IOS APP 1: PART 3 & PART 4

PREPARED BY FARHAJ AHMED

iOS App 1: Part 3 and Part 4

You've built the user interface for *Bull's Eye* and you know how to find the current position of the slider. That already knocks quite a few items off the to-do list. This chapter takes care of a few other items from the to-do list and covers the following items:

- **Improve the slider**: Set the initial slider value (in code) to be whatever value was set in the storyboard instead of assuming an initial value.
- **Generate the random number**: Generate the random number to be used as the target by the game.
- **Add rounds to the game**: Add the ability to start a new round of the game.
- **Display the target value**: Display the generated target number on screen.

Improving the slider

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.

iOS App 1: Part 3 and Part 4

Getting the initial slider value

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.

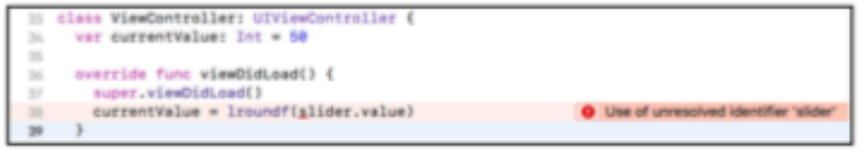
➤ Change `viewDidLoad()` to the following:

```
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.



Xcode error message about missing identifier

> **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.

iOS App 1: Part 3 and Part 4

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 they 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.

Setting up outlets

So, with this newly gained knowledge of variables and their scopes, 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:

App crash when outlet is not connected

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.

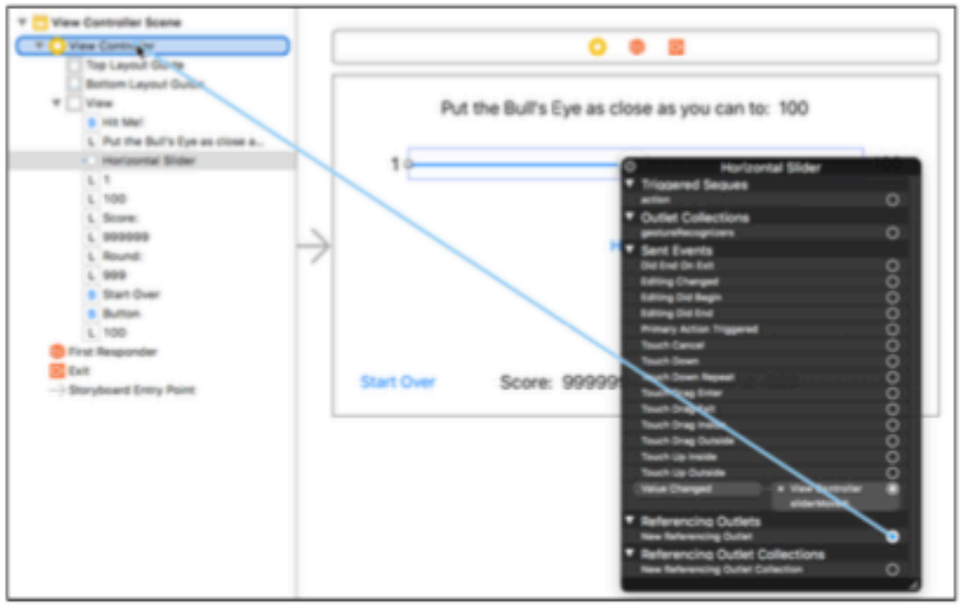
Let's set up the connection in storyboard now.

➤ 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 pop-up menu works exactly the same as the Connections inspector. I just wanted to show you this alternative approach.

iOS App 1: Part 3 and Part 4

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



Connecting the slider to the outlet

➤ In the pop-up 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 but 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.

> **Exercise**: 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.

Commenting

You've probably noticed the green text that begins 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.

Generating 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.

iOS App 1: Part 3 and Part 4

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.

With previous versions of Swift, you had to use external methods such as `arc4random_uniform()`, but as of Swift 4.2, Swift numeric types such as `Int` have the built-in ability to generate random numbers. How handy, right?

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 = 0
```

You might wonder why we didn't specify the type of the `targetValue` variable, similar to what we'd done earlier for `currentValue`. This is because Swift is able to *infer* the type of variables if it has enough information to work with. Here, for example, you initialize `targetValue` with 0 and, since 0 is an integer value, the compiler knows that `targetValue` will be of type `Int`.

We'll discuss Swift type inference again later on but, for the time being, the important point is that `targetValue` is initialized to 0. That 0 is never used in the game; it will always be overwritten by the random value that 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 the user selection.

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 = Int.random(in: 1...100)
```

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

```
override func viewDidLoad() {
  super.viewDidLoad()
  currentValue = lroundf(slider.value)
  targetValue = Int.random(in: 1...100)
}
```

iOS App 1: Part 3 and Part 4

What are you doing here? You call the `random()` function built into `Int` to get an arbitrary integer (whole number) between 1 and 100. The `1...100` part is known as a *closed range* wherein you specify a starting value and an ending value to specify a range of numbers. The `...` part indicates that you want the range to include the closing value (100), but if you wanted a range without the final value, then you would specify the range as `1..<100` and would get only values from 1 to 99.

All clear? Onwards!

Displaying 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 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. The + is also new but is simply used here to combine two strings. We could just as easily have written it as a single long string, but it might not have looked as good to the reader.

➤ Run the app and try it out!

The alert shows the target value on a new line

iOS App 1: Part 3 and Part 4

Note: Earlier, you 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.

Programming languages often use the same symbols for different purposes, depending on the context. After all, there are only so many symbols to go around!

Adding rounds to the game

If you press the Hit Me! button a few times, you'll notice that the random number never changes. I'm afraid the game won't be much fun that way. This happens because you generate the random number in `viewDidLoad()` and never again afterwards. The `viewDidLoad()` method is only called once when the view controller is created during app startup. The item on the to-do list actually said: "Generate a random number *at the start of each round*". Let's talk about what a round means in terms of this game.

When the game starts, the player has a score of 0 and the round number is 1. You set the slider halfway (to value 50) and calculate a random number. Then you wait for the player to press the Hit Me! button. As soon as they do, the round ends.

You calculate the points for this round and add them to the total score. Then you increment the round number and start the next round. You reset the slider to the halfway position again and calculate a new random number. Rinse, repeat.

Starting a new round

Whenever you find yourself thinking something along the lines of, "At this point in the app we have to do such and such," then it makes sense to create a new method for it. This method will nicely capture that functionality in a self-contained unit of its own.

➤ With that in mind, add the following new method to **ViewController.swift**:

```
func startNewRound() {
  targetValue = Int.random(in: 1...100)
  currentValue = 50
  slider.value = Float(currentValue)
}
```

iOS App 1: Part 3 and Part 4

It doesn't really matter where you put the code, as long as it is inside the `ViewController` implementation (within the class curly brackets), so that the compiler knows it belongs to the `ViewController` object.

It's not very different from what you did before, except that you moved the logic for setting up a new round into its own method, `startNewRound()`. The advantage of doing this is that you can execute this logic from more than one place in your code.

Using the new method

First, you'll call this new method from `viewDidLoad()` to set up everything for the very first round. Recall that `viewDidLoad()` happens just once when the app starts up, so this is a great place to begin the first round.

➤ Change `viewDidLoad()` to:

```
override func viewDidLoad() {
  super.viewDidLoad()
  startNewRound()  // Replace previous code with this
}
```

Note that you've removed some of the existing statements from `viewDidLoad()` and replaced them with just the call to `startNewRound()`.

You will also call `startNewRound()` after the player pressed the Hit Me! button, from within `showAlert()`.

➤ Make the following change to `showAlert()`:

```
@IBAction func showAlert() {
  . . .

  startNewRound()
}
```

The call to `startNewRound()` goes at the very end, right after `present(alert, …)`.

Until now, the methods from the view controller have been invoked for you by UIKit when something happened: `viewDidLoad()` is performed when the app loads, `showAlert()` is performed when the player taps the button, `sliderMoved()` when the player drags the slider, and so on. This is the event-driven model we talked about earlier.

It is also possible to call methods directly, which is what you're doing here. You are sending a message from one method in the object to another method in that same object.

In this case, the view controller sends the `startNewRound()` message to itself in order to set up the new round. Program execution will then switch to that method and execute its statements one-by-one. When there are no more statements in the method, it returns to the calling method and continues with that — either `viewDidLoad()`, if this is the first time, or `showAlert()` for every round after.

Calling methods in different ways

Sometimes, you may see method calls written like this:

```
self.startNewRound()
```

That does the exact same thing as `startNewRound()` without `self.` in front. Recall how I just said that the view controller sends the message to itself. Well, that's exactly what `self` means.

To call a method on an object, you'd normally write:

```
receiver.methodName(parameters)
```

The `receiver` is the object you're sending the message to. If you're sending the message to yourself, then the receiver is `self`. But because sending messages to `self` is very common, you can also leave this special keyword out for many cases.

To be fair, this isn't exactly the first time you've called methods. `addAction()` is a method on `UIAlertController` and `present()` is a method that all view controllers have, including yours.

When you write Swift programs, a lot of what you do is calling methods on objects, because that is how the objects in your app communicate.

The advantages of using methods

I hope you can see the advantage of putting the "new round" logic into its own method. If you didn't, the code for `viewDidLoad()` and `showAlert()` would look like this:

```
override func viewDidLoad() {
  super.viewDidLoad()

  targetValue = Int.random(in: 1...100)
  currentValue = 50
  slider.value = Float(currentValue)
```

```
}

@IBAction func showAlert() {
  . . .

  targetValue = Int.random(in: 1...100)
  currentValue = 50
  slider.value = Float(currentValue)
}
```

Can you see what is going on here? The same functionality is duplicated in two places. Sure, it is only three lines of code but, often, the code you duplicate could be much larger.

And what if you decide to make a change to this logic (as you will shortly)? Then you will have to make the same change in two places.

You might be able to remember to do so if you recently wrote this code and it is still fresh in memory, but, if you have to make that change a few weeks down the road, chances are that you'll only update it in one place and forget about the other.

Code duplication is a big source of bugs. So, if you need to do the same thing in two different places, consider making a new method for it instead of duplicating code.

Naming methods

The name of the method also helps to make it clear as to what it is supposed to be doing. Can you tell at a glance what the following does?

```
targetValue = Int.random(in: 1...100)
currentValue = 50
slider.value = Float(currentValue)
```

You probably have to reason your way through it: "It is calculating a new random number and then resets the position of the slider, so I guess it must be the start of a new round."

Some programmers will use a comment to document what is going on (and you can do thatm too), but, in my opinion, the following is much clearer than the above block of code with an explanatory comment:

```
startNewRound()
```

This line practically spells out for you what it will do. And if you want to know the specifics of what goes on in a new round, you can always look up the `startNewRound()` method implementation.

iOS App 1: Part 3 and Part 4

Well-written source code speaks for itself. I hope I have convinced you of the value of making new methods!

➤ Run the app and verify that it calculates a new random number between 1 and 100 after each tap on the button.

You should also have noticed that, after each round, the slider resets to the halfway position. That happens because `startNewRound()` sets `currentValue` to 50 and then tells the slider to go to that position. That is the opposite of what you did before (you used to read the slider's position and put it into `currentValue`), but I thought it would work better in the game if you start from the same position in each round.

> **Exercise**: Just for fun, modify the code so that the slider does not reset to the halfway position at the start of a new round.

Type conversion

By the way, you may have been wondering what `Float(...)` does in this line:

```
slider.value = Float(currentValue)
```

Swift is a *strongly typed* language, meaning that it is really picky about the shapes that you can put into the boxes. For example, if a variable is an `Int`, you cannot put a `Float`, or a non-whole number, into it, and vice versa.

The value of a `UISlider` happens to be a `Float` — you've seen this when you printed out the value of the slider — but `currentValue` is an `Int`. So the following won't work:

```
slider.value = currentValue
```

The compiler considers this an error. Some programming languages are happy to convert the `Int` into a `Float` for you, but Swift wants you to be explicit about such conversions.

When you say `Float(currentValue)`, the compiler takes the integer number that's stored in `currentValue` and converts it into a new `Float` value that it can pass on to the `UISlider`.

Because Swift is stricter about this sort of thing than most other programming languages, it is often a source of confusion for newcomers to the language. Unfortunately, Swift's error messages aren't always very clear about what part of the code is wrong or why.

Just remember, if you get an error message saying, "Cannot assign value of type 'something' to type 'something else'," then you're probably trying to mix incompatible data types. The solution is to explicitly convert one type to the other — if conversion is allowed, of course — as you've done here.

Displaying the target value

Great, you figured out how to calculate the random number and how to store it in an instance variable, `targetValue`, so that you can access it later.

Now, you are going to show that target number on the screen. Without it, the player won't know what to aim for and that would make the game impossible to win.

Setting up the storyboard

When you set up the storyboard, you added a label for the target value (top-right corner). The trick is to put the value from the `targetValue` variable into this label. To do that, you need to accomplish two things:

1. Create an outlet for the label so you can send it messages.
2. Give the label new text to display.

This will be very similar to what you did with the slider. Recall that you added an `@IBOutlet` variable so you could reference the slider anywhere from within the view controller. Using this outlet variable you could ask the slider for its value, through `slider.value`. You'll do the same thing for the label.

➤ In **ViewController.swift**, add the following line below the other outlet declaration:

```
@IBOutlet weak var targetLabel: UILabel!
```

➤ In **Main.storyboard**, click to select the correct label — the one at the very top that says "100."

➤ Go to the **Connections inspector** and drag from **New Referencing Outlet** to the yellow circle at the top of your view controller in the central scene.

You could also drag to the **View Controller** in the Document Outline — there are many ways to do the same thing in Interface Builder.

Connecting the target value label to its outlet

➤ Select **targetLabel** from the pop-up, and the connection is made.

Displaying the target value via code

➤ On to the good stuff! Add the following method below `startNewRound()` in **ViewController.swift**:

```
func updateLabels() {
  targetLabel.text = String(targetValue)
}
```

You're putting this logic in a separate method because it's something you might use from different places.

The name of the method makes it clear what it does: It updates the contents of the labels. Currently it's just setting the text of a single label, but later on you will add code to update the other labels as well (total score, round number).

The code inside `updateLabels()` should have no surprises for you, although you may wonder why you cannot simply do:

```
targetLabel.text = targetValue
```

The answer again is that you cannot put a value of one data type into a variable of another type — the square peg just won't go in the round hole.

The `targetLabel` outlet references a `UILabel` object. The `UILabel` object has a `text` property, which is a `String` object. So, you can only put `String` values into `text`, but `targetValue` is an `Int`. A direct assignment won't fly because an `Int` and a `String` are two very different kinds of things.

So, you have to convert the `Int` into a `String`, and that is what `String(targetValue)` does. It's similar to what you've done before with `Float(…)`.

Just in case you were wondering, you could also convert `targetValue` to a `String` by using string interpolation, like you've done before:

```
targetLabel.text = "\(targetValue)"
```

Which approach you use is a matter of taste. Either approach will work fine.

Notice that `updateLabels()` is a regular method — it is not attached to any UI controls as an action — so it won't do anything until you actually call it. You can tell because it doesn't say `@IBAction` before `func`.

Action methods vs. normal methods

So what is the difference between an action method and a regular method?

Answer: Nothing.

An action method is really just the same as any other method. The only special thing is the `@IBAction` attribute, which allows Interface Builder to see the method so you can connect it to your buttons, sliders and so on.

Other methods, such as `viewDidLoad()`, don't have the `@IBAction` specifier. This is good because all kinds of mayhem would occur if you hooked these up to your buttons.

This is the simple form of an action method:

```
@IBAction func showAlert()
```

You can also ask for a reference to the object that triggered this action, via a parameter:

```
@IBAction func sliderMoved(_ slider: UISlider)
@IBAction func buttonTapped(_ button: UIButton)
```

But the following method cannot be used as an action from Interface Builder:

```
func updateLabels()
```

That's because it is not marked as `@IBAction` and as a result, Interface Builder can't see it. To use `updateLabels()`, you will have to call it yourself.

iOS App 1: Part 3 and Part 4

Calling the method

The logical place to call `updateLabels()` would be after each call to `startNewRound()`, because that is where you calculate the new target value. So, you could always add a call to `updateLabels()` in `viewDidLoad()` and `showAlert()`, but there's another way, too!

What is this other way, you ask? Well, if `updateLabels()` is always (or at least in your current code) called after `startNewRound()`, why not call `updateLabels()` directly from `startNewRound()` itself? That way, instead of having two calls in two separate places, you can have a single call.

➤ Change `startNewRound()` to:

```
func startNewRound() {
    targetValue = Int.random(in: 1...100)
    currentValue = 50
    slider.value = Float(currentValue)
    updateLabels()  // Add this line
}
```

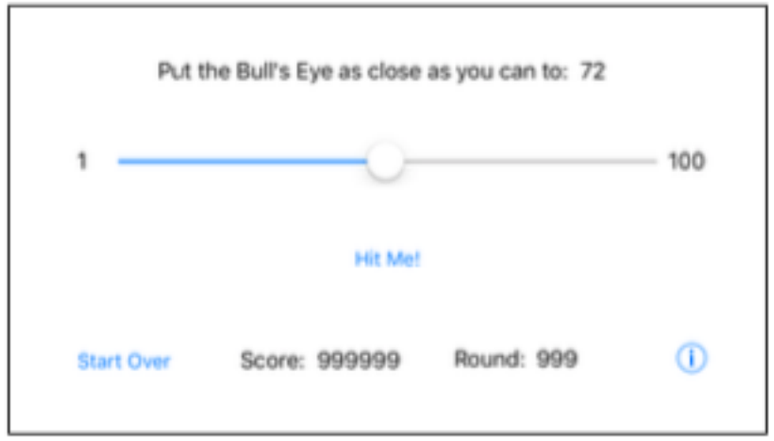
You should be able to type just the first few letters of the method name, like **upd**, and Xcode will show you a list of suggestions matching what you typed. Press **Enter** (or **Tab**) to accept the suggestion (if you are on the right item — or scroll the list to find the right item and then press Enter):

Xcode autocomplete offers suggestions

Also worth noting is that you don't have to start typing the method (or property) name you're looking from the beginning — Xcode uses fuzzy search and typing "dateL" or "label" should help you find "updateLabels" just as easily.

iOS App 1: Part 3 and Part 4

➤ Run the app and you'll actually see the random value on the screen. That should make it a little easier to aim for.



The label in the top-right corner now shows the random value

iOS App 1: Part 3 and Part 4

OK, so you have made quite a bit of progress on the game, and the to-do list is getting ever shorter! So what's next on the list now that you can generate a random number and display it on screen?

A quick look at the task list shows that you now have to "compare the value of the slider to that random number and calculate a score based on how far off the player is." Let's get to it!

This chapter covers the following:

- **Get the difference**: Calculate the difference between the target value and the value that the user selected.
- **Other ways to calculate the difference**: Other approaches to calculating the difference.
- **What's the score?**: Calculate the user's score based on the difference value.
- **The total score**: Calculate the player's total score over multiple rounds.
- **Display the score**: Display the player score on screen.
- **One more round...**: Implement updating the round count and displaying the current round on screen.

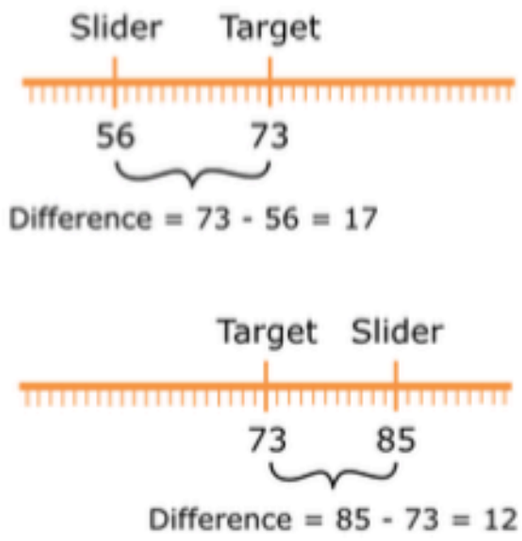
iOS App 1: Part 3 and Part 4

Getting the difference

Now that you have both the target value (the random number) and a way to read the slider's position, you can calculate how many points the player scored.

The closer the slider is to the target, the more points for the player.

To calculate the score for each round, you look at how far off the slider's value is from the target:



Calculating the difference between the slider position and the target value

A simple approach to finding the distance between the target and the slider is to subtract `currentValue` from `targetValue`.

Unfortunately, that gives a negative value if the slider is to the right of the target because now `currentValue` is greater than `targetValue`.

You need some way to turn that negative value into a positive value — or you end up subtracting points from the player's score (unfair!).

Doing the subtraction the other way around — `currentValue` minus `targetValue` — won't always solve things either because, then, the difference will be negative if the slider is to the left of the target instead of the right.

Hmm, it looks like we're in trouble here…

iOS App 1: Part 3 and Part 4

> **Exercise**: How would you frame the solution to this problem if I asked you to solve it in natural language? Don't worry about how to express it in computer language for now, just think it through in plain English.

I came up with something like this:

- *If the slider's value is greater than the target value, then the difference is: slider value minus the target value.*
- *However, if the target value is greater than the slider value, then the difference is: target value minus the slider value.*
- *Otherwise, both values must be equal, and the difference is zero.*

This will always lead to a difference that is a positive number, because you always subtract the smaller number from the larger one.

Do the math:

If the slider is at position 60 and the target value is 40, then the slider is to the right of the target value, and the difference is 60 – 40 = 20.

However, if the slider is at position 10 and the target is 30, then the slider is to the left of the target and has a smaller value. The difference here is 30 – 10 = also 20.

Algorithms

What you've just done is come up with an *algorithm*, which is a fancy term for a series of steps for solving a computational problem. This is only a very simple algorithm, but it is an algorithm nonetheless.

There are many famous algorithms, such as *quicksort* for sorting a list of items and *binary search* for quickly searching through such a sorted list. Other people have already invented many algorithms that you can use in your own programs — that'll save you a lot of thinking!

However, in the programs that you write, you'll probably have to come up with a few algorithms of your own at some time or other. Some are simple such as the one above; others can be pretty hard and might cause you to throw up your hands in despair. But that's part of the fun of programming.

The academic field of Computer Science concerns itself largely with studying algorithms and finding better ones.

iOS App 1: Part 3 and Part 4

You can describe any algorithm in plain English. It's just a series of steps that you perform to calculate something. Often, you can perform that calculation in your head or on paper, the way you did above. But for more complicated algorithms, doing that might take you forever so, at some point, you'll have to convert the algorithm to computer code.

The point I'm trying to make is this: If you ever get stuck and you don't know how to make your program calculate something, take a piece of paper and try to write out the steps in English. Set aside the computer for a moment and think the steps through. How would you perform this calculation by hand?

Once you know how to do that, converting the algorithm to code should be a piece of cake.

The difference algorithm

Getting back to the problem at hand, it is possible you came up with a different way to solve it. I'll show you two alternatives in a minute, but let's convert the above algorithm to computer code first:

```
var difference: Int
if currentValue > targetValue {
  difference = currentValue - targetValue
} else if targetValue > currentValue {
  difference = targetValue - currentValue
} else {
  difference = 0
}
```

The `if` construct is new. It allows your code to make decisions, and it works much like you would expect:

```
if something is true {
  then do this
} else if something else is true {
  then do that instead
} else {
  do something when neither of the above are true
}
```

Basically, you put a *logical condition* after the `if` keyword. If that condition turns out to be true, for example `currentValue` is greater than `targetValue`, then the code in the block between the `{ }` brackets is executed.

However, if the condition is not true, then the computer looks at the `else if` condition and evaluates that. There may be more than one `else if`, and code execution moves one by one from top to bottom until one condition proves to be true.

If none of the conditions are found to be valid, then the code in the final `else` block is executed.

In the implementation of this little algorithm, you first create a local variable named `difference` to hold the result. This will either be a positive whole number or zero, so an `Int` will do:

```
var difference: Int
```

Then you compare the `currentValue` against the `targetValue`. First, you determine if `currentValue` is greater than `targetValue`:

```
if currentValue > targetValue {
```

The `>` is the *greater-than* operator. The condition `currentValue > targetValue` is considered true if the value stored in `currentValue` is at least one higher than the value stored in `targetValue`. In that case, the following line of code is executed:

```
    difference = currentValue - targetValue
```

Here, you subtract `targetValue` (the smaller one) from `currentValue` (the larger one) and store the result in the `difference` variable.

Notice how I chose variable names that clearly describe what kind of data the variables contain. Often, you will see code such as this:

```
a = b - c
```

It is not immediately clear what this is supposed to mean, other than that some arithmetic is taking place. The variable names "a", "b" and "c" don't give any clues as to their intended purpose or what kind of data they might contain.

Back to the `if` statement. If `currentValue` is equal to or less than `targetValue`, the condition is untrue (or *false* in computer-speak) and execution will move on to the next condition:

```
} else if targetValue > currentValue {
```

The same thing happens here as before, except now the roles of `targetValue` and `currentValue` are reversed. The computer will only execute the following line when `targetValue` is the greater of the two values:

```
    difference = targetValue - currentValue
```

This time, you subtract `currentValue` from `targetValue` and store the result in the `difference` variable.

iOS App 1: Part 3 and Part 4

There is only one situation you haven't handled yet: when `currentValue` and `targetValue` are equal. If this happens, the player has put the slider exactly at the position of the target random number, a perfect score. In that case, the difference is 0:

```
} else {
  difference = 0
}
```

Since, by now, you've already determined that one value is not greater than the other, nor is it smaller, you can only draw one conclusion: The numbers must be equal.

Displaying the difference

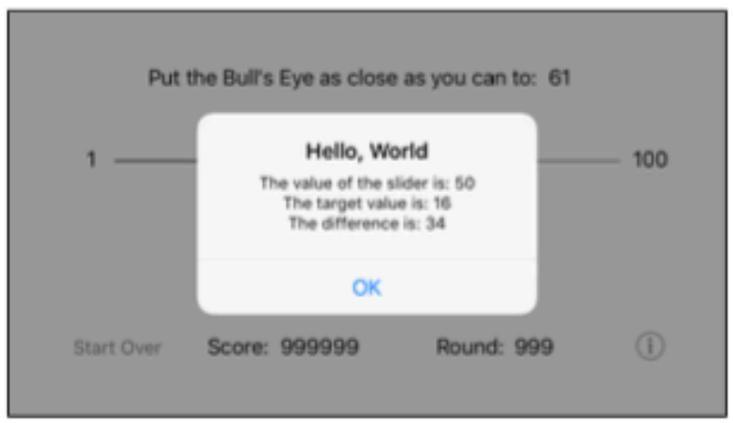
➤ Let's put this code into action. Add it to the top of `showAlert()`:

```
@IBAction func showAlert() {
  var difference: Int
  if currentValue > targetValue {
    difference = currentValue - targetValue
  } else if targetValue > currentValue {
    difference = targetValue - currentValue
  } else {
    difference = 0
  }

  let message = "The value of the slider is: \(currentValue)" +
                "\nThe target value is: \(targetValue)" +
                "\nThe difference is: \(difference)"
  . . .
}
```

Just so you can see that it works, you add the `difference` value to the alert message as well.

➤ Run it and see for yourself.



The alert shows the difference between the target and the slider

iOS App 1: Part 3 and Part 4

Simplifying the algorithm

I mentioned earlier that there are other ways to calculate the difference between `currentValue` and `targetValue` as a positive number. The above algorithm works well, but it is eight lines of code. I think we can come up with a simpler approach that takes up fewer lines.

The new algorithm goes like this:

1. *Subtract the target value from the slider's value.*
2. *If the result is a negative number, then multiply it by -1 to make it a positive number.*

Here, you no longer avoid the negative number since computers can work just fine with negative numbers. You simply turn it into a positive number.

> **Exercise**: Convert the above algorithm into source code. Hint: The English description of the algorithm contains the words "if" and "then," which is a pretty good indication that you'll have to use an `if` statement.

You should have arrived at something like this:

```
var difference = currentValue - targetValue
if difference < 0 {
  difference = difference * -1
}
```

This is a pretty straightforward translation of the new algorithm.

You first do the subtraction and put the result into the `difference` variable.

Notice that you can create the new variable and assign the result of a calculation to it, all in one line. You don't need to put it onto two different lines, like so:

```
var difference: Int
difference = currentValue - targetValue
```

Also, in the one-liner version, you didn't have to tell the compiler that `difference` takes `Int` values. Because both `currentValue` and `targetValue` are `Int`s, Swift is smart enough to figure out that difference should also be an `Int`.

This feature, as mentioned before, is called *type inference* and it's one of the big selling points of Swift.

Once you have the subtraction result, you use an `if` statement to determine whether `difference` is negative, i.e., less than zero. If it is, you multiply by -1 and put the new result — now a positive number — back into the `difference` variable.

When you write,

```
difference = difference * -1
```

the computer first multiplies `difference`'s value by -1. Then, it puts the result of that calculation back into `difference`. In effect, this overwrites `difference`'s old contents (the negative number) with the positive number.

Because this is a common thing to do, there is a handy shortcut:

```
difference *= -1
```

The *= operator combines * and = into a single operation. The end result is the same: The variable's old value is gone and it now contains the result of the multiplication.

You could also have written this algorithm as follows:

```
var difference = currentValue - targetValue
if difference < 0 {
  difference = -difference
}
```

Instead of multiplying by -1, you now use the negation operator to ensure `difference`'s value is always positive. This works because negating a negative number makes it positive again. (Ask a math professor if you don't believe me.)

Using the new algorithm

➤ Give these new algorithms a try. You should replace the old stuff at the top of `showAlert()` as follows:

```
@IBAction func showAlert() {
  var difference = currentValue - targetValue
  if difference < 0 {
    difference = difference * -1
  }

  let message = . . .
}
```

When you run this new version of the app (try it!), it should work exactly the same as before. The result of the computation does not change, only the technique used changed.

iOS App 1: Part 3 and Part 4

Another variation

The final alternative algorithm I want to show you uses a function. You've already seen functions a few times before. For example, when you used `lroundf()` for rounding off the slider's decimals. Similarly, to make sure a number is always positive, you can use the `abs()` function. If you took math in school, you might remember the term "absolute value," which is the value of a number without regard to its sign. That's exactly what you need here, and the standard library contains a convenient function for it, which allows you to reduce this entire algorithm down to a single line of code:

```
let difference = abs(targetValue - currentValue)
```

It really doesn't matter whether you subtract `currentValue` from `targetValue` or the other way around. If the number is negative, `abs()` turns it positive. It's a handy function to remember.

➤ Make the change to `showAlert()` and try it out:

```
@IBAction func showAlert() {
  let difference = abs(targetValue - currentValue)
  let message = . . .
}
```

It doesn't get much simpler than that!

> **Exercise**: Something else has changed... can you spot it?

Answer: You wrote **let** `difference` instead of **var** `difference`.

Variables and constants

Swift makes a distinction between variables and *constants*. Unlike a variable, the value of a constant, as the name implies, cannot change. You can only put something into the box of a constant once and cannot replace it with something else afterwards. The keyword `var` creates a variable while `let` creates a constant. That means `difference` is now a constant, not a variable.

In the previous algorithms, the value of `difference` could possibly change. If it was negative, you turned it positive. That required `difference` to be a variable, because only variables can have their value change.

Now that you can calculate the whole thing in a single line, `difference` will never have to change once you've given it a value. In that case, it's better to make it a constant with

`let`. Why is that better? It makes your intent clear, which in turn helps the Swift compiler understand your program better. By the same token, `message`, `alert`, and `action` are also constants (and have been all along!). Now, you know why you declared these objects with `let` instead of `var`. Once they've been given a value, they never need to change.

Constants are very common in Swift. Often, you only need to hold onto a value for a very short time. If, in that time, the value never has to change, it's best to make it a constant (`let`) and not a variable (`var`).

What's the score?

Now that you know how far off the slider is from the target, calculating the player's score for each round is easy.

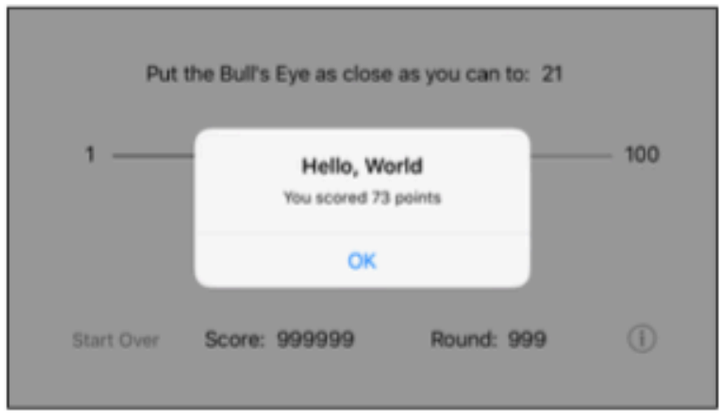
➤ Change `showAlert()` to:

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

  let message = "You scored \(points) points"
  . . .
}
```

The maximum score you can get is 100 points if you put the slider right on the target and the difference is 0. The farther away from the target you are, the fewer points you earn.

➤ Run the app and score some points!



The alert with the player's score for the current round

iOS App 1: Part 3 and Part 4

Exercise: Because the maximum slider position is 100 and the minimum is 1, the biggest difference is 100 - 1 = 99. That means the absolute worst score you can have in a round is 1 point. Explain why this is so. (Eek! It requires math!)

Showing the total score

In this game, you want to show the player's total score on the screen. After every round, the app should add the newly scored points to the total and then update the score label.

Storing the total score

Because the game needs to keep the total score around for a long time, you will need an instance variable.

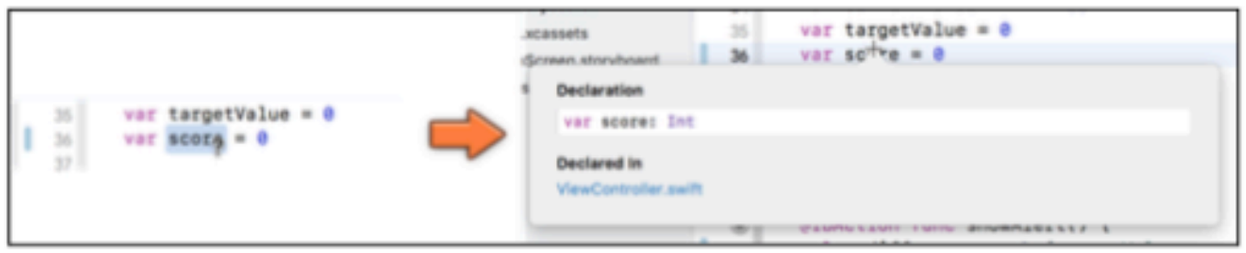
➤ Add a new `score` instance variable to **ViewController.swift**:

```
class ViewController: UIViewController {

  var currentValue: Int = 0
  var targetValue = 0
  var score = 0               // add this line
```

Again, we make use of type inference to not specify a type for `score`.

Note: If you are not sure about the inferred type of a variable, there is an easy way to find out. Simply hold down the **Alt** / **Option** key, and hover your cursor over the variable in question. The variable will be highlighted in blue and your cursor will turn into a question mark. Now, click on the variable and you will get a handy pop-up, which tells you the type of the variable, as well as the source file in which the variable was declared.



Discover the inferred type for a variable

➤ Now that you're using type inference, you can clean up `currentValue` as well (and make its initial value 0, if you haven't already):

```
    var currentValue = 0
```

Thanks to type inference, you only have to specify the data type when you're not giving the variable an initial value. But most of the time, you can safely make Swift guess at the type.

I think type inference is pretty sweet! It will definitely save you some, uh, typing (in more ways than one!).

Updating the total score

Now, `showAlert()` can be amended to update this `score` variable.

➤ Make the following changes:

```
@IBAction func showAlert() {
  let difference = abs(targetValue - currentValue)
  let points = 100 - difference
  score += points        // add this line
  let message = "You scored \(points) points"
  . . .
}
```

Nothing too shocking, here. You just added the following line:

```
score += points
```

This adds the points that the user scored in this round to the total score. You could also have written it like this:

```
score = score + points
```

Personally, I prefer the shorthand += version, but either one is OK. Both accomplish exactly the same thing.

Displaying the score

To display your current score, you're going to do the same thing that you did for the target label: hook up the score label to an outlet and put the score value into the label's `text` property.

iOS App 1: Part 3 and Part 4

> **Exercise**: See if you can do the above by yourself. You've already done these things before for the target value label, so you should be able to repeat those steps for the score label.

Done? You should have added this line to **ViewController.swift**:

```
@IBOutlet weak var scoreLabel: UILabel!
```

Then, you connect the relevant label on the storyboard (the one that says 999999) to the new `scoreLabel` outlet.

Unsure how to connect the outlet? There are several ways to make connections from user interface objects to the view controller's outlets:

- Control-click on the object to get a context-sensitive pop-up menu. Then, drag from New Referencing Outlet to View controller (you did this with the slider).
- Go to the Connections Inspector for the label. Drag from New Referencing Outlet to View controller (you did this with the target label).
- Control-drag **from** View controller to the label (give this one a try now) — doing it the other way, Control-dragging from the label to View controller, won't work.

There is more than one way to skin a cat — er, connect outlets.

Great, that gives you a `scoreLabel` outlet that you can use to display the score. Now, where in the code can you do that? In `updateLabels()`, of course.

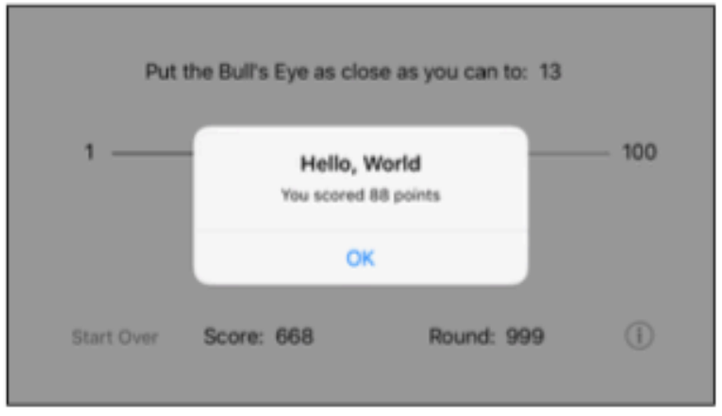
➤ Back in **ViewController.swift**, change `updateLabels()` to the following:

```
func updateLabels() {
  targetLabel.text = String(targetValue)
  scoreLabel.text = String(score)     // add this line
}
```

Nothing new, here. You convert the score — which is an `Int` — into a `String` and then pass that string to the label's `text` property. In response to that, the label will redraw itself with the new score.

iOS App 1: Part 3 and Part 4

➤ Run the app and verify that the points for this round are added to the total score label whenever you tap the button.



The score label keeps track of the player's total score

One more round...

Speaking of rounds, you also have to increment the round number each time the player starts a new round.

> **Exercise**: Keep track of the current round number (starting at 1) and increment it when a new round starts. Display the current round number in the corresponding label. I may be throwing you into the deep end here, but if you've been able to follow the instructions so far, then you've already seen all the pieces you will need to pull this off. Good luck!

If you guessed that you had to add another instance variable, then you are right. You should add the following line (or something similar) to **ViewController.swift**:

```
var round = 0
```

It's also OK if you included the name of the data type, even though that is not strictly necessary:

```
var round: Int = 0
```

Also, add an outlet for the label:

```
@IBOutlet weak var roundLabel: UILabel!
```

As before, you should connect the label to this outlet in Interface Builder.

> **Note**: Don't forget to make those connections.
>
> Forgetting to make the connections in Interface Builder is an often-made mistake, especially by yours truly.
>
> It happens to me all the time that I make the outlet for a button and write the code to deal with taps on that button but, when I run the app, it doesn't work. Usually, it takes me a few minutes and some head scratching to realize that I forgot to connect the button to the outlet or the action method.
>
> You can tap on the button all you want but, unless that connection exists, your code will not respond.

Finally, `updateLabels()` should be modified like this:

```
func updateLabels() {
  targetLabel.text = String(targetValue)
  scoreLabel.text = String(score)
  roundLabel.text = String(round)    // add this line
}
```

Did you also figure out where to increment the `round` variable?

I'd say the `startNewRound()` method is a pretty good place. After all, you call this method whenever you start a new round. It makes sense to increment the round counter there.

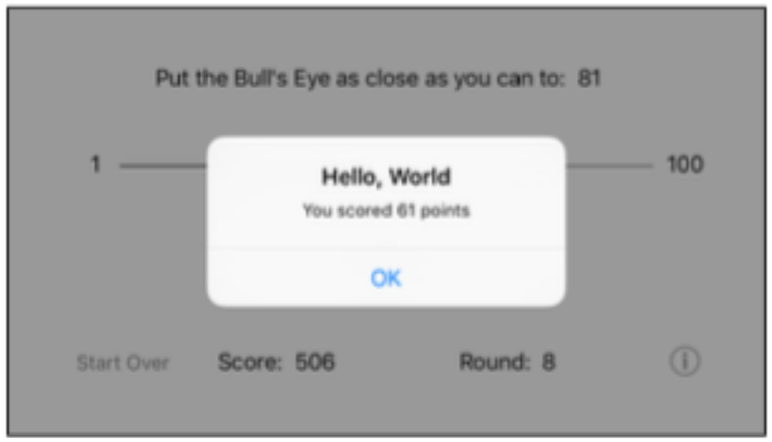
➤ Change `startNewRound()` to:

```
func startNewRound() {
  round += 1           // add this line
  targetValue = ...
}
```

Note that, when you declared the `round` instance variable, you gave it a default value of 0. Therefore, when the app starts up, `round` is initially 0. When you call `startNewRound()` for the very first time, it adds 1 to this initial value and, as a result, the first round is properly counted as round 1.

iOS App 1: Part 3 and Part 4

➤ Run the app and try it out. The round counter should update whenever you press the Hit Me! button.



The round label counts how many rounds have been played

You're making great progress; well done!

FEEDBACK TIME

http://bit.ly/iOSFeedback

THANK YOU