Project 8: 7 Swifty Words

Overview

Brief: Build a word-guessing game and master strings once and for all.
Learn: addTarget(), enumerate(), countElements(), find(), join(), stringByReplacingOccurrencesOfString(), property observers, range operators.

- Setting up
- Buttons… buttons everywhere.
- Loading a level: addTarget and shuffling arrays
- It's play time: find and join
- Property observers: didSet
- Wrap up

Setting up

This is the final game you'll be making with UIKit; every game after this one will use Apple's SpriteKit library for high-performance 2D drawing. To make this last UIKit effort count, we're going to have a fairly complicated user interface so you can go out with a bang. We're also going to mix in some great new Swift techniques, including property observers, searching through arrays, modulo, array enumeration, ranges and more!

Of course, you're probably wondering what kind of game we're going to make, and I have some bad news for you: it's another word game. But there's good news too: it's a pretty darn awesome word game, based on the popular indie game 7 Little Words. This will also be our first game exclusively targeting iPad, and you'll soon see why – we're using a lot of space in our user interface!

So, go ahead and create a new `Single View Application` project in `Xcode`, this time selecting `iPad` for your device, then save it somewhere. Now go to the project editor and deselect `Portrait` and `Upside Down` orientations.

What's that? You don't know where the project editor is? I'm sure I told you to remember where the project editor was! OK, here's how to find it, one last time, quoted from `Project 6`:

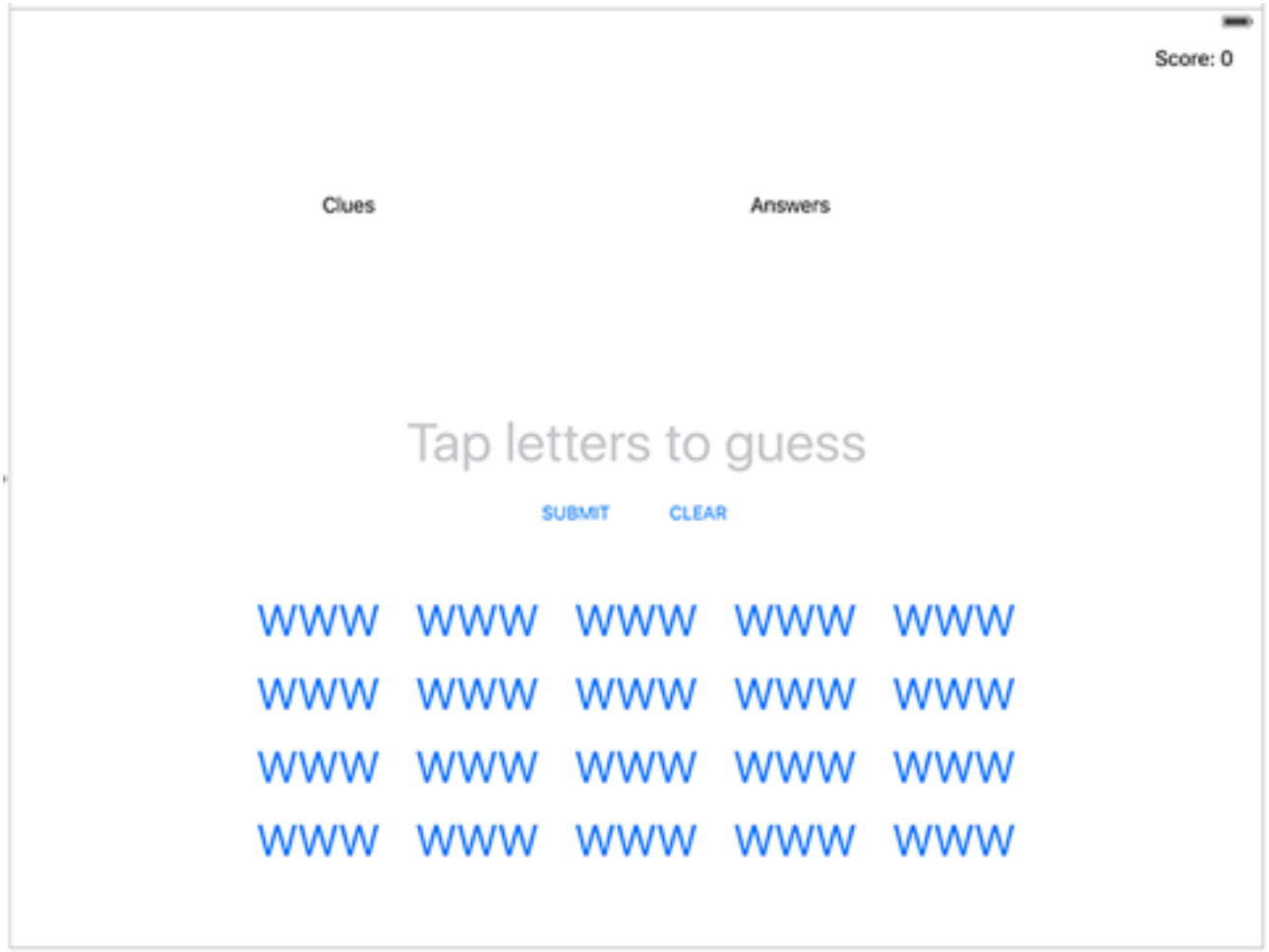
Press `Cmd + 1` to show the project navigator on the left of your `Xcode` window, select your project (it's the first item in the pane), then to the right of where you just clicked will appear another pane showing "PROJECT" and "TARGETS", along with some more information in the center. The left pane can be hidden by clicking the disclosure button in the top-left of the project editor, but hiding it will only make things harder to find, so please make sure it's visible!

This view is called the project editor, and contains a huge number of options that affect the way your app works. You'll be using this a lot in the future, so remember how to get here! Select `Project 2` under `TARGETS`, then choose the `General` tab, and scroll down until you see four checkboxes called `Device Orientation`. You can select only the ones you want to support.

Buttons… buttons everywhere.

Our user interface for this game is going to have two large `UILabels`, one small `UILabel`, one large `UITextField`, twenty (count 'em!) big `UIButtons`, then two small `UIButtons`. This is probably the most complicated user interface we're going to make in this entire series, so don't worry if it takes you 20 minutes or so to put together – the end result is definitely worth it!

The picture below shows how your finished layout should look if you've followed all the instructions. If you're seeing something slightly different, that's OK. If you're seeing something very different, you should probably try again!



Our game is designed for `iPads`, and specifically for `iPads in` landscape orientation. Using a square `Interface Builder` canvas is great for when you want to support a variety of devices where things resize, but here we want one single design with lots of things on. So, we're going to make the canvas the exact right size: go to the file inspector (`Alt + Cmd + 1`) and deselect the checkbox that says, `“Use Size Classes”`.

You'll be asked what size class data you want to keep, so select `iPad` then choose `Disable Size Classes`. Now select your view controller, go to the attributes inspector, then change `Orientation` from `“Inferred”` to

“Landscape”. This gives us a canvas that shaped and sized exactly correctly for an iPad screen, so we're all set to go!

Let's start with the twenty big buttons, because you need to follow my instructions carefully and once these are placed you'll be able to see the big plan.

What you need to do is place twenty UIButtons in a grid that's five across and four down. The top-left button should be at X:200 and Y:470. All buttons should be 120 wide and 60 high; each column should be 130 points apart, and each row should be 60 points apart. That ought to be enough for you to make the entire grid, but for the sake of clarity, this means that:

- The second button on the top row should be at X:330 Y:470.
- The third button on the top row should be at X:460 Y:470 (and so on)
- The first button on the second row should be at X:200 Y:530.
- The second button on the second row should be at X:330 Y:530 (and so on)

Once you've placed all the buttons, click and drag over them so they are all selected, then tap your left cursor key eight times to move every button eight points to the left. On the eighth tap, you should see a long, blue, vertical line appear in the center of the button group, which is telling you the buttons are centered horizontally, so these buttons are placed correctly.

With the buttons still selected, go to the attributes inspector and change Type to be “Custom” and Tag to be 1001. The first one disables an Apple animation that will otherwise cause problems later, and the second change sets the tag for all the buttons simultaneously. You should also click the small T button next to the button font, and in the popover that appears make sure the size is set to 36.

When designing this, I gave these buttons the text `WWW` because that's the largest string they'll need to hold. You should also give them a text color; I chose a shade of blue similar to the `iOS` default.

Now create two more buttons, both `75` wide and `44` high. Place the first at `X: 425 Y:390` and give it the title `SUBMIT`, and place the second at `X:525 Y:390` and give it the title `CLEAR`. That's all the buttons we'll need for the game.

Place a text field and make it `535` wide by `80` high, then position it at `X: 245 Y: 315`. You'll find that you can't resize the text field's height by default, and that's because it has a rounded rectangle border around it that must be an exact size. To change the height you must make the textfield bordlerless: in the identity inspector, choose the first of the four options next to `Border Style`, then you can adjust the height freely.

Give this text field the placeholder text `“Tap letters to guess”`, then give it a nice and big font size – `44` points ought to do. Finally, make its text aligned to the center rather than the left, so everything lines up neatly.

Now create two labels. Make the first one `400` wide by `280` high, with position `X:255 Y:20`, then give it the text `“Clues”`. Make the second one `165` wide by `280` high, and position it at `X:605 Y:20`, then give it the text `“Answers”`. Both of these should be given font size `24`, but where it says `“Number of lines”` set the value to be `0` – that means `“let this text go over as many lines as it needs”`.

Finally, create one last label of width `170` and height `40`, at `X:830` and `Y:20`. Set this to have text alignment right and the text `“Score: 0”`.

That's the layout complete. If you've played any games like `7 Little Words` before, you'll already know exactly how the user interface functions. If not, we'll show seven clues in the label marked `"Clues"`, and each of those clues can be spelled by tapping the letters in the buttons. When a user has spelled the word they want, they click `Submit` to try it out, and if the answer is correct they'll see it in the `Answers` label. Otherwise, that answers label just shows the number of letters in the correct answer.

Before you exit `Interface Builder`, switch to the assistant editor and create four outlets: one for the clue label (call it `cluesLabel`), one for the `Answers` label (call it `answersLabel`), one for the text field (call it `currentAnswer`) and one for the score (call it `scoreLabel`). Please also create two actions: one from the submit button (call it `submitTapped()`) and one from the clear button (call it `clearTapped()`).

That's it! That's the most complicated storyboard you'll make in any project in this entire series. Fortunately, from here on the rest is all coding and lots of fun, so let's get onto the best bit…

Loading a level: addTarget and shuffling arrays

This game asks players to spell seven words out of various letter groups, and each word comes with a clue for them to guess. It's important that the total number of letter groups adds up to `20`, as that's how many buttons you have. I created the first level for you, and it looks like this:

```
HA|UNT|ED: Ghosts in residence
LE|PRO|SY: A Biblical skin disease
TW|ITT|ER: Short but sweet online chirping
```

```
OLI|VER: Has a Dickensian twist
ELI|ZAB|ETH: Head of state, British style
SA|FA|RI: The zoological web
POR|TL|AND: Hipster heartland
```

As you can see, I've used the pipe symbol to split up my letter groups, meaning that one button will have "HA", another "UNT", and another "ED". There's then a colon and a space, followed by a simple clue. This level is in the files for this project you should download from GitHub. You should copy `level1.txt` into your Xcode project as you have done before.

Our first task will be to load the level and configure all the buttons to show a letter group. We're going to need three arrays to handle this: one to store all the buttons, one to store the buttons that are currently being used to spell an answer, and one for all the possible solutions. Further, we need two integers: one to hold the player's score, which will start at `0` but obviously change during play, and one to hold the current level.

So, declare these properties just below the current `@IBOutlets` from Interface Builder:

```
var letterButtons = [UIButton]()
var activatedButtons = [UIButton]()
var solutions = [String]()

var score = 0
var level = 1
```

Now, you'll notice we don't have any `@IBOutlet` references to any of our buttons, and that's entirely intentional: it wouldn't be very smart to create an `@IBOutlet`

for every button. Interface Builder does have a solution to this, called `Outlet Collections`, which are effectively an `IBOutlet` array, but even that solution requires you to Ctrl-drag from every button and quite frankly I don't think you have the patience after spending so much time in `Interface Builder`!

As a result, we're going to take a simple shortcut. And this shortcut will also deal with calling methods when any of the buttons are tapped, so all in all it's a clean and easy solution. The shortcut is this: all our buttons have the tag `1001`, so we can loop through all the views inside our view controller, and modify them only if they have tag `1001`. Add this code to your `viewDidLoad()` method beneath the call to super:

```
for subview in view.subviews where subview.tag == 1001 {
    let btn = subview as! UIButton
    letterButtons.append(btn)
    btn.addTarget(self, action: #selector(letterTapped),
forControlEvents: .TouchUpInside)
}
```

As you can see, `view.subviews` is an array containing all the `UIViews` that are currently placed in our view controller, which is all the buttons and labels, plus that text field. I've used a more enhanced version of a regular for loop that adds a where condition so that the only items inside the loop are subviews with that tag. If we find a view with tag `1001`, we typecast it as a `UIButton` then append it to our buttons array.

We also take this opportunity to use a new method, called `addTarget()`. This is the code version of Ctrl-dragging in a storyboard and it lets us attach a method to the button click. You should remember `.TouchUpInside` from all the button

actions you have made, because that's the event that means the button was tapped.

By adding `#selector(letterTapped)` to each button in code, we're saving ourselves a lot of Ctrl-dragging in `Interface Builder`. When the `letterTapped()` method is called, the button that was tapped will be sent as a parameter, which is perfect for us because we can read the letter group on the button and use it to spell words. But more on that later – for now, we want to finish loading the level.

We're going to isolate level loading into a single level, called `loadLevel()`. This needs to do two things: load and parse our level text file in the format I showed you earlier, then randomly assign letter groups to buttons. In `Project 5` you already learned how to create `Strings` using `contentsOfFile` to load files from disk, and we'll be using that to load our level. In that same project you learned how to use `componentsSeparatedByString()` to split up a string into an array, and we'll use that too.

We'll also need to use the array shuffling code from the `GameplayKit framework` that we've used before. But: there are some new things to learn, honest! First, we'll be using the `enumerate()` method to loop over an array. We haven't used this before, but it's helpful because it passes you each object from an array as part of your loop, as well as that object's position in the array. You're also going to meet the characters.count property of strings, which returns how many letters are in that string.

`There's also a new string method to learn, called stringByReplacingOccurrencesOfString()`. This lets you specify two parameters, and replaces all instances of the first parameter with the second

parameter. We'll be using this to convert `“HA|UNT|ED”` into `HAUNTED` so we have a list of all our solutions.

Before I show you the code, watch out for how I use the method's three variables: `clueString` will store all the level's clues, `solutionString` will store how many letters each answer is (in the same position as the clues), and `letterBits` is an array to store all letter groups: HA, UNT, ED, and so on.

Here's the `loadLevel()` method:

```
func loadLevel() {
    var clueString = ""
    var solutionString = ""
    var letterBits = [String]()

    if let levelFilePath =
NSBundle.mainBundle().pathForResource("level\(level)",
ofType: "txt") {
        if let levelContents = try? String(contentsOfFile:
levelFilePath, usedEncoding: nil) {
            var lines =
levelContents.componentsSeparatedByString("\n")
            lines =
GKRandomSource.sharedRandom().arrayByShufflingObjectsInArray(
lines) as! [String]

            for (index, line) in lines.enumerate() {
                let parts =
line.componentsSeparatedByString(": ")
                let answer = parts[0]
                let clue = parts[1]
```

```
                clueString += "\(index + 1). \(clue)\n"

                let solutionWord =
answer.stringByReplacingOccurrencesOfString("|", withString:
"")
                solutionString += "\
(solutionWord.characters.count) letters\n"
                solutions.append(solutionWord)

                let bits =
answer.componentsSeparatedByString("|")
                letterBits += bits
            }
        }
    }

   // Now configure the buttons and labels
}
```

If you read all that and it made sense first time, great! You can skip over the next few paragraphs and jump to the bit the bold text `“All done!”`. If you read it and only some made sense, these next few paragraphs are for you.

First, the method uses `pathForResource()` and `String's contentsOfFile` to find and load the level string from the disk. String interpolation is used to combine `“level”` with our current level number, making `“level1.txt”`. The text is then split into an array by breaking on the `\n` character (that's line break, remember), then shuffled so that the game is a little different each time.

Our loop uses the `enumerate()` method to go through each item in the lines array. This is different to how we normally loop through an array, but `enumerate()` is helpful here because it tells us where each item was in the array so we can use that information in our clue string. In the code above, `enumerate()` will place the item into the line variable and its position into the index variable.

We already split the text up into lines based on finding `\n`, but now we split each line up based on finding `: `, because each line has a colon and a space separating its letter groups from its clue. We put the first part of the split line into answer and the second part into clue, for easier referencing later.

Now, here's something new: you've already seen how string interpolation can turn `level\(level)` into `"level1"` because the level variable is set to `1`, but here we're adding to the `clueString` variable `using \(index + 1)`. Yes, we're actually doing basic math in our string interpolation. This is needed because the array indexes start from `0`, which looks strange to players, so we add `1` to make it count from `1` to `7`.

Next comes our new string method call, `stringByReplacingOccurrencesOfString()`. We're asking it to replace all instances of `|` with an empty string, so `HA|UNT|ED` will become `HAUNTED`. We then use `characters.count` to get the length of our string then use that in combination with string interpolation to add to our solutions string.

Finally, we make yet another call to `componentsSeparatedByString()` to turn the string `"HA|UNT|ED"` into an array of three elements, then add all three to our `letterBits` array

All done!

Time for some more code: our current `loadLevel()` method ends with a comment saying `// Now configure the buttons and labels`, and we're going to fill that in with the final part of the method. This needs to set the `cluesLabel` and `answersLabel` text, shuffle up our buttons and letter groups, then assign letter groups to buttons.

Before I show you the actual code, there's a new string method to introduce, and it's another long one: `stringByTrimmingCharactersInSet()` removes any letters you specify from the start and end of a string. It's most frequently used with the parameter `.whitespaceAndNewlineCharacterSet()`, which trims spaces, tabs and line breaks, and we need exactly that here because our clue string and solutions string will both end up with an extra line break.

Put this code where the comment was:

```
cluesLabel.text =
clueString.stringByTrimmingCharactersInSet(.whitespaceAndNewl
ineCharacterSet())
answersLabel.text =
solutionString.stringByTrimmingCharactersInSet(.whitespaceAnd
NewlineCharacterSet())

letterBits =
GKRandomSource.sharedRandom().arrayByShufflingObjectsInArray(
letterBits) as! [String]
letterButtons =
GKRandomSource.sharedRandom().arrayByShufflingObjectsInArray(
letterButtons) as! [UIButton]
```

```
if letterBits.count == letterButtons.count {
    for i in 0 ..< letterBits.count {
        letterButtons[i].setTitle(letterBits[i],
forState: .Normal)
    }
}
```

That code uses yet another type of loop, and this time it's a range: `for i in 0 ..< letterButtons.count` means "`count from 0 up to but not including the number of buttons`". This is useful because we have as many items in our `letterBits` array as our `letterButtons` array. Looping from `0` to `19` (inclusive) means we can use the i variable to set a button to a letter group.

The `..<` operator is called the "`half-open range operator`" because it does not include the upper limit. Instead, it counts to one below. There's a closed range operator, `...`, which includes the upper limit, but we don't want that here because an array of `20` items will have numbers `0` to `19`.

Before you run your program, make sure you add a call to `loadLevel()` in your `viewDidLoad()` method. Once that's done, you should be able to see all the buttons and clues configured correctly. Now all that's left is to let the player, well, play.

It's play time: find and join

We need to add three more methods to our view controller in order to finish this game: one to handle letter buttons being tapped, another to handle the current word being cleared, and a third to handle the current word being submitted. The

first two are extremely easy, so let's get those done so we can get onto the serious stuff.

First, we already used the `addTarget()` method in `viewDidLoad()` to make all our letter buttons call the method `letterTapped()`, and you should remember that actually had to specify `letterTapped:` because we want to receive the button that was tapped as a parameter for our method. Add this method now somewhere in your code:

```
func letterTapped(btn: UIButton) {
    currentAnswer.text = currentAnswer.text! +
btn.titleLabel!.text!
    activatedButtons.append(btn)
    btn.hidden = true
}
```

That does three things: gets the text from the title label of the button that was tapped and appends it to the current text of the answer text field, then appends the button to the activatedButtons array, and finally hides the button. We need to force unwrap both the title label and its text, because both might not exist – and yet we know they do.

The `activatedButtons` array is being used to hold all buttons that the player has tapped before submitting their answer. This is important because we're hiding each button as it is tapped, so when the user taps `“Clear”` we need to know which buttons are currently in use so we can re-show them. You already created an empty `@IBAction` method for clear being tapped, so fill it in like this:

```
@IBAction func clearTapped(sender: AnyObject) {
    currentAnswer.text = ""
```

```
    for btn in activatedButtons {
        btn.hidden = false
    }

    activatedButtons.removeAll()
}
```

As you can see, this method removes the text from the current answer text field, unhides all the activated buttons, then removes all the items from the `activatedButtons` array.

That just leaves one final method, and you already created its stub: the `submitTapped()` method for when the player taps the submit button.

This method will use another new function called `indexOf()`, which searches through an array for an item and, if it finds it, tells you its position. The return value is optional so that in situations where nothing is found you won't get a value back, so we need to unwrap its return value carefully.

If the user gets an answer correct, we're going to change the answers label so that rather than saying "7 LETTERS" it says "HAUNTED", so they know which ones they have solved already. The way we're going to do this is delightfully simple: `indexOf()` will tell us which solution matched their word, and that we can use that position to find the matching clue text. All we need to do is split the answer label text up by `\n`, replace the line at the solution position with the solution itself, then re-join the clues label back together.

You've already learned how to use `componentsSeparatedByString()` to split text into an array, and now it's time to meet its counterpart:

`joinWithSeparator()`. This makes an array into a single string, with each array element separated by the string specified in its parameter.

Once that's done, we clear the current answer text field and add one to the score. If the score is evenly divisible by `7`, we know they have found all seven words so we're going to show a `UIAlertController` that will prompt the user to go to the next level.

The “evenly divisible” task is easy to do in `Swift` (and indeed any sensible programming language) thanks to a dedicated modulo operator: `%`. Modulo is division with remainder, so `10 % 3` means “`tell me what number remains when you divide 10 evenly into 3 parts`”. `3` goes into `10` three times (making nine), with remainder `1`, so `10 % 3` is `1`, `11 % 3` is `2`, and `12 % 3` is `0` – i.e., `12` divides perfectly into `3` with no remainder. If score `% 7` is `0`, we know they have answered all seven words correctly.

That's all the parts explained, so here's the final `submitTapped()` method:

```
@IBAction func submitTapped(sender: AnyObject) {
    if let solutionPosition =
solutions.indexOf(currentAnswer.text!) {
        activatedButtons.removeAll()

        var splitClues =
answersLabel.text!.componentsSeparatedByString("\n")
        splitClues[solutionPosition] = currentAnswer.text!
        answersLabel.text =
splitClues.joinWithSeparator("\n")

        currentAnswer.text = ""
```

```
        score += 1

        if score % 7 == 0 {
            let ac = UIAlertController(title: "Well done!",
message: "Are you ready for the next level?", preferredStyle:
.Alert)
            ac.addAction(UIAlertAction(title: "Let's go!",
style: .Default, handler: levelUp))
            presentViewController(ac, animated: true,
completion: nil)
        }
    }
}
```

The `levelUp()` call in there is just to get you started – there isn't a level up here, because I only created one level! But if you wanted to make more levels and continue the game, you'd need a `levelUp()` method something like this:

```
func levelUp(action: UIAlertAction!) {
    level += 1
    solutions.removeAll(keepCapacity: true)

    loadLevel()

    for btn in letterButtons {
        btn.hidden = false
    }
}
```

As you can see, that code clears out the existing solutions array before refilling it inside `loadLevel()`. Then of course you'd need to create `level2.txt`,

`level3.txt` and so on. To get you started, I've made an example `level2.txt` for you inside the `Content` folder – try adding that to the project and see what you think. Any further levels are for you to do – just make sure there's a total of `20` letter groups each time!

Property observers: didSet

There's one last thing to cover before this project is done, and it's really small and really easy: `property observers`.

Right now we have a property called score that is set to `0` when the game is created and increments by one whenever an answer is found. But we don't do anything with that score, so our score label is never updated.

One solution to this problem is to use something like `scoreLabel.text = "Score: \(score)"` whenever the score value is changed, and that's perfectly fine to begin with. But what happens if you're changing the score from several places? You need to keep all the code synchronised, which is unpleasant.

`Swift` has a simple and classy solution called property observers, and it lets you execute code whenever a property has changed. To make them work, you need to declare your data type explicitly (in our case we need an `Int`), then use either `didSet` to execute code when a property has just been set, or `willSet` to execute code before a property has been set.

In our case, we want to add a property observer to our score property so that we update the score label whenever the score value was changed. So, change your score property to this:

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

Note that when you use a property observer like this, you need to explicitly declare its type otherwise `Swift` will complain.

Using this method, any time score is changed by anyone, our score label will be updated. That's it, the project is done!

Wrap up

Yes, it took quite a lot of storyboard work to get this project going, but I hope it has shown you that you can make some great games using just the `UIKit` tools you already know.

Of course, at the same time as making another game, you've made several steps forward in your `iOS development` quest, this time learning about `addTarget()`, `enumerate()`, `count()`, `find()`, `join()`, `stringByReplacingOccurrencesOfString()`, property observers, range operators and the difference between `Swift strings` and `NSString`.

Looking at that list, it should be clear that you are increasingly dealing with specific bits of code (i.e., functions like `find()`) when you're developing `UIKit` projects. This is because you're starting to build up a great repertoire of code, so there is simply less to teach. That's not to say there isn't a lot of new things still to come – in

fact, the next few projects all introduce several big new things – but it does mean your knowledge is starting to mature.

If you're looking to improve this project, see if you can make it deduct points if the player makes an incorrect guess - this is just a matter of extending the `submitAnswer()` method so that if `find()` failed to find the guess then you remove points.