HACKING WITH SWIFT

GUIDE BOOK

Tips, tricks, milestones, and exercises to help you learn faster

Paul Hudson

Hacking with Swift Guide Book

Paul Hudson

Version: 20190529

Contents

Introduction

Hello.

Hacking with Swift is a complete, standalone tutorial series that teaches Swift and iOS at the same time, using real-world apps and real-world problems. I wrote it to be the course I wish I had when I was learning: practical and hands-on, full of things to learn in every chapter, but also just enough repetition to ensure new concepts had time to sink in.

While Hacking with Swift is both complete and standalone, it is ultimately just a series of hand-picked iOS projects sorted by increasing difficulty. It *doesn't* make any attempt to help you monitor your progress, and so it's easy to rush through the projects, copy what I've written, but feel like you're getting nowhere – feel like you would be unable to produce your own, original apps.

That's where this guide book comes in: I've written it to act as a companion book to Hacking with Swift. It contains 11 milestones: one after the language introduction, then one every 3 projects until the end of project 30.

Each milestone contains a summary of what you've learned so far, along with a few refreshers on key parts of the code, and a challenge – a task you should try and do by yourself in order to help your new knowledge sink in. On occasion I provide example solutions to these challenges, but mostly they are entirely down to you.

So: go ahead and read the Hacking with Swift introduction chapter now, which gives you a primer on the Swift programming language. Once you're done, come back here and read the milestone.

Good luck! And remember: don't spend all your time sharpening your pencil when you should be drawing.

Milestone 1
Learning the Swift language

What you learned

Swift is technically a C-like language, but as you've seen it has a number of features that really make it stand out from the pack. Please don't worry if you read this chapter and thought, "you know, that just makes no sense" – that's OK. In fact, it's perfect normal. Some features, not least optionals, take a good long time to sink in, and you'll have lots more practice with them as the series progresses.

This was a big chapter with lots of ground covered, so your head might be spinning a bit. If you find something in the list below that you don't remember, just flick back and re-read. Don't worry, though: all these things will be covered again in future tutorials, so you'll have more than enough chance to learn!

- Xcode's playgrounds are a great way to test run your code and see immediate results. We don't use them much in Hacking with Swift because we're building real projects, but you'll find them invaluable in your own learning.
- Swift coders like to make things constant using **let** where possible, rather than variable using **var**. Xcode even warns you if you get this wrong.
- There are lots of types of data, such as strings ("Hello"), integers (5), doubles (5.5555), and booleans (true).
- You can tell Swift exactly what data type a value holds, but it's preferable to let Swift figure it out from your code.
- You can perform arithmetic using +, **-**, *****, and /, and compare with >, <, and ==.
- String interpolation lets you place values into strings. For example, **"Your name is \(name)"** will put the value of **name** directly into the string.
- Arrays let you group lots of values together, and are type safe – that means you can't add a number to a string array. You access items in an array by referring to their position, e.g. **names[4]**. These positions count from zero.
- Dictionaries also let you group values together, but you get to specify a key rather than just using numbers. For example, **person["month"]**.
- Conditional statements, e.g. **if person == "hater"**, let you take some action depending on the state of your program. You can combine multiple conditions together using **&&**.

- Swift has several types of loops. You saw **for i in 1...10**, which counts from 1 up to 10 inclusive, **for song in songs**, which loops every item item in the **songs** array, and also **while true**, which causes the loop to go around forever until you exit.
- You can use **continue** to stop the current run of the loop and continue from the next item – it effectively jumps back to the top of the loop and carries on.
- You can use **break** to exit a loop entirely.
- Using **switch/case** blocks is a good way to execute lots of different code depending on a value. You can use ranges, e.g. **2...3**, but you must always cover every possible value. You can provide a **default** case to mean "anything not already matched."
- Functions let you write chunks of re-usable code that you can call from elsewhere in your program. They can accept parameters, and can return values.
- Any data type in Swift can be optional, which means it might not have a value. For example, a variable holding someone's age would be 18 if that person was 18, or 0 if that person were only a few weeks old. But what if you didn't know their age? That's where optionality comes in – you can set the number to **nil**, meaning "no value yet." (This is the most confusing thing in Swift, and takes some getting used to.)
- Swift forces you to unwrap optionals before you use them, to avoid the chance of you accessing non-existent data. This is done using **if let**.
- Optionals have a special variety called implicitly unwrapped optionals, which can also have no value but don't need to be unwrapped before you use them. If you try this and get it wrong, your code will crash – be careful!
- Optional chaining lets you safely write complex instructions using optionals. For example **thingA?.thingB?.thingC = "meh"** will do nothing if either **thingA**, **thingB**, or **thingC** are **nil**.
- The nil coalescing operator, **??**, lets you provide a default value if an optional is empty.
- Enumerations – or just enums for short – let you declare a new data type that contains a set of specific values, such as **rainy**, **cloudy**, and **sunny**. Swift lets you add extra values to its enums, to track things like "how cloudy is it?"
- Structs and classes are complex data types: they hold multiple values inside them (called "properties"), and can also have internal functions (called "methods") that operate on those internal values.

- Classes are more powerful than structs because one class can inherit from (i.e. build upon) another. However, that adds extra complexity, which isn't always helpful.
- You can attach **willSet** and **didSet** observers to properties, which means you can ask for code to be run when that property changes.
- You can make attach access control to properties, which stops other people touching them directly. This is helpful because it forces other developers to use methods to read and write values, which means you can attach extra functionality as needed.
- If you know you're working with an object of type **A** and Swift *thinks* you're working with an object of type **B**, you can use **as?** typecasting to tell Swift what it really has. You can use **as!** instead, but again your app will crash if you're wrong.
- Closures are a bit like functions that you can place inside variables and pass around. They capture any values that they need to work.

Phew! That's a lot, but again: all those things will be covered again in more detail in upcoming tutorials – this was just a primer.

Key points

I hope you can recognize that Swift takes code safety *seriously*. This takes a number of forms, of which the most important ones are:

1. All your variables and constants must have a specific type, like **Int** or **String**. There is a special **Any** type, used when you want to mix data together, but even then you'll need to typecast values back to **Int** or whatever in order to use them.
2. If you create a function that says it will return a value, Swift will ensure it always does, no matter what route is taken through its code.
3. Optionals ensure you always work with real values. Swift won't let you touch optionals if they are nil, unless you specifically force override it – there's a reason **!** is sometimes called "the crash operator".
4. When you use a **switch/case** block, it must be exhaustive – it must cover all possible cases.

Let's take another look at some optional code. Consider this:

```
var a: Int? = 5
var b: Int? = 10
```

That creates two variables, **a** and **b**, both of which store optional integers. That means they might contain a number like 0, 10, 2309832, or they also be nil – they might have no value at all. How would you add those two numbers together?

Remember: Swift doesn't let you use optionals directly; you need to unwrap them safely first. This is because they can be nil, and "5 + nil" is undefined behavior that used to crash your app or produce weird behavior before Swift came along with its optionals.

In this case, we can unwrap **a** and **b** using **if let**. This creates new constants that are regular integers rather than optional, so we can work with them directly:

```
if let unwrappedA = a {
   if let unwrappedB = b {
        let c = unwrappedA + unwrappedB
    }
}
```

That **let c** line will only be reached if both **a** and **b** had a value, so Swift knows it's safe to run.

Swift lets you merge those two **if let** statements into one, like this:

```
if let unwrappedA = a, let unwrappedB = b {
    let c = unwrappedA + unwrappedB
}
```

While that's common practice, I prefer to avoid it in this book because it's a bit harder to read.

Another important take-away is that Swift is a *modern* programming language. It used many of the latest development techniques when it launched, as has since evolved further. This means we benefit from great features like optionals, closures, **switch/case** ranges, and more.

Finally, I hope you feel like you're able to go back to the playground now and start writing a little of your own code. Sure, there's no iOS code yet, but you can still have a go at attempting the challenge below.

Challenge

At this point you should be fairly comfortable with data types, conditions, loops, functions, and more, so your first challenge is to complete the Fizz Buzz test. This is a famous test commonly used to root out bad programmers during job interviews, and it goes like this:

- Write a function that accepts an integer as input and returns a string.
- If the integer is evenly divisible by 3 the function should return the string "Fizz".
- If the integer is evenly divisible by 5 the function should return "Buzz".
- If the integer is evenly divisible by 3 and 5 the function should return "Fizz Buzz"
- For all other numbers the function should just return the input number.

To solve this challenge you'll need to use quite a few skills you learned in this tutorial:

1. Write a function called **fizzbuzz()**. It should accept an **Int** parameter, and return a **String**.
2. You'll need to use **if** and **else if** conditions to check the input number.
3. You need use modulus, **%**, to check for even division. You'll also need to use **&&** to check for two things at once, because "Fizz Buzz" should only be printed if the input number is evenly divisible by 3 *and* 5.

Here are some test cases for you to use:

```
fizzbuzz(number: 3)
fizzbuzz(number: 5)
fizzbuzz(number: 15)
fizzbuzz(number: 16)
```

When your code is complete, that code should produce "Fizz", "Buzz", "Fizz Buzz" and "16".

Please try to code your answer now. I've written my own solution below to get you started, but it will help you internalize Swift better if you try writing your own code first. Remember: there is no learning without struggle, so you should just try – don't worry if you have a hard time or don't manage to finish.

If you're finding it hard, here are some hints. I suggest you read these only if you really need them, and even then only read one at a time – you should try to solve the challenge while reading as few hints as possible!

- Your function should start with **func fizzbuzz(number: Int) -> String {**. That is, it should accept an integer and return a string.
- You return values using the **return** keyword, e.g. **return "Fizz"**.
- Checking that a number is evenly divisible by 3 is done using **if number % 3 == 0**.
- Check that it's evenly divisible by 3 and 5 is done using **if number % 3 == 0 && number % 5 == 0**.

OK, that's it for hints. Below is my solution:

```
func fizzbuzz(number: Int) -> String {
   if number % 3 == 0 && number % 5 == 0 {
      return "Fizz Buzz"
   } else if number % 3 == 0 {
      return "Fizz"
   } else if number % 5 == 0 {
      return "Buzz"
   } else {
      return String(number)
   }
}
```

Milestone 2
Welcome to UIKit

What you learned

You’ve made your first two projects now, and completed a technique project too – this same cadence of app, game, technique is used all the way up to project 30, and you’ll start to settle into it as time goes by.

Both the app and the game were built with UIKit – something we’ll continue for two more milestones – so that you can really start to understand how the basics of view controllers work. These really are a fundamental part of any iOS app, so the more experience I can give you with them the better.

At this point you’re starting to put your Swift knowledge into practice with real context: actual, hands-on projects. Because most iOS apps are visual, that means you’ve started to use lots of things from UIKit, not least:

- Table views using **UITableView**. These are used everywhere in iOS, and are one of the most important components on the entire platform.
- Image views using **UIImageView**, as well as the data inside them, **UIImage**. Remember: a **UIImage** contains the pixels, but a **UIImageView** displays them – i.e., positions and sizes them. You also saw how iOS handles retina and retina HD screens using @2x and @3x filenames.
- Buttons using **UIButton**. These don’t have any special design in iOS by default – they just look like labels, really. But they do respond when tapped, so you can take some action.
- View controllers using **UIViewController**. These give you all the fundamental display technology required to show one screen, including things like rotation and multi-tasking on iPad.
- System alerts using **UIAlertController**. We used this to show a score in project 2, but it’s helpful for any time you need the user to confirm something or make a choice.
- Navigation bar buttons using **UIBarButtonItem**. We used the built-in action icon, but there are lots of others to choose from, and you can use your own custom text if you prefer.
- Colors using **UIColor**, which we used to outline the flags with a black border. There are lots of built-in system colors to choose from, but you can also create your own.
- Sharing content to Facebook and Twitter using **UIActivityViewController**. This same

class also handles printing, saving images to the photo library, and more.

One thing that might be confusing for you is the relationship between **CALayer** and **UIView**, and **CGColor** and **UIColor**. I've tried to describe them as being "lower level" and "higher level", which may or may not help. Put simply, you've seen how you can create apps by building on top of Apple's APIs, and that's exactly how Apple works too: their most advanced things like **UIView** are built on top of lower-level things like **CALayer**. Some times you need to reach down to these lower-level concept, but most of the time you'll stay in UIKit.

If you're concerned that you won't know when to use UIKit or when to use one of the lower-level alternatives, don't worry: if you try to use a **UIColor** when Xcode expects a **CGColor**, it will tell you!

Both projects 1 and 2 worked extensively in Interface Builder, which is a running theme in this series: although you *can* do things in code, it's generally preferable not to. Not only does this mean you can see exactly how your layout will look when previewed across various device types, but you also open the opportunity for specialized designers to come in and adjust your layouts without touching your code. Trust me on this: keeping your UI and code separate is A Good Thing.

Three important Interface Builder things you've met so far are:

1. Storyboards, edited using Interface Builder, but used in code too by setting storyboard identifiers.
2. Outlets and action from Interface Builder. Outlets connect views to named properties (e.g. **imageView**), and actions connect them to methods that get run, e.g. **buttonTapped()**.
3. Auto Layout to create rules for how elements of your interface should be positioned relative to each other.

You'll be using Interface Builder a *lot* throughout this series. Sometimes we'll make interfaces in code, but only when needed and always with good reason.

There are three other things I want to touch on briefly, because they are important.

First, you met the **Bundle** class, which lets you use any assets you build into your projects, like images and text files. We used that to get the list of NSSL JPEGs in project 1, but we'll use it again in future projects.

Second, loading those NSSL JPEGs was done by scanning the app bundle using the **FileManager** class, which lets you read and write to the iOS filesystem. We used it to scan directories, but it can also check if a file exists, delete things, copy things, and more.

Finally, you learned how to generate truly random numbers using Swift's **Int.random(in:)** method. Swift has lots of other functionality for randomness that we'll be looking at in future projects.

Key points

There are five important pieces of code that are important enough they warrant some revision. First, this line:

```
let items = try! fm.contentsOfDirectory(atPath: path)
```

The **fm** was a reference to **FileManager** and **path** was a reference to the resource path from **Bundle**, so that line pulled out an array of files at the directory where our app's resources lived. But do you remember why the **try!** was needed?

When you ask for the contents of a directory, it's possible – although hopefully unlikely! – that the directory doesn't actually exist. Maybe you meant to look in a directory called "files" but accidentally wrote "file". In this situation, the **contentsOfDirectory()** call will fail, and Swift will throw an exception – it will literally refuse to continue running your code until you handle the error.

This is important, because handling the error allows your app to behave well even when things go wrong. But in this case we got the path straight from iOS – we didn't type it in by hand, so if reading from our own app's bundle doesn't work then clearly something is very wrong indeed.

Swift requires any calls that might fail to be called using the **try** keyword, which forces you to add code to catch any errors that might result. However, because we know this code will work – it can't possibly be mistyped – we can use the **try!** keyword, which means "don't make me catch errors, because they won't happen." Of course, if you're wrong – if errors *do* happen – then your app will crash, so be careful!

The second piece of code I'd like to look at is this method:

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

That was used in project 1 to make the table view show as many rows as necessary for the **pictures** array, but it packs a lot into a small amount of space.

To recap, we used the Single View App template when creating project 1, which gave us a single **UIViewController** subclass called simply **ViewController**. To make it use a table instead, we changed **ViewController** so that it was based on **UITableViewController**, which provides default answers to lots of questions: how many sections are there? How many rows? What's in each row? What happens when a row is tapped? And so on.

Clearly we don't want the default answers to each of those questions, because our app needs to specify how many rows it wants based on its own data. And that's where the **override** keyword comes in: it means "I know there's a default answer to this question, but I want to provide a new one." The "question" in this case is "numberOfRowsInSection", which expects to receive an **Int** back: how many rows should there be?

The last two pieces of code I want to look at again are these:

```
let cell = tableView.dequeueReusableCell(withIdentifier:
"Picture", for: indexPath)

if let vc =
storyboard?.instantiateViewController(withIdentifier: "Detail")
as? DetailViewController {
}
```

Both of these are responsible for linking code to storyboard information using an identifier string. In the former case, it's a cell reuse identifier; in the latter, it's a view controller's storyboard identifier. You always need to use the same name in Interface Builder and your code – if you don't, you'll get a crash because iOS doesn't know what to do.

The second of those pieces of code is particularly interesting, because of the **if let** and **as? DetailViewController**. When we dequeued the table view cell, we used the built-in "Basic" style – we didn't need to use a custom class to work with it, so we could just crack on and set its text.

However, the detail view controller has its own custom thing we need to work with: the **selectedImage** string. That doesn't exist on a regular **UIViewController**, and that's what the **instantiateViewController()** method returns – it doesn't know (or care) what's inside the storyboard, after all. That's where the **if let** and **as?** typecast comes in: it means "I want to treat this is a **DetailViewController** so please try and convert it to one."

Only if that conversion works will the code inside the braces execute – and that's why we can access the **selectedImage** property safely.

Challenge

You have a rudimentary understanding of table views, image views, and navigation controllers, so let's put them together: your challenge is to create an app that lists various world flags in a table view. When one of them is tapped, slide in a detail view controller that contains an image view, showing the same flag full size. On the detail view controller, add an action button that lets the user share the flag picture and country name using **UIActivityViewController**.

To solve this challenge you'll need to draw on skills you learned in tutorials 1, 2, and 3:

1. Start with a Single View App template, then change its main **ViewController** class so that builds on **UITableViewController** instead.
2. Load the list of available flags from the app bundle. You can type them directly into the code if you want, but it's preferable not to.
3. Create a new Cocoa Touch Class responsible for the detail view controller, and give it properties for its image view and the image to load.
4. You'll also need to adjust your storyboard to include the detail view controller, including using Auto Layout to pin its image view correctly.
5. You will need to use **UIActivityViewController** to share your flag.

As always, I'm going to provide some hints below, but I suggest you try to complete as much of the challenge as you can before reading them.

Hints:

- To load the images from disk you need to use three lines of code: **let fm = FileManager.default**, then **let path = Bundle.main.resourcePath!**, then finally **let items = try! fm.contentsOfDirectory(atPath: path)**.
- Those lines end up giving you an array of all items in your app's bundle, but you only want the pictures, so you'll need to use something like the **hasSuffix()** method.
- Once you have made **ViewController** build on **UITableViewController**, you'll need to override its **numberOfRowsInSection** and **cellForRowAt** methods.

- You’ll need to assign a cell prototype identifier in Interface Builder, such as “Country”. You can then dequeue cells of that type using **tableView.dequeueReusableCell(withIdentifier: "Country", for: indexPath)**.
- The **didSelectItemAt** method is responsible for taking some action when the user taps a row.
- Make sure your detail view controller has a property for the image name to load, as well as the **UIImageView** to load it into. The former should be modified from **ViewController** inside **didSelectItemAt**; the latter should be modified in the **viewDidLoad()** method of your detail view controller.

Bonus tip: try setting the **imageView** property of the table view cell. Yes, they have one. And yes, it automatically places an image right there in the table view cell – it makes a great preview for every country.

Milestone 3
WebKit and Closures

What you learned

Project 4 showed how easy it is to build complex apps: Apple's WebKit framework contains a complete web browser you can embed into any app that needs HTML to be displayed. That might be a small snippet you've generated yourself, or it might be a complete website as seen in project 4.

After that, project 5 showed you how to build your second game, while also sneaking in a little more practice with **UITableViewController**. Starting from project 11 we'll be switching to SpriteKit for games, but there are lots of games you can make in UIKit too.

WebKit is the second external framework we've used, after the Social framework in project 3. These frameworks always deliver lots of complex functionality grouped together for one purpose, but you also learned lots of other things too:

- Delegation. We used this in project 4 so that WebKit events get sent to our **ViewController** class so that we can act on them.
- We used **UIAlertController** with its **.actionSheet** style to present the user with options to choose from. We gave it a **.cancel** button without a handler, which dismisses the options.
- You saw you can place **UIBarButtonItems** into the **toolbarItems** property, then have a **UIToolbar** shown by the navigation controller. We also used the **.flexibleSpace** button type to make the layout look better.
- You met Key-Value Observing, or KVO, which we used to update the loading progress in our web browser. This lets you monitor any property in all of iOS and be notified when it changes.
- You learned how to load text files from disk using **contentsOf**.
- We added a text field to **UIAlertController** for the first time, then read it back using **ac?.textFields?[0]**. We'll be doing this in several other projects in this series.
- You dipped your toes into the world of closures once again. These are complicated beasts when you're learning, but at this point in your Swift career just think of them as functions you can pass around in variables or as parameters to other functions.
- You worked with some methods for string and array manipulation: **contains()**, **remove(at:)**, **firstIndex(of:)**.

On top of that, we also took a deep-dive into the world of Auto Layout. We used this briefly in projects 1 and 2, but you've now learned more ways to organize your designs: Visual Format Language and anchors. There are other ways yet to come, and soon you'll start to find you prefer one method over another – and that's OK. I'm showing you them all so you can find what works best for you, and we all have our own preferences!

Key points

There are three pieces of code I'd like to revisit because they carry special significance.

First, let's consider the **WKWebView** from project 4. We added this property to the view controller:

```
var webView: WKWebView!
```

Then added this new **loadView()** method:

```
override func loadView() {
    webView = WKWebView()
    webView.navigationDelegate = self
    view = webView
}
```

The **loadView()** method is often not needed, because most view layouts are loaded from a storyboard. However, it's common to write part or all of your user interface in code, and for those times you're likely to want to replace **loadView()** with your own implementation.

If you wanted a more complex layout – perhaps if you wanted the web view to occupy only part of the screen – then this approach wouldn't have worked. Instead, you would need to load the normal storyboard view then use **viewDidLoad()** to place the web view where you wanted it.

As well as overriding the **loadView()** method, project 4 also had a **webView** property. This was important, because as far as Swift is concerned the regular **view** property is just a **UIView**.

Yes, *we* know it's actually a **WKWebView**, but Swift doesn't. So, when you want to call any methods on it like reload the page, Swift won't let you say **view.reload()** because as far as it's concerned **UIView** doesn't have a **reload()** method.

That's what the property solves: it's like a permanent typecast for the **view**, so that whenever we need to manipulate the web view we can use that property and Swift will let us.

The second interesting piece of code is this, taken from project 5:

```
if let startWords = try? String(contentsOf: startWordsURL) {
   allWords = startWords.components(separatedBy: "\n")
}
```

This combines **if let** and **try?** in the same expression, which is something you'll come across a lot. The **contentsOf** initializer for strings lets you load some text from disk. If it succeeds you'll get the text back, but if it fails Swift will complain loudly by throwing an exception.

You learned about **try**, **try!**, and **try?** some time ago, but I hope now you can see why it's helpful to have all three around. What **try?** does is say, "instead of throwing an exception, just return nil if the operation failed." And so, rather than **contentsOf** returning a **String** it will actually return a **String?** – it might be some text, or it might be nil. That's where **if let** comes in: it checks the return value from **contentsOf** and, if it finds valid text, executes the code inside the braces.

The last piece of code I'd like to review is this:

```
view.addConstraints(NSLayoutConstraint.constraints(withVisualFo
rmat:"V:|[label1(labelHeight)]-[label2(labelHeight)]-
[label3(labelHeight)]-[label4(labelHeight)]-
[label5(labelHeight)]->=10-|", options: [], metrics: metrics,
views: viewsDictionary))
```

I think that – in just one line of code – demonstrates the advantages of using Visual Format Language: it lines up five labels, one above the other, each with equal height, with a small amount of space between them, and 10 or more points of space at the end.

That line also demonstrates the disadvantage of Visual Format Language: it has a tendency to look like line noise! You need to read it very carefully, sometimes jumping back and forward, in order to unpick what it's trying to do. VFL is the quickest and easiest way to solve many

Auto Layout problems in an expressive way, but as you progress through this course you'll learn alternatives such as **UIStackView** that can do the same thing without the complex syntax.

Challenge

It's time to put your skills to the test by making your own complete app from scratch. This time your job is to create an app that lets people create a shopping list by adding items to a table view.

The best way to tackle this app is to think about how you build project 5: it was a table view that showed items from an array, and we used a **UIAlertController** with a text field to let users enter free text that got appended to the array. That forms the foundation of this app, except this time you don't need to validate items that get added – if users enter some text, assume it's a real product and add it to their list.

For bonus points, add a left bar button item that clears the shopping list – what method should be used afterwards to make the table view reload all its data?

Here are some hints in case you hit problems:

- Remember to change **ViewController** to build on **UITableViewController**, then change the storyboard to match.
- Create a **shoppingList** property of type **[String]** to hold all the items the user wants to buy.
- Create your **UIAlertController** with the style **.alert**, then call **addTextField()** to let the user enter text.
- When you have a new shopping list item, make sure you **insert()** it into your **shoppingList** array before you call the **insertRows(at:)** method of your table view – your app will crash if you do this the wrong way around.

You might be tempted to try to use **UIActivityViewController** to share the finished shopping list by email, but if you do that you'll hit a problem: you have an array of strings, not a single string.

There's a special method that can create one string from an array, by stitching each part together using a separator you provide. I'll be going into it in project 8, but if you're keen to try it now here's some code to get you started:

```
let list = shoppingList.joined(separator: "\n")
```

That will create a new **list** constant that is a regular string, with each shopping list item separated by “\n” – that’s Swift’s way of representing a new line.

Milestone 4
JSON and GCD

What you learned

Projects 7, 8, and 9 were the first in the series I consider to be "hard": you had to parse JSON data, you had to create a complex layout for 7 Swifty Words, and you took your first steps towards creating multithreaded code – code that has iOS do more than one thing at a time.

None of those things were easy, but I hope you felt the results were worth the effort. And, as always, don't worry if you're not 100% on them just – we'll be using **Codable** and GCD more in future projects, so you'll have ample chance to practice.

- You've now met **UITabBarController**, which is another core iOS component – you see it in the App Store, Music, iBooks, Health, Activity, and more.
- Each item on the tab bar is represented by a **UITabBarItem** that has a title and icon. If you want to use one of Apple's icons, it means using Apple's titles too.
- We used **Data** to load a URL, with its **contentsOf** method. That then got fed to **JSONDecoder** so that we could read it in code.
- We used **WKWebView** again, this time to show the petition content in the app. This time, though, we wanted to load our own HTML rather than a web site, so we used the **loadHTMLString()** method.
- Rather than connect lots of actions in Interface Builder, you saw how you could write user interfaces in code. This was particularly helpful for the letter buttons of 7 Swifty Words, because we could use a nested loop.
- In project 7 we used property observers, using **didSet**. This meant that whenever the **score** property changed, we automatically updated the **scoreLabel** to reflect the new score.
- You learned how to execute code on the main thread and on background threads using **DispatchQueue**, and also met the **performSelector(inBackground:)** method, which is the easiest way to run one whole method on a background thread.
- Finally, you learned several new methods, not least **enumerated()** for looping through arrays, **joined()** for bringing an array into a single value, and **replacingOccurrences()** to change text inside a string.

Key points

There are three Swift features that are so important – and so commonly used – that they are worth revising to make sure you're comfortable with them.

The first piece of code I'd like to look at is this one:

```
for (index, line) in lines.enumerated() {
    let parts = line.components(separatedBy: ": ")
```

This might seem like a new way of writing a loop, but really it's just a variation of the basic **for** loop. A regular **for** loop returns one value at a time from an array for you to work with, but this time we're calling the **enumerated()** method on the array, which causes it to return *two* things for each item in the array: the item's position in the array, as well as the item itself.

It's common to see **enumerated()** in a loop, because its behavior of delivering both the item and its position is so useful. For example, we could use it to print out the results of a race like this:

```
let results = ["Paul", "Sophie", "Lottie", "Andrew", "John"]

for (place, result) in results.enumerated() {
    print("\(place + 1). \(result)")
}
```

Note that I used **\(place + 1)** to print out each person's place in the results, because array positions all count from 0.

The second piece of code we're going to review is this:

```
var score: Int = 0 {
    didSet {
        scoreLabel.text = "Score: \(score)"
    }
}
```

This is a property observer, and it means that whenever the **score** integer changes the label will be updated to match. There are other ways you could ensure the two remain in sync: we could have written a **setScore()** method, for example, or we could just have updated the **scoreLabel** text by hand whenever the **score** property changed.

The former isn't a bad idea, but you do need to police yourself to ensure you never set **score** directly - and that's harder than you think! The second *is* a bad idea, however: duplicating code can be problematic, because if you need to change something later you need to remember to update it everywhere it's been duplicated.

The final piece of code I'd like to look at again is this:

```
DispatchQueue.global().async { [weak self] in
   // do background work

   DispatchQueue.main.async {
      // do main thread work
   }
}
```

That code uses Grand Central Dispatch to perform some work in the background, then perform some more work on the main thread. This is *extremely* common, and you'll see this same code appear in many projects as your skills advance.

The first part of the code tells GCD to do the following work on a background thread. This is useful for any work that will take more than a few milliseconds to execute, so that's anything to do with the internet, for example, but also any time you want to do complex operations such as querying a database or loading files.

The second part of the code runs after your background work has completed, and pushes the remaining work back to the main thread. This is where you present the user with the results of your work: the database results that matched their search, the remote file you fetched, and so on.

It is *extremely* important that you only ever update your user interface from the main thread – trying to do it from a background thread will cause your app to crash in the best case, or – much worse – cause weird inconsistencies in your app.

Challenge

This is the first challenge that involves you creating a game. You'll still be using UIKit, though, so it's a good chance to practice your app skills too.

The challenge is this: make a hangman game using UIKit. As a reminder, this means choosing a random word from a list of possibilities, but presenting it to the user as a series of underscores. So, if your word was "RHYTHM" the user would see "??????".

The user can then guess letters one at a time: if they guess a letter that it's in the word, e.g. H, it gets revealed to make "?H??H?"; if they guess an incorrect letter, they inch closer to death. If they seven incorrect answers they lose, but if they manage to spell the full word before that they win.

That's the game: can you make it? Don't underestimate this one: it's called a challenge for a reason – it's supposed to stretch you!

The main complexity you'll come across is that Swift has a special data type for individual letters, called **Character**. It's easy to create strings from characters and vice versa, but you do need to know how it's done.

First, the individual letters of a string are accessible simply by treating the string like an array – it's a bit like an array of **Character** objects that you can loop over, or read its **count** property, just like regular arrays.

When you write **for letter in word**, the **letter** constant will be of type **Character**, so if your **usedLetters** array contains strings you will need to convert that letter into a string, like this:

```
let strLetter = String(letter)
```

Note: unlike regular arrays, you can't read letters in strings just by using their integer positions – they store each letter in a complicated way that prohibits this behavior.

Once you have the string form of each letter, you can use **contains()** to check whether it's inside your **usedLetters** array.

That's enough for you to get going on this challenge by yourself. As per usual there are some hints below, but it's always a good idea to try it yourself before reading them.

- You already know how to load a list of words from disk and choose one, because that's exactly what we did in tutorial 5.
- You know how to prompt the user for text input, again because it was in tutorial 5. Obviously this time you should only accept single letters rather than whole words – use **someString.count** for that.
- You can display the user's current word and score using the **title** property of your view controller.
- You should create a **usedLetters** array as well as a **wrongAnswers** integer.
- When the player wins or loses, use **UIAlertController** to show an alert with a message.

Still stuck? Here's some example code you might find useful:

```
let word = "RHYTHM"
var usedLetters = ["R", "T"]
var promptWord = ""

for letter in word.characters {
   let strLetter = String(letter)

   if usedLetters.contains(strLetter) {
      promptWord += strLetter
   } else {
      promptWord += "?"
   }
}

print(promptWord)
```

Milestone 5

Collection Views and SpriteKit

What you learned

You probably haven't realized it yet, but the projects you just completed were some of the most important in the series. The end results are nice enough, but what they *teach* is more important – you made your first class completely from scratch, which is how you'll tackle many, many problems in the future:

- You met **UICollectionView** and saw how similar it is to **UITableView**. Sure, it displays things in columns as well as rows, but the method names are so similar I hope you felt comfortable using it.
- You also designed a custom **UICollectionViewCell**, first in the storyboard and then in code. Table views come with several useful cell types built in, but collection views don't so you'll always need to design your own.
- We used **UIImagePickerController** for the first time in this project. It's not that easy to use, particularly in the way it returns its selected to you, but we'll be coming back to it again in the future so you'll have more chance to practice.
- The **UUID** data type is used to generate universally unique identifiers, which is the easiest way to generate filenames that are guaranteed to be unique. We used them to save images in project 10 by converting each **UIImage** to a **Data** that could be written to disk, using **jpegData()**.
- You met Apple's **appendingPathComponent()** method, as well as my own **getDocumentsDirectory()** helper method. Combined, these two let us create filenames that are saved to the user's documents directory. You could, in theory, create the filename by hand, but using **appendingPathComponent()** is safer because it means your code won't break if things change in the future.

Project 11 was the first game we've made using SpriteKit, which is Apple's high-performance 2D games framework. It introduced a huge range of new things:

- The **SKSpriteNode** class is responsible for loading and drawing images on the screen.
- You can draw sprites using a selection of blend modes. We used **.replace** for drawing the background image, which causes SpriteKit to ignore transparency. This is faster, and

perfect for our solid background image.

- You add physics to sprite nodes using **SKPhysicsBody**, which has **rectangleOf** and **circleWithRadius** initializers that create different shapes of physics bodies.
- Adding actions to things, such as spinning around or removing from the game, is done using **SKAction**.
- Angles are usually measured in radians using **CGFloat**.

Lastly, you also met **UserDefaults**, which lets you read and write user preferences, and gets backed up to iCloud automatically. You shouldn't abuse it, though: try to store only the absolute essentials in **UserDefaults** to avoid causing performance issues. In fact, tvOS limits you to just 500KB of **UserDefaults** storage, so you have no choice!

For the times when your storage needs are bigger, you should either use the **Codable** protocol to convert your objects to JSON, or use **NSKeyedArchiver** and **NSKeyedUnarchiver**. These convert your custom data types into a **Data** object that you can read and write to disk.

Key points

There are three pieces of code worth looking at again before you continue on to project 13.

First, lets review the initializer for our custom **Person** class from project 10:

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

There are two interesting things in that code. First, we need to use **self.name = name** to make our intention clear: the class had a property called **name**, and the method had a parameter called **name**, so if we had just written **name = name** Swift wouldn't know what we meant.

Second, notice that we wrote **init** rather than **func init**. This is because **init()**, although it looks like a regular method, is special: technically it's an *initializer* rather than a *method*. Initializers are things that create objects, rather than being methods you call on them later on.

The second piece of code I'd like to review is this, taken from project 11:

```
if let touch = touches.first {
   let location = touch.location(in: self)
   let box = SKSpriteNode(color: UIColor.red, size:
CGSize(width: 64, height: 64))
   box.position = location
   addChild(box)
}
```

We placed that inside the **touchesBegan()** method, which is triggered when the user touches the screen. This method gets passed a set of touches that represent the user's fingers on the screen, but in the game we don't really care about multi-touch support we just say **touches.first**.

Now, obviously the **touchesBegan()** method only gets triggered when a touch has actually started, but **touches.first** is still optional. This is because the set of touches that gets passed in doesn't have any special way of saying "I contain at least one thing". So, even though we know there's going to be at least one touch in there, we still need to unwrap the optional. This is one of those times when the force unwrap operator would be justified:

```
let touch = touches.first!
```

The last piece of code I'd like to review in this milestone is from project 12, where we had these three lines:

```
let defaults = UserDefaults.standard
defaults.set(25, forKey: "Age")
let array = defaults.object(forKey:"SavedArray") as?
[String] ?? [String]()
```

The first one gets access to the app's built-in **UserDefaults** store. If you were wondering, there are alternatives: it's possible to request access to a shared **UserDefaults** that more than one app can read and write to, which is helpful if you ship multiple apps with related functionality.

The second line writes the integer 25 next to the key "Age". **UserDefaults** is a key-value store, like a dictionary, which means every value must be read and written using a key name like "Age". In my own code, I always use the same name for my **UserDefaults** keys as I do for my properties, which makes your code easier to maintain.

The third line is the most interesting: it retrieves the object at the key "SavedArray" then tries to typecast it as a string array. If it succeeds – if an object was found at "SavedArray" and if it could be converted to a string array – then it gets assigned to the **array** constant. But if either of those fail, then the nil coalescing operator (the **??** part) ensures that **array** gets set to an empty string array.

Challenge

Your challenge is to put two different projects into one: I'd like you to let users take photos of things that interest them, add captions to them, then show those photos in a table view. Tapping the caption should show the picture in a new view controller, like we did with project 1. So, your finished project needs to use elements from both project 1 and project 12, which should give you ample chance to practice.

This will require you to use the **picker.sourceType = .camera** setting for your image picker controller, create a custom type that stores a filename and a caption, then show the list of saved pictures in a table view. **Remember:** using the camera is only possible on a physical device.

It might sound counter-intuitive, but trust me: one of the best ways to learn things deeply is to learn them, forget them, then learn them again. So, don't be worried if there are some things you don't recall straight away: straining your brain for them, or perhaps re-reading an older chapter just briefly, is a great way to help your iOS knowledge sink in a bit more.

Here are some hints in case you hit problems:

- You'll need to make **ViewController** build on **UITableViewController** rather than just **UIViewController**.
- Just like in project 10, you should create a custom type that stores an image filename and a caption string, then use either **Codable** or **NSCoding** to load and save that.
- Use a **UIAlertController** to get the user's caption for their image – a single text field is enough.
- You'll need to design your detail view controller using Interface Builder, then call **instantiateViewController** to load it when a table view row is tapped.

Milestone 6

Core Image and Core Animation

What you learned

Project 13 was a trivial application if you look solely at the amount of code we had to write, but it's remarkable behind the scenes thanks to the power of Core Images. Modern iPhones have extraordinary CPU and GPU hardware, and Core Image uses them both to the full so that advanced image transformations can happen in real time – if you didn't try project 13 on a real device, you really ought to if only to marvel at how incredibly fast it is!

You also met UIKit's animation framework for the first time, which is a wrapper on top of another of Apple's cornerstone frameworks: Core Animation. This is a particularly useful framework to get familiar with because of its simplicity: you tell it what you want ("move this view to position X/Y") then tell it a duration ("move it over three seconds"), and Core Animation figures out what each individual frame looks like.

Here are just some of the other things you've now learned:

- How to let the user select from a range of values using **UISlider**.
- How to move, rotate, and scale views using **CGAffineTransform**.
- Saving images back to the user's photo library by calling the **UIImageWriteToSavedPhotosAlbum()** function.
- Create Core Image contexts using **CIContext**, and creating then applying Core Image filters using **CIFilter**.
- Changing sprite images without moving the sprite, using **SKTexture**.
- Cropping a sprite so that only part of it is visible, using **SKCropNode**.
- More **SKActions**, including **moveBy(x:y:)**, **wait(forDuration:)**, **sequence()**, and even how to run custom closures using **run(block:)**.
- The **asyncAfter()** method of GCD, which causes code to be run after a delay.
- Plus you had more practice using **UIImagePickerController**, to select pictures from the user's photo library. We'll be using this again – it's a really helpful component to have in your arsenal!

Key points

There are three pieces of code I'd like to revisit briefly, just to make sure you understand them fully.

First, I want to look more closely at how closure capturing works with **asyncAfter()**. Here's some code as an example:

```
DispatchQueue.main.asyncAfter(deadline: .now() + 1) { [unowned
self] in
    self.doStuff()
}
```

The call to **asyncAfter()** needs two parameters, but we pass the second one in using trailing closure syntax because it's clearer. The first parameter is specified as a **DispatchTime** value, which is the exact time to execute the code. When we specify **.now()**, Swift is smart enough to realize that means **DispatchTime.now()**, which is the current time. It then lets us add 1 second to it, so that the finished deadline ends up being one second away from now.

Then there's the **[unowned self]**. This is called a capture list, and it gives Swift instructions how to handle values that get used inside the closure. In this case, **self.doStuff()** references **self**, so the capture list **[unowned self]** means "I know you want to capture **self** strongly so that it can be used later, but I want you not to have any ownership at all."

If the closure runs and **self** has become nil for some reason, the call to **self.doStuff()** will crash: we told Swift not to worry about ownership, then accidentally let **self** get destroyed, so the crash is our own fault. As an alternative, we could have written **[weak self]**, which would capture **self** in the closure as an optional. With that change, you'd need to run this code instead:

```
self?.doStuff()
```

Closure capturing can be a complicated topic. In this case, **[unowned self]** isn't really even needed because there's no chance of a reference cycle: **self** doesn't own **DispatchQueue.main**, so the reference will be destroyed once the closure finishes. However, there's no *harm* adding **[weak self]**, which is why I often include it.

The second piece of code I'd like to review is this, taken from project 15:

```
UIView.animate(withDuration: 1, delay: 0, options: [],
animations: {
   switch self.currentAnimation {
   case 0:
      break

   default:
      break
   }
}) { finished in
   sender.isHidden = false
}
```

At this point in your Swift career you've seen several functions that accept closures, but this one takes *two*: one is a set of animations to perform, and one is a set of actions to run when the animations have completed. I didn't include closure capturing here because it isn't needed – the animation closure will be used once then destroyed.

I don't want to sound like a broken record, but: if you use **[weak self]** when it isn't needed, nothing happens. But if you *don't* use it and it *was* needed, bad things will happen - at the very least you'll leak memory.

The final piece of code to review is this, also taken from project 15:

```
self.imageView.transform = CGAffineTransform(rotationAngle:
CGFloat.pi)
self.imageView.transform = CGAffineTransform.identity
```

There are a few things in there that I'd like to recap just briefly:

1. The **self** is required when accessing **imageView**, because we're inside a closure and Swift wants us to explicitly acknowledge we recognize **self** will be captured.
2. The **CGFloat.pi** constant is equal to 180 degrees, and it also has equivalents in **Float.pi** and **Double.pi** – this is just to save you having to typecast values.
3. The identity matrix, **CGAffineTransform.identity**, has no transformations: it's not scaled, moved, or rotated, and it's useful for resetting a transform.

Challenge

Your challenge is to make an app that contains facts about countries: show a list of country names in a table view, then when one is tapped bring in a new screen that contains its capital city, size, population, currency, and any other facts that interest you. The type of facts you include is down to you – Wikipedia has a huge selection to choose from.

To make this app, I would recommend you blend parts of project 1 project 7. That means showing the country names in a table view, then showing the detailed information in a second table view.

How you load data into the app is going to be an interesting problem for you to solve. I suggested project 7 above because a sensible approach would be to create a JSON file with your facts in, then load that in using **contentsOf** and parse it using **Codable**. Regardless of how you end up solving this, I suggest you *don't* just hard-code it into the app – i.e., typing all the facts manually into your Swift code. You're better than that!

Go ahead and try coding it now. If you hit problems, here are some hints:

- You should create a custom **Country** struct that has properties for each of the facts you have in your JSON file. You can then have a **[Country]** array in your view controller.
- When using a table view in your detail view controller, try setting the **numberOfLines** property of the cell's text label to be 0. That ought to allow the cell to fill up to two lines of text by default.
- Don't forget all the little UI touches: adding a disclosure indicator to the countries table, adding titles to the navigation controller, and so on. You could even add an action button to the detail view that shares a fact about the selected country.

Milestone 7
Extensions and Debugging

What you learned

Although these projects shouldn't have been too difficult for someone at your level, they definitely covered some important skills that you'll come back to time and time again.

You also completed the technique project all about debugging. Although I covered the main techniques in that chapter, debugging isn't really something that can be *taught* – it's a skill you acquire over time with experience. As you progress, you'll learn to recognize certain kinds of errors as ones you've solved before, so over time you get faster and spotting - and fixing – mistakes.

Some other things you learned in this milestone are:

- MapKit, Apple's incredible, free map framework that lets you add maps, satellite maps, route directions, and more.
- How to use **Timer** to trigger a repeating method every few seconds. Remember to call **invalidate()** on your time when you want it to stop!
- In SpriteKit, you learned to use the **advanceSimulationTime()** method to force a particle system to move forward a set number of seconds so that it looks like it has existed for a while.
- We used the **SKPhysicsBody(texture:)** initializer to get pixel-perfect physics. This is computationally expensive, so you should use it only when strictly needed.
- We used the **linearDamping** and **angularDamping** properties of **SKPhysicsBody** to force sprites to retain their speed and spin, rather than slowing down over time – project 23 was set in outer space, after all!
- You learned how the **assert()** function lets you ensure your app's state is exactly what you thought it was.
- You set your first breakpoints, and even attached a condition to it. Trust me on this: you will use breakpoints thousands – if not hundreds of thousands! - of times in your Swift career, so this is a great skill to have under your belt.

Key points

There are two pieces of code I'd like to review before you continue, because both are hugely unappreciated.

First, the **assert()** function. If you remember, you place calls to **assert()** in your code whenever you want to say, "I believe X must be the case." What X is depends on you: "this array will contain 10 items," or "the user must be logged in," or "the in-app purchase content was unlocked" are all good examples.

When the **assert()** line hits, iOS validates that your statement is true. If the array *doesn't* contain 10 items, or the in-app purchase *wasn't* unlocked, the assertion will fail and your app will crash immediately. Xcode will print the location of the crash – e.g. ViewController.swift line 492 – along with any custom message you attached, such as "The in-app purchase failed to unlock content."

Obviously crashing your app sounds bad on the surface, but remember: assertions are automatically removed when you build your app in release mode – i.e., for the App Store. This means not only will your app not crash because an assertion failed, but also that those assertions aren't even checked in the first place – Xcode won't even run them. This means you can – and should! – add assertions liberally throughout your code, because they have zero performance impact on your finished, shipping app.

Think of assertions as "hard-core mode" for your app: if your app runs perfectly even with assertions in every method, it will *definitely* work when users get hold of it. And if your assertions *do* make your app crash while in development, that's perfect: it means your app's state wasn't what you thought, so either your logic is wrong or your assertion was.

The second piece of code to review is the **Timer** class, which we used like this:

```
gameTimer = Timer.scheduledTimer(timeInterval: 0.35, target:
self, selector: #selector(createEnemy), userInfo: nil, repeats:
true)
```

Timers are rudimentary things, but perfectly fit many games – that code will cause the **createEnemy()** method to be called approximately every 0.35 seconds until it's cancelled.

In this case, the **gameTimer** value was actually a property that belong to the game scene. Cunningly, **Timer** maintains a strong reference to its target – i.e., it stores it and won't let it be destroyed while the timer is still active – which means this forms a strong reference cycle. That is, the game scene owns the timer, and the timer owns the game scene, which means neither of them will be ever destroyed unless you manually call **invalidate()**.

In project 17 this isn't a problem, but we'll be returning to this theme in project 30 when it *is* a problem – watch out for it!

By the way: the **Timer** class really does offer only *approximate* accuracy, meaning that the **createEnemy()** method will be called roughly every 0.35 seconds, but it might take 0.4 seconds one time or 0.5 seconds another. In the context of Space Race this isn't a problem, but remember: iOS wants you to draw at 60 or 120 frames per second, which gives you between 8 and 16 milliseconds to do all your calculation and rendering – a small delay from **Timer** might cause problems in more advanced games.

The iOS solution to this is called **CADisplayLink**, and it causes a method of yours to be called every time drawing has just finished, ensuring you always get the maximum allotment of time for your calculations. This isn't covered in Hacking with Swift, but you'll find an explanation and code example in this article in my Swift Knowledge Base: **How to synchronize code to drawing using CADisplayLink**.

Challenge

It's time to put your skills to the test and make your own app, starting from a blank canvas. This time your challenge is to make a shooting gallery game using SpriteKit: create three rows on the screen, then have targets slide across from one side to the other. If the user taps a target, make it fade out and award them points.

How you implement this game really depends on what kind of shooting gallery games you've played in the past, but here are some suggestions to get you started:

- Make some targets big and slow, and others small and fast. The small targets should be worth more points.
- Add "bad" targets – things that *cost* the user points if they get shot accidentally.
- Make the top and bottom rows move left to right, but the middle row move right to left.
- Add a timer that ticks down from 60 seconds. When it hits zero, show a Game Over message.
- Try going to **https://openclipart.org/** to see what free artwork you can find.
- Give the user six bullets per clip. Make them tap a different part of the screen to reload.

Those are just suggestions – it's your game, so do what you like!

Tip: I made a SpriteKit shooting gallery game in my book **Hacking with macOS** – the SpriteKit **code for that project** is compatible with iOS, but rather than just reading my code you might prefer to just take **my assets** and use them to build your own project.

As always, please try to code the challenge yourself before reading any of the hints below.

- Moving the targets in your shooting gallery is a perfect job for the **moveBy()** action. Use a sequence so that targets move across the screen smoothly, then remove themselves when they are off screen.
- You can create a timer using an **SKLabelNode**, a **secondsRemaining** integer, and a **Timer** that takes 1 away from **secondsRemaining** every 1 second.
- Make sure you call **invalidate()** when the time runs out.

- Use **nodes(at:)** to see what was tapped. If you don't find a node named "Target" in the returned array, you could subtract points for the player missing a shot.
- You should be able to use a property observer for both player score and number of bullets remaining in clip. Changing the score or bullets would update the appropriate **SKLabelNode** on the screen.

Milestone 8
Maps and Notifications

What you learned

These three projects were a mixed bag in terms of difficulty: although Safari extensions are clearly a bit of a wart in Apple's APIs, it's still marvelous to be able to add features directly to one of the most important features in iOS. As for the Fireworks Night project, I hope it showed you it doesn't take much in the way of graphics to make something fun!

You also learned about local notifications, which might seem trivial at first but actually open up a huge range of possibilities for your apps because you can prompt users to take action even when your app isn't running.

The best example of this is the Duolingo app – it sets "You should practice your language!" reminders for 1 day, 2 days, and 3 days after the app was most recently launched. If you launch the app before the reminders appear, they just clear them and reset the timer so you never notice them.

Here's a quick reminder of the things we covered:

- How to make extensions for Safari by connecting Swift code to JavaScript. Getting the connection working isn't too easy, but once it's set up you can send whatever you want between the two.
- Editing multi-line text using **UITextView**. This is used by apps like Mail, Messages, and Notes, so you'll definitely use it in your own apps.
- You met Objective-C's **NSDictionary** type. It's not used much in Swift because you lose Swift's strong typing, but it's occasionally unavoidable.
- We used the iOS **NotificationCenter** center class to receive system messages. Specifically, we requested that a method be called when the keyboard was shown or hidden so that we can adjust the insets of our text view. We'll be using this again in a later project, so you have ample chance for practice.
- The **follow()** SKAction, which causes a node to follow a bezier path that you specify. Use **orientToPath: true** to make the sprite rotate as it follows.
- The **color** and **colorBlendFactor** properties for **SKSpriteNode**, which let you dynamically recolor your sprite.

- The **motionBegan()** method, which gets called on your view controllers when the user shakes their device.
- Swift's **for case let** syntax for adding a condition to a loop.
- The UserNotifications framework, which allows you to create notifications and attach them to triggers.

Key points

There are three pieces of code I'd like to review, just to make sure you understand them fully.

The first thing I'd like to recap is **NotificationCenter**, which is a system-wide broadcasting framework that lets you send and receive messages. These messages come in two forms: messages that come from iOS, and messages you send yourself. Regardless of whether the messages come from, **NotificationCenter** is a good example of *loose coupling* – you don't care who subscribes to receive your messages, or indeed if anyone at all does; you're just responsible for posting them.

In project 19 we used **NotificationCenter** so that iOS notified us when the keyboard was shown or hidden. This meant registering for the **Notification.Name.UIKeyboardWillChangeFrame** and **Notification.Name.UIKeyboardWillHide**: we told iOS we want to be notified when those events occurred, and asked it to execute our **adjustForKeyboard()** method. Here's the code we used:

```
let notificationCenter = NotificationCenter.default
notificationCenter.addObserver(self, selector:
#selector(adjustForKeyboard), name:
Notification.Name.UIKeyboardWillHide, object: nil)
```

There are lots of these events – just try typing **Notification.Name.** and letting autocomplete show you what's available. For example, in project 28 we use the **Notification.Name.UIApplicationWillResignActive** event to detect when the app moves to the background.

Like I said, it's also possible to send your own notifications using **NotificationCenter**. Their names are just strings, and only your application ever sees them, so you can go ahead and make as many as you like. For example, to post a "UserLoggedIn" notification, you would write this:

```
let notificationCenter = NotificationCenter.default
notificationCenter.post(name:
Notification.Name("UserLoggedIn"), object: nil)
```

If no other part of your app has subscribed to receive that notification, nothing will happen. But you can make any other objects subscribe to that notification – it could be one thing, or ten things, it doesn't matter. This is the essence of loose coupling: you're transmitting the event to everyone, with no direct knowledge of who your receivers are.

The second piece of code I'd like to review is this, taken from project 21:

```
let center = UNUserNotificationCenter.current()

center.requestAuthorization(options: [.alert, .badge, .sound])
{ (granted, error) in
   if granted {
      print("Yay!")
   } else {
      print("D'oh")
   }
}
```

In that code, everything from **{ (granted, error) in** to the end is a closure: that code *won't* get run straight away. Instead, it gets passed as the second parameter to the **requestAuthorization()** method, which stores the code. This is important – in fact essential – to the working of this code, because iOS needs to ask the user for permission to show notifications.

iPhones can do literally billions of things every second, so in the time it takes for the "Do you want to allow notifications" message to appear, then for the user to read it, consider it, then make a choice, the iPhone CPU has done countless other things.

It would be a pretty poor experience if your app had to pause completely while the user was thinking, which is why closures are used: you tell iOS what to do when the user has made a decision, but that code only gets called when that decision is finally made. As soon as you call **requestAuthorization()**, execution continues immediately on the very next line after it – iOS doesn't stop while the user thinks. Instead, you sent the closure – the code to run – to the notification center, and that's what will get called when the user makes a choice.

Finally, let's take another look at **for case let** syntax. Its job is to perform some sort of filtering on our data based on the result of a check, which means inside the Swift loop the compiler has more information about the data it's working with.

For example, if we wanted to loop over all the subviews of a **UIView**, we'd write this:

```
for subview in view.subviews {
   print("Found a subview with the tag: \(subview.tag)")
}
```

All views have a tag, which is an identifying number we can use to distinguish between views in some specific circumstances.

However, what if wanted to find all the labels in our subviews and print out their text? We can't print out the text above, because a regular **UIView** doesn't have a **text** property, so we'd probably write something like this:

```
for subview in view.subviews {
   guard let label = subview as? UILabel else { continue }
   print("Found a label with the text: \(label.text)")
}
```

That certainly works, but this is a case where **for case let** can do the same job in less code:

```
for case let label as UILabel in view.subviews {
   print("Found a label with text \(label.text)")
}
```

for case let can also do the job of checking optionals for a value. If it finds a value inside it will unwrap it and provide that inside the loop; if there is no value that element will be skipped.

The syntax for this is a little curious, but I think you'll appreciate its simplicity:

```
let names = ["Bill", nil, "Ted", nil]

for case let name? in names {
   print(name)
}
```

In that code the **names** array will be inferred as **[String?]** because elements are either strings or **nil**. Using **for case let** there will skip the two **nil** values, and unwrap and print the two strings.

Challenge

Have you ever heard the phrase, “imitation is the highest form of flattery”? I can’t think of anywhere it’s more true than on iOS: Apple sets an extremely high standard for apps, and encourages us all to follow suit by releasing a vast collection of powerful, flexible APIs to work with.

Your challenge for this milestone is to use those API to imitate Apple as closely as you can: I’d like you to recreate the iOS Notes app. I suggest you follow the iPhone version, because it’s fairly simple: a navigation controller, a table view controller, and a detail view controller with a full-screen text view.

How much of the app you imitate is down to you, but I suggest you work through this list:

1. Create a table view controller that lists notes. Place it inside a navigation controller. (Project 1)
2. Tapping on a note should slide in a detail view controller that contains a full-screen text view. (Project 19)
3. Notes should be loaded and saved using **Codable**. You can use **UserDefaults** if you want, or write to a file. (Project 12)
4. Add some toolbar items to the detail view controller – “delete” and “compose” seem like good choices. (Project 4)
5. Add an action button to the navigation bar in the detail view controller that shares the text using **UIActivityViewController**. (Project 3)

Once you’ve done those, try using Interface Builder to customize the UI – how close can you make it look like Notes?

Note: the official Apple Notes app supports rich text input and media; don’t worry about that, focus on plain text.

Go ahead and try now. Remember: don’t fret if it sounds hard – it’s *supposed* to stretch you.

Here are some hints in case you hit a problem:

- You could represent each note using a custom **Note** class if you wanted, but to begin with perhaps just make each note a string that gets stored in a **notes** array.
- If you do intend to go down the custom class route for notes, make sure you conform to **Codable** – you might need to re-read project 12.
- Make sure you use **NotificationCenter** to update the insets for your detail text view when the keyboard is shown or hidden.
- Try changing the **tintColor** property in Interface Builder. This controls the color of icons in the navigation bar and toolbar, amongst other things.

Milestone 9
Beacons and Extensions

What you learned

If everything is going to plan you should be starting to find the code for these projects easier and easier. That's not to say it's all plain sailing from now on – there are still some tough things to learn! – but it does show that your skills are advancing and you're starting to retain what you've learned.

Let's recap what you've learned in this milestone:

- You met **CLLocationManager** from the Core Location framework, which is the central point for location permissions and updates in iOS.
- You learned to set "Always" or "When in use" in your Info.plist, so that iOS can show a meaningful permission request to your user.
- We used **CLBeaconRegion** to scan for a particular iBeacon, using a UUID, major number, and minor number. That's enough to identify anywhere in the world uniquely.
- When a beacon was being ranged, we used **CLProximity** to determine how close it was. iBeacons use extremely low signal strengths to preserve battery life, so the proximity levels are quite vague!
- You learned how to draw custom paths using **UIBezierPath**, then render them in SpriteKit using **SKShapeNode** – we used these to draw the swiping glow effect in Swifty Ninja.
- We made the bomb sound play using **AVAudioPlayer**, because we wanted to be able to stop it at any point. In project 36 you'll learn about **SKAudioNode**, which is able to achieve similar results.
- You learned how Swift strings are more than just arrays of characters, which is why we can't write **someString[3]** by default.
- We looked at common methods of **String** and added some more of our own using extensions.
- I introduced you to **NSAttributedString**, and how it lets us add colors, fonts, and more to text.

Key points

Before you continue to the next milestone, there are two things I'd like to discuss briefly.

First, project 22 introduced Core Location to enable scanning for iBeacons. That's just one of several things that Core Location does, and I couldn't possibly continue without at least giving you a taste of the others. For example, Core Location's functionality includes:

- Providing co-ordinates for the user's location at a granularity you specify.
- Tracking the places the user has visited.
- Indoor location, even down to what floor a user is on, for locations that have been configured by Apple.
- Geocoding, which converts co-ordinates to user-friendly names like cities and streets.

Using these things starts with what you have already: modifying the Info.plist to provide a description of how you intend to use location data, then requesting permission. If you intend to use visit tracking you should request the "always" permission because visits are delivered to you in the background.

Once you have permission, try using this to get the user's location just once, rather than ongoing:

```
locationManager = CLLocationManager()
manager.delegate = self

   // request the user's coordinates
locationManager.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: Error) {
   print("Failed to find user's location: \
(error.localizedDescription)")
}
```

You can also request visit monitoring, like this:

```
// start monitoring visits
locationManager.startMonitoringVisits()
```

When the user arrives or departs from a location, you'll get a callback method if you implement it. The method is the same regardless of whether the user arrived or departed at the location, so you need to check the **departureDate** property to decide.

Here's an example to get you started:

```
func locationManager(_ manager: CLLocationManager, didVisit
visit: CLVisit) {
   if visit.departureDate == Date.distantFuture {
      print("User arrived at location \(visit.coordinate) at
time \(visit.arrivalDate)")
   } else {
      print("User departed location \(visit.coordinate) at time
\(visit.departureDate)")
   }
}
```

Note: the definition of a "visit" is pretty vague because iOS can't tell whether the user has walked into a store or is just standing at a bus stop or sitting in traffic.

The second thing I'd like to discuss further is Swift extensions. These are extraordinarily powerful, because you can extend specific types (e.g. **Int** and **String**) but also whole protocols of types (e.g. "all collections".) Protocol extensions allow us to build up functionality extremely quickly, and using it extensively – a technique known as protocol-oriented programming – is common.

We just wrote several extensions on **String**, which is what we call a *concrete* data type – a thing you can actually make. We can write extensions for other concrete types like **Int**, like this:

```
extension Int {
   var isOdd: Bool {
      return !self.isMultiple(of: 2)
   }

   var isEven: Bool {
      return self.isMultiple(of: 2)
   }
}
```

However, that will only extend **Int** – Swift has a variety of different sizes and types of integers to handle very specific situations. For example, **Int8** is a very small integer that holds between -128 and 127, for times when you don't need much data but space is really restricted. Or there's **UInt64**, which holds much larger numbers than a regular **Int**, but those numbers must always be positive.

Making extensions for whole protocols at once adds our functionality to many places, which in the case of integers means we can add **isOdd** and **isEven** to **Int**, **Int8**, **UInt64**, and more by extending the **BinaryInteger** protocol that covers them all:

```
extension BinaryInteger {
   var isOdd: Bool {
      return !self.isMultiple(of: 2)
```

```
    }

    var isEven: Bool {
        return self.isMultiple(of: 2)
    }
}
```

However, where things get *really* interesting is if when we want only a subset of a protocol to be extended. For example, Swift has a **Collection** protocol that covers arrays, dictionaries, sets, and more, and if we wanted to write a method that counted how many odd and even numbers it held we might start by writing something like this:

```
extension Collection {
    func countOddEven() -> (odd: Int, even: Int) {
        // start with 0 even and odd
        var even = 0
        var odd = 0

        // go over all values
        for val in self {
            if val.isMultiple(of: 2) {
                // this is even; add one to our even count
                even += 1
            } else {
                // this must be odd; add one to our odd count
                odd += 1
            }
        }

        // send back our counts as a tuple
        return (odd, even)
    }
}
```

However, that code won't work. You see, we're trying to extend all collections, which means we're asking Swift to make the method available on arrays like this one:

```
let names = ["Arya", "Bran", "Rickon", "Robb", "Sansa"]
```

That array contains strings, and we can't check whether a string is a multiple of 2 – it just doesn't make sense.

What we *mean* to say is "add this method to all collections that contain integers, regardless of that integer type." To make this work, you need to specify a **where** clause to filter where the extension is applied: we want this extension only for collections where the elements inside that collection conform to the **BinaryInteger** protocol.

This is actually surprisingly easy to do – just modify the extension to this:

```
extension Collection where Element: BinaryInteger {
```

As you'll learn, these extension constraints are extraordinarily powerful, particularly when you constrain using a protocol rather than specific type. For example, if you extend **Array** so that your methods only apply to arrays that hold **Comparable** objects, the methods in that extension gain access to a whole range of built-in methods such as **firstIndex(of:)**, **contains()**, **sort()**, and more – because Swift knows the elements must all conform to **Comparable**.

If you want to try such a constraint yourself – and trust me, you'll need it for one of the challenges coming up! – write your extensions like this:

```
extension Array where Element: Comparable {
    func doStuff(with: Element) {
    }
}
```

Inside the **doStuff()** method, Swift will ensure that **Element** automatically means whatever type of element the array holds.

That’s just a teaser of what’s to come once your Swift skills advance a little further, but I hope you’re starting to see why Swift is called a protocol-oriented programming language – you can extend specific types if you want to, but it’s far more efficient – and powerful! – to extend whole groups of them at once.

Challenge

Your challenge this time is *not* to build a project from scratch. Instead, I want you to implement three Swift language extensions using what you learned in project 24. I've ordered them easy to hard, so you should work your way from first to last if you want to make your life easy!

Here are the extensions I'd like you to implement:

1. Extend **UIView** so that it has a **bounceOut(duration:)** method that uses animation to scale its size down to 0.0001 over a specified number of seconds.
2. Extend **Int** with a **times()** method that runs a closure as many times as the number is high. For example, **5.times { print("Hello!") }** will print "Hello" five times.
3. Extend **Array** so that it has a mutating **remove(item:)** method. If the item exists more than once, it should remove only the first instance it finds. Tip: you will need to add the **Comparable** constraint to make this work!

As per usual, please try and complete this challenge yourself before you read my hints below. And again, don't worry if you find this challenge *challenging* – the clue is in the name, these are designed to make you think!

Here are some hints in case you hit problems:

1. Animation timings are specified using a **TimeInterval**, which is really just a **Double** behind the scenes. You should specify your method as **bounceOut(duration: TimeInterval)**.
2. If you've forgotten how to scale a view, look up **CGAffineTransform** in project 15.
3. To add **times()** you'll need to make a method that accepts a closure, and that closure should accept no parameters and return nothing: **() -> Void**.
4. Inside **times()** you should make a loop that references **self** as the upper end of a range – that's the value of the integer you're working with.
5. Integers can be negative. What happens if someone writes **let count = -5** then uses **count.times { … }** and how can you make that better?

6. When it comes to implementing the **remove(item:)** method, make sure you constrain your extension like this: **extension Array where Element: Comparable**.
7. You can implement **remove(item:)** using a call to **firstIndex(of:)** then **remove(at:)**.

Those hints ought to be enough for you to solve the complete challenge, but if you still hit problems then read over my solutions below, or put this all into a playground to see it in action.

```
import UIKit

// extension 1: animate out a UIView
extension UIView {
   func bounceOut(duration: TimeInterval) {
      UIView.animate(withDuration: duration) { [unowned self]
in
         self.transform = CGAffineTransform(scaleX: 0.0001, y:
0.0001)
      }
   }
}

// extension 2: create a times() method for integers
extension Int {
   func times(_ closure: () -> Void) {
      guard self > 0 else { return }

      for _ in 0 ..< self {
         closure()
      }
   }
}
```

```
// extension 3: remove an item from an array
extension Array where Element: Comparable {
   mutating func remove(item: Element) {
      if let location = self.firstIndex(of: item) {
         self.remove(at: location)
      }
   }
}

// some test code to make sure everything works
let view = UIView()
view.bounceOut(duration: 3)

5.times { print("Hello") }

var numbers = [1, 2, 3, 4, 5]
numbers.remove(item: 3)
```

Milestone 10

Core Motion and Core Graphics

What you learned

All three of the projects in this milestone drew on frameworks outside UIKit: you tried the Multipeer Connectivity framework, Core Motion, and Core Graphics. These all form part of the wider iOS ecosystem, and I hope you're starting to realize just how far-reaching Apple's frameworks are – they really have done most of the work for you!

Let's recap what you've learned in this milestone:

- We covered **UIImagePickerController** again, and I hope at this point you're feeling you could almost use it blindfold! That's OK, though: we learn through repetition, so I'm not afraid to repeat things when it matters.
- We also repeated **UICollectionView**, and again you should be feeling pretty good about it by now.
- You met the MultipeerConnectivity framework, which is designed for ad-hoc networking. We sent images in project 25, but any kind of data works great – yes, even data that represents custom classes you encoded using **Codable**.
- We finally used **categoryBitMask**, **contactTestBitMask**, and **collisionBitMask** in all their glory, setting up different kinds of collisions for our ball. Remember, "category" defines what something is, "collision" defines what it should be bounce off, and "contact" defines what collisions you want to be informed on.
- We used **CMMotionManager** from Core Motion to read the accelerometer. Make sure you call **startAccelerometerUpdates()** *before* you try to read the **accelerometerData** property.
- The **#targetEnvironment(simulator)** code allowed us to add testing code for using the simulator, while also keeping code to read the accelerometer on devices. Xcode automatically compiles the correct code depending on your build target.

And then there's Core Graphics. Project 27 was one of the longest technique projects in all of Hacking with Swift, and with good reason: Core Graphics packed with features, and there are lots of methods and properties you need to learn to make use of it fully.

Here are some of the Core Graphics things we covered:

- The **UIGraphicsImageRenderer** is the primary iOS graphics renderer, and can export **UIImages** just by calling its **image()** method and providing some drawing code.
- You can get access to the underlying Core Graphics context by reading its **cgContext** property. This is where most of the interesting functionality is.
- You call **setFillColor()**, **setStrokeColor()**, **setLineWidth()** and such *before* you do your drawing. Think of Core Graphics like a state machine: set it up as you want, do some drawing, set it up differently, do some more drawing, and so on.
- Because Core Graphics works at a lower level than UIKit, you need to use **CGColor** rather than **UIColor**. Yes, the CG is short for "Core Graphics".
- You learned that you can call **translateBy()** and **rotate(by:)** on a Core Graphics context to affect the way it does drawing.
- We tried using **move(to:)** and **addLine(to:)** to do custom line drawing. In combination with rotation, this created an interesting effect with very little code.
- We drew attributed strings and a **UIImage** straight into a Core Graphics context by calling their **draw()** method.

Once you get used to Core Graphics, you start to realize that its incredible speed allows you to do procedurally generated graphics for games without too much work. More on *that* in project 29…

Key points

There are two things I'd like to discuss briefly, both extending what you already learned to help push your skills even further.

First, the **#targetEnvironment(simulator)** compiler directive. Swift has several of these, and I want to demonstrate two just briefly: **#line** and **#if swift**. **#line** is easy enough: when your code gets built it automatically gets replaced with the current line number. You can also use **#filename** and **#function**, and the combination of these are very useful in debugging strings.

The **#if swift** directive allows you to conditionally compile code depending on the Swift compiler version being used. So, you could write Swift 4.2 code and Swift 5.0 code in the same file, and have Xcode choose the right version automatically.

Now why, do you think, might you want such functionality? Well, there are two situations that you're likely to encounter:

1. You create a library that you distribute as Swift source code. Supporting more than one version of Swift helps reduce complexity for your users without breaking their code.
2. You want to experiment with a future version of Swift without breaking your existing code. Having both in the same file means you can toggle between them easily enough.

Here's some example code to get you started:

```
#if swift(>=5.0)
print("Running Swift 5.0 or later")
#else
print("Running Swift 4.2")
#endif
```

The second thing I'd like to touch on briefly is image rendering. Hacking with Swift is written specifically for the latest and greatest APIs from Apple, because if you're learning from scratch it's usually not worth bothering learning older technologies.

However, the case of image rendering is unique because the old technology – i.e., everything before iOS 10.0 – takes only a minute to learn. So, I want to show you just quickly how to render images before iOS 10, because it's likely you'll come across it in the wider world.

Here's the iOS 10 and later code we would use to render a circle:

```
let renderer = UIGraphicsImageRenderer(size: CGSize(width: 512,
height: 512))
let img = renderer.image { ctx in
    ctx.cgContext.setFillColor(UIColor.red.cgColor)
    ctx.cgContext.setStrokeColor(UIColor.green.cgColor)
    ctx.cgContext.setLineWidth(10)

    let rectangle = CGRect(x: 0, y: 0, width: 512, height: 512)
    ctx.cgContext.addEllipse(in: rectangle)
    ctx.cgContext.drawPath(using: .fillStroke)
}
```

To convert that to pre-iOS 10 rendering, you need to learn four new functions:

1. **UIGraphicsBeginImageContextWithOptions()**. This starts a new Core Graphics rendering pass. Pass it your size, then a rendering scale, and whether the image should be opaque. If you want to use the current device's scale, use 0 for the scale parameter.
2. Just starting a rendering pass doesn't give you a context. To do that, you need to use **UIGraphicsGetCurrentContext()**, which returns a **CGContext?**. It's optional because of course Swift doesn't know we just started a rendering pass.
3. Call **UIGraphicsGetImageFromCurrentImageContext()** when you want to extract a **UIImage** from your rendering. Again, this returns an optional (in this case **UIImage?**) because Swift doesn't know a rendering pass is active.
4. Call **UIGraphicsEndImageContext()** when you've finished, to free up the memory from your rendering.

As you can see, the older calls are a little more flaky: having extra optionality is never welcome, and you need to remember to both start and end your rendering blocks. Still, if you want to give it a try, here’s the same circle rendering code rewritten for iOS 9.3 and earlier:

```
UIGraphicsBeginImageContextWithOptions(CGSize(width: 512,
height: 512), false, 0)

if let ctx = UIGraphicsGetCurrentContext() {
   ctx.setFillColor(UIColor.red.cgColor)
   ctx.setStrokeColor(UIColor.green.cgColor)
   ctx.setLineWidth(10)

   let rectangle = CGRect(x: 5, y: 5, width: 502, height: 502)
   ctx.addEllipse(in: rectangle)
   ctx.drawPath(using: .fillStroke)
}

if let img = UIGraphicsGetImageFromCurrentImageContext() {
   // "img" is now a valid UIImage – use it here!
}

UIGraphicsEndImageContext()
```

Challenge

Your challenge for this milestone is to create a meme generation app using **UIImagePickerController**, **UIAlertController**, and Core Graphics. If you aren't familiar with them, memes are a simple format that shows a picture with one line of text overlaid at the top and another overlaid at the bottom.

Your app should:

- Prompt the user to import a photo from their photo library.
- Show an alert with a text field asking them to insert a line of text for the top of the meme.
- Show a second alert for the bottom of the meme.
- Render their image plus both pieces of text into one finished **UIImage** using Core Graphics.
- Let them share that result using **UIActivityViewController**.

Both the top and bottom pieces of text should be optional; the user doesn't need to provide them if they don't want to.

Try to solve the challenge now. As per usual, there are some hints below in case you hit problems.

1. Your UI can be pretty simple: a large image view, with three buttons below: Import Picture, Set Top Text, and Set Bottom Text.
2. Both pieces of text can be read in using a **UIAlertController** with a text field inside.
3. When rendering your finished image, make sure you draw your **UIImage** first, then add the text on top.
4. **NSAttributedString** has keys to specify the stroke width and color of text, which would make it more readable – can you experiment to figure it out?

OK, that's enough hints – get coding!

Milestone 11

Secrets and Explosions

What you learned

Project 29 was a serious game with a lot going on, not least the dynamically rendered buildings with destructible terrain, the scene transitions and the UIKit/SpriteKit integration.

And in project 30 we took our first steps outside of Xcode and into Instruments. I could write a whole book on Instruments, partly because it's extremely powerful, but also because it's extremely complicated. As per usual, I tried to cherrypick things so you can see useful, practical benefits from what I was teaching, and certainly you have the skills now to be able to diagnose and result a variety of performance problems on iOS.

Here are some of the things you learned in this milestone:

- How to access the keychain using SwiftKeychainWrapper.
- How to force the keyboard to disappear by calling **resignFirstResponder()** on a text view. (And remember: it also works on text fields.)
- How to detect when your app is moving to the background by registering for the **UIApplication.willResignActiveNotification** notification.
- How to use **LAContext** from the LocalAuthentication framework to require Touch ID authentication.
- Using the **stride()** function to loop across a range of numbers using a specific increment, e.g. from 0 to 100 in tens.
- Creating colors using the hue, saturation, and brightness. As I said, keeping the saturation and brightness constant while choosing different hues helps you create similar color palettes easily.
- SpriteKit texture atlases. These are automatically made by Xcode if you place images into a folder with the **.atlas** extension, and are drawn significantly quicker than regular graphics.
- Using **usesPreciseCollisionDetection** to make collisions work better with small, fast-moving physics bodies.
- Transitioning between scenes with the **presentScene()** method and passing in a transition effect. We'll be using this again in project 36, so you'll have ample time to practice transitions.

- Using the blend mode **.clear** to erase parts of an image. Once that was done, we just recalculated the pixel-perfect physics to get destructible terrain.
- Adding dynamic shadows to views using **layer.shadowRadius** and other properties – and particularly how to use the **layer.shadowPath** property to save shadow calculation.
- The importance of using **dequeueReusableCell(withIdentifier:)** so that iOS can re-use cells rather than continually creating new ones.
- How the **UIImage(named:)** initializer has an automatic cache to help load common images. When you don't need that, use the **UIImage(contentsOfFile:)** initializer instead.

Key points

There are two things I'd like to review for this milestone.

First, the **weak** keyword. We used it in project 29 to add a property to our game scene:

```
weak var viewController: GameViewController!
```

We also added the opposite property to the game view controller:

```
var currentGame: GameScene!
```

This approach allowed the game scene to call methods on the view controller, and vice versa. At the time I explained why one was **weak** and the other was not – do you remember? I hope so, because it's important!

There are four possibilities:

1. Game scene holds strong view controller and view controller holds strong game scene.
2. Game scene holds strong view controller and view controller holds weak game scene.
3. Game scene holds weak view controller and view controller holds weak game scene.
4. Game scene holds weak view controller and view controller holds strong game scene.

Remember, "strong" means "I want to own this; don't let this memory be destroyed until I'm done with it," and weak means "I want to use this but I don't want to own it; I'm OK if it gets destroyed, so don't keep it around on my account."

Now, the view controller has an **SKView**, which is what renders SpriteKit content. That's owned strongly, because obviously the view controller can't really work without something to draw to. And that **SKView** has a **scene** property, which is the current **SKScene** visible on the screen. That's *also* strongly owned. As a result, the view controller already - albeit indirectly – has strong ownership of the game scene.

As a result, both options 1 and 2 will cause a strong reference cycle, because they would cause the game scene to have a strong reference to something that has a strong reference back to the game scene. This isn't necessarily *bad* as long as you remember to break the strong reference cycle, but let's face it: why take the risk?

That leaves options 3 and 4: both have the game scene using a **weak** reference to the view controller, but one has a weak reference going back the other way and the other has a strong one. Which is better? Honestly, I'm not sure it matters: using a strong reference wouldn't result in anything new because there's already the indirect strong reference in place. So, use whichever you prefer!

The second thing I'd like to cover is much easier: it's the **UIImage(contentsOfFile:)** initializer for **UIImage**. Like I said in project 30, the **UIImage(named:)** initializer has a built-in cache system to help load and store commonly used images – anything you load with that initializer will automatically be cached, and so load instantly if you request it again.

Of course, if you *don't* want something to be cached, that's the wrong solution, which is where **UIImage(contentsOfFile:)** comes in: give it a path and it will load the image, with no magic caching ever happening.

The downside is that **UIImage(named:)** automatically finds images inside your app bundle, whereas **UIImage(contentsOfFile:)** does not. So, you need to write code like this:

```
let path = Bundle.main.path(forResource: someImage, ofType:
nil)!
imageView.image = UIImage(contentsOfFile: path)
```

That's hardly a lot of code, but it's never nice writing even simple repetitive code – and look at that force unwrap after **path()**! Wouldn't it be great to get rid of it? I'm going to show you how to do just that, and, as a bonus, I'm also going to teach you something new: convenience initializers.

You've already seen initializers – we've used dozens of them. They are special methods that create things, like **UILabel()** or **UIImage(named:)**. Swift has complex rules about its initializers, all designed to stop you trying to access something that hasn't been created yet.

Fortunately, there's one *easy* part, which is convenience initializers, which are effectively wrappers around basic initializers that are designed to make coding a bit more pleasant. A convenience initializer is able to do some work before calling a regular initializer, which in our case means we can add a wrapper around **UIImage(contentsOfFile:)** so that it's nicer to call.

To make things even better, we're also going to get rid of the force unwrap. Remember, **path(forResource:)** can return nil because the file you requested might not exist. Force unwrapping it works on occasion if you know something definitely exists, but it's usually a better idea to use an alternative such as failable initializers.

You've already used failable initializers several times – both **UIImage(named:)** and **UIImage(contentsOfFile:)** are failable, for example. A failable initializer is one that might return a valid created object, or it might fail and return nil. We're going to use this so that we can return **nil** if the image name can't be found in the app bundle.

So, leveraging the power of Swift extensions that you learned in project 24, here's a **UIImage** extension that creates a new, failable, convenience initializer called **UIImage(uncached:)**. It works like **UIImage(named:)** in that you don't need to provide the full bundle path, but it *doesn't* have the downside of caching images you don't intend to use more than once.

Here's the code:

```
extension UIImage {
   convenience init?(uncached name: String) {
      if let path = Bundle.main.path(forResource: name, ofType:
nil) {
         self.init(contentsOfFile: path)
      } else {
         return nil
      }
   }
}
```

Note the **init?** syntax that marks this as an initializer that returns an optional.

Challenge

Your challenge is to create a memory pairs game that has players find pairs of cards – it's sometimes called Concentration, Pelmanism, or Pairs. At the very least you should:

- Come up with a list of pairs. Traditionally this is two pictures, but you could also use capital cities (e.g. one card says France and its match says Paris), languages (e.g one card says "hello" and the other says "bonjour"), and so on.
- Show a grid of face-down cards. How many is down to you, but if you're targeting iPad I would have thought 4x4 or more.
- Let the player select any two cards, and show them face up as they are tapped.
- If they match remove them; if they don't match, wait a second then turn them face down again.
- Show a You Win message once all are matched.

You can use either SpriteKit or UIKit depending on which skill you want to practice the most, but I think you'll find UIKit much easier.

Don't under-estimate this challenge! To make it work you're going to need to draw on a wide variety of skills, and it *will* push you. That's the point, though, so take your time and give yourself space to think.

If you're looking for a more advanced challenge, go for a variant of the game that uses word pairs and add a parental option that lets them create new cards. This would mean:

- Authenticating users using Touch ID or Face ID.
- Showing a new view controller that lists all existing cards and lets them enter a new card.
- You can use a **UIAlertController** with one or two text fields for your card entry, depending on what kind of game you've made.

Please go ahead and try to solve the challenge now. My hints are below, but please try to avoid reading them unless you're really struggling.

- Start small. Seriously! Find something really simple that works, and only try something bigger or better once your simplest possible solution actually works.
- If you're using UIKit, you could try to solve this using a **UICollectionView**. This gives you a natural grid, as well as touch handling for selecting cells, but make sure you think carefully about cells being re-used – this might prove more difficult than you thought.
- An easier approach is to lay out your cards much like we did with the word letters in project 8, 7 Swifty Words. You could show your card backs as a button image, then when the button is tapped show the other side of the card – which might be as simple as changing the picture and making the button's text label have a non-clear color, or perhaps using Core Graphics to render the text directly onto the card front image.
- If you made the buttons work and want to try something fancier, you can actually create a flip animation to toggle between views – see my article **How to flip a UIView with a 3D effect: transition(with:)** for more information.
- In terms of tracking the game state it really only has three states: player has chosen zero cards, player has chosen one card (flip it over), and player has chosen two cards (flip the second one over). When they've chosen two cards you need to decide whether you have a match, then either remove the cards or flip them back down and go back to the first state.
- For the content to show, you can just type in a list of words/images into your code if you want, but you're welcome to use **Codable** if you want to push yourself.

Again, this is *not* an easy challenge so please take your time and don't feel bad when you find yourself having to look back at previous projects.