Chapter 3: Outlets

By Fahim Farook and Matthijs Hollemans

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.

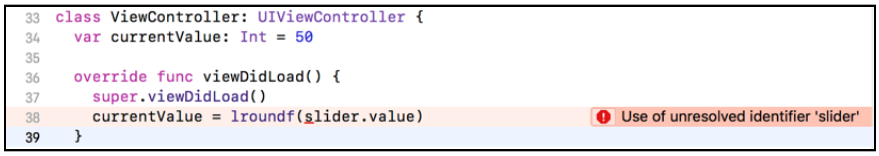
➤ Change `viewDidLoad()` to the following:

```
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.



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

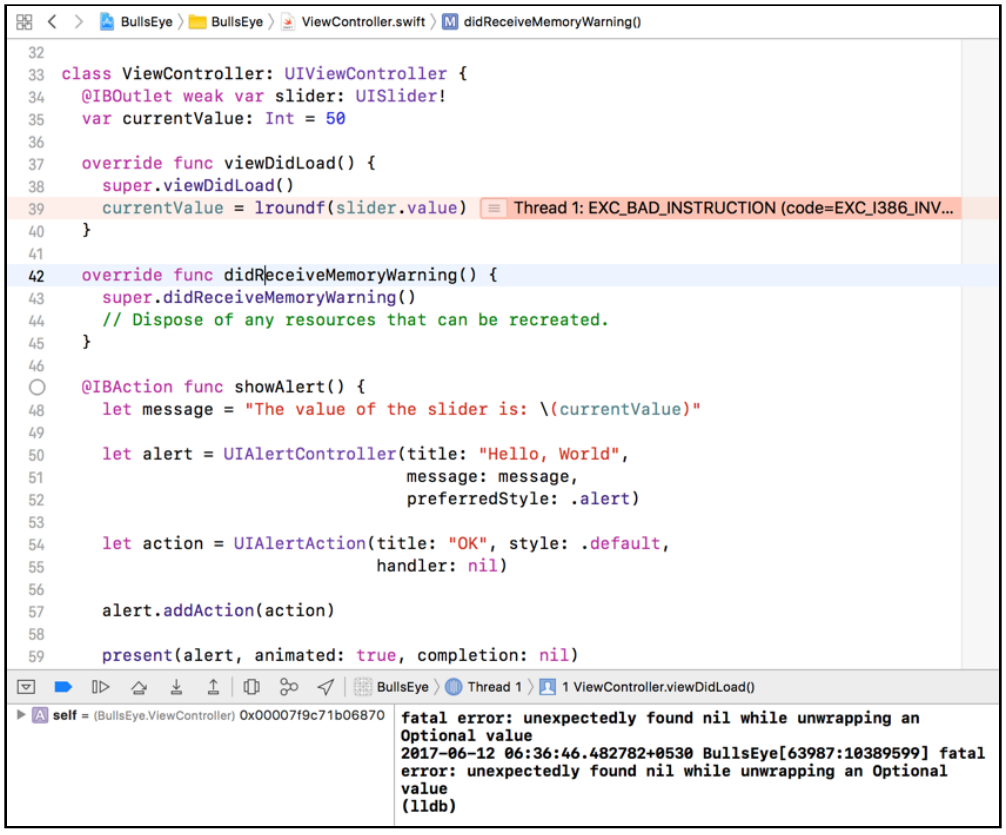
variables - at the top of the class implementation.

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:



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

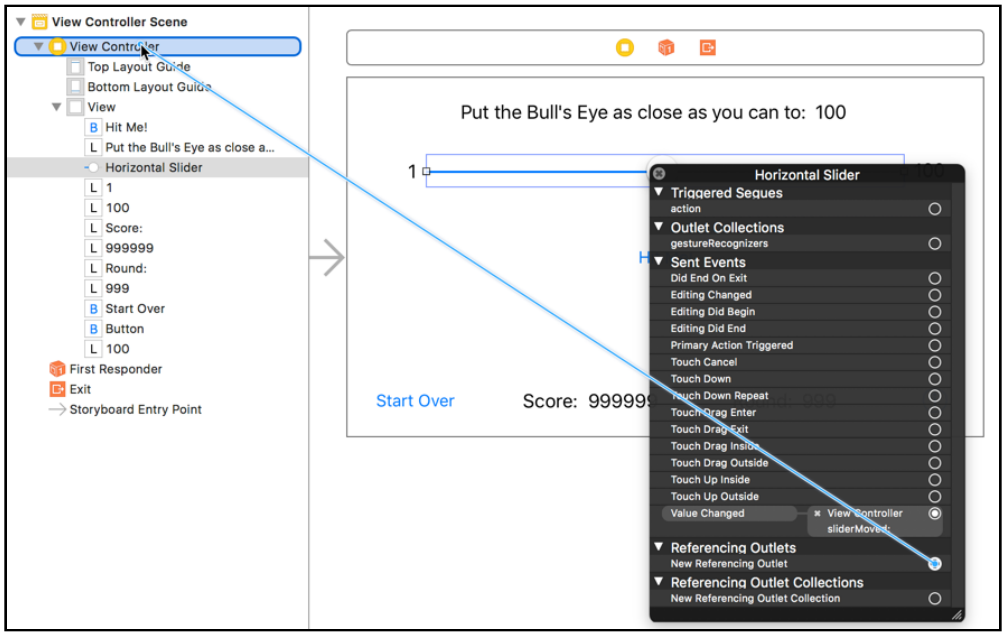
ran and `viewDidLoad()` was called, it tried to find the matching connection in the storyboard and could not - and crashed.

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:

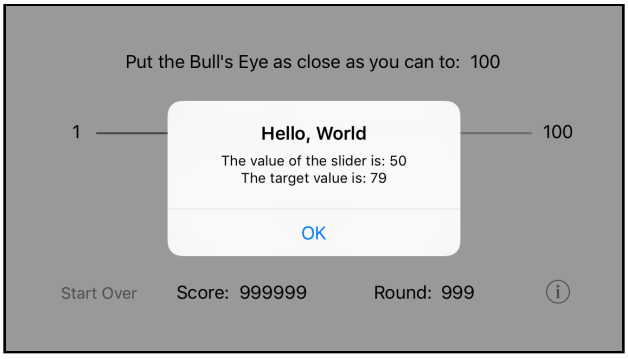
```
  @IBAction func showAlert() {
    let message = "The value of the slider is: \(currentValue)" +
                  "\nThe target value is: \(targetValue)"

    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!



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

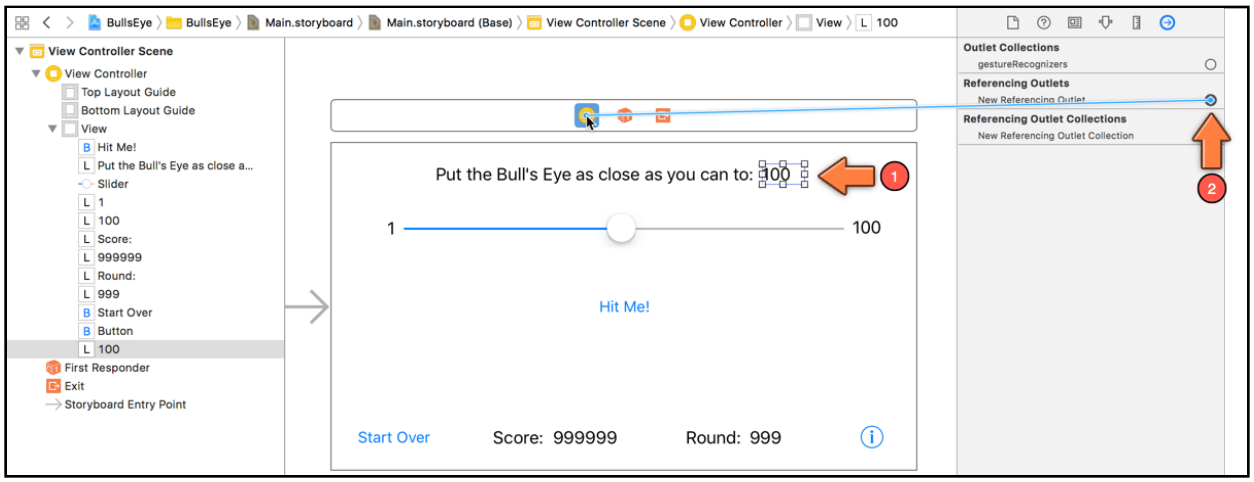
This will be very similar to what you did with the slider. Recall that you added an `@IBOutlet` variable so you could reference the slider anywhere from within the view controller. Using this outlet variable you could ask the slider for its value, through `slider.value`. You'll do the same thing for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

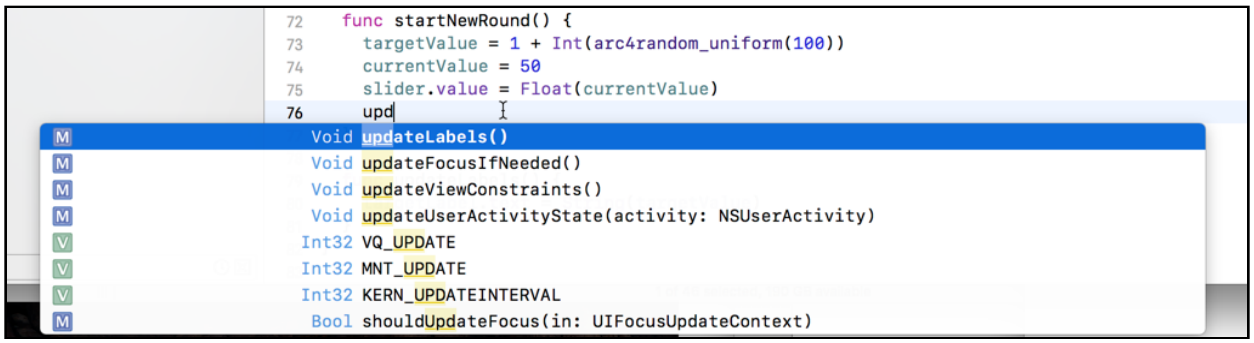
The logical place to call `updateLabels()` would be after each call to `startNewRound()`, because that is where you calculate the new target value. So, you could always add a call to `updateLabels()` in `viewDidLoad()` and `showAlert()`, but there's another way too!

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
}
```

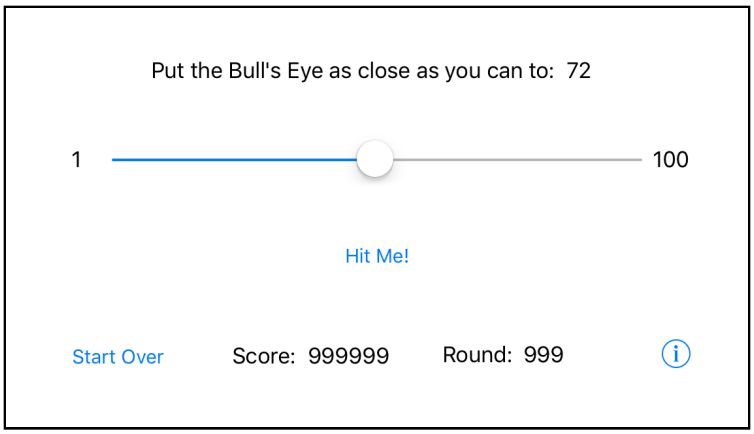
You should be able to type just the first few letters of the method name, like **upd**, and Xcode will show you a list of suggestions matching what you typed. Press **Enter** (or **Tab**) to accept the suggestion (if you are on the right item - or scroll the list to find the right item and then press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



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 **03 - Outlets** in the Source Code folder.