Project 5. Word Scramble

Overview

Brief: Create an anagram game while learning about closures and booleans.
Learn: NSString, closures, method return values, booleans, NSRange.

- Setting up
- Reading from disk: contentsOfFile
- Pick a word, any word: UIAlertController
- Prepare for submission: lowercaseString and NSIndexPath
- Returning values: contains
- Or else what?
- Wrap up

Setting up

Projects 1 to 4 were all fairly easy, because my goal was to teach you as much about Swift without scaring you away, while also trying to make something useful. But now that you're hopefully starting to become familiar with the core tools of iOS development, it's time to change up a gear and tackle something a bit meatier.

In this project you're going to learn how to make a word game that deals with anagrams, but as per usual I'll be hijacking it as a method to teach you more about iOS development. This time around we're going back to the table views as seen in Project 1, but you're also going to learn how to load text from files, how to ask

for user input in UIAlertController, and get a little more insight to how closures work.

In Xcode, create a new Master-Detail Application called Project5. Select iPhone for your target, then click Next to save it somewhere. Now open Interface Builder with Main.storyboard and delete the detail view controller entirely – it's the one on the far right. Right-click on DetailViewController.swift in the project navigator (the pane on the left showing all your files; Cmd + 1 shows it), then choose Delete and click "Move to Trash" when prompted.

Doing this will cause quite a few errors to appear in your project, but it's easily fixed – we just need to delete quite a bit of Apple's template! First, open the AppDelegate.swift file, and look for this code near the top of the file:

```
func application(application: UIApplication,
didFinishLaunchingWithOptions launchOptions: [NSObject:
AnyObject]?) -> Bool {
    // Override point for customization after application
launch.
    let splitViewController =
self.window!.rootViewController as! UISplitViewController
    let navigationController =
splitViewController.viewControllers[splitViewController.viewC
ontrollers.count-1] as! UINavigationController

navigationController.topViewController!.navigationItem.leftBa
rButtonItem = splitViewController.displayModeButtonItem()
    splitViewController.delegate = self

    return true
```

```
}
```

Please delete everything in the body of that method except `return true`, like this:

```
func application(application: UIApplication,
didFinishLaunchingWithOptions launchOptions: [NSObject:
AnyObject]?) -> Bool {
     return true
}
```

Staying in that same file, scroll to the bottom to find this method with an extremely long signature:

```
func splitViewController(splitViewController:
UISplitViewController, collapseSecondaryViewController
secondaryViewController:UIViewController,
ontoPrimaryViewController
primaryViewController:UIViewController) -> Bool {
```

Please delete that whole method.

Next, open the `MasterViewController.swift` file, and delete everything except `super.viewDidLoad()` from the `viewDidLoad()` method, then delete the `viewWillAppear()`, `insertNewObject()` and `prepareForSegue()` methods entirely.

Finally, find the `tableView(_:canEditRowAtIndexPath:)` and `tableView(_:commitEditingStyle:forRowAtIndexPath:)` methods and delete those too.

The last code change is to find and delete this property near the top of the class:

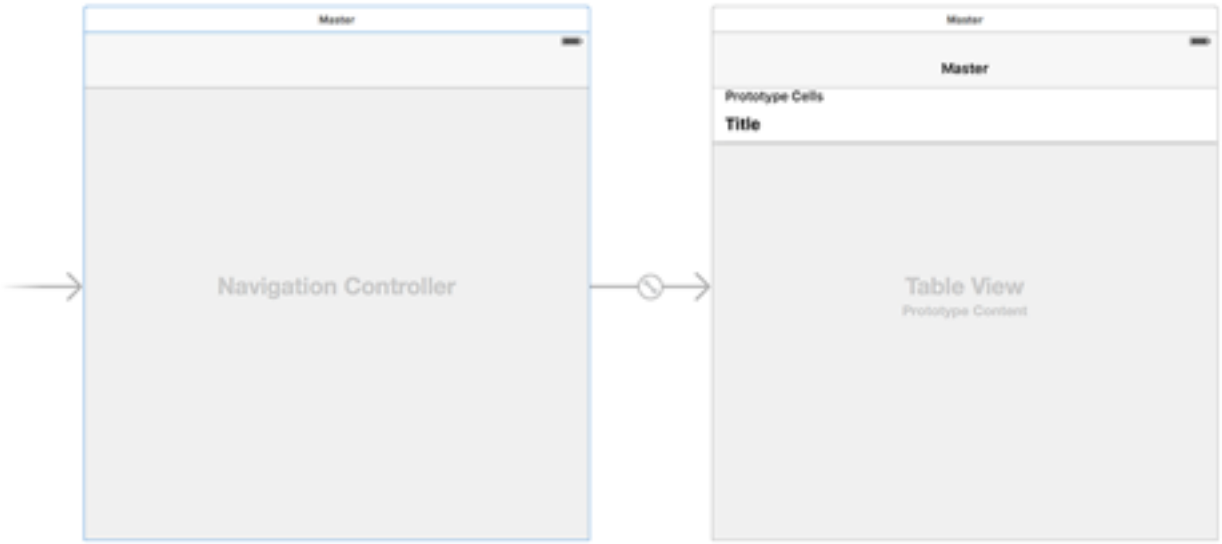
```
var detailViewController: DetailViewController? = nil
```

That's all the code cleaned up, but we need to do a little Interface Builder surgery before we're complete. As you saw back in Project 1, Apple's Master-Detail Application template sets up a split view controller plus two navigation controllers, as well as a table view controller and a regular view controller with a detail label – this is all rather overkill for our simple application, so we're going to 3/5ths of that to make our whole app simpler.

So, open Main.storyboard in Interface Builder. Now select and delete the Split View Controller (the one on the far left with a dark gray background color), then delete the bottom navigation controller and its associated view controller. You should be left with just two things: a navigation controller, and, to its right, a table view controller.

Apple had it set up so that the split view controller was the initial view controller, which is what gets shown when the application launches. We just deleted that, so we need to pick a new initial view controller instead. To do that, select the remaining navigation controller, go to the Attributes Inspector (Alt + Cmd + 4), and select the "Is Initial View Controller" checkbox that's about half way down the list of options. If it's worked, you should see an arrow appear to the left of the navigation controller.

The finished `Interface Builder` layout should look like the screenshot below:



The project is now so small that you might wander why we didn't just start from scratch! Still, there's a fair amount remaining, and deleting stuff can be quite cathartic. Even better: your project is now a clean table view project, ready for customisation – let's do this!

(PS: Is it faster to start with a `Single View application`? Possibly, but possibly not; at least this way you get some experience in cleaning up `Apple's` templates. Plus, deleting stuff is fun!)

Reading from disk: contentsOfFile

We're going to make an anagram game, where the user is asked to make words out of a larger word. We're going to put together a list of possible starter words for the game, and that list will be stored in a separate file. But how we get the text from the file into the app? Well, it turns out that `Swift's String` data type makes it a cinch – thanks, `Apple`!

In the Content folder you'll find the file start.txt. Please drag that into your Xcode project, making sure that "Copy items if needed" is checked.

The start.txt file contains over 12,000 eight-letter words we can use for our game, all stored one word per line. We need to turn that into an array of words we can play with. Behind the scenes, those line breaks are marked with a special line break character that is usually expressed as "\n". So, we need to load that word list into a string, then split it into an array by breaking up wherever we see \n.

First, go to the start of your class and make a new array. There will be an existing one there from Apple's template, so put this alongside:

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

While you're there, you might as well change Apple's array to be [String] rather than [AnyObject], because we'll only ever be storing strings in there. You'll need to adjust the table view's cellForRowAtIndexPath method from this:

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

…to this:

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

We did this in Project 1 too, so hopefully it's not too hard. This is a result of the objects array no longer containing AnyObject, but just strings.

Second, loading our array. This is done in three parts: finding the path to our `start.txt` file, loading the contents of that file, then splitting it into an array.

Finding a path to a file is something you'll do a lot, because even though you know the file is called "`start.txt`" you don't know where it might be on the filesystem. So, we use a built-in method of `NSBundle` to find it: `pathForResource()`. This takes as its parameters the name of the file and its path extension, and returns a `String?` – i.e., you either get the path back or you get `nil`.

Loading a file into a string is also something you'll need to get familiar with, and again there's an easy way to do it: when you create an `String` instance, you can ask it to create itself from the contents of a file at a particular path. You can also provide it with parameters to tell you the text encoding that was used, but for this project we don't care.

Finally, we need to split our single string into an array of strings based on wherever we find a line break (`\n`). This is as simple as another method call on `String`: `componentsSeparatedByString()`. Tell it what string you want to use as a separator (for us, that's `\n`), and you'll get back an array.

Before we get onto the code, there are two things you should know: `pathForResource()` and creating an `String` from the contents of a file both return `String?`, which means we need to check and unwrap the optional using `if/let` syntax.

OK, time for some code. Put this into `viewDidLoad()`, after the super call:

```
if let startWordsPath =
NSBundle.mainBundle().pathForResource("start", ofType: "txt")
{
```

```
    if let startWords = try? String(contentsOfFile:
startWordsPath, usedEncoding: nil) {
        allWords =
startWords.componentsSeparatedByString("\n")
    }
} else {
    allWords = ["silkworm"]
}
```

If you look carefully, there's a new keyword in there: `try?`. You already saw `try!` previously, and really we could use that here too because we're loading a file from our app's bundle so any failure is likely to be catastrophic. However, this way I have a chance to teach you something new: `try?` means "`call this code, and if it throws an error just send me back nil instead`". This means the code you call will always work, but you need to unwrap the result carefully.

As you can see, that code carefully checks for and unwraps the contents of our start file, then converts it to an array. When it has finished, `allWords` will contain `12,000+` strings ready for us to use in our game.

To prove that everything is working before we continue, let's create a new method called `startGame()`. This will be called every time we want to generate a new word for the player to work with:

```
func startGame() {
    allWords =
GKRandomSource.sharedRandom().arrayByShufflingObjectsInArray(
allWords) as! [String]
    title = allWords[0]
    objects.removeAll(keepCapacity: true)
    tableView.reloadData()
```

```
}
```

Note the `first line` there is what shuffles the words arrow. We used this is in `Project 2`, so you should remember that you need to import the `GameplayKit` framework in order for that to work.

`Line 2` sets our view controller's title to be the first word in the array, once it has been shuffled. This will be the word the player has to find.

`Line 3` removes all values from the objects array. This array was created for us by the `Xcode` template, and we'll be using it to store the player's answers so far. We aren't adding anything to it right now, so `removeAll()` won't do anything just yet.

`Line 4` is the interesting part: it calls the `reloadData()` method of tableView. That table view was declared up near the… hey, wait a minute! Where's that table view come from? Well, we certainly didn't make it. Instead, it's made for us because – dramatic drum roll – `MasterViewController` doesn't inherit from `UIViewController`.

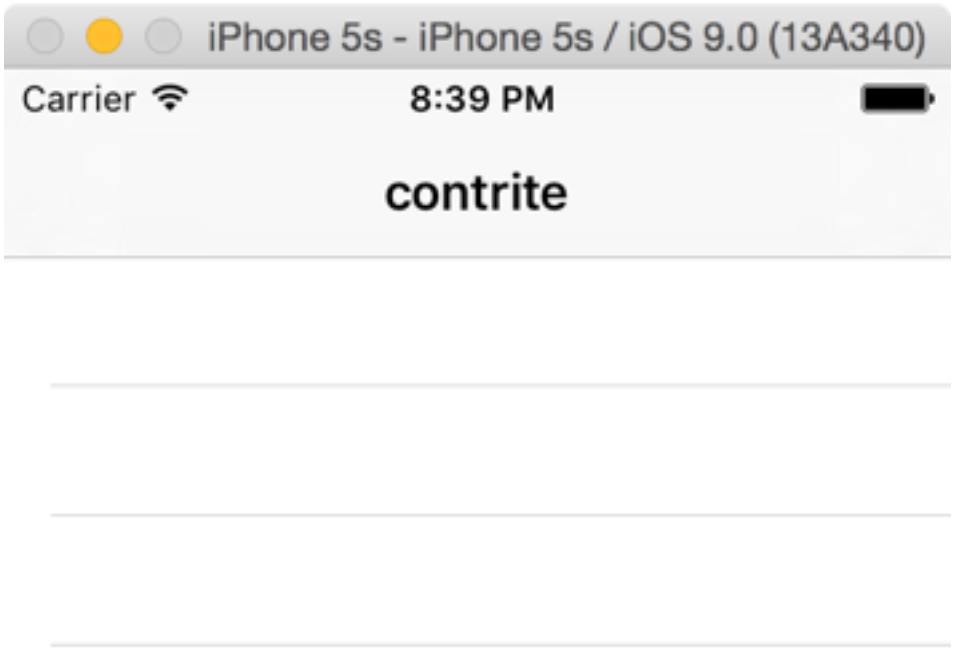
Yes, I know I said `UIViewController` is used for all the screens in your app, and it is, just sometimes not directly. In this case, `MasterViewController` inherits from `UITableViewController`, and `UITableViewController` inherits from `UIViewController`. There's an inheritance chain, with each component addings its own functionality.

Don't fear: most of your view controllers will inherit either directly from `UIViewController` or will go through `UITableViewController` first. It's just that table views are so pervasive in `iOS` that `Apple` took the opportunity to bake in some extra behavior for developers.

So what does `UITableViewController` do that `UIViewController` doesn't? Well, it comes with a full-screen table, for starters. When the view is shown, `UITableViewController` automatically flashes the scrollbars of the table so users know they can scroll, and if the keyboard appears then `UITableViewController` automatically adjusts itself so its content doesn't go beneath the keyboard.

Anyway, `UITableViewController` is what we are based on for `MasterViewController`, and that's where our `tableView` comes from. Calling `reloadData()` will cause the table view to check how many rows it has and reload them all.

Our table view doesn't have any rows yet, so this won't do anything. However, the method is ready to be used, and allows us to check we've loaded all the data correctly, so add this just before the end of `viewDidLoad()`:




```
startGame()
```

Now press Cmd + R to run the app, and you ought to see an eight-letter word at the top, ready for play to begin.

Pick a word, any word: UIAlertController

This game will prompt the user to enter a word that can be made from the eight-letter prompt word. For example, if the eight-letter word is “agencies”, the user could enter “cease”. We're going to solve this with UIAlertController, because it's a nice fit, and also gives me the chance to introduce some new teaching. I'm all about ulterior motives!

Add this code to viewDidLoad(), just after the call to super:

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

That creates a new UIBarButtonItem using the “add” system item, and configured it to run a method called promptForAnswer() when tapped. This method is going to show a UIAlertController with space for the user to enter an answer, and when the user clicks Submit to that alert controller the answer is checked to make sure it's valid.

Before I give you the code, let me explain what you need to know.

You see, we're about to use a closure, and things get a little complicated. As a reminder, these are chunks of code that can be treated like a variable – we can send the closure somewhere, where it gets stored away and executed later. To make this work, Swift takes a copy of the code and captures any objects it references, so it can use them later.

But there's a problem: what if the closure references the view controller? Then what could happen is a strong reference cycle: the view controller owns an object that owns a closure that owns the view controller, and nothing could ever be destroyed.

I'm going to try (and likely fail!) to give you a metaphorical example, so please bear with me. Imagine if you built two cleaning robots, red and blue. You told the red robot, “`don't stop cleaning until the blue robot stops`”, and you told the blue robot “`don't stop cleaning until the red robot stops`”. When would they stop cleaning? Never, because neither will make the first move.

This is the problem we are facing with a strong reference cycle: object `A` owns object `B`, and object `B` owns a closure that referenced object `A`. And when closures are created, they capture everything they use, thus object `B` owns object `A`.

Strong reference cycles used to be hard to find, but you'll be glad to know `Swift` makes them trivial. In fact, `Swift` makes it so easy that you will use its solution even when you're not sure if there's a cycle simply because you might as well.

So, please brace yourself: we're about to take our first look at actual closures. The syntax will hurt. And when you finally understand it, you'll come across examples online that make your brain hurt all over again.

Ready? Here's the `promptForAnswer()` method:

```
func promptForAnswer() {
    let alertController = UIAlertController(title: "Enter answer", message: nil, preferredStyle: .Alert)

alertController.addTextFieldWithConfigurationHandler(nil)
```

```
    let submitAction = UIAlertAction(title: "Submit", style:
.Default) { [unowned self, alertController] (action:
UIAlertAction!) in
        let answer = alertController.textFields![0]
        self.submitAnswer(answer.text!)
    }

    alertController.addAction(submitAction)

    presentViewController(alertController, animated: true,
completion: nil)
}
```

That one method introduces quite a few new things, but before we look at them let's eliminate the easy stuff.

- Creating a new `UIAlertController:` we did that in `Project 2`.
- `addTextFieldWithConfigurationHandler()` just adds an editable text field to the `UIAlertController`. We could do more with it, but it's enough for now.
- `addAction()` is used to add a `UIAlertAction` to a `UIAlertController`. We used this in `Project 2` also.
- `presentViewController` is also from `Project 2`. Clearly `Project 2` was brilliant!

That leaves the tricky stuff: creating `submitAction`. These handful of lines of code demonstrate no fewer than five new things to learn, all of which are important. I'm going to sort them easiest first, starting with `UITextField`.

You've already seen `UILabel`: it's a simple `UIView` subclass that shows a string of uneditable text on the screen. `UITextField` is similar, except it's editable. We added a single text field to the `UIAlertController` using its `addTextFieldWithConfigurationHandler()` method, and we now read out the value that was inserted.

Next up is something called trailing closure syntax. I know, I know: you haven't even learned about regular closures yet, and you're already having to learn about trailing closures! Well, they are related, and trailing closures aren't hard, so give it a chance.

Here's part of a line of code from `Project 2`:

```
UIAlertAction(title: "Continue", style: .Default, handler:
askQuestion)
```

This is from a similar situation: we're using `UIAlertController` and `UIAlertAction` to add buttons that the user can tap on. Back then, we used a separate method (`askQuestion()`) to avoid having to explain closures too early, but you can see that I'm passing `askQuestion` to the handler parameter of the `UIAlertAction`.

Closures can be thought of as a bit like anonymous functions. That is, rather than passing the name of a function to execute, we're just passing a lump of code. So we could conceptually rewrite that line to be this:

```
UIAlertAction(title: "Continue", style: .Default, handler:
{ CLOSURE CODE HERE })
```

But that has one critical problem: it's ugly! If you're executing lots of code inside the closure, it looks strange to have a one-line function call taking a 10-line parameter.

So, `Swift` has a solution, called trailing closure syntax. Any time you are calling a method that expects a closure as its final parameter – and there are many of them – you can eliminate that final parameter entirely, and pass it inside braces instead. This is optional and automatic, and would turn our conceptual code into this:

```
UIAlertAction(title: "Continue", style: .Default) {
    CLOSURE CODE HERE
}
```

Everything from the opening brace to the close is part of the closure, and is passed into the `UIAlertAction` creation as its last parameter. Easy!

Next, `(action: UIAlertAction!) in`. If you remember `Project 2`, we had to modify the `askQuestion()` method so that it accepted a `UIAlertAction` parameter saying what button was tapped, like this:

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

We had no choice but to do that, because the handler parameter for UIAlertAction expects a method that takes itself as a parameter. And that's what's happening here: we're giving the `UIAlertAction` some code to execute when it is tapped, and it wants to know that that code accepts a parameter of type `UIAlertAction`.

The in keyword is important: everything before that describes the closure; everything after that is the closure. So (`action: UIAlertAction!`) in means that it accepts one parameter in, of type `UIAlertAction`.

I used this way of writing the closure because it's so similar to that used in Project 2. However, Swift knows what kind of closure this needs to be, so we could simplify it a little: from this…

```
(action: UIAlertAction!) in
```

…to this:

```
action in
```

In our current project, we could simplify this even further: we don't make any reference to the action parameter inside the closure, which means we don't need to give it a name at all. In Swift, to leave a parameter unnamed you just use an underscore character, like this:

```
_ in
```

Fourth and fifth are going to be tackled together: unowned and self..

Swift "captures" any constants and variables that are used in a closure, based on the values of the closure's surrounding context. That is, if you create an integer, a string, an array and another class outside of the closure, then use them inside the closure, Swift captures them.

This is important, because the closure references the variables, and might even change them. But I haven't said yet what "capture" actually means, and that's because it depends what kind of data you're using. Fortunately, Swift hides it all away so you don't have to worry about it…

…except for those strong reference cycles I mentioned. Those you need to worry about. That's where objects can't even be destroyed because they all hold tightly on to each other – known as strong referencing.

`Swift's` solution is to let you declare that some variables aren't held onto quite so tightly. It's a two-step process, and it's so easy you'll find yourself doing it for everything just in case.

First, you must tell `Swift` what variables you don't want strong references for. This is done in one of two ways: `unowned` or `weak`. These are somewhat equivalent to implicitly unwrapped optionals (`unowned`) and regular optionals (`weak`): a weakly owned reference might be `nil`, so you need to unwrap it; an `unowned` reference is one you're certifying cannot be `nil` and so doesn't need to be unwrapped, however you'll hit a problem if you were wrong.

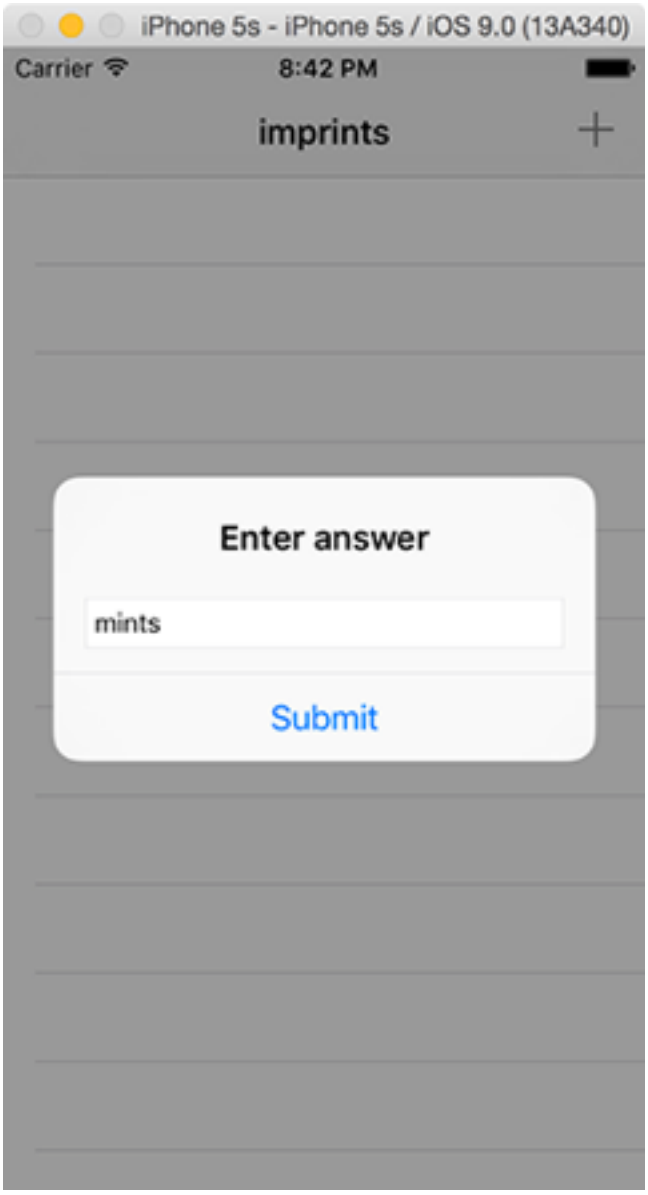
In our code we use this: `[unowned self, alertController]`. That declares `self` (the current view controller) and `alertController` (our `UIAlertController`) to be captured as `unowned` references inside the closure. It means the closure can use them, but won't create a strong reference cycle because we've made it clear the closure doesn't own either of them.

But that's not enough for `Swift`. Inside our method we're calling the `submitAnswer()` method of our view controller. We haven't created it yet, but you should be able to see it's going to take the answer the user entered and try it out in the game.

This `submitAnswer()` method is external to the closures current context, so when you're writing it you might not realise that calling `submitAnswer()` implicitly requires that self be captured by the closure. That is, the closure can't call `submitAnswer()` if it doesn't capture the view controller.

We've already declared that `self` is `unowned` by the closure, but `Swift` wants us to be absolutely sure we know what we're doing: every call to a method or property of the current view controller must prefixed with "`self.`", as in `self.submitAnswer()`.

In `Project 1`, I told you there were two trains of thought regarding use of self, and said, "`The first group of people never like to use self. unless it's required, because when it's required it's actually important and meaningful, so using it in places where it isn't required can confuse matters`".



Implicit capture of `self` in closures is that place when using self is required and meaningful: `Swift` won't let you avoid it here. By restricting your use of `self` to closures, you can easily check your code doesn't have any reference cycles by searching for "`self`" – there ought not to be too many to look through!

Prepare for submission: lowercaseString and NSIndexPath

You can breathe again: we're done with closures for now. I know that wasn't easy, but once you understand basic closures you really have come a long in your Swift adventure.

We're going to do some much easier coding now, because believe it or not we're not that far from making this game actually work!

First, let's make your code compile again, because right now it's calling `self.submitAnswer()` and we haven't made that method yet. So, add this new method somewhere in the class:

```
func submitAnswer(answer: String) {
}
```

That's right, it's empty – it's enough to make the code compile cleanly so we can carry on.

We have now gone over the structure of a closure: trailing closure syntax, unowned self, a parameter being passed in, then the need for self. to make capturing clear.

We haven't really talked about the actual content of our closure, because there isn't a lot to it. As a reminder, here's how it looks:

```
let answer = alertController.textFields![0]
self.submitAnswer(answer.text!)
```

The first line force unwraps the array of text fields (it's optional because there might not be any; we can force unwrap because we know we added one), then tells Swift to treat it as a UITextField. The second line pulls out the text from the text field and passes it to our (all-new-albeit-empty) submitAnswer() method.

This method needs to check whether the player's word can be made from the given letters. It needs to check whether the word has been used already, because obviously we don't want duplicate words. It also needs to check whether the word is actually a valid English word, because otherwise the user can just type in nonsense.

If all three of those checks pass, submitAnswer() needs to add the word to the objects array, then insert a new row in the table view. We could just use the table view's reloadData() method to force a full reload, but that's not very efficient when we're changing just one row.

Here's our first pass at filling in the submitAnswer() method:

```
func submitAnswer(answer: String) {
    let lowerAnswer = answer.lowercaseString

    if wordIsPossible(lowerAnswer) {
        if wordIsOriginal(lowerAnswer) {
            if wordIsReal(lowerAnswer) {
                objects.insert(answer, atIndex: 0)

                let indexPath = NSIndexPath(forRow: 0,
inSection: 0)
```

```
tableView.insertRowsAtIndexPaths([indexPath],
withRowAnimation: .Automatic)
                }
            }
        }
}
```

Ignore the `wordIsPossible()`, `wordIsOriginal()` and `wordIsReal()` methods for now – let's focus on the rest of the code first.

If a user types `"cease"` as a word that can be made out of our started word `"agencies"`, it's clear that is correct because there is one `"c"`, two `"e"`, one `"a"` and one `"s"`. But what if they type `"Cease"`? Now it has a capital `C`, and `"agencies"` doesn't have a capital `C`. Yes, that's right: strings are case-sensitive, which means `Cease` is not cease is not `CeasE` is not `CeAsE`.

The solution to this is quite simple: all the starter words are lowercase, so when we check the player's answer we immediately lowercase it using its `lowercaseString` property. This is stored in the `lowerAnswer` constant because we want to use it several times.

We then have three `if` statements, one inside another. These are called nested statements, because you nest one inside the other. Only if all three statements are `true` (the word is possible, the word hasn't been used yet, and the word is a real word), does the main block of code execute.

Once we know the word is good, we do three things: insert the new word into our objects array at index `0`. This means `"add it to the start of the array"`, and means that the newest words will appear at the top of the table view.

The next two things are related: we insert a new row into the table view. Given that the table view gets all its data from the objects array, this might seem strange. After all, we just inserted the word into the objects array, so why do we need to insert anything into the table view?

The answer is animation. Like I said, we could just call the `reloadData()` method and have the table do a full reload of all rows, but it means a lot of extra work for one small change, and also causes a jump – the word wasn't there, and now it is.

This can be hard for users to track visually, so using `insertRowsAtIndexPaths()` lets us tell the table view that a new row has been placed at a specific place in the array so that it can animate the new cell appearing. Adding one cell is also significantly easier than having to reload everything, as you might imagine!

There are two quirks here that require a little more detail. First, `NSIndexPath` is something we looked at briefly in `Project 1`, as it contains a section and a row for every item in your table. As with `Project 1` we aren't using sections here, but the row number should equal the position we added the item in the array – position `0`, in this case.

Second, the `withRowAnimation` parameter lets you specify how the row should be animated in. Whenever you're adding and removing things from a table, the `.Automatic` value means "`do whatever is the standard system animation for this change`". In this case, it means "`slide the new row in from the top`".

Your code won't compile yet, because we still have three missing methods that need to be done. Add these three beneath the `submitAnswer()` method to get everything working again:

```
func wordIsPossible(word: String) -> Bool {
    return true
}

func wordIsOriginal(word: String) -> Bool {
    return true
}

func wordIsReal(word: String) -> Bool {
    return true
}
```

We'll look at what that does shortly, then fill it out with lots of real code. But for now, press `Cmd + R` to play back what you have – you should be able to tap the `+` button and enter words into the alert.

Returning values: contains

None of our custom methods have returned values so far. We used the return keyword briefly in Project 4 to exit the web view's `decidePolicyForNavigationAction` method early, but it didn't send any data back. So, let's take things another step forward.

As I said, the return keyword exits a method at any time it's used. If you use return by itself, it exits the method and does nothing else. But if you use return with a value, it sends that value back to whatever called the method.

Before you can send a value back, you need to tell `Swift` that you expect to return a value. `Swift` will automatically check that a value is returned and it's of the right data type, so this is important. We just put in stubs (empty methods that do nothing) for three new methods, each of which returns a value. Let's take a look at one in more detail:

```
func wordIsOriginal(word: String) -> Bool {
    return true
}
```

The method is called `wordIsOriginal()`, and it takes one parameter that's a string. But before the opening brace there's something new: `-> Bool`. This tells `Swift` that the method will return a boolean value, which is the name for a value that can be either `true` or `false`.

The body of the method has just one line of code: `return true`. This is how the return statement is used to send a value back to its caller: we're returning `true` from this method, so the caller can use this method inside an if statement to check for `true` or `false`.

This method can have as much code as it needs in order to evaluate fully whether the word has been used or not, including calling any other methods it needs. We're going to change it so that it calls another method, which will check whether our objects array already contains the word that was provided. Replace its current return true code with this:

```
return !objects.contains(word)
```

There are two new things here. First, `contains()` is a method that checks whether the array specified in parameter 1 (`objects`) contains the value specified in parameter 2 (`word`). If it does contain the value, `contains()` returns `true`; if not, it returns `false`. Second, the `!` symbol. You've seen this before as the way to `force unwrap optional` variables, but here it's something different: it means not.

The difference is small but important: when used before a variable or constant, `!` means `“not”` or `“opposite”`. So if `contains()` returns `true`, `!` flips it around to make it `false`, and vice versa. When used after a variable or constant, `!` means `“force unwrap this optional variable”`.

This is used because our method is called `wordIsOriginal()`, and should return `true` if the word has never been used before. If we had used `return objects.contains(word)`, then it would do the opposite: it would return `true` if the word had been used and `false` otherwise. So, by using `!` we're flipping it around so that the method returns `true` if the word is new.

That's one method down. Next is the `wordIsPossible()`, which also takes a string as its only parameter and returns a `Bool` – `true` or `false`. This one is more complicated, but I've tried to make the algorithm as simple as possible.

How can we be sure that `“cease”` can be made from `“agencies”`, using each letter only once? The solution I've adopted is to loop through every letter in the player's answer, seeing whether it exists in the eight-letter start word we are playing with. If it does exist, we remove the letter from the start word, then continue the loop. So, if we try to use a letter twice, it will exist the first time, but then get removed so it doesn't exist the next time, and the check will fail.

You already met the `rangeOfString()` method in `Project 4`, so this should be straightforward:

```
func wordIsPossible(word: String) -> Bool {
     var tempWord = title!.lowercaseString

     for letter in word.characters {
          if let pos = tempWord.rangeOfString(String(letter))
{
               tempWord.removeAtIndex(pos.startIndex)
          } else {
               return false
          }
     }

     return true
}
```

Our usage of `rangeOfString()` is a little different than from `Project 4`. Remember, `rangeOfString()` returns an optional position of where the item was found – meaning that it might be `nil`. So, we wrap the call into an `if/let` to safely `unwrap the optional`.

The usage is also different because we use `String(letter)` rather than just letter. This is because our for loop is used on a string, and it pulls out every letter in the string as a new data type called `Character` – i.e., a single letter. `rangeOfString()` expects a string, not a character, so we need to create a string from the character using `String(letter)`.

If the letter was found in the string, we use `removeAtIndex()` to remove the used letter from the `tempWord` variable. This is why we need the `tempWord` variable at all: because we'll be removing letters from it so we can check again the next time the loop goes around.

The method ends with `return true`, because this line is reached only if every letter in the user's word was found in the start word no more than once. If any letter isn't found, or is used more than possible, one of the `return false` lines would have been hit, so by this point we're sure the word is good.

`Important`: we have told `Swift` that we are returning a boolean value from this method, and it will check every possible outcome of the code to make sure a boolean value is returned no matter what.

Time for the final method. Replace the current `wordIsReal()` method with this:

```
func wordIsReal(word: String) -> Bool {
    let checker = UITextChecker()
    let range = NSMakeRange(0, word.characters.count)
    let misspelledRange =
checker.rangeOfMisspelledWordInString(word, range: range,
startingAt: 0, wrap: false, language: "en")

    return misspelledRange.location == NSNotFound
}
```

There's a new class here, called `UITextChecker`. This is an `iOS class` that is designed to spot spelling errors, which makes it perfect for knowing if a given word is real or not. We're creating a new instance of the class and putting it into the checker constant for later.

There's a new function call here too, called `NSMakeRange()`. This is used to make a string range, which is a value that holds a start position and a length. We want to examine the whole string, so we use `0` for the start position and the string's length for the length.

Next, we call the `rangeOfMisspelledWordInString()` method of our `UITextChecker` instance. This wants five parameters, but we only care about the first two and the last one: the first parameter is our string, word, the second is our range to scan (the whole string), and the last is the language we should be checking with, where en selects `English`.

Parameters three and four aren't useful here, but for the sake of completeness: parameter three selects a point in the range where the text checker should start scanning, and parameter four lets us set whether the `UITextChecker` should start at the beginning of the range if no misspelled words were found starting from parameter three. Neat, but not helpful here.

Calling `rangeOfMisspelledWordInString()` returns an `NSRange` structure, which tells us where the misspelling was found. But what we care about was whether any misspelling was found, and if nothing was found our `NSRange` will have the special location `NSNotFound`. Usually location would tell you where the misspelling started, but `NSNotFound` is telling us the word is spelled correctly – i.e., it's a valid word.

Here the return statement is used in a new way: as part of an operation involving ==. This is a very common way to code, and what happens is that `==` returns `true` or `false` depending on whether `misspelledRange.location` is equal to `NSNotFound`. That `true` or `false` is then given to return as the return value for the method.

We could have written that same line across multiple lines, but it's not common:

```
if misspelledRange.location == NSNotFound {
     return true
} else {
     return false
}
```

That completes the third of our missing methods, so the project is almost complete. Run it now and give it a thorough test!

Or else what?

There remains one problem to fix with our code, and it's quite a tedious problem. If the word is possible and original and real, we add it to the list of found words then insert it into the table view. But what if the word isn't possible? Or if it's possible but not original? In this case, we reject the word and don't say why, so the user gets no feedback.

So, the final part of our project is to give users feedback when they make an invalid move. This is tedious because it's just adding else statements to all the if statements in `submitAnswer()`, each time configuring a message to show to users.

Here's the adjusted method:

```
func submitAnswer(answer: String) {
     let lowerAnswer = answer.lowercaseString
```

```
    let errorTitle: String
    let errorMessage: String

    if wordIsPossible(lowerAnswer) {
        if wordIsOriginal(lowerAnswer) {
            if wordIsReal(lowerAnswer) {
                objects.insert(answer, atIndex: 0)

                let indexPath = NSIndexPath(forRow: 0,
inSection: 0)

tableView.insertRowsAtIndexPaths([indexPath],
withRowAnimation: .Automatic)

                return
            } else {
                errorTitle = "Word not recognised"
                errorMessage = "You can't just make them
up, you know!"
            }
        } else {
            errorTitle = "Word used already"
            errorMessage = "Be more original!"
        }
    } else {
        errorTitle = "Word not possible"
        errorMessage = "You can't spell that word from '\
(title!.lowercaseString)'!"
    }
```

```
    let ac = UIAlertController(title: errorTitle, message:
errorMessage, preferredStyle: .Alert)
    ac.addAction(UIAlertAction(title: "OK", style: .Default,
handler: nil))
    presentViewController(ac, animated: true, completion:
nil)
}
```

As you can see, every `if` statement now has a matching `else` statement so that the user gets appropriate feedback. All the elses are effectively the same (albeit with changing text): set the values for `errorTitle` and `errorMessage` to something useful for the user. The only interesting exception is the last one, where we use `string interpolation` (remember `Project 2`?) to show the view controller's title as a lowercase string.

If the user enters a valid answer, a call to return forces `Swift` to exit the method immediately once the table has been updated. This is helpful, because at the end of the method there is code to create a new `UIAlertController` with the error title and message that was set, add an OK button with a `nil` handler (i.e., just dismiss the alert), then show the alert. So, this error will only be shown if something went wrong.

This demonstrates one important tip about `Swift` constants: both `errorTitle` and `errorMessage` were declared as constants, which means their value cannot be changed once set. I didn't give either of them an initial value, and that's OK – `Swift` lets you do this as long as you do provide a value before the constants are read, and also as long as you don't try to change the value again later on.

Other than that, your project is done. Go and play!

Wrap up

You've made it this far, so your Swift learning really is starting to come together, and I hope this project has shown you that you can make some pretty advanced things with your knowledge.

In this project, you learned a little bit more about UITableView: how to reload their data and how to insert rows. You also learned how to add text fields to UIAlertController so that you can accept user input. But you also learned some serious core stuff: more about Swift strings, closures, method return values, booleans, NSRange, and more. These are things you're going to use in dozens of projects over your Swift coding career, and things we'll be returning to again and again in this series.

You may already have plans for how you'd like to improve this game, but if not here are four ideas to get you started:

1. Disallow answers that are shorter than three letters. The easiest way to accomplish this is to put a check into wordIsReal() that returns false if the word length is under three letters.
2. Refactor all the else statements we just added so that they call a new method called showErrorMessage(). This should accept an error message and a title, and do all the UIAlertController work from there.
3. Disallow answers that are just the start word. Right now, if the start word is "agencies" the user can just submit "agencies" as an answer, which is too easy – stop them from doing that.
4. Fix our start.txt loading code. If we the pathForResource() call returns nil we load an array containing one word: silkworm.

But what if `pathForResource()` succeeds, but creating an `NSString` using `contentsOfFile` fails? Then the array is empty! Make a new `loadDefaultWords()` method that can be used for both failures.