Ray Wenderlich para mim 1 de ago

José, welcome back to the **raywenderlich.com iOS Apprentice Email Course**!

Last time, you completed storing the value of the slider into a variable and showing it via an alert. That's great, but you can still improve on it a little.

What if you decide to set the initial value of the slider in the storyboard to something other than 50, say 1 or 100? Then `currentValue` would be wrong again because the app always assumes it will be 50 at the start. You'd have to remember to also fix the code to give `currentValue` a new initial value.

Take it from me, that kind of thing is hard to remember, especially when the project becomes bigger and you have dozens of view controllers to worry about, or when you haven't looked at the code for weeks.

Improve the slider

Open up the Bull's Eye project where you left it off last time (or download the starter project from the corresponding forum discussion thread), and take a look at Xcode.

To fix this issue once and for all, you're going to do some work inside the `viewDidLoad()` method in **ViewController.swift**. That method currently looks like this:

```
override func viewDidLoad() {
  super.viewDidLoad()
  // Do any additional setup after loading the view,
  // typically from a nib.
}
```

When you created this project based on Xcode's template, Xcode inserted the `viewDidLoad()` method into the source code. You will now add some code to it.

The `viewDidLoad()` message is sent by UIKit immediately after the view controller loads its user interface from the storyboard file. At this point, the view controller isn't visible yet, so this is a good place to set instance variables to their proper initial values.

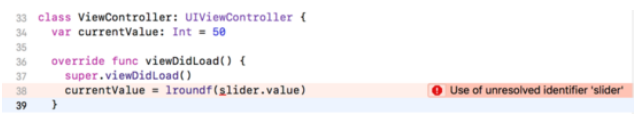
➤ Change `viewDidLoad()` to the following:

```
override func viewDidLoad() {
  super.viewDidLoad()
  currentValue = lroundf(slider.value)
}
```

The idea is that you take whatever value is set on the slider in the storyboard (whether it is 50, 1, 100, or anything else) and use that as the initial value of `currentValue`.

Recall that you need to round off the number, because `currentValue` is an `Int` and integers cannot take decimal (or fractional) numbers.

Unfortunately, Xcode immediately complains about these changes even before you try to run the app.



> **Note:** Xcode tries to be helpful and it analyzes the program for mistakes as you're typing. Sometimes you may see temporary warnings and error messages that will go away when you complete the changes that you're making.
> Don't be too intimidated by these messages; they are only short-lived while the code is in a state of flux.

The above happens because `viewDidLoad()` does not know of anything named `slider`.

Then why did this work earlier, in `sliderMoved()`? Let's take a look at that method again:

```
@IBAction func sliderMoved(_ slider: UISlider) {
  currentValue = lroundf(slider.value)
}
```

Here you do the exact same thing: you round off `slider.value` and put it into `currentValue`. So why does it work here but not in `viewDidLoad()`?

The difference is that in the code above, `slider` is a **parameter** of the `sliderMoved()` method. Parameters are the things inside the parentheses following a method's name. In this case, there's a single parameter named `slider`, which refers to the `UISlider` object that sent this action message.

Action methods can have a parameter that refers to the UI control that triggered the method. This is convenient when you wish to refer to that object in the method, just as you did here (the object in question being the `UISlider`).

When the user moves the slider, the `UISlider` object basically says, "Hey view controller, I'm a slider object and I just got moved. By the way, here's my phone number so you can get in touch with me."

The `slider` parameter contains this "phone number" but it is only valid for the duration of this particular method.

In other words, `slider` is **local**; you cannot use it anywhere else.

Locals

When I first introduced variables, I mentioned that each variable has a certain lifetime, known as its **scope**. The scope of a variable depends on where in your program you defined that variable.

There are three possible scope levels in Swift:

1. **Global scope.** These objects exist for the duration of the app and are accessible from anywhere.
2. **Instance scope.** This is for variables such as `currentValue`. These objects are alive for as long as the object that owns them stays alive.
3. **Local scope.** Objects with a local scope, such as the `slider` parameter of `sliderMoved()`, only exist for the duration of that method. As soon as the execution of the program leaves this method, the local objects are no longer accessible.

Let's look at the top part of `showAlert()`:

```
@IBAction func showAlert() {
  let message = "The value of the slider is: \(currentValue)"

  let alert = UIAlertController(title: "Hello, World",
                                message: message,
                        preferredStyle: .alert)

  let action = UIAlertAction(title: "OK", style: .default,
                             handler: nil)
  . . .
```

Because the `message`, `alert`, and `action` objects are created inside the method, they have local scope. They only come into existence when the `showAlert()` action is performed and cease to exist when the action is done.

As soon as the `showAlert()` method completes, i.e. when there are no more statements for it to execute, the computer destroys the `message`, `alert`, and `action` objects and their storage space is cleared out.

The `currentValue` variable, however, lives on forever… or at least for as long as the `ViewController` does (which is until the user terminates the app). This type of variable is named an **instance variable**, because its scope is the same as the scope of the object instance it belongs to.

In other words, you use instance variables if you want to keep a certain value around, from one action event to the next.

Set up outlets

So, with this newly-gained knowledge of variables and their scope, how do you fix the error that you encountered?

The solution is to store a reference to the slider as a new instance variable, just like you did for `currentValue`. Except that this time, the data type of the variable is not `Int`, but `UISlider`. And you're not using a regular instance variable but a special one called an **outlet**.

➤ Add the following line to **ViewController.swift**:

```
@IBOutlet weak var slider: UISlider!
```

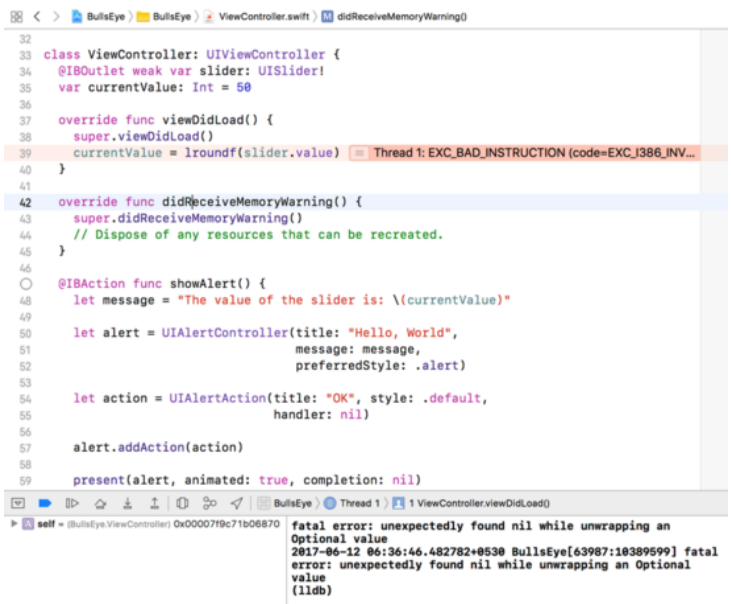
It doesn't really matter where this line goes, just as long as it is somewhere inside the brackets for `class ViewController`. I usually put outlets with the other instance variables - at the top of the class implementation.

This line tells Interface Builder that you now have a variable named `slider` that can be connected to a `UISlider` object. Just as Interface Builder likes to call methods "actions", it calls these variables outlets. Interface Builder doesn't see any of your other variables, only the ones marked with `@IBOutlet`.

Don't worry about `weak` or the exclamation point for now. Why these are necessary will be explained later on. For now, just remember that a variable for an outlet needs to be declared as `@IBOutlet weak var` and has an exclamation point at the end. (Sometimes you'll see a question mark instead; all this hocus pocus will be explained in due time.)

Once you add the `slider` variable, you'll notice that the Xcode error goes away. Does that mean that you can run your app now? Try it and see what happens.

The app crashes on start with an error similar to the following:



So, what happened?

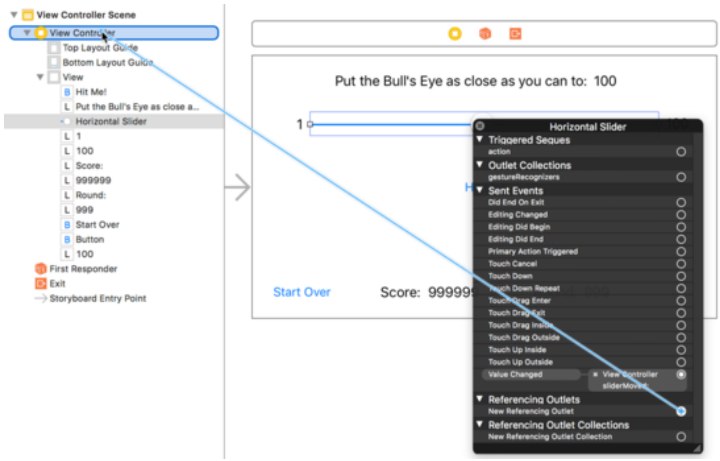
Remember that an outlet has to be **connected** to something in the storyboard. You defined the variable, but you didn't actually set up the connection yet. So, when the app ran and `viewDidLoad()` was called, it tried to find the matching connection in the storyboard and could not - and crashed.

Let's set up the connection in storyboard now.

➤ Open the storyboard. Hold **Control** and click on the **slider**. Don't drag anywhere though, a menu should pop up that shows all the connections for this slider. (Instead of Control-clicking you can also right-click once.)

This popup menu works exactly the same as the Connections inspector. I just wanted to show you this alternative approach.

➤ Click on the open circle next to **New Referencing Outlet** and drag to **View Controller**:



➤ In the popup that appears, select **slider**.

This is the outlet that you just added. You have successfully connected the slider object from the storyboard to the view controller's `slider` outlet.

Now that you have done all this set up work, you can refer to the slider object from anywhere inside the view controller using the `slider` variable.

With these changes in place, it no longer matters what you choose for the initial value of the slider in Interface Builder. When the app starts, `currentValue` will always correspond to that setting.

➤ Run the app and immediately press the Hit Me! button. It correctly says: "The value of the slider is: 50". Stop the app, go into Interface Builder and change the initial value of the slider to something else, say, 25. Run the app again and press the button. The alert should read 25 now.

> **Note:** When you change the slider value, (or the value in any Interface Builder field), remember to tab out of field when you make a change. If you make the change bur your cursor remains in the field, the change might not take effect. This is something which can trip you up often :]

Put the slider's starting position back to 50 when you're done playing.

> **Challenge:** Give `currentValue` an initial value of 0 again. Its initial value is no longer important – it will be overwritten in `viewDidLoad()` anyway – but Swift demands that all variables always have some value and 0 is as good as any.

Comments

You've seen green text that begin with `//` a few times now. As I explained earlier briefly, these are comments. You can write any text you want after the `//` symbol as the compiler will ignore such lines from the `//` to the end of the line completely.

```
// I am a comment! You can type anything here.
```

Anything between the `/*` and `*/` markers is considered a comment as well. The difference between `//` and `/* */` is that the former only works on a single line, while the latter can span multiple lines.

```
/*
   I am a comment as well!
   I can span multiple lines.
 */
```

The `/* */` comments are often used to temporarily disable whole sections of source code, usually when you're trying to hunt down a pesky bug, a practice known as "commenting out". (You can use the **Cmd-/** keyboard shortcut to comment/uncomment the currently selected lines, or if you have nothing selected, the current line.)

The best use for comment lines is to explain how your code works. Well-written source code is self-explanatory but sometimes additional clarification is useful. Explain to whom? To yourself, mostly.

Unless you have the memory of an elephant, you'll probably have forgotten exactly how your code works when you look at it six months later. Use comments to jog your memory.

Generate the random number

You still have quite a ways to go before the game is playable. So, let's get on with the next item on the list: generating a random number and displaying it on the screen.

Random numbers come up a lot when you're making games because games often need to have some element of unpredictability. You can't really get a computer to generate numbers that are truly random and unpredictable, but you can employ a **pseudo-random generator** to spit out numbers that at least appear to be random. You'll use my favorite, `arc4random_uniform()`.

Before you generate the random value though, you need a place to store it.

➤ Add a new variable at the top of **ViewController.swift**, with the other variables:

```
var targetValue: Int = 0
```

If you don't tell the compiler what kind of variable `targetValue` is, then it doesn't know how much storage space to allocate for it, nor can it check if you're using the variable properly everywhere.

Variables in Swift must always have a value, so here you give it the initial value 0. That 0 is never used in the game; it will always be overwritten by the random value you'll generate at the start of the game.

I hope the reason is clear why you made `targetValue` an instance variable.

You want to calculate the random number in one place – like in `viewDidLoad()` – and then remember it until the user taps the button, in `showAlert()` when you have to check this value against what the user selected.

Next, you need to generate the random number. A good place to do this is when the game starts.

➤ Add the following line to `viewDidLoad()` in **ViewController.swift**:

```
targetValue = 1 + Int(arc4random_uniform(100))
```

The complete `viewDidLoad()` should now look like this:

```
override func viewDidLoad() {
  super.viewDidLoad()
  currentValue = lroundf(slider.value)
  targetValue = 1 + Int(arc4random_uniform(100))
}
```

What did you do here? You call the `arc4random_uniform()` function to get an arbitrary integer (whole number) between 0 and 99.

Why is the highest value 99 when the code says 100, you ask? That is because `arc4random_uniform()` treats the upper limit as exclusive. It only goes up-to 100, not up-to-and-including. To get a number that is truly in the range 1 - 100, you add 1 to the result of `arc4random_uniform()`.

Display the random number

➤ Change `showAlert()` to the following:

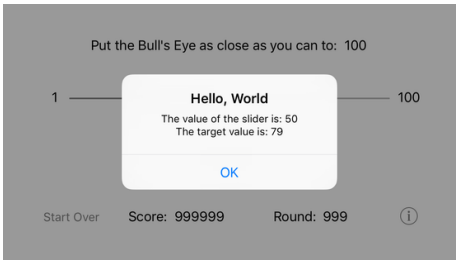
```
@IBAction func showAlert() {
  let message = "The value of the slider is: \(currentValue)" +
                "\nThe target value is: \(targetValue)"

  let alert = . . .
}
```

Tip: Whenever you see `. . .` in a source code listing I mean that as shorthand for: this part didn't change. Don't go replacing the existing code with an actual ellipsis! :]

You've simply added the random number, which is now stored in `targetValue`, to the message string. This should look familiar to you by now: the `\(targetValue)` placeholder is replaced by the actual random number.

The `\n` character sequence is new. It means that you want to insert a special "new line" character at that point, which will break up the text into two lines so the message is a little easier to read.

➤ Run the app and try it out!



> **Note:** Earlier you've used the + operator to add two numbers together (just like how it works in math) but here you're also using + to glue different bits of text into one big string.
> Swift allows the use of the same operator for different tasks, depending on the data types involved. If you have two integers, + adds them up. But with two strings, + concatenates, or combines, them into a longer string.

Bull's Eye is coming along nicely! Next time, you'll update the game to show the target number in the label at the top of the screen.

Talk to you again soon! :]

- Ray

If you'd like to stop receiving the iOS Apprentice Email Course but still stay subscribed to raywenderlich.com Weekly, click here.
