas directory into your Xcode project so that it gets added to the build.

That's it – enjoy your efficiency improvements!

For more information see Hacking with Swift tutorial 29.

How to create shapes using SKShapeNode

Availability: iOS 7.0 or later.

SpriteKit's **SKShapeNode** class is a fast and convenient way to draw arbitrary shapes in your games, including circles, lines, rounded rectangles and more. You can assign a fill color, a stroke color and width, plus other drawing options such as whether it should glow – yes, really.

The example code below draws a rounded rectangle smack in the middle of the game scene, giving it a red fill color and a 10-point blue border:

```
let shape = SKShapeNode()
shape.path = UIBezierPath(roundedRect: CGRect(x: -128, y: -128,
```

```
width: 256, height: 256), cornerRadius: 64).CGPath

shape.position = CGPoint(x: CGRectGetMidX(frame), y:
CGRectGetMidY(frame))

shape.fillColor = UIColor.redColor()

shape.strokeColor = UIColor.blueColor()

shape.lineWidth = 10

addChild(shape)
```

For more information see Hacking with Swift tutorial 17.

How to emit particles using SKEmitterNode

Availability: iOS 7.0 or later.

SpriteKit has built-in support for particle systems, which are a realistic and fast way to create effects such as smoke, fire and snow. Even better, Xcode has a built-in visual particle editor so that you can tweak your designs until they look exactly right.

To get started, right-click on your project in Xcode and choose New File. Select iOS > Resource > SpriteKit Particle File, then choose the Smoke template and click Next to name your effect. Once that's done, your particle will be opened immediately in the visual editor so you can adjust its design.

When it comes to using your effect, just create a new **SKEmitterNode** object using the name of your particle effect, like this:

```
if let particles = SKEmitterNode(fileNamed: "yourParticleFile.sks") {
  particles.position = player.position
  addChild(particles)
}
```

Obviously you will want to set your own position rather than using an example **player** node.

For more information see Hacking with Swift tutorial 11.

How to generate a random number with GKRandomSource

Availability: iOS 9.0 or later.

GameplayKit is a powerful new framework introduced in iOS 9.0, and one of the (many!) things it does is provide a number of ways to generate random numbers easily. To get started, import the framework into your code like this:

```
import GameplayKit
```

You can immediately start generating random numbers just by using this code:

```
print(GKRandomSource.sharedRandom().nextInt())
```

That code produces a number between -2,147,483,648 and 2,147,483,647, so if you're happy with negative numbers then you're basically done. Alternatively, if you want a random number between and an upper bound (inclusive!), you should use this code instead:

```
print(GKRandomSource.sharedRandom().nextIntWithUpperBound(10))
```

That will return a number between 0 and 10, including 0 and 10 themselves.

For more information see Hacking with Swift tutorial 35.

How to roll a dice using GameplayKit and GKRandomDistribution

Availability: iOS 9.0 or later.

GameplayKit's random number generator includes help constructors that produces numbers in a specific range, simulating a six-sided die and a 20-sided die. To get started you should import the GameplayKit framework like this:

```
import GameplayKit
```

You can then create a virtual six-sided die and "roll" it like this:

```
let d6 = GKRandomDistribution.d6()
d6.nextInt()
```

For a 20-sided die, do this:

```
let d20 = GKRandomDistribution.d20()
d20.nextInt()
```

If you want to set your own range for the virtual die, there's a special constructor just for you:

```
let massiveDie = GKRandomDistribution(lowestValue: 1, highestValue: 556)
massiveDie.nextInt()
```

For more information see Hacking with Swift tutorial 35.

How to run SKActions in a group

Availability: iOS 7.0 or later.

SpriteKit action groups let you run multiple SpriteKit actions simultaneously. The grouped

actions become a new action that can go into a sequence, and SpriteKit automatically ensures all actions in a group finish before the sequence continues.

The code below makes a spaceship shrink down to 10% of its original size while fading out, with both actions happening at the same time:

```
let sprite = SKSpriteNode(imageNamed:"Spaceship")


let scale = SKAction.scaleTo(0.1, duration: 0.5)

let fade = SKAction.fadeOutWithDuration(0.5)

let group = SKAction.group([scale, fade])


sprite.runAction(group)
```

For more information see Hacking with Swift tutorial 17.

How to run SKActions in a sequence
Availability: iOS 7.0 or later.

One of the great features of SpriteKit's actions is that they can be chained together using action sequences. SpriteKit automatically ensures each action finishes before the next one begins – all you need to do is create the actions then put them into an array.

The example below makes a spaceship shrink down to 10% of its original size before fading out:

```
let sprite = SKSpriteNode(imageNamed:"Spaceship")


let scale = SKAction.scaleTo(0.1, duration: 0.5)

let fade = SKAction.fadeOutWithDuration(0.5)

let sequence = SKAction.sequence([scale, fade])
```

```
sprite.runAction(sequence)
```

For more information see Hacking with Swift tutorial 14.

How to stop an SKPhysicsBody responding to physics using its dynamic property

Availability: iOS 7.0 or later.

Enabling physics in SpriteKit is just one line of code, but sometimes you want your physics to be a little more nuanced. For example, your player might have circle physics and should respond to gravity, whereas walls might have rectangle physics and not respond to gravity – they are there to be bounced off, but nothing more.

This problem is solved in SpriteKit by using the **dynamic** property. It's **true** by default, which means that your objects respond to the world's environment as you would expect, but if you set it to be **false** then you get an object that has active physics but doesn't move as a result of those physics.

Here's an example:

```
let wall = SKSpriteNode(imageNamed: "wall")

wall.position = CGPoint(x: 512, y: 0)

wall.physicsBody = SKPhysicsBody(circleOfRadius: wall.size.width /
2.0)

wall.physicsBody!.dynamic = false

addChild(wall)
```

For more information see Hacking with Swift tutorial 11.

How to write text using SKLabelNode

Availability: iOS 7.0 or later.

The **SKLabelNode** class is a fast and efficient way to draw text in SpriteKit games. To use it, first create a property in your game scene:

```
var scoreLabel: SKLabelNode!
```

Now create the label node by telling it want font use, its alignment, and also an initial text value if you want one. This code creates a label node using the Chalkduster font, places it in the top-right corner of the screen, and gives it the initial text "Score: 0":

```
scoreLabel = SKLabelNode(fontNamed: "Chalkduster")
scoreLabel.text = "Score: 0"
scoreLabel.horizontalAlignmentMode = .Right
scoreLabel.position = CGPoint(x: 980, y: 700)
addChild(scoreLabel)
```

With that score label in place, you can now create a **score** integer property to store the actual number of a player's score, then use a property observer to modify the label whenever the score changes:

```
var score: Int = 0 {
    didSet {
        scoreLabel.text = "Score: \(score)"
    }
}
```

For more information see Hacking with Swift tutorial 11.

locationInNode(): How to find a touch's location in a node

Availability: iOS 7.0 or later.

It's just one line of code to find where the user touched the screen when you're using SpritKit, and that one line can even be used to calculative relative positions of a touch compared to any node in your game.

To get started, you should implement **touchesBegan()** in your SpriteKit node or scene. This will get called when the user starts touching the node, regardless of where on the node. To locate the exact position, call **locationInNode()** on any **UITouch**, passing in the node you want to check, like this:

```
override func touchesBegan(touches: Set<UITouch>, withEvent event:
UIEvent?) {

  if let touch = touches.first {

    let location = touch.locationInNode(self)

    print(location)

  }

}
```

For more information see Hacking with Swift tutorial 11.

presentScene(): How to change SKScene with a transition

Availability: iOS 7.0 or later.

You can change between SpriteKit scenes by calling the **presentScene()** method on your **SKView**. This can be called either just with a new scene, or with a new scene and a transition animation to use, depending on the effect you want. Here's an example with a transition:

```
let scene = NewGameScene(fileNamed: "NewGameScene")!
let transition = SKTransition.moveInWithDirection(.Right, duration:
```

```
1)
self.view?.presentScene(scene, transition: transition)
```

There are several beautiful transition types you can try, with the
SKTransition.doorwayWithDuration(1) transition looking particularly neat.

For more information see Hacking with Swift tutorial 29.

Language

How to check for valid method input using the guard keyword
Availability: iOS 7.0 or later.

The **guard** keyword was introduced in Swift to signal early returns, which is a coding
technique that effectively means "make sure all these things are set up before I start doing
the real work in my function, others bail out."

For example, if you want to ensure a **submit()** is only ever run if an existing **name** property
has a value, you would do this:

```
func submit() {
  guard name != nil else { return }

  doImportantWork(name)
}
```

This might seem like a job for a regular **if** statement, and to be fair that's correct – the two are
very similar. The advantage with **guard**, however, is that it makes your intention clear: these

values need to be set up correctly before continuing.

The **guard** keyword is also helpful because it can be used to check and unwrap optionals that remain unwrapped until the end of the method. For example:

```
func submit() {
  guard let unwrappedName = name else { return }

  doImportantWork(unwrappedName)
}
```

So, if **name** is **nil** the method will return; otherwise, it will be safely unwrapped into **unwrappedName**.

How to convert a float to a CGFloat

Availability: iOS 7.0 or later.

The **Float** and **CGFloat** data types sound so similar you might think they were identical, but they aren't: **CGFloat** is flexible in that its precision adapts to the type of device it's running on, whereas **Float** is always a fixed precision. Thus, you never lose precision converting from **Float** to **CGFloat**, whereas you might going the other way.

To convert, just use the **CGFloat** constructor, like this:

```
let myCGFloat = CGFloat(myFloat)
```

How to convert a float to an int

Availability: iOS 7.0 or later.

You can convert between a **Float** and an **Int** just by using the integer's constructor, like this:

```
let myFloat: Float = 10.756
let myInt = Int(myFloat)
```

Note that when you do this, your number will automatically be rounded downwards. In the example above, the integer will be 10, not 11.

How to convert a string to a double
Availability: iOS 7.0 or later.

Swift strings don't have a built-in way to convert to a **Double**, but their **NSString** counterparts do. To convert between strings and doubles, just do this:

```
let myString = "556"
let myFloat = (myString as NSString).doubleValue
```

How to convert a string to a float
Availability: iOS 7.0 or later.

There are several ways to convert between a string and a **Float**, but the easiest way is to use **NSString** as an intermediate because that comes with several helpers built right in:

```
let myString = "556"
let myFloat = (myString as NSString).floatValue
```

How to convert a string to an NSString

Availability: iOS 7.0 or later.

When Swift originally launched, NSString (older iOS strings) and native Swift strings were completely interchangeable, as were NSArray and Swift arrays, plus NSDictionary and Swift dictionaries. This got changed in Swift 1.2 so that you need to explicitly cast between these data types, and this remains the same in Swift 2.0.

So, to cast between Swift strings and NSString, you need to do a simple typecast like this:

```
let str = "Hello"
let otherStr = str as NSString
```

Note that you don't need to force the typecast because the two data types are still interoperable.

How to convert a string to an int

Availability: iOS 7.0 or later.

If you have an integer hiding inside a string, you can convert between the two just by using the integer's constructor, like this:

```
let myString = "556"
let myInt = Int(myString)
```

As with other data types (**Float** and **Double**) you can also convert by using **NSString**:

```
let myString = "556"
let myInt = (myString as NSString).integerValue
```

How to convert an int to a float

Availability: iOS 7.0 or later.

Swift's **Float** data type has a built-in constructor that can convert from integers with no extra work from you. For example, to convert the integer 556 into its **Float** equivalent, you'd use this:

```
let myInt = 556
let myFloat = Float(myInt)
```

How to convert an int to a string

Availability: iOS 7.0 or later.

Swift's string interpolation means you can convert all sorts of data – including integers – to a string in just one line of code:

```
let str = "\(myInt)"
```

However, the more common way is just to use the string constructor, like this:

```
let str = String(myInt)
```

How to create an Objective C bridging header to use code in Swift

Availability: iOS 7.0 or later.

If you want to use Objective-C code in your Swift app – and let's face it, that's going to happen quite a lot! – then you need to create a bridging header that allows your Swift code to work with your Objective-C code.

To create an Objective-C bridging header file, all you need to do is drag some Objective-C code into your Swift project – Xcode should prompt you with the message "Would you like to configure an Objective-C bridging header?" Click "Creating Bridging Header" and you'll see a file called **YourProjectName-Bridging-Header.h** appear in your project.

But that's only half the problem: Xcode has created the bridging header and modified your build settings so that it gets used, but it hasn't actually put anything into it. If you want to start using your Objective-C code in Swift, you need to add import lines to that bridging header file, like this:

```
#import "YourFile.h"
```

You can add as many of these as you want, and indeed you'll want to import all the Objective-C code you want to use in Swift.

How to delay execution of code using the defer keyword

Availability: iOS 7.0 or later.

The **defer** keyword is new in Swift 2 and lets you schedule some code to be run at a later date. That later date is when your code exits its current scope, which might be when a function returns or at the end of a loop, for example.

If you've used other programming languages, **defer** will seem similar to **try/finally**. Any code you defer will run no matter what, even if you throw an exception.

In the example code below, the **closeFile()** function will get called no matter how the **writeLog()** function ends:

```
func writeLog() {
    let file = openFile()
    defer { closeFile(file) }

    let hardwareStatus = fetchHardwareStatus()
    guard hardwareStatus != "disaster" else { return }
    file.write(hardwareStatus)

    let softwareStatus = fetchSoftwareStatus()
    guard softwareStatus != "disaster" else { return }
    file.write(softwareStatus)

    let networkStatus = fetchNetworkStatus()
    guard neworkStatus != "disaster" else { return }
    file.write(networkStatus)
}
```

How to find the maximum of three numbers
Availability: iOS 2.0 or later.

The easiest way to find the maximum of three numbers is to use the **max()** function twice: once with your first two numbers, and again with your third number and the result of the first call. Here's an example:

```
let first = 10
```

```
let second = 15

let third = 18


let largest = max(max(first, second), third)
```

How to find the maximum of two numbers

Availability: iOS 2.0 or later.

To find the largest of any two integers, use the **max()** function like this:

```
let first = 10

let second = 15


let largest = max(first, second)
```

This also works with floating-point numbers, as long as both numbers are floats – you can't mix data types.

How to find the minimum of three numbers

Availability: iOS 2.0 or later.

You can find the minimum of three numbers by using the **min()** function twice. This function takes either two integers or two floating-point numbers, but can't accept mixed types. Here's an example:

```
let first = 10

let second = 15

let third = 18
```

```
let smallest = min(min(first, second), third)
```

How to find the minimum of two numbers

Availability: iOS 2.0 or later.

To find the minimum of two numbers, either both integer or both floating point (not mixed!), use the **min()** function. For example:

```
let first = 10

let second = 15


let smallest = min(first, second)
```

How to force your program to crash with assert()

Availability: iOS 7.0 or later.

This might seem like a strange topic – after all, why would anyone want their program to crash? Well, the answer is two-fold.

First, if something has gone wrong that leaves your program in an unsafe state, continuing might mean corrupting user data.

Second, if you're debugging your app (i.e., it's still in development), having your app refuse to continue if a serious problem is found is a huge advantage and a very common way to spot problems.

Swift lets you force an app crash using the **assert()** function. This takes two parameters: a condition to check, and a message to print if the assertion fails. Helpfully, any calls to

assert() are ignored when your app is compiled in release mode (i.e., for the App Store), which means these checks have no impact on your code's final performance.

Here are two examples of **assert()** being used:

```
assert(1 == 1, "Maths failure!")
assert(1 == 2, "Maths failure!")
```

The first one asserts that 1 is equal to 1, which is clearly true, so nothing will happen. The second one asserts that 1 is equal to 2, which is clearly false, so that assertion will fail: your app will halt, and the message "Maths failure!" will be printed out to help you identify the problem.

Because assertions are ignored in release builds, you don't need to worry about running expensive checks in your assertions. For example:

```
assert(myReallySlowMethod() == false, "The slow method returned
false, which is a bad thing!")
```

In release builds, that code will never be run, so you won't see any performance impact.

How to print debug text in Swift
Availability: iOS 7.0 or later.

You can write text to the Xcode debug console using the **print()** function in Swift. In older versions of Swift this was named **println()**, but this changed in Xcode 7 and Swift 2.

How to unwrap an optional in Swift
Availability: iOS 7.0 or later.

Optional values are a central concept in Swift, although admittedly they can be a little hard to understand at first. Put simply, an optional value is one that may or may not exist, which means Swift won't let you use it by accident – you need to either check whether it has a value and unwrap it, or force unwrap. Of the two options the first is definitely preferable, because it's significantly safer.

To check whether an optional has a value then unwrap it all in one, you should use **if/let** syntax, like this:

```
// fetch an example optional string

let optionalString = fetchOptionalString()


// now unwrap it

if let unwrapped = optionalString {

  print(unwrapped)

}
```

In that example, the **print(unwrapped)** line will only be executed if **optionalString** has a value. If that line is reached, you can know for sure that **unwrapped** has a value that you can use, which makes that code safe.

For more information see Hacking with Swift tutorial 1.

How to use #available to check for API availability
Availability: iOS 7.0 or later.

One of my favorite Xcode 7 features is the ability to have Xcode automatically check API availability for you, which means it will refuse to run code that is not available on the minimum iOS version you support.

Of course, there are times when you really do need to use a newer feature, for example if you want to use **UIStackView** where it's available but otherwise show a message to users asking

them to upgrade. For this, Swift has **#available**, which lets you state that a certain block of code should only execute on specific versions of iOS.

To use the previous example, this code checks whether the user has iOS 9.0 or later on their device:

```
if #available(iOS 9, *) {
    // use UIStackView
} else {
    // show sad face emoji
}
```

Any code inside the **// use UIStackView** block can be executed as if your deployment target were iOS 9.0.

If you want, you can mark whole functions or classes as requiring a specific iOS version by using **@available**, like this:

```
@available(iOS 9, *)
func useStackView() {
    // use UIStackView
}
```

How to use compiler directives to detect the iOS Simulator
Availability: iOS 7.0 or later.

Swift makes it easy to write special code that should be executed only in the iOS Simulator. This is helpful to test situations where the simulator and devices don't match, for example testing the accelerometer or camera.

If you want certain code to be run only in the iOS simulator, you should use this:

```
#if (arch(i386) || arch(x86_64))

// your code

#endif
```

Any code between the **#if** and **#endif** won't even exist when the app is run on devices, so it has zero performance impact. If you want to specify alternate code that should only be run on devices (and never on the simulator) you should use **#else**, like this:

```
func updateMotion() {

#if (arch(i386) || arch(x86_64))

  // we're on the simulator - calculate pretend movement

    if let currentTouch = lastTouchPosition {

        let diff = CGPoint(x: currentTouch.x - player.position.x, y:
currentTouch.y - player.position.y)

        physicsWorld.gravity = CGVector(dx: diff.x / 100, dy:
diff.y / 100)

    }

#else

  // we're on a device – use the accelerometer

    if let accelerometerData = motionManager.accelerometerData {

        physicsWorld.gravity = CGVector(dx:
accelerometerData.acceleration.y * -50, dy:
accelerometerData.acceleration.x * 50)

    }

#endif

}
```

For more information see Hacking with Swift tutorial 26.

How to use try/catch in Swift to handle exceptions

Availability: iOS 7.0 or later.

The new try/catch syntax was added in Swift 2.0 to make exception handling clearer and safer. It's made up of three parts: **do** starts a block of code that might fail, **catch** is where execution gets transferred if any errors occur, and any function calls that might fail need to be called using **try**.

Here's a working example that loads an input.txt file from the app bundle into a string:

```
if let filename = NSBundle.mainBundle().pathForResource("input",
ofType: "txt") {

  do {

    let str = try String(contentsOfFile: filename, usedEncoding: nil)

    print(str)

  } catch {

    print("The file could not be loaded")

  }

}
```

There are two other ways of using **try**, but neither are really recomended. The first is like this:

```
let str = try! String(contentsOfFile: filename, usedEncoding: nil)
```

Note the exclamanation mark: **try!**. This means "I realize this call might throw an exception, but trust me: it never, ever will." This is useful only if you're 100% sure the call is safe. In our example we're loading a file from the app bundle, and if that file isn't there it means our app is corrupted, so it's OK to use here. You don't need do/catch when you use **try!**.

The second option is **try?** which means "if this call throws an exception, just return nil instead." This is closer to the Objective C way of handling errors, which was a bit scruffy. If

this is your preferred way of handling errors, then go for it! You don't need do/catch when use **try?**, but you should check and unwrap the result carefully.

What are lazy variables?

Availability: iOS 7.0 or later.

It's very common in iOS to want to create complex objects only when you need them, largely because with limited computing power at your disposal you need to avoid doing expensive work unless it's really needed.

Swift has a mechanism built right into the language that enables just-in-time calculation of expensive work, and it is called a *lazy variable*. These variables are created using a function you specify only when that variable is first requested. If it's never requested, the function is never run, so it does help save processing time.

I don't want to produce a complicated example because that would rather defy the point, so instead I've built a simple (if silly!) one: imagine you want to calculate a person's age using the Fibonacci sequence. This sequence goes 0, 1, 1, 2, 3, 5, 8, 13, 21, and so on – each number is calculated by adding the previous two numbers in the sequence. So if someone was aged 8, their Fibonacci sequence age would be 21, because that's at position 8 in the sequence.

I chose this because the most common pedagogical way to teach the Fibonacci sequence is using a function like this one:

```
func fibonacci(num: Int) -> Int {
  if num < 2 {
    return num
  } else {
    return fibonacci(num - 1) + fibonacci(num - 2)
  }
}
```

That function calls itself, which makes it a *recursive function*, and actually it's quite slow. If you try to calculate the Fibonacci value of something over, say, 21, expect it to be slow in a playground!

Anyway, we want to create a **Person** struct that has an age property and a **fibonacciAge** property, but we don't want that second one to be evaluated unless it's actually used. So, create this struct now:

```
struct Person {

  var age = 16

  lazy var fibonacciOfAge: Int = {

    return fibonacci(self.age)

  }()
}
```

There are five important things to note in that code:

- The lazy property is marked as **lazy var**. You can't make it **lazy let** because lazy properties must always be variables.
- Because the actual value is created by evaluation, you need to declare its data type up front. In the case of the code above, that means declaring the property as **Int**.
- Once you've set your data type, you need to use an open brace ("{") to start your block of code, then "}" to finish.
- You need to use **self** inside the function. In fact, if you're using a class rather than a structure, you should also declare **[unowned self]** inside your function so that you don't create a strong reference cycle.
- You need to end your lazy property with **()**, because what you're actually doing is making a call to the function you just created.

Once that code is written, you can use it like this:

```
var singer = Person()
singer.fibonacciOfAge
```

Remember, the point of lazy properties is that they are computed only when they are first needed, after which their value is saved. This means if you create 1000 singers and never touch their **fibonacciOfAge** property, your code will be lightning fast because that lazy work is never done.

What are property observers

Availability: iOS 7.0 or later.

Property observers are Swift's way of letting you attach functionality to changes in property values. For example, you might want to say, "whenever the player's score changes, update this label to show their new score." Here's a basic example that prints message to the debug console when a variable changes:

```
var score = 0 {
  willSet {
    print("Score is about to change to \(newValue)")
  }

  didSet {
    print("Score just changed from \(oldValue) to \(score)")
  }
}


score = 10
```

For more information see Hacking with Swift tutorial 1.

What are the changes in Swift 1.2?

Availability: iOS 7.0 or later.

Swift 1.2 was an interim release that fixed some early confusions and annoyances in the language. Its changes weren't big, but they did help clean up and clarify Swift, and helped tide us all over until the release of Swift 2.

The important changes are:

• You can now check and unwrap multiple optionals using **if/let** rather than create a so-called "pyramid of doom" with nested statements.
• Many Objective C types that were being passed around now had correct nullability values set. This was done by modifying Objective C then having many people scour through existing Apple code to add new annotations.
• Downcasting (a typecast from a higher type in your class hierarchy to a lower type) is now done using **as!** and **as?** to mark forced downcasting and optional downcasting respectively.
• Swift strings, arrays and dictionaries now no longer automatically typecast to **NSString**, **NSArray** and **NSDictionary**.
• A new **Set** data type was introduced to handle arrays where each value can appear only once.
• Constants can now be declared without a value, as long as they are provided with a value before they are used.
• Incremental build support was added, which makes it more efficient to build larger Swift projects.

What are the changes in Swift 2.0?
Availability: iOS 7.0 or later.

Swift 2.0 introduced a lot of major language changes. You can read my full article explaining the changes with code examples **by clicking here**, but here are the highlights:

• Checked exceptions using **try/catch** • Automatically synthesized headers
• The **guard** keyword to check input while unwrapping optionals
• Measuring strings is now done using **characters.count** • Delayed code execution using the **defer** keyword
• You now get mutability warnings if you declare variables that never get changed
• API availability checking is now built right in
• The **performSelector()** family of functions is now available

What does an exclamation mark mean?

Availability: iOS 7.0 or later.

Swift uses exclamation marks to signal both force unwrapping of optionals and expicitly unwrapped optionals. The former means "I know this optional variable definitely has a value, so let me use it directly." The latter means "this variable is going to be nil initially then will definitely have a value afterwards, so don't make me keep unwrapping it."

Broadly speaking, using exclamation marks is frowned upon because "trust me it's safe" isn't as good as the compiler absolutely enforcing it. That being said, it's your code: if you know something cannot be nil (usually because if it were nil your program would explode!) then do what works best.

For more information see Hacking with Swift tutorial 1.

What does override mean?

Availability: iOS 7.0 or later.

The **override** is used when you want to write your own method to replace an existing one in a parent class. It's used commonly when you're working with **UIViewControllers**, because view controllers already come with lots of methods like **viewDidLoad()** and **viewWillAppear()**. When you want to override these default methods, you need to specify this with the **override** keyword.

Now, you might be wondering why the **override** keyword is even needed, but it's really about ensuring your code is safe – if you write a method and accidentally name it the same as something that already exists, Xcode will simply refuse to build your app until you add in the **override** keyword. Similarly, if you use **override** on a method that doesn't override something that already exists, Xcode will refuse to build.

What does unowned mean?

Availability: iOS 7.0 or later.

Unowned variables are similar to weak variables in that they provide a way to reference data without having ownership. However, weak variables can become **nil** – they are effectively optional. In comparison, unowned variables must never be set to nil once they have been initialized, which means you don't need to worry about unwrapping optionals.

The most common place you'll see unowned variables is with closures that declare **[unowned self]** – this means "I want to reference **self** inside this closure but I don't want to own it." Why **unowned** rather than **weak**? Both would work, but let's face it: if **self** is nil inside a closure, something has gone wrong!

What does weak mean?
Availability: iOS 7.0 or later.

Unless you specific otherwise, all Swift properties are strong, which means they will not be removed from RAM until whatever owns them is removed from RAM. So, if you create an array in your view controller and you want it to stick around until the view controller is destroyed, that's what strong does.

Weak on the other hand is there when you want to say "I want to be able to reference this variable, but I don't mind if it goes away, so I don't want to own it." This might seem strange: after all, where's the point in having a reference to a variable that might not be there?

Well, the answer lies in a thing called reference cycles. If object A has a strong variable pointing at object B, and object B has a strong variable pointing at object A, neither object would ever be deleted because they both keep each other alive.

In this situation, having one of the objects change their property to be weak would solve the problem. For example, object A has a strong variable to object B, but object B has a weak variable pointing at object A. This guarantees that B cannot be destroyed while A still exists, but A can be destroyed because B doesn't have a strong variable owning it.

For more information see Hacking with Swift tutorial 1.

What is AnyObject?
Availability: iOS 7.0 or later.

This is one of those things that sounds obvious in retrospect: **AnyObject** is Swift's way of saying, "I don't mind what type of object you pass in here, it could be a string, it could be a string, it could be a number, it could be an array, or it could be a custom type you defined yourself.

If you were wondering: the reason numbers work even though they clearly aren't objects is because Swift silently *makes* them objects when they need to conform to **AnyObject**. Magic!

What is a CGFloat?

Availability: iOS 7.0 or later.

A **CGFloat** is a specialised form of **Float** that holds either 32-bits of data or 64-bits of data depending on the platform. The CG tells you it's part of Core Graphics, and it's found throughout UIKit, Core Graphics, Sprite Kit and many other iOS libraries.

If you have a **Float** or **Double** and need a **CGFloat**, you can convert it like this:

```
let myCGFloat = CGFloat(myDouble)
```

What is a closure?

Availability: iOS 7.0 or later.

If you're here because you find closures hard, that's OK: most people find closures hard. But in truth, closures aren't actually that complicated, so I hope I can explain them to you quickly and easily.

Here's my best, simplest definition: a closure is a kind of anonymous function that gets stored as a variable so it can be called later on, and has the special ability to remember the state of your program when you used it.

Some detail:

• "Anonymous function": that is, a closure is a block of code you define, starting with { and

ending with }. It's anonymous because it doesn't have a name – it doesn't need a name, because it gets stored as a variable.

 • "Stored as a variable": yes, the closure code literally gets saved as a variable, for example, **myCode**. Whoever is storing the closure (normally one of Apple's libraries) can then "call" that variable to run your closure's code.

 • "Called later on": once your closure has been stored away by iOS, it can be called a second later, a minute later, an hour later or never, depending on the situation. For example, when you say "run this code when my animation completes," iOS will make sure it happens at the right time.

 • "Remember the state of your program": if your closure references some variables that you had created, Swift will automatically take a copy of those variables so they can be used later. Remember, your closure can be called 20 minutes after you created it, so being able to store the original program state is important.

The truth is that you've probably used closures without realising it. Even a simple UIView animation call uses closures for the animations, and optionally also for the completion block. Just think of it as a chunk of code that gets called later on, and you're most of the way there.

What is a delegate in iOS?

Availability: iOS 2.0 or later.

Delegates are extremely common in iOS development, but fortunately they are easy to understand: a delegate is any object that should be notified when something interesting has happened. What that "something interesting" means depends on the context: for example, a table view's delegate gets notified when the user taps on a row, whereas a navigation controller's delegate gets notified when the user moves between view controllers.

When you agree to be the delegate for an object, you will almost certainly need to conform to a specific protocol, such as **UITableViewDelete**. These protocols will usually have some optional methods that you can implement if you care when something happens, for example, table views can notify you when users *deselect* a row, but most developers don't care. These protocols may also have some required methods that you *must* implement.

For more information see Hacking with Swift tutorial 1.

What is a dictionary?

Availability: iOS 7.0 or later.

A dictionary is a collection of values stored at named positions. Whereas you would access values in an array using **myArray[5]**, with a dictionary you use named positions such as **myDict["Paul"]** or **myDict["Scotland"]**. You don't even need to use strings for the positions – you can use another object if you choose, such as dates.

These named positions are called "keys", so dictionaries represent what's known as a key-value pair: each key has exactly one value, and each can appear only once in a dictionary.

For more information see Hacking with Swift tutorial 0.

What is a double?
Availability: iOS 7.0 or later.

The **Double** data type is the standard Swift way of storing decimal numbers such as 3.1, 3.14159 and 16777216.333921. Whenever you create a variable or constant that holds a number like this, Swift automatically assumes it's a **Double** rather than a **Float**, because it has a higher precision and therefore less likely to lose valuable accuracy.

For more information see Hacking with Swift tutorial 0.

What is a float?
Availability: iOS 7.0 or later.

The **Float** data type stores low-precision decimal numbers such as 3.1, 3.14159 and 556.9. It is not used that often in Swift, partly because **Double** is the default for these kinds of numbers (it has a higher precision), and partly because when you come across libraries that use regular floats they are more likely to want **CGFloat** instead.

For more information see Hacking with Swift tutorial 0.

What is a nib?
Availability: iOS 7.0 or later.

NIBs and XIBs are files that describe user interfaces, and are built using Interface Builder. In fact, the acronym "NIB" comes from "NexSTEP Interface Builder", and "XIB" from "Xcode Interface Builder". NIBs and XIBs are effectively the same thing: XIBs are newer and are used while you're developing, whereas NIBs are what get produced when you create a build.

In modern versions of iOS and Mac OS X, NIBs and XIBs have effectively been replaced by storyboards, although you may still meet them if you work on older projects.

What is a protocol?

Availability: iOS 2.0 or later.

A protocol is a collection of methods that describe a specific set of similar actions or behaviors. I realise that probably didn't help much, so I'll try to rephrase in more detail: how many rows should a table view have? How many sections? What should the section titles be? Can the user move rows? If so, what should happen when they do?

All those questions concern a similar thing: data going into a **UITableView**. As a result, they all go into a single protocol, called **UITableViewDataSource**. Some of the behaviors inside that protocol are optional. For example, **canEditRowAtIndexPath** is optional and defaults to true if you don't provide a value yourself.

When you work in Swift you will frequently have to make your class conform to a protocol. This is done by adding the protocol name to your class definition, like this:

```
class ViewController: UIViewController, UITableViewDataSource {
```

When you do that – when you promise Swift that your class conforms to a protocol – you can be darn sure it checks to make sure you're right. And that means it will refuse to build your code if you haven't added support for all the required methods, which is a helpful security measure.

For more information see Hacking with Swift tutorial 1.

What is a selector?

Availability: iOS 7.0 or later.

Selectors are effectively the names of methods on an object or struct, and they are used to execute some code at runtime. They were common in Objective C, but the earliest versions of Swift didn't include some core selector functionality so their use declined for a while. As of Swift 2 that functionality (things like **performSelectorInBackground**) has been restored.

In Swift, selectors are most commonly found when working with the target/action paradigm that you find in classes such as **NSTimer** and **UIBarButtonItem**. For example, when you create a timer you need to tell it who to notify when the timer fires (the target) and what selector should be called (the action). The same goes for bar button items: when the button is tapped, what selector should be called, and on what object?

What is a storyboard?

Availability: iOS 7.0 or later.

Storyboards were introduced way back in iOS 5 as a way to revamp interface design for iOS. At the time they didn't add much in the way of features that weren't available with the older XIBs, but in subsequent releases Apple have added helpful new features such as layout guides that make them much more useful – and arguably indispensible since iOS 7.

All new iOS projects come with at least one storyboard ready to use: Main.storyboard. Inside that you can create as many interfaces as you want, each representing one view controller in your app. You can then design in segues (pronounced "segway", like the gyro-bike things) that transition between view controllers – all without a single line of code.

There is one drawback to storyboards, and it's something you'll hit fairly quickly: if you have more than four view controllers in your app, you'll probably find it a little cumbersome to navigate around, particularly if you're working on a laptop. If you're working on something important, move your view controllers around: keep them organised from the beginning, otherwise your storyboard will soon become a nightmare!

For more information see Hacking with Swift tutorial 1.

What is a struct?

Availability: iOS 7.0 or later.

Classes and structures (structs) are so similar in Swift that it's easy to get them confused at first, but actually there are some important underlying differences:

 • A struct cannot inherit from another kind of struct, whereas classes can build on other classes.
 • You can change the type of an object at runtime using typecasting. Structs cannot have inheritance, so have only one type.
 • If you point two variables at the same struct, they have their own independent copy of the data. With objects, they both point at the same variable.

That last point is particularly important: with a struct you know your data is fixed in place, like an integer or other value. This means if you pass your struct into a function, you know it's not going to get modified.

What is a tuple?

Availability: iOS 7.0 or later.

Tuples in Swift occupy the space between dictionaries and structures: they hold very specific types of data (like a struct) but can be created on the fly (like dictionaries). They are commonly used to return multiple values from a function call.

You can create a basic tuple like this:

```
let person = (name: "Paul", age: 35)
```

As you can see, it looks like an anonymous struct: you can read **person.name** and **person.age** just like you would with a struct. But, helpfully, we haven't had to define the struct ahead of time – this is something made to be thrown away. It also means you don't get to conform to protocols or write methods inside your tuples, but that's OK.

Tuples can be accessed using element names ("name" and "age" above), or using a position

in the tuple, e.g. 0 and 1. You don't have to give your tuple elements names if you don't want to, but it's a good idea.

To give you a fully fledged tuple example, here's a function that splits a name like "Paul Hudson" in two, and returns a tuple containing the first name (Paul) and the last name (Hudson). Obviously this just a trivial example – it makes no attempt to cater for middle names, honorifics, or languages where family names come first!

```
func splitName(name: String) -> (firstName: String, lastName: String)
{
  let split = name.componentsSeparatedByString(" ")
  return (split[0], split[1])
}


let split = splitName("Paul Hudson")
split.0
split.1
split.firstName
split.lastName
```

As you can see, the return value from that function is **(firstName: String, lastName: String)**, which is a tuple with named elements. Those elements then get accessed using **split.0**, **split.1**, **split.firstName** and **split.lastName**.

What is an optional value in Swift?
Availability: iOS 7.0 or later.

Swift optionals are one of the most confusing parts of the language for beginners, but actually are fairly easy to understand. Put simply, if I declare a variable as an integer, that means it must hold a number. That number might be 0, 1, -1, 159, -758119, or whatever, but it's definitely a number. This works great for telling me, for example, where in an array a certain element can be found.

But what happens if I ask for the position of an element that doesn't exist in an array? Clearly returning 0 or any positive number isn't helpful, because you wouldn't be able to tell whether 0 meant "not found" or meant "found at the first position in an array." That's where optional values come in: an optional data type might have a value (0, 1, -1, etc) or might have no value at all.

Being able to say "has no value" for any kind of data is really important, and it's baked right into the core of Swift. You see, by default Swift won't let you work directly with optional values, because trying to work on data that isn't there causes a crash – imagine trying to uppercase someone's name when they haven't entered it yet. So, Swift forces you to check and unwrap optionals safely: if the optional has a value do something with it, otherwise do something else.

For more information see Hacking with Swift tutorial 1.

What is copy on write?
Availability: iOS 7.0 or later.

Copy on write is a common computing technique that helps boost performance when copying structures. To give you an example, imagine an array with 1000 things inside it: if you copied that array into another variable, Swift would have to copy all 1000 elements even if the two arrays ended up being the same.

This problem is solved using copy on write: when you point two variables at the same array they both point to the same underlying data. Swift promises that structs like arrays and dictionaries are copied as values, like numbers, so having two variables point to the same data might seem to contradict that. The solution is simple but clever: if you modify the second variable, Swift takes a full copy at that point so that only the second variable is modified - the first isn't changed.

So, by delaying the copy operation until it's actually needed, Swift can ensure that no wasted work is done.

Warning: copy on write is a feature specifically added to Swift arrays and dictionaries; you don't get it for free in your own data types.

What is whole module optimization?

Availability: iOS 7.0 or later.

Whole module optimization is a compiler pass that can add significant performance gains, and so it's always worth enabling when doing a release build of your app for the App Store. How it works is quite simple: when Swift builds the final version of your app it combines all your source files together and can evaluate the whole structure of your program at once. This lets it make extra optimizations that would be impossible before, when every file was optimized individually.

What's the difference between let and var?

Availability: iOS 7 or later.

Swift lets you create both variables and constants as ways to reference your data, but there's a strong push (even Xcode warnings!) if you create things as variables then never change them. To make a constant, use **let** like this:

```
let x = 10
```

To make a variable, use **var** like this:

```
var x = 10
```

The reason Swift strongly encourages you to use constants wherever possible is because it's safer: if you say "this value will never change," then Swift will refuse to let you change it even by accident. It also opens the possibility of compiler optimizations if the system knows certain data will not change.

For more information see Hacking with Swift tutorial 1.

Libraries

How to get a Cover Flow effect on iOS

Availability: iOS 5.0 or later.

You can get an instant Cover Flow effect on iOS by using the marvellous and free iCarousel library. You can download it from **https://github.com/nicklockwood/iCarousel** and drop it into your Xcode project fairly easily by adding a bridging header (it's written in Objective C).

If you haven't added Objective C code to a Swift project before, follow these steps:

 • **Download iCarousel** and unzip it
 • Go into the folder you unzipped, open its iCarousel subfolder, then select iCarousel.h and iCarousel.m and drag them into your project navigation – that's the left pane in Xcode. Just beloe Info.plist is fine.
 • Check "Copy items if needed" then click Finish.
 • Xcode will prompt you with the message "Would you like to configure an Objective-C bridging header?" Click "Create Bridging Header"
 • You should see a new file in your project, named YourProjectName-Bridging-Header.h.
 • Add this line to the file: **#import "iCarousel.h"**

Once you've added iCarousel to your project you can start using it. Here's a complete, albeit simplified, example:

```
override func viewDidLoad() {

  super.viewDidLoad()


  let carousel = iCarousel(frame: CGRect(x: 0, y: 0, width: 300,
height: 200))

  carousel.dataSource = self

  carousel.type = .CoverFlow

  view.addSubview(carousel)


}
```

```
func numberOfItemsInCarousel(carousel: iCarousel) -> Int {

  return 10

}


func carousel(carousel: iCarousel, viewForItemAtIndex index: Int,
reusingView view: UIView?) -> UIView {

  let imageView: UIImageView


  if view != nil {

    imageView = view as! UIImageView

  } else {

    imageView = UIImageView(frame: CGRect(x: 0, y: 0, width: 128,
height: 128))

  }


  imageView.image = UIImage(named: "example")


  return imageView

}
```

That example loads the same image for all 10 carousel slides, so you'll need to change that to load data from your app.

If you have the time, do check out the other carousel types that iCarousel offers – they're quite remarkable!

How to make empty UITableViews look more attractive using DZNEmptyDataSet
Availability: iOS 7.0 or later.

If you use table views or collection views and you want to take one simple step to make your app both more attractive and more user-friendly, let me tell you what the pros do: we use **DZNEmptyDataSet**. This simple, free, open source library is designed to handle the case when your data source is empty by showing some prompt text, and optionally also a button or an image.

What I love about this library is that it's so astonishingly simple, and it even uses **NSAttributedString** so you can provide custom formatting.

First things first: **go here and click Download Zip** to get the source code to **DZNEmptyDataSet**. Now unzip it, then look inside its Source folder for two files: **UIScrollView+EmptyDataSet.h** and **UIScrollView+EmptyDataSet.m**.

Drag these into your Xcode project, and Xcode should prompt you with the message "Would you like to configure an Objective-C bridging header?" Click "Creating Bridging Header" and you'll see a file called **YourProjectName-Bridging-Header.h** appear in your project. Open that file for editing in Xcode and give it this text:

```
#import "UIScrollView+EmptyDataSet.h"
```

This is required because **DZNEmptyDataSet** is written in Objective-C, so these steps are required to make it available to use in Swift.

Next, tell Swift that your current table view controller (or collection view controller) conforms to the **DZNEmptyDataSetSource** and **DZNEmptyDataSetDelegate** protocols like this:

```
class MasterViewController: UITableViewController,
DZNEmptyDataSetSource, DZNEmptyDataSetDelegate {
```

You then need to add these three lines of code to your **viewDidLoad()** method:

```
tableView.emptyDataSetSource = self

tableView.emptyDataSetDelegate = self
```

```
tableView.tableFooterView = UIView()
```

The first two lines set your code up ready to provide various **DZNEmptyDataSet** elements; the third one is just there to make your interface cleaner.

One of the great things about **DZNEmptyDataSet** is that you only need to provide what you want. This means you can provide just a heading, or perhaps a heading and an image, or a heading, a description, an image and even a button. Even better, the button is made for you: all you need to do is tell **DZNEmptyDataSet** what its title should be.

The example code below sets up a title, a description, an image and a button, and even provides a response to the button being tapped. Remember: you don't need all these, just the ones you want to use in your app.

```
func titleForEmptyDataSet(scrollView: UIScrollView!) ->
NSAttributedString! {

  let str = "Welcome"

  let attrs = [NSFontAttributeName:
UIFont.preferredFontForTextStyle(UIFontTextStyleHeadline)]

  return NSAttributedString(string: str, attributes: attrs)

}


func descriptionForEmptyDataSet(scrollView: UIScrollView!) ->
NSAttributedString! {

  let str = "Tap the button below to add your first grokkleglob."

  let attrs = [NSFontAttributeName:
UIFont.preferredFontForTextStyle(UIFontTextStyleBody)]

  return NSAttributedString(string: str, attributes: attrs)

}


func imageForEmptyDataSet(scrollView: UIScrollView!) -> UIImage! {

  return UIImage(named: "taylor-swift")

}
```

```swift
func buttonTitleForEmptyDataSet(scrollView: UIScrollView!, forState
state: UIControlState) -> NSAttributedString! {

    let str = "Add Grokkleglob"

    let attrs = [NSFontAttributeName:
UIFont.preferredFontForTextStyle(UIFontTextStyleCallout)]

    return NSAttributedString(string: str, attributes: attrs)

}


func emptyDataSetDidTapButton(scrollView: UIScrollView!) {

    let ac = UIAlertController(title: "Button tapped!", message: nil,
preferredStyle: .Alert)

    ac.addAction(UIAlertAction(title: "Hurray", style: .Default,
handler: nil))

    presentViewController(ac, animated: true, completion: nil)

}
```

How to parse JSON using SwiftyJSON

Availability: iOS 7.0 or later.

SwiftyJSON is a super-simplified JSON parsing library that gives you clearer syntax than the
built-in iOS libraries, and better yet it's completely free. You can **download it from here** but
at the time of writing its Xcode 7 / Swift 2 support was kept in a separate branch – if that's
still the case when you click that link, **try downloading from here instead**.

Unzip the file you downloaded, then look in its Source directory and drag SwiftyJSON.swift
into your Xcode project. To use SwiftyJSON, you need to convert your JSON string into an
NSData object, then send it in for parsing. Once that's done, you simply request data in the
format you want, and (here's the awesome bit) *SwiftyJSON is guaranteed to return
something.* That "something" is going to be your data, if all things are in good shape. But if
you requested the wrong thing (either with a typo, or because you didn't understand your
JSON structure correctly) or if the JSON has changed, SwiftyJSON will just return a default
value instead.

To get you started, here is some example JSON:

```
let json = "{ \"people\": [{ \"firstName\": \"Paul\", \"lastName\":
\"Hudson\", \"isAlive\": true }, { \"firstName\": \"Angela\",
\"lastName\": \"Merkel\", \"isAlive\": true }, { \"firstName\":
\"George\", \"lastName\": \"Washington\", \"isAlive\": false } ] }";
```

That contains an array of three people, each of which have a first name, a last name, and an "is alive" status. To parse that using SwiftyJSON and print out all the first names, here's the code:

```
if let data = json.dataUsingEncoding(NSUTF8StringEncoding) {

  let json = JSON(data: data)


  for item in json["people"].arrayValue {

    print(item["firstName"].stringValue)

  }

}
```

It's the **arrayValue** and **stringValue** properties that do all the magic: the first one returns the array of people or an empty array if the "people" element didn't exist, and the second one returns the "firstName" property of a person, or an empty string if it wasn't set. So, no matter what happens, that code will work, which means it's easy to write and safe to run.

Sometimes JSON has quite deeply nested dictionaries, but that's OK: SwiftyJSON can navigate through multiple levels in one call, and if any one level fails you'll still get back your default value. For example, if you have JSON like this:

```
{

   "metadata":{

       "responseInfo":{
```

```
        "status":200,

        "developerMessage":"OK",

    }

  }

}
```

You might want to check that the status code is 200 before continuing. To do that, just read the "metaData", "responseInfo" and "status" values all at once, and ask SwiftyJSON for its **intValue** – you'll either get the correct number (200) or 0 if any of those values don't exist. Like this:

```
if json["metadata"]["responseInfo"]["status"].intValue == 200 {

    // we're OK to parse!

}
```

When Swift 1.2 came out with its smarter ability to do optional unwrapping, some people thought SwiftyJSON was dead in the water. Those people must never have used SwiftyJSON because it does so much more than unwrap things safely – have fun!

For more information see Hacking with Swift tutorial 7.

Location

How to add a button to an MKMapView annotation
Availability: iOS 2.0 or later.

The built-in **MKPinAnnotationView** annotation view has a **rightCalloutAccessoryView** property that can be set to any kind of **UIView**, including buttons. The button doesn't need to

have an action attached to it, because there's a separate method that gets called when it's tapped.

First up, here's how you'd create a button inside an annotation view:

```
let btn = UIButton(type: .DetailDisclosure)
annotationView.rightCalloutAccessoryView = btn
```

For context, here's a complete implementation of **viewForAnnotation** that uses a button. This is taken from project 19 of Hacking with Swift, where I created a class called **Capital** that implemented the **MKAnnotation** protocol – you'll need to adjust this for your own annotation type:

```
func mapView(mapView: MKMapView, viewForAnnotation annotation:
MKAnnotation) -> MKAnnotationView? {

  let identifier = "Capital"


  if annotation.isKindOfClass(Capital.self) {

    if let annotationView =
mapView.dequeueReusableAnnotationViewWithIdentifier(identifier) {

      annotationView.annotation = annotation

      return annotationView

    } else {

      let annotationView = MKPinAnnotationView(annotation:annotation,
reuseIdentifier:identifier)

      annotationView.enabled = true

      annotationView.canShowCallout = true


      let btn = UIButton(type: .DetailDisclosure)

      annotationView.rightCalloutAccessoryView = btn

      return annotationView

    }
```

```
    }


    return nil

}
```

When it comes to detecting taps on your button, implement the **calloutAccessoryControlTapped** method. This tells you the annotation view that was tapped (from which you can pull out the annotation), the control that was tapped (in our case it's a button), and also the map view the whole thing belongs to. Here's an example:

```
func mapView(mapView: MKMapView, annotationView view:
MKAnnotationView, calloutAccessoryControlTapped control: UIControl) {

  let capital = view.annotation as! Capital

  let placeName = capital.title

  let placeInfo = capital.info


  let ac = UIAlertController(title: placeName, message: placeInfo,
preferredStyle: .Alert)

  ac.addAction(UIAlertAction(title: "OK", style: .Default, handler:
nil))

  presentViewController(ac, animated: true, completion: nil)

}
```

For more information see Hacking with Swift tutorial 19.

How to add an MKMapView using MapKit

Availability: iOS 2.0 or later.

Map views are pretty easy in iOS, largely because they are baked right into Interface Builder. That's right: open your storyboard, drag a map view into your view, and you're already most of the way there! But there is one further thing to do, which is where people get confused: by

default, the map framework won't actually be loaded when your app is run, which will make your app crash when it tries to show the map view.

The solution is simple: go to your project navigation, choose the Capabilities tab, then look for the "Maps" item and set it to be On. That's it!

For more information see Hacking with Swift tutorial 19.

How to detect iBeacons

Availability: iOS 7.0 or later.

Detecting iBeacons requires a number of steps. But first you need to decide whether you want to detect beacons only when your app is running, or whether you want beacons to be detected even if your app isn't in the background.

Have you decided? Good, because you need to set one of two keys in your Info.plist depending on your choice. If you want to detect beacons only when your app is running, add the key **NSLocationWhenInUseUsageDescription** and a short string explaining how you'll use the location, e.g. "We want to detect where you are in our store."

If you want the app to detect beacons even when it isn't running (a feat accomplished by handing control of scanning over to the OS), you should use the **NSLocationAlwaysUsageDescription** key instead.

With that done, we can start to scan for beacons. Open your class in Xcode (it could be a view controller, but it doesn't have to be), then import the Core Location framework like this:

```
import CoreLocation
```

Now tell Swift that your class conforms to the **CLLocationManagerDelegate** protocol so that you can start to receive location updates. If you're using a view controller subclass, your code will look something like this:

```
class ViewController: UIViewController, CLLocationManagerDelegate {
```

iBeacon tracking is done using the **CLLocationManager** class, which is also responsible for requesting location permission from users. You need to create a property for this in your class so that you can store the active location manager, so add this:

```
var locationManager: CLLocationManager!
```

If you're using a view controller, you'll probably want to initialize this property in **viewDidLoad()**, like this:

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

    locationManager = CLLocationManager()
    locationManager.delegate = self
    locationManager.requestAlwaysAuthorization()
}
```

If you're using another type of class, you should amend that appropriately. Note that that calls **requestAlwaysAuthorization()**, which is the correct method to use if you set the **NSLocationAlwaysUsageDescription** Info.plist key. If you set **NSLocationWhenInUseUsageDescription** instead, you should use **requestWhenInUseAuthorization()** instead.

Once you request permission to use your user's location, they'll see an alert with the message you wrote earlier. When they make a choice you'll get a delegate callback called **didChangeAuthorizationStatus**, at which point you can check whether they are authorized you or not:

```
func locationManager(manager: CLLocationManager!,
didChangeAuthorizationStatus status: CLAuthorizationStatus) {
```

```
    if status == .AuthorizedAlways {

        if
CLLocationManager.isMonitoringAvailableForClass(CLBeaconRegion.self)
{

            if CLLocationManager.isRangingAvailable() {

                startScanning()

            }

        }

    }

}
```

Don't worry, we haven't written the **startScanning()** method yet. If you're using "when in use" mode you should check for **.AuthorizedWhenInUse** rather than **.AuthorizedAlways**.

Once you've been authorized to scan for iBeacons, you can create **CLBeaconRegion** objects and pass them to the location manager. Each **CLBeaconRegion** is uniquely identified by a long number (it's UUID), and optionally also major and minor numbers. As well as monitoring for a beacon's existence, we're also going to ask iOS to range the beacon for us – i.e., tell us how close it thinks we are.

Here's the code:

```
func startScanning() {

    let uuid = NSUUID(UUIDString: "5A4BCFCE-174E-4BAC-
A814-092E77F6B7E5")

    let beaconRegion = CLBeaconRegion(proximityUUID: uuid, major:
123, minor: 456, identifier: "MyBeacon")


    locationManager.startMonitoringForRegion(beaconRegion)

    locationManager.startRangingBeaconsInRegion(beaconRegion)

}
```

Once you're ranging for beacons, you'll get a delegate callback called **didRangeBeacons** every second or so, at which point you can read a beacon's distance using its **proximity** value and take appropriate action.

For example, we can make our view change color depending on how far away an iBeacon is with this code:

```
func locationManager(manager: CLLocationManager!, didRangeBeacons
beacons: [AnyObject]!, inRegion region: CLBeaconRegion!) {

    if beacons.count > 0 {

        let beacon = beacons[0] as! CLBeacon

        updateDistance(beacon.proximity)

    } else {

        updateDistance(.Unknown)

    }

}


func updateDistance(distance: CLProximity) {

    UIView.animateWithDuration(0.8) {

        switch distance {

        case .Unknown:

            self.view.backgroundColor = UIColor.grayColor()


        case .Far:

            self.view.backgroundColor = UIColor.blueColor()


        case .Near:

            self.view.backgroundColor = UIColor.orangeColor()


        case .Immediate:

            self.view.backgroundColor = UIColor.redColor()
```

```
        }
    }
}
```

For more information see Hacking with Swift tutorial 22.

How to find directions using MKMapView and MKDirectionsRequest

Availability: iOS 6.0 or later.

MapKit is great for letting users navigate from place to place, but also makes it easy for you to plot directions from one place to another. You just tell iOS where you're starting from, where you're going, as well as how you're travelling (by car, foot, or mass transit), and it will find routes for you.

First, make sure you have a map view in your app, and have the Maps entitlement enabled. Now add this code:

```
import MapKit

import UIKit


class ViewController: UIViewController {

  @IBOutlet weak var mapView: MKMapView!


  override func viewDidLoad() {

    super.viewDidLoad()


    let request = MKDirectionsRequest()

    request.source = MKMapItem(placemark: MKPlacemark(coordinate:
CLLocationCoordinate2D(latitude: 40.7127, longitude: -74.0059),
addressDictionary: nil))
```

```
    request.destination = MKMapItem(placemark:
MKPlacemark(coordinate: CLLocationCoordinate2D(latitude: 37.783333,
longitude: -122.416667), addressDictionary: nil))

    request.requestsAlternateRoutes = true

    request.transportType = .Automobile


    let directions = MKDirections(request: request)


    directions.calculateDirectionsWithCompletionHandler { [unowned
self] response, error in

        guard let unwrappedResponse = response else { return }


        for route in unwrappedResponse.routes {

            self.mapView.addOverlay(route.polyline)


self.mapView.setVisibleMapRect(route.polyline.boundingMapRect,
animated: true)

        }

    }

  }


  func mapView(mapView: MKMapView, rendererForOverlay overlay:
MKOverlay) -> MKOverlayRenderer {

    let renderer = MKPolylineRenderer(polyline: overlay as!
MKPolyline)

    renderer.strokeColor = UIColor.blueColor()

    return renderer

  }

}
```

That example requests driving directions between New York and San Francisco. It asks for alternate routes if they exist (spoiler: they do), then sets up a closure to run when the directions come back that adds them as overlays to the map. To make the overlays draw,

you need to implement the **rendererForOverlay** method, but that's just three lines as you can see.

Note: because I request alternative routes if they exist, I loop through the array of returned routes to add them all to the map. The **setVisibleMapRect()** method is called once for each route, but fortunately that isn't a problem as all routes have the same start and end location!

How to make an iPhone transmit an iBeacon
Availability: iOS 7.0 or later.

iOS 7.0 introduced not only the ability to detect iBeacons, but also the ability to create iBeacons – for iPhones and iPads to broadcast their own beacon signal that can then be detected by other devices. To make this work, you add these two imports:

```
import CoreBluetooth
import CoreLocation
```

Now make your view controller (or other class) conform to the **CBPeripheralManagerDelegate** protocol, like this:

```
class ViewController: UIViewController, CBPeripheralManagerDelegate {
```

To make your beacon work, you need to create three properties: the beacon itself, plus two Bluetooth properties that store configuration and management information. Add these three now:

```
var localBeacon: CLBeaconRegion!
var beaconPeripheralData: NSDictionary!
var peripheralManager: CBPeripheralManager!
```

Finally the code: here are three functions you can use to add local beacons to your app. The first one creates the beacon and starts broadcasting, the second one stops the beacon, and the third one acts as an intermediary between your app and the iOS Bluetooth stack:

```swift
func initLocalBeacon() {

  if localBeacon != nil {

    stopLocalBeacon()

  }


  let localBeaconUUID = "5A4BCFCE-174E-4BAC-A814-092E77F6B7E5"

  let localBeaconMajor: CLBeaconMajorValue = 123

  let localBeaconMinor: CLBeaconMinorValue = 456


  let uuid = NSUUID(UUIDString: localBeaconUUID)!

  localBeacon = CLBeaconRegion(proximityUUID: uuid, major:
localBeaconMajor, minor: localBeaconMinor, identifier: "Your private
identifer here")


  beaconPeripheralData =
localBeacon.peripheralDataWithMeasuredPower(nil)

  peripheralManager = CBPeripheralManager(delegate: self, queue: nil,
options: nil)

}


func stopLocalBeacon() {

  peripheralManager.stopAdvertising()

  peripheralManager = nil

  beaconPeripheralData = nil

  localBeacon = nil

}


func peripheralManagerDidUpdateState(peripheral: CBPeripheralManager)
```

```
{

  if peripheral.state == .PoweredOn {

    peripheralManager.startAdvertising(beaconPeripheralData as!
[String: AnyObject]!)

  } else if peripheral.state == .PoweredOff {

    peripheralManager.stopAdvertising()

  }

}
```

How to request a user's location only once using requestLocation

Availability: iOS 9.0 or later.

iOS 9.0 introduces a simple way to request a user's location just once, and it's called **requestLocation()**. Calling this method returns immediately (meaning that your code carries on executing) but when iOS has managed (or failed) to get a fix on the user's location you will be told. Below is a complete example:

```
import CoreLocation

import UIKit


class ViewController: UIViewController, CLLocationManagerDelegate {

  let manager = CLLocationManager()


  override func viewDidLoad() {

    manager.delegate = self

    manager.requestLocation()

  }


  func locationManager(manager: CLLocationManager, didUpdateLocations
```

```
locations: [CLLocation]) {

    if let location = locations.first {

      print("Found user's location: \(location)")

    }

  }


  func locationManager(manager: CLLocationManager, didFailWithError
error: NSError) {

    print("Failed to find user's location: \
(error.localizedDescription)")

  }

}
```

Media

CIDetectorTypeFace: How to detect faces in a UIImage
Availability: iOS 5.0 or later.

Core Image has a number of feature detectors built right in, including the ability to detect faces, eyes, mouths, smiles and even blinking in pictures. When you ask it to look for faces in a picture, it will return you an array of all the faces it found, with each one containing face feature details such as eye position. Here's an example:

```
if let inputImage = UIImage(named: "taylor-swift") {

  let ciImage = CIImage(CGImage: inputImage.CGImage!)


  let options = [CIDetectorAccuracy: CIDetectorAccuracyHigh]

  let faceDetector = CIDetector(ofType: CIDetectorTypeFace, context:
nil, options: options)
```

```
    let faces = faceDetector.featuresInImage(ciImage)

    if let face = faces.first as? CIFaceFeature {
      print("Found face at \(face.bounds)")

      if face.hasLeftEyePosition {
        print("Found left eye at \(face.leftEyePosition)")
      }

      if face.hasRightEyePosition {
        print("Found right eye at \(face.rightEyePosition)")
      }

      if face.hasMouthPosition {
        print("Found mouth at \(face.mouthPosition)")
      }
    }
}
```

How to choose a photo from the camera roll using UIImagePickerController

Availability: iOS 2.0 or later.

The **UIImagePickerController** class is a super-simple way to select and import user photos into your app. As a bonus, it also automatically handles requesting user permission to read the photo library, so all you need to do is be ready to respond when the user selects a photo.

First, make sure your view controller conforms to the **UINavigationControllerDelegate** and **UIImagePickerControllerDelegate** protocols, like this:

```
class ViewController: UIViewController,
UINavigationControllerDelegate, UIImagePickerControllerDelegate {
```

Now you need three more pieces of code: one to show the image picker, one to handle the user tapping cancel, and one to handle the user selecting a photo. Here is some example code to get you started:

```
func selectPicture() {

  let picker = UIImagePickerController()

  picker.allowsEditing = true

  picker.delegate = self

  presentViewController(picker, animated: true, completion: nil)

}


func imagePickerControllerDidCancel(picker: UIImagePickerController)
{

  dismissViewControllerAnimated(true, completion: nil)

}


func imagePickerController(picker: UIImagePickerController,
didFinishPickingMediaWithInfo info: [String : AnyObject]) {

  var newImage: UIImage


  if let possibleImage = info["UIImagePickerControllerEditedImage"]
as? UIImage {

    newImage = possibleImage

  } else if let possibleImage =
info["UIImagePickerControllerOriginalImage"] as? UIImage {

    newImage = possibleImage

  } else {
```

```
        return
    }


    // do something interesting here!
    print(newImage.size)


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

To use that code in your own project, replace the call to **print()** with something useful – you have the image, now what?

For more information see Hacking with Swift tutorial 10.

How to convert text to speech using AVSpeechSynthesizer, AVSpeechUtterance and AVSpeechSynthesisVoice

Availability: iOS 7.0 or later.

If you're looking for text-to-speech conversion, it's baked right into iOS thanks to the **AVSpeechSynthesizer** class and its friends. As you can tell from the "AV" part of its name, you'll need to add AVFoundation to your project, like this:

```
import AVFoundation
```

With that done, you can speak whatever you want. For example, to say "Hello world" in a very slow British accent, use this:

```
let utterance = AVSpeechUtterance(string: "Hello world")
utterance.voice = AVSpeechSynthesisVoice(language: "en-GB")
utterance.rate = 0.1
```

```
let synthesizer = AVSpeechSynthesizer()
synthesizer.speakUtterance(utterance)
```

You can omit the **rate** property entirely to have a natural-speed voice, or change the language to "en-US" (English, American accent), "en-IE" (English, Irish accent), "en-AU" (English, Australian accent) or whichever other accents Apple chooses to add in the future.

How to create a PDF417 barcode

Availability: iOS 9.0 or later.

PDF417 barcodes - most frequently seen on boarding passes at airports, but also seen in digital postage stamps and other places – are built right into iOS as of iOS 9.0. This function below accepts a string as its only parameter and returns a **UIImage** containing the PDF417 barcode representing that string:

```
func generatePDF417BarcodeFromString(string: String) -> UIImage? {

  let data = string.dataUsingEncoding(NSISOLatin1StringEncoding)

  if let filter = CIFilter(name: "CIPDF417BarcodeGenerator") {
    filter.setValue(data, forKey: "inputMessage")
    let transform = CGAffineTransformMakeScale(3, 3)


    if let output =
filter.outputImage?.imageByApplyingTransform(transform) {

      return UIImage(CIImage: output)

    }

  }


  return nil
```

```
}
```

```
let image = generatePDF417BarcodeFromString("Hacking with Swift")
```

How to create a QR code

Availability: iOS 7.0 or later.

iOS has a built-in QR code generator, but it's a bit tricksy to use because it's exposed as a Core Image filter that needs various settings to be applied. Also, it generates codes where every bit is just one pixel across, which looks terrible if you try to stretch it inside an image view.

So, here's a simple function that wraps up QR code generation while also scaling up the QR code so it's a respectable size:

```
func generateQRCodeFromString(string: String) -> UIImage? {
  let data = string.dataUsingEncoding(NSISOLatin1StringEncoding)

  if let filter = CIFilter(name: "CIQRCodeGenerator") {
    filter.setValue(data, forKey: "inputMessage")
    filter.setValue("H", forKey: "inputCorrectionLevel")
    let transform = CGAffineTransformMakeScale(10, 10)

    if let output =
filter.outputImage?.imageByApplyingTransform(transform) {
      return UIImage(CIImage: output)
    }
  }

  return nil
}
```

```
let image = generateQRCodeFromString("Hacking with Swift is the best
iOS coding tutorial I've ever read!")
```

How to create a barcode

Availability: iOS 8.0 or later.

You can generate a string into a traditional barcode using iOS using Core Image, but you should make sure and convert your input string to an **NSData** using **NSASCIIStringEncoding** to ensure compatibility. Here's a function you can use that wraps it all up neatly, including scaling up the barcode so it's a bit bigger:

```
func generateBarcodeFromString(string: String) -> UIImage? {

  let data = string.dataUsingEncoding(NSASCIIStringEncoding)


  if let filter = CIFilter(name: "CICode128BarcodeGenerator") {

    filter.setValue(data, forKey: "inputMessage")

    let transform = CGAffineTransformMakeScale(3, 3)


    if let output =
filter.outputImage?.imageByApplyingTransform(transform) {

      return UIImage(CIImage: output)

    }

  }


  return nil

}


let image = generateBarcodeFromString("Hacking with Swift")
```

How to filter images using Core Image and CIFilter

Availability: iOS 5.0 or later.

Core Image is the one of the most powerful frameworks available to iOS developers: it makes hardware-accelerated image manipulation ridiculously easy, which means you get to add powerful graphical effects to your apps and games with very little work.

Most of the work is done by choosing the right **CIFilter**. Apple's official documentation goes into great detail about the various filters you can use, and you can also read Hacking with Swift project 13 for a hands-on tutorial showing off various effects. The code below applies a 50% sepia tone effect to an image:

```
let inputImage = UIImage(named: "taylor-swift")!
let context = CIContext(options: nil)

if let currentFilter = CIFilter(name: "CISepiaTone") {
  let beginImage = CIImage(image: inputImage)
  currentFilter.setValue(beginImage, forKey: kCIInputImageKey)
  currentFilter.setValue(0.5, forKey: kCIInputIntensityKey)

  if let output = currentFilter.outputImage {
    let cgimg = context.createCGImage(output, fromRect:
output.extent)

    let processedImage = UIImage(CGImage: cgimg)

    // do something interesting with the processed image
  }
}
```

For more information see Hacking with Swift tutorial 13.

How to highlight text to speech words being read using AVSpeechSynthesizer

Availability: iOS 7.0 or later.

iOS has text-to-speech synthesis built right into the system, but even better is that it allows you to track when individual words are being spoken so that you can highlight the words on the screen. This is extremely easy to do thanks to the **AVSpeechSynthesizerDelegate** protocol: you get two callbacks in the form of **willSpeakRangeOfSpeechString** and **didFinishSpeechUtterance**, where you can do your work.

First, make sure you import AVFoundation into your project. Now make your class conform to the **AVSpeechSynthesizerDelegate** protocol. For example, if you're using a regular view controller, you would write this:

```swift
class ViewController: UIViewController, AVSpeechSynthesizerDelegate {
```

Place a label into your view controller, then hook it up to an outlet called **label**. Now add these two methods:

```swift
func speechSynthesizer(synthesizer: AVSpeechSynthesizer!,
willSpeakRangeOfSpeechString characterRange: NSRange, utterance:
AVSpeechUtterance!) {

  let mutableAttributedString = NSMutableAttributedString(string:
utterance.speechString)

mutableAttributedString.addAttribute(NSForegroundColorAttributeName,
value: UIColor.redColor(), range: characterRange)

  label.attributedText = mutableAttributedString

}


func speechSynthesizer(synthesizer: AVSpeechSynthesizer!,
didFinishSpeechUtterance utterance: AVSpeechUtterance!) {

  label.attributedText = NSAttributedString(string:
```

```
utterance.speechString)

}
```

Finally, you need to trigger the text-to-speech engine – this might be by a button press perhaps, but it's down to you. Here's the method I attached to a button press:

```
@IBAction func speak(sender: AnyObject) {
  let string = label.text
  let utterance = AVSpeechUtterance(string: string)
  utterance.voice = AVSpeechSynthesisVoice(language: "en-GB")

  let synthesizer = AVSpeechSynthesizer()
  synthesizer.delegate = self
  synthesizer.speakUtterance(utterance)
}
```

How to make resizable images using resizableImageWithCapInsets()

Availability: iOS 2.0 or later.

If you use a small image in a large image view, you can make the image stretch to fit if you want to but it probably won't look great. iOS provides an alternative known as *resizable images*, which is where you define part of an image as being fixed in size and let iOS stretch the remainder.

This technique is common with button graphics: you make the corners fixed in size, then stretch the center part as big as it needs to be. The center part ought to be just one pixel by one pixel in size so that it stretches perfectly, but you can also ask iOS to repeat the center area as a tile if that's what you want.

This example code below creates a resizable image by defining the corners as 8 points each

and stretching the rest:

```
if let img = UIImage(named: "button") {

  let resizable = img.resizableImageWithCapInsets(UIEdgeInsets(top:
8, left: 8, bottom: 8, right: 8), resizingMode: .Stretch)

}
```

How to play sounds using AVAudioPlayer

Availability: iOS 2.2 or later.

The most common way to play a sound on iOS is using **AVAudioPlayer**, and it's popular for a reason: it's easy to use, you can stop it whenever you want, and you can adjust its volume as often as you need. The only real catch is that you must store your player as a property or other variable that won't get destroyed straight away – if you don't, the sound will stop immediately.

AVAudioPlayer is part of the AVFoundation framework, so you'll need to import that:

```
import AVFoundation
```

Like I said, you need to store your audio player as a property somewhere so it is retained while the sound is playing. In our example we're going to play a bomb explosion sound, so I created a property for it like this:

```
var bombSoundEffect: AVAudioPlayer!
```

With those two lines of code inserted, all you need to do is play your audio file. This is done first by finding where the sound is in your project using **pathForResource()**, then creating a file URL out of it. That can then get passed to **AVAudioPlayer** to create an audio player object, at which point – finally – you can play the sound. Here's the code:

```
let path = NSBundle.mainBundle().pathForResource("sliceBombFuse.caf",
ofType:nil)!

let url = NSURL(fileURLWithPath: path)


do {

  let sound = try AVAudioPlayer(contentsOfURL: url)

  bombSoundEffect = sound

  sound.play()

} catch {

  // couldn't load file :(

}
```

If you want to stop the sound, you should use its **stop()** method. But be warned: if you try to stop a sound that doesn't exist your app will crash, so it's best to check that it exists first like this:

```
if bombSoundEffect != nil {

    bombSoundEffect.stop()

    bombSoundEffect = nil

}
```

For more information see Hacking with Swift tutorial 17.

How to record audio using AVAudioRecorder
Availability: iOS 3.0 or later.

While it's not *hard* to record audio with an iPhone, it does take quite a bit of code so give yourself a few minutes to get this implemented. First you need to import the **AVFoundation**

framework into your view controller, like this:

```
import AVFoundation
```

You will need to add three properties to your view controller: a button for the user to tap to start or stop recording, an audio session to manage recording, and an audio recorder to handle the actual reading and saving of data. You can create the button in Interface Builder if you prefer; we'll be doing it in code here.

Put these three properties into your view controller:

```
var recordButton: UIButton!

var recordingSession: AVAudioSession!
var audioRecorder: AVAudioRecorder!
```

Recording audio requires a user's permission to stop malicious apps doing malicious things, so we need to request recording permission from the user. If they grant permission, we'll create our recording button. Put this into **viewDidLoad()**:

```
recordingSession = AVAudioSession.sharedInstance()

do {
    try
recordingSession.setCategory(AVAudioSessionCategoryPlayAndRecord)
    try recordingSession.setActive(true)
    recordingSession.requestRecordPermission() { [unowned self]
(allowed: Bool) -> Void in
        dispatch_async(dispatch_get_main_queue()) {
            if allowed {
                self.loadRecordingUI()
```

```
            } else {
                // failed to record!
            }
        }
    }
} catch {
    // failed to record!
}
```

You should replace the **// failed to record!** comment with a meaningful error alert to your user, or perhaps an on-screen label.

I made the code for **loadRecordingUI()** separate so you can replace it easily either with IB work or something else. Here's the least you need to do:

```
func loadRecordingUI() {
    recordButton = UIButton(frame: CGRect(x: 64, y: 64, width: 128, height: 64))
    recordButton.setTitle("Tap to Record", forState: .Normal)
    recordButton.titleLabel?.font = UIFont.preferredFontForTextStyle(UIFontTextStyleTitle1)
    recordButton.addTarget(self, action: "recordTapped", forControlEvents: .TouchUpInside)
    view.addSubview(recordButton)
}
```

That configures the button to call a method called **recordTapped()** when it's tapped. Don't worry, we haven't written that yet!

Before we write the code for **recordTapped()** we need to do a few other things. First, we need a method to start recording. This needs to decide where to save the audio, configure the recording settings, then start recording. Here's the code:

```swift
func startRecording() {
    let audioFilename =
getDocumentsDirectory().stringByAppendingPathComponent("recording.m4a
")

    let audioURL = NSURL(fileURLWithPath: audioFilename)

    let settings = [
        AVFormatIDKey: Int(kAudioFormatMPEG4AAC),
        AVSampleRateKey: 12000.0,
        AVNumberOfChannelsKey: 1 as NSNumber,
        AVEncoderAudioQualityKey: AVAudioQuality.High.rawValue
    ]

    do {
        audioRecorder = try AVAudioRecorder(URL: audioURL, settings:
settings)
        audioRecorder.delegate = self
        audioRecorder.record()

        recordButton.setTitle("Tap to Stop", forState: .Normal)
    } catch {
        finishRecording(success: false)
    }
}
```

That code won't build just yet, because it has two problems. First, it uses the method
getDocumentsDirectory(), which is a helper method I include in most of my projects. Here it
is:

```
class func getDocumentsDirectory() -> String {

    let paths =
NSSearchPathForDirectoriesInDomains(.DocumentDirectory, .UserDomainMa
sk, true)

    let documentsDirectory = paths[0]

    return documentsDirectory

}
```

Second, it assigns **self** to be the delegate of the audio recorder, which means you need to conform to the **AVAudioRecorderDelegate** protocol like this:

```
class ViewController: UIViewController, AVAudioRecorderDelegate {
```

With the code written to start recording, we need matching code to finish recording. This will tell the audio recorder to stop recording, then put the button title back to either "Tap to Record" (if recording finished successfully) or "Tap to Re-record" if there was a problem. Here's the code:

```
func finishRecording(success success: Bool) {

    audioRecorder.stop()

    audioRecorder = nil


    if success {

        recordButton.setTitle("Tap to Re-record", forState: .Normal)

    } else {

        recordButton.setTitle("Tap to Record", forState: .Normal)

        // recording failed :(

    }

}
```

With those two in place, we can finally write **recordTapped()**, because it just needs to call either **startRecording()** or **finishRecording()** depending on the state of the audio recorder. Here's the code:

```
func recordTapped() {
    if audioRecorder == nil {
        startRecording()
    } else {
        finishRecording(success: true)
    }
}
```

Before you're done, there's one more thing to be aware of: iOS might stop your recording for some reason out of your control, such as if a phone call comes in. We are the delegate of the audio recorder, so if this situation crops up you'll be sent a **audioRecorderDidFinishRecording()** message that you can pass on to **finishRecording()** like this:

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

For more information see Hacking with Swift tutorial 33.

How to record user videos using ReplayKit
Availability: iOS 9.0 or later.

ReplayKit is one of the great new features introduced in iOS 9.0, and it's trivial to add to your projects. What's more, it's not just for games – you can record any kind of app just fine. I should add, though, that the recording quality is fairly low, so it's not worth trying to record fine details.

Below is a complete example of how to use ReplayKit to record the screen. You'll need to add a navigation controller around your view controller because the code uses the right bar button item to start and stop recording, and of course it's down to you to add some interesting user interface that's actually worth recording!

```swift
import ReplayKit

import UIKit


class ViewController: UIViewController,
RPPreviewViewControllerDelegate {

  override func viewDidLoad() {

    super.viewDidLoad()


    navigationItem.rightBarButtonItem = UIBarButtonItem(title:
"Start", style: .Plain, target: self, action: "startRecording")

  }


  func startRecording() {

    let recorder = RPScreenRecorder.sharedRecorder()


    recorder.startRecordingWithMicrophoneEnabled(true) { [unowned
self] (error) in

      if let unwrappedError = error {

        print(unwrappedError.localizedDescription)

      } else {

        self.navigationItem.rightBarButtonItem =
UIBarButtonItem(title: "Stop", style: .Plain, target: self, action:
"stopRecording")

      }
```

```
        }

    }


    func stopRecording() {

        let recorder = RPScreenRecorder.sharedRecorder()


        recorder.stopRecordingWithHandler { [unowned self] (preview,
error) in

            self.navigationItem.rightBarButtonItem = UIBarButtonItem(title:
"Start", style: .Plain, target: self, action: "startRecording")


            if let unwrappedPreview = preview {

                unwrappedPreview.previewControllerDelegate = self

                self.presentViewController(unwrappedPreview, animated: true,
completion: nil)

            }

        }

    }


    func previewControllerDidFinish(previewController:
RPPreviewViewController) {

        dismissViewControllerAnimated(true, completion: nil)

    }

}
```

ReplayKit does three more cool things for you:

• It automatically asks the user for permission to start recording the first time you try.
• While recording, no system user interface is shown, which means you can't see messages coming in from other apps.
• When the user stops recording, they'll get the chance to preview their recording, make changes, save it to their devices and share it – your app doesn't get access to the recording.

How to render a UIView to a UIImage

Availability: iOS 7.0 or later.

You can render any **UIView** into a **UIImage** in just four lines of code, and that even handles drawing all the subviews automatically. Here's the code:

```
UIGraphicsBeginImageContextWithOptions(view.bounds.size, true, 0)
view.drawViewHierarchyInRect(view.bounds, afterScreenUpdates: true)
let image = UIGraphicsGetImageFromCurrentImageContext()
UIGraphicsEndImageContext()
```

Helpfully, that code works equally well no matter what the view contains - if you're using UIKit, SpriteKit, Metal or whatever, it all works.

How to save a UIImage to a file using UIImagePNGRepresentation

Availability: iOS 2.0 or later.

If you've generated an image using Core Graphics, or perhaps rendered part of your layout, you might want to save that out as either a PNG or a JPEG. Both are easy thanks to two functions: **UIImagePNGRepresentation()** and **UIImageJPEGRepresentation()**, both of which convert a **UIImage** into an **NSData** so you can write it out.

Here's an example:

```
if let image = UIImage(named: "example.png") {
  if let data = UIImagePNGRepresentation(image) {
    let filename =
getDocumentsDirectory().stringByAppendingPathComponent("copy.png")
    data.writeToFile(filename, atomically: true)
  }
```

```
}
```

That call to **getDocumentsDirectory()** is a little helper function I include in most of my projects, because it makes it easy to locate the user's documents directory where you can save app files. Here it is:

```
func getDocumentsDirectory() -> NSString {

    let paths =
NSSearchPathForDirectoriesInDomains(.DocumentDirectory, .UserDomainMask, true)

    let documentsDirectory = paths[0]

    return documentsDirectory

}
```

If you want to save your image as a JPEG rather than a PNG, use this code instead:

```
if let image = UIImage(named: "example.png") {

  if let data = UIImageJPEGRepresentation(image, 0.8) {

    let filename =
getDocumentsDirectory().stringByAppendingPathComponent("copy.png")

    data.writeToFile(filename, atomically: true)

  }

}
```

The second parameter to **UIImageJPEGRepresentation()** is a float that represents JPEG quality, where 1.0 is highest and 0.0 is lowest.

For more information see Hacking with Swift tutorial 10.

How to scan a QR code

Availability: iOS 8.0 or later.

iOS has built-in support for scanning QR codes using AVFoundation, but the code isn't easy: you need to create a capture session, create a preview layer, handle delegate callbacks, and more. To make it easier for you, I've created a **UIViewController** subclass that does all the hard work for you – you just need to modify the **foundCode()** method to do something more interesting.

Note: rotation when using the camera can be quite ugly, which is why most apps fix the orientation as you see below.

```
import AVFoundation

import UIKit


class ScannerViewController: UIViewController,
AVCaptureMetadataOutputObjectsDelegate {

  var captureSession: AVCaptureSession!

  var previewLayer: AVCaptureVideoPreviewLayer!


  override func viewDidLoad() {

    super.viewDidLoad()


    view.backgroundColor = UIColor.blackColor()

    captureSession = AVCaptureSession()


    let videoCaptureDevice =
AVCaptureDevice.defaultDeviceWithMediaType(AVMediaTypeVideo)

    let videoInput: AVCaptureDeviceInput


    do {

      videoInput = try AVCaptureDeviceInput(device:
videoCaptureDevice)
```

```
    } catch {

      return

    }


    if (captureSession.canAddInput(videoInput)) {

      captureSession.addInput(videoInput)

    } else {

      failed();

      return;

    }


    let metadataOutput = AVCaptureMetadataOutput()


    if (captureSession.canAddOutput(metadataOutput)) {

      captureSession.addOutput(metadataOutput)


      metadataOutput.setMetadataObjectsDelegate(self, queue:
dispatch_get_main_queue())

      metadataOutput.metadataObjectTypes =
[AVMetadataObjectTypeQRCode]

    } else {

      failed()

      return

    }


    previewLayer = AVCaptureVideoPreviewLayer(session:
captureSession);

    previewLayer.frame = view.layer.bounds;

    previewLayer.videoGravity = AVLayerVideoGravityResizeAspectFill;

    view.layer.addSublayer(previewLayer);
```

```
    captureSession.startRunning();

}


func failed() {

    let ac = UIAlertController(title: "Scanning not supported",
message: "Your device does not support scanning a code from an item.
Please use a device with a camera.", preferredStyle: .Alert)

    ac.addAction(UIAlertAction(title: "OK", style: .Default, handler:
nil))

    presentViewController(ac, animated: true, completion: nil)

    captureSession = nil

}


override func viewWillAppear(animated: Bool) {

    super.viewWillAppear(animated)


    if (captureSession?.running == false) {

      captureSession.startRunning();

    }

}


override func viewWillDisappear(animated: Bool) {

    super.viewWillDisappear(animated)


    if (captureSession?.running == true) {

      captureSession.stopRunning();

    }

}


func captureOutput(captureOutput: AVCaptureOutput!,
didOutputMetadataObjects metadataObjects: [AnyObject]!,
fromConnection connection: AVCaptureConnection!) {
```

```
        captureSession.stopRunning()


    if let metadataObject = metadataObjects.first {

        let readableObject = metadataObject as!
AVMetadataMachineReadableCodeObject;



AudioServicesPlaySystemSound(SystemSoundID(kSystemSoundID_Vibrate))

        foundCode(readableObject.stringValue);

    }


    dismissViewControllerAnimated(true, completion: nil)

  }


  func foundCode(code: String) {

    print(code)

  }


  override func prefersStatusBarHidden() -> Bool {

    return true

  }


  override func supportedInterfaceOrientations() ->
UIInterfaceOrientationMask {

    return .Portrait

  }
}
```

How to scan a barcode

Availability: iOS 7.0 or later.

iOS supports barcode scanning out of the box, but to be honest it's not that easy to do. So, here's a complete **UIViewController** subclass that you can add to your Swift project and get immediate support with no hassle – all you need to do is update the **foundCode()** method to take some interesting action, then present this view controller when you're ready:

```swift
import AVFoundation
import UIKit


class ScannerViewController: UIViewController,
AVCaptureMetadataOutputObjectsDelegate {
  var captureSession: AVCaptureSession!
  var previewLayer: AVCaptureVideoPreviewLayer!


  override func viewDidLoad() {
    super.viewDidLoad()


    view.backgroundColor = UIColor.blackColor()
    captureSession = AVCaptureSession()


    let videoCaptureDevice =
AVCaptureDevice.defaultDeviceWithMediaType(AVMediaTypeVideo)
    let videoInput: AVCaptureDeviceInput


    do {
      videoInput = try AVCaptureDeviceInput(device:
videoCaptureDevice)
    } catch {
      return
    }


    if (captureSession.canAddInput(videoInput)) {
```

```
    captureSession.addInput(videoInput)

} else {

    failed();

    return;

}


let metadataOutput = AVCaptureMetadataOutput()


if (captureSession.canAddOutput(metadataOutput)) {

    captureSession.addOutput(metadataOutput)


    metadataOutput.setMetadataObjectsDelegate(self, queue:
dispatch_get_main_queue())

    metadataOutput.metadataObjectTypes =
[AVMetadataObjectTypeEAN8Code, AVMetadataObjectTypeEAN13Code,
AVMetadataObjectTypePDF417Code]

} else {

    failed()

    return

}


previewLayer = AVCaptureVideoPreviewLayer(session:
captureSession);

previewLayer.frame = view.layer.bounds;

previewLayer.videoGravity = AVLayerVideoGravityResizeAspectFill;

view.layer.addSublayer(previewLayer);


captureSession.startRunning();

}


func failed() {

let ac = UIAlertController(title: "Scanning not supported",
```

```
message: "Your device does not support scanning a code from an item.
Please use a device with a camera.", preferredStyle: .Alert)

    ac.addAction(UIAlertAction(title: "OK", style: .Default, handler:
nil))

    presentViewController(ac, animated: true, completion: nil)

    captureSession = nil

  }


  override func viewWillAppear(animated: Bool) {

    super.viewWillAppear(animated)


    if (captureSession?.running == false) {

      captureSession.startRunning();

    }

  }


  override func viewWillDisappear(animated: Bool) {

    super.viewWillDisappear(animated)


    if (captureSession?.running == true) {

      captureSession.stopRunning();

    }

  }


  func captureOutput(captureOutput: AVCaptureOutput!,
didOutputMetadataObjects metadataObjects: [AnyObject]!,
fromConnection connection: AVCaptureConnection!) {

    captureSession.stopRunning()


    if let metadataObject = metadataObjects.first {

      let readableObject = metadataObject as!
AVMetadataMachineReadableCodeObject;
```

```
AudioServicesPlaySystemSound(SystemSoundID(kSystemSoundID_Vibrate))

        foundCode(readableObject.stringValue);

    }


    dismissViewControllerAnimated(true, completion: nil)

  }


  func foundCode(code: String) {

    print(code)

  }


  override func prefersStatusBarHidden() -> Bool {

    return true

  }


  override func supportedInterfaceOrientations() ->
UIInterfaceOrientationMask {

    return .Portrait

  }

}
```

How to turn on the camera flashlight to make a torch

Availability: iOS 6.0 or later.

There is one simple property required to enable or disable a device's torch, but you do need
to put in some wrapper code to make it work safely. Specifically, you need to use the
lockForConfiguration() and **unlockForConfiguration()** methods of the **AVCaptureDevice**
class in order to make sure only one app can control the torch at a time.

You will need to import the AVFoundation framework, because that's where the **AVCaptureDevice** class comes from. Once that's done, add this function to your code and you're good to code:

```swift
func toggleTorch(on on: Bool) {

  let device =
AVCaptureDevice.defaultDeviceWithMediaType(AVMediaTypeVideo)


  if device.hasTorch {

    do {

      try device.lockForConfiguration()


      if on == true {

        device.torchMode = .On

      } else {

        device.torchMode = .Off

      }


      device.unlockForConfiguration()

    } catch {

      print("Torch could not be used")

    }

  } else {

    print("Torch is not available")

  }

}
```

With that, you can now turn the torch on like this:

```swift
toggleTorch(on: true)
```

UIImageWriteToSavedPhotosAlbum(): how to write to the iOS photo album

Availability: iOS 2.0 or later.

It's not hard to save an image straight to the user's photo library, but I have to admit the syntax isn't immediately obvious! iOS has a function called **UIImageWriteToSavedPhotosAlbum()** that takes four parameters: parameter one is the image to save, parameters two and three set a delegate and selector to send when the image has been written successfully, and parameter four is any additional context information you wan to send.

For example, you might use it like this:

```
UIImageWriteToSavedPhotosAlbum(yourImage, self,
"image:didFinishSavingWithError:contextInfo:", nil)
```

That will write the image to the photo library, then call a method when it completes. That method needs to be named very precisely, which is where it's easy to go wrong. Using the call above, you need to write your callback method like this:

```
func image(image: UIImage, didFinishSavingWithError error: NSError?,
contextInfo:UnsafePointer<Void>) {

    if error == nil {

        let ac = UIAlertController(title: "Saved!", message: "Your
altered image has been saved to your photos.",
preferredStyle: .Alert)

        ac.addAction(UIAlertAction(title: "OK", style: .Default,
handler: nil))

        presentViewController(ac, animated: true, completion: nil)

    } else {

        let ac = UIAlertController(title: "Save error", message:
error?.localizedDescription, preferredStyle: .Alert)
```

```
        ac.addAction(UIAlertAction(title: "OK", style: .Default,
handler: nil))

        presentViewController(ac, animated: true, completion: nil)

    }

}
```

For more information see Hacking with Swift tutorial 13.

Strings

How to capitalize words in a string using capitalizedString

Availability: iOS 2.0 or later.

Swift offers several ways of adjusting the letter case of a string, but if you're looking for title case – that is, Text Where The First Letter Of Each String Is Capitalized - then you need to use the **capitalizedString** property, like this:

```
let str = "sunday, monday, happy days"
print(str.capitalizedString)
```

How to convert a string to lowercase letters

Availability: iOS 2.0 or later.

You can convert any string to lowercase – that is, going from "HELLO" to "hello" – by accessing its **lowercaseString** property, like this:

```
let str = "Sunday, Monday, Happy Days"

print(str.lowercaseString)
```

That will output "sunday, monday, happy days" to the Xcode console.

How to convert a string to uppercase letters

Availability: iOS 2.0 or later.

If you want to convert a string to uppercase – that is, WHERE EVERY LETTER IS A CAPITAL LETTER – you should use the **uppercaseString** property of your string, like this:

```
let str = "Sunday, Monday, Happy Days"

print(str.uppercaseString)
```

That code will print "SUNDAY, MONDAY, HAPPY DAYS" into the Xcode console.

How to detect a URL in a String using NSDataDetector

Availability: iOS 4.0 or later.

The **NSDataDetector** class makes it easy to detect URLs inside a string using just a few lines of code. This example loops through all URLs in a string, printing each one out:

```
let input = "This is a test with the URL https://
www.hackingwithswift.com to be detected."

let detector = try! NSDataDetector(types:
NSTextCheckingType.Link.rawValue)

let matches = detector.matchesInString(input, options: [], range:
NSMakeRange(0, input.characters.count))


for match in matches {
```

```
    let url = (input as NSString).substringWithRange(match.range)

    print(url)

}
```

Note that it takes a shortcut by casting to an **NSString** so that **substringWithRange()** can be used – this is because the matches returned by the data detector have an **NSRange** rather than a Swift string range. If you want to do things the "official way" you should use this helper function:

```
func rangeFromNSRange(nsRange: NSRange, forString str: String) ->
Range<String.Index>? {

    if let from = String.Index(str.utf16.startIndex + nsRange.location,
within: str),

        let to = String.Index(str.utf16.startIndex + nsRange.location +
nsRange.length, within: str) {

            return from ..< to

    }


    return nil

}
```

With that in place you can now pull out a match like this:

```
let url = input.substringWithRange(rangeFromNSRange(match.range,
forString: input)!)
```

How to get the length of a string
Availability: iOS 7.0 or later.

In Swift 2 you should write **yourString.characters.count** to return the length of a string. This is different from Swift 1.2, where you used **count(yourString)**, and different again from Swift 1.0, where you used **countElements(yourString)**. Hopefully the Swift 2.0 code will stay the same for a while!

How to load a string from a file in your bundle
Availability: iOS 2.0 or later.

Although it's preferable to use Swift strings for most operations, there are some things that are just done better by good old **NSString** – not least reading text in from a file. In fact, **NSString** has a constructor just for this purpose, called **contentsOfFile**, and here it is in action:

```
if let filepath = NSBundle.mainBundle().pathForResource("example",
ofType: "txt") {

  do {

    let contents = try NSString(contentsOfFile: filepath,
usedEncoding: nil) as String

    print(contents)

  } catch {

    // contents could not be loaded

  }

} else {

  // example.txt not found!

}
```

That code loads a file called **example.txt** into a string called **contents**. Note that I'm typecasting the **NSString** to a regular Swift string so you can use it normally afterwards.

For more information see Hacking with Swift tutorial 1.

How to load a string from a website URL

Availability: iOS 2.0 or later.

It takes just a few lines of Swift code to load the contents of a website URL, but there are three things you need to be careful with:

 • Creating a **NSURL** might fail if you pass a bad site, so you need to unwrap its optional return value.
 • Koading a URL's contents might fail because the site might be down (for example), so it might throw an error. This means you need to wrap the call into a **do/catch** block.
 • Accessing network data is slow, so you really want to do this on a background thread.

Here's the code:

```
if let url = NSURL(string: "https://www.hackingwithswift.com") {
  do {
    let contents = try NSString(contentsOfURL: url, usedEncoding:
nil)
    print(contents)
  } catch {
    // contents could not be loaded
  }
} else {
  // the URL was bad!
}
```

If you want to run that on a background thread (and you really ought to!) you should either use GCD's **dispatch_async()** or **performSelectorInBackground()**.

How to loop through letters in a string

Availability: iOS 7.0 or later.

You can loop through every character in a string by using its **characters** property, which is an array containing each individual character inside a string. Thanks to Swift's extended support for international languages and emoji, this works great no matter what kind of language you're using.

This code prints out each character one at a time:

```
let str = "sunday, monday, happy days"
for char in str.characters {
  print("Found character: \(char)")
}
```

How to measure a string

Availability: iOS 7.0 or later.

Cunningly, Apple has changed the way you measure strings in Swift three times now, so you'd be forgiven for not knowing how it's done. As of Swift 2, the correct way to measure strings is like this:

```
let str = "Hello, world"
let count = str.characters.count
```

How to parse a sentence using NSLinguisticTagger

Availability: iOS 5.0 or later.

If you're looking to parse natural language entered by a user, you're looking for **NSLinguisticTagger**: it automatically recognises English words (and words in other languages too, if you ask) and tells you what kind of word it is. That is, this magic little class distinguishes between verbs, nouns, adjectives and so on, so you can focus on the

important stuff: how do I (verb) this (noun)?

Here's an example to get you started:

```
let options = NSLinguisticTaggerOptions.OmitWhitespace.rawValue |
NSLinguisticTaggerOptions.JoinNames.rawValue

let tagger = NSLinguisticTagger(tagSchemes:
NSLinguisticTagger.availableTagSchemesForLanguage("en"), options:
Int(options))


let inputString = "This is a very long test for you to try"

tagger.string = inputString


let range = NSMakeRange(0, inputString.characters.count)

tagger.enumerateTagsInRange(range, scheme:
NSLinguisticTagSchemeNameTypeOrLexicalClass, options:
NSLinguisticTaggerOptions(rawValue: options)) { tag, tokenRange,
sentenceRange, stop in

  let token = (inputString as
NSString).substringWithRange(tokenRange)

  print("\(tag): \(token)")

}
```

When you loop through the matches found by an **NSLinguisticTagger**, you get back an **NSRange** describing where in the string each item was found. This is a bit ugly because the Swift way is to use string indexes, so you need to cast the Swift string to an **NSString**.

If you want to make things slightly nicer, add this helper function:

```
func rangeFromNSRange(nsRange: NSRange, forString str: String) ->
Range<String.Index>? {

  if let from = String.Index(str.utf16.startIndex + nsRange.location,
within: str),

    let to = String.Index(str.utf16.startIndex + nsRange.location +
```

```
       nsRange.length, within: str) {

          return from ..< to

     }


     return nil

}
```

With that in place you can now pull out the token like this:

```
let token =
inputString.substringWithRange(rangeFromNSRange(tokenRange,
forString: inputString)!)
```

How to repeat a string

Availability: iOS 7.0 or later.

Swift strings have a built-in constructor that lets you create strings by repeating a character a certain number of times. You can use it like this:

```
let str = String(count: 10, repeatedValue: Character("~"))
```

If you want to repeat a string rather than just a single character, you'll want to create a function something like this:

```
extension String {
  public init?(byRepeatingString str: String, count: Int) {
    var newString = ""

    for _ in 0 ..< count {
```

```
        newString += str

    }


    self.init(newString)

  }

}
```

To call that constructor use this:

```
let repeated = String(byRepeatingString: "hello", count: 4)
```

How to reverse a string using reverse()

Availability: iOS 7.0 or later.

Reversing a string in Swift is done by using the **reverse()** method on its characters, then creating a new string out of the result. Here's the code:

```
let str = "Hello, world!"
let reversed = String(str.characters.reverse())
print(reversed)
```

That will print "!dlrow ,olleH" to the Xcode console.

How to split a string into an array: componentsSeparatedByString()

Availability: iOS 7.0 or later.

You can convert a string to an array by breaking it up by a substring using the

componentsSeparatedByString() method. For example, you can split a string up by a comma and space like this:

```
let str = "Andrew, Ben, John, Paul, Peter, Laura"
let array = str.componentsSeparatedByString(", ")
```

That will return an array of six items, one for each name.

For more information see Hacking with Swift tutorial 5.

How to test localization by setting a debug locale and double length pseudolanguage

Availability: iOS 2.0 or later.

If you want to check how your app works when running on devices with other languages, you have two options: you can either instruct the simulator to use a specific language where you have a localization in place, or you can have it use a special "Double length pseudolanguage" that basically acts as a stress test.

Both of these options live under the the scheme settings for your app, which you can get to by holding down Alt then going to the Product menu and clicking "Run…" – holding down Alt makes it say "Run…" rather than "Run", which is what triggers the scheme settings window.

In the scheme settings window, click the dropdown next to Application Language. You can either choose a language that you have localized to, or choose Double Length Pseudolanguage. This option effectively makes all your strings take up twice as much space on the screen, which shows you at a glance if your interface will cope with languages that have longer words than your own.

How to trim whitespace in a string

Availability: iOS 2.0 or later.

It's not hard to trim whitespace from a string in Swift, but the syntax is a little wordy – or

"self-descriptive" if you're feeling optimistic. You need to use the **stringByTrimmingCharactersInSet()** method and provide a list of the characters you want to trim. If you're just using whitespace (tabs, spaces and new lines) you can use the predefined **whitespaceAndNewlineCharacterSet()** list of characters, like this:

```
let str = "  Taylor Swift  "

let trimmed =
str.stringByTrimmingCharactersInSet(NSCharacterSet.whitespaceAndNewlineCharacterSet())
```

That will set **trimmed** to be "Taylor Swift".

How to use string interpolation to combine strings, integers and doubles

Availability: iOS 7.0 or later.

String interpolation is Swift's way of letting you insert variables and constants into strings. But at the same time, you can also perform simple operations as part of your interpolation, such as changing letter case and basic mathematics. Swift is also smart enough to understand *how* to bring values into strings, meaning that you can use other strings, integers and floating-point numbers just fine.

Here's an example to get you started:

```
var name = "Paul"

var age = 35

var longestPi = 3.141592654


var combined = "This person's name is \(name.uppercaseString), their
age is \(age) so in \(age) years time they'll be \(age + age), and
they know π up to \(longestPi)"
```

For more information see Hacking with Swift tutorial 0.

NSRegularExpression: How to match regular expressions in strings

Availability: iOS 4.0 or later.

The **NSRegularExpression** class lets you find and replace substrings using regular expressions, which are concise and flexible descriptions of text. For example, if we wanted to pull "Taylor Swift" out of the string "My name is Taylor Swift", we could write a regular expression that matches the text "My name is " followed by any text, then pass that to the **NSRegularExpression** class.

The example below does just that. Note that we need to pull out the second match range because the first range is the entire matched string, whereas the second range is just the "Taylor Swift" part:

```
do {

  let input = "My name is Taylor Swift"

  let regex = try NSRegularExpression(pattern: "My name is (.*)",
options: NSRegularExpressionOptions.CaseInsensitive)

  let matches = regex.matchesInString(input, options: [], range:
NSMakeRange(0, input.characters.count))


  if let match = matches.first {

    let range = match.rangeAtIndex(1)

    if let swiftRange = rangeFromNSRange(range, forString: input) {

      let name = input.substringWithRange(swiftRange)

    }

  }

} catch {

  // regex was bad!

}
```

That code uses a little helper function that you should take: **rangeFromNSRange()**. Annoyingly, regular expression matches demand Swift strings as input then return **NSString** and **NSRange** in their output. This function converts from **NSRange** to Swift string ranges:

```
func rangeFromNSRange(nsRange: NSRange, forString str: String) ->
Range<String.Index>? {

  let fromUTF16 = str.utf16.startIndex.advancedBy(nsRange.location,
limit: str.utf16.endIndex)

  let toUTF16 = fromUTF16.advancedBy(nsRange.length, limit:
str.utf16.endIndex)




  if let from = String.Index(fromUTF16, within: str),

    let to = String.Index(toUTF16, within: str) {

      return from ..< to

  }


  return nil
}
```

Replacing text in a string using stringByReplacingOccurrencesOfString()

Availability: iOS 2.0 or later.

It's easy to replace text inside a string thanks to the (exceedingly long!) method **stringByReplacingOccurrencesOfString()**. This is a string method, and you tell it what to look for and what to replace it with, and you're done.

For example:

```
let str = "Swift 1.2 is the best version of Swift to learn, so if
you're starting fresh you should definitely learn Swift 1.2."

let replaced = str.stringByReplacingOccurrencesOfString("1.2",
withString: "2.0")
```

That will make **replaced** equal to "Swift 2.0 is the best version of Swift to learn, so if you're starting fresh you should definitely learn Swift 2.0."

writeToFile(): How to save a string to a file on disk

Availability: iOS 2.0 or later.

All strings have a **writeToFile()** method that lets you save the contents of the string to disk. You need to provide a filename to write to, plus two more parameters: whether the write should be atomic, and what string encoding to use. The second parameter should nearly always be **true** because it avoids concurrency problems. The third parameter should nearly always be **NSUTF8StringEncoding**, which is pretty much the standard for reading and writing text.

Be warned: writing a string to disk can throw an exception, so you need to catch any errors and warn the user.

Here's the code:

```
let str = "Super long string here"

let filename =
getDocumentsDirectory().stringByAppendingPathComponent("output.txt")


do {

  try str.writeToFile(filename, atomically: true, encoding:
NSUTF8StringEncoding)

} catch {

  // failed to write file – bad permissions, bad filename, missing
permissions, or more likely it can't be converted to the encoding
```

```
}
```

That code uses a helper function called **getDocumentsDirectory()**, which finds the path to where you can save your app's files. Here it is:

```
func getDocumentsDirectory() -> NSString {

  let paths =
NSSearchPathForDirectoriesInDomains(.DocumentDirectory, .UserDomainMa
sk, true)

  let documentsDirectory = paths[0]

  return documentsDirectory

}
```

System

How to cache data using NSCache

Availability: iOS 4.0 or later.

Here's an easy win for you that will make your apps immediately much better: **NSCache** is a specialized class that behaves similarly to an **NSMutableDictionary** with one major difference: iOS will automatically remove objects from the cache if the device is running low on memory.

Helpfully, if the system does encounter memory pressure **NSCache** will automatically start to remove items without you knowing about it, which means you won't get a memory warning unless even more RAM needs to be cleared. It will also remove items intelligently, trying to keep as much cached as possible.

Here's how to use it, imagining a fictional class called **ExpensiveObjectClass** that you want to compute as infrequently as you can:

```
let cache = NSCache()

let myObject: ExpensiveObjectClass


if let cachedVersion = cache.objectForKey("CachedObject") as?
ExpensiveObjectClass {

  // use the cached version

  myObject = cachedVersion

} else {

  // create it from scratch then store in the cache

  myObject = ExpensiveObjectClass()

  cache.setObject(myObject, forKey: "CachedObject")

}
```

How to copy objects in Swift using copy()

Availability: iOS 7.0 or later.

There are two main complex data types in Swift – objects and structs – and they do so many things similarly that you'd be forgiven for not being sure exactly where they differ. Well, one of the key areas is down to copying: two variables can point at the same object so that changing one changes them both, whereas if you tried that with structs you'd find that Swift creates a full copy so that changing the copy does not affect the original.

Having lots of objects point at the same data can be useful, but frequently you'll want to modify *copies* so that modifying one object doesn't have an effect on anything else. To make this work you need to do three things:

 • Make your class conform to **NSCopying**. This isn't strictly required, but it makes your intent clear.
 • Implement the method **copyWithZone()**, where the actual copying happens.
 • Call **copy()** on your object.

Here's an example of a **Person** class that conforms fully to the **NSCopying** protocol:

```
class Person: NSObject, NSCopying {
  var firstName: String
  var lastName: String
  var age: Int

  init(firstName: String, lastName: String, age: Int) {
    self.firstName = firstName
    self.lastName = lastName
    self.age = age
  }

  func copyWithZone(zone: NSZone) -> AnyObject {
    let copy = Person(firstName: firstName, lastName: lastName, age:
age)
    return copy
  }
}
```

Note that **copyWithZone()** is implemented by creating a new **Person** object using the current person's information.

With that done, you can test out your copying like this:

```
let paul = Person(firstName: "Paul", lastName: "Hudson", age: 35)
let sophie = paul.copy() as! Person

sophie.firstName = "Sophie"
sophie.age = 5
```

```
print("\(paul.firstName) \(paul.lastName) is \(paul.age)")

print("\(sophie.firstName) \(sophie.lastName) is \(sophie.age)")
```

How to copy text to the clipboard using UIPasteboard
Availability: iOS 3.0 or later.

You can write to and read from the iOS clipboard by using the **UIPasteboard** class, which has a **generalPasteboard()** method that returns the shared system method of copying and pasting data between apps. Using this you can write text to the clipboard just like this:

```
let pasteboard = UIPasteboard.generalPasteboard()

pasteboard.string = "\(number)"
```

To read text back from the clipboard, you should unwrap its optional value like this:

```
let pasteboard = UIPasteboard.generalPasteboard()

if let string = pasteboard.string {

  // text was found and placed in the "string" constant

}
```

How to create a peer-to-peer network using the multipeer connectivity framework
Availability: iOS 7.0 or later.

The **MultipeerConnectivity** framework is designed to allow ad hoc data transfer between

devices that are in close proximity. The connection is started managed for you by iOS, but you're responsible for presenting useful interface to your users and for understanding the data that is being sent and received.

First things first, import the **MultipeerConnectivity** framework:

```
import MultipeerConnectivity
```

Now define these three properties to hold the multipeer session information:

```
var peerID: MCPeerID!

var mcSession: MCSession!

var mcAdvertiserAssistant: MCAdvertiserAssistant!
```

The peer ID is simply the name of the current device, which is useful for identifying who is joining a session. We're just going to use the current device's name when creating our connection, but we're also going to require encryption. Add this to your **viewDidLoad()** method:

```
peerID = MCPeerID(displayName: UIDevice.currentDevice().name)

mcSession = MCSession(peer: peerID, securityIdentity: nil,
encryptionPreference: .Required)

mcSession.delegate = self
```

You will need to tell iOS that your view controller conforms to the **MCSessionDelegate** and **MCBrowserViewControllerDelegate** delegates, like this:

```
class ViewController: UIViewController, MCSessionDelegate,
MCBrowserViewControllerDelegate {
```

Conforming to those two delegates means implementing quite a few methods. Fortunately, five of them are super simple because three are empty and the other two just dismiss a view controller. Add this code now:

```
func session(session: MCSession!, didReceiveStream stream:
NSInputStream!, withName streamName: String!, fromPeer peerID:
MCPeerID!) {


}


func session(session: MCSession!, didStartReceivingResourceWithName
resourceName: String!, fromPeer peerID: MCPeerID!, withProgress
progress: NSProgress!) {


}


func session(session: MCSession!, didFinishReceivingResourceWithName
resourceName: String!, fromPeer peerID: MCPeerID!, atURL localURL:
NSURL!, withError error: NSError!) {


}


func browserViewControllerDidFinish(browserViewController:
MCBrowserViewController!) {

    dismissViewControllerAnimated(true, completion: nil)

}


func browserViewControllerWasCancelled(browserViewController:
MCBrowserViewController!) {

    dismissViewControllerAnimated(true, completion: nil)

}
```

Next comes a slightly harder method: you need to do something when clients connect or

disconnect. That something could just be "I don't care; do nothing," or it might be where you show a message on the screen to tell your user. Here's a basic version that just prints out a status message to the Xcode log:

```
func session(session: MCSession!, peer peerID: MCPeerID!,
didChangeState state: MCSessionState) {

    switch state {

    case MCSessionState.Connected:

        println("Connected: \(peerID.displayName)")


    case MCSessionState.Connecting:

        println("Connecting: \(peerID.displayName)")


    case MCSessionState.NotConnected:

        println("Not Connected: \(peerID.displayName)")

    }

}
```

Time for the important stuff: sending and receiving data. Now, obviously the data you will send and receive depends on what your app does, so you will need to customize this code to fit your needs. In the example I'm going to give, we'll use sending and receiving images, but you could just as easily send strings or anything else.

So, here's how to encode an image into an **NSData** then send that to all connected peers:

```
func sendImage(img: UIImage) {

  if mcSession.connectedPeers.count > 0 {

    if let imageData = UIImagePNGRepresentation(img) {

      do {

        try mcSession.sendData(imageData, toPeers:
mcSession.connectedPeers, withMode: .Reliable)
```

```
        } catch let error as NSError {

            let ac = UIAlertController(title: "Send error", message:
error.localizedDescription, preferredStyle: .Alert)

            ac.addAction(UIAlertAction(title: "OK", style: .Default,
handler: nil))

            presentViewController(ac, animated: true, completion: nil)

        }

    }

  }

}
```

To receive that on the other side, you need a method like this:

```
func session(session: MCSession!, didReceiveData data: NSData!,
fromPeer peerID: MCPeerID!) {

    if let image = UIImage(data: data) {

        dispatch_async(dispatch_get_main_queue()) { [unowned self] in

            // do something with the image

        }

    }

}
```

Note that I've explicitly pushed the work to the main thread so that you're safe to do UI work.

All that remains now is to either host a session or join a session. Add these two methods to your code, then call whichever one you need:

```
func startHosting(action: UIAlertAction!) {

    mcAdvertiserAssistant = MCAdvertiserAssistant(serviceType: "hws-
kb", discoveryInfo: nil, session: mcSession)

    mcAdvertiserAssistant.start()
```

```
}

func joinSession(action: UIAlertAction!) {
    let mcBrowser = MCBrowserViewController(serviceType: "hws-kb",
session: mcSession)

    mcBrowser.delegate = self

    presentViewController(mcBrowser, animated: true, completion: nil)
}
```

Note: to test this code you'll need either two iOS devices or one device and the simulator.

For more information see Hacking with Swift tutorial 25.

How to create rich formatted text strings using NSAttributedString

Availability: iOS 6.0 or later.

Attributed strings are strings with formatting attached, which means fonts, colors, alignment, line spacing and more. They are supported in many places around iOS, which means you can assign a fully formatted string to a **UILabel** and have it look great with no further work.

Please keep in mind, when working with fonts it's preferable to use Dynamic Type where possible so that a user's font size settings are honored. The example code below creates an attributed string using the "Headline" Dynamic Type size, then colors it purple. That is then placed into a **UILabel** by setting its **attributedText** property:

```
let titleAttributes = [NSFontAttributeName:
UIFont.preferredFontForTextStyle(UIFontTextStyleHeadline),
NSForegroundColorAttributeName: UIColor.purpleColor()]

let titleString = NSAttributedString(string: "Read all about it!",
attributes: titleAttributes)

myLabel.attributedText = titleString
```

For more information see Hacking with Swift tutorial 32.

How to detect when your app moves to the background

Availability: iOS 4.0 or later.

There are two ways to be notified when your app moves to the background: implement the **applicationWillResignActive()** method in your app delegate, or register for the **UIApplicationWillResignActiveNotification** notification anywhere in your app. This particular notification is sent as soon as your app loses focus, meaning that it's triggered when the user taps the home button once (to return to the home screen) or double taps the home button (to enter multi-tasking).

If you want to go down the app delegate route, you'll find a stub for **applicationWillResignActive()** already in your AppDelegate.swift file. If you want to look for the notification, use this:

```
override func viewDidLoad() {

  let notificationCenter = NSNotificationCenter.defaultCenter()

  notificationCenter.addObserver(self, selector:
"appMovedToBackground", name:
UIApplicationWillResignActiveNotification, object: nil)

}


func appMovedToBackground() {

  print("App moved to background!")

}
```

For more information see Hacking with Swift tutorial 28.

How to detect which country a user is in

Availability: iOS 2.0 or later.

Being able to provide users with location-specific information immediately makes your app more useful, but asking for a precise location brings up a permission alert and might make them suspicious. Fortunately there's a coarse-grained way you can figure out a user's locate without asking for location permission: **NSLocale**.

A *locale* is a user's region setting on their device, and you can read it without asking for permission. For example, if the locale is en-US it means they speak English and are in the US; if it's fr-CA it means they speak French are in Canada. This is all wrapped up inside **NSLocale** and you can query various information from it, but for our simple purpose we're just going to ask what country the user is in:

```
let locale = NSLocale.currentLocale()
if let country = locale.objectForKey(NSLocaleCountryCode) as? String
{
  if country == "US" {
    print("An American")
  } else {
    print("Not an American")
  }
}
```

Now, there is a catch, but this is actually a bonus feature in my eyes: if a user travels abroad, their device will still be configured for their home country, so an American visting France will still say "US". Yes, that means you can't use it for location information, but actually it works out better for a lot of apps – for example, why would an American want to see distances in metres rather than miles just because they are traveling?

How to find the user's documents directory

Availability: iOS 2.0 or later.

Every iOS app gets a slice of storage just for itself, meaning that you can read and write your

app's files there without worrying about colliding with other apps. This is called the user's documents directory, and it's exposed both in code (as you'll see in a moment) and also through iTunes file sharing.

Unfortunately, the code to find the user's documents directory isn't very memorable, so I nearly always use this helpful function – and now you can too!

```
func getDocumentsDirectory() -> String {

    let paths =
NSSearchPathForDirectoriesInDomains(.DocumentDirectory, .UserDomainMask, true)

    let documentsDirectory = paths[0]

    return documentsDirectory

}
```

For more information see Hacking with Swift tutorial 10.

How to format dates with an ordinal suffix using NSNumberFormatter's OrdinalStyle

Availability: iOS 9 or later.

As of iOS 9.0, Apple introduced a simple way to make ordinal style numbers, which is a fancy way of saying 1st, 2nd, 3rd or 100th – the kind of numbers you normally write for dates, for example. This uses the new **NSNumberFormatterStyle.OrdinalStyle** style of writing numbers with **NSNumberFormatter**, like this:

```
let formatter = NSNumberFormatter()

formatter.numberStyle = .OrdinalStyle

let first = formatter.stringFromNumber(1)

let second = formatter.stringFromNumber(2)

let tenth = formatter.stringFromNumber(10)

let oneThousandAndFirst = formatter.stringFromNumber(1001)
```

How to generate a random identifier using NSUUID

Availability: iOS 6.0 or later.

A UUID is a *universally unique identifier*, which means if you generate a UUID right now using **NSUUID** it's guaranteed to be unique across all devices in the world. This means it's a great way to generate a unique identifier for users, for files, or anything else you need to reference individually – guaranteed.

Here's how to create a UUID as a string:

```
let uuid = NSUUID().UUIDString
```

How to generate random numbers in iOS 8 and below

Availability: iOS 2.0 or later.

iOS 9.0 introduces Gameplay and its great new random tools, but if you have to support iOS 8.0 and earlier here's the code you're looking for:

```
let randNum = arc4random_uniform(6)
```

That will generate a number between 0 and 5. If you're looking for a random number between any two numbers, try this helper function:

```
func RandomInt(min min: Int, max: Int) -> Int {
    if max < min { return min }
    return Int(arc4random_uniform(UInt32((max - min) + 1))) + min
}
```

That generates numbers inclusive, meaning that **RandomInt(0, 6)** can return 0, 6, or any value in between.

For more information see Hacking with Swift tutorial 35.

How to handle the HTTPS requirements in iOS 9 with App Transport Security

Availability: iOS 9.0 or later.

As of iOS 9.0, you can't work with HTTP web data by default, because it's blocked by something called App Transport Security that effectively requires data to be transmitted securely. If possible, you should switch to HTTPS and use that instead, but if that's not possible for some reason – e.g. if you're working with a third-party website – then you need to tell iOS to make exceptions for you.

Note: the very fact that iOS calls these "exceptions" does imply the exception option may go away in the future. Exceptions be defined per-site or globally, although if you're going to make exceptions obviously it's preferable to do it for individual sites. This is all set inside your application's Info.plist file, and this is one of the very few times when editing your plist as source code is faster than trying to use the GUI editor in Xcode. So, right-click on your Info.plist and choose Open As > Source Code.

Your plist should end like this:

```
</dict>
</plist>
```

Just before that, I'd like you to paste this:

```
<key>NSAppTransportSecurity</key>
<dict>
```

```
        <key>NSExceptionDomains</key>

        <dict>

          <key>hackingwithswift.com</key>

          <dict>

            <key>NSIncludesSubdomains</key>

            <true/>

            <key>NSThirdPartyExceptionAllowsInsecureHTTPLoads</key>

            <true/>

          </dict>

        </dict>

</dict>
```

That requests an exception for the site **hackingwithswift.com** so that it can be loaded using regular HTTP rather than HTTPS. Note that I've set **NSIncludesSubdomains** to be **true** because the site redirects you to **www.hackingwithswift.com**, which is a subdomain.

(Very observant readers might note that **hackingwithswift.com** actually supports HTTPS and thus doesn't need App Transport Security, but you do still need to point to **https://** otherwise the request will fail.)

If you want to add a global exception – effectively to mimic pre-iOS 9.0 behavior – you'll want to use this instead:

```
<key>NSAppTransportSecurity</key>

<dict>

  <key>NSAllowsArbitraryLoads</key>

  <true/>

</dict>
```

Again, let me advise caution: these exceptions could go away in any future release, so please don't rely extensively on them.

How to identify an iOS device uniquely with identifierForVendor

Availability: iOS 6.0 or later.

Early iOS releases gave every device a unique identifier, but this was soon abused by developers to identify individual users uniquely – something that Apple really dislikes. So, as of iOS 6.0 Apple removed the truly unique identifier and instead introduced an identifier for each vendor: a UUID that's the same for all apps for a given developer for each user, but varies between developers and between devices.

That is, if a user has five of your apps installed and five of mine, your five will all share the same vendor identifier, and my five will all share the same vendor identifier, but our two identifiers will be different.

Here's how to use it:

```
if let uuid =
UIDevice.currentDevice().identifierForVendor?.UUIDString {

    print(uuid)

}
```

How to make an action repeat using NSTimer

Availability: iOS 2.0 or later.

Timers are a great way to run code on a repeating basis, and iOS has the **NSTimer** class to handle it for you. First, create a property of the type **NSTimer!**. For example:

```
var gameTimer: NSTimer!
```

You can then create that timer and tell it to execute every five seconds, like this:

```
gameTimer = NSTimer.scheduledTimerWithTimeInterval(5, target: self,
selector: "runTimedCode", userInfo: nil, repeats: true)
```

The **runTimedCode** selector means that the timer will call a method named **runTimedCode()** every five seconds until the timer is terminated, so you'll need to add this method to your class:

```
func runTimedCode() {


}
```

Important note: because your object has a property to store the timer, and the timer calls a method on the object, you have a strong reference cyle that means neither object can be freed. To fix this, make sure you invalidate the timer when you're done with it, such as when your view is about to disappear:

```
gameTimer.invalidate()
```

For more information see Hacking with Swift tutorial 20.

How to make tappable links in NSAttributedString
Availability: iOS 6.0 or later.

You can make tappable hyperlinks in any attributed string, which in turn means you can add tappable hyperlinks to any UIKit control. If you're working with **UITextView** (which is likely, let's face it), you get basic tappable hyperlink just by enabling the "Links" data detector in Interface Builder, but that doesn't work for arbitrary strings – for example, maybe you want the word "click here" to tappable.

Below is a complete example of arbitrary tappable hyperlinks using a **UITextView**. Make sure your text view has "Selectable" enabled, as this is required by iOS:

```
class ViewController: UIViewController, UITextViewDelegate {

  @IBOutlet var textView: UITextView!


  override func viewDidLoad() {

    let attributedString = NSMutableAttributedString(string: "Want to
learn iOS? You should visit the best source of free iOS tutorials!")

    attributedString.addAttribute(NSLinkAttributeName, value:
"https://www.hackingwithswift.com", range: NSMakeRange(19, 55))


    textView.attributedText = attributedString

  }


  func textView(textView: UITextView, shouldInteractWithURL URL:
NSURL, inRange characterRange: NSRange) -> Bool {

    UIApplication.sharedApplication().openURL(URL)

    return false

  }

}
```

There are two important things to note about this technique. First, the tap cannot be very brief, which means quick taps are ignored by iOS. If you find find this annoying you might consider something like this: **https://gist.github.com/benjaminbojko/ c92ac19fe4db3302bd28**. Second, this technique is easily used with custom URL schemes, e.g. yourapp://, which you can catch and parse inside **shouldInteractWithURL** to trigger your own behaviors.

How to open a URL in Safari
Availability: iOS 2.0 or later.

If you want the user to exit your app and show a website in Safari, it's just one line of code in Swift. I'll make it three here because I'll create the URL in the code too, then safely unwrap it:

```
if let url = NSURL(string: "https://www.hackingwithswift.com") {
  UIApplication.sharedApplication().openURL(url)
}
```

It's worth adding that since iOS 9 you have the option of using **SFSafariViewController** inside your app, which recreates the entire Safari experience right inside your app. See project 32 for a tutorial on how to do this.

How to parse JSON using NSJSONSerialization

Availability: iOS 5.0 or later.

The built-in iOS way of parsing JSON is called **NSJSONSerialization** and it can convert a JSON string into a collection of dictionaries, arrays, strings and numbers in just a few lines of code.

In the example below, I create a dummy piece of JSON that contains three names in an array cunningly called "names". This then gets sent to **NSJSONSerialization** (via convert it into an **NSData**, which is how **NSJSONSerialization** likes to receive its content), and I conditionally pull out and print the **names** array:

```
let str = "{\"names\": [\"Bob\", \"Tim\", \"Tina\"]}"

let data = str.dataUsingEncoding(NSUTF8StringEncoding,
allowLossyConversion: false)!


do {
  let json = try NSJSONSerialization.JSONObjectWithData(data,
options: []) as! [String: AnyObject]

  if let names = json["names"] as? [String] {

    print(names)
```

```
    }

} catch let error as NSError {

    print("Failed to load: \(error.localizedDescription)")

}
```

There are a couple of things that might confuse you there. First, because parsing JSON will fail if the JSON isn't valid, you need to use try/catch and have some sort of error handling. Second, you need to force typecast the JSON to be a dictionary of type **[String: AnyObject]** so that you can start working with your JSON values. Third, you don't know for sure that any values exist inside the JSON, so you need to conditionally check for and unwrap the **names** value.

For more information see Hacking with Swift tutorial 7.

How to post messages using NSNotificationCenter
Availability: iOS 2.0 or later.

iOS notifications are a simple and powerful way to send data in a loosely coupled way. That is, the sender of a notification doesn't have to care about who (if anyone) receives the notification, it just posts it out there to the rest of the app and it could be picked up by lots of things or nothing depending on your app's state.

As a basic example, you might want various parts of your app to do some work when the user logs in – you might want some views to refresh, you might want a database to update itself, and so on. To do this, just post a notification name like this:

```
let nc = NSNotificationCenter.defaultCenter()

nc.postNotificationName("UserLoggedIn", object: nil)
```

Note: it is preferable, for type safety, to define your notification names as static strings that belong to a class or struct or other global form so that you don't make a typo and introduce bugs.

To register to catch a notification being posted, use this:

```
nc.addObserver(self, selector: "userLoggedIn", name: "UserLoggedIn",
object: nil)
```

That will call a **userLoggedIn()** method when your notification is posted.

How to read the contents of a directory using NSFileManager
Availability: iOS 2.0 or later.

If you want to work with files **NSFileManager** almost certainly has the answer, and it's no different in this case: it has a method called **contentsOfDirectoryAtPath()** that lists all the files in a specific directory. For example, we could have it list all the files in our app's resource directory like this:

```
let fm = NSFileManager.defaultManager()

let path = NSBundle.mainBundle().resourcePath!


do {

  let items = try fm.contentsOfDirectoryAtPath(path)


  for item in items {

    print("Found \(item)")

  }
} catch {

  // failed to read directory – bad permissions, perhaps?

}
```

In this particular case the **try** should never fail, but you should still have the **catch** block in there just in case.

For more information see Hacking with Swift tutorial 1.

How to run code after a delay using dispatch_after() and performSelector

Availability: iOS 4.0 or later.

There are two ways to run code after a delay using Swift: GCD and
performSelector(_:withObject:afterDelay:). The first one has some clumsy syntax because
you need to specify a time in nanoseconds – most people just use the constant
NSEC_PER_SEC to let them specify seconds instead.

The easiest way to use GCD to run code after a delay is by using a helper function like this
one:

```
func runAfterDelay(delay: NSTimeInterval, block: dispatch_block_t) {
  let time = dispatch_time(DISPATCH_TIME_NOW, Int64(delay *
Double(NSEC_PER_SEC)))
  dispatch_after(time, dispatch_get_main_queue(), block)
}
```

With that function in place you can now use code like this:

```
runAfterDelay(0.5) {
  yourCodeHere()
}
```

An alternative option is to use **performSelector(_:withObject:afterDelay:)**, which lets you
specify a method to call after a certain time has elapsed – and helpfully that time is specified
in seconds, which makes it easier to remember!

To call the **yourCodeHere()** method after 1 second, you would use this code:

```
performSelector("yourCodeHere", withObject: nil, afterDelay: 1)
```

How to run code asynchronously using dispatch_async()

Availability: iOS 4.0 or later.

iOS gives you two ways to run code asynchronously: GCD and
performSelectorInBackground(). The first option looks like this:

```
dispatch_async(dispatch_get_global_queue(QOS_CLASS_USER_INITIATED,
0)) {
    runSomeCodeHere()
}
```

The **QOS_CLASS_USER_INITIATED** quality of service setting is the highest priority after
QOS_CLASS_USER_INTERACTIVE. You can also use **QOS_CLASS_UTILITY** (lower
priority) or **QOS_CLASS_BACKGROUND** (lowest priority.)

The second option looks like this:

```
performSelectorInBackground("runSomeCodeHere", withObject: nil)
```

You'll need to replace "runSomeCodeHere" with the name of an actual method. If you want
to pass a parameter, make sure and use "runSomeCodeHere:" and provide a value for the
withObject parameter.

For more information see Hacking with Swift tutorial 9.

How to run code at a specific time

Availability: iOS 2.0 or later.

You can use **performSelector(_:withObject:afterDelay:)** to run a method after a certain number of seconds have passed, but if you want to run code at a specific time – say at exactly 4pm – then you should use **NSTimer** instead. This class is great for executing code repeatly at a specific time interval, but it's also great for running code at an exact time that you specify.

This is accomplished using an **NSTimer** constructor that accepts an **NSDate** for when the timer should fire. You can make this date however you want, which is what makes this approach so flexible.

As a simple example, this will create a timer that calls a **runCode()** method in five seconds:

```
let date = NSDate().dateByAddingTimeInterval(5)

let timer = NSTimer(fireDate: date, interval: 0, target: self,
selector: "runCode", userInfo: nil, repeats: false)

NSRunLoop.mainRunLoop().addTimer(timer, forMode:
NSRunLoopCommonModes)
```

Notice how you can specify a **interval** parameter? That's for when you set **repeats** to be **true**. If you have an **NSDate** 5 seconds from now and an **interval** of 1 (after setting **repeat** to be true!), it means "call **runCode()** after five seconds, then every one second after that."

How to run code on the main thread using dispatch_async()

Availability: iOS 4.0 or later.

Swift offers you two ways to run code on the main thread: GCD and **performSelectorOnMainThread**. The first option looks like this:

```
dispatch_async(dispatch_get_main_queue()) {

  yourCodeHere()

}
```

The second option looks like this:

```
performSelectorOnMainThread("yourCodeHere", withObject: nil,
waitUntilDone: false)
```

The GCD option (the first one) has the advantage that you can write your code inline, whereas the second one requires a dedicated method you can call.

For more information see Hacking with Swift tutorial 9.

How to save and load objects with NSKeyedArchiver and NSKeyedUnarchiver

Availability: iOS 2.0 or later.

You can write any kind of object to disk as long as it supports the **NSCoding** protocol – which includes strings, arrays, dictionaries, **UIView**, **UIColor** and more right out of the box. To write to disk, use this:

```
let data = NSKeyedArchiver.archivedDataWithRootObject(self)

let filename =
getDocumentsDirectory().stringByAppendingPathComponent(uuid)


data.writeToFile(filename, atomically: true)
```

That call to **getDocumentsDirectory()** is a small helper function I frequently use to write files to disk:

```
func getDocumentsDirectory() -> String {

    let paths =
NSSearchPathForDirectoriesInDomains(.DocumentDirectory, .UserDomainMa
sk, true)

    let documentsDirectory = paths[0]

    return documentsDirectory

}
```

When you want to read your object back you need to use **unarchiveObjectWithFile()**, but be warned: the file might not exist or might not be unarchivable, so this method returns an optional value that you need to unwrap carefully.

For example, if you want to unarchive an array of strings, you would use this code:

```
if let names = NSKeyedUnarchiver.unarchiveObjectWithFile(filename)
as? [String] {

  savedNames = names

}
```

How to save user settings using NSUserDefaults
Availability: iOS 2.0 or later.

All iOS apps have a built in data dictionary that stores small amounts of user settings for as long as the app is installed. This system, called **NSUserDefaults** can save integers, booleans, strings, arrays, dictionaries, dates and more, but you should be careful not to save too much data because it will slow the launch of your app.

Here's an example of setting some values:

```
let defaults = NSUserDefaults.standardUserDefaults()
```

```
defaults.setInteger(25, forKey: "Age")

defaults.setBool(true, forKey: "UseTouchID")

defaults.setDouble(M_PI, forKey: "Pi")


defaults.setObject("Paul Hudson", forKey: "Name")

defaults.setObject(NSDate(), forKey: "LastRun")
```

When you set values like that, they become permanent – you can quit the app then re-launch and they'll still be there, so it's the ideal way to store app configuration data.

As mentioned, you can use **NSUserDefaults** to store arrays and dictionaries, like this:

```
let array = ["Hello", "World"]

defaults.setObject(array, forKey: "SavedArray")


let dict = ["Name": "Paul", "Country": "UK"]

defaults.setObject(dict, forKey: "SavedDict")
```

When it comes to reading data back, it's still easy but has an important proviso: **NSUserDefaults** will return a default value if the setting can't be found. You need to know what these default values are so that you don't confuse them with real values that you set. Here they are:

- **integerForKey()** returns an integer if the key existed, or 0 if not.
- **boolForKey()** returns a boolean if the key existed, or false if not.
- **floatForKey()** returns a float if the key existed, or 0.0 if not.
- **doubleForKey()** returns a double if the key existed, or 0.0 if not.
- **objectForKey()** returns AnyObject? so you need to conditionally typecast it to your data type.

With that in mind, you can read values back like this:

```
let defaults = NSUserDefaults.standardUserDefaults()


let age = defaults.integerForKey("Age")

let useTouchID = defaults.boolForKey("UseTouchID")

let pi = defaults.doubleForKey("Pi")
```

When retrieving objects, the result is optional. This means you can either accept the optionality, or typecast it to a non-optional type and use the nil coalescing operator to handle missing values. For example:

```
let array = defaults.objectForKey("SavedArray") as? [String] ??
[String]()
```

For more information see Hacking with Swift tutorial 12.

How to set local alerts using UILocalNotification

Availability: iOS 8.0 or later.

Local notifications are messages that appear on the user's lock screen when your app isn't running. The user can then swipe to unlock their device and go straight to your app, at which point you can act on the notification.

As you might imagine, you need to ask for permission in order to show messages on the lock screen. Here's how that's done:

```
let notificationSettings =
UIUserNotificationSettings(forTypes: .Alert | .Badge | .Sound,
categories: nil)

UIApplication.sharedApplication().registerUserNotificationSettings(no
tificationSettings)
```

That will show an alert to the user asking them if they want to let you show notifications. When it comes to scheduling a notification, you should check your permission setting to see whether you have permission before you try to do anything.

Here's the code required to show a local notification, including how to first check you have permission:

```
func scheduleLocal(sender: AnyObject) {

  let settings =
UIApplication.sharedApplication().currentUserNotificationSettings()


  if settings.types == .None {

      let ac = UIAlertController(title: "Can't schedule", message:
"Either we don't have permission to schedule notifications, or we
haven't asked yet.", preferredStyle: .Alert)

      ac.addAction(UIAlertAction(title: "OK", style: .Default,
handler: nil))

      presentViewController(ac, animated: true, completion: nil)

      return

  }


    let notification = UILocalNotification()

    notification.fireDate = NSDate(timeIntervalSinceNow: 5)

    notification.alertBody = "Hey you! Yeah you! Swipe to unlock!"

    notification.alertAction = "be awesome!"

    notification.soundName = UILocalNotificationDefaultSoundName

    notification.userInfo = ["CustomField1": "w00t"]

UIApplication.sharedApplication().scheduleLocalNotification(notificat
ion)

}
```

The **fireDate** property means this notification will appear five seconds after you schedule it,

although as you might imagine you probably want to specify an exact date in here. The **alertAction** property should complete the string "Swipe to…", so in this case it will read "Swipe to be awesome!"

That code also sets a **userInfo** property on the notification, which is a dictionary where you can store any kind of context data you want. This dictionary gets given back to you when the user unlocks their device using your notification, so you can act on the alert in a meaningful way.

If your app gets launched when the user unlocked via a local notification, you should check for the presence of the **UIApplicationLaunchOptionsLocalNotificationKey** key in your app delegate's **didFinishLaunchingWithOptions** method, like this:

```
func application(application: UIApplication,
didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?)
-> Bool {

    if let options = launchOptions {

        if let notification =
options[UIApplicationLaunchOptionsLocalNotificationKey] as?
UILocalNotification {

            if let userInfo = notification.userInfo {

                let customField1 = userInfo["CustomField1"] as!
String

                // do something neat here

            }

        }

    }


    return true

}
```

That code pulls out the **CustomField1** value that was set in the **userInfo** earlier.

It's also possible that your app receives a local notification while it's being used by the user,

in which case your app delegate will have a different method called:

```
func application(application: UIApplication,
didReceiveLocalNotification notification: UILocalNotification) {

    if let userInfo = notification.userInfo {

        let customField1 = userInfo["CustomField1"] as! String

        println("didReceiveLocalNotification: \(customField1)")

    }

}
```

For more information see Hacking with Swift tutorial 21.

How to spell out numbers using NSNumberFormatter's SpellOutStyle

Availability: iOS 2.0 or later.

iOS makes it easy to convert numbers like 10 or 100 into their written equivalents: "ten" and "one hundred", and it even handles other languages. For example, to convert the number 556 into "five hundred fifty-six", you would use this code:

```
let formatter = NSNumberFormatter()

formatter.numberStyle = NSNumberFormatterStyle.SpellOutStyle

let english = formatter.stringFromNumber(556)
```

If you wanted to get that in Spanish, you would set a locale like this:

```
formatter.locale = NSLocale(localeIdentifier: "es_ES")

let spanish = formatter.stringFromNumber(556)
```

Running that code would make the **english** constant equal to **five hundred fifty-six** and the **spanish** constant equal to **quinientos cincuenta y seis**.

How to stop the screen from going to sleep

Availability: iOS 2.0 or later.

You can stop the iOS screen sleeping by using the **idleTimerDisabled** property of your application. When set to true, this means the screen will never dim or go to sleep while your app is running, so be careful – you don't want to waste your user's battery life!

Here's an example:

```
UIApplication.sharedApplication().idleTimerDisabled = true
```


How to store NSUserDefaults options in iCloud

Availability: iOS 5.0 or later.

iOS has a built-in iCloud sync system called **NSUbiquitousKeyValueStore**, but to be honest it's pretty unpleasant to work with. Fortunately, other developers have written simple wrappers around it so that you can forget about iCloud and focus on the interesting things instead – i.e., the rest of your app.

One such example is called **MKiCloudSync** and it's **available from here**. It's open source and so easy to use you literally don't notice that it's there once you've added it to your app – it just silently syncs your **NSUserDefaults** values to and from iCloud.

To use it, **go here** and click Download Zip. Inside the zip file you'll find **MKiCloudSync.h** and **MKiCloudSync.m**, and you should drag them both into your Xcode project. Xcode will ask you if you want to create an Objective-C bridging header, and you should click "Create Bridging Header" - this is required because MKiCloudSync is written in Objective-C rather than Swift.

To actually use the library, open your new bridging header (it'll be called something like

YourProject-Bridging-Header.h) and add this:

```
#import "MKiCloudSync.h"
```

Now open your AppDelegate.swift file, find the **didFinishLaunchingWithOptions** method, and add this line to it:

```
MKiCloudSync.startWithPrefix("sync")
```

The "sync" part is important, because changes are you won't want to sync *everything* to iCloud. With that prefix, MKiCloudSync will copy to iCloud only **NSUserDefaults** keys that start with **sync** – you can now choose what you want to sync just by naming your keys appropriately.

There is one final, important thing to do: you need to enable iCloud for your app. This is done inside the Capabilities tab of your target's settings – find iCloud, then flick its switch to be On.

How to synchronize code to drawing using CADisplayLink
Availability: iOS 3.1 or later.

Lots of beginners think **NSTimer** is a great way to handle running apps or games so that update code is executed every time the screen is redrawn. Their logic is simple: update the app every 60th of a second and you're perfectly placed for smooth redraws. The problem is, they are forgetting that **NSTimer** doesn't offer precise firing and can drift earlier or later than requested updates, and also has no idea about screen redraws and so could happily fire 10ms after a screen redraw just happened – and when you're working to 16.666ms frames, 10ms is a long time!

A smarter and faster solution is the **CADisplayLink** class, which automatically calls a method you define as soon as a screen redraw happens, so you always have maximum time to execute your update code. It's extremely simple to use – here's an example to get you started:

```
let displayLink = CADisplayLink(target: self, selector: "update")

displayLink.addToRunLoop(NSRunLoop.currentRunLoop(), forMode:
NSDefaultRunLoopMode)
```

That will call a method called **update()** every 60th of a second by default. You can see it in
action with this method stub:

```
func update() {

  print("Updating!")

}
```

How to use Core Motion to read accelerometer data

Availability: iOS 4.0 or later.

Core Motion makes it ridiculously easy to read the accelerometer from iPhones and iPads,
and it even takes care of managing how the accelerometer and gyroscope work together to
report orientation. To get started import the Core Motion framework like this:

```
import CoreMotion
```

Now create a property that can store a **CMMotionManager**, like this:

```
var motionManager: CMMotionManager!
```

When you're ready to start reading accelerometer data (this will be inside **viewDidLoad()** for
most people), go ahead and create your motion manager then call its
startAccelerometerUpdates() method:

```
motionManager = CMMotionManager()

motionManager.startAccelerometerUpdates()
```

Finally, read the accelerometer data as often as you want. It's optional, though, so make sure you unwrap it carefully.

For example, if you want to change the gravity of a SpriteKit physics world so that tipping your device makes things roll around, you'd look for something like this in your **update()** method:

```
if let accelerometerData = motionManager.accelerometerData {

    physicsWorld.gravity = CGVector(dx:
accelerometerData.acceleration.y * -50, dy:
accelerometerData.acceleration.x * 50)

}
```

For more information see Hacking with Swift tutorial 26.

How to use Core Spotlight to index content in your app

Availability: iOS 9.0 or later.

One particularly popular feature in iOS 9.0 is the ability to have your app's content appear inside the iOS Spotlight search so that users can search it alongside their other device content.

First up, add these two imports to your class:

```
import CoreSpotlight

import MobileCoreServices
```

Now I'm going to give you the code to handle indexing an item, and for this we'll create a method called **indxItem()** that takes three parameters: the title of the item, a description string for the item, plus a unique identifier. What that unique identifier is depends on you project, but it should be a string. Here's the method:

```
func indexItem(title: String, desc: String, identifier: String) {

    let attributeSet = CSSearchableItemAttributeSet(itemContentType:
kUTTypeText as String)

    attributeSet.title = title

    attributeSet.contentDescription = desc


    let item = CSSearchableItem(uniqueIdentifier: "\(identifier)",
domainIdentifier: "com.hackingwithswift", attributeSet: attributeSet)


CSSearchableIndex.defaultSearchableIndex().indexSearchableItems([item
]) { (error: NSError?) -> Void in

        if let error = error {

            print("Indexing error: \(error.localizedDescription)")

        } else {

            print("Search item successfully indexed!")

        }

    }

}
```

That wraps the title and description up inside a **CSSearchableItemAttributeSet**, which in turn goes inside a **CSSearchableItem**, and from there to Spotlight to index. If you have several items to index you can have them processed all at once and it works faster.

Note that you should change **domainIdentifier** to your own domain, e.g. **com.yoursite**.

Now that your item is indexed, it will be available in Spotlight searches immediately. If a user finds one of your index items and taps it, your app will get launched and you should be able

to pull out the unique identifier of the item that was tapped – this tells you what item was tapped so that you can take appropriate action.

Put this code inside your app delegate:

```swift
func application(application: UIApplication, continueUserActivity
userActivity: NSUserActivity, restorationHandler: ([AnyObject]?) ->
Void) -> Bool {

    if userActivity.activityType == CSSearchableItemActionType {

        if let uniqueIdentifier = userActivity.userInfo?
[CSSearchableItemActivityIdentifier] as? String {

            doSomethingCoolWith(uniqueIdentifier)

        }

    }


    return true

}
```

That's it!

For the sake of completeness, here's how you remove an item from the Spotlight index:

```swift
func deindexItem(identifier: String) {

CSSearchableIndex.defaultSearchableIndex().deleteSearchableItemsWithI
dentifiers(["\(identifier)"]) { (error: NSError?) -> Void in

        if let error = error {

            print("Deindexing error: \(error.localizedDescription)")

        } else {

            print("Search item successfully removed!")

        }

    }
```

```
}
```

For more information see Hacking with Swift tutorial 32.

How to use Touch ID to authenticate users by fingerprint
Availability: iOS 8.0 or later.

Touch ID is an easy and secure way for users to authenticate themselves, so its no surprise that it's caught on so quickly among apps. Authenticating with Touch ID automatically uses the fingerprints registered by the user when they set up Touch ID, and you never have access to those fingerprints, which means it's both low-friction and extra-secure.

To get started, you need to import the LocalAuthentication framework like this:

```
import LocalAuthentication
```

The actual act of authenticating users has a number of possible results, and you need to catch them all:

 • The user might not have a Touch ID-capable device.
 • The user might have a Touch ID-capable device, but might not have configured it.
 • The user might hit cancel when asked to authenticate.
 • The user might ask to enter a passcode rather than use Touch ID.

The first three are easily handled just by showing a message with useful information, but the last one is a quirk of Touch ID: Apple insists that your app provide a passcode method of authentication as a back up. More annoyingly, you need to request and store this passcode yourself – it's not even done by Apple using the system unlock code!

Asking for and setting a passcode is easy enough, so I'll leave that to you. The important bit is asking for Touch ID authentication, which is done using this code:

```
func authenticateUser() {
```

```
    var context = LAContext()

    var error: NSError?


    if
context.canEvaluatePolicy(.DeviceOwnerAuthenticationWithBiometrics,
error: &error) {

        var reason = "Identify yourself!"



context.evaluatePolicy(.DeviceOwnerAuthenticationWithBiometrics,
localizedReason: reason) {

            [unowned self] (success: Bool, authenticationError:
NSError?) -> Void in


            dispatch_async(dispatch_get_main_queue()) {
                if success {
                    self.runSecretCode()
                } else {
                    if let error = authenticationError {
                if error.code == LAError.UserFallback.rawValue {
                        let ac = UIAlertController(title: "Passcode? Ha!",
message: "It's Touch ID or nothing – sorry!", preferredStyle: .Alert)
                        ac.addAction(UIAlertAction(title: "OK",
style: .Default, handler: nil))
                        self.presentViewController(ac, animated: true,
completion: nil)

                        return
                    }
                }


            let ac = UIAlertController(title: "Authentication failed",
message: "Your fingerprint could not be verified; please try again.",
preferredStyle: .Alert)

            ac.addAction(UIAlertAction(title: "OK", style: .Default,
```

```
handler: nil))

            self.presentViewController(ac, animated: true, completion:
nil)

                }

            }

        }

    } else {

        let ac = UIAlertController(title: "Touch ID not available",
message: "Your device is not configured for Touch ID.",
preferredStyle: .Alert)

    ac.addAction(UIAlertAction(title: "OK", style: .Default, handler:
nil))

    self.presentViewController(ac, animated: true, completion: nil)

    }

}
```

For more information see Hacking with Swift tutorial 28.

NSTextEffectLetterpressStyle: How to add a letterpress effect to text

Availability: iOS 7.0 or later.

You can add a subtle embossing effect to any text in your app using **NSAttributedString** and **NSTextEffectLetterpressStyle**. As an example, this code creates an attributed string using 24-point Georga Bold in red, with Apple's letterpress effect applied, then writes it into a label:

```
let attrs = [NSForegroundColorAttributeName: UIColor.redColor(),

  NSFontAttributeName: UIFont(name: "Georgia-Bold", size: 24)!,

  NSTextEffectAttributeName: NSTextEffectLetterpressStyle]


let string = NSAttributedString(string: "Hello, world!", attributes:
attrs)
```

```
yourLabel.attributedText = string
```

UIColor

How to convert a HTML name string into a UIColor
Availability: iOS 2.0 or later.

HTML color names let you use familiar titles like "steel blue" and "mint cream" rather than hex values, but sadly these standardised names aren't available in iOS – or at least not by default. Fortunately, it's easy to add an extension to **UIColor** that maps these names to hexadecimal color values, then add another extension to convert hex colors to **UIColors**. Here's the code:

```
extension UIColor {
  public convenience init?(hexString: String) {
    let r, g, b, a: CGFloat

    if hexString.hasPrefix("#") {
      let start = hexString.startIndex.advancedBy(1)
      let hexColor = hexString.substringFromIndex(start)

      if hexColor.characters.count == 8 {
        let scanner = NSScanner(string: hexColor)
        var hexNumber: UInt64 = 0

        if scanner.scanHexLongLong(&hexNumber) {
          r = CGFloat((hexNumber & 0xff000000) >> 24) / 255
```

```swift
            g = CGFloat((hexNumber & 0x00ff0000) >> 16) / 255

            b = CGFloat((hexNumber & 0x0000ff00) >> 8) / 255

            a = CGFloat(hexNumber & 0x000000ff) / 255


            self.init(red: r, green: g, blue: b, alpha: a)

            return

        }

    }

  }


  return nil

}


public convenience init?(name: String) {
  let allColors = [
    "aliceblue": "#F0F8FFFF",

    "antiquewhite": "#FAEBD7FF",

    "aqua": "#00FFFFFF",

    "aquamarine": "#7FFFD4FF",

    "azure": "#F0FFFFFF",

    "beige": "#F5F5DCFF",

    "bisque": "#FFE4C4FF",

    "black": "#000000FF",

    "blanchedalmond": "#FFEBCDFF",

    "blue": "#0000FFFF",

    "blueviolet": "#8A2BE2FF",

    "brown": "#A52A2AFF",

    "burlywood": "#DEB887FF",

    "cadetblue": "#5F9EA0FF",

    "chartreuse": "#7FFF00FF",
```

```
"chocolate": "#D2691EFF",

"coral": "#FF7F50FF",

"cornflowerblue": "#6495EDFF",

"cornsilk": "#FFF8DCFF",

"crimson": "#DC143CFF",

"cyan": "#00FFFFFF",

"darkblue": "#00008BFF",

"darkcyan": "#008B8BFF",

"darkgoldenrod": "#B8860BFF",

"darkgray": "#A9A9A9FF",

"darkgrey": "#A9A9A9FF",

"darkgreen": "#006400FF",

"darkkhaki": "#BDB76BFF",

"darkmagenta": "#8B008BFF",

"darkolivegreen": "#556B2FFF",

"darkorange": "#FF8C00FF",

"darkorchid": "#9932CCFF",

"darkred": "#8B0000FF",

"darksalmon": "#E9967AFF",

"darkseagreen": "#8FBC8FFF",

"darkslateblue": "#483D8BFF",

"darkslategray": "#2F4F4FFF",

"darkslategrey": "#2F4F4FFF",

"darkturquoise": "#00CED1FF",

"darkviolet": "#9400D3FF",

"deeppink": "#FF1493FF",

"deepskyblue": "#00BFFFFF",

"dimgray": "#696969FF",

"dimgrey": "#696969FF",

"dodgerblue": "#1E90FFFF",
```

```
"firebrick": "#B22222FF",

"floralwhite": "#FFFAF0FF",

"forestgreen": "#228B22FF",

"fuchsia": "#FF00FFFF",

"gainsboro": "#DCDCDCFF",

"ghostwhite": "#F8F8FFFF",

"gold": "#FFD700FF",

"goldenrod": "#DAA520FF",

"gray": "#808080FF",

"grey": "#808080FF",

"green": "#008000FF",

"greenyellow": "#ADFF2FFF",

"honeydew": "#F0FFF0FF",

"hotpink": "#FF69B4FF",

"indianred": "#CD5C5CFF",

"indigo": "#4B0082FF",

"ivory": "#FFFFF0FF",

"khaki": "#F0E68CFF",

"lavender": "#E6E6FAFF",

"lavenderblush": "#FFF0F5FF",

"lawngreen": "#7CFC00FF",

"lemonchiffon": "#FFFACDFF",

"lightblue": "#ADD8E6FF",

"lightcoral": "#F08080FF",

"lightcyan": "#E0FFFFFF",

"lightgoldenrodyellow": "#FAFAD2FF",

"lightgray": "#D3D3D3FF",

"lightgrey": "#D3D3D3FF",

"lightgreen": "#90EE90FF",

"lightpink": "#FFB6C1FF",
```

```
"lightsalmon": "#FFA07AFF",

"lightseagreen": "#20B2AAFF",

"lightskyblue": "#87CEFAFF",

"lightslategray": "#778899FF",

"lightslategrey": "#778899FF",

"lightsteelblue": "#B0C4DEFF",

"lightyellow": "#FFFFE0FF",

"lime": "#00FF00FF",

"limegreen": "#32CD32FF",

"linen": "#FAF0E6FF",

"magenta": "#FF00FFFF",

"maroon": "#800000FF",

"mediumaquamarine": "#66CDAAFF",

"mediumblue": "#0000CDFF",

"mediumorchid": "#BA55D3FF",

"mediumpurple": "#9370D8FF",

"mediumseagreen": "#3CB371FF",

"mediumslateblue": "#7B68EEFF",

"mediumspringgreen": "#00FA9AFF",

"mediumturquoise": "#48D1CCFF",

"mediumvioletred": "#C71585FF",

"midnightblue": "#191970FF",

"mintcream": "#F5FFFAFF",

"mistyrose": "#FFE4E1FF",

"moccasin": "#FFE4B5FF",

"navajowhite": "#FFDEADFF",

"navy": "#000080FF",

"oldlace": "#FDF5E6FF",

"olive": "#808000FF",

"olivedrab": "#6B8E23FF",
```

```
"orange": "#FFA500FF",

"orangered": "#FF4500FF",

"orchid": "#DA70D6FF",

"palegoldenrod": "#EEE8AAFF",

"palegreen": "#98FB98FF",

"paleturquoise": "#AFEEEEFF",

"palevioletred": "#D87093FF",

"papayawhip": "#FFEFD5FF",

"peachpuff": "#FFDAB9FF",

"peru": "#CD853FFF",

"pink": "#FFC0CBFF",

"plum": "#DDA0DDFF",

"powderblue": "#B0E0E6FF",

"purple": "#800080FF",

"rebeccapurple": "#663399FF",

"red": "#FF0000FF",

"rosybrown": "#BC8F8FFF",

"royalblue": "#4169E1FF",

"saddlebrown": "#8B4513FF",

"salmon": "#FA8072FF",

"sandybrown": "#F4A460FF",

"seagreen": "#2E8B57FF",

"seashell": "#FFF5EEFF",

"sienna": "#A0522DFF",

"silver": "#C0C0C0FF",

"skyblue": "#87CEEBFF",

"slateblue": "#6A5ACDFF",

"slategray": "#708090FF",

"slategrey": "#708090FF",

"snow": "#FFFAFAFF",
```

```
    "springgreen": "#00FF7FFF",

    "steelblue": "#4682B4FF",

    "tan": "#D2B48CFF",

    "teal": "#008080FF",

    "thistle": "#D8BFD8FF",

    "tomato": "#FF6347FF",

    "turquoise": "#40E0D0FF",

    "violet": "#EE82EEFF",

    "wheat": "#F5DEB3FF",

    "white": "#FFFFFFFF",

    "whitesmoke": "#F5F5F5FF",

    "yellow": "#FFFF00FF",

    "yellowgreen": "#9ACD32FF"

    ]


    let cleanedName = name.stringByReplacingOccurrencesOfString(" ",
withString: "").lowercaseString


    if let hexString = allColors[cleanedName] {

      self.init(hexString: hexString)

    } else {

      return nil

    }

  }

}
```

With that done, here's how you create a color:

```
let steelBlue = UIColor(name: "steel blue")
```

How to convert a hex color to a UIColor

Availability: iOS 2.0 or later.

Here's a simple extension to **UIColor** that lets you create colors from hex strings. The new method is a failable initializer, which means it returns nil if you don't specify a color in the correct format. It should be a # symbol, followed by red, green, blue and alpha in hex format, for a total of nine characters. For example, #ffe700ff is gold.

Here's the code:

```
extension UIColor {
  public convenience init?(hexString: String) {
    let r, g, b, a: CGFloat


    if hexString.hasPrefix("#") {
      let start = hexString.startIndex.advancedBy(1)
      let hexColor = hexString.substringFromIndex(start)


      if hexColor.characters.count == 8 {
        let scanner = NSScanner(string: hexColor)
        var hexNumber: UInt64 = 0


        if scanner.scanHexLongLong(&hexNumber) {
          r = CGFloat((hexNumber & 0xff000000) >> 24) / 255
          g = CGFloat((hexNumber & 0x00ff0000) >> 16) / 255
          b = CGFloat((hexNumber & 0x0000ff00) >> 8) / 255
          a = CGFloat(hexNumber & 0x000000ff) / 255


          self.init(red: r, green: g, blue: b, alpha: a)
          return
```

```
            }

        }

    }


    return nil

  }

}
```

If you wanted it always to return a value, change **init?** to be **init** then change the **return nil** line at the end to be **return UIColor.blackColor()** or whatever you'd like the default value to be.

To use the extension, write code like this:

```
let gold = UIColor(hexString: "#ffe700ff")
```

How to create custom colors using UIColor RGB and hues

Availability: iOS 2.0 or later.

Although there are quite a few built-in UIColors, you'll want to create your own very frequently. This can be done in a number of ways, but the most common is specifying individual vaues for red, green, blue and alpha, like this:

```
let col1 = UIColor(red: 1, green: 0, blue: 0, alpha: 1)
```

Each of those numbers need to be between 0 and 1.

An alternative way is to specify color values as hue, saturation and brightness, or HSB. Hue is a value between 0 and 1 on a color wheel, where 0 and 1 are both red. Saturation is how deep the color should be (so 0 is just gray) and brightness is how light the shade should be.

Here's how it's done:

```
let col2 = UIColor(hue: 0, saturation: 0.66, brightness: 0.66, alpha:
1)

let col3 = UIColor(hue: 0.25, saturation: 0.66, brightness: 0.66,
alpha: 1)

let col4 = UIColor(hue: 0.5, saturation: 0.66, brightness: 0.66,
alpha: 1)

let col5 = UIColor(hue: 0.75, saturation: 0.66, brightness: 0.66,
alpha: 1)
```

The advantage to using HSB rather than RGB is that you can generate very similar colors by keeping the saturation and brightness constant and changing only the hue – the code above generates some nice pastel shades of red, green, cyan and magenta, for example.

UIKit

Changing which UITabBarController tabs can be edited
Availability: iOS 2.0 or later.

If you have a More tab in your tab bar controller, this will automatically get an Edit button so that users can drag tabs around to customize the user interface. This doesn't actually save the tab ordering for you, which means the tabs will revert on next run unless you persist the user's choices yourself, but it does do everything else for you.

By default, users can move any and all tabs, but if you want to force some tabs to be in place you should set the **customizableViewControllers** property of your tab bar controller. This should be an array of the view controllers you want to give your users access to edit, or an empty array if you want the Edit button to go away entirely.

If your tab bar controller is your window's root view controller (for example, if you started with

the Xcode tab bar template project), you can allow users to customize the first three view controllers like this:

```
func application(application: UIApplication,
didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?)
-> Bool {

  if let tabBarController = window?.rootViewController as?
UITabBarController {

    let slice = tabBarController.viewControllers![0...2]

    let array = Array(slice)


    tabBarController.customizableViewControllers = array

  }


  return true

}
```

Place that into your AppDelegate.swift file in place of the existing **didFinishLaunchingWithOptions** method, and you're done.

How set different widths for a UISegmentedControl's elements

Availability: iOS 5.0 or later.

Segmented controls give each segment equal width by default, which is aesthetically pleasing when you have space to spare but technically irritating when space is tight. Rather than try to squash too much into a small space, you have two options: set custom segment widths, or ask iOS to size them individually for you.

The first option looks like this:

```
segmentedControl.setWidth(100, forSegmentAtIndex: 0)
```

```
segmentedControl.setWidth(50, forSegmentAtIndex: 1)
```

That gives you individually sized segments while sticking to a value you define, which means you get to tweak the aesthetics as you want. The second option looks like this:

```
segmentedControl.apportionsSegmentWidthsByContent = true
```

That hands full control over to iOS, which is probably the best thing to do most of the time.

How to add Retina and Retina HD graphics to your project
Availability: iOS 8.0 or later.

iOS has a simple, beautiful solution for handling Retina and Retina HD graphics, and in fact it does almost all the work for you – all you have to do is name your assets correctly.

Imagine you have an image called taylor.png, which is 100x100 pixels in size. That will look great on non-Retina devices, which means iPad 2 and the first-generation iPad Mini. If you want it to look great on Retina devices (which means iPad 3, 4, Air, Air 2, Mini 2 and Mini 3, plus iPhone 4s, 5, 5s, and 6) you need to provide a second image called taylor@2x.png that is 200x200 pixels in size – i.e., exactly twice the width and height.

Retina HD devices – at the time of writing that's just the iPhone 6 Plus – have an even higher resolution, so if you want your image to look great there you should provide a third image called taylor@3x.png that is 300x300 pixels in size – i.e., exactly three times the width and height of the original.

If you're not using an asset catalog, you can just drag these images into your project to have iOS use them. If you are using an asset catalog, drag them into your asset catalog and you should see Xcode correctly assign them to 1x, 2x and 3x boxes for the image. **It's critical you name the files correctly** because that's what iOS uses to load the correct resolution.

With that done, you just need to load taylor.png in your app, and iOS will automatically load the correct version of it depending on the user's device.

How to add a UITextField to a UIAlertController

Availability: iOS 8.0 or later.

The **UIAlertController** class from iOS 8.0 lets you add as many text fields as you need, and you can read the value of those text fields when the user taps a button.

The example below creates an alert controller with one button and a text field. When the button is tapped, the text of the text field is pulled out, at which point it's down to you to do something interesting with it:

```
func promptForAnswer() {
    let ac = UIAlertController(title: "Enter answer", message: nil,
preferredStyle: .Alert)

    ac.addTextFieldWithConfigurationHandler(nil)


    let submitAction = UIAlertAction(title: "Submit",
style: .Default) { [unowned self, ac] (action: UIAlertAction!) in
        let answer = ac.textFields![0] as! UITextField

        // do something interesting with "answer" here
    }


    ac.addAction(submitAction)


    presentViewController(ac, animated: true, completion: nil)
}
```

For more information see Hacking with Swift tutorial 5.

How to add a bar button to a navigation bar

Availability: iOS 2.0 or later.

Navigation bars are one of the most common user interface components in iOS, so being able to add buttons to them is something you'll do *a lot*. You can add buttons to the left and right side of a navigation bar, and as of iOS 5.0 you can add more than one to either side.

Note: bar button items don't belong to the **UINavigationBar** directly. Instead, they belong to a **UINavigationItem** that is currently active on the navigation bar, which in turn is usually owned by the view controller that is currently active on the screen. So, to create bar button items for your view controller, you would do this:

```
navigationItem.leftBarButtonItem =
UIBarButtonItem(barButtonSystemItem: .Add, target: self, action:
"addTapped")

navigationItem.rightBarButtonItem = UIBarButtonItem(title: "Add",
style: .Plain, target: self, action: "addTapped")
```

That will call the **addTapped()** method on the current view controller when either button is tapped. Note that the first one uses a standard system icon (recommended when it's available!) and the second one uses text.

Like I said, as of iOS 5.0 you can attach more than one bar button item to either side of the navigation bar, like this:

```
let add = UIBarButtonItem(barButtonSystemItem: .Add, target: self,
action: "addTapped")

let play = UIBarButtonItem(title: "Play", style: .Plain, target:
self, action: "addTapped")


navigationItem.rightBarButtonItems = [add, play]
```

How to add a button to a UITableViewCell
Availability: iOS 2.0 or later.

There are two steps to add a working button to a table view cell. The first step is to add a button like this:

```
cell.accessoryType = .DetailDisclosureButton
```

The second step is to take action when the button is tapped by creating the **accessoryButtonTappedForRowWithIndexPath** method:

```
override func tableView(tableView: UITableView,
accessoryButtonTappedForRowWithIndexPath indexPath: NSIndexPath) {

    doSomethingWithItem(indexPath.row)

}
```

That's it!

How to add a custom view to a UIBarButtonItem

Availability: iOS 2.0 or later.

Most **UIBarButtonItems** contain either an icon or some text, but they can do so much more – in fact, you can embed any kind of **UIView** subclass inside a bar button item, then put that button into a navigation bar or toolbar as you normally would.

For example, you can create a **UIProgressView** and place it into a bar button like this:

```
var progressView = UIProgressView(progressViewStyle: .Default)

progressView.sizeToFit()


let progressButton = UIBarButtonItem(customView: progressView)
```

For more information see Hacking with Swift tutorial 4.

How to add a flexible space to a UIBarButtonItem

Availability: iOS 2.0 or later.

There's a special kind of **UIBarButtonItem** called **FlexibleSpace**, and this acts like a spring between other buttons, pushing them to one side. A flexible space will always expand to take up as much room as possible, splitting space evenly between other flexible spaces if they exist.

For example, if you add this button to a toolbar, it will sit on the left edge of the toolbar:

```
let refresh = UIBarButtonItem(barButtonSystemItem: .Refresh, target:
self, action: "refreshTapped")
```

If you create and add a flexible space first, then that button will be pushed to the right edge as the flexible space expands to take up most of the toolbar. Here's how you create the flexible space:

```
let spacer = UIBarButtonItem(barButtonSystemItem: .FlexibleSpace,
target: nil, action: nil)
```

For more information see Hacking with Swift tutorial 4.

How to add a section header to a table view

Availability: iOS 2.0 or later.

You can use the built-in iOS table section headers by returning a value from **titleForHeaderInSection** like this:

```
override func tableView(tableView: UITableView,
titleForHeaderInSection section: Int) -> String? {

  return "Section \(section)"

}
```

If you want to return a custom header view with something more than just some text, you should use **viewForHeaderInSection** instead, like this:

```
override func tableView(tableView: UITableView,
viewForHeaderInSection section: Int) -> UIView? {

  let vw = UIView()

  vw.backgroundColor = UIColor.blackColor()


  return vw

}
```

How to add a shadow to a UIView

Availability: iOS 3.2 or later.

iOS can dynamically generate shadows for any **UIView**, and these shadows automatically adjust to fit the shape of the item in question – even following the curves of text inside a **UILabel**. This functionality is built right in, so all you need to do is configure its properties, and there are four you need to care about:

 • **shadowColor** sets the color of the shadow, and needs to be a **CGColor**.
 • **shadowOpacity** sets how transparent the shadow is, where 0 is invisible and 1 is as strong as possible.
 • **shadowOffset** sets how far away from the view the shadow should be, to give a 3D offset effect.
 • **shadowRadius** sets how wide the shadow should be.

Here's a simple example to get you started:

```
yourView.layer.shadowColor = UIColor.blackColor().CGColor

yourView.layer.shadowOpacity = 1

yourView.layer.shadowOffset = CGSizeZero

yourView.layer.shadowRadius = 10
```

Be warned: generating shadows dynamically is expensive, because iOS has to draw the shadow around the exact shape of your view's contents. If you can, set the **shadowPath** property to a specific value so that iOS doesn't need to calculate transparency dynamically. For example, this creates a shadow path equivalent to the frame of the view:

```
yourView.layer.shadowPath = UIBezierPath(rect:
yourView.bounds).CGPath
```

Alternatively, ask iOS to cache the rendered shadow so that it doesn't need to be redrawn:

```
yourView.layer.shouldRasterize = true
```

How to add blur and vibrancy using UIVisualEffectView
Availability: iOS 8.0 or later.

As of iOS 8.0, visual effects such as blur and vibrancy are a cinch because Apple provides a built in **UIView** subclass that does all the hard work: **UIVisualEffectView**. For example, if you want to blur an image, you would use this code:

```
let imageView = UIImageView(image: UIImage(named: "example"))

imageView.frame = view.bounds

imageView.contentMode = .ScaleToFill
```

```
view.addSubview(imageView)


let blurEffect = UIBlurEffect(style: .Dark)

let blurredEffectView = UIVisualEffectView(effect: blurEffect)

blurredEffectView.frame = imageView.bounds

view.addSubview(blurredEffectView)
```

As well as blurring content, Apple also lets you add a "vibrancy" effect to your views – this is a translucency effect designed to ensure that text is readable when it's over any kind of blurred background, and it's used to create that soft glow effect you see in the notification center.

We could extend the previous example so that it adds a segmented control in the middle of the view, using a vibrancy effect. This is accomplished by created a second **UIVisualEffectView** inside the first one, this time using **UIVibrancyEffect** to create the glow. Note that you need to use the same blur type for both your visual effect views, otherwise the glow effect will be incorrect.

```
let segmentedControl = UISegmentedControl(items: ["First Item",
"Second Item"])

segmentedControl.sizeToFit()

segmentedControl.center = view.center


let vibrancyEffect = UIVibrancyEffect(forBlurEffect: blurEffect)

let vibrancyEffectView = UIVisualEffectView(effect: vibrancyEffect)

vibrancyEffectView.frame = imageView.bounds


vibrancyEffectView.contentView.addSubview(segmentedControl)

blurredEffectView.contentView.addSubview(vibrancyEffectView)
```

Warning: you need to add child views to the **contentView** property of a **UIVisualEffectView** otherwise they will not be drawn correctly.

How to add iAd to a view controller

Availability: iOS 4.0 or later.

There are three ways to add iAd to an app. Helpfully, their use cases are pretty clear because the easiest option give you no control, the hardest option gives you lots of control, and the middle option gives you as much control as you want.

First, the easiest option – put this into your view controller:

```
canDisplayBannerAds = true
```

You will need to import the iAd framework to make that compile correctly. Yes, that's all it takes to get iAd banners showing in your view. If you don't mind not being able to have control over when adverts show or where they should go, then you're basically done.

The second option is to place an iAd inside Interface Builder. This gives you more control because now you can place it where you want it (within reason – Apple does have rules!), you can attach Auto Layout constraints, you can configure its delegate, and so on.

While that second option is good, you really need to at least know the third option in order to make use of it: making the iAd entirely in code. To make this work you need to create a property to store the banner view, like this:

```
var bannerView: ADBannerView!
```

Creating a banner view isn't complicated, but you should take care to give it good Auto Layout constraints. This is particularly important when it comes to taking action if there is no banner to show – is hiding the banner view enough, or do you need to change other constraints too?

In the most simple case you can configure an iAd banner in code like this:

```
bannerView = ADBannerView(adType: .Banner)

bannerView.setTranslatesAutoresizingMaskIntoConstraints(false)

bannerView.delegate = self

bannerView.hidden = true

view.addSubview(bannerView)


let viewsDictionary = ["bannerView": bannerView]

view.addConstraints(NSLayoutConstraint.constraintsWithVisualFormat("H
:|[bannerView]|", options: .allZeros, metrics: nil, views:
viewsDictionary))

view.addConstraints(NSLayoutConstraint.constraintsWithVisualFormat("V
:[bannerView]|", options: .allZeros, metrics: nil, views:
viewsDictionary))
```

As soon as you set your view controller to be the delegate of the iAd banner, you need to tell Swift you conform to the **ADBannerViewDelegate** protocol:

```
class ViewController: UIViewController, ADBannerViewDelegate {
```

Now that you're the delegate of the banner view, you should use **bannerViewDidLoadAd()** as your signal to show the banner view, and **bannerView(_:didFailToReceiveAdWithError:)** as your signal to hide it, like this:

```
func bannerViewDidLoadAd(banner: ADBannerView!) {

    bannerView.hidden = false

}


func bannerView(banner: ADBannerView!, didFailToReceiveAdWithError
error: NSError!) {

    bannerView.hidden = true
```

```
}
```

For more information see Hacking with Swift tutorial 18.

How to adjust a UIScrollView to fit the keyboard
Availability: iOS 2.0 or later.

If your user interface brings up the keyboard, you should respond by adjusting your layout so that all parts are still visible. If you're using a **UIScrollView** or any classes that have a scroll view as part of their layout (table views and text views, for example), this means adjusting the **contentInset** property to account for the keyboard.

First you need to register for keyboard change notifications. Put this into your **viewDidLoad()** method:

```
let notificationCenter = NSNotificationCenter.defaultCenter()

notificationCenter.addObserver(self, selector: "adjustForKeyboard:",
name: UIKeyboardWillHideNotification, object: nil)

notificationCenter.addObserver(self, selector: "adjustForKeyboard:",
name: UIKeyboardWillChangeFrameNotification, object: nil)
```

Now add this method somewhere else in your class:

```
func adjustForKeyboard(notification: NSNotification) {
    let userInfo = notification.userInfo!


    let keyboardScreenEndFrame =
(userInfo[UIKeyboardFrameEndUserInfoKey] as! NSValue).CGRectValue()
    let keyboardViewEndFrame =
view.convertRect(keyboardScreenEndFrame, fromView: view.window)
```

```
    if notification.name == UIKeyboardWillHideNotification {

        yourTextView.contentInset = UIEdgeInsetsZero

    } else {

        yourTextView.contentInset = UIEdgeInsets(top: 0, left: 0,
bottom: keyboardViewEndFrame.height, right: 0)

    }


    yourTextView.scrollIndicatorInsets = yourTextView.contentInset


    let selectedRange = yourTextView.selectedRange

    yourTextView.scrollRangeToVisible(selectedRange)

}
```

That example code is for adjusting text views. If you want it to apply to a regular scroll view, just take out the last two lines - they are in there so that the text view readjusts itself so the user doesn't lose their place while editing.

For more information see Hacking with Swift tutorial 16.

How to adjust image content mode using aspect fill, aspect fit and scaling

Availability: iOS 2.0 or later.

All views (including those that don't hold images) have a content mode that affects the way they draw their content. The default is **Scale To Fill** because it's fastest: the contents of the view just get stretched up (or down) to fit the space available. But there are two others that you'll be using a lot: Aspect Fit and Aspect Fill.

"Aspect Fit" means "stretch this image up as large as it can go, but make sure that all the image is visible while keeping its original aspect ratio." This is useful when you want an image to be as large as possible without stretching its proportions, and it's probably the most commonly used content mode.

"Aspect Fit" means "stretch this image up as large as it can go, cropping off any parts that don't fit while keeping its original aspect ratio." This is useful when you want an image to fill its image view, even when that means losing either the horizontal or vertical edges. If you want to force an image to fill a specific space, but you want to keep its aspect ratio, this is the one you should use.

For more information see Hacking with Swift tutorial 1.

How to animate views using animateWithDuration()

Availability: iOS 4.0 or later.

Animation in iOS is done by starting an animation block, then telling iOS what changes you want to make. Because the animation block is active, those changes won't happen straight away – instead, iOS will execute them smoothly over the time you specified, so you don't have to worry when it will finish or what all the intermediate states are.

Here's a basic example to make a view fade out:

```
UIView.animateWithDuration(1) {

  viewToAnimate.alpha = 0

}
```

If you want to remove the view from its superview once the fade has finished, you can use a more advanced version of the same method that gives you a completion block – a closure that will be run once the animation finishes. Here's how that looks:

```
UIView.animateWithDuration(1, animations: {

  viewToAnimate.alpha = 0

}) { _ in

  viewToAnimate.removeFromSuperview()

}
```

You can also specify a delay before the animation starts, and even control the acceleration and deceleration curves of the animation, like this:

```
UIView.animateWithDuration(1, delay: 1, options:
UIViewAnimationOptions.CurveEaseIn, animations: {

  viewToAnimate.alpha = 0

}) { _ in

  viewToAnimate.removeFromSuperview()

}
```

How to animate views with spring damping using animateWithDuration()

Availability: iOS 7.0 or later.

Spring animations work by changing from a start state to an end state, with a slight overshoot and bounce at the end. For example, if you want to animate a view moving from X 0 to X 100, it might move to X 120 before bouncing back to X 80, then X 110 and finally X 100, as if the animation were attached to a spring.

Spring animations are built into iOS as of iOS 7.0 and require two values: how "springy" the spring should be, and how fast it should start. The first value is specified with **usingSpringWithDamping**, where higher values make the bouncing finish faster. The second value is specified with **initialSpringVelocity**, where higher values give the spring more initial momentum.

Here's the code to make a view fade out, then fade it the tiniest bit, then fade out again – all done using a spring animation:

```
UIView.animateWithDuration(1, delay: 1, usingSpringWithDamping: 0.5,
initialSpringVelocity: 5, options: .CurveEaseInOut, animations: {

  viewToAnimate.alpha = 0

}) { _ in
```

```
    viewToAnimate.removeFromSuperview()

}
```

How to animate when your size class changes: willTransitionToTraitCollection()

Availability: iOS 8.0 or later.

A size class change is usually triggered by your user rotating their device, but it can also happen for example when using the new iOS 9.0 multitasking to adjust window splits. Your UI needs to look great in all size classes it supports, which means you either create multiple variations of your layouts inside Interface Builder (this is the preferred route) or you make changes in code.

More often than not, I find myself mixing approaches: I do the vast majority of work inside IB, then make minor changes by hand inside the **willTransitionToTraitCollection()** method. When this is called, you'll be given a **UIViewControllerTransitionCoordinator** object (yes, that's an extremely long name!) to work with, which allows you to animate your changes as needed.

To give you a very visible demonstration of how this works, I've written some example code below that adjusts the background color of the current view. **You should run this using the iOS simulator using an iPhone.** The reason that this requires the iPhone simulator rather than the iPad simulator is that iPads have the same size classes in portrait and landscape, which makes the changes harder to spot.

Anyway, put this code into a view controller, then try it on an iPhone. When you rotate the simulator, the screen will change between red and blue, or green and blue, depending on the rotation. The important thing is that the change is animated because it's placed inside a call to **animateAlongsideTransition()**, which automatically makes your animation match the rotation animation.

Using this method requires two closures: the first is where you make the changes you want to animate, and the second is code to be run when the animation completes. So, when the new vertical size class is compact, the screen will animate from blue to red, then jump back to blue. I realise this isn't directly useful in your own apps, but that's because you'll want to

make your own changes – just take the code below and replace the background color changes with your own logic.

```
override func willTransitionToTraitCollection(newCollection:
UITraitCollection, withTransitionCoordinator coordinator:
UIViewControllerTransitionCoordinator) {

  coordinator.animateAlongsideTransition({ [unowned self] _ in

    if newCollection.verticalSizeClass == .Compact {

      self.view.backgroundColor = UIColor.redColor()

    } else {

      self.view.backgroundColor = UIColor.greenColor()

    }

  }) { [unowned self] _ in

    self.view.backgroundColor = UIColor.blueColor()

  }
}
```

How to bring a subview to the front of a UIView

Availability: iOS 2.0 or later.

UIKit draws views back to front, which means that views higher up the stack are drawn on top of those lower down. If you want to bring a subview to the front, there's a method just for you: **bringSubviewToFront()**. Here's an example:

```
parentView.bringSubviewToFront(childView)
```

This method can also be used to bring any subview to the front, even if you're not sure where it is:

```
childView.superview.bringSubviewToFront(childView)
```

How to change the scroll indicator inset for a UIScrollView

Availability: iOS 2.0 or later.

It's common to adjust content insets of a scroll view or any class that embeds one (table view, text view, etc) so that you control the scrolling mechanism precisely, but whenever you change the content inset it's a good idea also to change the scroll indicator inset: the visual indicator bar on the right that shows users how far they have left to scroll.

Changing this value adds a tiny bit of UI polish and it's easy to do:

```
scrollView.scrollIndicatorInsets = UIEdgeInsets(top: 30, left: 0, bottom: 0, right: 10)
```

How to check a string is spelled correctly using UITextChecker

Availability: iOS 3.2 or later.

You can draw on the iOS dictionary in just a few lines of code thanks to the **UITextChecker** class. Tell it the range of the string you want to check (this could be the whole string or just part of it), then ask it to tell you where the spelling error is. If it says there are no errors, the word is good. Here's the code:

```
func wordIsReal(word: String) -> Bool {

  let checker = UITextChecker()

  let range = NSMakeRange(0, word.characters.count)

  let misspelledRange = checker.rangeOfMisspelledWordInString(word,
range: range, startingAt: 0, wrap: false, language: "en")


  return misspelledRange.location == NSNotFound
```

```
}
```

Note that **rangeOfMisspelledWordInString()** accepts a language parameter, so you can change that as needed.

For more information see Hacking with Swift tutorial 5.

How to convert a CGPoint in one UIView to another view using convertPoint()

Availability: iOS 2.0 or later.

Each view has its own co-ordinate system, meaning that if I tap a button and ask iOS where I tapped, it will tell me where I tapped *relative to the top-left of the button*. This is usually what you want, but if you want to translate a position in one view into a position it's easy enough to do.

As an example, this code creates two views, creates a virtual "tap", then converts it from the first view's co-ordinate space to the second's:

```
let view1 = UIView(frame: CGRect(x: 50, y: 50, width: 128, height: 128))

let view2 = UIView(frame: CGRect(x: 200, y: 200, width: 128, height: 128))


let tap = CGPoint(x: 10, y: 10)
let convertedTap = view1.convertPoint(tap, toView: view2)
```

That will set **convertedTap** to X -140.0, Y -140.0.

How to create Auto Layout constraints in code: constraintsWithVisualFormat()

Availability: iOS 6.0 or later.

While the complexities of Auto Layout make it something most people prefer to grapple with using Interface Builder, it does have some cool tricks up its sleeve if you prefer to work in code. One of those tricks is called Visual Format Language (VFL) and lets you use ASCII art to tell iOS how you want your views laid out.

First, here's a dummy test case you can copy and paste into your project. It creates five labels of different colors and adds them all to your view:

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

  let label1 = UILabel()
  label1.translatesAutoresizingMaskIntoConstraints = false
  label1.backgroundColor = UIColor.redColor()
  label1.text = "THESE"

  let label2 = UILabel()
  label2.translatesAutoresizingMaskIntoConstraints = false
  label2.backgroundColor = UIColor.cyanColor()
  label2.text = "ARE"

  let label3 = UILabel()
  label3.translatesAutoresizingMaskIntoConstraints = false
  label3.backgroundColor = UIColor.yellowColor()
  label3.text = "SOME"

  let label4 = UILabel()
  label4.translatesAutoresizingMaskIntoConstraints = false
  label4.backgroundColor = UIColor.greenColor()
  label4.text = "AWESOME"
```

```
    let label5 = UILabel()

    label5.translatesAutoresizingMaskIntoConstraints = false

    label5.backgroundColor = UIColor.orangeColor()

    label5.text = "LABELS"


    view.addSubview(label1)

    view.addSubview(label2)

    view.addSubview(label3)

    view.addSubview(label4)

    view.addSubview(label5)

}
```

If you run the project, you'll see the labels are all bunched up in the top-left corner, which doesn't look great. To fix this, we're going to use VFL to have each label occupy the full width of the screen, then spaced out so they are position below each other.

When you use VFL you need to create a dictionary of the views you want to work with. This dictionary needs to have the name of the view inside VFL and a reference to the view itself, but in practice most people just use the same name for VFL as the variable like this:

```
let viewsDictionary = ["label1": label1, "label2": label2, "label3": label3, "label4": label4, "label5": label5]
```

Put that just below the final **addSubview()** call.

Now for the VFL itself: put this directly beneath the previous line in order to have every label stretch out to occupy the full width of the screen:

```
for label in viewsDictionary.keys {
```

```
view.addConstraints(NSLayoutConstraint.constraintsWithVisualFormat("H
:|[\(label)]|", options: [], metrics: nil, views: viewsDictionary))

}
```

Finally, add this to make the views lay themselves out below each other:

```
view.addConstraints(NSLayoutConstraint.constraintsWithVisualFormat("V
:|[label1]-[label2]-[label3]-[label4]-[label5]", options: [],
metrics: nil, views: viewsDictionary))
```

This is only the beginning of what VFL can do – you should definitely read my Auto Layout tutorial for more details.

For more information see Hacking with Swift tutorial 6.

How to create a page curl effect using UIPageViewController

Availability: iOS 5.0 or later.

When iBooks first launched in iOS 3.2, its page curl effect was almost addictive: it moved so fluently with your finger that it felt you were touching real paper. From iOS 5.0 on this page curl effect is available for every developer as part of the **UIPageViewController** class. Its API isn't immediately obvious to newbies, though, so I'm going to give you a complete example.

In the code below, the page view controller is created in **viewDidLoad()**. I also create five **UIViewControllers** to serve as pages inside the app, then tell the page view controller to start with the first one. I put in a couple of helper methods so that the view controllers could have random background colors so you can see it all working.

Most of the work is done by the **viewControllerBeforeViewController** and **viewControllerAfterViewController** methods, which must either return a view controller to show before or after the current one (when the users starts to turn the page) or **nil** to mean the user is at the end and there are no more pages to show in that direction.

To make this work in your own app, you'll obviously want to replace the plain view controller

pages with your own **UIViewController** subclass that does something more interesting. If you're showing quite a few different pages, you should probably create them on demand rather than creating an array of them all up front.

Anyway, here is the complete example – you can use this with the Xcode "Single View Application" to get a page view controller up and running immediately:

```
import UIKit


class ViewController: UIViewController,
UIPageViewControllerDataSource, UIPageViewControllerDelegate {

  var pageController: UIPageViewController!

  var controllers = [UIViewController]()


  override func viewDidLoad() {

    super.viewDidLoad()


    pageController = UIPageViewController(transitionStyle: .PageCurl,
navigationOrientation: .Horizontal, options: nil)

    pageController.dataSource = self

    pageController.delegate = self


    addChildViewController(pageController)

    view.addSubview(pageController.view)


    let views = ["pageController": pageController.view] as [String:
AnyObject]

view.addConstraints(NSLayoutConstraint.constraintsWithVisualFormat("H
:|[pageController]|", options: [], metrics: nil, views: views))

view.addConstraints(NSLayoutConstraint.constraintsWithVisualFormat("V
:|[pageController]|", options: [], metrics: nil, views: views))
```

```
    for _ in 1 ... 5 {

        let vc = UIViewController()

        vc.view.backgroundColor = randomColor()

        controllers.append(vc)

    }


    pageController.setViewControllers([controllers[0]],
direction: .Forward, animated: false, completion: nil)

  }




  func pageViewController(pageViewController: UIPageViewController,
viewControllerBeforeViewController viewController: UIViewController)
-> UIViewController? {

    if let index = controllers.indexOf(viewController) {

      if index > 0 {

        return controllers[index - 1]

      } else {

        return nil

      }

    }


    return nil

  }


  func pageViewController(pageViewController: UIPageViewController,
viewControllerAfterViewController viewController: UIViewController) -
> UIViewController? {

    if let index = controllers.indexOf(viewController) {

      if index < controllers.count - 1 {

        return controllers[index + 1]

      } else {
```

```
        return nil

      }

    }


    return nil

  }


  func randomCGFloat() -> CGFloat {

    return CGFloat(arc4random()) / CGFloat(UInt32.max)

  }


  func randomColor() -> UIColor {

      return UIColor(red: randomCGFloat(), green: randomCGFloat(),
blue: randomCGFloat(), alpha: 1)

  }

}
```

How to create a parallax effect in UIKit

Availability: iOS 7.0 or later.

Parallax effects have been standard since iOS 7.0, and the **UIInterpolatingMotionEffect**
class makes this easy by automatically smoothing accelerometer input so your views can
adjust to tilt data.

If you want to have a **UIView** respond to tilting, add this function to your code then call it on
any view you want:

```
func addParallaxToView(vw: UIView) {

  let amount = 100
```

```
    let horizontal = UIInterpolatingMotionEffect(keyPath: "center.x",
type: .TiltAlongHorizontalAxis)

    horizontal.minimumRelativeValue = -amount

    horizontal.maximumRelativeValue = amount


    let vertical = UIInterpolatingMotionEffect(keyPath: "center.y",
type: .TiltAlongVerticalAxis)

    vertical.minimumRelativeValue = -amount

    vertical.maximumRelativeValue = amount


    let group = UIMotionEffectGroup()

    group.motionEffects = [horizontal, vertical]

    vw.addMotionEffect(group)

}
```

How to create custom menus using UIMenuController
Availability: iOS 3.0 or later.

iOS has a built-in menu system that, while *useful*, doesn't actually get much *use* – because users don't expect to see it, developers don't use it, thus making it even less likely that users expect to see it.

Anyway, if you want to attach multiple actions to elements in your UI – pieces of text in a text view or web view, table view rows, and so on – you might find iOS menus are for you, so you need to turn to **UIMenuController**. This has extremely simple API: you just create a **UIMenuItem** object for every action you want, then register them all and wait for the user to do something.

Below is a complete example for a view controller that has a web view inside it – you'll need to create that in your storyboard. The code sets up a new menu item named "Grok" that runs the **runGrok()** method when tapped. I've made it do something real: when the user selects some text, they tap Grok to have that printed out to the Xcode console.

Here's the code:

```swift
class ViewController: UIViewController, UITextViewDelegate {
  @IBOutlet weak var webView: UIWebView!

  override func viewDidLoad() {
    super.viewDidLoad()

    webView.loadHTMLString("<p>Hello, world!</p>", baseURL: nil)
    enableCustomMenu()
  }

  func enableCustomMenu() {
    let lookup = UIMenuItem(title: "Grok", action: "runGrok")
    UIMenuController.sharedMenuController().menuItems = [lookup]
  }

  func disableCustomMenu() {
    UIMenuController.sharedMenuController().menuItems = nil
  }

  func runGrok() {
    let text =
webView.stringByEvaluatingJavaScriptFromString("window.getSelection()
.toString();")

    print(text)
  }
}
```

How to create popover menus using UIPopoverPresentationController

Availability: iOS 8.0 or later.

Show a **UIAlertController** action sheet on iPad isn't as easy as on iPhone. The reason for this is simple: on iPhone the action sheet slides up from the bottom, effectively owning the user's attention until it's dismissed, whereas on iPad it could be shown from anywhere. In fact, if you just try and show one on an iPad like this, your app crashes:

```
let ac = UIAlertController(title: "Hello!", message: "This is a test.", preferredStyle: .ActionSheet)

presentViewController(ac, animated: true, completion: nil)
```

The solution is to use a **UIPopoverPresentationController**, which gets created for you when you try to access the **popoverPresentationController** property of a **UIAlertController**. With this, you can tell it where to show from (and what view those coordinates relate to) before presenting the action sheet, which makes it work correctly on iPad.

To rewrite the previous lines so they work, you'd do this:

```
let ac = UIAlertController(title: "Hello!", message: "This is a test.", preferredStyle: .ActionSheet)


let popover = ac.popoverPresentationController

popover?.sourceView = view

popover?.sourceRect = CGRect(x: 32, y: 32, width: 64, height: 64)


presentViewController(ac, animated: true, completion: nil)
```

How to customize swipe edit buttons in a UITableView

Availability: iOS 8.0 or later.

As of iOS 8.0 there's an easy way to customize the list of buttons that appaer when the user swipes from right to left: **editActionsForRowAtIndexPath**. Return an array of **UITableViewRowAction** objects that have titles and styles (and also background colors if you want to customize their appearance), and iOS does the rest.

When you create a **UITableViewRowAction** object you give it a trailing closure describing what should happen when the user taps the button. You'll get reminded of what action triggered the code, and you'll also be given the index path of the row where the user was tapping. For example, you might do this:

```
override func tableView(tableView: UITableView,
editActionsForRowAtIndexPath indexPath: NSIndexPath) ->
[UITableViewRowAction]? {

  let delete = UITableViewRowAction(style: .Destructive, title:
"Delete") { (action, indexPath) in

    // delete item at indexPath

  }


  let share = UITableViewRowAction(style: .Normal, title: "Disable")
{ (action, indexPath) in

    // share item at indexPath

  }


  share.backgroundColor = UIColor.blueColor()


  return [delete, share]
}
```

Note that the first button uses a **Destructive** style so it will be colored red by default, but the second button specifically has a blue color assigned to it.

How to deselect a UITableViewCell using clearsSelectionOnViewWillAppear

Availability: iOS 3.2 or later.

When a user taps a table view row, it automatically gets highlighted by iOS, and frequently we use that action to show another view controller with more detailed information. When the user goes back, though, you probably want their selection to go away so that it doesn't remain selected, and if you're using a **UITableViewController** that's easy to do with **clearsSelectionOnViewWillAppear**If you set this property to be **true** the user's selected cell will automatically be deselected when they return to the table view. It does this intelligently, though: the row starts selected, and animates to be deselected, meaning that the user gets a brief reminder of the row they tapped before it gets deselected.

How to detect a double tap gesture

Availability: iOS 3.2 or later.

The iOS **UITapGestureRecognizer** class has a built-in way to detect a double tap on any view. All you need to do is create the recognizer, set its **numberOfTapsRequired** property to 2, then add it to the view you want to monitor.

Here's an example:

```
override func viewDidLoad() {

    super.viewDidLoad()


  let tap = UITapGestureRecognizer(target: self, action:
"doubleTapped")

  tap.numberOfTapsRequired = 2

  view.addGestureRecognizer(tap)

}


func doubleTapped() {

  // do something cool here
```

```
}
```

How to detect edge swipes

Availability: iOS 7.0 or later.

Detecting pan gestures is easy enough with a regular **UIPanGestureRecognizer**, but there's a special gesture recognizer to use if you want to detect the user swiping from the edge of their screen. The example below demonstrates detecting the user swiping from the left edge of the screen:

```
override func viewDidLoad() {

  let edgePan = UIScreenEdgePanGestureRecognizer(target: self,
action: "screenEdgeSwiped:")

  edgePan.edges = .Left


  view.addGestureRecognizer(edgePan)

}


func screenEdgeSwiped(recognizer: UIScreenEdgePanGestureRecognizer) {

  if recognizer.state == .Recognized {

    print("Screen edge swiped!")

  }

}
```

How to detect when the Back button is tapped

Availability: iOS 2.0 or later.

You probably already know that **viewWillDisappear()** is called when a view controller is about to go away, and that's also called when the user taps the Back button in a navigation controller. Problem is, the same method is called when the user moves forward, i.e. when you push another view controller onto the stack.

The solution is simple: create a Boolean property called **goingForwards** in your view controller, and set it to **true** before pushing any view controller onto the navigation stack, then set it back to **false** when the view controller is shown again. This way, when **viewWillDisappear()** is called you can check **goingForwards**: if it's false, the user tapped Back.

How to dim the screen
Availability: iOS 2.0 or later.

There is no built-in way to dim the screen unless you're presenting a view controller, at which point iOS dims the background view controller for you. Instead, if you want to dim stuff you need to do it yourself: create a full-screen **UIView** with a translucent background color (I find 66% black works best) then set its alpha to be 0. When you want things to dim, set the alpha to be 1.

How to draw custom views in Interface Builder using IBDesignable
Availability: iOS 8.0 or later.

You've always been able to have custom views inside your apps, but if you're having a hard time visualizing how they look at design time then you should try **@IBDesignable**: it lets you see exactly how your custom views look inside IB, and if you combine it with **@IBInspectable** you can even adjust your view's design there too.

This example view draws an ellipse that fills itself. If you add this to your project, create a view, then set that view to have this custom subclass, you'll see an ellipse appear immediately. You can move the view or resize it, and the ellipse will be updated. Plus, because I used **@IBInspectable** you can adjust the colors and stroke width right inside the attributes inspector, helping you make sure your UI looks exactly as you expect.

```
@IBDesignable class EllipseView: UIView {
   @IBInspectable var strokeWidth: CGFloat = 0

   @IBInspectable var fillColor: UIColor = UIColor.blackColor()
   @IBInspectable var strokeColor: UIColor = UIColor.clearColor()

   override func drawRect(rect: CGRect) {
      let context = UIGraphicsGetCurrentContext()
      let rectangle = CGRectInset(frame, strokeWidth / 2, strokeWidth /
2)

      CGContextSetFillColorWithColor(context, fillColor.CGColor)
      CGContextSetStrokeColorWithColor(context, strokeColor.CGColor)
      CGContextSetLineWidth(context, strokeWidth)

      CGContextAddEllipseInRect(context, rectangle)
      CGContextDrawPath(context, .FillStroke)
   }
}
```

How to find a UIView subview using viewWithTag()

Availability: iOS 2.0 or later.

If you need a quick way to get hold of a view inside a complicated view hierarchy, you're looking for **viewWithTag()** – give it the tag to find and a view to search from, and this method will search all subviews, and all sub-subviews, and so on, until it finds a view with the matching tag number. The method returns an optional **UIView** because it might not find a view with that tag, so unwrap it carefully.

Here's an example:

```
if let foundView = view.viewWithTag(0xDEADBEEF) {

  foundView.removeFromSuperview()

}
```

Easy to remember tags such as **0xDEADBEEF** are quite common amongst coders.

NB: Extensive use of **viewWithTag()** is a sign of code structure. It's good for the occasional shortcut, but really shouldn't be relied on for serious development.

For more information see Hacking with Swift tutorial 25.

How to flip a UIView with a 3D effect: transitionWithView()
Availability: iOS 2.0 or later.

iOS has a built-in way to transition between views, and you can use this to produce 3D flips in just a few lines of code. Here's a basic example that flips between two views:

```
func flip() {

  let transitionOptions: UIViewAnimationOptions =
[.TransitionFlipFromRight, .ShowHideTransitionViews]


  UIView.transitionWithView(firstView, duration: 1.0, options:
transitionOptions, animations: {

    self.firstView.hidden = true

  }, completion: nil)


  UIView.transitionWithView(secondView, duration: 1.0, options:
transitionOptions, animations: {

    self.secondView.hidden = false

  }, completion: nil)
```

```
}
```

Here's a basic test harness you can use to see that method in action:

```
var firstView: UIView!
var secondView: UIView!

override func viewDidLoad() {
  super.viewDidLoad()

  firstView = UIView(frame: CGRect(x: 32, y: 32, width: 128, height:
128))
  secondView = UIView(frame: CGRect(x: 32, y: 32, width: 128, height:
128))

  firstView.backgroundColor = UIColor.redColor()
  secondView.backgroundColor = UIColor.blueColor()

  secondView.hidden = true

  view.addSubview(firstView)
  view.addSubview(secondView)

  performSelector("flip", withObject: nil, afterDelay: 2)
}
```

Try experimenting with the different values of **UIViewAnimationOptions** to see what other animations are available.

How to give UITableViewCells a selected color other than grey

Availability: iOS 7.0 or later.

Ever since iOS 7.0, table view cells have been grey when tapped, even when you specifically told Interface Builder you wanted them to be blue. Don't worry, though: it's an easy thing to change, as long as you don't mind writing three lines of code. Specifically, you need to add something like this to your **cellForRowAtIndexPath** method:

```
let backgroundView = UIView()
backgroundView.backgroundColor = UIColor.redColor()
cell.selectedBackgroundView = backgroundView
```

You can customize the view as much as you want to, but remember to keep the overall amount of work low to avoid hurting scroll performance.

How to give a UINavigationBar a background image: setBackgroundImage()

Availability: iOS 5.0 or later.

You can call **setBackgroundImage()** on any navigation bar, providing an image to use and the bar metrics you want it to affect, and you're done. Bar metrics left you specify what sizes of bars you want to change: should this by all bars, just phone-sized bars, or just phone-sized bars on landscape?

Here's an example that changes the navigation bar background image to a file called "navbar-background.png" (you'll want to change that!) across all device sizes:

```
let img = UIImage(named: "navbar-background")
navigationController?.navigationBar.setBackgroundImage(img,
forBarMetrics: .Default)
```

How to give a UIStackView a background color

Availability: iOS 9.0 or later.

You can't do this – **UIStackView** is a non-drawing view, meaning that **drawRect()** is never called and its background color is ignored. If you desperately want a background color, consider placing the stack view inside another **UIView** and giving that view a background color.

How to hide passwords in a UITextField

Availability: iOS 2.0 or later.

User text in a **UITextField** is visible by default, but you can enable the iOS text-hiding password functionality just by setting your text field's **secureTextEntry** property to be true, like this:

```
textField.secureTextEntry = true
```

How to hide the navigation bar using hidesBarsOnSwipe

Availability: iOS 8.0 or later.

iOS 8.0 gives **UINavigationController** a simple property that masks some complex behavior. If you set **hidesBarsOnSwipe** to be true for any **UINavigationController**, then iOS automatically adds a tap gesture recognizer to your view to handle hiding (and showing) the navigation bar as needed. This means you can mimic Safari's navigation bar behavior in just one line of code, like this:

```
navigationController?.hidesBarsOnSwipe = true
```

Remember to set this back to **false** when you want to stop the behavior from happening.

How to hide the navigation bar using hidesBarsOnTap

Availability: iOS 8.0 or later.

As of iOS 8.0 it's easy to make a navigation bar automatically hide when the user taps the screen, but only when it's part of a **UINavigationController**. When set to **true**, the **hidesBarsOnTap** property of a navigation controller automatically adds a tap gesture recognizer to your view to handle hiding (and showing) the navigation bar as needed.

Code:

```
navigationController?.hidesBarsOnTap = true
```

Remember to set this back to **false** when you want to stop the behavior from happening.

For more information see Hacking with Swift tutorial 1.

How to hide the status bar

Availability: iOS 7.0 or later.

You can hide the status bar in any or all of your view controllers just by adding this code:

```
override func prefersStatusBarHidden() -> Bool {
  return true
}
```

Any view controller containing that code will hide the status bar by default. If you want to animate the status bar in or out, just call **setNeedsStatusBarAppearanceUpdate()** on your view controller – that will force **prefersStatusBarHidden()** to be called again, at which point you can return a different value. If you want, your call to **setNeedsStatusBarAppearanceUpdate()** can actually be inside an animation block, which causes the status bar to hide or show in a smooth way.

How to let users tap on a UITableViewCell while editing is enabled

Availability: iOS 3.0 or later.

As soon as you set the **editing** property of a **UITableView** to be true, its cells stop being tappable. This is often a good idea, because if a user explicitly enabled editing mode they probably want to delete or move stuff, and it's only going to be annoying if they can select rows by accident.

Of course, as always, there are times when you specifically want both actions to be available - for the user to be able to move or delete a cell, and also tap on it to select. If that's the situation you find yourself in right now, here's the line of code you need:

```
tableView.allowsSelectionDuringEditing = true
```

How to load a HTML string into a WKWebView or UIWebView: loadHTMLString()

Availability: iOS 2.0 or later.

If you want to generate HTML locally and show it inside your app, it's easy to do in both **UIWebView** and **WKWebView**. First, here's the code for **UIWebView**:

```
let webView = UIWebView()

webView.loadHTMLString("<html><body><p>Hello!</p></body></html>",
baseURL: nil)
```

And now here's the code for **WKWebView**:

```
let webView = WKWebView()
webView.loadHTMLString("<html><body><p>Hello!</p></body></html>",
baseURL: nil)
```

If you want to load resources from a particular place, such as JavaScript and CSS files, you can set the **baseURL** parameter to any **NSURL**. This could, for example, be the resource path for your app bundle, which would allow you to use local images and other assets alongside your generated HTML.

For more information see Hacking with Swift tutorial 7.

How to make UITableViewCell separators go edge to edge
Availability: iOS 8.0 or later.

All table view cells have a separator underneath them by default, and that separator likes to start a little way from the left edge of the screen for stylistic reasons. If this clashes with your own personal aesthetic, you might *think* it's easy to remove but Apple has made the matter quite confused by changing its mind more than once.

If you absolutely, definitely want to remove the separator inset from all cells, you need to do two things. First, add these two lines of code to your table view controller's **viewDidLoad()** method:

```
tableView.layoutMargins = UIEdgeInsetsZero
tableView.separatorInset = UIEdgeInsetsZero
```

Now look for you **cellForRowAtIndexPath** method and add this:

```
cell.layoutMargins = UIEdgeInsetsZero
```

Done!

How to make UITableViewCells auto resize to their content
Availability: iOS 8.0 or later.

If you're using Auto Layout, you can have your table view cells automatically size to fit their content by adding two methods to your table view controller: **heightForRowAtIndexPath** and **estimatedHeightForRowAtIndexPath**. If both of these are implemented, and both return **UITableViewAutomaticDimension**, then your table view cells will be measured using Auto Layout and automatically fit their content.

If you're using the built-in table cell styles (e.g. Default or Subtitle) then you should make sure you modify the **numberOfLines** properties of your cell's labels so that the text can grow as needed.

In case you're still not sure, here's some example code:

```
override func tableView(tableView: UITableView,
heightForRowAtIndexPath indexPath: NSIndexPath) -> CGFloat {

    return UITableViewAutomaticDimension

}


override func tableView(tableView: UITableView,
estimatedHeightForRowAtIndexPath indexPath: NSIndexPath) -> CGFloat {

    return UITableViewAutomaticDimension

}
```

For more information see Hacking with Swift tutorial 32.

How to make a button glow when tapped with showsTouchWhenHighlighted

Availability: iOS 2.0 or later.

If you want an easy way to let users know when a **UIButton** was tapped, try setting its **showsTouchWhenHighlighted** property to be true. This will render a circular glow effect behind the button when it's tapped, which is particularly effective on text-only buttons.

Here's the code:

```
btn.showsTouchWhenHighlighted = true
```

If you're using Interface Builder, look for and check the "Shows touch when highlighted" option for your button.

How to make a clear button appear in a textfield

Availability: iOS 2.0 or later.

If you want to let users clear their entry on a **UITextField**, the standard approach is to add a clear button to the right edge of the text field. This can be done in two ways, depending on what effect you want:

```
textField.clearButtonMode = .Always
textField.clearButtonMode = .WhileEditing
```

The first will always show the clear button, and the second will only show it while the user is editing the text.

How to make the master pane always visible in a

UISplitViewController

Availability: iOS 8.0 or later.

Split view controllers on iPad have an automatic display mode, which means in landscape both the left and right view controllers are visible, but in portrait the left view controller slides over and away as needed.

If this isn't preferable – if, for example, you want to mimic the way the Settings app works in portrait – you can force both view controllers to be visible at all times like this:

```
splitViewController.preferredDisplayMode =
UISplitViewControllerDisplayMode.AllVisible
```

How to pad a UITextView by setting its text container inset

Availability: iOS 7.0 or later.

You can force the text of any **UITextView** to have padding – i.e., to be indented from its edges – by setting its **textContainerInset** property to a value of your choosing. For example, to give a text view insets of 50 points from each edge, you would use this code:

```
textView.textContainerInset = UIEdgeInsets(top: 50, left: 50, bottom:
50, right: 50)
```

How to print using UIActivityViewController

Availability: iOS 6.0 or later.

Printing in iOS used to be done using **UIPrintInteractionController**, and, while that still works, it has a much better replacement in the form of **UIActivityViewController**. This new class is responsible for taking a wide variety of actions of which printing is just one, but users can also tweet, post to Facebook, send by email, and any other action that has been registered by another app.

If you have a **UIImage** you want to print, you can just pass it in. If you want to print text, you can wrap it inside an **NSAttributedString** with some formatting, then place that inside a **UISimpleTextPrintFormatter** object, then print *that* – iOS automatically takes care of pagination, margins and more.

Below are two example functions that print an image and some text to help get you started:

```
func shareImage(yourImage: UIImage) {

    let vc = UIActivityViewController(activityItems: [yourImage],
applicationActivities: [])

    presentViewController(vc, animated: true, completion: nil)

}


func shareText(text: String) {

  let attrs = [NSFontAttributeName: UIFont.systemFontOfSize(72),
NSForegroundColorAttributeName: UIColor.redColor()]

  let str = NSAttributedString(string: text, attributes: attrs)

  let print = UISimpleTextPrintFormatter(attributedText: str)


  let vc = UIActivityViewController(activityItems: [print],
applicationActivities: nil)

  presentViewController(vc, animated: true, completion: nil)

}
```

For more information see Hacking with Swift tutorial 3.

How to put a background picture behind UITableViewController

Availability: iOS 3.2 or later.

You can put any type of **UIView** behind a table view, and iOS automatically resizes it to fit the table. So, adding a background picture is just a matter of using a **UIImageView** like this:

```
tableView.backgroundView = UIImageView(image: UIImage(named: "taylor-
swift"))
```

How to read a title from a UIPickerView using titleForRow
Availability: iOS 2.0 or later.

As soon as you start using **UIPickerView** for the first time, you realise it doesn't have a built-in way to read the title of any of its items. The reason for this is obvious in retrospect, but don't worry if you didn't get it at first: you should read the title straight from the picker's data source.

You should already have conformed to the **UIPickerViewDataSource** and **UIPickerViewDelegate** protocols, which means implementing the **titleForRow** picker view method. If you want to read the title of the selected item later, you can do one of the following:

 • Read straight from the array you used to populate the picker view. This is the most common method, but of course it only works if the data is simple.
 • Write a new method named something like **titleForPickerRow()** that you can use in your data source and to read the title later. This is preferred if it takes some work to calculate row titles, but really it's better to cache this kind of thing if the work is non-trivial.
 • Use the same method call as the picker view: **pickerView(_:titleForRow:forComponent:)**. Yes, that just calls the method you implemented, but it's neat and self-describing so as long as your data doesn't take time to calculate this is fine to use.

If you want to try the last option, here's some example code:

```
let title = pickerView(yourPickerView, titleForRow: 0, forComponent:
0)
```

How to recolor UIImages using template images and

imageWithRenderingMode

Availability: iOS 7.0 or later.

Template images are the iOS 7.0 way of tinting any kind of image when it's inside a **UIImageView**. This is usually used to mimic the tinting of button images (as seen in toolbars and tab bars) but it works anywhere you want to dynamically recolor an image.

To get started, load an image then call **imageWithRenderingMode()** on it, like this:

```
if let myImage = UIImage(named: "myImage") {

  let tintableImage = myImage.imageWithRenderingMode(.AlwaysTemplate)

  imageView.image = tintableImage

}
```

The tint color of a **UIImageView** is the standard iOS 7 blue by default, but you can change it easily enough:

```
imageView.tintColor = UIColor.redColor()
```

How to register a cell for UICollectionView reuse

Availability: iOS 6.0 or later.

If you're working entirely in code, you can register a **UICollectionViewCell** subclass for use with your collection view, so that new cells are dequeued and re-use automatically by the system.

Here's the most basic form of this technique:

```
collectionView.registerClass(UICollectionViewCell.self,
forCellWithReuseIdentifier: "Cell")
```

That registers a basic collection view cell, which you can then customise in code if you want to. You can then dequeue a cell with this:

```
func collectionView(collectionView: UICollectionView,
cellForItemAtIndexPath indexPath: NSIndexPath) ->
UICollectionViewCell {

  let cell =
collectionView.dequeueReusableCellWithReuseIdentifier("Cell",
forIndexPath: indexPath)

  return cell

}
```

If a cell doesn't already exist that can be re-used, a new one will be created automatically.

As you might imagine, you will most of the time want to create your own custom **UICollectionViewCell** subclass and use that instead, but the code is the same – just use your class name instead.

If you're working with Interface Builder, all this work is done for you by creating prototype cells.

How to register a cell for UITableViewCell reuse
Availability: iOS 6.0 or later.

Reusing table view cells has been one of the most important performance optimizations in iOS ever since iOS 2.0, but it was only with iOS 6.0 that the API got cleaned up a little with the addition of two new methods on **UITableView**: **registerClass()** and **registerNib()**.

Both of these method calls take a parameter **forCellReuseIdentifier**, which is a string that lets you register different kinds of table view cells. For example, you might have a reuse identifier "DefaultCell", another one called "Heading cell", another one "CellWithTextField", and so on. Re-using different cells this way helps save system resources.

If you want to use **registerClass()**, you provide a table view cell class as its first parameter.

This is useful if your cell is defined entirely in code. As an example, this uses the default **UITableViewCell** class:

```
tableView.registerClass(UITableViewCell.self, forCellReuseIdentifier:
"DefaultCell")
```

You can then dequeue that cell like this:

```
func tableView(tableView: UITableView, cellForRowAtIndexPath
indexPath: NSIndexPath) -> UITableViewCell {

  let cell =
tableView.dequeueReusableCellWithIdentifier("DefaultCell")!

  return cell

}
```

If there aren't any cells created that can be reused, iOS will automatically create them – this API really is very easy.

The other option is to use **registerNib()**, which takes an Interface Builder nib file and loads from there. Nibs contain the class name to use along with their design, so this method is more common. Here's an example

```
tableView.registerNib(UINib(nibName: "yourNib", bundle: nil),
forCellReuseIdentifier: "CellFromNib")
```

Although knowing the above code is definitely useful, if you're using storyboards you will find it easier to create prototype cells and give them a reuse identifier directly inside Interface Builder.

For more information see Hacking with Swift tutorial 33.

How to remove a UIView from its superview with removeFromSuperview()

Availability: iOS 2.0 or later.

If you created a view dynamically and want it gone, it's a one-liner in Swift thanks to the **removeFromSuperview()** method. When you call this, the view gets removed immediately and possibly also destroyed – it will only be kept around if you have a reference to it elsewhere. Here's how it's done:

```
yourView.removeFromSuperview()
```

How to remove cells from a UITableViww

Availability: iOS 2.0 or later.

It's easy to delete rows from a table view, but there is one catch: you need to remove it from the data source first. If you don't do this, iOS will realize there's a mis-match between what the data source thinks should be showing and what the table view is actually showing, and you'll get a crash.

So, to remove a cell from a table view you first remove it from your data source, then you call **deleteRowsAtIndexPaths()** on your table view, providing it with an array of index paths that should be zapped. You can create index paths yourself, you just need a section and row number.

Here's some example code to get you started:

```
objects.removeAtIndex(0)


let indexPath = NSIndexPath(forItem: 0, inSection: 0)

tableView.deleteRowsAtIndexPaths([indexPath],
withRowAnimation: .Fade)
```

How to respond to the device being shaken

Availability: iOS 3.0 or later.

You can make any **UIViewController** subclass respond to the device being shaken by overriding the **motionBegan** method. This is use to handle motion (shaking) but in theory also remote control actions – although I can't say I've ever seen someone write code to handle that!

This code will print a message every time the device is shaken:

```
override func motionBegan(motion: UIEventSubtype, withEvent event:
UIEvent?) {

  print("Device was shaken!")

}
```

For more information see Hacking with Swift tutorial 20.

How to scale, stretch, move and rotate UIViews using CGAffineTransform

Availability: iOS 2.0 or later.

Every **UIView** subclass has a **transform** property that lets you manipulate its size, position and rotation using something called an affine transform. This property is animatable, which means yuo can make a view smoothly double in size, or make it spin around, just by changing one value.

Here are some examples to get you started:

```
imageView.transform = CGAffineTransformMakeScale(2, 2)
imageView.transform = CGAffineTransformMakeTranslation(-256, -256)
```

```
imageView.transform = CGAffineTransformMakeRotation(CGFloat(M_PI))

imageView.transform = CGAffineTransformIdentity
```

The first one makes an image view double in size, the second one makes it move up and left 256 points, the third one makes it spin around 180 degrees (the values are expressed in radians), and the fourth one sets the image view's transform back to "identity" – this means "reset."

For more information see Hacking with Swift tutorial 15.

How to send an email

Availability: iOS 3.0 or later.

In the MessageUI framework lies the **MFMailComposeViewController** class, which handles sending emails from your app. You get to set the recipients, message title and message text, but you don't get to send it – that's for the user to tap themselves.

Here's some example code:

```
func sendEmail() {
  if MFMailComposeViewController.canSendMail() {
    let mail = MFMailComposeViewController()
    mail.mailComposeDelegate = self
    mail.setToRecipients(["paul@hackingwithswift.com"])
    mail.setMessageBody("<p>You're so awesome!</p>", isHTML: true)

    presentViewController(mail, animated: true, completion: nil)
  } else {
    // show failure alert
  }
}
```

```
func mailComposeController(controller: MFMailComposeViewController,
didFinishWithResult result: MFMailComposeResult, error: NSError?) {
    controller.dismissViewControllerAnimated(true, completion: nil)
}
```

Make sure you add **import MessageUI** to any Swift file that uses this code!

Note that not all users have their device configure to send emails, which is why we need to check the result of **canSendMail()** before trying to send. Note also that you need to catch the **didFinishWithResult** callback in order to dismiss the mail window.

Warning: this code frequently fails in the iOS Simulator. If you want to test it, try on a real device.

How to set a custom title view in a UINavigationBar
Availability: iOS 2.0 or later.

Each view controller has a **navigationItem** property that dictates how it customizes the navigation bar if it is viewed inside a navigation controller. This is where you add left and right bar button items, for example, but also where you can set a title view: any **UIView** subclass that is used in place of the title text in the navigation bar.

For example, if you wanted to show an image of your logo rather than just some text you would use this:

```
navigationItem.titleView = UIImageView(image: UIImage(named: "logo"))
```

How to set prompt text in a navigation bar
Availability: iOS 2.0 or later.

You should already know that you can give a title to a navigation controller's bar by setting

the **title** property of your view controllers or a view controller's **navigationItem**, but did you also know that you can provide prompt text too? When provided, this causes your navigation bar to double in height and show both the title and prompt.

It's easy to do:

```
navigationItem.title = "Your title text here"
navigationItem.prompt = "Your prompt text here"
```

Your prompt text appears above your title text, so choose carefully.

How to set the tabs in a UITabBarController

Availability: iOS 2.0 or later.

If you're creating your tab bar controller from scratch, or if you just want to change the set up of your tabs at runtime, you can do so just by setting the **viewControllers** property of your tab bar controller. This expects to be given an array of view controllers in the order you want them displayed, and you should already have configured each view controller to have its own **UITabBarItem** with a title and icon.

If your tab bar controller is the root view controller of your window, you should be able to write something like this:

```
if let tabBarController = window?.rootViewController as?
UITabBarController {
  let first = FirstViewController()
  let second = SecondViewController()


  tabBarController.viewControllers = [first, second]
}
```

For more information see Hacking with Swift tutorial 7.

How to set the tint color of a UIView

Availability: iOS 7.0 or later.

The **tintColor** property of any **UIView** subclass lets you change the coloring effect applied to it. The exact effect depends on what control you're changing: for navigation bars and tab bars this means the text and icons on their buttons, for text views it means the selection cursor and highlighted text, for progress bars it's the track color, and so on.

tintColor can be set for any individual view to color just one view, for the whole view in your view controller to color all its subviews, or even for the whole window in your application so that all views and subviews are tinted at once.

To tint just the current view controller, use this code:

```
override func viewDidLoad() {

  view.tintColor = UIColor.redColor()

}
```

If you want to tint all views in your app, put this in your AppDelegate.swift:

```
func application(application: UIApplication,
didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?)
-> Bool {

  // Override point for customization after application launch.

  window?.tintColor = UIColor.redColor()


  return true

}
```

How to share content with UIActivityViewController

Availability: iOS 6 or later.

Before iOS 6.0 was released there were a number of third-party libraries that tried to simplify the sharing of content, but even with those libraries in place it was still far too hard. Fortunately, Apple added **UIActivityViewController**, a class that makes sharing to any service as simple as telling it what kind of content you have.

The nice thing about **UIActivityViewController** is that it automatically takes advantage of the apps the user has installed. If they have configured Twitter, they can post tweets; if they have configured Facebook, they can post to their timeline; if they have a printer configured, they can print your images; and more. It takes no extra work from you: you just tell iOS what kind of content you want to share, and it does the rest.

Here's how you share an image:

```
if let image = UIImage(named: "myImage") {
  let vc = UIActivityViewController(activityItems: [image],
applicationActivities: [])
  presentViewController(vc, animated: true, completion: nil)
}
```

And here's an example of sharing a text and an image:

```
let shareText = "Hello, world!"

if let image = UIImage(named: "myImage") {
  let vc = UIActivityViewController(activityItems: [shareText,
image], applicationActivities: [])
  presentViewController(vc, animated: true, completion: nil)
}
```

If you want to share a URL to a website, make sure you wrap it up in an **NSURL** first.

For more information see Hacking with Swift tutorial 3.

How to share content with the Social framework and SLComposeViewController

Availability: iOS 6.0 or later.

The **UIActivityViewController** class is the iOS way of sharing almost anything to almost anywhere, but what if you don't want to let users choose? Well, iOS has a tool for that too, and it's called the Social framework. Start by importing that now:

```
import Social
```

You can now create and present a **SLComposeViewController** that allows the user to share to Facebook like this:

```
let vc = SLComposeViewController(forServiceType:
SLServiceTypeFacebook)

vc.setInitialText("Look at this great picture!")

vc.addImage(UIImage(named: "myImage.jpg")!)

vc.addURL(NSURL(string: "https://www.hackingwithswift.com"))

presentViewController(vc, animated: true, completion: nil)
```

That attaches initial text, an image and a URL all to that share sheet, although the user can customize the text before posting. If you want to use Twitter instead, try this:

```
let vc = SLComposeViewController(forServiceType:
SLServiceTypeTwitter)
```

For more information see Hacking with Swift tutorial 3.

How to show and hide a toolbar inside a UINavigationController

Availability: iOS 3.0 or later.

All navigation controllers have a toolbar built right in, but it's not showing by default. And even if it were showing, it doesn't have any items by default – that's down to you fill in.

To get started, give a view controller some toolbar items by setting its **toolbarItems** property like this:

```
let add = UIBarButtonItem(barButtonSystemItem: .Add, target: self,
action: "addTapped")
let spacer = UIBarButtonItem(barButtonSystemItem: .FlexibleSpace,
target: self, action: nil)
toolbarItems = [add, spacer]
```

You can now tell the navigation controller to show its toolbar like this:

```
navigationController?.setToolbarHidden(false, animated: false)
```

If you animate between two view controllers with different toolbar items, iOS automatically animates their change.

For more information see Hacking with Swift tutorial 4.

How to stop Auto Layout and autoresizing masks conflicting: translatesAutoresizingMaskIntoConstraints

Availability: iOS 7.0 or later.

If you create any views in code – text views, buttons, labels, etc – you need to be careful how you add Auto Layout constraints to them. The reason for this is that iOS creates constraints for you that match the new view's size and position, and if you try to add your own constraints these will conflict and your app will break.

There are two solutions. First, don't add Auto Layout constraints to views created in code. Sound like a bad idea? That's because it is: like it or lump it, Auto Layout is something you want on your side. So, that leaves option two: tell iOS *not* to create Auto Layout constraints automatically, and that's done with this line of code:

```
yourView.translatesAutoresizingMaskIntoConstraints = false
```

For more information see Hacking with Swift tutorial 6.

How to stop empty row separators appearing in UITableView

Availability: iOS 2.0 or later.

Table views show separators between empty rows by default, which looks quite strange when you have only a handful of visible rows. Fortunately, one simple line of code is all it takes to force iOS not to draw these separators, and it's this:

```
tableView.tableFooterView = UIView()
```

What's actually happening is that you're creating an empty **UIView** and making it act as the footer of the table – this is the bottom most thing visible in the table. When iOS reaches the bottom of the cells you provide, it draws this view at the end rather than drawing empty rows and their separators, so it totally clears up the problem.

How to stop users selecting text in a UIWebView or

WKWebView

Availability: iOS 2.0 or later.

Using a webview to show rich media easily is a common thing to do, but by default users can select the text and that makes it look a little less like native code. To fix this, add the following CSS to the HTML you load, and users won't be able to select anything again:

```
<style type="text/css">
* {
    -webkit-touch-callout: none;
    -webkit-user-select: none;
}
</style>
```

How to stop your view going under the navigation bar using edgesForExtendedLayout

Availability: iOS 7.0 or later.

As of iOS 7.0, all views automatically go behind navigation bars, toolbars and tab bars to provide what Apple calls "context" – having some idea of what's underneath the UI (albeit blurred out with a frosted glass effect) gives users an idea of what else is just off screen.

If this is getting in your way (and honestly it does get in the way surprisingly often) you can easily disable it for a given view controller by modifying its **edgesForExtendedLayout** property.

For example, if you don't want a view controller to go behind any bars, use this:

```
edgesForExtendedLayout = .None
```

How to style the font in a UINavigationBar's title

Availability: iOS 6.0 or later.

If you're setting title's in a navigation bar, you can customise the font, size and color of those titles by adjusting the **titleTextAttributes** attribute for your navigation bar. To do this on a single bar just set it directly whenever you want to; to change all bars, set it inside your app delegate using the appearance proxy for **UINavigationBar** so that it kicks in before the first bar is loaded.

Here's an example that makes title text be 24-point Georgia Bold in red:

```
let attrs = [
    NSForegroundColorAttributeName : UIColor.redColor(),
    NSFontAttributeName : UIFont(name: "Georgia-Bold", size: 24)!
]

UINavigationBar.appearance().titleTextAttributes = attrs
```

How to support pinch to zoom in a UIScrollView

Availability: iOS 2.0 or later.

Making a scroll view zoom when you pinch is a multi-step approach, and you need to do all the steps in order for things to work correctly.

First, make sure your scroll view has a maximum zoom scale larger than the default of 1.0. You can change this in Interface Builder if you want, or use the **maximumZoomScale** property in code.

Second, make your view controller the delegate of your scroll view. Again, you can do this in Interface Builder by Ctrl-dragging from the scroll view to your view controller.

Third, make your view controller conform to the **UIScrollViewDelegate** protocol, then add the **viewForZoomingInScrollView()** method, like this:

```
func viewForZoomingInScrollView(scrollView: UIScrollView) -> UIView?
{

   return someView

}
```

That's it for code, but make sure you create your layouts consistently – whether you use Auto Layout or not, you need to be careful to **follow Apple's instructions**.

How to swipe to delete UITableViewCells

Availability: iOS 2.0 or later.

It takes just one method to enable swipe to delete in table views:
tableView(_:commitEditingStyle:forRowAtIndexPath:). This method gets called when a user tries to delete one of your table rows using swipe to delete, but its very presence is what enables swipe to delete in the first place – that is, iOS literally checks to see whether the method exists, and, if it does, enables swipe to delete.

When you want to handle deleting, you have to do three things: first, check that it's a delete that's happening and not an insert (this is down to how you use the UI); second, delete the item from the data source you used to build the table; and third, call **deleteRowsAtIndexPaths()** on your table view.

It is crucial that you do those things in exactly that order. iOS checks the number of rows before and after a delete operation, and expects them to add up correctly following the change.

Here's an example that does everything correctly:

```
override func tableView(tableView: UITableView, commitEditingStyle
editingStyle: UITableViewCellEditingStyle, forRowAtIndexPath
indexPath: NSIndexPath) {

   if editingStyle == .Delete {

      objects.removeAtIndex(indexPath.row)
```

```
        tableView.deleteRowsAtIndexPaths([indexPath],
withRowAnimation: .Fade)

    } else if editingStyle == .Insert {

        // Create a new instance of the appropriate class, insert it
into the array, and add a new row to the table view.

    }

}
```

How to use Dynamic Type to resize your app's text

Availability: iOS 7.0 or later.

As of iOS 7.0 users can set a system-wide preferred font size for all apps, but many programmers ignore this setting much to user's annoyance. You're not one of *those* developers, are you? Of course not! So here's how to honor a user's font settings using **UIFont**:

```
let headlineFont =
UIFont.preferredFontForTextStyle(UIFontTextStyleHeadline)
let subheadFont =
UIFont.preferredFontForTextStyle(UIFontTextStyleSubheadline)
```

And that's it! This technology is called Dynamic Type, and it's powerful because that code will return correctly sized fonts for the user's preference, which means your app's text will shrink or grow as needed.

Note that it is technically possible for users to change their Dynamic Type setting while your app is running. If you want to cover this corner case, use **NSNotificationCenter** to subscribe to the **UIContentSizeCategoryDidChangeNotification** notification then refresh your user interface if you receive it.

For more information see Hacking with Swift tutorial 32.

How to use IBInspectable to adjust values in Interface Builder

Availability: iOS 8.0 or later.

The **@IBInspectable** keyword lets you specify that some parts of a custom **UIView** subclass should be configurable inside Interface Builder. Only some kinds of values are supported (booleans, numbers, strings, points, rects, colors and images) but that ought to be enough for most purposes.

When your app is run, the values that were set in Interface Builder are automatically set, just like any other IB value. Neat, huh?

Here's an example that creates a **GradientView** class. This wraps the **CAGradientLayer** class up in a **UIView** that you can place anywhere in your app. Even better, thanks to **@IBInspectable** you can customize the colors in your gradient right inside IB. Add this class to your project now:

```
@IBDesignable class GradientView: UIView {
  @IBInspectable var startColor: UIColor = UIColor.whiteColor()
  @IBInspectable var endColor: UIColor = UIColor.whiteColor()

  override class func layerClass() -> AnyClass {
    return CAGradientLayer.self
  }

  override func layoutSubviews() {
    (layer as! CAGradientLayer).colors = [startColor.CGColor,
endColor.CGColor]
  }
}
```

Now go to IB, drop a **UIView** on to your storyboard, then change its class to be **GradientView**. Once that's done, Xcode will compile your project automatically, and then inside the attributes inspector you'll see two color selectors for the start and end color.

Note: **@IBInspectable** frequently does not play nicely with type inference, which is why I've explicitly declared both the type (**UIColor**) and default value (**UIColor.whiteColor()**).

How to use SFSafariViewController to show web pages in your app

Availability: iOS 9.0 or later.

If a user clicks a web link in your app, you used to have two options before iOS 9.0 came along: exit your app and launch the web page in Safari, or bring up a new web view controller that you've designed, along with various user interface controls. Exiting your app is rarely what users want, so unsurprisingly lots of app ended up creating mini-Safari experiences to browse inside their app.

As of iOS 9.0, Apple allows you to embed Safari right into your app, which means you get its great user interface, you get its access to stored user data, and you even get Reader Mode right out of the box. To get started, import the SafariServices framework into your view controller, like this:

```
import SafariServices
```

Now make your view controller conform to the **SFSafariViewControllerDelegate** protocol, like this:

```
class ViewController: UIViewController,
SFSafariViewControllerDelegate {
```

Now for the main piece of code, although I think you'll agree it's easy:

```
let urlString = "https://www.hackingwithswift.com"
```

```
if let url = NSURL(string: urlString) {

  let vc = SFSafariViewController(URL: url, entersReaderIfAvailable:
true)

  vc.delegate = self


  presentViewController(vc, animated: true, completion: nil)

}
```

That's all it takes to launch Safari inside your app now – cool, huh? We need to assign ourselves as the delegate of the Safari view controller because when the user taps "Done" inside Safari we need to dismiss it. Make sure you add this method to your view controller:

```
func safariViewControllerDidFinish(controller:
SFSafariViewController) {

    dismissViewControllerAnimated(true, completion: nil)

}
```

For more information see Hacking with Swift tutorial 32.

How to use light text color in the status bar

Availability: iOS 7.0 or later.

As of iOS 7.0, all view controllers set their own status bar style by default, which means they can have black text or white text depending on what looks best for your view controller. If you want to have light text in the status bar, add this code to your view controller:

```
override func preferredStatusBarStyle() -> UIStatusBarStyle {

  return .LightContent

}
```

If you want to change the status bar color dynamically, you should call **setNeedsStatusBarAppearanceUpdate()** on your view controller, which will force **preferredStatusBarStyle()** to be called again. Pro tip: you can put **setNeedsStatusBarAppearanceUpdate** inside an animation block to have the change animate.

What are size classes?

Availability: iOS 8.0 or later.

Size Classes are the iOS method of creating adaptable layouts that look great on all sizes and orientations of iPhone and iPad. For example, you might want to say that your UI looks mostly the same in portrait and landscape, but on landscape some extra information is visible. You could do this in code by checking for a change in the size of your view controller and trying to figure out what it means, but that's a huge waste of time – particularly now that iPad has multiple different sizes thanks to multitasking in iOS 9.

With Size Classes, you don't think about orientation or even device size. You care about whether you are running in a compact size or regular size, and iOS takes care of mapping that to various device sizes and orientations. iOS will also tell you when your size class changes so you can update your UI.

For example, an iPad app running full screen in portrait has regular horizontal and vertical size classes. In landscape, it also has regular horizontal and vertical size classes. If your app is used in iOS 9 multitasking, then its size class can be one of the following:

 • If the apps are running with an even split in landscape, both have compact horizontal and regular vertical size classes.
 • If the apps are running with an uneven split in landscape, the primary app has a regular horizontal class and the second has a compact horizontal size class. Both apps have regular vertical classes.
 • If the apps are running with an uneven split in portrait, both apps have compact horizontal size classes and regular vertical size classes.

Size Classes can be implemented in code if you want, but it's much easier to use Interface Builder. The key is to change only the bits you have to – try to share as much of your user interface as possible!

For more information see Hacking with Swift tutorial 31.

What are the different UIStackView distribution types?

Availability: iOS 9.0 or later.

One of the most compelling reasons to upgrade to iOS 9.0 is the new **UIStackView** class it introduced, which offers a simplified way of doing layouts in iOS. To give you more control over how it arranges their subviews, stack views offer five different distribution types for you to try, and here's what they do:

 • **Fill** makes one subview take up most of the space, while the others remain at their natural size. It decides which view to stretch by examining the content hugging priority for each of the subviews.
 • **Fill Equally** adjusts each subview so that it takes up equal amount of space in the stack view. All space will be used up.
 • **Fill Proportionally** is the most interesting, because it ensures subviews remain the same size relative to each other, but still stretches them to fit the available space. For example, if one view is 100 across and another is 200, and the stack view decides to stretch them to take up more space, the first view might stretch to 150 and the other to 300 – both going up by 50%.
 • **Equal Spacing** adjusts the spacing between subviews without resizing the subviews themselves.
 • **Equal Centering** attempts to ensure the centers of each subview are equally spaced, irrespective of how far the edge of each subview is positioned.

For more information see Hacking with Swift tutorial 31.

What is content compression resistance?

Availability: iOS 9.0 or later.

When Auto Layout has determined there isn't enough space to accomodate all your views at their natural size, it has to make a decision: one or more of those views needs to be squashed to make space for the others, but which one? That's where content compression resistance comes in: it's a value from 1 to 1000 that determines how happy you are for the view to be squashed if needed.

If you set a view's content compression resistance to be 1, it will be first in line to be squashed. If you set it to be 1000, it won't ever be squashed. The value is 750 by default, which means "I'd really prefer this not be squashed", but you might find you need to set it to be 751 or 749 on occasion, which means "I'd still really prefer this not to be squashed, but if there's no other choice…"

locationInView(): How to find a touch's location in a view
Availability: iOS 2.0 or later.

When the user starts touching an iOS screen, **touchesBegan()** is immediately called with a set of **UITouches**. If you want to find where the user touched, you need to take one of those touches then use **locationInView()** on it, like this:

```
override func touchesBegan(touches: Set<UITouch>, withEvent event:
UIEvent?) {

  if let touch = touches.first {

    let position = touch.locationInView(view)

    print(position)

  }

}
```

That will make **position** a **CGPoint** representing where the user touched in the current view. You can if you want pass a different view to **locationInView()**, and it will tell you where the touch was relative to that other view instead.

stringByEvaluatingJavaScriptFromString(): How to run JavaScript on a UIWebView
Availability: iOS 2.0 or later.

You can run custom JavaScript on a **UIWebView** using the method **stringByEvaluatingJavaScriptFromString()**. The method returns an optional string, which

means if the code returns a value you'll get it back otherwise you'll get back **nil**.

Here's an example that pulls out the current page's title:

```
let pageTitle =
webView.stringByEvaluatingJavaScriptFromString("document.title")
```

Note: if you're using a **WKWebView** you can use its **title** property directly to get the same thing.

WKWebView

How to enable back and forward swiping gestures in WKWebView
Availability: iOS 8.0 or later.

One of the many advantages of **WKWebView** over **UIWebView** is its ability to draw on some of the native user interface of Safari. It's a long way from the **SFSafariViewController** that was introduced in iOS 9.0, but you can enable the built-in gestures that let users go back and forward by swiping left and right.

Here's the code:

```
webView.allowsBackForwardNavigationGestures = true
```

For more information see Hacking with Swift tutorial 4.

How to monitor WKWebView page load progress using key-

value observing

Availability: iOS 8.0 or later.

iOS often uses a delegate system to report important changes, such as when a table view cell has been tapped or when a web page has finished loading. But the delegate system only goes so far, and if you want fine-grained detailed information sometimes you need to turn to KVO, or "key-value observing."

In the case of seeing how much of a page has loaded in **WKWebView**, KVO is exactly what you need: each web view has a property called **estimatedProgress**, and you can be asked to be notified when that value has changed.

First, create a progress view that will be used to show the loading progress:

```
progressView = UIProgressView(progressViewStyle: .Default)

progressView.sizeToFit()
```

You can place that anywhere you like. Now add the current view controller as an observer of the **estimatedProgress** property of your **WKWebView**, like this:

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

The **.New** in that line of code means "when the value changes, tell me the new value."

Finally, implement the **observeValueForKeyPath()** method in your view controller, updating the progress view with the estimated progress from the web view, like this:

```
override func observeValueForKeyPath(keyPath: String, ofObject
object: AnyObject, change: [NSObject: AnyObject], context:
UnsafeMutablePointer<Void>) {

    if keyPath == "estimatedProgress" {

        progressView.progress = Float(webView.estimatedProgress)
```

```
        }
    }
```

For more information see Hacking with Swift tutorial 4.

What's the difference between UIWebView and WKWebView?
Availability: iOS 8.0 or later.

The **UIWebView** class has been around since iOS 2.0 as a way to show HTML content inside your app, but iOS 8.0 introduced **WKWebView** as an alternative - what's the difference?

Well, there are several differences, but two are particularly important. First, **UIWebView** is part of UIKit, and thus is available to your apps as standard. You don't need to import anything – it's just there. This also means it's available inside Interface Builder, so you can drag and drop web view into your designs.

Second, **WKWebView** is run in a separate process to your app so that it can draw on native Safari JavaScript optimizations. This means **WKWebView** loads web pages faster and more efficiently than **UIWebView**, and also doesn't have as much memory overhead for you.

In iOS 8.0 **WKWebView** was unable to load local files, but this got fixed in iOS 9.0. The main reason to use **UIWebView** nowadays is for access to older features such as "Scale pages to fit" - this is not available in **WKWebView**.

For more information see Hacking with Swift tutorial 4.

evaluateJavaScript(): How to run JavaScript on a WKWebView
Availability: iOS 8.0 or later.

Using **evaluateJavaScript()** you can run any JavaScript in a **WKWebView** and read the result in Swift. This can be any JavaScript you want, which effectively means you can dig right into a page and pull out any kind of information you want.

Here's an example to get you started:

```
webView.evaluateJavaScript("document.getElementById('someElement').in
nerText") { (result, error) in

  if error != nil {

    print(result)

  }

}
```

For more information see Hacking with Swift tutorial 4.

Xcode

How to create exception breakpoints in Xcode
Availability: iOS 7.0 or later.

Exception breakpoints are a powerful debugging tool that remarkably few people know about, so please read the following carefully and put it into practice!

A regular breakpoint is on a line you specify, and causes the debugger to pause execution at that point so you can evaluate your program's state. An *exception* breakpoint tells the debugger to pause whenever a problem is encountered anywhere in your program, so you can evaluate your program's state before it crashes.

Exception breakpoints are trivial to set up: go to the Breakpoint Navigation (Cmd+7), then click the + button in the bottom left and choose Add Exception Breakpoint. You can leave it there if you want to, but it's preferable to make one further change to reduce unnecessary messages: right-click on your new breakpoint, choose Edit Breakpoint, then change the Exception value from "All" to "Objective-C".

For more information see Hacking with Swift tutorial 18.

How to debug view layouts in Xcode

Availability: iOS 8.0 or later.

View debugging lets you visualize exactly how your app is drawing to the screen by exploding your UI into 3D. So, if you're sure you added a button but you just can't see it, view debugging is for you: you can spin your interface around inside Xcode, and you'll probably find your button lurking behind another view because of a bug.

To activate view debugging, first you need to be using iOS 8.0, either on a device or the simulator. Run your app, and browse to the view controller you want to inspect. Now go to Xcode and look just below the main text editor, where the row of debugging buttons live: you want to click the button that has three rectangles in, just to the left of the location arrow.

When you use view debugging your app is paused, so make sure and tell Xcode to continue execution when you're done by pressing Cmd+Ctrl+Y.

For more information see Hacking with Swift tutorial 18.

How to load assets from Xcode asset catalogs

Availability: iOS 7 or later.

Xcode asset catalogs are a smart and efficient way to bring together your artwork in a single place. But they are also optimized for performance: when your app is built, your assets converted to an optimized binary format for faster loading, so they are recommended for all kinds of apps unless you have a specific reason to avoid them. (Note: SpriteKit games should texture atlases if possible.

If you don't already have an asset catalog in your project, you can create one by right-click on your project and choosing New File. From "iOS" choose "Resource" then Asset Catalog, then click Next and name your catalog. You can now select your new asset catalog in Xcode, and drag pictures directly into it.

Images stored inside asset catalog all retain their original filename, minus the path extension

part. For example, "taylor-swift.png" will just appear as "taylor-swift" inside your asset catalog, and that's how you should refer to it while loading too.

Asset catalogs automatically keep track of Retina and Retina HD images, but it's recommended that you name your images smartly to help make the process more smooth: taylor-swift.png, taylor-swift@2x.png and taylor-swift@3x.png are the best way to name your files for standard, Retina and Retina HD resolutions respectively.

For more information see Hacking with Swift tutorial 2.