Asynchronous Code | iOS Apprentice Part 10

Ray Wenderlich para mim 11:07

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

There is something that currently bothers me about Bull's Eye. You may have noticed it too…

As soon as you tap the Hit Me! button and the alert pops up, the slider immediately jumps back to its center position, the round number increments, and the target label already gets the new random number.

What happens is that the new round already gets started while you're still watching the results of the last round. That's a little confusing (and annoying).

It would be better to wait on starting the new round until **after** the player has dismissed the alert popup. Only then is the current round truly over.

Note: Prefer learning via video? You might like to check out videos #32-33 of the free video version of this course, which correspond to this email.

Asynchronous code execution

Maybe you're wondering why this isn't already happening? After all, in `showAlert()` you only call `startNewRound()` after you've shown the alert popup:

```
@IBAction func showAlert() {
  . . .
  let alert = UIAlertController(. . .)
  let action = UIAlertAction(. . .)
  alert.addAction(action)

  // Here you make the alert visible:
  present(alert, animated: true, completion: nil)

  // Here you start the new round:
  startNewRound()
}
```

Contrary to what you might expect, `present(alert:animated:completion:)` doesn't hold up execution of the rest of the method until the alert popup is dismissed. That's how alerts on other platforms tend to work, but not on iOS.

Instead, `present(alert:animated:completion:)` puts the alert on the screen and immediately returns control to the next line of code in the method. The rest of the `showAlert()` method is executed right away, and the new round already starts before the alert popup has even finished animating.

In programmer-speak, alerts work **asynchronously**. We'll talk much more about that in a later chapter, but what it means for you right now is that you don't know in advance when the alert will be done. But you can bet it will be well after `showAlert()` has finished.

Alert event handling

So, if your code execution can't wait in `showAlert()` until the popup is dismissed, then how do you wait for it to close?

The answer is simple: events! As you've seen, a lot of the programming for iOS involves waiting for specific events to occur – buttons being tapped, sliders being moved, and so on. This is no different. You have to wait for the "alert dismissed" event somehow. In the mean time, you simply do nothing.

Here's how it works:

For each button on the alert, you have to supply a `UIAlertAction` object. This object tells the alert what the text on the button is – "OK" – and what the button looks like (you're using the default style here):

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

The third parameter, `handler`, tells the alert what should happen when the button is pressed. This is the "alert dismissed" event you've been looking for.

Currently `handler` is `nil`, which means nothing happens. To change this, you'll need to give the `UIAlertAction` some code to execute when the button is tapped. When the user finally taps OK, the alert will remove itself from the screen and jump to your code. That's your cue to take it from there.

This is also known as the **callback** pattern. There are several ways this pattern manifests on iOS. Often you'll be asked to create a new method to handle the event. But here you'll use something new: a **closure**.

To fix this, 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 open up **ViewController.swift**.

➤ Change the bottom bit of `showAlert()` to:

```
@IBAction func showAlert() {
  . . .
  let alert = UIAlertController(. . .)

  let action = UIAlertAction(title: "OK", style: .default,
                             handler: { action in
                                  self.startNewRound()
                             })

  alert.addAction(action)
  present(alert, animated: true, completion: nil)
}
```

Two things have happened here:

1. You removed the call to `startNewRound()` from the bottom of the method. (Don't forget this part!)

2. You placed it inside a block of code that you gave to `UIAlertAction`'s `handler` parameter.

Such a block of code is called a closure. You can think of it as a method without a name. This code is not performed right away. Rather, it's performed only when the OK button is tapped. This particular closure tells the app to start a new round (and update the labels) when the alert is dismissed.

➤ Run it and see for yourself. I think the game feels a lot better this way.

Self
You may be wondering why in the handler block you did `self.startNewRound()` instead of just writing `startNewRound()` like before.
The `self` keyword allows the view controller to refer to itself. That shouldn't be too strange a concept. When you say, "I want ice cream," you use the word "I" to refer to yourself. Similarly, objects can talk about (or to) themselves as well.
Normally you don't need to use `self` to send messages to the view controller, even though it is allowed. The exception: inside closures you **do** have to use `self` to refer to the view controller.
This is a rule in Swift. If you forget `self` in a closure, Xcode doesn't want to build your app (try it out). This rule exists because closures can "capture" variables, which comes with surprising side effects. You'll learn more about that in later chapters.

Start over

No, you’re not going to throw away the source code and start this project all over! I’m talking about the game’s “Start Over” button. This button is supposed to reset the score and start over from the first round.

One use of the Start Over button is for playing against another person. The first player does ten rounds, then the score is reset and the second player does ten rounds. The player with the highest score wins.

And now, it's time for a challenge!

Try to implement the Start Over button on your own. You’ve already seen how you can make the view controller react to button presses, and you should be able to figure out how to change the `score` and `round` variables.

Challenge: If you haven't already, stop reading this email and implementing the Start Over button. If you'd like to compare what you wrote with ours, stop by the discussion thread on our forums. We'd love to see what you came up with! :]

If you finished the challenge, congrats! Now your game is pretty polished and your task list is getting ever shorter :]

- Ray