# Chapter 5: Polish

By Fahim Farook and Matthijs Hollemans

At this point, your game is fully playable. The gameplay rules are all implemented and the logic doesn't seem to have any big flaws. As far as I can tell, there are no bugs either. But there's still some room for improvement.

This chapter will cover the following:

- **Tweaks:** Small UI tweaks to make the game look and function better.
- **The alert:** Updating the alert view functionality so that the screen updates *after* the alert goes away.
- **Start over:** Resetting the game to start afresh.

## Tweaks

Obviously, the game is not very pretty yet and you will get to work on that soon. In the mean time, there are a few smaller tweaks you can make.

### The alert title

Unless you already changed it, the title of the alert still says "Hello, World!" You could give it the name of the game, *Bull's Eye*, but I have a better idea. What if you change the title depending on how well the player did?

If the player put the slider right on the target, the alert could say: "Perfect!" If the slider is close to the target but not quite there, it could say, "You almost had it!" If the player is way off, the alert could say: "Not even close..." And so on. This gives the player a little more feedback on how well they did.

**Exercise:** Think of a way to accomplish this. Where would you put this logic and how would you program it? Hint: there are an awful lot of "if's" in the preceding sentences.

The right place for this logic is `showAlert()`, because that is where you create the `UIAlertController`. You already do some calculations to create the message text and now you will do something similar for the title text.

➤ Here is the changed method in its entirety - replace the existing method with it:

```
@IBAction func showAlert() {
  let difference = abs(targetValue - currentValue)
  let points = 100 - difference
  score += points

  // add these lines
  let title: String
  if difference == 0 {
    title = "Perfect!"
  } else if difference < 5 {
    title = "You almost had it!"
  } else if difference < 10 {
    title = "Pretty good!"
  } else {
    title = "Not even close..."
  }

  let message = "You scored \(points) points"

  let alert = UIAlertController(title: title,  // change this
                                message: message,
                                preferredStyle: .alert)

  let action = UIAlertAction(title: "OK", style: .default,
                             handler: nil)
  alert.addAction(action)
  present(alert, animated: true, completion: nil)

  startNewRound()
}
```

You create a new local string named `title`, which will contain the text that is set for the alert title. Initially, this `title` doesn't have any value. (We'll discuss the `title` variable and how it is set up a bit more in detail just a little further on.)

To decide which title text to use, you look at the difference between the slider position and the target:

- If it equals 0, then the player was spot-on and you set `title` to "Perfect!" .
- If the difference is less than 5, you use the text "You almost had it!"

- A difference less than 10 is “Pretty good!”
- However, if the difference is 10 or greater, then you consider the player’s attempt “Not even close…”

Can you follow the logic here? It’s just a bunch of `if` statements that consider the different possibilities and choose a string in response.

When you create the `UIAlertController` object, you now give it this `title` string instead of a fixed text.

## Constant initialization

In the above code, did you notice that `title` was declared explicitly as being a `String` value? And did you ask yourself why type inference wasn't used there instead? Also, you might have noticed that `title` is actually a constant and yet the code appears to set its value in multiple places. How does that work?

The answer to all of these questions lies in how constants (or `let` values, if you prefer) are initialized in Swift.

You could certainly have used type inference to declare the type for `title` by setting the initial declaration to:

```
let title = ""
```

But do you see the issue there? Now you've actually set the value for `title` and since it's a constant, you can't change the value again. So, the following lines where the `if` condition logic sets a value for `title` would now throw a compiler error since you are trying to set a value to a constant which already has a value. (Go on, try it in your own project! You know you want to … :])

One way to fix this would be to declare `title` as a variable rather than a constant. Like this:

```
var title = ""
```

The above would work fine, and the compiler error would go away and everything would work fine. But you've got to ask yourself, do you really need a variable there? Or, would a constant do? I personally prefer to use constants where possible since they have less risk of unexpected side-effects because the value was accidentally changed in some fashion - for example, because one of your team members changed the code to use a variable that you had originally depended on being unchanged. That is why the code was written the way it was. But you can decide to carve out your own path since either approach would work.

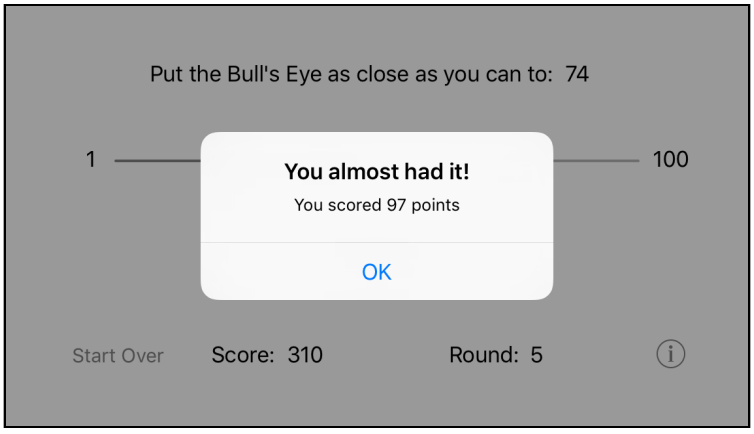
But if you do declare `title` as a constant, how is it that your code above assigns multiple values to it? The secret is in the fact that while there are indeed multiple values being assigned to `title` , only one value would be assigned per each call to `showAlert` since the branches of an `if` condition are mutually exclusive. So, since `title` starts out without a value (the `let title: String` line only assigns a type, not a value), as long as the code ensures that `title` would always be initialized to a value before the value stored in `title` is accessed, the compiler will not complain.

Again, you can test for this by removing the `else` condition in the block of code where a value is assigned to `title`. Since an `if` condition is only one branch of a test, you need an `else` branch in order for the tests (and the assignment to `title`) to be exhaustive. So, if you remove the `else` branch, Xcode will immediately complain with an error like: "Constant 'title' used before being initialized".

```
59      let title: String
60      if difference == 0 {
61        title = "Perfect!"
62      } else if difference < 5 {
63        title = "You almost had it!"
64      } else if difference < 10 {
65        title = "Pretty good!"
66  //    } else {
67  //      title = "Not even close..."
68      }
69
70      let message = "You scored \(points) points"
71
72      let alert = UIAlertController(title: title,      Constant 'title' used before being initialized
73                                    message: message,
74                                    preferredStyle: .alert)
```

*A constant needs to be initialized exhaustively*

Run the app and play the game for a bit. You'll see that the title text changes depending on how well you're doing. That `if` statement sure is handy!




*The alert with the new title*

## Bonus points

**Exercise:** Give players an additional 100 bonus points when they get a perfect score. This will encourage players to really try to place the bull's eye right on the target. Otherwise, there isn't much difference between 100 points for a perfect score and 98 or 95 points if you're close but not quite there.

Now there is an incentive for trying harder – a perfect score is no longer worth just 100 but 200 points! Maybe you can also give the player 50 bonus points for being just one off.

➤ Here is how I would have made these changes:

```
@IBAction func showAlert() {
  let difference = abs(targetValue - currentValue)
  var points = 100 - difference     // change let to var

  let title: String
  if difference == 0 {
    title = "Perfect!"
    points += 100                   // add this line
  } else if difference < 5 {
    title = "You almost had it!"
    if difference == 1 {            // add these lines
      points += 50
    }
  } else if difference < 10 {
    title = "Pretty good!"
  } else {
    title = "Not even close..."
  }
  score += points                   // move this line here
  . . .
}
```

You should notice a few things:

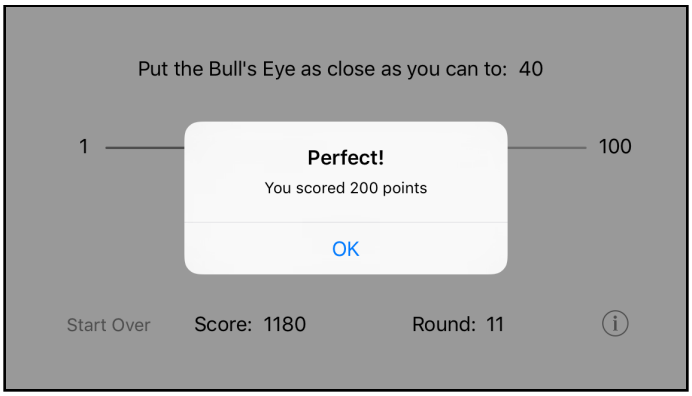
- In the first `if` you'll see a new statement between the curly brackets. When the difference is equal to zero, you now not only set `title` to "Perfect!" but also award an extra 100 points.
- The second `if` has changed too. There is now an `if` inside another `if`. Nothing wrong with that! You want to handle the case where `difference` is 1 in order to give the player bonus points. That happens inside the new `if` statement.

  After all, if the difference is more than 0 but less than 5, it could be 1 (but not necessarily all the time). Therefore, you perform an additional check to see if the difference truly was 1, and if so, add 50 extra points.

- Because these new `if` statements add extra points, `points` can no longer be a constant; it now needs to be a variable. That's why you changed it from `let` to `var`.
- Finally, the line `score += points` has moved below the `if`s. This is necessary because the app updates the `points` variable inside those `if` statements (if the conditions are right) and you want those additional points to count towards the final score.

If your code is slightly different, then that's fine too, as long as it works! There is often more than one way to program something, and if the results are the same, then any approach is equally valid.

➤ Run the app to see if you can score some bonus points!




*Raking in the points...*

## Local variables recap

I would like to point out once more the difference between local variables and instance variables. As you should know by now, a local variable only exists for the duration of the method that it is defined in, while an instance variable exists as long as the view controller (or any object that owns it) exists. The same thing is true for constants.

In `showAlert()`, there are six locals and you use three instance variables:

```
let difference = abs(targetValue - currentValue)
var points = 100 - difference
let title = . . .
score += points
let message = . . .
let alert = . . .
let action = . . .
```

**Exercise:** Point out which are the locals and which are the instance variables in the `showAlert()` method. Of the locals, which are variables and which are constants?

Locals are easy to recognize, because the first time they are used inside a method their name is preceded with `let` or `var`:

```
let difference = . . .
var points = . . .
let title = . . .
let message = . . .
let alert = . . .
let action = . . .
```

This syntax creates a new variable (`var`) or constant (`let`). Because these variables and constants are created inside the method, they are locals.

Those six items – `difference`, `points`, `title`, `message`, `alert`, and `action` – are restricted to the `showAlert()` method and do not exist outside of it. As soon as the method is done, the locals cease to exist.

You may be wondering how `difference`, for example, can have a different value every time the player taps the Hit Me button, even though it is a constant – after all, aren't constants given a value just once, never to change afterwards?

Here's why: each time a method is invoked, its local variables and constants are created anew. The old values have long been discarded and you get brand new ones.

When `showAlert()` is called, it creates a completely new instance of `difference` that is unrelated to the previous one. That particular constant value is only used until the end of `showAlert()` and then it is discarded.

The next time `showAlert()` is called after that, it creates yet another new instance of `difference` (as well as new instances of the other locals `points`, `title`, `message`, `alert`, and `action`). And so on... There's some serious recycling going on here!

But inside a single invocation of `showAlert()`, `difference` can never change once it has a value assigned. The only local in `showAlert()` that can change is `points`, because it's a `var`.

The instance variables, on the other hand, are defined outside of any method. It is common to put them at the top of the file:

```
class ViewController: UIViewController {
  var currentValue = 0
  var targetValue = 0
  var score = 0
  var round = 0
```

As a result, you can use these variables inside any method, without the need to declare them again, and they will keep their values till the object holding them (the view controller in this case) ceases to exist.

If you were to do this:

```
@IBAction func showAlert() {
  let difference = abs(targetValue - currentValue)
  var points = 100 - difference

  var score = score + points       // doesn't work!
  . . .
}
```

Then things wouldn't work as you'd expect them to. Because you now put `var` in front of `score`, you have made it a new local variable that is only valid inside this method.

In other words, this won't add `points` to the *instance variable* `score` but to a new *local variable* that also happens to be named `score`. The instance variable `score` never gets changed, even though it has the same name.

Obviously that is not what you want to happen here. Fortunately, the above won't even compile. Swift knows there's something fishy about that line.

> **Note:** To make a distinction between the two types of variables, so that it's always clear at a glance how long they will live, some programmers prefix the names of instance variables with an underscore.
>
> They would name the variable `_score` instead of just `score`. Now there is less confusion because names beginning with an underscore won't be mistaken for being locals. This is only a convention. Swift doesn't care one way or the other how you spell your instance variables.
>
> Other programmers use different prefixes, such as "m" (for member) or "f" (for field) for the same purpose. Some even put the underscore *behind* the variable name. Madness!

# The alert

There is something that bothers me about the game. 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.

## 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*.

➤ 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.

**Exercise:** 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.

How did you do? If you got stuck, then follow the instructions below.

## The new method

First, add a method to **ViewController.swift** that starts a new game. I suggest you put it near `startNewRound()` because the two are conceptually related.

➤ Add the new method:

```
func startNewGame() {
  score = 0
  round = 0
  startNewRound()
}
```

This method resets `score` and `round` to zero, and starts a new round as well.

Notice that you set `round` to 0 here, not to 1. You use 0 because incrementing the value of `round` is the first thing that `startNewRound()` does.

If you were to set `round` to 1, then `startNewRound()` would add another 1 to it and the first round would actually be labeled round 2.

So, you begin at 0, let `startNewRound()` add one and everything works great.

(It's probably easier to figure this out from the code than from my explanation. This should illustrate why we don't program computers in English.)

You also need an action method to handle taps on the Start Over button. You could write a new method like the following:

```
@IBAction func startOver() {
  startNewGame()
}
```

But you'll notice that this method simply calls the previous method you added :] So, why not cut out the middleman? You can simply change the method you added previously to be an action instead, like this:

```
@IBAction func startNewGame() {
  score = 0
  round = 0
  startNewRound()
}
```

You could follow either of the above approaches since both are valid. Personally, I like to have less code since that means there's less stuff to maintain (and less of a chance of screwing something up :]). Sometimes, there could also be legitimate reasons for having a seperate action method which calls your own method, but in this particular case, it's better to keep things simple.

Just to keep things consistent, in `viewDidLoad()` you should replace the call to `startNewRound()` with `startNewGame()`. Because `score` and `round` are already 0 when the app starts, it won't really make any difference to how the app works, but it does make the intention of the source code clearer. (If you wonder if you can call an `IBAction`

method directly instead of hooking it up to an action in the storyboard, yes, you certainly can do so.)

➤ Make this change:

```
override func viewDidLoad() {
  super.viewDidLoad()
  startNewGame()        // this line changed
}
```

## Connect the outlet

Finally, you need to connect the Start Over button to the action method.

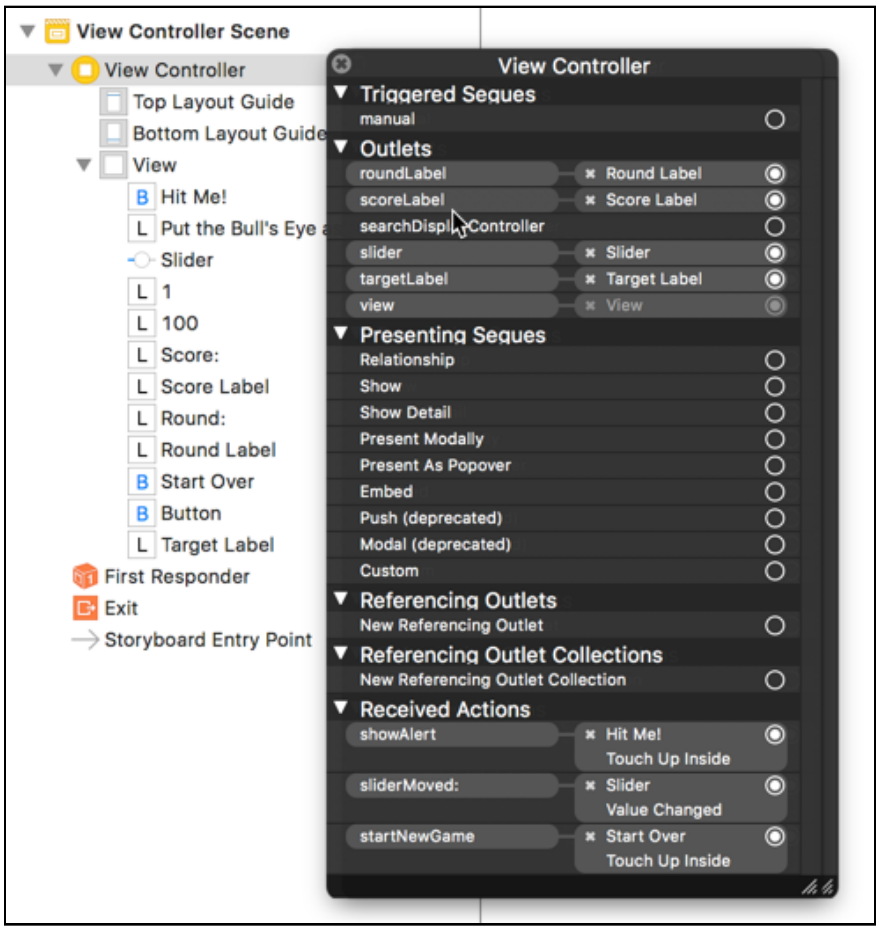
➤ Open the storyboard and Control-drag from the **Start Over** button to **View Controller**. Let go of the mouse button and pick **startNewGame** from the popup if you opted to have `startNewGame()` as the action method. Otherwise, pick the name of your action method .

That connects the button's Touch Up Inside event to the action you have just defined.

➤ Run the app and play a few rounds. Press Start Over and the game puts you back at square one.

Tip: If you're losing track of what button or label is connected to what method, you can click on **View Controller** in the storyboard to see all the connections that you have made so far.

You can either right-click on View Controller to get a popup, or simply view the connections in the **Connections inspector**. This shows all the connections for the view controller.




*All the connections from View Controller to the other objects*

Now your game is pretty polished and your task list is getting ever shorter :]

You can find the project files for the current version of the app under **05 - Polish** in the Source Code folder.