Chapter 4: Rounds and Score

By Fahim Farook and Matthijs Hollemans

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:

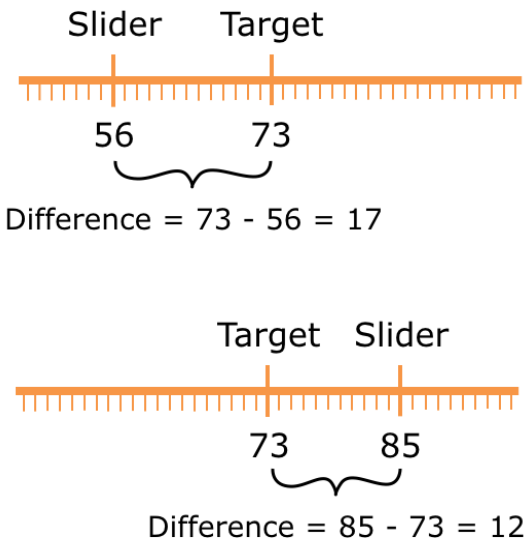
- **Get the difference:** Calculate the difference between the target value and the value that the user selected.
- **Other ways to calculate the difference:** Other approaches to calculating the difference.
- **What's the score?:** Calculate the user's score based on the difference value.
- **The total 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.

Get 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...

> **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 onscreen 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 one 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.

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 you would 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 your cxode, it is possible you came up with a different way to solve this little problem, and I'll show you two alternatives in a minute, but let's convert this one 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 from English. Generally, it works like this:

```
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 it tries them one by one from top to bottom until one proves to be true.

If none of the conditions are found to be valid, then the code in the `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 the `currentValue` variable is at least one higher than the value stored in the `targetValue` variable. 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 difference 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 the program will skip the code block and move on to the next condition:

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

The same thing happens here as before, except that 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.

There is only one situation you haven't handled yet, and that is 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 at this point 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.

Display the difference

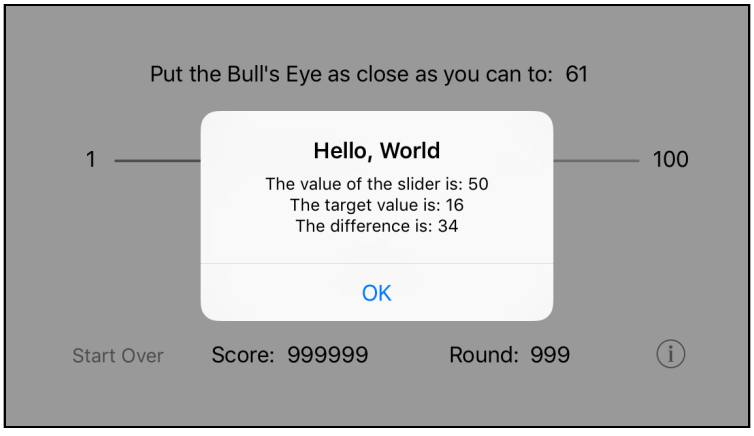
➤ Let's put this algorithm into action. Add it to the top of `showAlert()`:

```
@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

Other ways to calculate the difference

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 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 subtract the two variables 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 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.)

Use 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 you used changed.

Another variation

The final alternative algorithm I want to show you uses a function.

You've already seen functions a few times before: you used `arc4random_uniform()` when you made random numbers and `lroundf()` for rounding off the slider's decimals.

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.

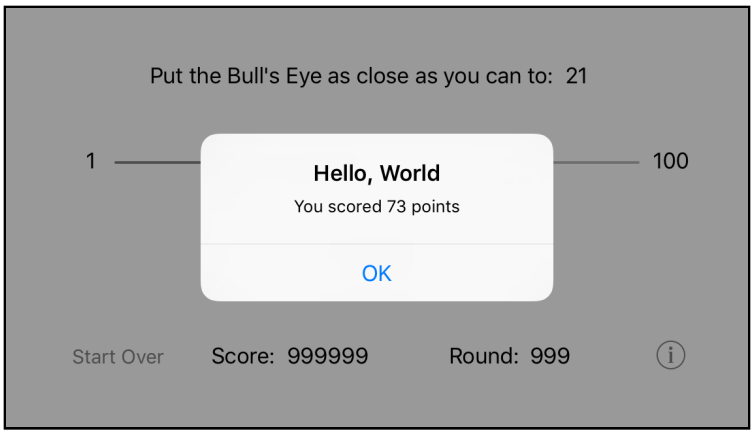
➤ Change `showAlert()` to:

```
@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 zero. The further 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

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!)

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.

Store 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**:

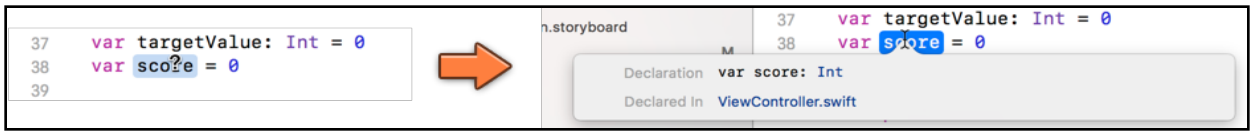
```
class ViewController: UIViewController {

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

Did you notice that? Unlike the other two instance variables, you did not state that `score` is an `Int`!

If you don't specify a data type, Swift uses *type inference* to figure out what type you meant. Because 0 is a whole number, Swift assumes that `score` should be an integer, and therefore automatically gives it the type `Int`. Handy!

> **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** 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

In fact, now that you know about type inference, you don't need to specify `Int` for the other instance variables either:

```
  var currentValue = 0
  var targetValue = 0
```

➤ Make the above changes.

Thanks to type inference, you only have to list the name of 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!).

Update 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 okay. Both accomplish exactly the same thing.

Display the score

In order to show your current score, you're going to do exactly 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.

> **Exercise:** See if you can do the above without my help. You've already done these things before for the target value label, so you should be able to repeat these steps by yourself for the score label.

Done? You should have done the following. You add 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 popup 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 the view controller, won't work.

There is more than one way to skin a cat, or, connect outlets :]

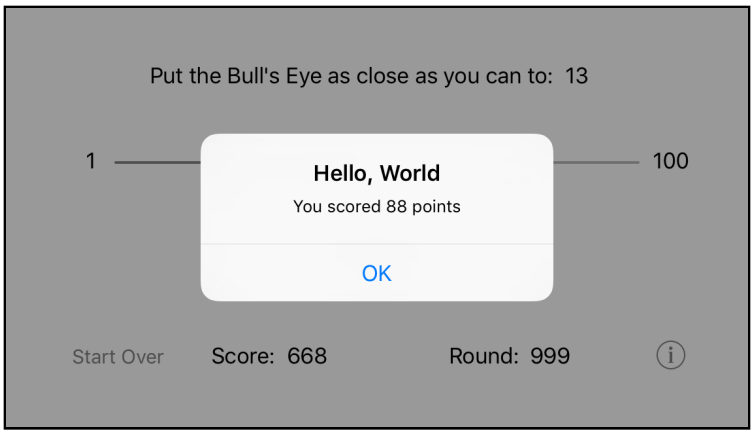
Great, that gives you a `scoreLabel` outlet that you can use to display the score. Now where in the code can you do that? In `updateLabels()`, of course.

➤ 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.

➤ 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 were right. You should have added 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.

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?

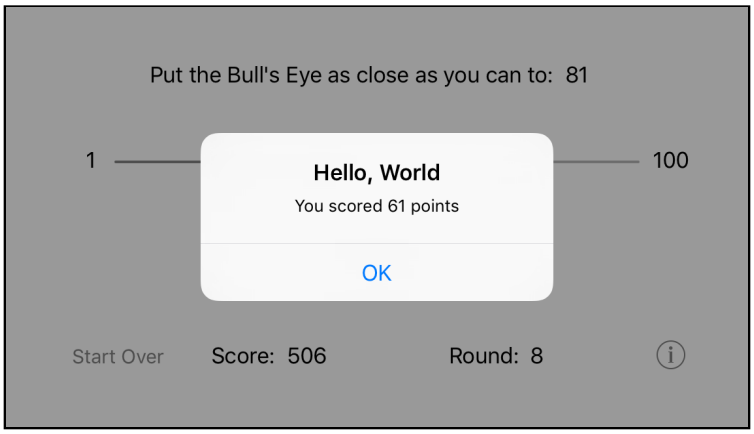
I'd say the `startNewRound()` method is a pretty good place. After all, you call this method whenever you start a new round. It makes sense to increment the round counter there.

➤ 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.

➤ 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!

You can find the project files for the app up to this point under **04 - Rounds and Score** in the Source Code folder. If you get stuck, compare your version of the app with these source files to see if you missed anything.