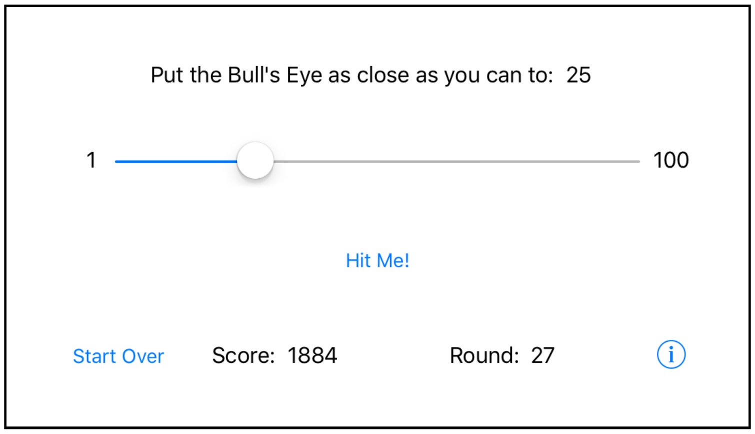
Chapter 3: Slider and Labels

Now that you have accomplished the first task of putting a button on the screen and making it show an alert, you'll simply go down the task list and tick off the other items.

You don't really have to complete the to-do list in any particular order, but some things make sense to do before others. For example, you cannot read the position of the slider if you don't have a slider yet.

So let's add the rest of the controls – the slider and the text labels – and turn this app into a real game!

When you're done, the app will look like this:



The game screen with standard UIKit controls

Hey, wait a minute… that doesn't look nearly as pretty as the game I promised you! The difference is that these are the standard UIKit controls. This is what they look like straight out of the box.

You've probably seen this look before because it is perfectly suitable for regular apps. But because the default look is a little boring for a game, you'll put some special sauce on top later to spiff things up.

In this chapter, you'll cover the following:

- **Portrait vs. landscape:** Switch your app to landscape mode.
- **Objects, data and methods:** A quick primer on the basics of object oriented programming.
- **Add the other controls:** Add the rest of the controls necessary to complete the user interface of your app.

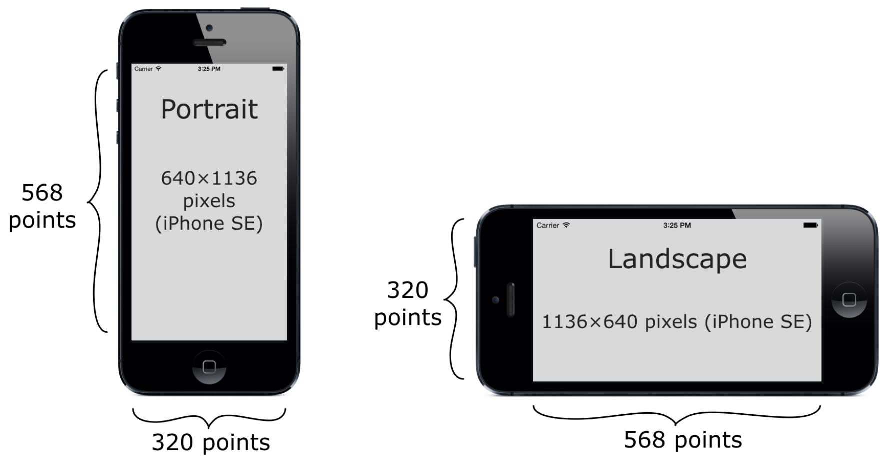
Portrait vs. landscape

Notice that in the previous screenshot, the dimensions of the app have changed: the iPhone is tilted on its side and the screen is wider but less tall. This is called *landscape* orientation.

You've no doubt seen landscape apps before on the iPhone. It's a common display orientation for games but many other types of apps work in landscape mode too, usually in addition to the regular "upright" *portrait* orientation.

For instance, many people prefer to write emails with their device flipped over because the wider screen allows for a bigger keyboard and easier typing.

In portrait orientation, the iPhone SE screen consists of 320 points horizontally and 568 points vertically. For landscape these dimensions are switched.



Screen dimensions for portrait and landscape orientation

So what is a *point*?

On older devices – up to the iPhone 3GS and corresponding iPod touch models, as well as the first iPads – one point corresponds to one pixel. As a result, these low-resolution devices don't look very sharp because of their big, chunky pixels.

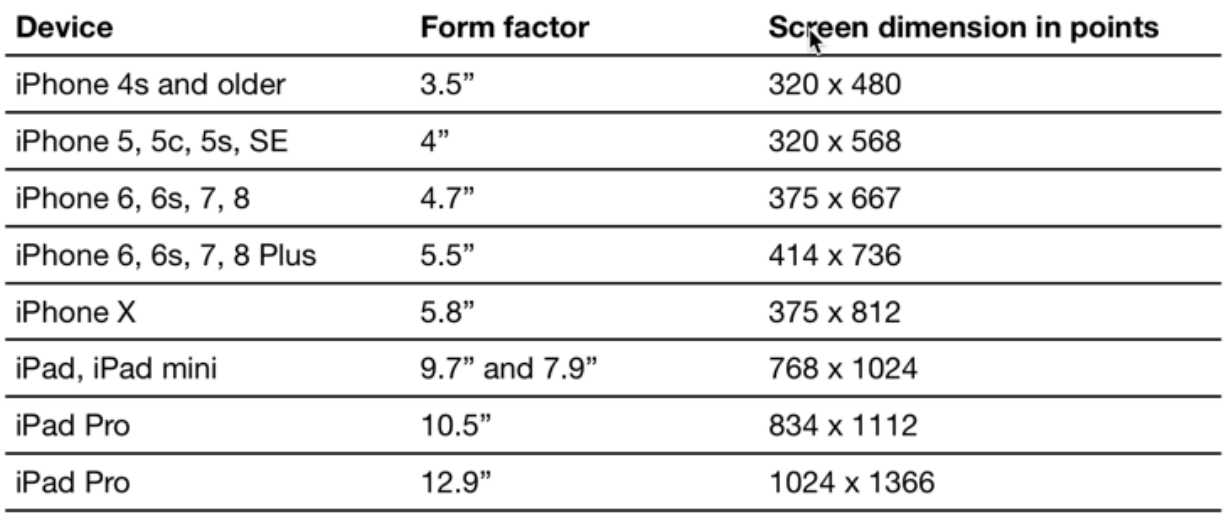
I'm sure you know what a pixel is? In case you don't, it's the smallest element that a screen is made up of. (That's how the word originated, a shortened form of pictures, PICS or PIX + ELement = PIXEL.) The display of your iPhone is a big matrix of pixels that each can have their own color, just like a TV screen. Changing the color values of these pixels produces a visible image on the display. The more pixels, the better the image looks.

On the high-resolution Retina display of the iPhone 4 and later models, one point actually corresponds to two pixels horizontally and vertically, so four pixels in total. It packs a lot of pixels in a very small space, making for a much sharper display, which accounts for the popularity of Retina devices.

On the Plus devices it's even crazier: they have a 3x resolution with *nine* pixels for every point. Insane! You need to be eagle-eyed to make out the individual pixels on these fancy Retina HD displays. It becomes almost impossible to make out where one pixel ends and the next one begins, that's how miniscule they are!

It's not only the number of pixels that differs between the various iPhone and iPad models. Over the years they have received different form factors, from the small 3.5-inch screen in the beginning all the way up to 12.9-inches on the iPad Pro model.

The form factor of the device determines the width and height of the screen in points:



Device	Form factor	Screen dimension in points
iPhone 4s and older	3.5"	320 x 480
iPhone 5, 5c, 5s, SE	4"	320 x 568
iPhone 6, 6s, 7, 8	4.7"	375 x 667
iPhone 6, 6s, 7, 8 Plus	5.5"	414 x 736
iPhone X	5.8"	375 x 812
iPad, iPad mini	9.7" and 7.9"	768 x 1024
iPad Pro	10.5"	834 x 1112
iPad Pro	12.9"	1024 x 1366

In the early days of iOS, there was only one screen size. But those days of “one size fits all” are long gone. Now we have a variety of screen sizes to deal with.

UIKit and other frameworks

iOS offers a lot of building blocks in the form of frameworks or “kits”. The UIKit framework provides the user interface controls such as buttons, labels and navigation bars. It manages the view controllers and generally takes care of anything else that deals with your app’s user interface. (That is what UI stands for: User Interface.)

If you had to write all that stuff from scratch, you’d be busy for a long while. Instead, you can build your app on top of the system-provided frameworks and take advantage of all the work the Apple engineers have already put in.

Any object you see whose name starts with UI, such as `UIButton`, comes from UIKit. When you’re writing iOS apps, UIKit is the framework you’ll spend most of your time with, but there are others as well.

Examples of other frameworks are Foundation, which provides many of the basic building blocks for building apps; Core Graphics for drawing basic shapes such as lines, gradients and images on the screen; AVFoundation for playing sound and video; and many others.

The complete set of frameworks for iOS is known collectively as Cocoa Touch.

Remember that UIKit works with points instead of pixels, so you only have to worry about the differences between the screen sizes measured in points. The actual number of pixels is only important for graphic designers because images are still measured in pixels.

Developers work in points, designers work in pixels.

The difference between points and pixels can be a little confusing, but if that is the only thing you’re confused about right now then I’m doing a pretty good job. ;-)

For the time being, you’ll work with just the iPhone SE screen size of 320×568 points – just to keep things simple. Later on you’ll also make the game fit on the other iPhone screens.

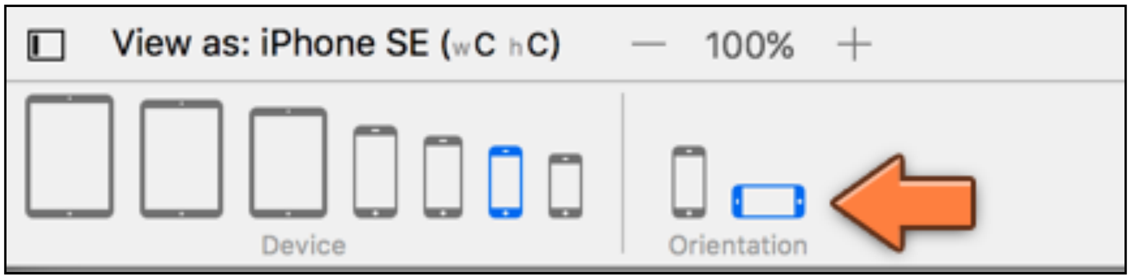
Convert the app to landscape

To switch the app from portrait to landscape, you have to do two things:

1. Make the view in **Main.storyboard** landscape instead of portrait.

2. Change the **Supported Device Orientations** setting of the app.

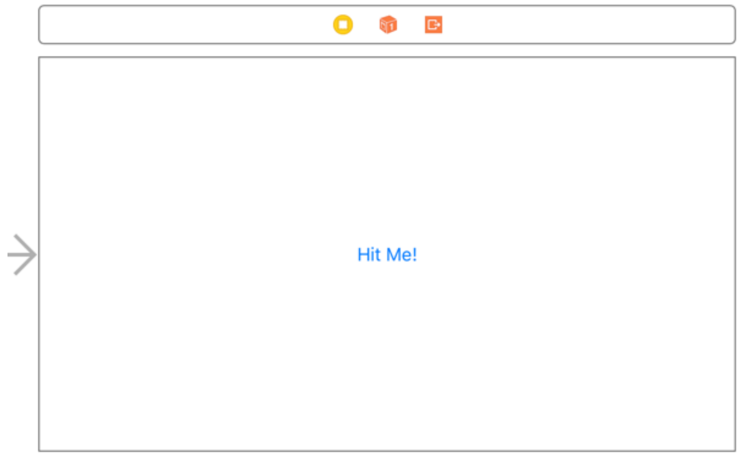
➤ Open **Main.storyboard**. In Interface Builder, in the **View as: iPhone SE** panel, change **Orientation** to landscape:



Changing the orientation in Interface Builder

This changes the dimensions of the view controller. It also puts the button off-center.

➤ Move the button back to the center of the view because an untidy user interface just won't do in this day and age.



The view in landscape orientation

That takes care of the view layout.

➤ Run the app on the iPhone SE Simulator. Note that the screen does not show up as landscape yet, and the button is no longer in the center.

➤ Choose **Hardware** → **Rotate Left** or **Rotate Right** from the Simulator's menu bar at the top of the screen, or hold ⌘ and press the left or right arrow keys on your keyboard. This will flip the Simulator around.

Now, everything will look as it should.

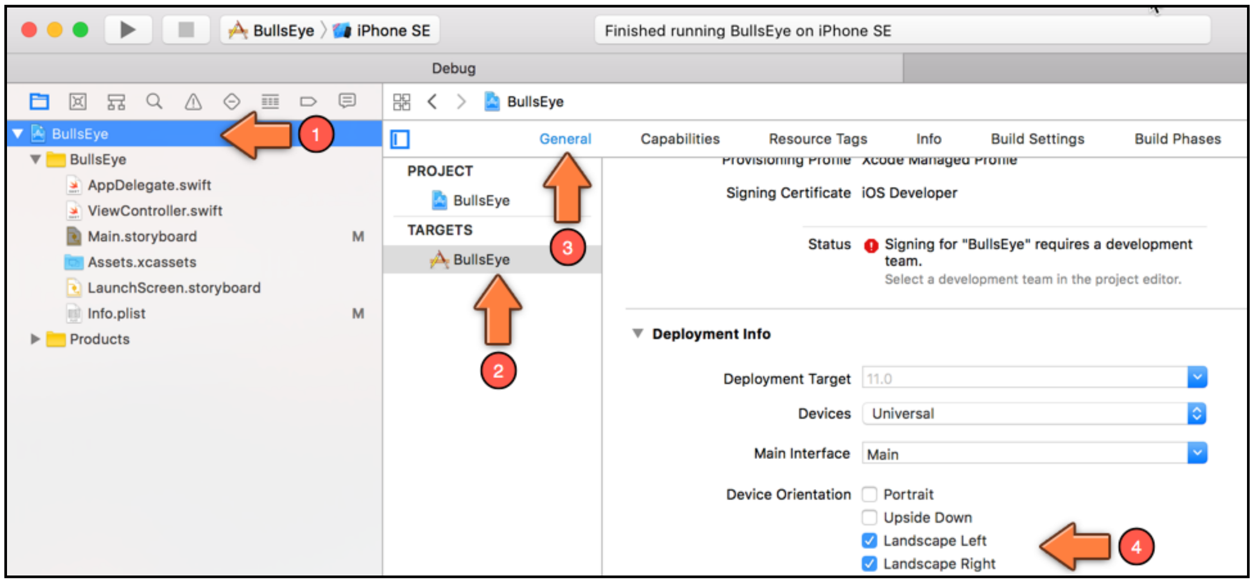
Notice that in landscape orientation the app no longer shows the iPhone's status bar.

This gives apps more room for their user interfaces.

To finalize the orientation switch, you should do one more thing. There is a configuration option that tells iOS what orientations your app supports. New apps that you make from a template always support both portrait and landscape orientations.

➤ Click the blue **BullsEye** project icon at the top of the **Project navigator**. The editor pane of the Xcode window now reveals a bunch of settings for the project.

➤ Make sure that the **General** tab is selected:



Signing Certificate: iOS Developer

Status: Signing for "BullsEye" requires a development team.
Select a development team in the project editor.

Deployment Info

Deployment Target: 11.0

Devices: Universal

Main Interface: Main

Device Orientation: Portrait [ ], Upside Down [ ], Landscape Left [✓], Landscape Right [✓]

The settings for the project

In the **Deployment Info** section, there is an option for **Device Orientation**.

➤ Check only the **Landscape Left** and **Landscape Right** options and leave the Portrait and Upside Down options unchecked.

Run the app again and it properly launches in the landscape orientation right from the start.

Objects, data and methods

Time for some programming theory. Yes, you cannot escape it. :]

Swift is a so-called “object-oriented” programming language, which means that most of the stuff you do involves objects of some kind. I already mentioned a few times that an app consists of objects that send messages to each other.

When you write an iOS app, you'll be using objects that are provided for you by the system, such as the `UIButton` object from UIKit, and you'll be making objects of your own, such as view controllers.

Objects

So what exactly *is* an object? Think of an object as a building block of your program.

Programmers like to group related functionality into objects. *This* object takes care of parsing a file, *that* object knows how to draw an image on the screen, and *that* object over there can perform a difficult calculation.

Each object takes care of a specific part of the program. In a full-blown app you will have many different types of objects (tens or even hundreds).

Even your small starter app already contains several different objects. The one you have spent the most time with so far is `ViewController`. The Hit Me button is also an object, as is the alert popup. And the text values that you put on the alert – "Hello, World" and "This is my first app!" – are also objects.

The project also has an object named `AppDelegate` - you're going to ignore that for the moment, but feel free to look at its source if you're curious. These object thingies are everywhere!

Data and methods

An object can have both *data* and *functionality*:

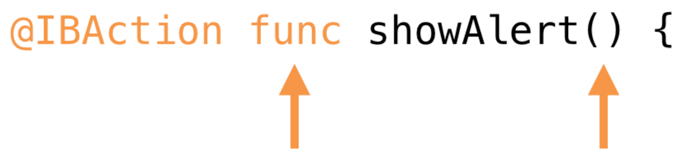
- An example of data is the Hit Me button that you added to the view controller earlier. When you dragged the button into the storyboard, it actually became part of the view controller's data. Data *contains* something. In this case, the view controller contains the button.

- An example of functionality is the `showAlert` action that you added to respond to taps on the button. Functionality *does* something.

The button itself also has data and functionality. Examples of button data are the text and color of its label, its position on the screen, its width and height, and so on. The button also has functionality: it can recognize that the user tapped on it and will trigger an action in response.

The thing that provides functionality to an object is commonly called a *method*. Other programming languages may call this a “procedure” or “subroutine” or “function”. You will also see the term function used in Swift; a method is simply a function that belongs to an object.

Your `showAlert` action is an example of a method. You can tell it’s a method because the line says `func` (short for “function”) and the name is followed by parentheses:



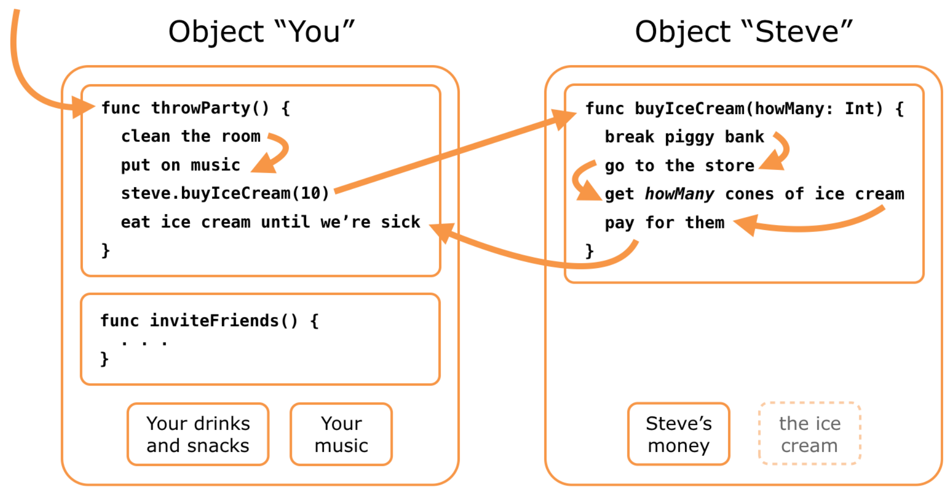
```
@IBAction func showAlert() {
```

All method definitions start with the word func and have parentheses

If you look through the rest of **ViewController.swift** you’ll see several other methods, such as `viewDidLoad()` and `didReceiveMemoryWarning()`.

These currently don’t do much; the Xcode template placed them there for your convenience. These specific methods are often used by view controllers, so it’s likely that you will need to fill them in at some point.

The concept of methods may still feel a little weird, so here’s an example:



```
func throwParty() {
  clean the room
  put on music
  steve.buyIceCream(10)
  eat ice cream until we’re sick
}

func inviteFriends() {
  . . .
}
```

```
func buyIceCream(howMany: Int) {
  break piggy bank
  go to the store
  get howMany cones of ice cream
  pay for them
}
```

Every party needs ice cream!

You (or at least an object named "You") want to throw a party, but you forgot to buy ice cream. Fortunately, you have invited the object named Steve who happens to live next door to a convenience store. It won't be much of a party without ice cream, so at some point during your party preparations you send object Steve a message asking him to bring some ice cream.

The computer now switches to object Steve and executes the commands from his `buyIceCream()` method, one by one, from top to bottom.

When the `buyIceCream()` method is done, the computer returns to your `throwParty()` method and continues with that, so you and your friends can eat the ice cream that Steve brought back with him.

The Steve object also has data. Before he goes to the store he has money. At the store he exchanges this money data for other, much more important, data: ice cream! After making that transaction, he brings the ice cream data over to the party (if he eats it all along the way, your program has a bug).

Messages

"Sending a message" sounds more involved than it really is. It's a good way to think conceptually of how objects communicate, but there really aren't any pigeons or mailmen involved. The computer simply jumps from the `throwParty()` method to the `buyIceCream()` method and back again.

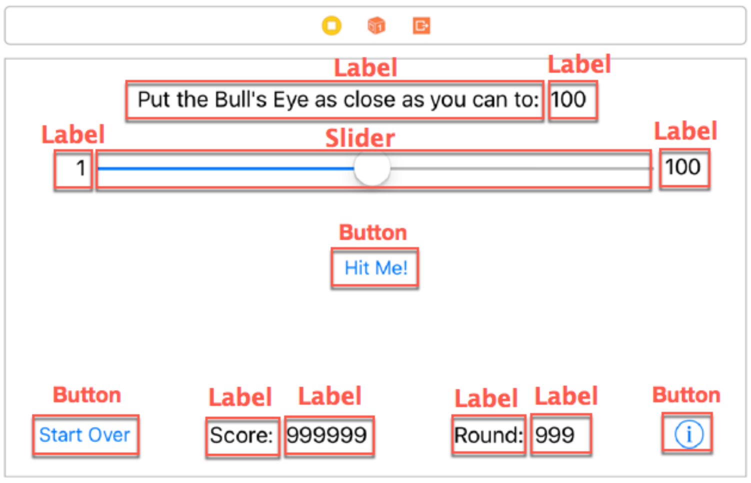
Often the terms "calling a method" or "invoking a method" are used instead. That means the exact same thing as sending a message: the computer jumps to the method you're calling and returns to where it left off when that method is done.

The important thing to remember is that objects have methods (the steps involved in buying ice cream) and data (the actual ice cream and the money to buy it with).

Objects can look at each other's data (to some extent anyway, just like Steve may not approve if you peek inside his wallet) and can ask other objects to perform their methods. That's how you get your app to do things. (But not all data from an object can be inspected by other objects and/or code - this is an area known as access control and you'll learn about this later.)

Add the other controls

Your app already has a button but you still need to add the rest of the UI controls, also known as "views". Here is the screen again, this time annotated with the different types of views:

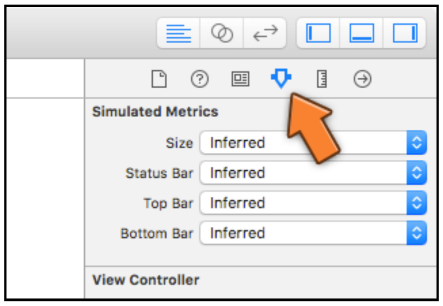


The different views in the game screen

As you can see, I put placeholder values into some of the labels (for example, "999999"). That makes it easier to see how the labels will fit on the screen when they're actually used. The score label could potentially hold a large value, so you'd better make sure the label has room for it.

➤ Try to re-create the above screen on your own by dragging the various controls from the Object Library on to your scene. You'll need a few new Buttons, Labels, and a Slider. You can see in the screenshot above how big the items should (roughly) be. It's OK if you're a few points off.

To tweak the settings of these views, you use the **Attributes inspector**. You can find this inspector in the right-hand pane of the Xcode window:



Simulated Metrics

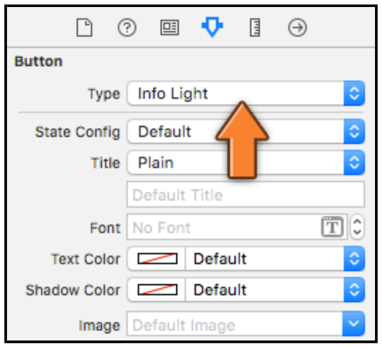
Size: Inferred
Status Bar: Inferred
Top Bar: Inferred
Bottom Bar: Inferred

View Controller

The Attributes inspector

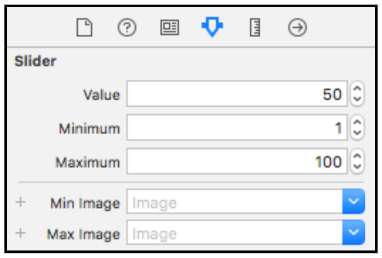
The inspector area shows various aspects of the item that is currently selected. The Attributes inspector, for example, lets you change the background color of a label or the size of the text on a button. You've already seen the Connections inspector that showed the button's actions. As you become more proficient with Interface Builder, you'll be using all of these inspector panes to configure your views.

➤ Hint: the ⓘ button is actually a regular Button, but its **Type** is set to **Info Light** in the Attributes inspector:


Button
Type: Info Light
State Config: Default
Title: Plain
Default Title
Font: No Font
Text Color: Default
Shadow Color: Default
Image: Default Image

The button type lets you change the look of the button

➤ Also use the Attributes inspector to configure the **slider**. Its minimum value should be 1, its maximum 100, and its current value 50.


Slider
Value: 50
Minimum: 1
Maximum: 100
Min Image: Image
Max Image: Image

The slider attributes

When you're done, you should have 12 user interface elements in your scene: one slider, three buttons and a whole bunch of labels. Excellent!

➤ Run the app and play with it for a minute. The controls don't really do much yet (except for the button that should still pop up the alert), but you can at least drag the slider around.

You can now tick a few more items off the to-do list, all without any programming! That is going to change really soon, because you will have to write Swift code to actually make the controls do anything.

The slider

The next item on your to-do list is: "Read the value of the slider after the user presses the Hit Me button."

If, in your messing around in Interface Builder, you did not accidentally disconnect the button from the `showAlert` action, you can modify the app to show the slider's value in the alert popup. (If you did disconnect the button, then you should hook it up again first. You know how, right?)

Remember how you added an action to the view controller in order to recognize when the user tapped the button? You can do the same thing for the slider. This new action will be performed whenever the user drags the slider.

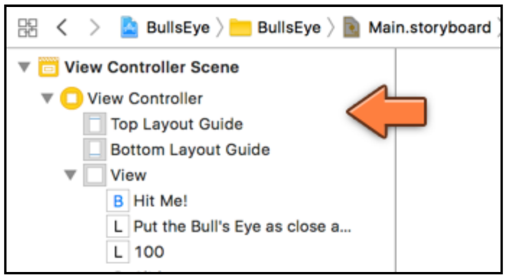
The steps for adding this action are largely the same as before.

➤ First, go to **ViewController.swift** and add the following at the bottom, just before the final closing curly bracket:

```
@IBAction func sliderMoved(_ slider: UISlider) {
  print("The value of the slider is now: \(slider.value)")
}
```

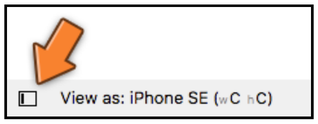
➤ Second, go to the storyboard and Control-drag from the slider to View Controller in the Document Outline. Let go of the mouse button and select **sliderMoved:** from the popup. Done!

Just to refresh your memory, the Document Outline sits on the left-hand side of the Interface Builder canvas. It shows the view hierarchy of the storyboard. Here you can see that the View Controller contains a view (succinctly named View) which in turn contains the sub-views you've added: the buttons and labels.



The Document Outline shows the view hierarchy of the storyboard

Remember, if the Document Outline is not visible, click the little icon at the bottom of the Xcode window to reveal it:

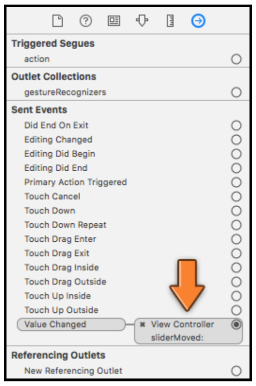


This button shows or hides the Document Outline

When you connect the slider, make sure to Control-drag to View Controller (the yellow circle icon), not View Controller Scene at the very top. If you don't see the yellow circle icon, then click the arrow in front of View Controller Scene (called the "disclosure triangle") to expand it.

If all went well, the `sliderMoved:` action is now hooked up to the slider's Value Changed event. This means the `sliderMoved()` method will be called every time the user drags the slider to the left or right.

You can verify that the connection was made by selecting the slider and looking at the **Connections inspector**:



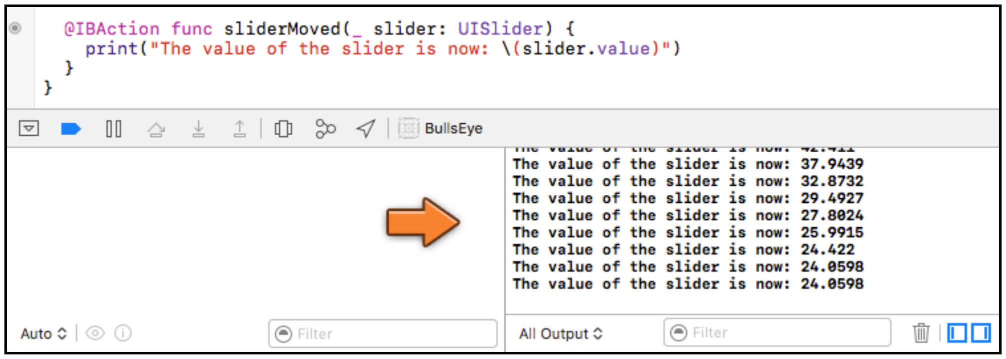
Triggered Segues
action
Outlet Collections
gestureRecognizers
Sent Events
Did End On Exit
Editing Changed
Editing Did Begin
Editing Did End
Primary Action Triggered
Touch Cancel
Touch Down
Touch Down Repeat
Touch Drag Enter
Touch Drag Exit
Touch Drag Inside
Touch Drag Outside
Touch Up Inside
Touch Up Outside
Value Changed — View Controller sliderMoved:
Referencing Outlets
New Referencing Outlet

The slider is now hooked up to the view controller

Note: Did you notice that the `sliderMoved:` action has a colon in its name but `showAlert` does not? That's because the `sliderMoved()` method takes a single parameter, `slider`, while `showAlert()` does not have any parameters. If an action method has a parameter, Interface Builder adds a `:` to the name. You'll learn more about parameters and how to use them soon.

➤ Run the app and drag the slider.

As soon as you start dragging, the Xcode window opens a new pane at the bottom, the **Debug area**, showing a list of messages:

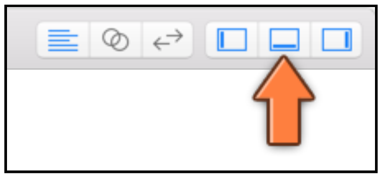


```
@IBAction func sliderMoved(_ slider: UISlider) {
  print("The value of the slider is now: \(slider.value)")
}
}
```

```
The value of the slider is now: 37.9439
The value of the slider is now: 32.8732
The value of the slider is now: 29.4927
The value of the slider is now: 27.8024
The value of the slider is now: 25.9915
The value of the slider is now: 24.422
The value of the slider is now: 24.0598
The value of the slider is now: 24.0598
```

Printing messages in the Debug area

Note: If for some reason the Debug area does not show up, you can always show (or hide) the Debug area by using the appropriate toolbar button on the top right corner of the Xcode window. You will notice from the above screenshot that the Debug area is split into two panes. You can control which of the panes is shown/hidden by using the two blue square icons shown above in the bottom right corner.



Show Debug area

If you swipe the slider all the way to the left, you should see the value go down to 1. All the way to the right, the value should stop at 100.

The `print()` function is a great help to show you what is going on in the app. Its entire purpose is to write a text message to the **Console** - the right-hand pane in the Debug area. Here, you used `print()` to verify that you properly hooked up the action to the slider and that you can read the slider value as the slider is moved.

I often use `print()` to make sure my apps are doing the right thing before I add more functionality. Printing a message to the Console is quick and easy.

Note: You may see a bunch of other messages in the Console too. This is debug output from UIKit and the iOS Simulator. You can safely ignore these messages.

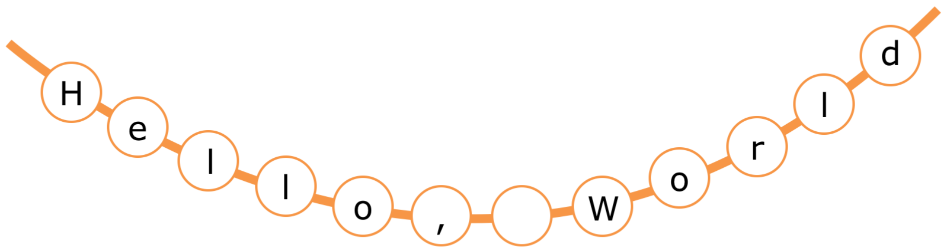
Strings

To put text in your app, you use something called a “string”. The strings you have used so far are:

```
"Hello, World"
"This is my first app!"
"Awesome"
"The value of the slider is now: \(slider.value)"
```

The first three were used to make the `UIAlertController`; the last one you used with `print()`.

Such a chunk of text is called a string because you can visualize the text as a sequence of characters, as if they were pearls in a necklace:



A string of characters

Working with strings is something you need to do all the time when you’re writing apps, so over the course of this book you’ll get quite experienced in using strings.

In Swift, to create a string, simply put the text in between double quotes. In other languages you can often use single quotes as well, but in Swift they must be double quotes. And they must be plain double quotes, not typographic “smart quotes”.

To summarize:

```
// This is the proper way to make a Swift string:
"I am a good string"

// These are wrong:
'I should have double quotes'
''Two single quotes do not make a double quote''
“My quotes are too fancy”
@"I am an Objective-C string"
```

Anything between the characters `\(` and `)` inside a string is special. The `print()` statement used the string, `"The value of the slider is now: \(slider.value)"`. Think of the `\( … )` as a placeholder: `"The value of the slider is now: X"`, where X will be replaced by the value of the slider.

Filling in the blanks this way is a very common way to build strings in Swift.

Variables

Printing information with `print()` to the Console is very useful during development of the app, but it's absolutely useless to the user because they can't see the Console when the app is running on a device.

Let's improve this to show the value of the slider in the alert popup. So how do you get the slider's value into `showAlert()`?

When you read the slider's value in `sliderMoved()`, that piece of data disappears when the action method ends. It would be handy if you could remember this value until the user taps the Hit Me button.

Fortunately, Swift has a building block for exactly this purpose: the *variable*.

➤ Open **ViewController.swift** and add the following at the top, directly below the line that says `class ViewController`:

```
var currentValue: Int = 0
```

You have now added a variable named `currentValue` to the view controller object.

The code should look like this (I left out the method code, also known as the method implementations):

```
import UIKit

class ViewController: UIViewController {
  var currentValue: Int = 0

  override func viewDidLoad() {
    . . .
  }

  override func didReceiveMemoryWarning() {
    . . .
  }

  @IBAction func showAlert() {
    . . .
  }

  @IBAction func sliderMoved(_ slider: UISlider) {
    . . .
  }
}
```

It is customary to add the variables above the methods, and to indent everything with a tab, or two to four spaces. Which one you use is largely a matter of personal preference. I like to use two spaces. (You can configure this in Xcode's preferences panel. From the menu bar choose **Xcode** → **Preferences...** → **Text Editing** and go to the **Indentation** tab.)

Remember when I said that a view controller, or any object really, could have both data and functionality? The `showAlert()` and `sliderMoved()` actions are examples of functionality, while the `currentValue` variable is part of the view controller's data.

A variable allows the app to remember things. Think of a variable as a temporary storage container for a single piece of data. Similar to how there are containers of all sorts and sizes, data comes in all kinds of shapes and sizes.

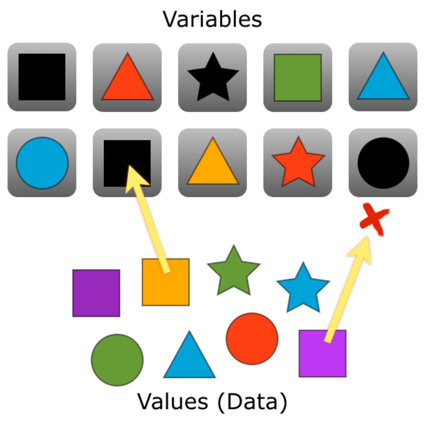
You don't just put stuff in the container and then forget about it. You will often replace its contents with a new value. When the thing that your app needs to remember changes, you take the old value out of the box and put in the new value.

That's the whole point behind variables: they can *vary*. For example, you will update `currentValue` with the new position of the slider every time the slider is moved.

The size of the storage container and the sort of values the variable can remember are determined by its *data type*, or just *type*.

You specified the type `Int` for the `currentValue` variable, which means this container can hold whole numbers (also known as "integers") between at least minus two billion and plus two billion. `Int` is one of the most common data types. There are many others though, and you can even make your own.

Variables are like children's toy blocks:



Variables are containers that hold values

The idea is to put the right shape in the right container. The container is the variable and its type determines what “shape” fits. The shapes are the possible values that you can put into the variables.

You can change the contents of each box later as long as the shape fits. For example, you can take out a blue square from a square box and put in a red square - the only thing you have to make sure is that both are squares.

But you can’t put a square in a round hole: the data type of the value and the data type of the variable have to match.

I said a variable is a *temporary* storage container. How long will it keep its contents? Unlike meat or vegetables, variables won’t spoil if you keep them for too long – a variable will hold onto its value indefinitely, until you put a new value into that variable or until you destroy the container altogether.

Each variable has a certain lifetime (also known as its *scope*) that depends on exactly where in your program you defined that variable. In this case, `currentValue` sticks around for just as long as its owner, `ViewController`, does. Their fates are intertwined.

The view controller, and thus `currentValue`, is there for the duration of the app. They don’t get destroyed until the app quits. Soon you’ll also see variables that are short-lived (also known as “local” variables).

Enough theory, let’s make this variable work for us.

➤ Change the contents of the `sliderMoved()` method in **ViewController.swift** to the following:

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

You removed the `print()` statement and replaced it with this line:

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

What is going on here?

You’ve seen `slider.value` before, which is the slider’s position at a given moment. This is a value between 1 and 100, possibly with digits behind the decimal point. And `currentValue` is the name of the variable you have just created.

To put a new value into a variable, you simply do this:

```
variable = the new value
```

This is known as "assignment". You *assign* the new value to the variable. It puts the shape into the box. Here, you put the value that represents the slider's position into the `currentValue` variable.

Functions

But what is the `lroundf` thing? Recall that the slider's value can be a non-whole number. You've seen this with the `print()` output in the Console as you moved the slider.

However, this game would be really hard if you made the player guess the position of the slider with an accuracy that goes beyond whole numbers. That will be nearly impossible to get right!

To give the player a fighting chance, you use whole numbers only. That is why `currentValue` has a data type of `Int`, because it stores *integers*, a fancy term for whole numbers.

You use the function `lroundf()` to round the decimal number to the nearest whole number and you then store that rounded-off number in `currentValue`.

Functions and methods

You've already seen that methods provide functionality, but *functions* are another way to put functionality into your apps (the name sort of gives it away, right?). Functions and methods are how Swift programs combine multiple lines of code into single, cohesive units.

The difference between the two is that a function doesn't belong to an object while a method does. In other words, a method is exactly like a function – that's why you use the `func` keyword to define them – except that you need to have an object to use the method. But regular functions, or *free functions* as they are sometimes called, can be used anywhere.

Swift provides your programs with a large library of useful functions. The function `lroundf()` is one of them and you'll be using quite a few others as you progress. `print()` is also a function, by the way. You can tell because the function name is always followed by parentheses that possibly contain one or more parameters.

➤ Now change the `showAlert()` method to the following:

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

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

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

  alert.addAction(action)

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

The line with `let message =` is new. Also note the other two small changes marked by comments.

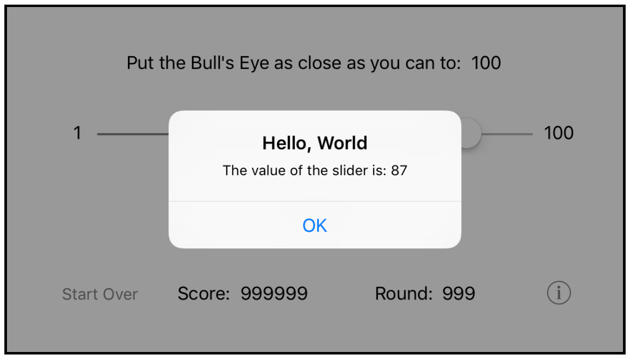
> **Note:** Anything appearing after two slashes `//` (and up to the end of that particular line) in Swift source code is treated as a comment - a note by the developer to themselves, or to other developers. The Swift compiler generally ignores comments - they are there for the convenience of humans.

As before, you create and show a `UIAlertController`, except this time its message says: “The value of the slider is: X”, where X is replaced by the contents of the `currentValue` variable (a whole number between 1 and 100).

Suppose `currentValue` is 34, which means the slider is about one-third to the left. The new code above will convert the string `"The value of the slider is: \(currentValue)"` into `"The value of the slider is: 34"` and put that into a new object named `message`.

The old `print()` did something similar, except that it printed the result to the Console. Here, however, you do not wish to print the result but show it in the alert popup. That is why you tell the `UIAlertController` that it should now use this new string as the message to display.

➤ Run the app, drag the slider, and press the button. Now the alert should show the actual value of the slider.


Put the Bull's Eye as close as you can to: 100
1
100
Hello, World
The value of the slider is: 87
OK
Start Over
Score: 999999
Round: 999

The alert shows the value of the slider

Cool. You have used a variable, `currentValue`, to remember a particular piece of data, the rounded-off position of the slider, so that it can be used elsewhere in the app, in this case in the alert's message text.

If you tap the button again without moving the slider, the alert will still show the same value. The variable keeps its value until you put a new one into it.

Your first bug

There is a small problem with the app, though. Maybe you've noticed it already. Here is how to reproduce the problem:

➤ Press the Stop button in Xcode to completely terminate the app, then press Run again. Without moving the slider, immediately press the Hit Me button.

The alert now says: "The value of the slider is: 0". But the slider's knob is obviously at the center, so you would expect the value to be 50. You've discovered a bug!

Exercise: Think of a reason why the value would be 0 in this particular situation (start the app, don't move the slider, press the button).

Answer: The clue here is that this only happens when you don't move the slider. Of course, without moving the slider the `sliderMoved()` message is never sent and you never put the slider's value into the `currentValue` variable.

The default value for the `currentValue` variable is 0, and that is what you are seeing here.

➤ To fix this bug, change the declaration of `currentValue` to:

```
var currentValue: Int = 50
```

Now the starting value of `currentValue` is 50, which should be the same value as the slider's initial position.

➤ Run the app again and verify that the bug is fixed.

You can find the project files for the app up to this point under **03 - Slider and Labels** in the Source Code folder.

Chapter 4: Outlets

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 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.

Improve 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.

Get 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.

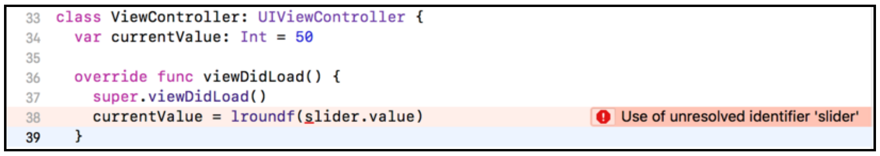
➤ Change `viewDidLoad()` to the following:

```
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.



```
class ViewController: UIViewController {
  var currentValue: Int = 50

  override func viewDidLoad() {
    super.viewDidLoad()
    currentValue = lroundf(slider.value)        Use of unresolved identifier 'slider'
  }
```

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.

3. **Local scope.** Objects with a local scope, such as the `slider` parameter of `sliderMoved()`, only exist for the duration of that method. As soon as the execution

of the program leaves this method, the local objects are no longer accessible.

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

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

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

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

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

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

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

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

Set up outlets

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

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

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

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

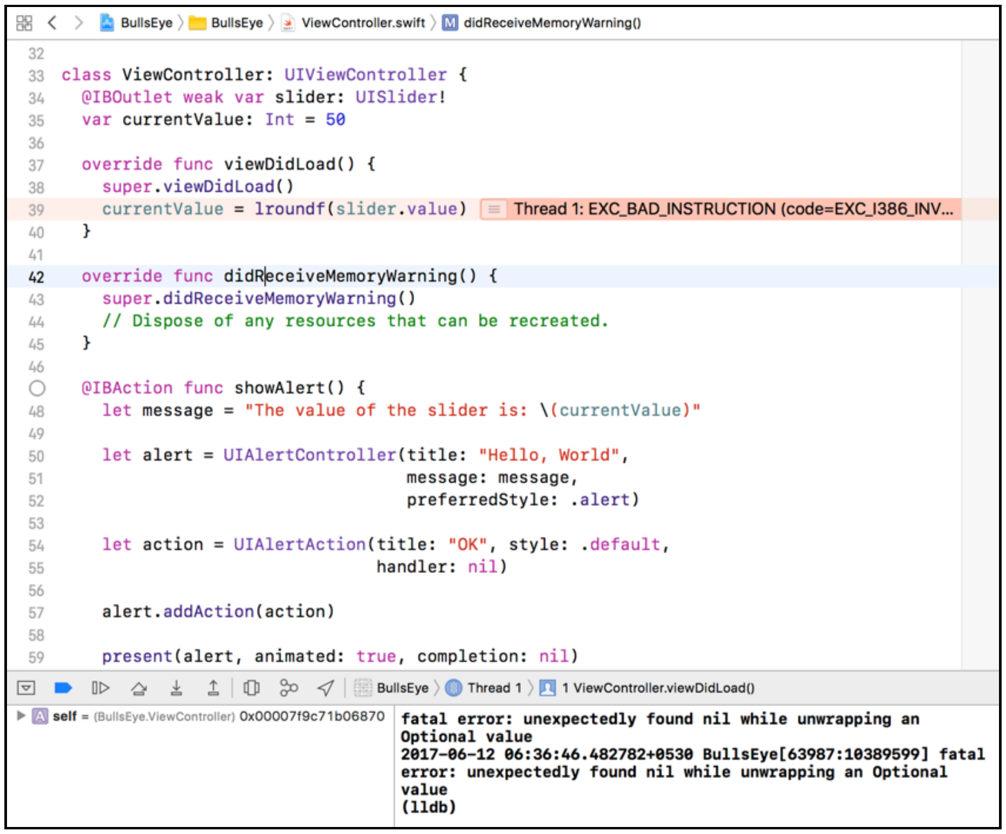
It doesn’t really matter where this line goes, just as long as it is somewhere inside the brackets for `class ViewController`. I usually put outlets with the other instance variables - at the top of the class implementation.

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

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

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

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



BullsEye 〉 BullsEye 〉 ViewController.swift 〉 M didReceiveMemoryWarning()

```
class ViewController: UIViewController {
  @IBOutlet weak var slider: UISlider!
  var currentValue: Int = 50

  override func viewDidLoad() {
    super.viewDidLoad()
    currentValue = lroundf(slider.value)    Thread 1: EXC_BAD_INSTRUCTION (code=EXC_I386_INV...
  }

  override func didReceiveMemoryWarning() {
    super.didReceiveMemoryWarning()
    // Dispose of any resources that can be recreated.
  }

  @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)

    alert.addAction(action)

    present(alert, animated: true, completion: nil)
```

BullsEye 〉 Thread 1 〉 1 ViewController.viewDidLoad()

self = (BullsEye.ViewController) 0x00007f9c71b06870

```
fatal error: unexpectedly found nil while unwrapping an
Optional value
2017-06-12 06:36:46.482782+0530 BullsEye[63987:10389599] fatal
error: unexpectedly found nil while unwrapping an Optional
value
(lldb)
```

App crash when outlet is not connected

So, what happened?

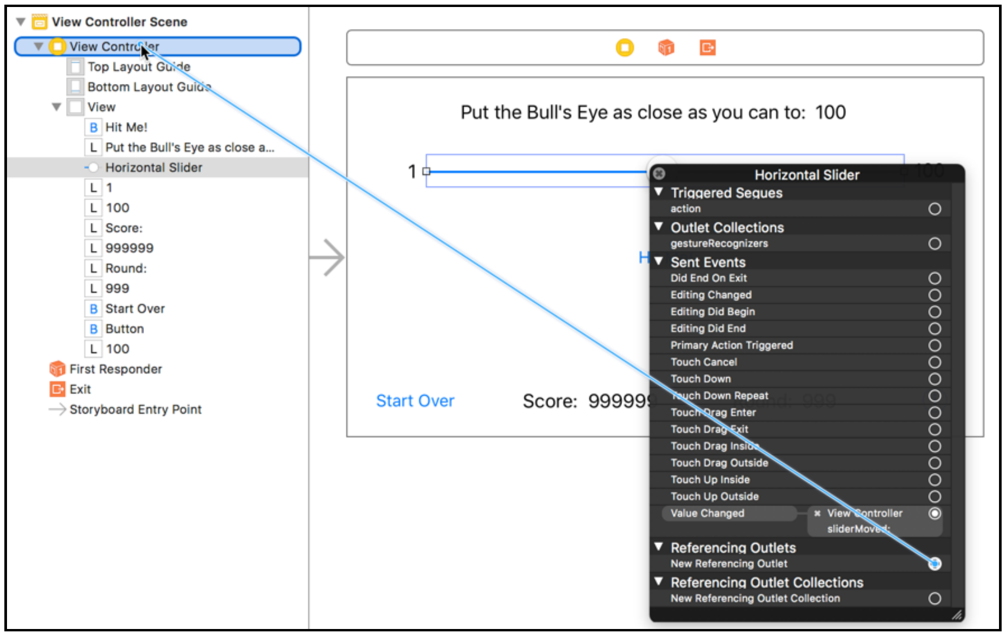
Remember that an outlet has to be *connected* to something in the storyboard. You defined the variable, but you didn't actually set up the connection yet. So, when the app ran and `viewDidLoad()` was called, it tried to find the matching connection in the storyboard and could not - and crashed.

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

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



Connecting the slider to the outlet

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

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

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

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

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

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

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

> **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.

Comments

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

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

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

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

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

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

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

Generate the random number

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
    currentValue = lroundf(slider.value)
    targetValue = 1 + Int(arc4random_uniform(100))
  }
```

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

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

Display the random number

➤ Change `showAlert()` to the following:

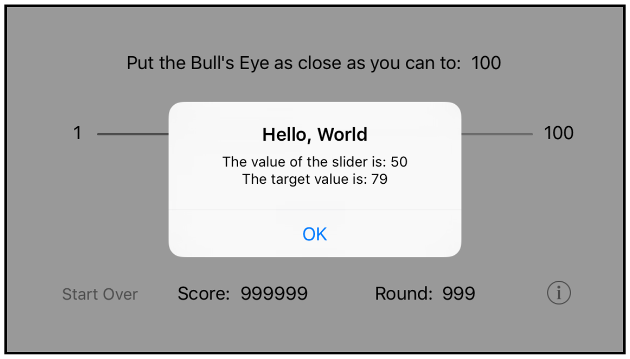
```
  @IBAction func showAlert() {
    let message = "The value of the slider is: \(currentValue)" +
                  "\nThe target value is: \(targetValue)"

    let alert = . . .
  }
```

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

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

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

➤ Run the app and try it out!


Put the Bull's Eye as close as you can to: 100

Hello, World

The value of the slider is: 50
The target value is: 79

The alert shows the target value on a new line

Note: Earlier you've used the + operator to add two numbers together (just like how it works in math) but here you're also using + to glue different bits of text into one big string.

Swift allows the use of the same operator for different tasks, depending on the data types involved. If you have two integers, + adds them up. But with two strings, + concatenates, or combines, them into a longer string.

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 :]

Add 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.

Start 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 = 1 + Int(arc4random_uniform(100))
  currentValue = 50
```

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

It doesn’t really matter where you put it, 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.

Use 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()
  }
```

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. The iPhone will then go 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.

Different ways to call methods

Sometimes you may see method calls written like this:

```
self.startNewRound()
```

That does the exact same thing as just `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 most 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 = 1 + Int(arc4random_uniform(100))
  currentValue = 50
  slider.value = Float(currentValue)
}

@IBAction func showAlert() {
  . . .

  targetValue = 1 + Int(arc4random_uniform(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 would have to duplicate will 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 this change in two places as well.

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.

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 = 1 + Int(arc4random_uniform(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 that 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 and look inside.

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(…)` and `Int(…)` do in these lines:

```
targetValue = 1 + Int(arc4random_uniform(100))
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 puts it into a new `Float` value that it can pass on to the `UISlider`.

Something similar happens with `arc4random_uniform()`, where the random number gets converted to an `Int` first before it can be stored in `targetValue`.

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, as you've done here.

Display 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...

Set up the storyboard

When you made the storyboard, you already 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

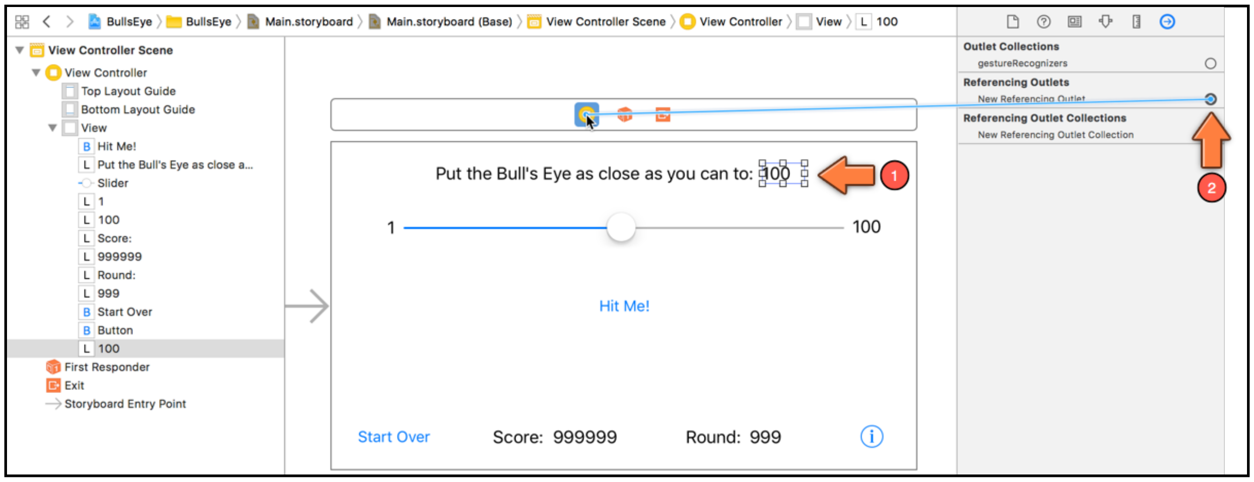
This will be very similar to what you did with the slider. Recall that you added an `@IBOutlet` variable so you could reference the slider anywhere from within the view controller. Using this outlet variable you could ask the slider for its value, through `slider.value`. You'll do the same thing for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

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

```
@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.)



Connecting the target value label to its outlet

➤ Select **targetLabel** from the popup, and the connection is made.

Display the target value via code

➤ Now 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 into its own 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(…)` and `Int(…)`.

Just in case you were wondering, you could also convert `targetValue` to a `String` by using it as a string with a placeholder 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` anywhere.)

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` specifier. This 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.

Call the method

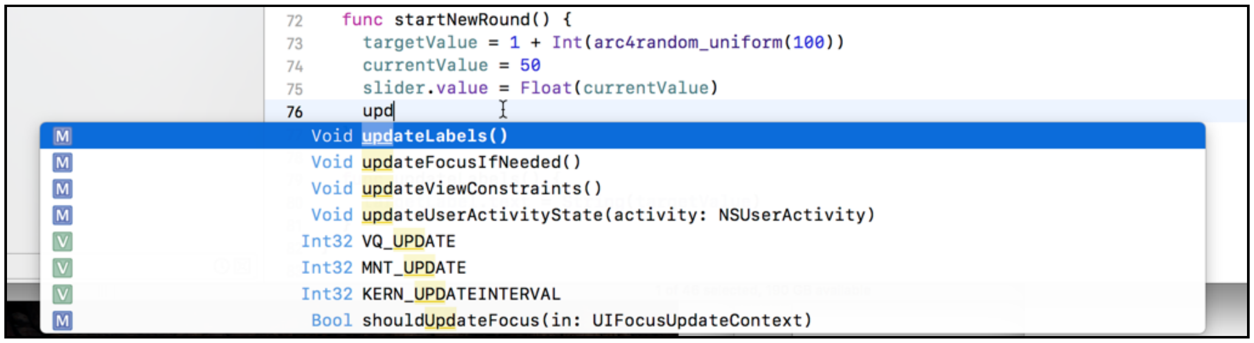
The logical place to call `updateLabels()` would be after each call to `startNewRound()`, because that is where you calculate the new target value. So, you could always add a call to `updateLabels()` in `viewDidLoad()` and `showAlert()`, but there's another way too!

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 = 1 + Int(arc4random_uniform(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):

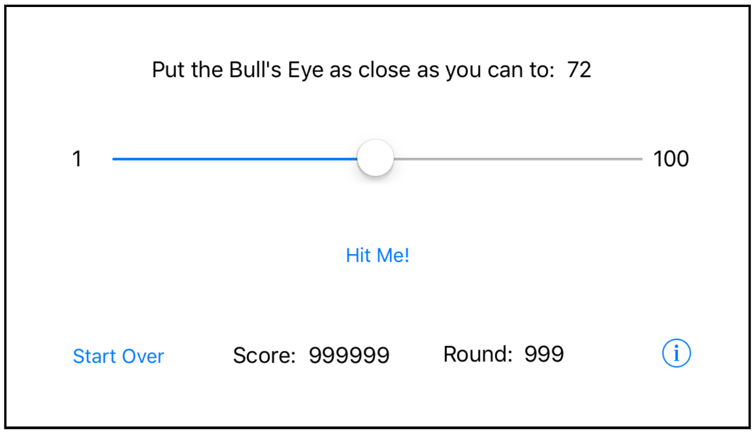


```
func startNewRound() {
    targetValue = 1 + Int(arc4random_uniform(100))
    currentValue = 50
    slider.value = Float(currentValue)
    upd
M   Void  updateLabels()
M   Void  updateFocusIfNeeded()
M   Void  updateViewConstraints()
M   Void  updateUserActivityState(activity: NSUserActivity)
V  Int32  VQ_UPDATE
V  Int32  MNT_UPDATE
V  Int32  KERN_UPDATEINTERVAL
M   Bool  shouldUpdateFocus(in: UIFocusUpdateContext)
```

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 "date" or "label" should help you find "updateLabels" just as easily.

➤ 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

You can find the project files for the app up to this point under **04 - Outlets** in the Source Code folder.

Chapter 5: Rounds and Score

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:

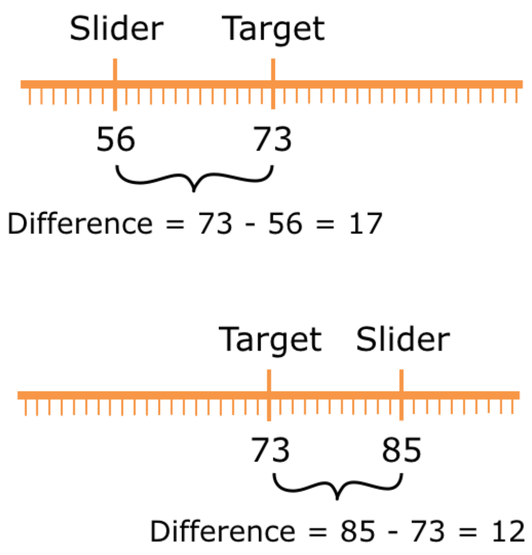
- **Get the difference:** Calculate the difference between the target value and the value that the user selected.
- **Other ways to calculate the difference:** Other approaches to calculating the difference.
- **What's the score?:** Calculate the user's score based on the difference value.
- **The total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

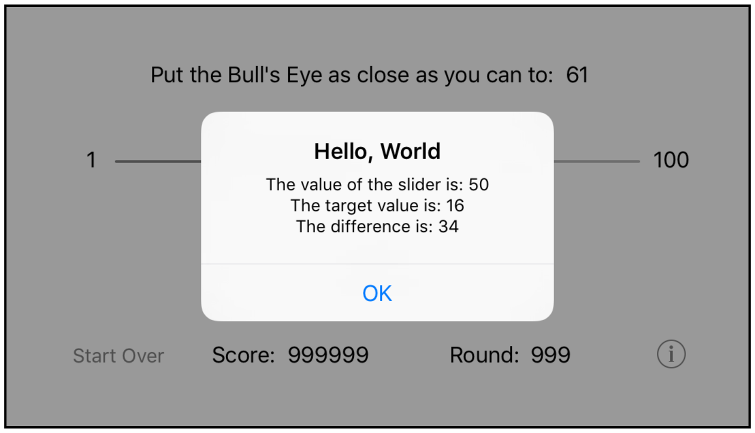
Display the difference

➤ 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 value of the slider is: 50
The target value is: 16
The difference is: 34
```

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?

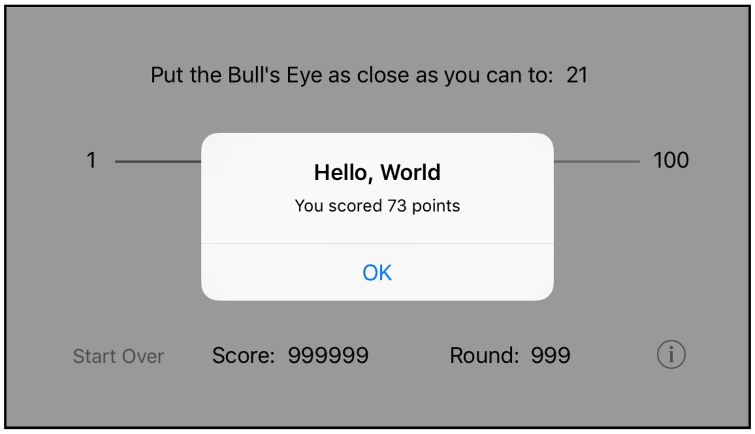
Now that you know how far off the slider is from the target, calculating the player's score for each round is easy.

➤ 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.

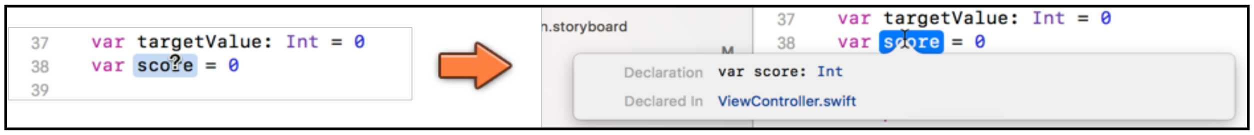
➤ Add a new `score` instance variable to **ViewController.swift**:

```
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.



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

```
Declaration   var score: Int
Declared In   ViewController.swift
```

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 :]

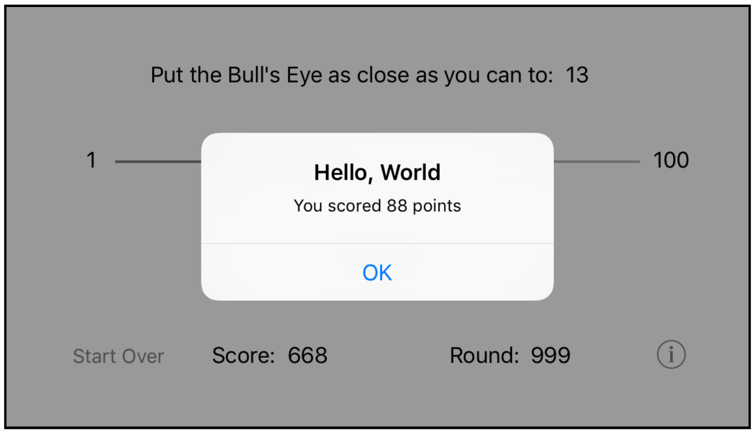
Great, that gives you a `scoreLabel` outlet that you can use to display the score. Now where in the code can you do that? In `updateLabels()`, of course.

➤ 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?

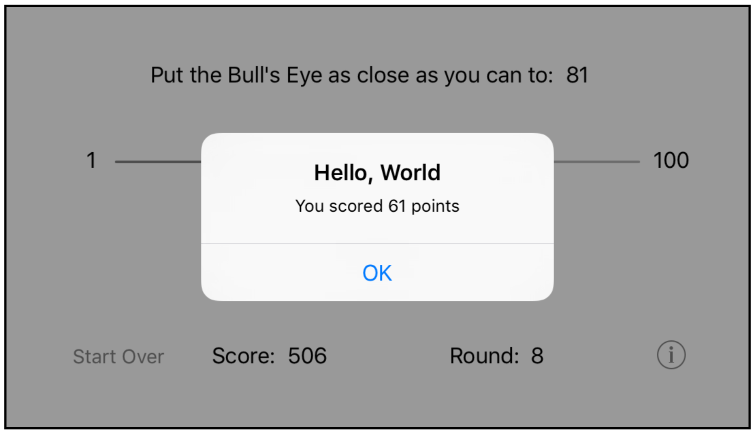
I’d say the `startNewRound()` method is a pretty good place. After all, you call this method whenever you start a new round. It makes sense to increment the round counter there.

➤ 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 **05 - 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.

Chapter 6: Polish

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 … :])

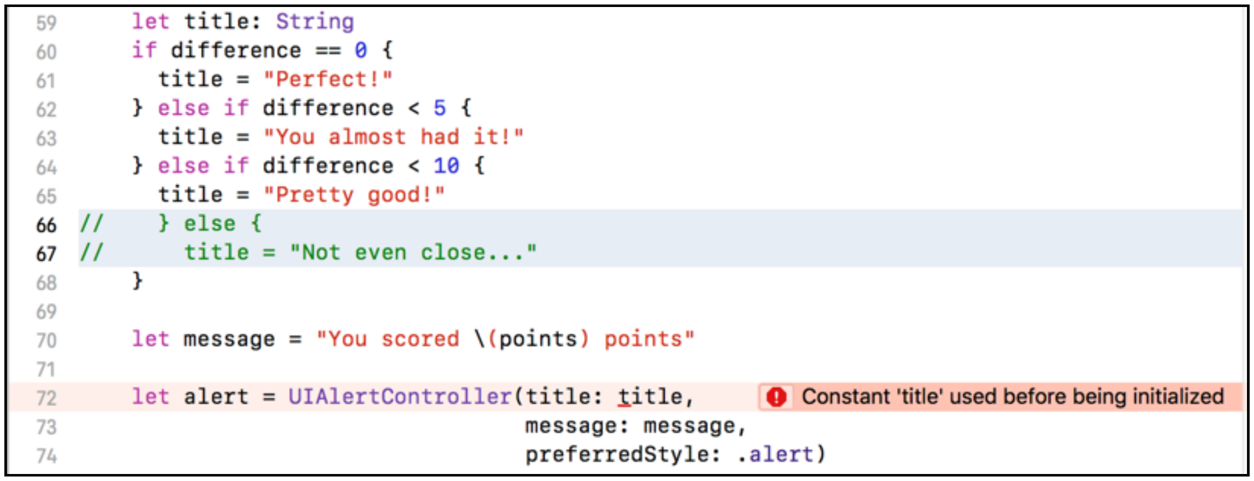
One way to fix this would be to declare `title` as a variable rather than a constant. Like this:

```
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".

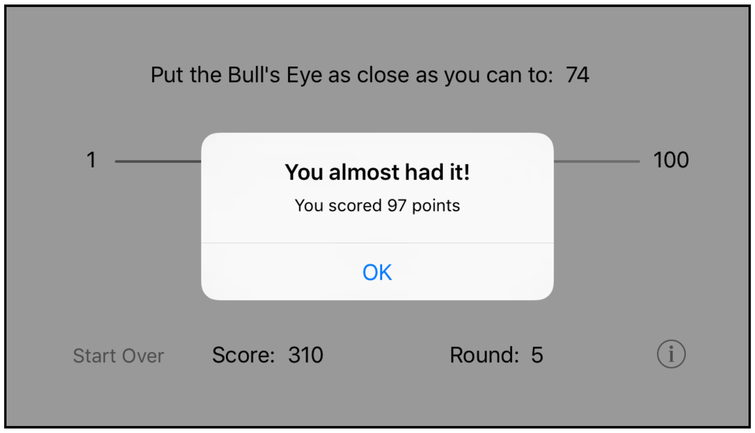


```
    let title: String
    if difference == 0 {
      title = "Perfect!"
    } else if difference < 5 {
      title = "You almost had it!"
    } else if difference < 10 {
      title = "Pretty good!"
//    } else {
//      title = "Not even close..."
    }

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

    let alert = UIAlertController(title: title,
                                  message: message,
                                  preferredStyle: .alert)
```

Constant 'title' used before being initialized

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!



Put the Bull's Eye as close as you can to: 74

You almost had it!

You scored 97 points

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.

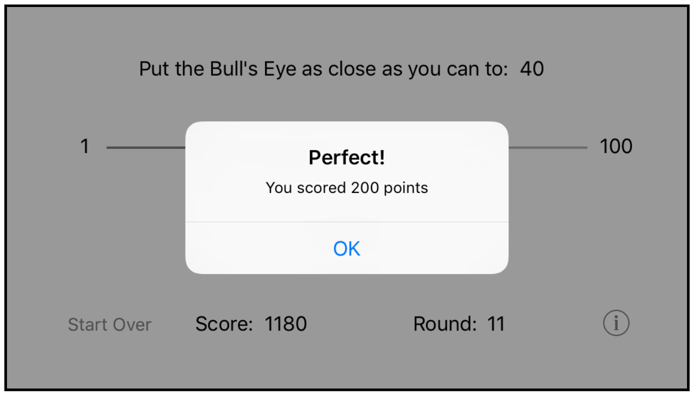
- The second `if` has changed too. There is now an `if` inside another `if`. Nothing wrong with that! You want to handle the case where `difference` is 1 in order to give the player bonus points. That happens inside the new `if` statement.

  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.

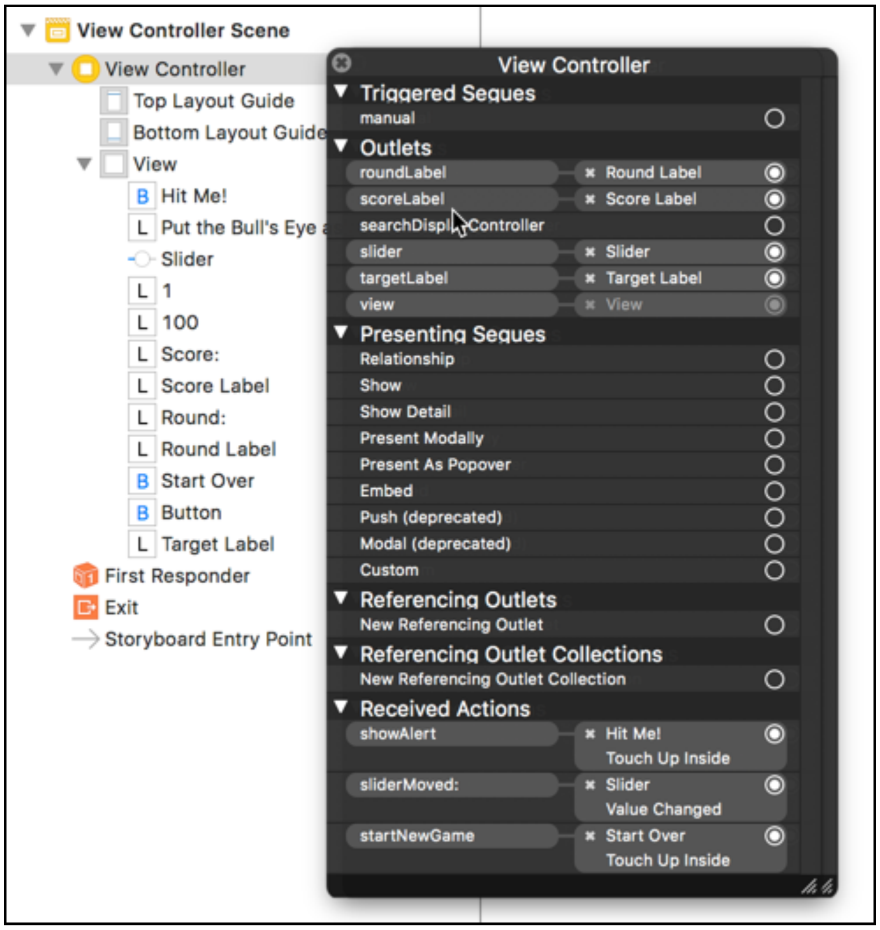
➤ Open the storyboard and Control-drag from the **Start Over** button to **View Controller**. Let go of the mouse button and pick **startNewGame** from the popup if you opted to have `startNewGame()` as the action method. Otherwise, pick the name of your action method .

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.



View Controller

Triggered Segues
manual

Outlets
roundLabel — Round Label
scoreLabel — Score Label
searchDisplayController
slider — Slider
targetLabel — Target Label
view — View

Presenting Segues
Relationship
Show
Show Detail
Present Modally
Present As Popover
Embed
Push (deprecated)
Modal (deprecated)
Custom

Referencing Outlets
New Referencing Outlet

Referencing Outlet Collections
New Referencing Outlet Collection

Received Actions
showAlert — Hit Me! Touch Up Inside
sliderMoved: — Slider Value Changed
startNewGame — Start Over Touch Up Inside

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 **06 - Polish** in the Source Code folder.