Chapter 2: Slider and Labels

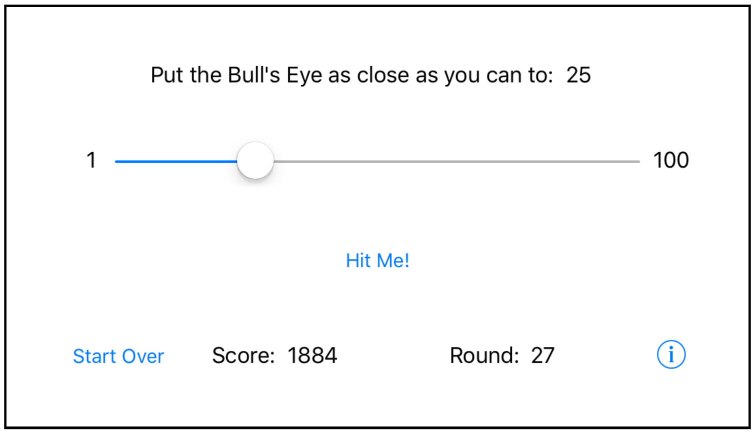
By Fahim Farook and Matthijs Hollemans

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.

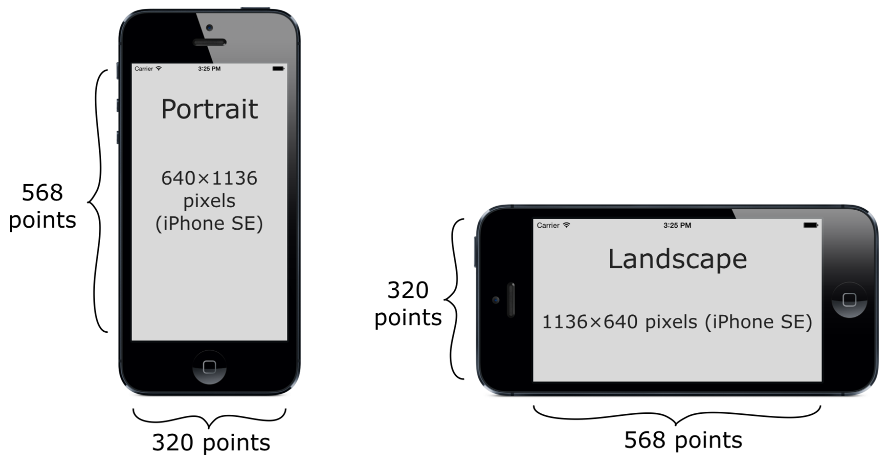
Portrait vs. landscape

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.

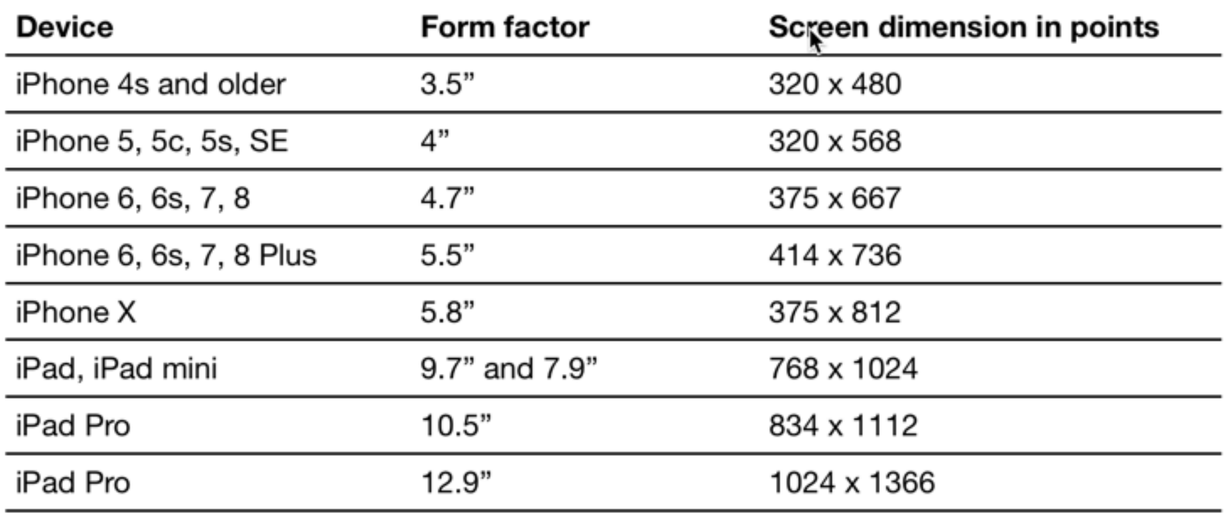
I'm sure you know what a pixel is? In case you don't, it's the smallest element that a screen is made up of. (That's how the word originated, a shortened form of pictures, PICS or PIX + ELement = PIXEL.) The display of your iPhone is a big matrix of pixels that each can have their own color, just like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

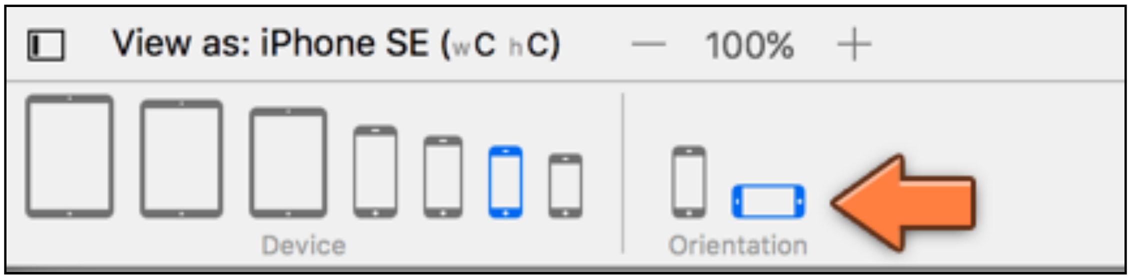
Convert the app to landscape

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.

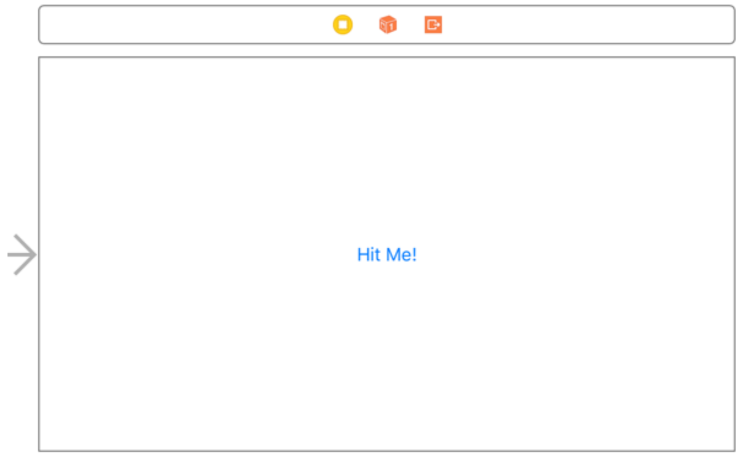
➤ Open **Main.storyboard**. In Interface Builder, in the **View as: iPhone SE** panel, change **Orientation** to landscape:



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.

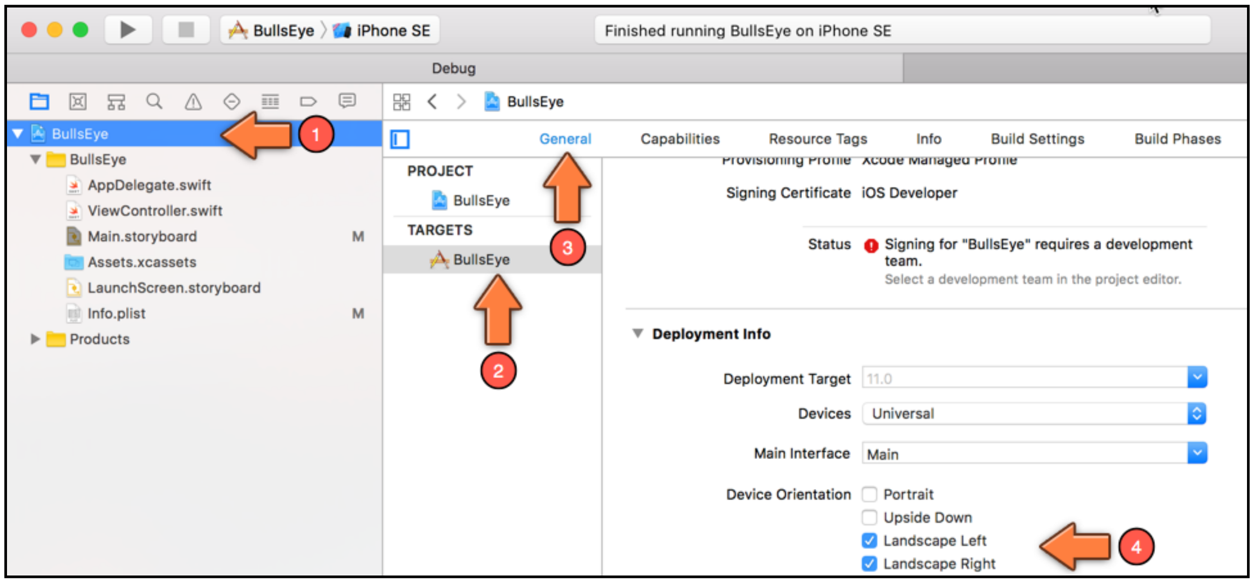
Notice that in landscape orientation the app no longer shows the iPhone's status bar.

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:



Provisioning Profile Xcode Managed Profile

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
- [x] Landscape Left
- [x] 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*:

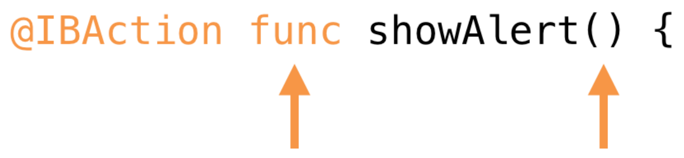
- An example of data is the Hit Me button that you added to the view controller earlier. When you dragged the button into the storyboard, it actually became part of the view controller's data. Data *contains* something. In this case, the view controller contains the button.

- 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:



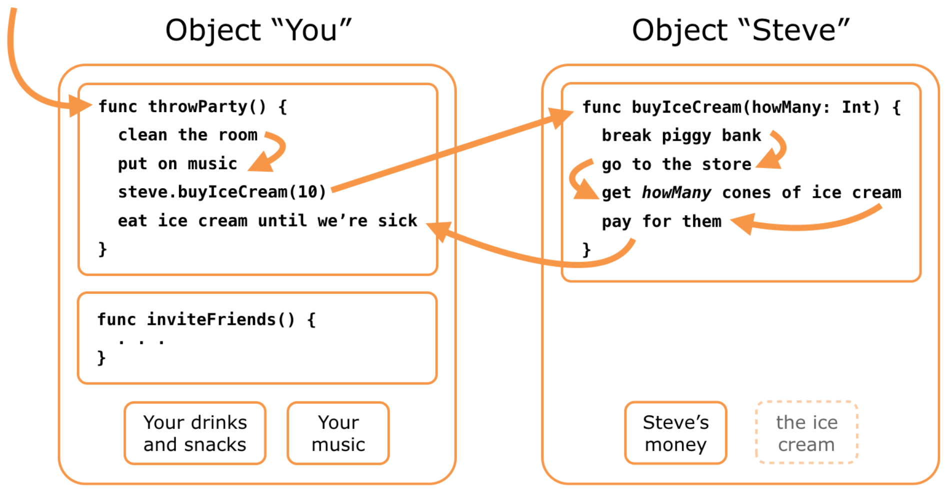
```
@IBAction func showAlert() {
```

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.

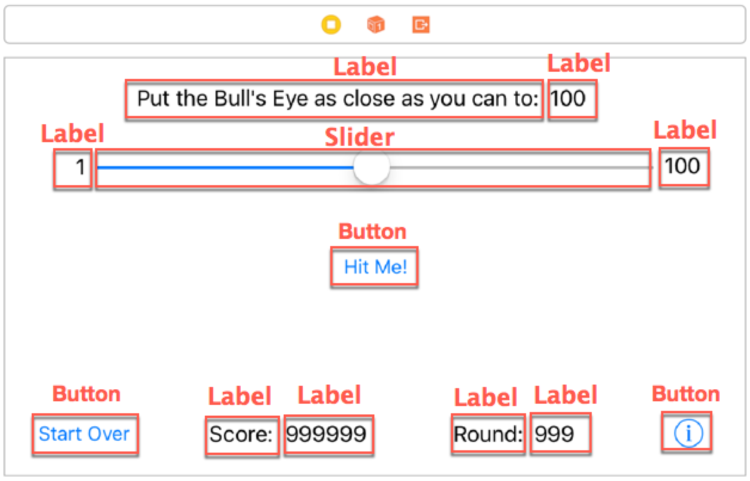
Often the terms "calling a method" or "invoking a method" are used instead. That means the exact same thing as sending a message: the computer jumps to the method you're calling and returns to where it left off when that method is done.

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:

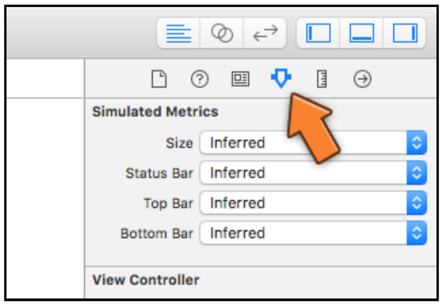


The different views in the game screen

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

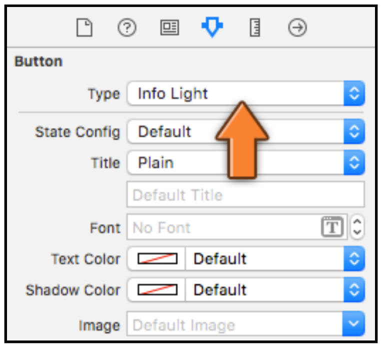
Size: Inferred
Status Bar: Inferred
Top Bar: Inferred
Bottom Bar: Inferred

View Controller

The Attributes inspector

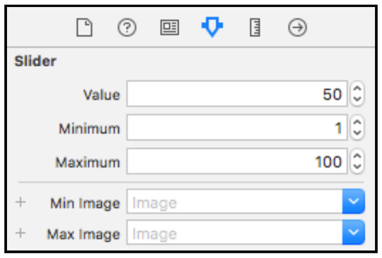
The inspector area shows various aspects of the item that is currently selected. The Attributes inspector, for example, lets you change the background color of a label or the size of the text on a button. You've already seen the Connections inspector that showed the button's actions. As you become more 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.

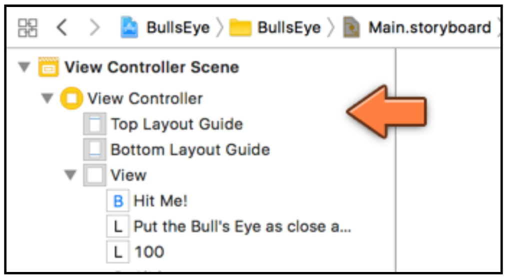
The steps for adding this action are largely the same as before.

➤ 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)")
}
```

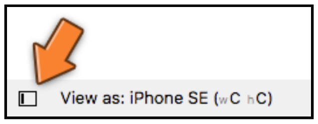
➤ Second, go to the storyboard and Control-drag from the slider to View Controller in the Document Outline. Let go of the mouse button and select **sliderMoved:** from the popup. Done!

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:

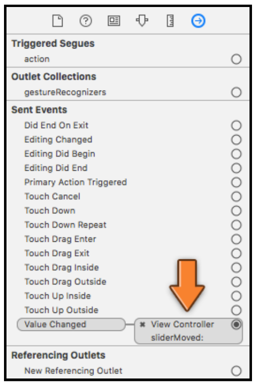


This button shows or hides the Document Outline

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**:

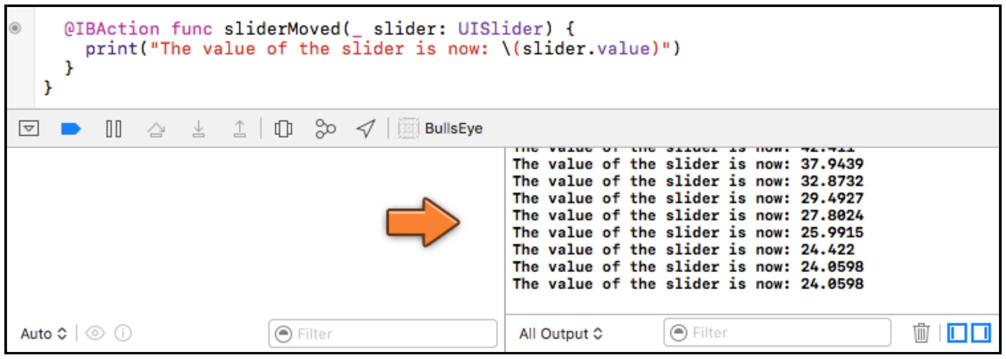


The slider is now hooked up to the view controller

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:

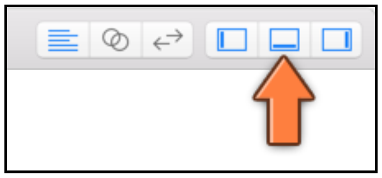


```
    @IBAction func sliderMoved(_ slider: UISlider) {
      print("The value of the slider is now: \(slider.value)")
    }
  }
```

```
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.

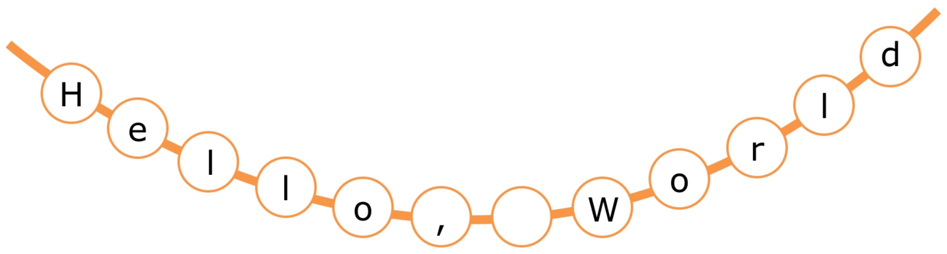
Strings

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.

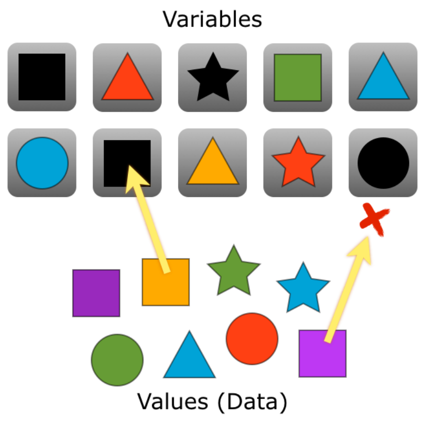
You don't just put stuff in the container and then forget about it. You will often replace its contents with a new value. When the thing that your app needs to remember changes, you take the old value out of the box and put in the new value.

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.

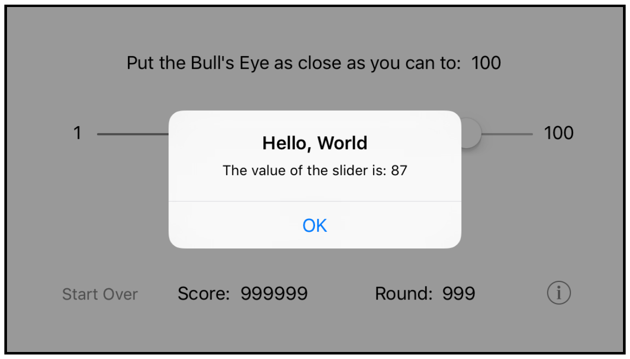
Note: Anything appearing after two slashes `//` (and up to the end of that particular line) in Swift source code is treated as a comment - a note by the developer to themselves, or to other developers. The Swift compiler generally ignores comments - they are there for the convenience of humans.

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 **02 - Slider and Labels** in the Source Code folder.