IOS APP 1: PART 1 & PART 2

PREPARED BY FARHAJ AHMED

iOS App 1: Intro to Application Development

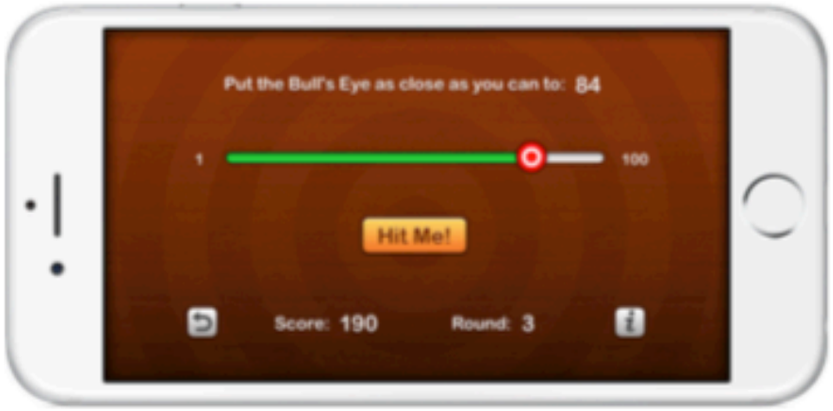
There's an old Chinese proverb that states, "A journey of a thousand miles begins with a single step." You are about to take that first step on your journey to iOS developer mastery. And you will take that first step by creating the *Bull's Eye* game.

This chapter covers the following:

- **The Bull's Eye game**: An introduction to the first app you'll make.
- **The one-button app**: Creating a simple one-button app in which the button can take an action based on a tap on the button.
- **The anatomy of an app**: A brief explanation as to the inner-workings of an app.

The Bull's Eye game

This is what the *Bull's Eye* game will look like when you're finished:

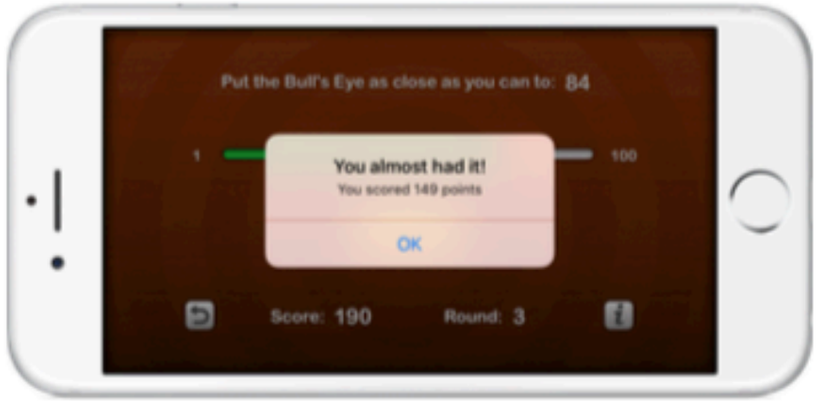


The finished Bull's Eye game

iOS App 1: Intro to Application Development

The objective of the game is to put the bull's eye, which is on a slider that goes from 1 to 100, as close to a randomly chosen target value as you can. In the screenshot above, the aim is to put the bull's eye at 84. Because you can't see the current value of the slider, you'll have to "eyeball" it.

When you're confident of your estimate, you press the "Hit Me!" button and a pop-up, also known as an alert, will tell you what your score is:



An alert pop-up shows the score

The closer to the target value you are, the more points you score. After you dismiss the alert pop-up by pressing the OK button, a new round begins with a new random target. The game repeats until the player presses the "Start Over" button (the curly arrow in the bottom-left corner), which resets the score to 0.

This game probably won't make you an instant millionaire on the App Store, but even future millionaires have to start somewhere!

Making a programming to-do list

> **Exercise**: Now that you've seen what the game will look like and what the gameplay rules are, make a list of all the things that you think you'll need to do in order to build this game. It's OK if you draw a blank, but give it a shot anyway.

I'll give you an example:

The app needs to put the "Hit Me!" button on the screen and show an alert pop-up when the user presses it.

Try to think of other things the app needs to do — it doesn't matter if you don't actually know how to accomplish these tasks. The first step is to figure out *what* you need to do; *how* to do these things is not important yet.

iOS App 1: Intro to Application Development

Once you know what you want, you can also figure out how to do it, even if you have to ask someone or look it up. But the "what" comes first. You'd be surprised at how many people start writing code without a clear idea of what they're actually trying to achieve. No wonder they get stuck!

Whenever I start working on a new app, I first make a list of all the different pieces of functionality I think the app will need. This becomes my programming to-do list. Having a list that breaks up a design into several smaller steps is a great way to deal with the complexity of a project.

You may have a cool idea for an app, but when you sit down to write the program it can seem overwhelming. There is so much to do... and where to begin? By cutting up the workload into small steps you make the project less daunting – you can always find a step that is simple and small enough to make a good starting point and take it from there.

It's no big deal if this exercise gives you some difficulty. You're new to all of this! As your understanding grows of how software and the development process works, it will become easier to identify the different parts that make up a design and to split it into manageable pieces.

This is what I came up with. I simply took the gameplay description and split it into very small chunks:

- Put a button on the screen and label it "Hit Me!"
- When the player presses the Hit Me! button, the app has to show an alert pop-up to inform the player how well he or she did. Somehow, you have to calculate the score and put that into this alert.
- Put text on the screen, such as the "Score:" and "Round:" labels. Some of this text changes over time; for example, the score, which increases when the player scores points.
- Put a slider on the screen with a range between the values 1 and 100.
- Read the value of the slider after the user presses the Hit Me! button.
- Generate a random number at the start of each round and display it on the screen. This is the target value.
- Compare the value of the slider to that random number and calculate a score based on how far off the player is. You show this score in the alert pop-up.

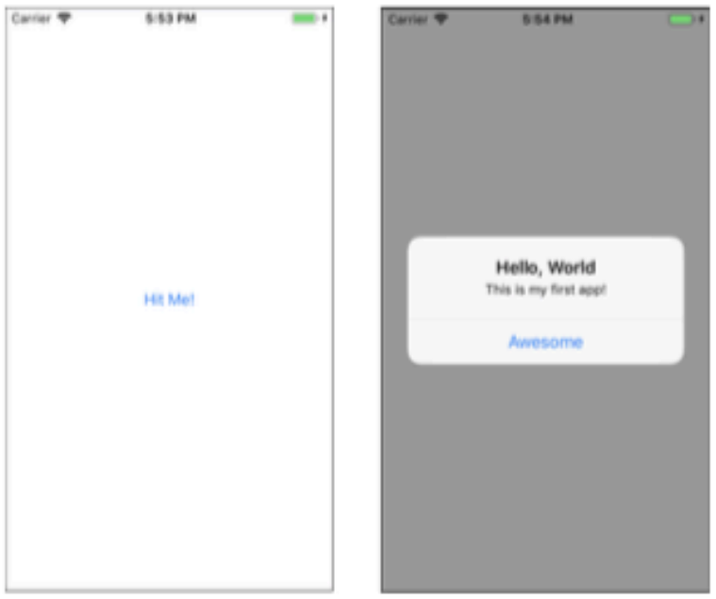
- Put the Start Over button on the screen. Make it reset the score and put the player back to the first round.
- Put the app in landscape orientation.
- Make it look pretty.

I might have missed a thing or two, but this looks like a decent list to start with. Even for a game as basic as this, there are quite a few things you need to do. Making apps is fun, but it's definitely a lot of work, too!

The one-button app

Let's start at the top of the list and make an extremely simple first version of the game that just displays a single button. When you press the button, the app pops up an alert message. That's all you are going to do for now. Once you have this working, you can build the rest of the game on this foundation.

The app will look like this:



The app contains a single button (left) that shows an alert when pressed (right)

Time to start coding! I'm assuming you have downloaded and installed the latest version of Xcode at this point.

In this book, you'll work with **Xcode 10.0** or better. Newer versions of Xcode may also work, but anything older than version 10.0 probably would be a no-go.

iOS App 1: Intro to Application Development

Because Swift is a very new language, it tends to change between versions of Xcode. If your Xcode is too old — or too new! — then not all of the code in this book may work properly. For this same reason, you're advised not to use beta versions of Xcode, only the official one from the Mac App Store.

Creating a new project

➤ Launch Xcode. If you have trouble locating the Xcode application, you can find it in the **/Applications/Xcode** folder or in your Launchpad. Because I use Xcode all of the time, I've placed it in my dock for easy access.

Xcode shows the "Welcome to Xcode" window when it starts:

Xcode bids you welcome

➤ Choose **Create a new Xcode project**. The main Xcode window appears with an assistant that lets you choose a template:

Choosing the template for the new project

iOS App 1: Intro to Application Development

Xcode has bundled templates for a variety of app styles. Xcode will make a pre-configured project for you based on the template you choose. The new project will already include some of the source files you need. These templates are handy because they can save you a lot of typing. They are ready-made starting points.

➤ Select **Single View Application** and press **Next**.

This opens a dialog wherein you can enter options for the new app.

Configuring the new project

➤ Fill out these options as follows:

- Product Name: **BullsEye**. If you want to use proper English, you can name the project Bull's Eye instead of BullsEye, but it's best to avoid spaces and other special characters in project names.
- Team: If you're already a member of the Apple Developer Program, this will show your team name. For now, it's best to leave this setting alone; we'll get back to this later on.
- Organization Name: Fill in your own name here or the name of your company.
- Organization Identifier: Mine says "com.razeware". That is the identifier I use for my apps. As is customary, it is my domain name written in reverse. You should use your own identifier here.

iOS App 1: Intro to Application Development

Pick something that is unique to you, either the domain name of your website (but backwards) or simply your own name. You can always change this later.

- Language: **Swift**

Make sure the three options at the bottom — Use Core Data, Include Unit Tests, and Include UI Tests — are *not* selected. You won't use those in this project.

➤ Press **Next**. Now, Xcode will ask where to save your project:

Choosing where to save the project

➤ Choose a location for the project files; for example, the Desktop or your Documents folder.

Xcode will automatically make a new folder for the project using the Product Name that you entered in the previous step (in your case, BullsEye), so you don't need to make a new folder yourself.

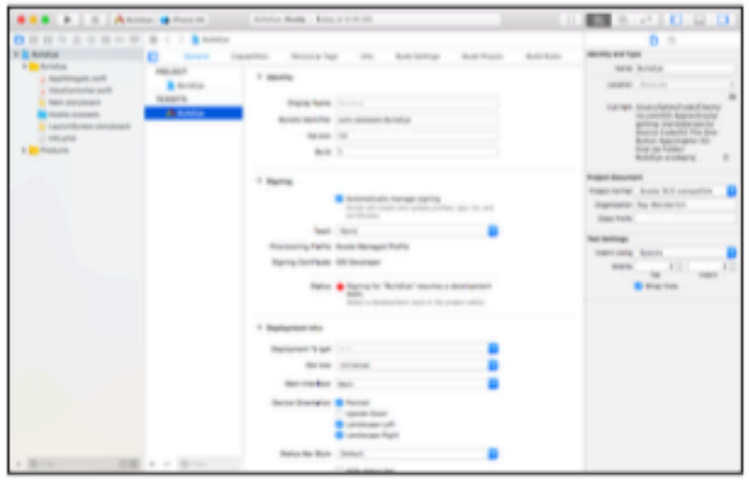
At the bottom of the File Save dialog, there is a checkbox that says, "Create Git repository on My Mac." You can ignore this for now. You'll learn about the Git version control system later on.

➤ Press **Create** to finish.

Xcode will now create a new project named BullsEye, based on the Single View Application template, in the folder you specified.

iOS App 1: Intro to Application Development

When it is done, the screen should look something like this:



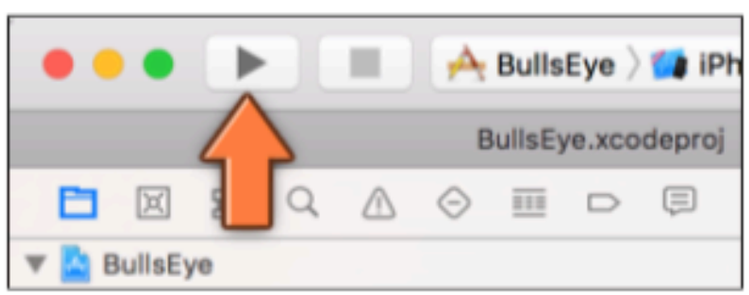
The main Xcode window at the start of your project

There may be small differences with what you're seeing on your own computer if you're using a version of Xcode newer than my version. Rest assured, any differences will only be superficial.

> **Note**: If you don't see a file named ViewController.swift in the list on the left but instead have ViewController.h and ViewController.m, then you picked the wrong language (Objective-C) when you created the project. Start over and be sure to choose Swift as the programming language.

Running your project

➤ Press the **Run** button in the top-left corner.

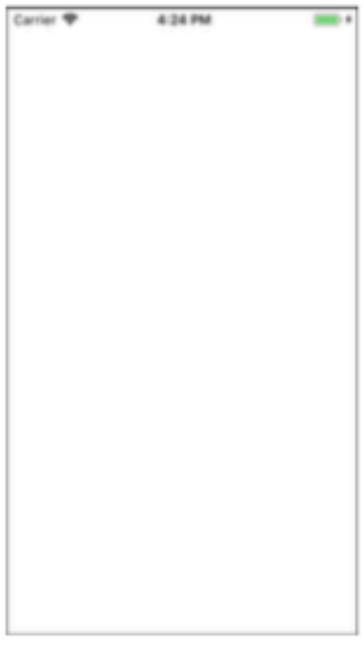


Press Run to launch the app

iOS App 1: Intro to Application Development

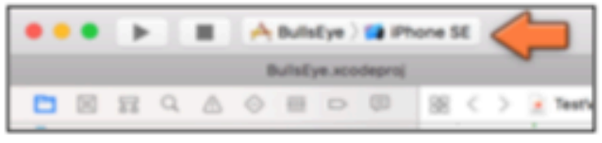
> **Note**: If this is the first time you're using Xcode, it may ask you to enable developer mode. Click **Enable** and enter your password to allow Xcode to make these changes.
>
> Also, make sure that you do not have your iPhone or iPad plugged in at this point to your computer — for example, for charging. If you do, it might switch to the actual device instead of the simulator for running the app and, since you are not yet set up for running on a device, this could result in errors that might leave you scratching your head.

Xcode will labor for a bit and then launch your brand new app in iOS Simulator. The app may not look like much yet — and there is not anything you can do with it either — but this is an important first milestone in your journey!



What an app based on the Single View Application template looks like

If Xcode says "Build Failed" or "Xcode cannot run using the selected device" when you press the Run button, then make sure the picker at the top of the window says **BullsEye > iPhone SE** (or any other model number) and not **Generic iOS Device**:



Making Xcode run the app on the Simulator

iOS App 1: Intro to Application Development

If your iPhone is currently connected to your Mac via USB cable, Xcode may have attempted to run the app on your iPhone, and that may not work without some additional setting up. I'll show you how to get the app to run on your iPhone so that you can show it off to your friends soon, but for now just stick with iOS Simulator.

➤ Next to the Run button is the **Stop** button (the square thingy). Press that to exit the app.

On your phone (or even the simulator), you'd use the Home button to exit an app (on the simulator, you could also use the **Hardware ▸ Home** item from the menu bar or use the handy ⇧+⌘+**H** shortcut), but that won't actually terminate the app. It will disappear from the simulator's screen, but the app stays suspended in the simulator's memory, just as it would on a real iPhone.

Until you press Stop, Xcode's Activity viewer at the top says, "Running BullsEye on iPhone SE":

The Xcode activity viewer

It's not really necessary to stop the app, as you can go back to Xcode and make changes to the source code while the app is still running. However, these changes will not become active until you press Run again. That will terminate any running version of the app, build a new version and launch it in the simulator.

What happens when you press Run?

Xcode will first *compile* your source code — that is, translate it – from Swift into machine code that the iPhone (or iOS Simulator) can understand. Even though the programming language for writing iOS apps is Swift or Objective-C, the iPhone itself doesn't speak those languages. A translation step is necessary.

The compiler is the part of Xcode that converts your Swift source code into executable binary code. It also gathers all the different components that make up the app — source files, images, storyboard files and so on — and puts them into the "application bundle."

This entire process is also known as *building* the app. If there are any errors (such as spelling mistakes in your code), the build will fail. If everything goes according to plan, Xcode copies the application bundle to the simulator or the iPhone and launches the app. All that from a single press of the Run button.

iOS App 1: Intro to Application Development

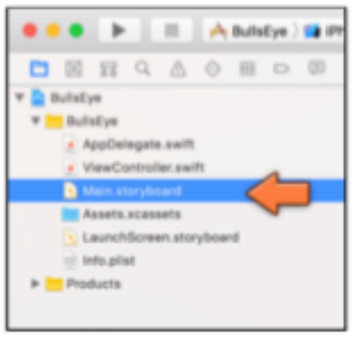
Adding a button

I'm sure you're as unimpressed as I am with an app that just displays a dull white screen! So let's make it a bit more interesting by adding a button to it.

The left pane of the Xcode window is named the **Navigator area**. The row of icons along the top lets you select a specific navigator. The default navigator is the **Project navigator**, which shows the files in your project.

The organization of these files corresponds to the project folder on your hard disk, but that isn't necessarily always so. You can move files around and put them into new groups and organize away to your heart's content. We'll talk more about the different files in your project later.

➤ In the **Project navigator**, find the item named **Main.storyboard** and click it once to select it:



The Project navigator lists the files in the project

Like a superhero changing his or her clothes in a phone booth, the main editing pane now transforms into the **Interface Builder**. This tool lets you drag-and-drop user interface components such as buttons to create the UI of your app. (OK, maybe a bad analogy, but Interface Builder is a super tool, in my opinion.)

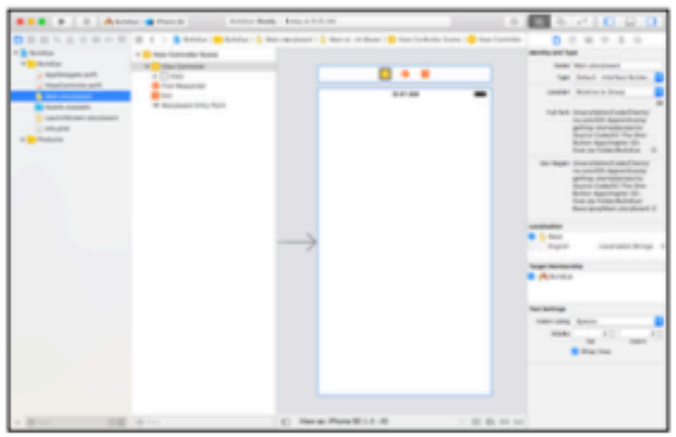
➤ If it's not already blue, click the **Hide or Show the Inspectors** button in Xcode's toolbar.

Click this button to show the Utilities pane

These toolbar buttons in the top-right corner change the appearance of Xcode. This one in particular opens a new pane on the right side of the Xcode window.

iOS App 1: Intro to Application Development

Your Xcode should now look something like this:



Editing Main.storyboard in Interface Builder

This is the *storyboard* for your app. The storyboard contains the designs for all of your app's screens and shows the navigation flow in your app from one screen to another.

Currently, the storyboard contains just a single screen or *scene*, represented by a rectangle in the middle of the Interface Builder canvas.

> **Note**: If you don't see the rectangle labeled "View Controller" but only an empty canvas, then use your mouse or trackpad to scroll the storyboard around a bit. Trust me, it's in there somewhere! Also make sure your Xcode window is large enough. Interface Builder takes up a lot of space…

The scene currently is probably set to the size of an iPhone 8. To keep things simple, you will first design the app for the iPhone SE, which has a slightly smaller screen. Later, you'll also make the app fit on the larger iPhone models.

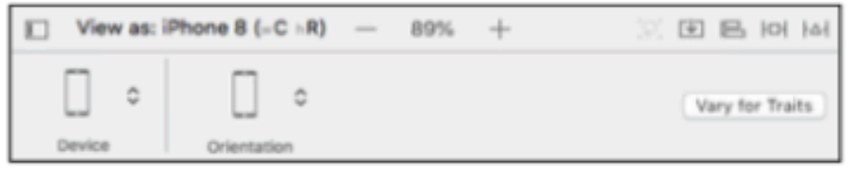
➤ At the bottom of the Interface Builder window, click **View as: iPhone 8** to open up the following panel:

Choosing the device type

iOS App 1: Intro to Application Development

Select the **iPhone SE**, the second smallest iPhone, thus resizing the preview UI you see in Interface Builder to be set to that of an iPhone SE. You'll notice that the scene's rectangle now becomes a bit smaller, corresponding to the screen size of the iPhone 5, iPhone 5s and iPhone SE models.

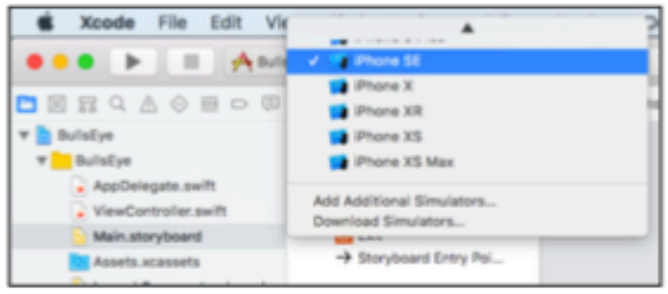
Do note that depending on the size of your Xcode window, the above panel might also look something like this:



Choosing the device type - compact view

If you get this screen, just select the **iPhone SE** from the list of choices you get when you click on **Device**.

➤ In the Xcode toolbar, make sure it says **BullsEye > iPhone SE** (next to the Stop button). If it doesn't, then click it and pick iPhone SE from the list:

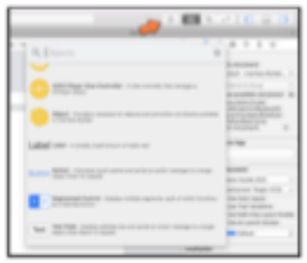


Switching the Simulator to iPhone SE

Now, when you run the app, it will run on the iPhone SE Simulator (try it out!).

Back to the storyboard:

➤ The first button on the top right toolbar shows the **Library** panel when you click it:



The Object Library

iOS App 1: Intro to Application Development

Scroll through the items in the Object Library list until you see **Button**. Alternatively, you can type the word "button" in to the search/filter box at the top.

➤ Click on **Button** and drag it onto the working area, on top of the scene's rectangle.

Dragging the button on top of the scene

That's how easy it is to add most new UI (user interface) items — just drag and drop. You'll do a lot of this, so take some time to get familiar with the process.

Drag and drop a few other controls, such as labels, sliders, and switches, just to get the hang of it. Once you are done, delete everything except for the first button you added.

This should give you some idea of the UI controls that are available in iOS. Notice that the Interface Builder helps you to lay out your controls by snapping them to the edges of the view and to other objects. It's a very handy tool!

➤ Double-click the button to edit its title. Call it Hit Me!

The button with the new title

iOS App 1: Intro to Application Development

It's possible that your button might have a border around it:

The button with a bounds rectangle

This border is not part of the button, it's just there to show you how large the button is. You can turn these borders on or off using the **Editor ▸ Canvas ▸ Show Bounds Rectangles** menu option.

When you're done playing with Interface Builder, press the Run button from Xcode's toolbar. The app should now appear in the simulator, complete with your "Hit Me!" button. However, when you tap the button, it doesn't do anything yet. For that, you'll have to write some Swift code!

Using the source code editor

A button that doesn't do anything when tapped is of no use to anyone. So let's make it show an alert pop-up. In the finished game, the alert will display the player's score; for now, you will limit yourself to a simple text message (the traditional "Hello, World!").

➤ In the **Project navigator**, click on **ViewController.swift**.

The Interface Builder will disappear and the editor area now contains a bunch of brightly colored text. This is the Swift source code for your app:

The source code editor

iOS App 1: Intro to Application Development

> **Note**: If your Xcode editor window does not show the line numbers as in the screenshot above, and you'd actually like to see the line numbers, from the menu bar choose **Xcode ▸ Preferences... ▸ Text Editing** and go to the **Editing** tab. There, you should see a **Line numbers** checkbox under **Show** - check it.

➤ Add the following lines directly **above** the very last } bracket in the file:

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

The source code for **ViewController.swift** should now look like this:

```
import UIKit

class ViewController: UIViewController {
  override func viewDidLoad() {
    super.viewDidLoad()
    // Do any additional setup after loading the view, typically from a
nib.
  }

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

Yes, the above code leaves out the bunch of green text above the code with a copyright notice. And that's totally fine since the green text is just a comment block, or notes for developers, and not a necessary part of the code. Before I can tell you what the above code means, I have to introduce the concept of a view controller.

> **Xcode will autosave**
>
> You don't have to save your files after you make changes to them because Xcode will automatically save any modified files when you press the Run button.
>
> Nevertheless, Xcode isn't the most stable piece of software out there and, occasionally, it may crash on you before it has had a chance to save your changes.
>
> Therefore, do remember to press **⌘+S** on a regular basis to save any changes that you've made.

View controllers

You've edited the **Main.storyboard** file to build the user interface of the app. It's only a button on a white background, but a user interface nonetheless. You also added source code to **ViewController.swift**. These two files — the storyboard and the Swift file —

iOS App 1: Intro to Application Development

together form the design and implementation of a *view controller*. A lot of the work in building iOS apps is making view controllers. The job of a view controller, generally, is to manage a single screen in your app.

Take a simple cookbook app, for example. When you launch the cookbook app, its main screen lists the available recipes. Tapping a recipe opens a new screen that shows the recipe in detail with an appetizing photo and cooking instructions. Each of these screens is managed by a view controller.

The view controllers in a simple cookbook app

What these two screens do is very different. One is a list of several items; the other presents a detail view of a single item.

That's why you need two view controllers: One that knows how to deal with lists and another that can handle images and cooking instructions. One of the design principles of iOS is that each screen in your app gets its own view controller.

Currently, *Bull's Eye* has only one screen (the white one with the button) and thus only needs one view controller. That view controller is simply named "ViewController," and the storyboard and Swift file work together to implement it. (If you are curious, you can check the connection between the screen and the code for it by switching to the Identity inspector on the right sidebar of Xcode in the storyboard view. The Class value shows the current class associated with the storyboard scene.)

Simply put, the Main.storyboard file contains the design of the view controller's user interface, while ViewController.swift contains its functionality — the logic that makes the user interface work, written in the Swift language.

Because you used the Single View Application template, Xcode automatically created the view controller for you. Later, you will add a second screen to the game and you will create your own view controller for that.

iOS App 1: Intro to Application Development

Making connections

The two lines of source code you just added to ViewController.swift lets Interface Builder know that the controller has a "showAlert" action, which presumably will show an alert pop-up. You will now connect the button on the storyboard to that action in your source code.

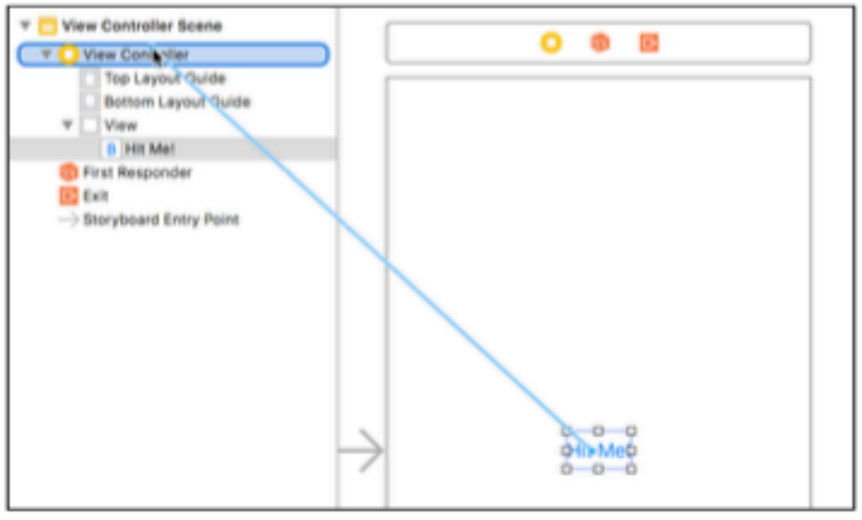
➤ Click **Main.storyboard** to go back into Interface Builder.

In Interface Builder, there should be a second pane on the left, next to the navigator area, called the **Document Outline**, that lists all the items in your storyboard. If you do not see that pane, click the small toggle button in the bottom-left corner of the Interface Builder canvas to reveal it.

The button that shows the Document Outline pane

➤ Click the **Hit Me button** once to select it.

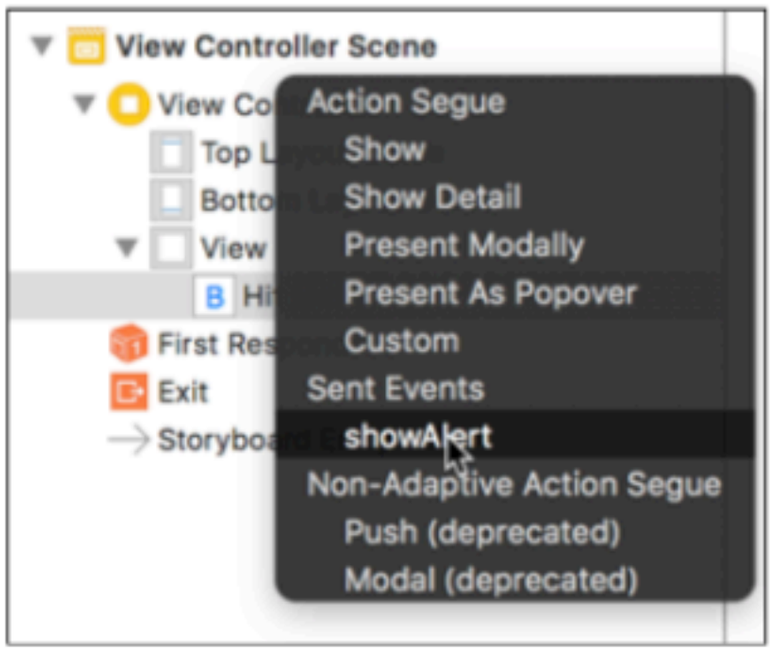
With the Hit Me button selected, hold down the **Control** key, click on the button and drag up to the **View Controller** item in the Document Outline. You should see a blue line going from the button up to View Controller. Please note that, instead of holding down Control, you can also right-click and drag, but don't let go of the mouse button before you start dragging.



Ctrl-drag from the button to View Controller

iOS App 1: Intro to Application Development

Once you're on View Controller, let go of the mouse button and a small menu will appear. It contains several sections: "Action Segue," "Sent Events," and "Non-Adaptive Action Segue," with one or more options below each. You're interested in the **showAlert** option under Sent Events. The Sent Events section shows all possible actions in your source code that can be hooked up to your storyboad — **showAlert** is the name of the action that you added earlier in the source code of **ViewController.swift**.



The pop-up menu with the showAlert action

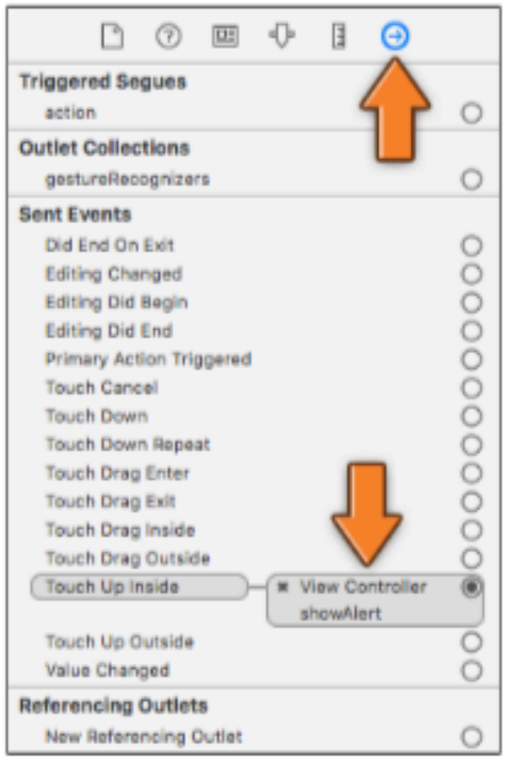
➤ Click on **showAlert** to select it. This instructs Interface Builder to make a connection between the button and the line `@IBAction func showAlert()`.

From now on, whenever the button is tapped the `showAlert` action will be performed. That is how you make buttons and other controls do things: You define an action in the view controller's Swift file and then you make the connection in Interface Builder.

You can see that the connection was made by going to the **Connections inspector** in the Utilities pane on the right side of the Xcode window. You should have the button selected when you do this.

iOS App 1: Intro to Application Development

➤ Click the small arrow-shaped button at the top of the pane to switch to the Connections inspector:



The inspector shows the connections from the button to any other objects

In the Sent Events section, the "Touch Up Inside" event is now connected to the showAlert action. You should also see the connection in the Swift file.

➤ Select **ViewController.swift** to edit it.

Notice how, to the left of the line with `@IBAction func showAlert()`, there is a solid circle? Click on that circle to reveal what this action is connected to.

A solid circle means the action is connected to something

iOS App 1: Intro to Application Development

Acting on the button

You now have a screen with a button. The button is hooked up to an action named `showAlert` that will be performed when the user taps the button. Currently, however, the action is empty and nothing will happen (try it out by running the app again, if you like). You need to give the app more instructions.

➤ In **ViewController.swift**, modify `showAlert` to look like the following:

```
@IBAction func showAlert() {
  let alert = UIAlertController(title: "Hello, World",
                                message: "This is my first app!",
                         preferredStyle: .alert)

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

  alert.addAction(action)

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

The new lines of code implement the actual alert display functionality. The commands between the `{ }` brackets of the action tell the iPhone what to do, and they are performed from top to bottom.

The code in `showAlert` creates an alert with a title "Hello, World," a message that states, "This is my first app!" and a single button labeled "Awesome." If you're not sure about the distinction between the title and the message: Both show text, but the title is slightly bigger and in a bold typeface.

➤ Click the **Run** button from Xcode's toolbar. If you didn't make any typos, your app should launch in iOS Simulator and you should see the alert box when you tap the button.

The alert pop-up in action

iOS App 1: Intro to Application Development

Congratulations, you've just written your first iOS app! What you just did may have seemed like gibberish to you, but that shouldn't matter. We'll take it one small step at a time.

You can strike off the first two items from the to-do list already: Putting a button on the screen and showing an alert when the user taps the button.

Take a little break, let it all sink in and come back when you're ready for more! You're only just getting started...

> **Note**: Just in case you get stuck, I have provided the complete Xcode projects, which are snapshots of the project as at the beginning and end of each chapter. That way, you can compare your version of the app to mine, or — if you really make a mess of things — continue from a version that is known to work.

You can find the project files for each chapter in the corresponding folder.

Problems?

If Xcode gives you a "Build Failed" error message after you press Run, then make sure you typed in everything correctly. Even the smallest mistake could potentially confuse Xcode.

It can be quite overwhelming at first to make sense of the error messages that Xcode spits out. A small typo at the top of a source file can produce several errors elsewhere in that file.

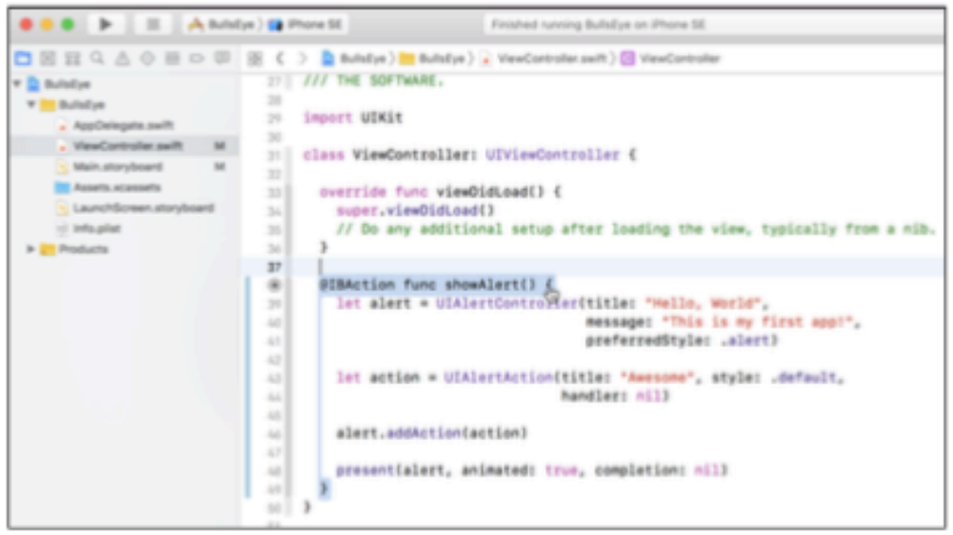
One common mistake is differences in capitalization. The Swift programming language is case-sensitive, which means it sees `Alert` and `alert` as two different names. Xcode complains about this with a "<something> undeclared" or "Use of unresolved identifier" error.

When Xcode says things like "Parse Issue" or "Expected <something>" then you probably forgot a curly bracket } or parenthesis) somewhere. Not matching up opening and closing brackets is a common error.

Tip: In Xcode, there are multiple ways to find matching brackets to see if they line up. If you move the editing cursor past a closing bracket, Xcode will highlight the corresponding opening bracket, or vice versa. You could also hold down the ⌘ key and move your mouse cursor over a line with a curly bracket and Xcode will highlight the full block from the opening curly bracket to the closing curly bracket (or vice versa) — nifty!

iOS App 1: Intro to Application Development

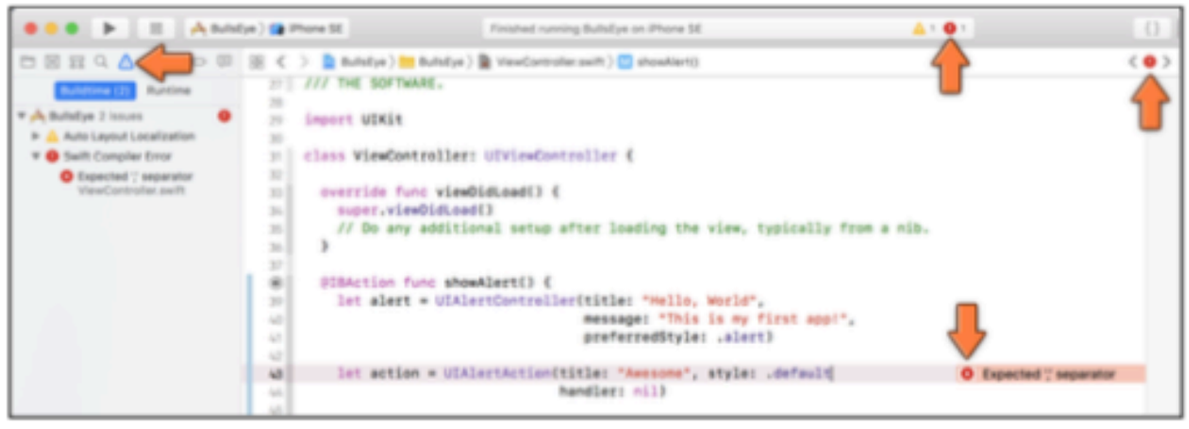
You can see the block, here:



Xcode shows you the complete block for curly brackets

Tiny details are very important when you're programming. Even one single misplaced character can prevent the Swift compiler from building your app.

Fortunately, such mistakes are easy to find.

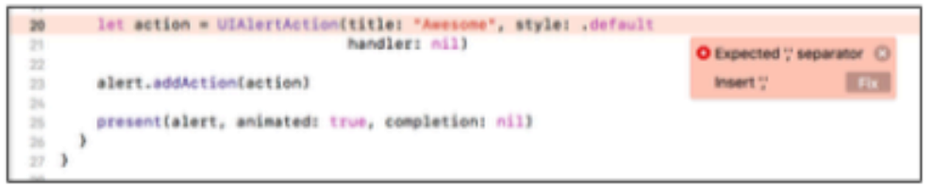


Xcode makes sure you can't miss errors

When Xcode detects an error, it switches the pane on the left from the Project navigator, to the **Issue navigator**, which shows all the errors and warnings that Xcode has found. (You can go back to the project navigator using the small icons along the top.)

In the above screenshot, apparently, I forgot a comma.

Click on the error message in the Issue navigator and Xcode takes you to the line in the source code with the error. Sometimes, depending on the error, it even suggests a fix:



Fix-it suggests a solution to the problem

It could be a bit of a puzzle to figure out what exactly you did wrong when your build fails — fortunately, Xcode lends a helping hand.

Errors and warnings

Xcode makes a distinction between errors (red) and warnings (yellow). Errors are fatal. If you get one, you cannot run the app until the error is fixed. Warnings are informative. Xcode just says, "You probably didn't mean to do this, but go ahead anyway."

In the previous screenshot showing all the error locations via arrows, you'll notice that there is a warning (a yellow triangle) in the Issue navigator. We'll discuss this particular warning and how to fix it later on.

Generally though, it is best to treat all warnings as if they were errors. Fix the warning before you continue and only run your app when there are zero errors and zero warnings. That doesn't guarantee the app won't have any bugs, but at least they won't be silly ones!

The anatomy of an app

It might be good at this point to get some sense of what goes on behind the scenes of an app.

An app is essentially made up of **objects** that can send messages to each other. Many of the objects in your app are provided by iOS; for example, the button is a `UIButton` object and the alert pop-up is a `UIAlertController` object. Some objects you will have to program yourself, such as the view controller.

These objects communicate by passing messages to each other. For example, when the user taps the Hit Me button in the app, that `UIButton` object sends a message to your view controller. In turn, the view controller may message more objects.

iOS App 1: Intro to Application Development

On iOS, apps are *event-driven*, which means that the objects listen for certain events to occur and then process them.

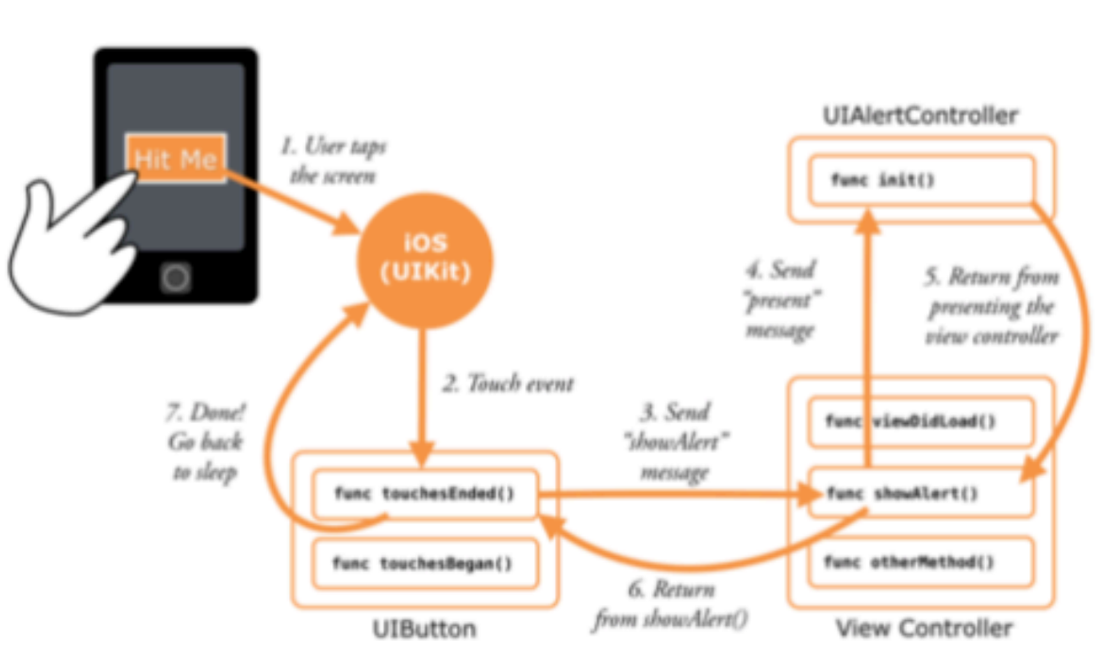
As strange as it may sound, an app spends most of its time doing... absolutely nothing. It just sits there waiting for something to happen. When the user taps the screen, the app springs to action for a few milliseconds, and then it goes back to sleep again until the next event arrives.

Your part in this scheme is that you write the source code for the actions that will be performed when your objects receive the messages for such events.

In the app, the button's Touch Up Inside event is connected to the view controller's `showAlert` action. So when the button recognizes it has been tapped, it sends the `showAlert` message to your view controller.

Inside `showAlert`, the view controller sends another message, `addAction`, to the `UIAlertController` object. And to show the alert, the view controller sends the `present` message.

Your whole app will be made up of objects that communicate in this fashion.



The general flow of events in an app

Maybe you have used PHP or Ruby scripts on your web site. This event-based model is different from how a PHP script works. The PHP script will run from top-to-bottom, executing the statements one-by-one until it reaches the end and then it exits.

iOS App 1: Intro to Application Development

Apps, on the other hand, don't exit until the user terminates them (or they crash!). They spend most of their time waiting for input events, then handle those events and go back to sleep.

Input from the user, mostly in the form of touches and taps, is the most important source of events for your app, but there are other types of events as well. For example, the operating system will notify your app when the user receives an incoming phone call, when it has to redraw the screen, when a timer has counted down, etc.

Everything your app does is triggered by some event.

iOS App 1: Intro to Application Development

Now that you've 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.

iOS App 1: Intro to Application Development

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.
- **Adding the other controls**: Add the rest of the controls necessary to complete the user interface of your app.

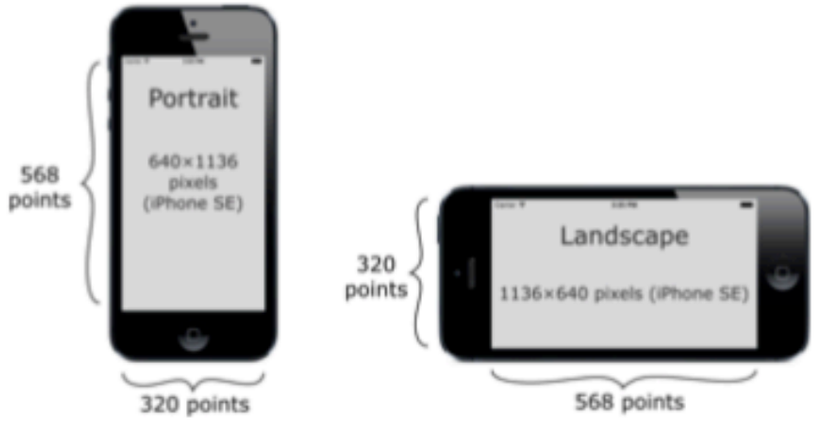
Portrait vs. landscape

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

You've no doubt seen landscape apps before on 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 in landscape 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

iOS App 1: Intro to Application Development

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 television 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, Xs	5.8"	375 x 812
iPhone Xr	6.1"	414 x 896
iPhone Xs Max	6.5"	414 x 896
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

iOS App 1: Intro to Application Development

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 other iPhone screens.

Converting 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.

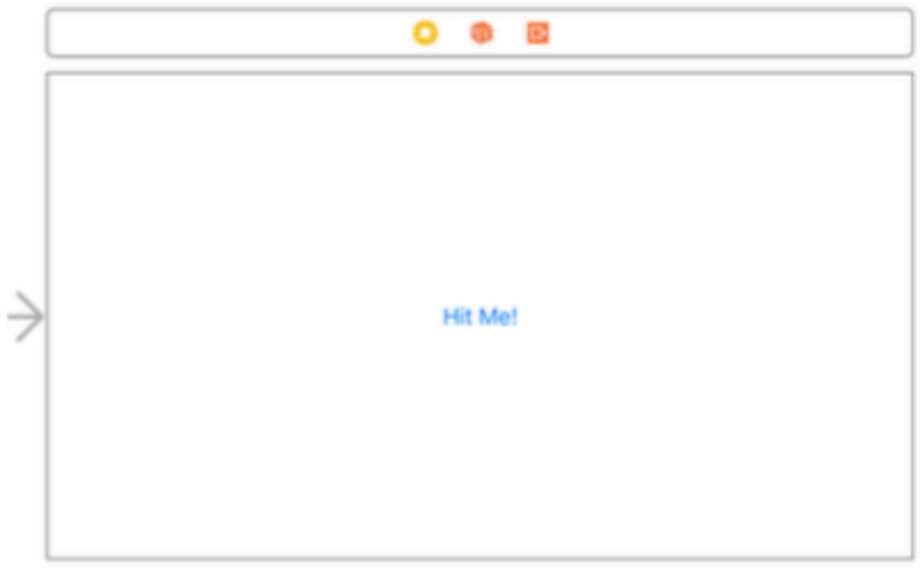
2. Change the **Supported Device Orientations** setting of the app.

➤ 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 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 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.

iOS App 1: Intro to Application Development

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:

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.

iOS App 1: Intro to Application Development

Understanding objects, data and methods

Time for some programming theory. No, you can't 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 pop-up. 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 it 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

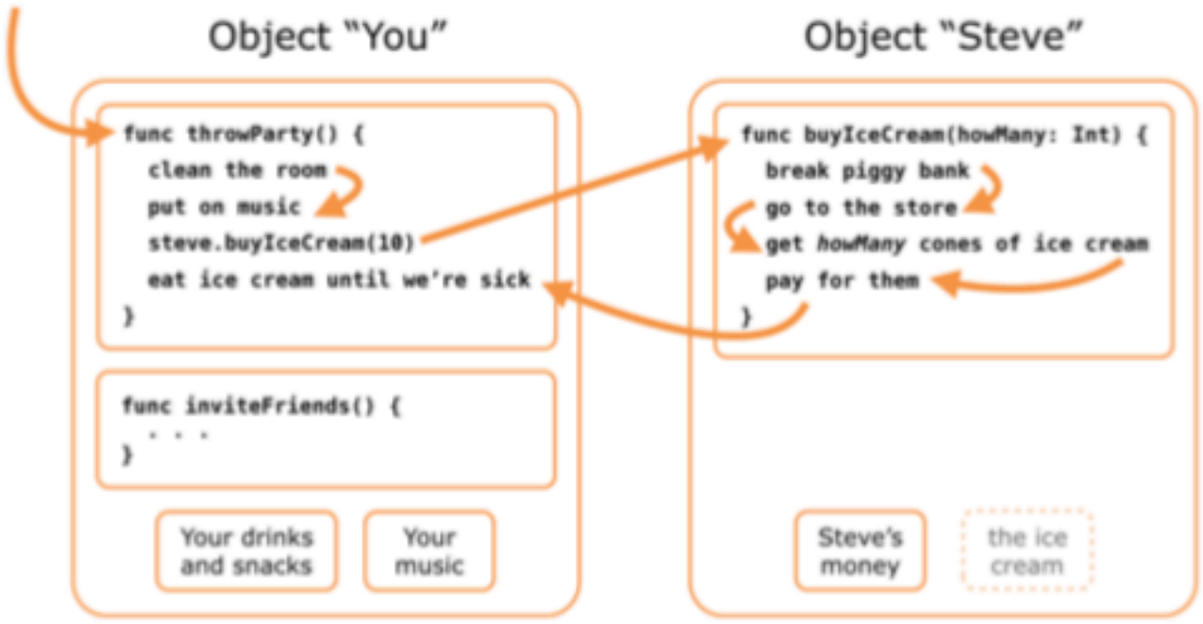
If you look through the rest of **ViewController.swift**, you'll see another method, `viewDidLoad()`.

It currently doesn't do much; the Xcode template placed it there for your convenience. It's a method that's often used by view controllers, so it's likely that you will need to add some code to it at some point.

> **Note**: These additional methods added by an Xcode template are known as "boilerplate code." If you don't need to add functionality to these boilerplate methods, feel free to remove them — it'll make your code cleaner and more compact.
>
> There's a caveat though; sometimes, the boilerplate code is needed in order not to get a compiler error. You will see this later on when we start using more complex view controllers. So if you remove the boilerplate code and get a compiler error, restore the code and try removing the code selectively until you figure out what is needed and what is not.

iOS App 1: Intro to Application Development

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



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).

iOS App 1: Intro to Application Development

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.

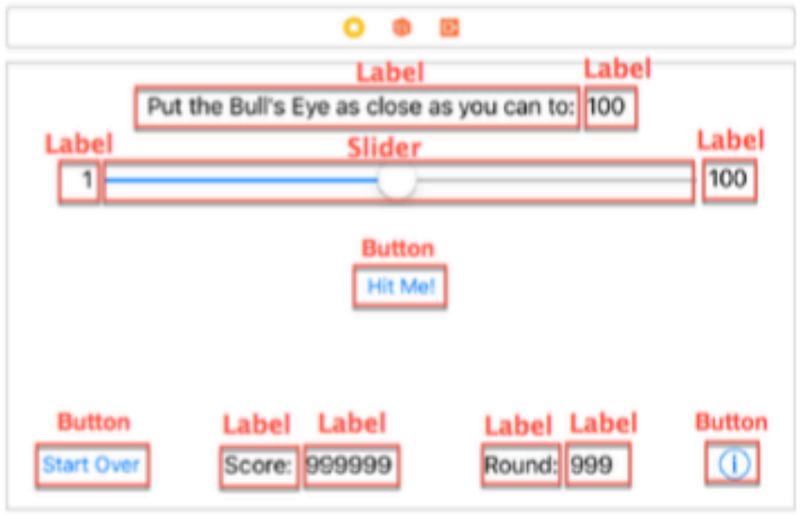
Often the terms "calling a method" or "invoking a method" are used instead. That means the exact same thing as sending a message: The computer jumps to the method you're calling and returns to where it left off when that method is done.

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.

Adding 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:



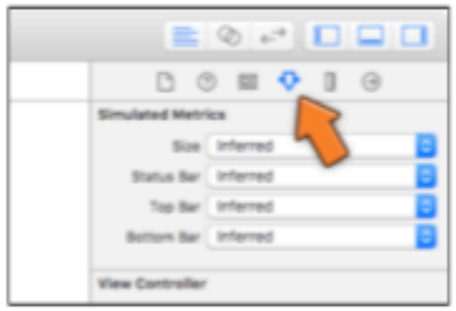
The different views in the game screen

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 onto 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.

> **Note**: It might seem a little annoying to use the Library panel since it goes away as soon as you drag an item from it. You then have to tap the icon on the toolbar to show the Library panel again to select another item. If you are placing multiple components, just hold down the Alt/Option key (⌥) as you drag an item from the Library panel — the Library panel will remain open, allowing you to select another item.

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:

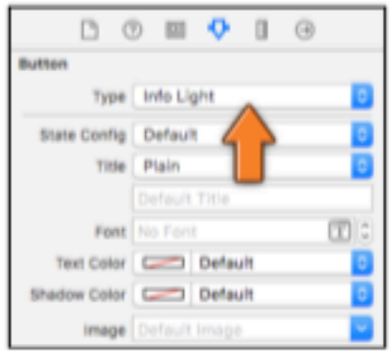


The Attributes inspector

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.

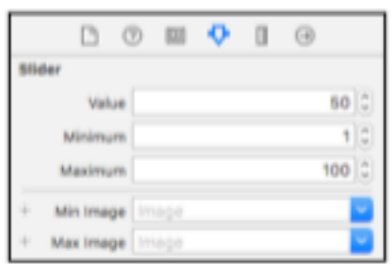
iOS App 1: Intro to Application Development

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



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.



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 much programming! That is going to change really soon, because you will have to write Swift code to actually make the controls do anything.

iOS App 1: Intro to Application Development

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 pop-up. (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.

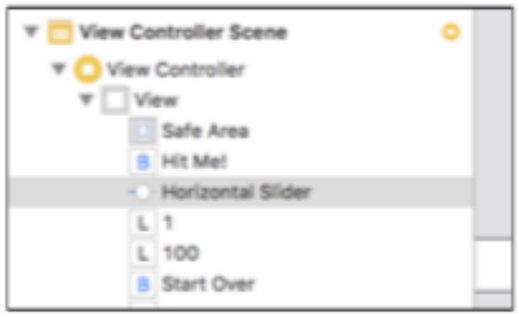
The steps for adding this action are largely the same as before.

➤ 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 pop-up. 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

iOS App 1: Intro to Application Development

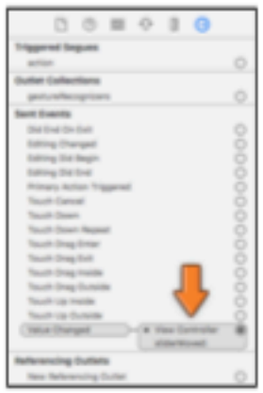
Remember, if the Document Outline is not visible, click the little icon at the bottom of the Xcode window to reveal it:

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, changing its value.

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.

iOS App 1: Intro to Application Development

➤ Run the app and drag the slider.

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

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 iOS Simulator. You can safely ignore these messages.

iOS App 1: Intro to Application Development

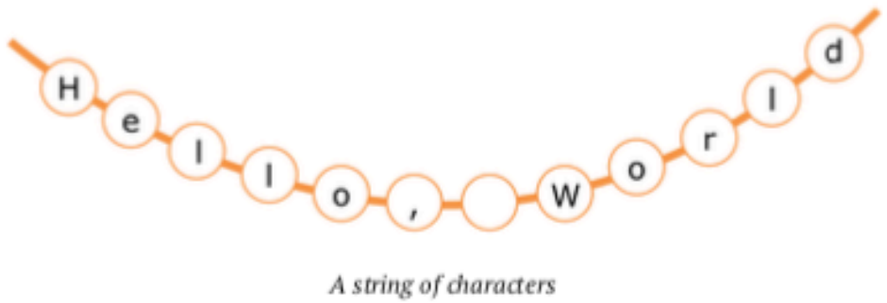
Strings

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 was 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 and is known as *string interpolation*.

Variables

Printing information with `print()` to the Console is very useful during the development process, but it's absolutely useless to users 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 pop-up. 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() {
    . . .
  }

  @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!

Creating your variable

➤ 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 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 pop-up. That is why you tell the `UIAlertController` that it should now use this new string as the message to display.

iOS App 1: Intro to Application Development

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

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 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.

FEEDBACK TIME

http://bit.ly/iOSFeedback

THANK YOU