We ❤ Swift

iOS Spellbook

fundamental spells and charms

Made with ❤ by Andrei Puni

Table of Contents

iOS Spellbook

The book is still under development. This is just part 1 of 3. The whole book has over 600 pages. More comming soon!

Thank you for your support and patience!

Version 0.1

Dedication

I dedicate this book to my amazing son Amon, and my lovely wife Zsu. Thank you for bringing so much joy in my life!

Welcome

This is not a lecture. This is the iOS Spellbook.

Our mutual goal is for you to become a great iOS developer. Keep in mind: always be attentive to each word in this book because we did too by resuming much information into few pages; you need to be well rested and 100% focused in every learning session; don't pause more than a day or two between learning sessions, this will help you spend less time remembering things you learned and spend more on learning and discovering new things.

That being said, we are looking forward to seeing your first iOS Applications.

Have fun!

About the book

The purpose of this book is to help you do 20% of the work you need to do in order to get 80% of the way to Expert iOS Developer. This book will not teach you all you need to know about iOS because no book can do that. Learning to be a great iOS developer takes years of practice and a lot of hard work.

When learning something new, almost everything seems impossible to understand and by the moment you feel like you finally got it, you realise there is much more to learn. Like my previous book, Swift Programming from Scratch, this book was designed using a principle named Cascading Information Theory, which is used in games. It gives you the minimum possible snippets of information to gain the appropriate level of understanding at each point. The book gradually shows new concepts giving you time to practice every one of them and get comfortable using them.

The assumption is that you know how to code in swift and that you understand basic Object Oriented Programming concepts. If you don't, then you can learn programming in Swift from Swift Programming from Scratch(book) and Object Oriented Programming in Swift(article).

Alongside the theory, you will find various exercises and projects. Solving them will help you solidify what you learn. You will also learn how to use different tools and workflows

The book is separated into 3 parts:

1) **The Fundamentals**

This part teaches you the first body of knowledge you need to start your iOS developer journey. With this knowledge you can make simple apps and learn to use different parts that make up more complex apps - like the one you want to build :)

2) **Exploring iOS**

In this part we are going to learn how to use different SDKs like the camera, accelerometer, notifications and so on. We are also going to study more complex UI components like table views and collection views.

3) **Workflow and Advanced Topics**

In this part we'll focus more on the other side of iOS development. How do you store your code, deploy or test your app. What tools do you use? How do you use open source? How do you give people your app and how to submit it on the app store

After reading the first part in order, you can pretty much take any path you desire until you finish the book. Some chapters require that others must have been previously read, in that case they will be indicated on the first page of that chapter.

First Steps

So you know how to code and you want to learn how to make iOS apps. The first thing you need to learn is how to use the tools of the trade:

Xcode is by far the most important tool for creating iOS apps. It's the first and only tool you need to learn to start making basic apps. In the first part of the book we are going to learn how to use Xcode to create apps and understand what an app is and how it works by studing it basic components.

Instead of listing every feature of Xcode has I'm going to take you step by step and make a simple iOS app. The goal of this chapter is to give you a taste of iOS development and to showcase the main features of Xcode.

What will I learn?

- How easy it is to make a simple app
- Xcode menus and when to use them
- Interface Builder basics
- How to connect code with interface
- How to use the iOS Simulator

Tap. Tap. Boom!

The simplest app I could think of is a counter app. And this is the least boring(but still simple) version of it.

The app has two interface components: a label and a button. The label displays the number of times the button has been pushed. The logic has a small twist, it shows an emoji every 3 taps. Hence the name "Tap. Tap. Boom!".

Step 0: Install Xcode

If you don't already have Xcode installed you can download it from this link - grab a coffee, it's going to take a while.

Step 1: Create an empty project

Open `Xcode` and select `Create a new Xcode project`. Choose the `Single View Application` template, set the language to `Swift` if needed, and name it `TapTapBoom`. Make sure you have simulator selected in the menu next to the play button.

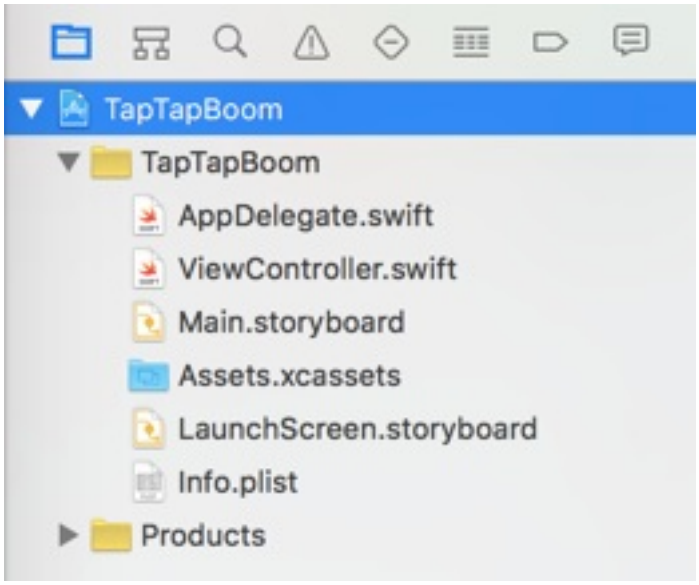
That's it! Let's see how the app looks at this point. Run the project by pressing on the play icon in the top left corner of Xcode.

The iOS simulator should start booting up. After it finishes you should see an empty white screen.

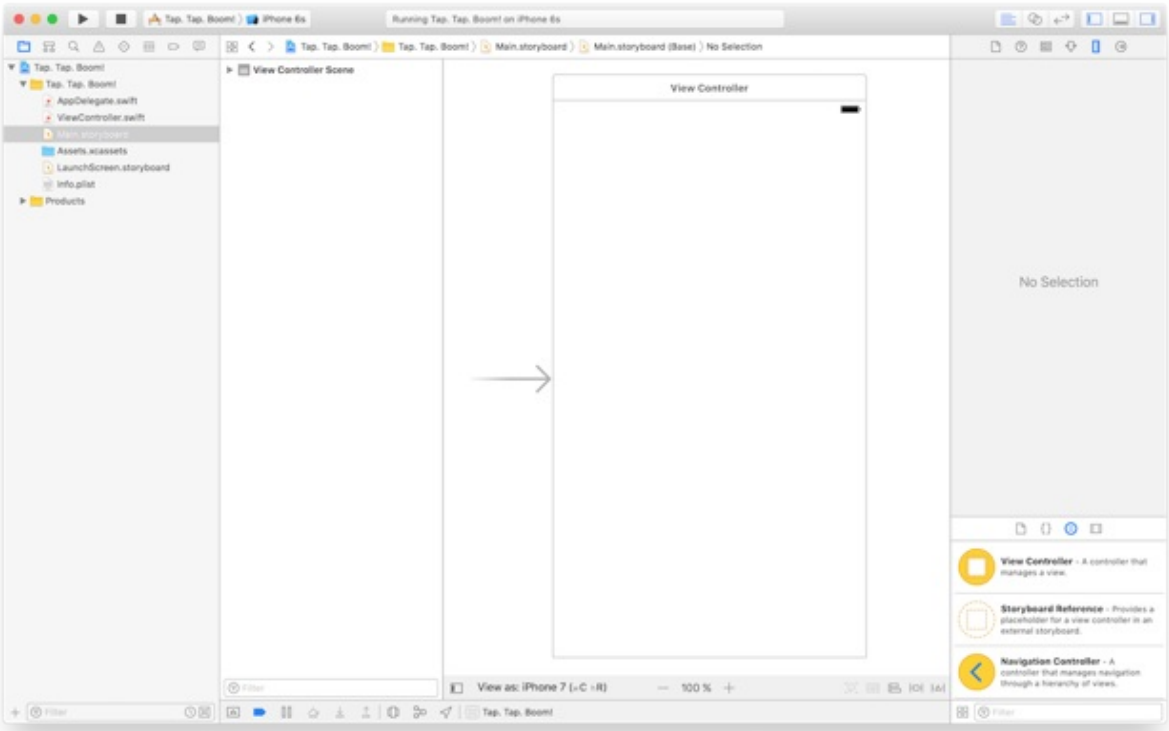
> 
>
> **Quick tip**
>
> In some cases the height of the simulator is bigger than your screen. You can resise the simulator by using the following shortcuts:
>
> - `Command` + `1` (`⌘` + `1`) for 100%
> - `Command` + `2` (`⌘` + `2`) for 75%
> - `Command` + `3` (`⌘` + `3`) for 50%
> - `Command` + `4` (`⌘` + `4`) for 33%
> - `Command` + `5` (`⌘` + `5`) for 25%

Step 2: Open Interface Builder

On the left side of Xcode you should see the project files listed. This part is called the `Navigator`.



In the navigator find and click on `Main.storyboard`. After that the middle part of Xcode should be replaced with `Interface Builder` showing an empty screen.

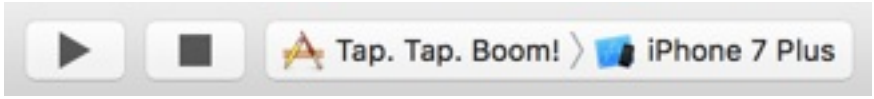


`Interface Builder` is the tool used to create interfaces. Starting with iOS 8 it's possible to make components that render in Interface Builder. So even if you decide to use code to make a part of your interface, you can still use it in `Interface Builder`.

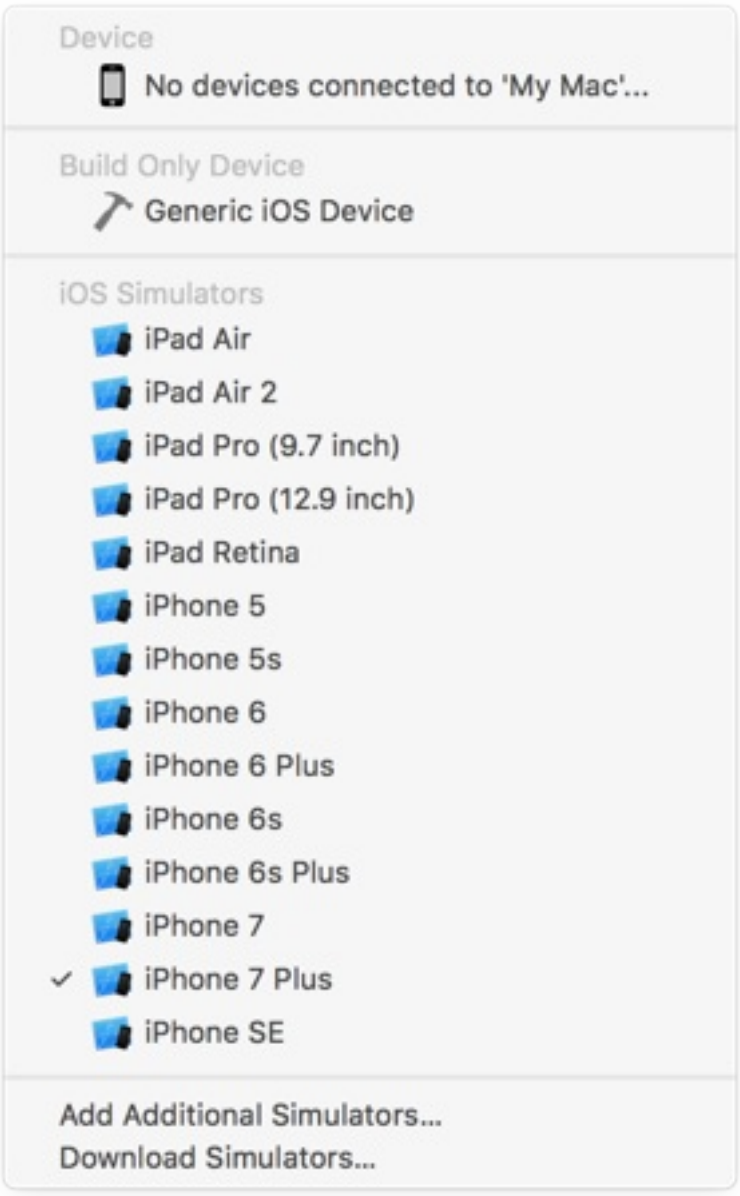
> 
>
> **Quick tip**
>
> You you need a custom control that Apple does not provide before starting to implement it search github and CococaControls for it.

Select `ViewController` from the side menu:

The default simulator is the iPhone 7 Plus one. You can change the simulated device from the menu on the top left side.



After clicking on `iPhone 7 Plus` icon, you will be shown a menu. If you have any devices connected it will show them in this list. That takes a few steps to setup, so for now let's stick with the simulator.



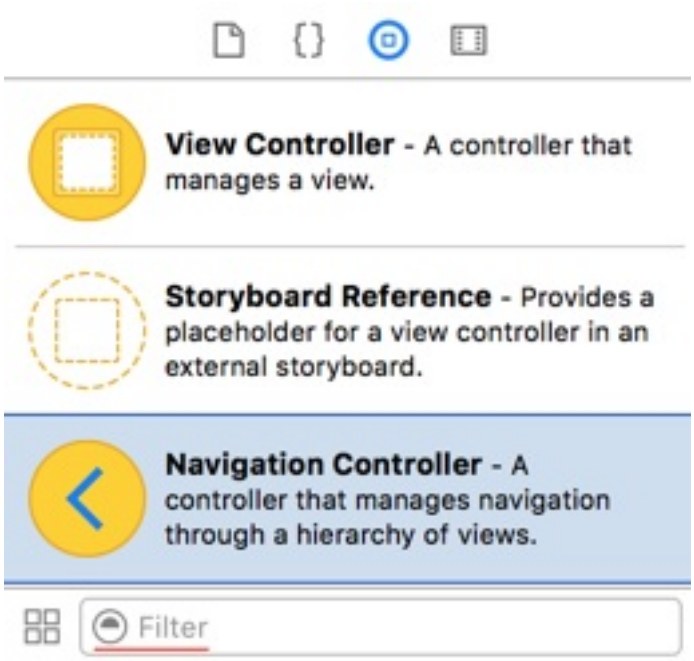
Quick tip

When developing app make a habit out of periodically switching between different simulators to test your app on all screen sizes as you develop it. Bugs take more resource to fix the more time passes since they where introduced in your code. Fixing them as soon as they are added will save you a lot of time.

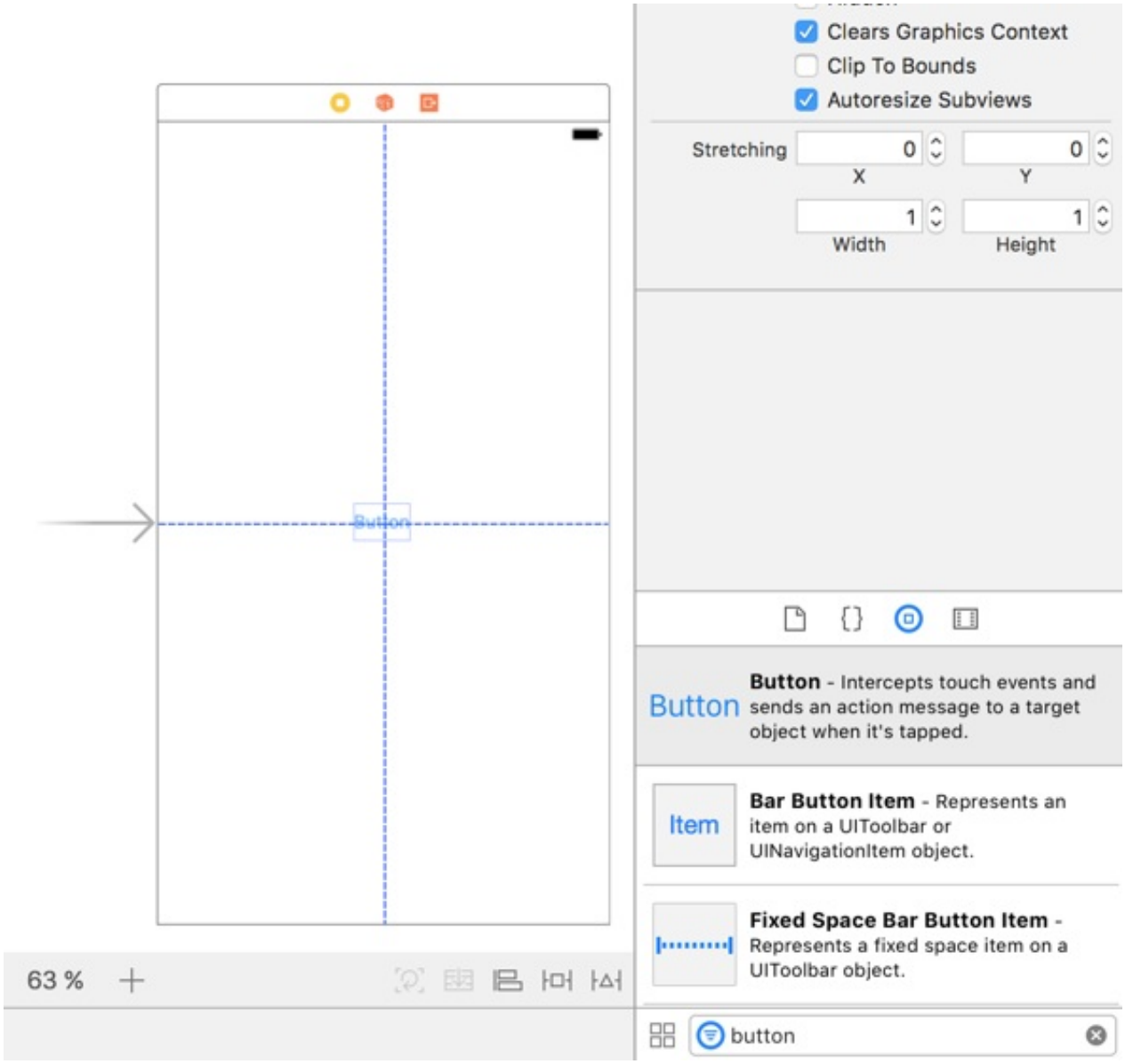
Step 3: Add a button

Let's add the first component. Find the `Component Library` in the bottom right corner of Xcode.

The default is the `Library` so you might have to change the tab by pressing  button:

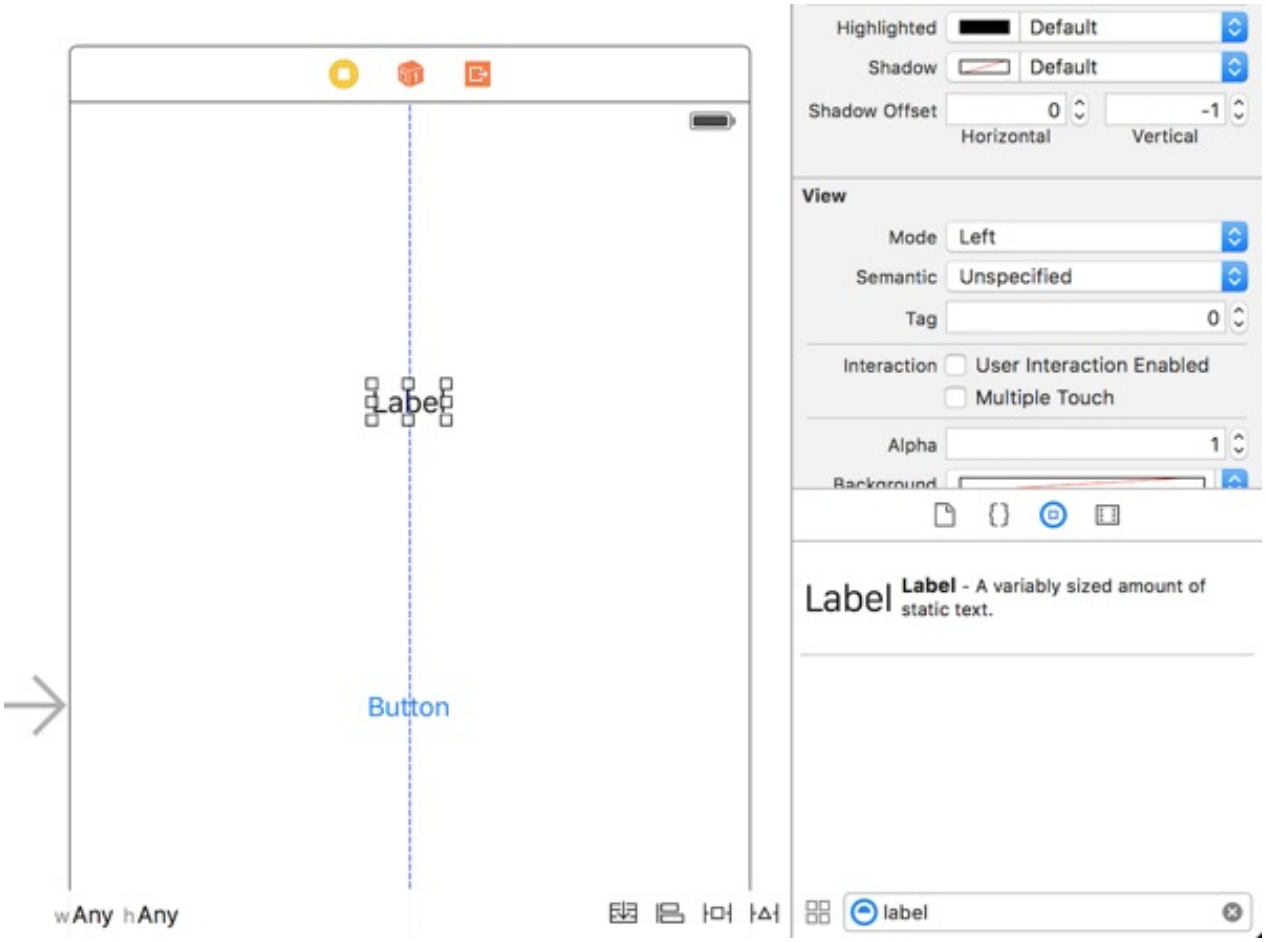


Using the filter function you can search for button. Drag and drop the button - change the title to `Tap Me!` by double clicking on it.



Step 4: Add a label

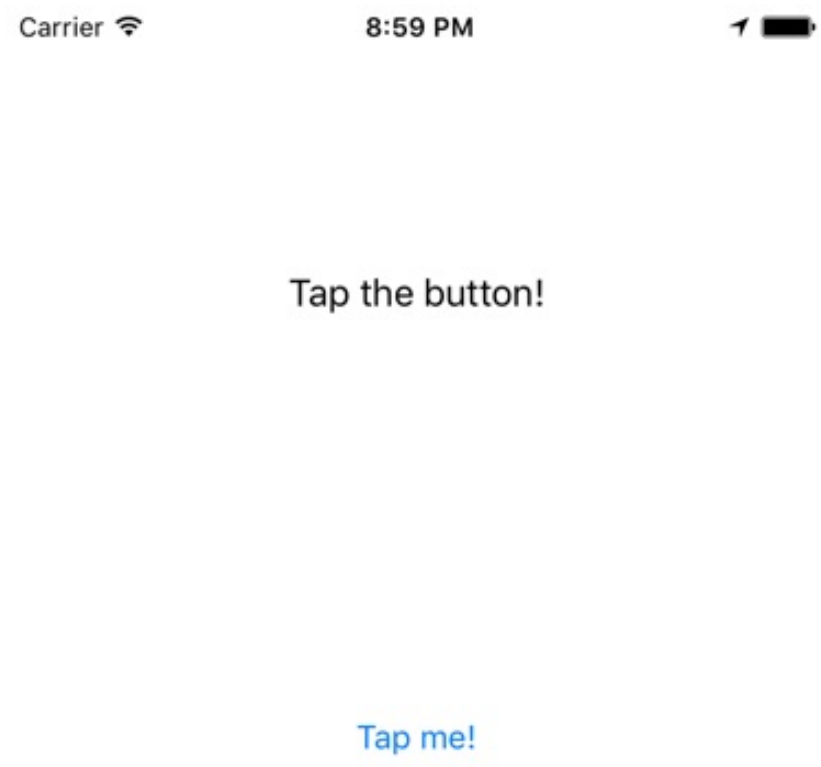
Now search for `label` in the `Component Library`. Drag and drop a label on the screen.



Change the text to `Tap the button!`.

Step 5: Test

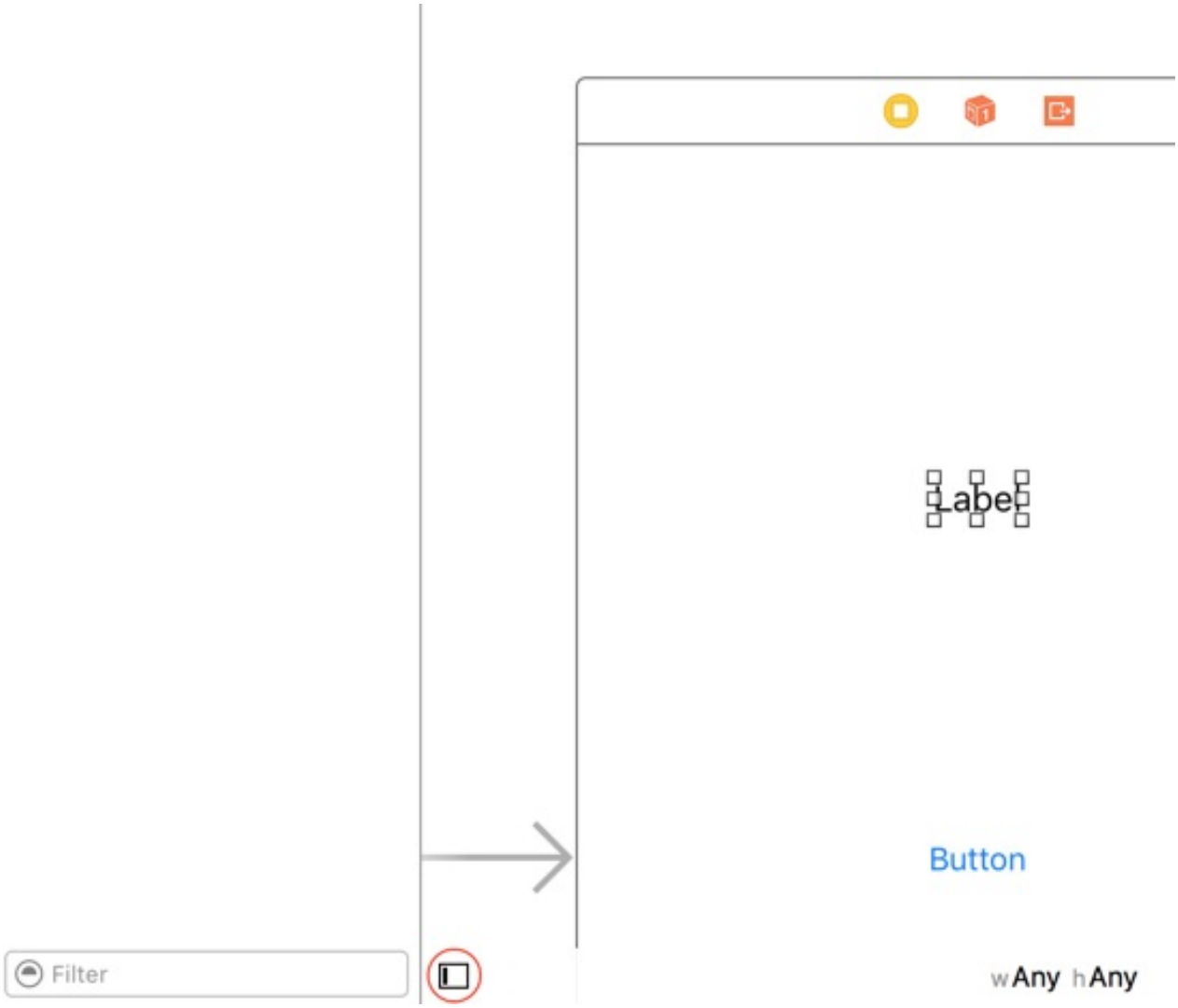
Run the app to see if the components are showing correctly, this time do it by pressing `Command` + `R` (`⌘` + `R`).



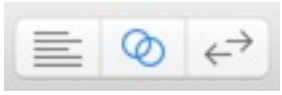
Step 6: Make the button do something

Time to connect interface with code. Literally.

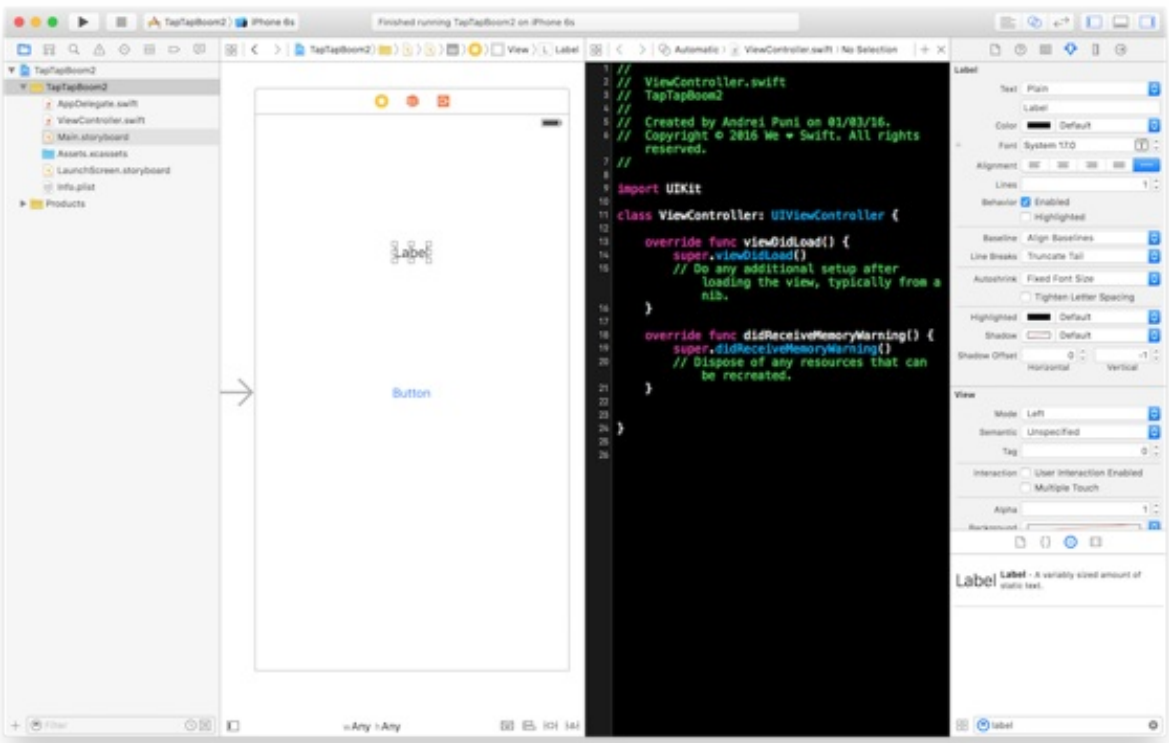
First thing make some space. Hide the `Document Outline` by pressing the button in the lower left corner of Xcode.



Open the assistant editor by pressing the *"two circles"* button

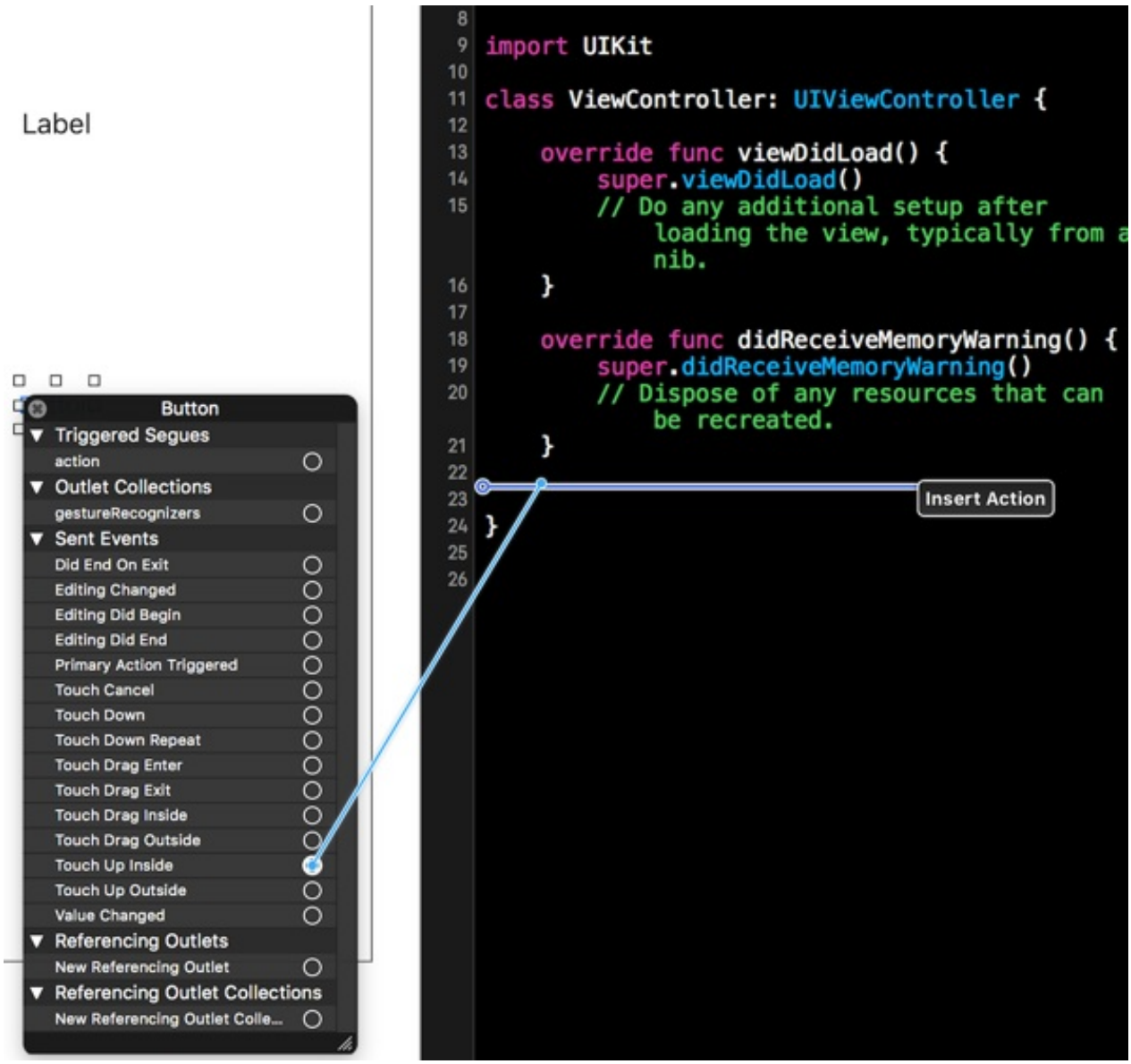


Now you should have the interface on the left side and the code on the right side.

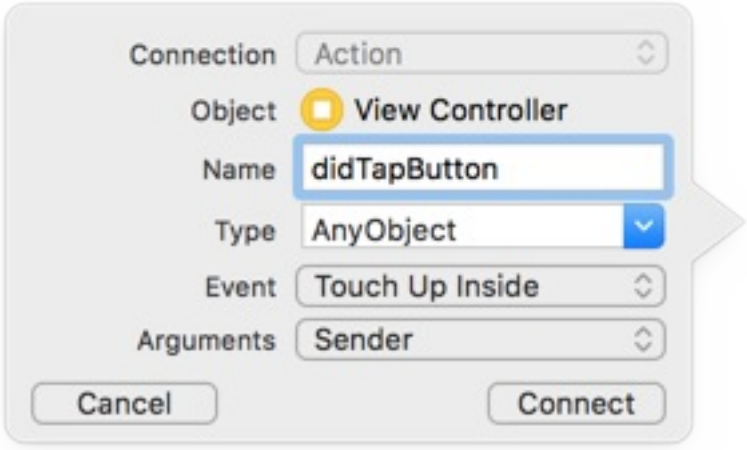


On the right side you should see the `ViewController` class. A view controller is an object that manages a view. The usual case is one screen has one view controller. More on this in the View Controller chapter. For now, just remember that view controllers have the code for a screen.

Right click on the button and drag (keep holding) to connect the `touchUpInside:` event to a line inside `ViewController` class.



Name the action `didTapButton`.



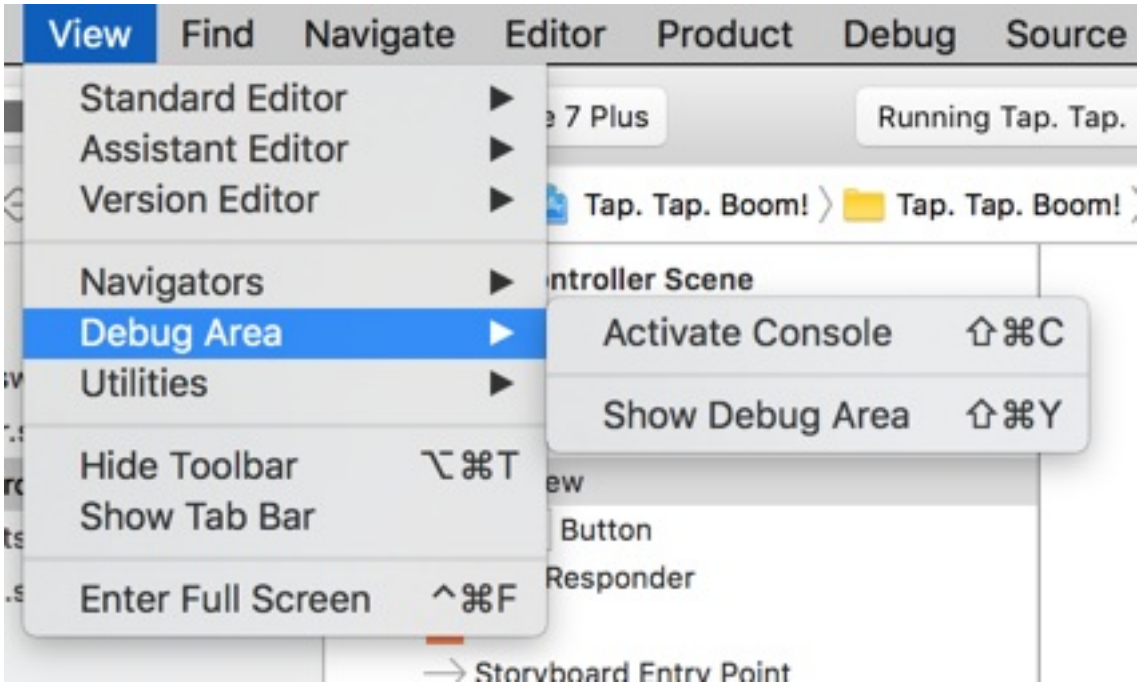
This will create a new method named `didTapButton(_:)` that will be called when the button is tapped. The parameter sent to it is the button that was pressed. So you can have multiple buttons send the same action - for example in a calculator app all the digits will be connected with `didTapDigit(_:)`.

Add a print command to the method to make it do something:

```
class ViewController: UIViewController {
    ...

    @IBAction func didTapButton(_ sender: AnyObject) {
        print("did tap button")
    }
}
```

Run the app to test your code. After tapping the button, a message should be displayed in the console from the `Debug Area`. The console will open by itself in the moment it has something to show.

You can show/hide the `Debug Area` from the menu `View > Debug Area > Show Debug Area` or by using the shortcut `Command` + `Shift` + `Y` (`⌘` + `⇧` + `Y`).



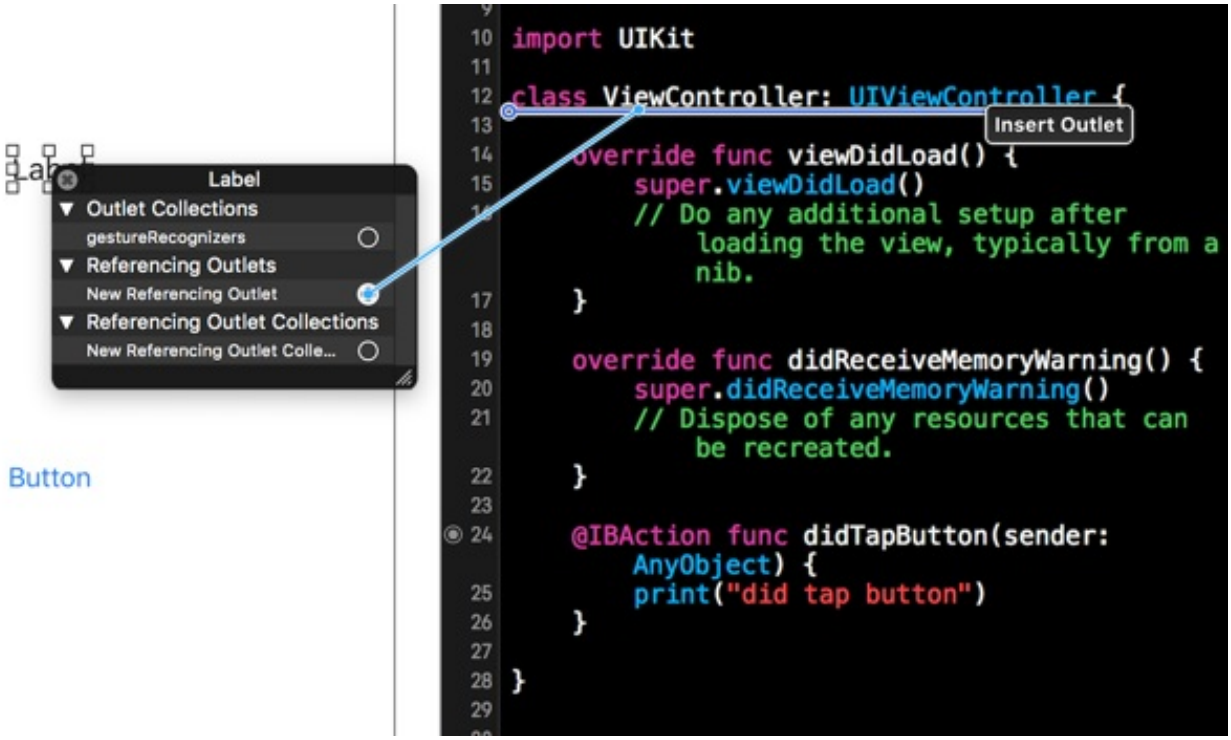
Step 7: Connect the label

Ok... consoles are super fun, now let's update something on the screen.

To change the text inside the label we need to get a reference to it. We can do this by creating a Referencing Outlet.

Almost same process we had with the button action. Right click on the label to display a menu:

Connect the New Referencing Outlet option with the top lines of the view controller:



Name it `label`. This will create a new property:

```
class ViewController: UIViewController {
    @IBOutlet weak var label: UILabel!

    ...
}
```

Step 8: Add the counter

Inside the `ViewController` add a property named `counter`.

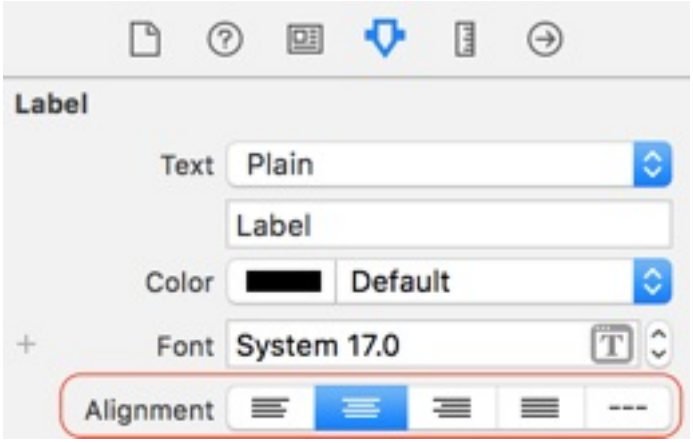
```
class ViewController: UIViewController {
    @IBOutlet weak var label: UILabel!

    var counter = 0

    ...
}
```

Step 9: Update and display the counter

```
class ViewController: UIViewController {

    ...

    @IBAction func didTapButton(_ sender: AnyObject) {
        counter += 1

        label.text = "\(counter) taps"
    }
}
```

If we run the code now we notice two problems:
1) the label is not centred
2) it says 1 taps instead of 1 tap

To fix the first problem for in `Interface Builder` and select the label. In the `Attributes Inspector` you set the text alignment property to centred.



To fix the second one create a `unit` string constant before setting the text. `unit` will be `"tap"` if the counter is `1` and `"taps"` otherwise.

```
class ViewController: UIViewController {

    ...

    @IBAction func didTapButton(_ sender: AnyObject) {
        counter += 1

        let unit: String
```

```
        if counter == 1 {
            unit = "tap"
        } else {
            unit = "taps"
        }

        label.text = "\(counter) \(unit)"
    }
}
```

Step 10: Boom!

And now for the *explosive* effect: if the counter is divisible by 3 then replace the text with `Boom!`.

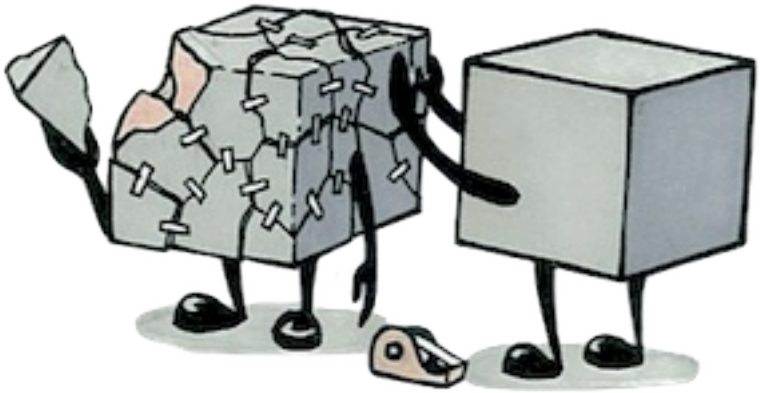
```
class ViewController: UIViewController {

    ...

    @IBAction func didTapButton(sender: AnyObject) {
        counter += 1

        let unit: String

        if counter == 1 {
            unit = "tap"
        } else {
            unit = "taps"
        }

        if counter % 3 == 0 {
            label.text = "Boom!"
        } else {
            label.text = "\(counter) \(unit)"
        }
    }
}
```

Tap, Tap, Boom! Tap, Tap, Boom! Tap, Tap, Boom! ...

Conclusion

Xcode is one of the most loved and intelligent programming environment. It's really extremely easy to use it, and it's the only tool you need to know for creating beautiful and powerful Apps! It helps you do everything from writing code to creating the interface and running the app in simulator or on your device.

App Anatomy



So you kinda know what's an app. It's that round cornered square on your phone. But what makes it tick?

In this chapter we are going to go over a few low level components that make an app. You don't have to understand every detail mentioned here so don't panic if you can't follow some of them. The purpose of this chapter is to show you what ingredients are used to make an app, how does the process of transforming them in to an app looks like and what technically what an app really is.

What's an app?

A computer system (or phone) runs all kinds of code. Three kinds to be precise:

1) **Kernel** - this is the code that gives a programable interface for the hardware.

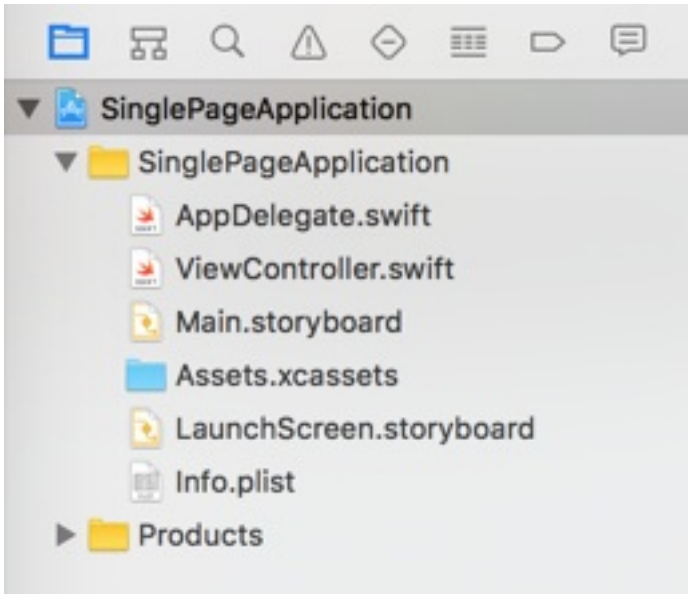
2) **Operating System** - in the case of the iPhone this is iOS, the OS is responsible for managing the systems resources and for running programs. They also abstract the hardware part of a computer system. That means you can make a program that uses a part of the system, in a general way - for example a game requires a system with a graphics card - the same code should work without any change on new hardware, because the system will give the same interface.

3) **Applications** - all programs that run on top of the operating system. Applications have access to the system's hardware trough the OS. In the case of iOS, this brings a couple of limitations - we call this set of limitations the app sandbox.

What makes an app?

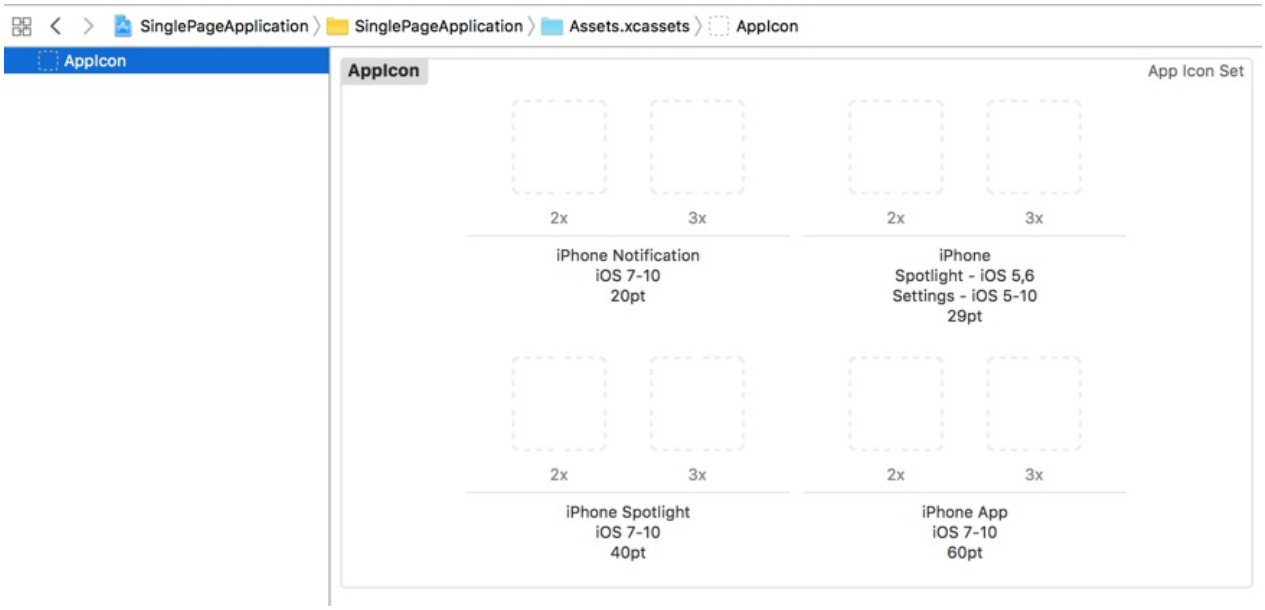
One of the things that helped me understand iOS and apps was to study the files that get generated when you create a new project.

Let's take a closer look at the `Single Page Application` template:



It generated 2 swift code files: AppDelegate and ViewController; 2 storyboard files; `Assets.xcassets` (Xcode Assets) and an `Info.plist` file.

The `Assets` file holds your app's resources. For example the app icon.



The `Info.plist` file is used to configure the app. It holds information about the app's behaviour that can be interpreted by iOS without running the app. For example the version of the app is store in this file.

Key	Type	Value
▼ Information Property List	Dictionary	(13 items)
Localization native development re...	String	en
Executable file	String	$(EXECUTABLE_NAME)
Bundle identifier	String	$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle version	String	1
Application requires iPhone enviro...	Boolean	YES
Launch screen interface file base...	String	LaunchScreen
Main storyboard file base name	String	Main
▼ Required device capabilities	Array	(1 item)
Item 0	String	armv7
▼ Supported interface orientations	Array	(3 items)
Item 0	String	Portrait (bottom home button)
Item 1	String	Landscape (left home button)
Item 2	String	Landscape (right home button)

Almost every time you want to use new feature in your app, you will have to update this file. For example, if you use the camera, you will have to add a message in `Info.plist`. That message will be displayed by the system when asking for permission to use the camera.

`LaunchScreen.storyboard` is a special interface file which is loaded and shown on the screen until the main interface file loads.

`Main.storyboard` is the main interface file. It defines the first screens the app will load. The first that gets shown is called `Initial View Controller` and it is marked with an arrow starts from nothing and points to it.

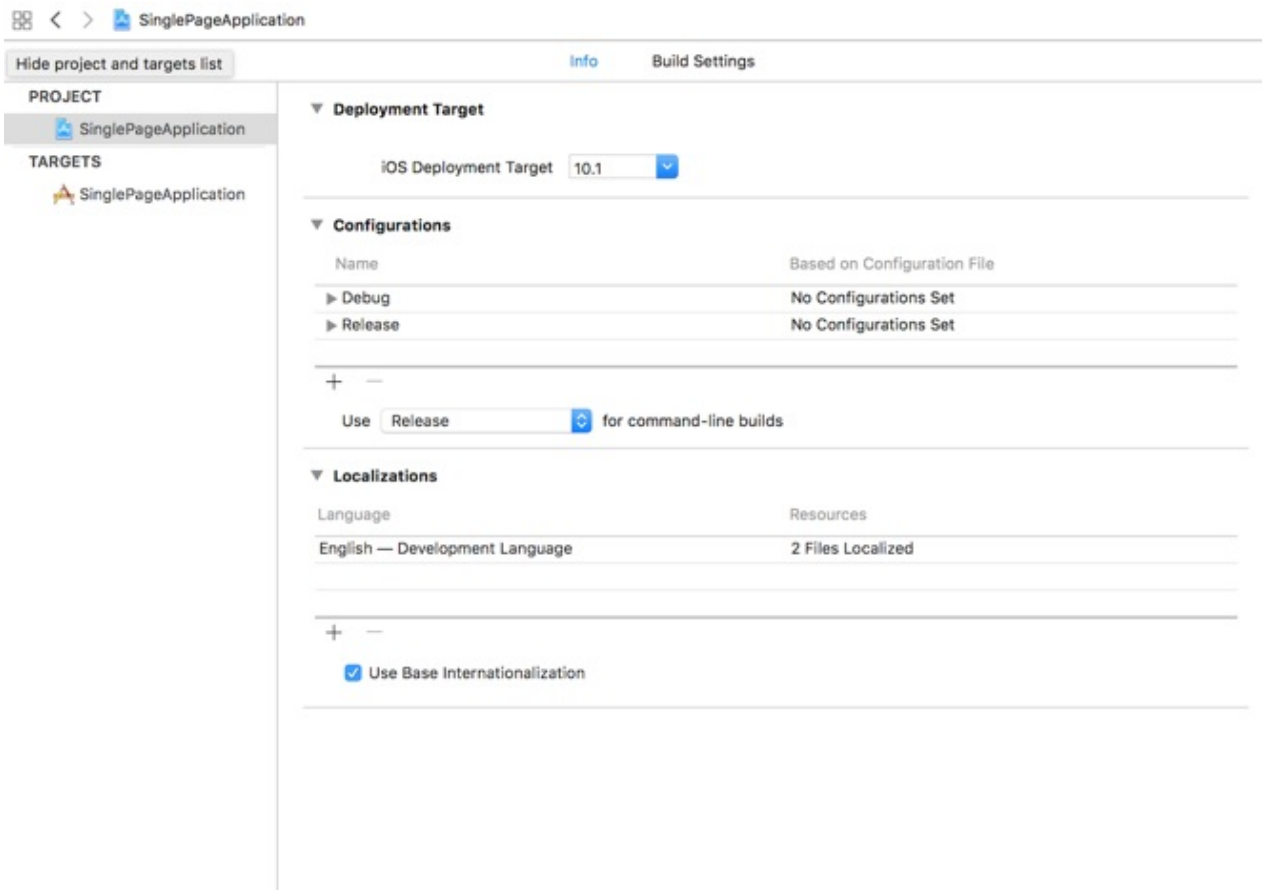
`ViewController.swift` is the swift file that describes the class on the main

view controller. By default it implements two methods: `viewDidLoad` and `didReceiveMemoryWarning`. The `viewDidLoad` method can be to run code right after the View loads, such as loading additional interface components after the interface from the storyboard loads or any additional setup your screen has to do (ex. start pulling data from a web server).

`AppDelegate.swift` is the file containing the class that implements the delegate for your application. The app delegate is notified of several app events, things like the app is going to go in the background (aka the user tapped the home button), or that the app will stop running and that it should save any data it needs to.

Xcode Project

Another file is the project file - the blue one on the top with the name of the project. If you click on it, you will see this menu:



Notice that the list on the left has two categories: `Project` and `Targets`. A project is a collection of files. To make an executable you have to define a target. A target is a subset of files from a project that get built together to make an executable/app - or a library.

If you click on the default target the menu on top will change and display more tabs.

These tabs lets you configure the way your files get built in to an app:

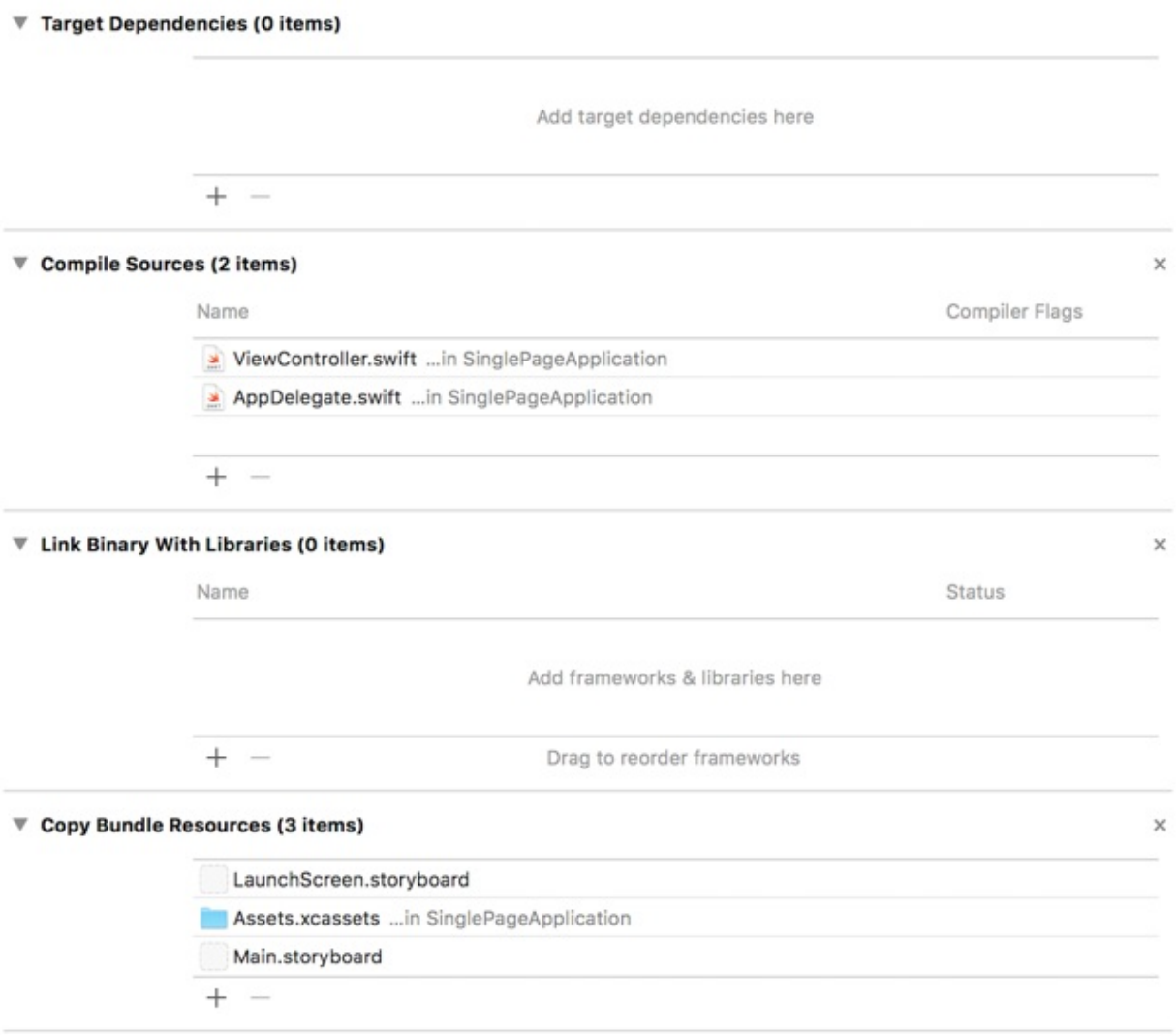
General - this view combines information from `Info.plist` with other configuration from the project file. Here you can change the `Display Name` of the application, version and build number, the main interface file, the launch screen interface file, supported orientations and much more...

Capabilities - this view lets you manage special features your app uses. For example: Push Notifications, In App Purchases, Game Center, Siri, Apple Pay, HealtKit, Maps.

Info - lets you edit `Info.plist`

Build Settings - lets you configure, at a low level, the way the code gets compiled and built.

Build Phases - this shows the phases the files go trough when the project is building



The first phase is `Target Dependencies`. If other targets are present here, they will get build before the current one. This lets you include a library into your project.

The second one is `Compile Sources`. This one compiles all swift files and links them together to form one binary file.

In the `Link Binary With Libraries` phase, the binary created in the previous one is "glued" with system libraries that provide extra features (e.g. Maps, Apple Pay)

In the `Copy Bundle Resources` phase, all resources used by your project get copied in the app `Resources` folder.

Build Rules - here you can see a list of rules for handling different file types. For example, storyboards and interface files are compiled to a binary format that is more efficient in term of storage than the one you use with interface builder. There's one rule that specifies that `.plist` files get copied and another one that compresses images before adding them in the `Resources` folder. A lot of neat things happen while building an app - you can read more examples by scrolling in the `Build Rules` tab.

To recap: a Xcode project is a bunch of files. It can have one or more targets. A target specifies how to take some - or all - of those files and mix them into an app, library or framework .

When you press play in Xcode you actually build the current target and upload it on your testing device or simulator, then you run it.

UIApplication

A thing is missing from the Swift project template. It's the `main.m` file that you can see if you create a new project with the language set to Objective-C:

```
#import
#import "AppDelegate.h"

int main(int argc, char * argv[]) {
    @autoreleasepool {
        return UIApplicationMain(argc, argv,
            nil, NSStringFromClass([AppDelegate class]));
    }
}
```

This is the last piece of the puzzle. After we untangle these lines of code you will understand how an app ticks!

iOS is written in Objective-C. Which is a language implemented on top of C by adding additional preprocessing steps. Preprocessing is simple text replacement, so Objective-C it's basically C with extra features, things like objects and protocols.

You can recognize preprocessor calls in code whenever an `@` or `#` symbol is present. A simple example of a preprocessor directive is `#include`. `#include` is followed by a path. If the path is delimited by double quotes `"` then its relative to the project, otherwise its a system library path or one mentioned in the `Build Settings` tab. The `#include` directive takes the content of a file, runs the preprocessor on it and then replaces itself with that. In other words it copies or includes a file.

Let's see what the first two lines do:

```
#import
#import "AppDelegate.h"
```

The first one imports the `UIKit` framework and the second one imports the `AppDelegate` interface - this lets the code use `UIKit` and `AppDelegate`.

Another thing about iOS is that it's actually UNIX at its core. Just like macOS, tvOS and watchOS.

In C, code execution starts from the `main` function, not from the first line. That's the starting point of all C programs. The main function gets a variable list of parameters. This comes from the fact that UNIX systems are operated using a console, you can invoke programs by typing the name of the program, followed by parameters or flags:

```
> add 1 2
> 3
```

So that's what all the craziness in the next line means:

```
int main(int argc, char * argv[]) {
    ...
}
```

That's the start of the UNIX program, so the line aftr will be the first one that gets executed. FYI `argc` is the number of parameters and `argv` is and array that holds them.

The main function has to return `0` if the program functioned as expected. That number is called the `exit code`, `0` means success. There are other exit codes.

The next line is a preprocessing step that creates an autorelease pool.

That's how the app handles memory management - all the objects created between `{` and `}` are registered in the same pool. That pool keeps track of what objects are used or not, so they can be removed from memory.

```
int main(int argc, char * argv[]) {
    ...
    return UIApplicationMain(argc, argv,
        nil, NSStringFromClass([AppDelegate class]));
    ...
}
```

That creates an `UIApplication` instance that knows to use an instance of `AppDelegate` as its delegate. That's how your code get's into the equation!

`UIApplication` is implemented by Apple so you don't have to know so many of the low level details mentioned in this chapter. It creates a delegate with your class. It creates an `UIWindow` that connects to the device's `UIScreen` - now the app will be able to show stuff on the screen!

After that it calls the `application(_: didFinishLaunchingWithOptions:)` method so you can do your custom startup logic.

It looks in the `Info.plist` file and notices that you are using `LaunchScreen.storyboard` as the launch screen - it will quickly display it on the screen while it loads your first view controller from `Main.storyboard`.

The `AppDelegate` receives general app events. Things like the app started, the app will/did go in the background, the app will/did come back from the background, the app will close. A more detailed description of the events can be found in Apple's reference.

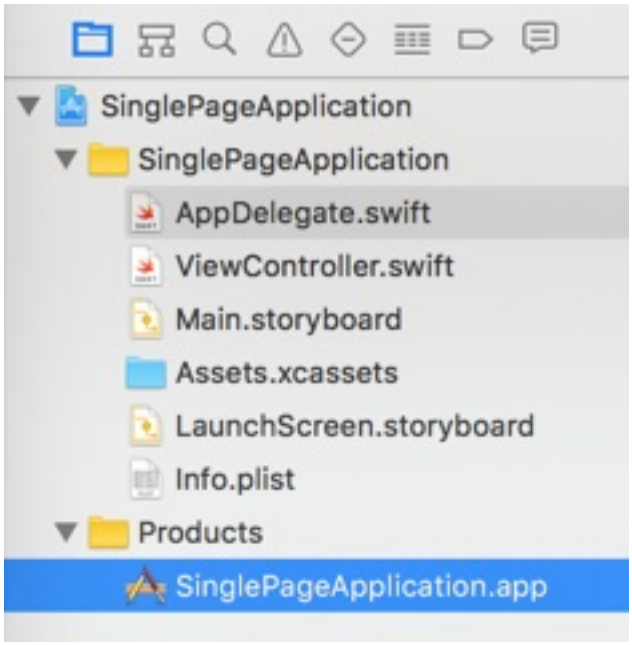
That was the magic behind the curtain. I hope I didn't bore you to death with all the low level details. Anyway, that's how an app ticks!

> TA DA!

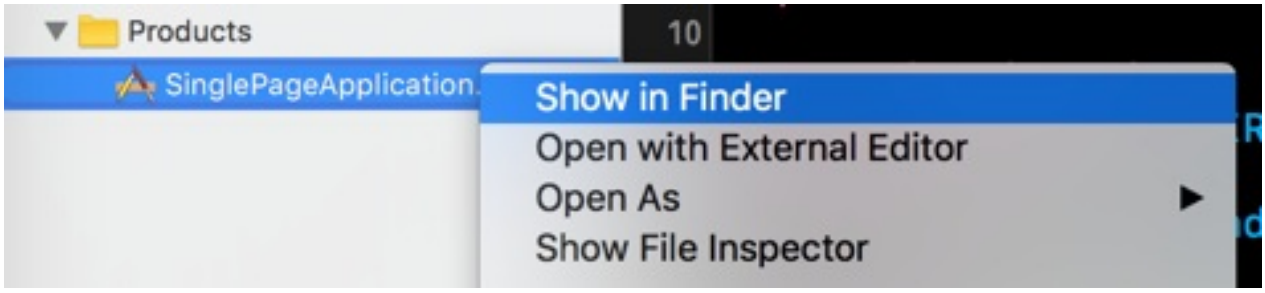
Ok, so what gets built?

When you build an app you get a file with the `.app` extension. That file is actually a folder that contains your apps resources, binaries and other information.

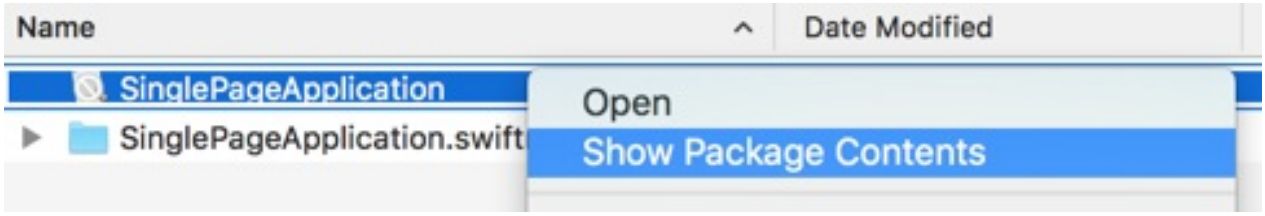
To generate the app you can either run the app with `Command` + `R` (`⌘` + `R`) or just build it with `Command` + `B` (`⌘` + `B`). After that you can find the application in the `Products` group:



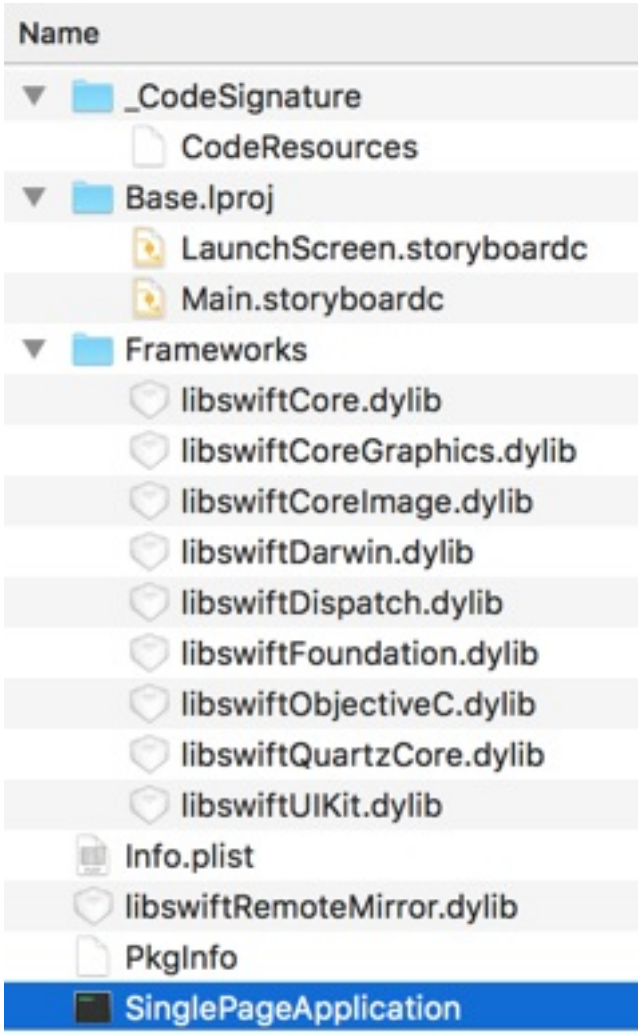
Right click to reveal the file in finder:



To show the contents of the app folder right click and select `Show Package Contents`:



Here is how the basic structure looks:



You can notice the following:

- `_CodeSignature` - a signature with information about who made the app
- `Base.lproj` - a folder with the compiled version of the apps interface files
- `Frameworks` - a list of frameworks and libraries your app needs - at minimum, you need the libraries that power Swift.
- `Info.plist` - a compiled version of `Info.plist`
- the executable with the name of the app, in this case `SinglePageApplication`
- If the app would have any resources, it would have a `Resources` folder containing them

Explore the App Delegate

Open `AppDelegate.swift` and print a different message on each method. On the simulator you can press the home button by holding `Command` + `Shift` + `H` (`⌘` + `⇧` + `H`). Try to trigger all of them :)

```
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window: UIWindow?

    func application(_ application: UIApplication,
     didFinishLaunchingWithOptions launchOptions:
        [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
        print("app loaded")
        return true
    }

    func applicationWillResignActive(
        _ application: UIApplication) {
        print("mom's calling")
    }

    func applicationDidEnterBackground(
        _ application: UIApplication) {
        print("goodbye, for now.")
    }

    func applicationWillEnterForeground(
        _ application: UIApplication) {
        print("darn it!")
    }

    func applicationDidBecomeActive(
        _ application: UIApplication) {
        print("hello!")
    }

    func applicationWillTerminate(
        _ application: UIApplication) {
        print("bye!")
    }
}
```

UIView Fundamentals

Now that we took a look at how the internal organs of an app work, we can start learning about its skin. Every time you look at your iPhone you see UIViews. UIViews everywhere!

In this chapter we are going to see what an `UIView` is, what it can do and how you can use it to make your apps.

Views 101

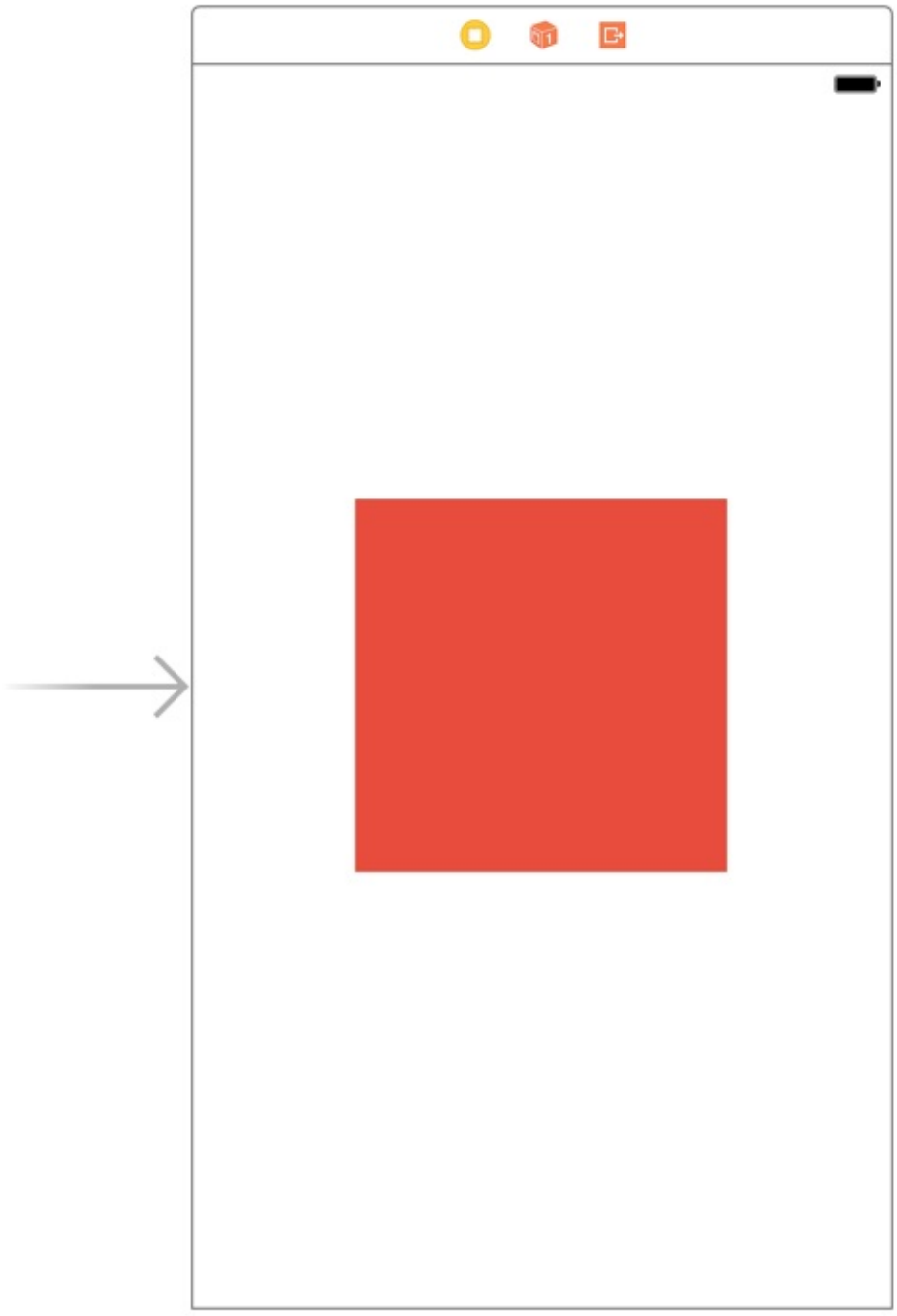
The UIView class defines a rectangular area on the screen on which it shows its content. It handles the rendering of any content in its area and also any interactions with that content - touches and gestures.

The most basic thing a view can do is to fill its area with a color.

Let's get started!

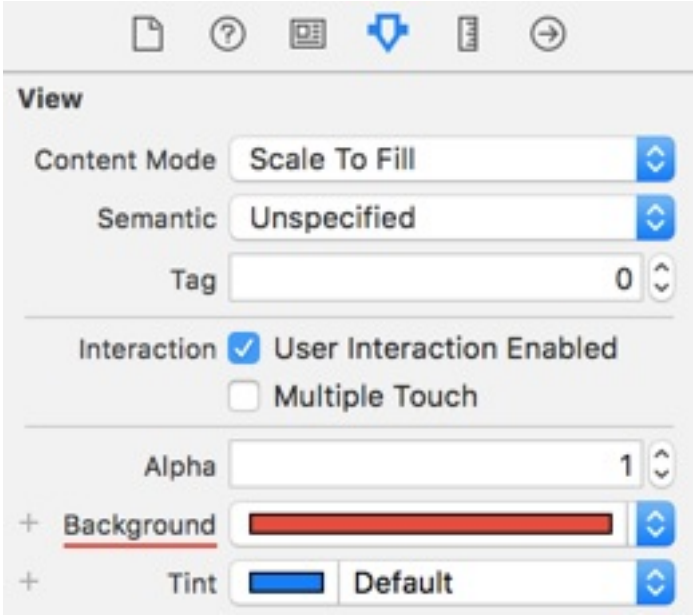
Download this starter project and click on the `Main.storyboard` to open `Interface Builder`. You should see a red square :)

>  Starter Project

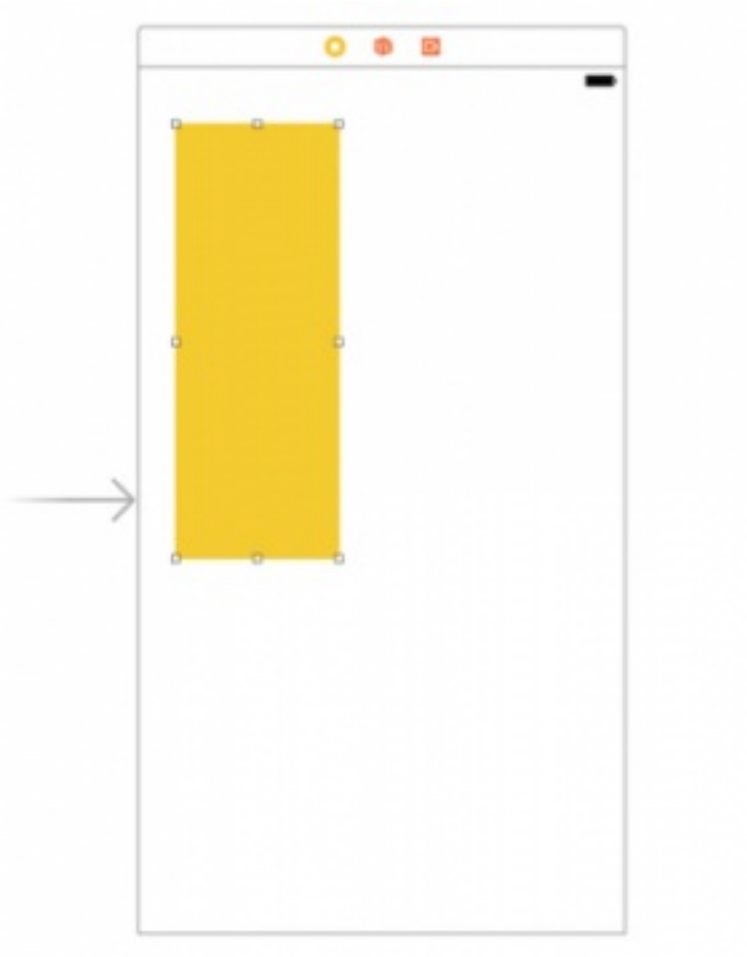


Click on it. Play around! Change its size, position.

You can change the background color from the left side menu:



I made it yellow and tall:

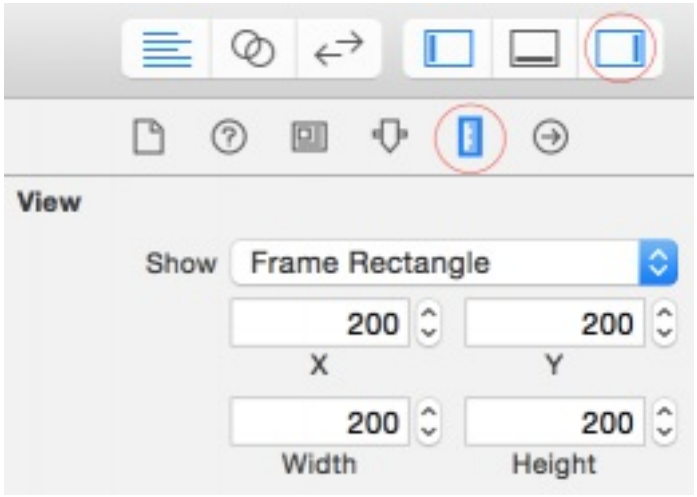


Awesome!

Now let's see what actually happened when we changed the view. The first

thing we did was to make the view taller and then move it around. Views draw content in a rectangular area. The size and position of that area are stored in the `frame` property. The frame has an `origin` - the position of the top left corner - and a `size` - the width and height of the view.

You can find the frame by opening the `Size Inspector`. Make sure your `Utilities Pane` is visible.

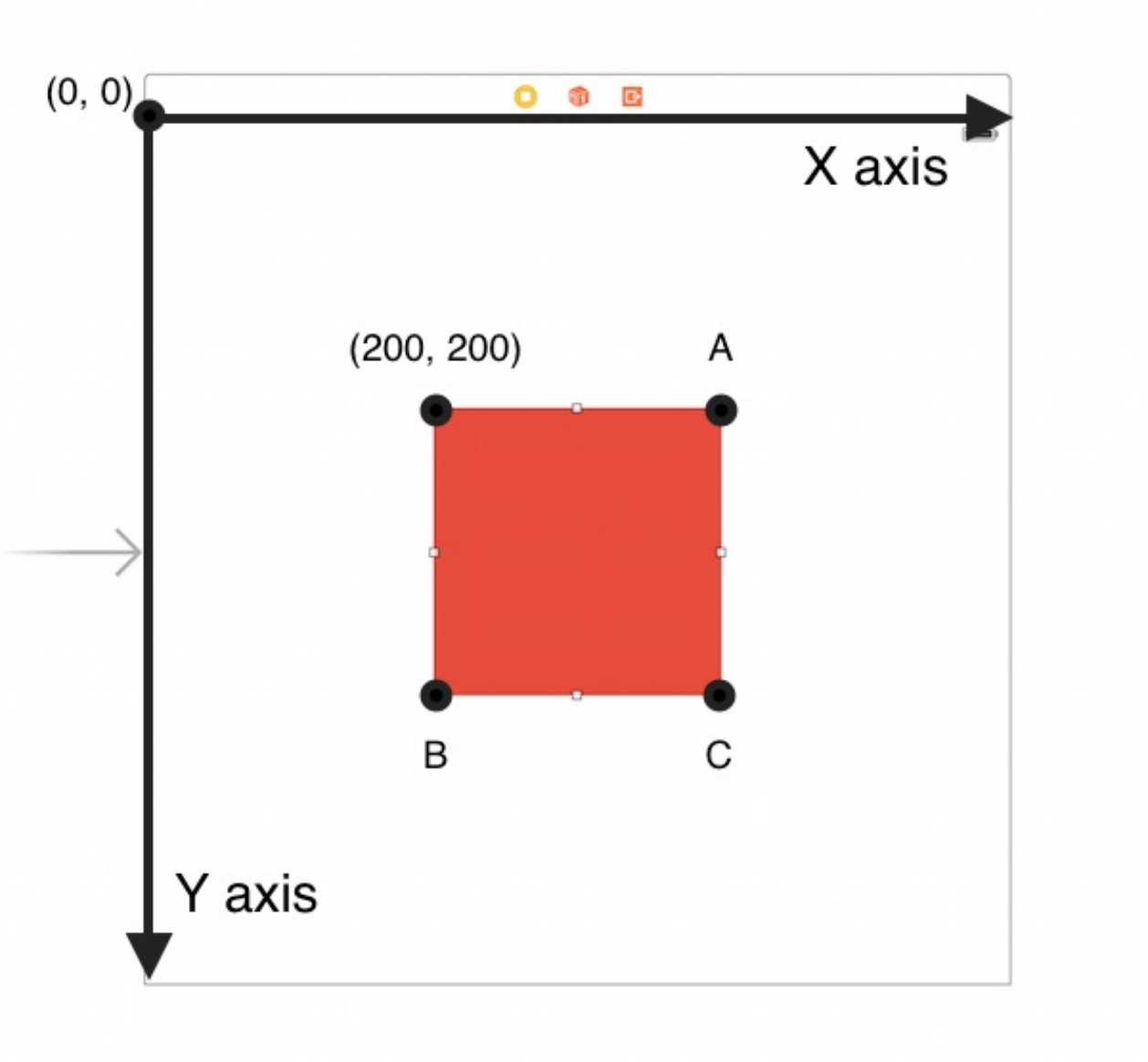


After that we changed the color into yellow. This actually sets the `backgroundColor` property of the view to a yellow color and that will make the view draw a yellow background.

Coordinate system

To understand how a frame represents a rectangle on the screen, we have to take a look at the coordinate system used by its views.

The red square has the origin in `(x: 200, y: 200)` and the size `(width: 200, height: 200)`.



The top left corner of the screen is the point `(0, 0)` or `CGPoint.zero`. In iOS we represent points using the `CGPoint` struct. The horizontal axis is the `X axis` and the vertical one is the `Y axis`. The more to the right a point is, the greater its x coordinate will be. The lower a point is on the screen, the greater its y coordinate will be.

The red square has a width of `200 points`. Can you guess the coordinates of points `A`, `B`, `C` from the image above?

The answers are on the next page.

Answers:

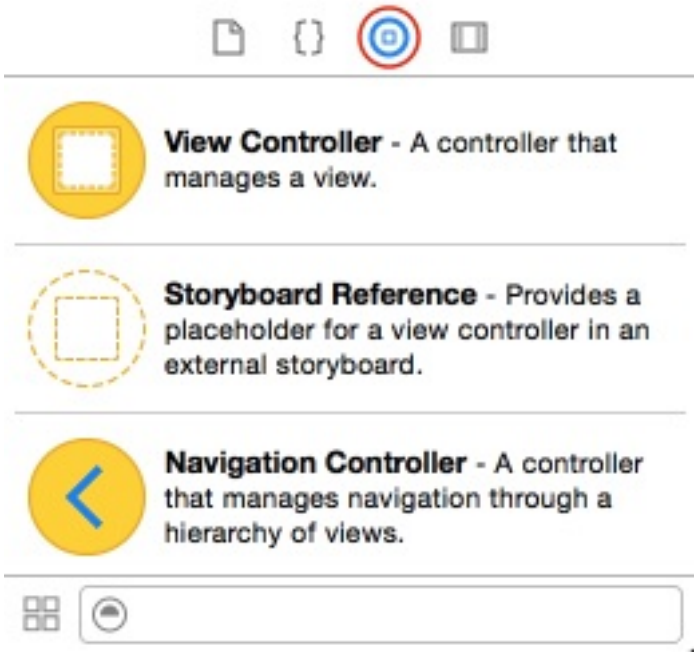
```
A = (400, 200)
```

```
B = (200, 400)
```

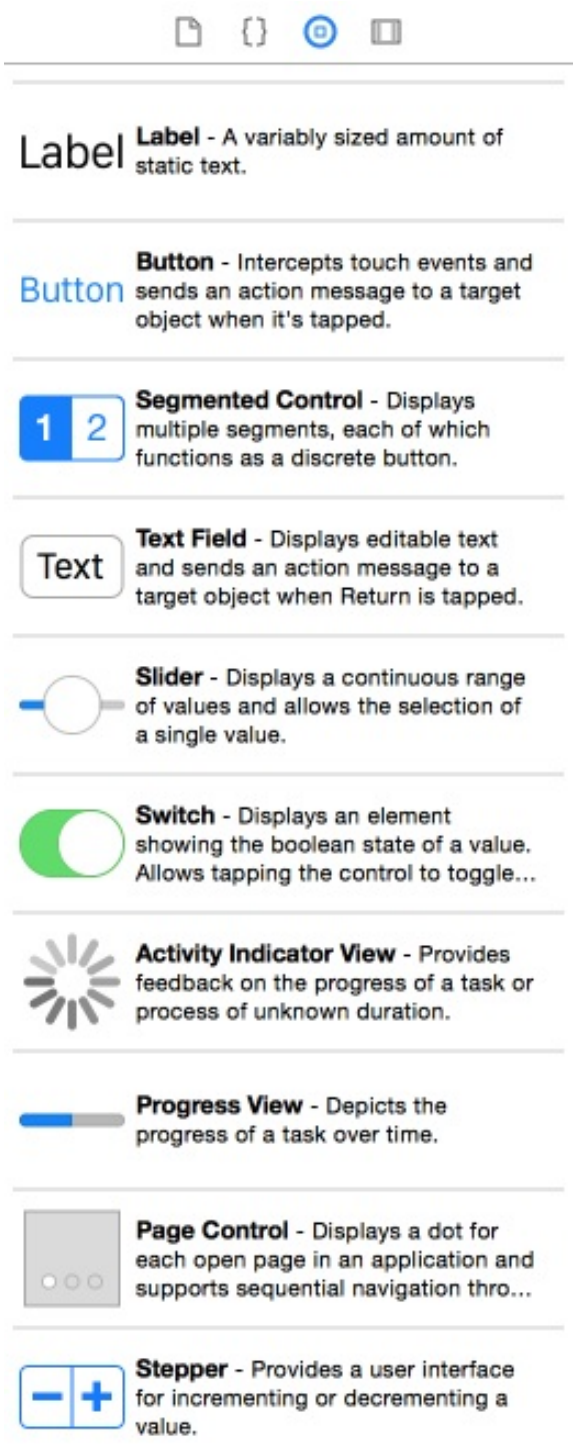
```
C = (400, 400)
```

Adding a new view

In the bottom of the `Utilities Pane` you can find the `Object Library`.



The object library contains all the types of objects that you can use to make the interface of your app. Here are a few:



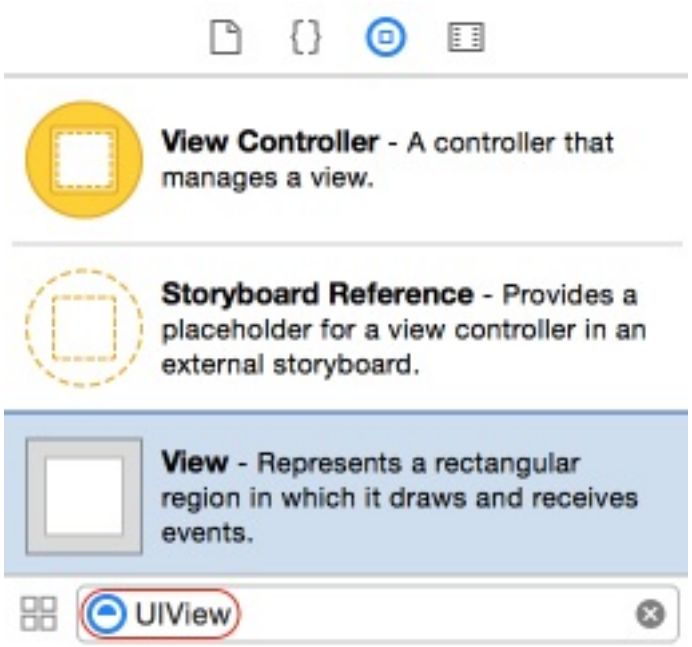
What do they have in common?

They are all subclasses of `UIView`. So all you learn about `UIView` applies to all of the components you will use :)

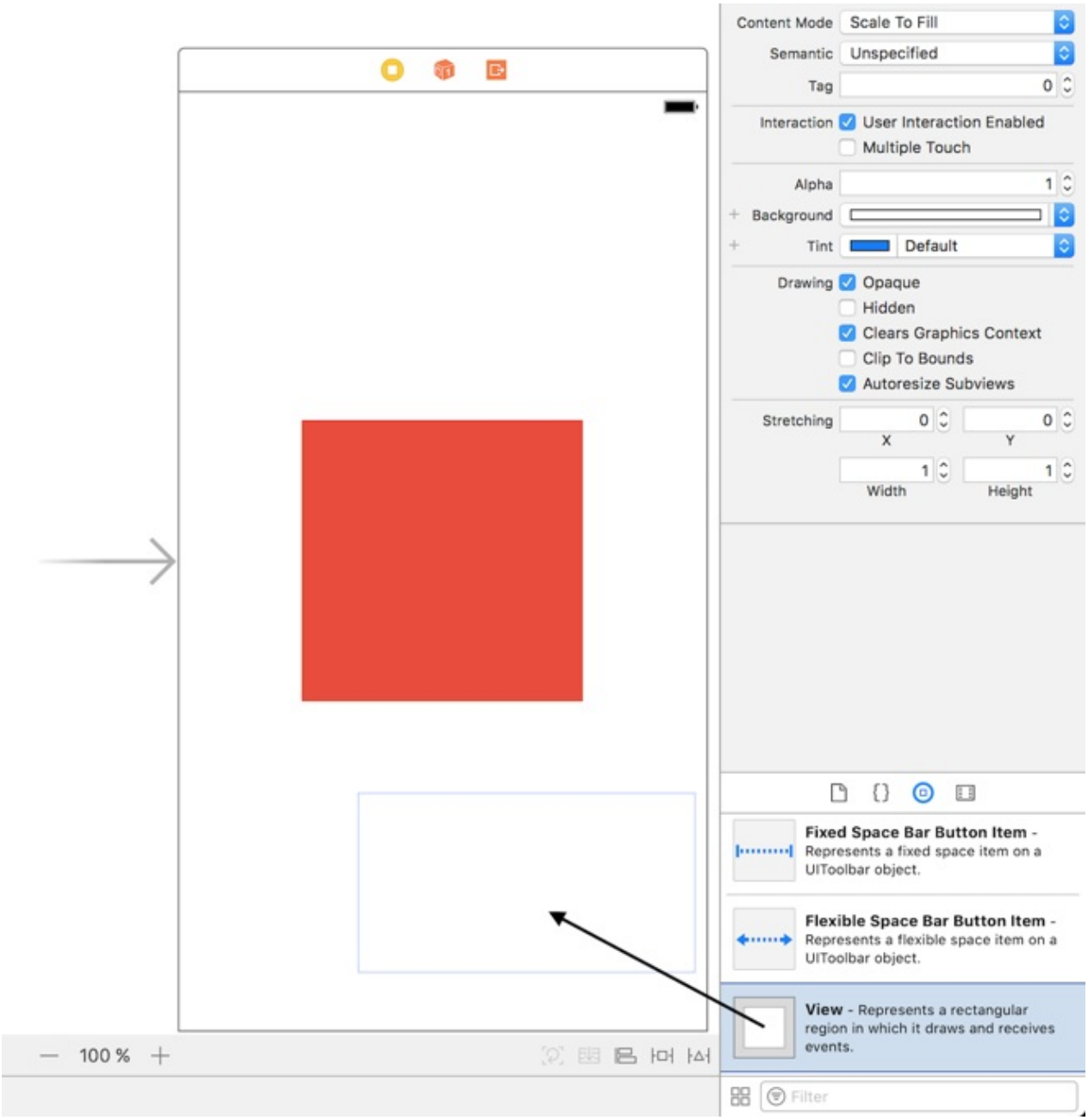
Ok, ok... Let's add that view!

In the bottom of the `Object Library` you will see a search field. Look for

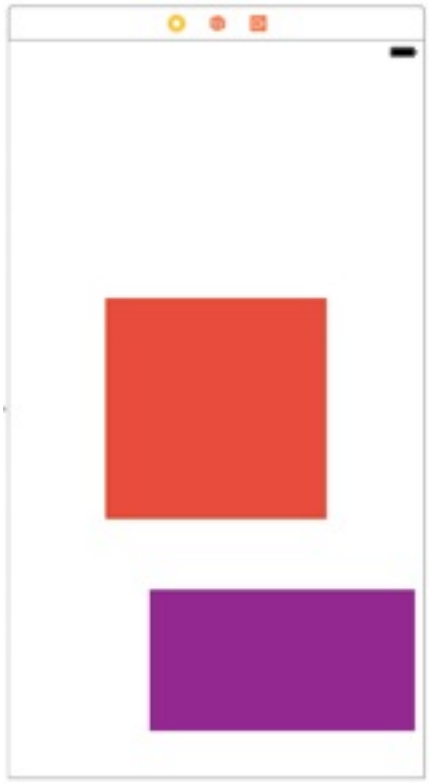
`UIView`.



Drag and drop the view object on the screen.



You will notice that the view is *invisible*. The default background color for a view is transparent - or clear color. In order to see the view, you will need to change the background color to something else.



Play around - see what other components you can add on the screen. We are going to go over them in detail later in the book.

Creating a view from code

The iOS SDK uses a design pattern named Model View Controller - MVC - to display data on the screen. The pattern splits the objects that make an app in three categories: models, views and controllers. The model or models, refers to different data your app might use - like a user profile or the GPS position of the phone. Views are viewable objects - things you can see on the screen. And finally, controllers are the glue in between.

> If you want to read more about object oriented programming and design patters in swift read our OOP guide.

In iOS, the MVC pattern is implemented with a small twist; all controllers are actually `View-Controllers` - controllers that have views. All view controllers are a subclass of `UIViewController` and their view can be accessed by

calling the `view` property. Because of this, a lot of functionality is built in `UIViewController` and `UIView`, that helps us developers deliver a consistent user experience.

To show a view on the screen you need to add it as a subview to a view that is already on the screen, for example, the current view controller's view or one of its subviews. You do this by calling the `addSubview` method.

First open `ViewController.swift`. The file should look like this:

```
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

    }

}
```

Zen bits

File and project templates usually contain redundant comments and boilerplate code. Take a few seconds and remove them after creating new files or projects. Your code will look much better ;)

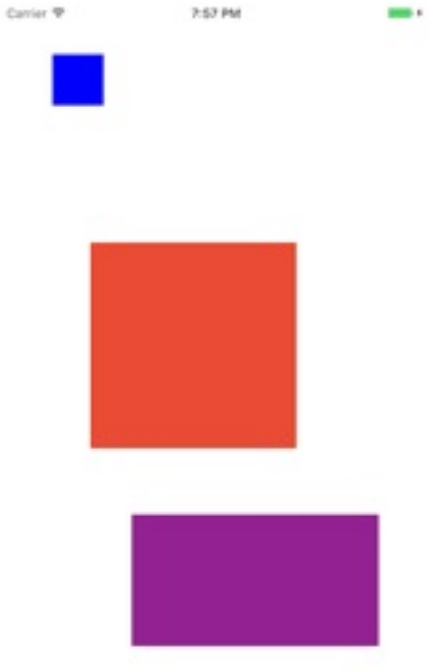
The `viewDidLoad` method is called when the screen has loaded its interface. In this case, whatever you have in your storyboard. User interfaces can also be loaded from NIB files or directly from code. `viewDidLoad` is the most common place where you will add you setup logic for your screen.

Let's add another view to the screen. First we create a frame for it. Then we instantiate a new `UIView` object for that frame. And to make it visible, we

are going to make the background color blue. After that we add it as a subview to `view`.

```
class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        let frame = CGRect(x: 50, y: 50, width: 50, height: 50)
        let blueSquare = UIView(frame: frame)
        blueSquare.backgroundColor = .blue

        view.addSubview(blueSquare)
    }

    ...
}
```

If you run the app now, you are going to see something like this:



The type of the `backgroundColor` property is `UIColor`. Because swift knows that you will assign a `UIColor` to it, infers that `.blue` actually means `UIColor.blue`.

Nesting views

Before we did not explain a step: adding `blueSquare` as a subview to `view` (the view of the view controller).

Views can have other views that are drawn inside them. They are called **subviews**. The coordinate system of subviews has the origin (the point `(0, 0)`) in the top left corner of the parent view - also called **superview**. The list of subviews a view has can be accessed by calling it's `subviews` property. You can also access the superview of a view by calling it's `superview` property.

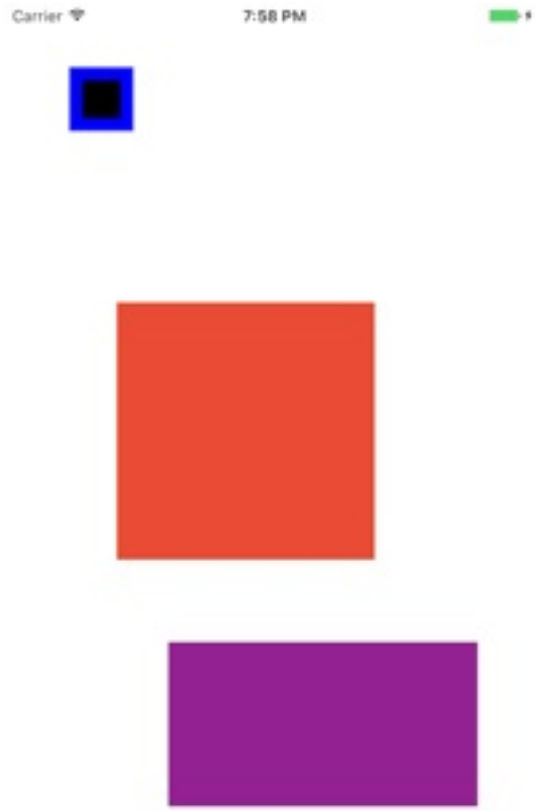
When we called `view.addSubview(blueSquare)` we added the blue square inside the view of the screen, which is inside a window. The `UIWindow` class inherits from `UIView` - its purpose is to draw a screen. Unless the iOS device running the app has access to another screen, the app will have only one window.

You can remove a view from it's superview by calling the `removeFromSuperview()` method.

Let's add another view inside `blueSquare`.

```
class ViewController: UIViewController {

    override func viewDidLoad() {
        ...

        let smallFrame = CGRect(x: 10, y: 10, width: 30, height: 30)
        let blackSquare = UIView(frame: smallFrame)
        blackSquare.backgroundColor = .black

        blueSquare.addSubview(blackSquare)
    }

    ...
}
```

If you run the app now, your screen should look like this:

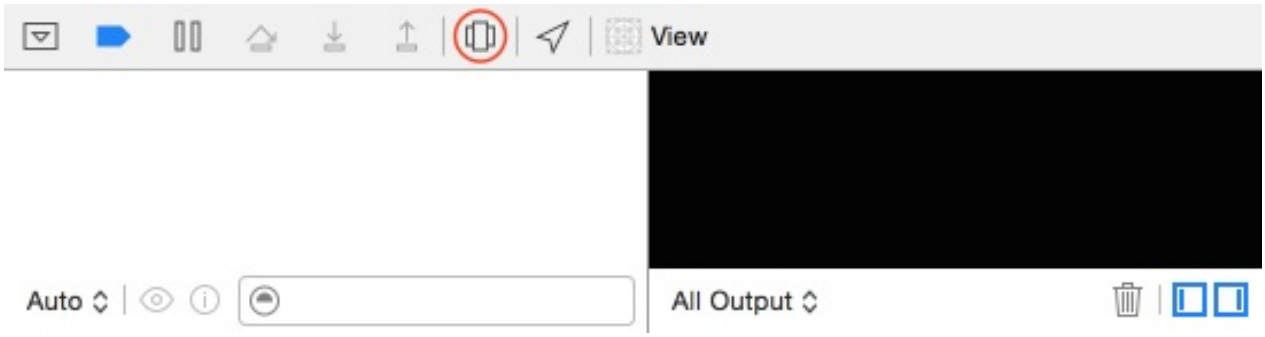


If you want to see the view hierarchy you can pause your app and take a look.

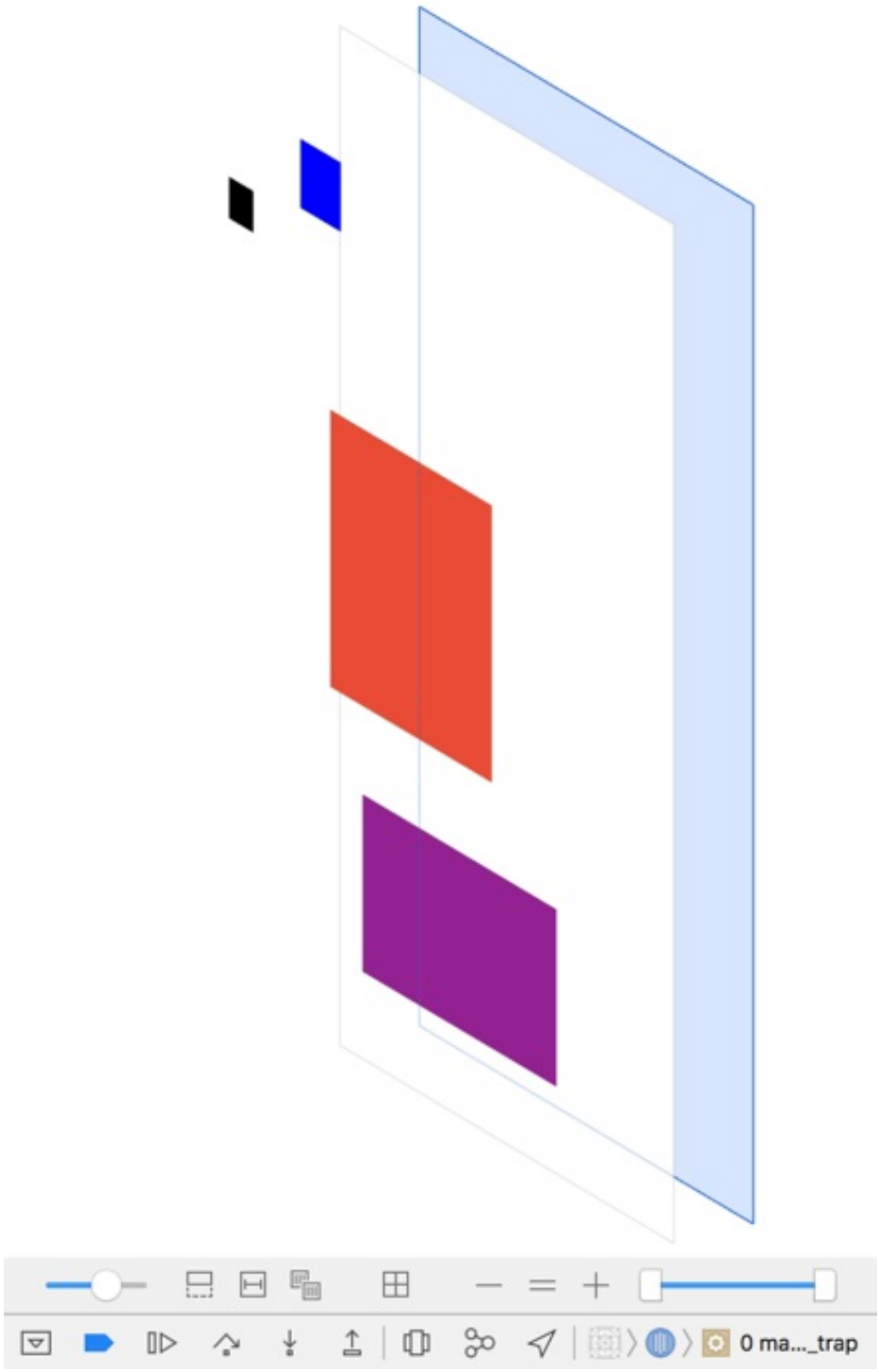
First open the `Debug Area`. In the top right corner of Xcode you have a segmented control that looks like this:

Make sure the middle on is blue.

Run the app. In the lower mid section of Xcode find and click `Debug View Hierarchy` button:



If you followed all the steps you should see an interactive 3D representation of the view hierarchy. You can move it around by clicking and dragging.



the blue rectangle in the back is the window

Interface Outlets

After designing your interface you might need to access a few components in order to populate them with content or add extra customisation.

You can get a reference to an interface object by creating an `IBOutlet` for it. To do this, add a property in your view controller with the type of the object you want to connect and mark it as an outlet using the `IBOutlet` declaration attribute.

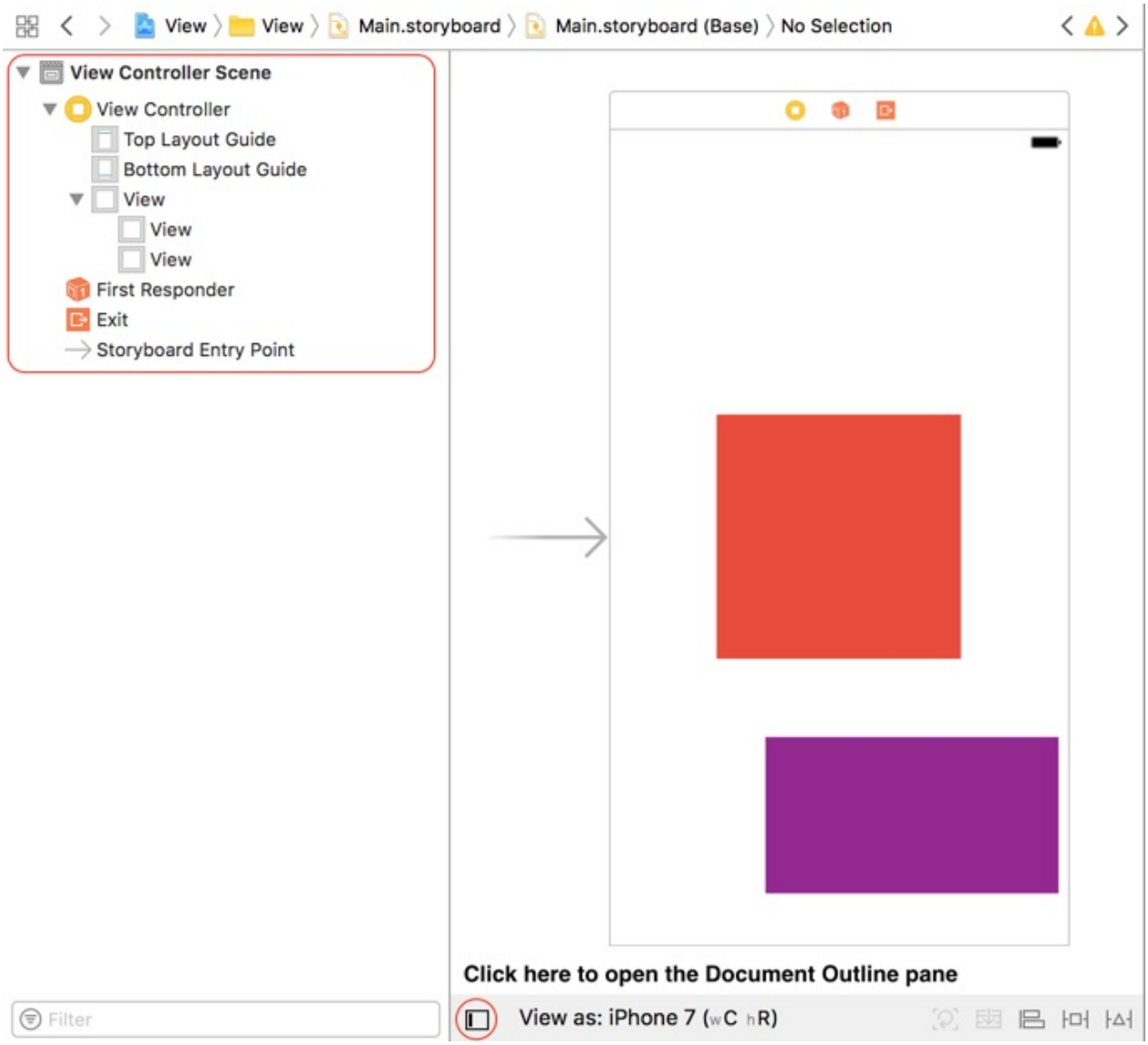
Let's connect that red square :)

First we make a property and mark it as an `IBOutlet`:

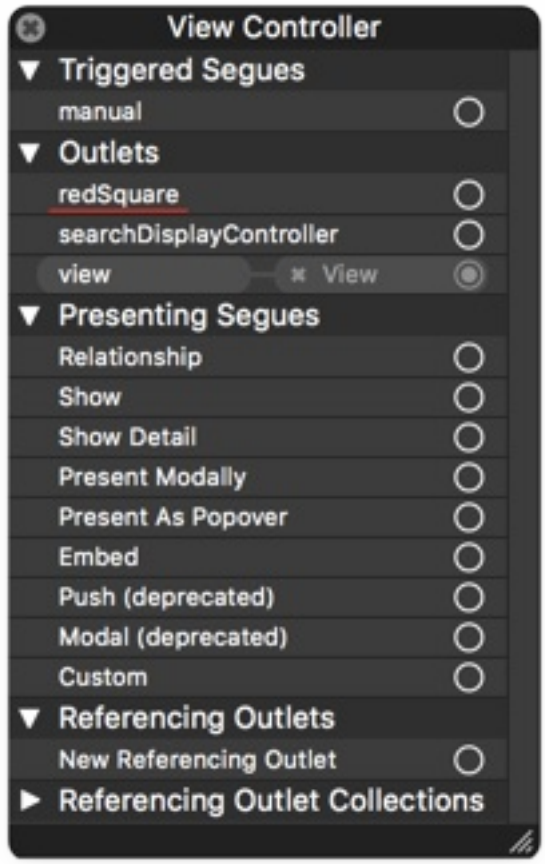
```
class ViewController: UIViewController {
    @IBOutlet var redSquare: UIView!

    ...
}
```

> The property is of type `UIView!` which is an implicitly unwrapped optional. The reason for this is that when you create the ViewController you don't have the interface loaded. But after that is done you can assume that you have these objects and don't want to test if they exist every time you use them.

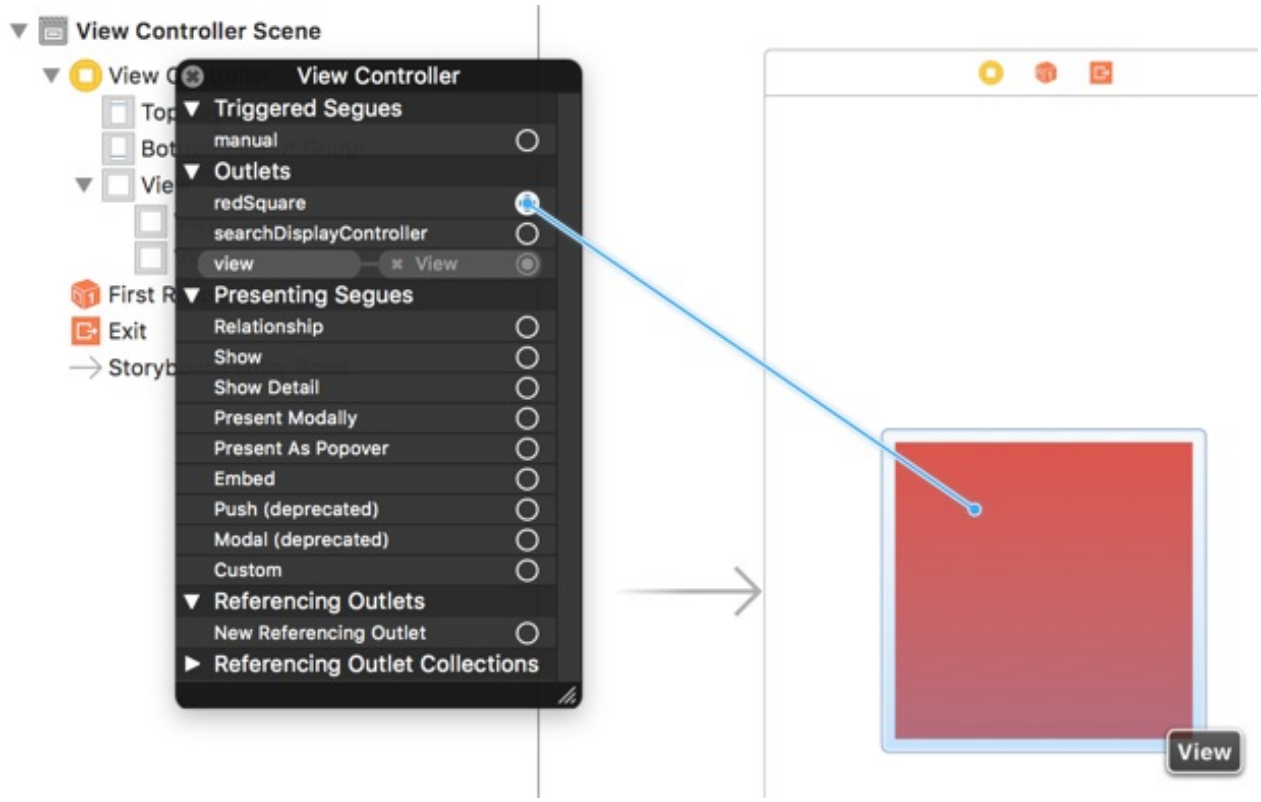
Open `Main.storyboard`. Make sure you have the `Document Outline` pane open - here you can see all of the objects form the interface.



Right click `View Controller` - it has a yellow icon. A popup should appear. In it there is a list of outlets you can connect to the view controller. If you look closely, you will spot the newly added `redSquare` outlet.



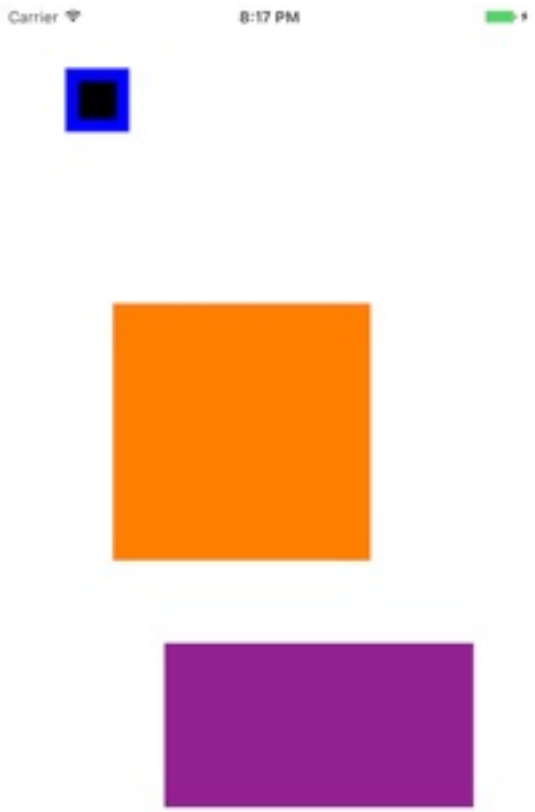
Connect it to the red view in the interface.



Great! Now that we have a reference to it, we can change its properties at runtime.

```
class ViewController: UIViewController {
    @IBOutlet var redSquare: UIView!

    override func viewDidLoad() {
        super.viewDidLoad()

        redSquare.backgroundColor = .orange

        ...
    }
}
```

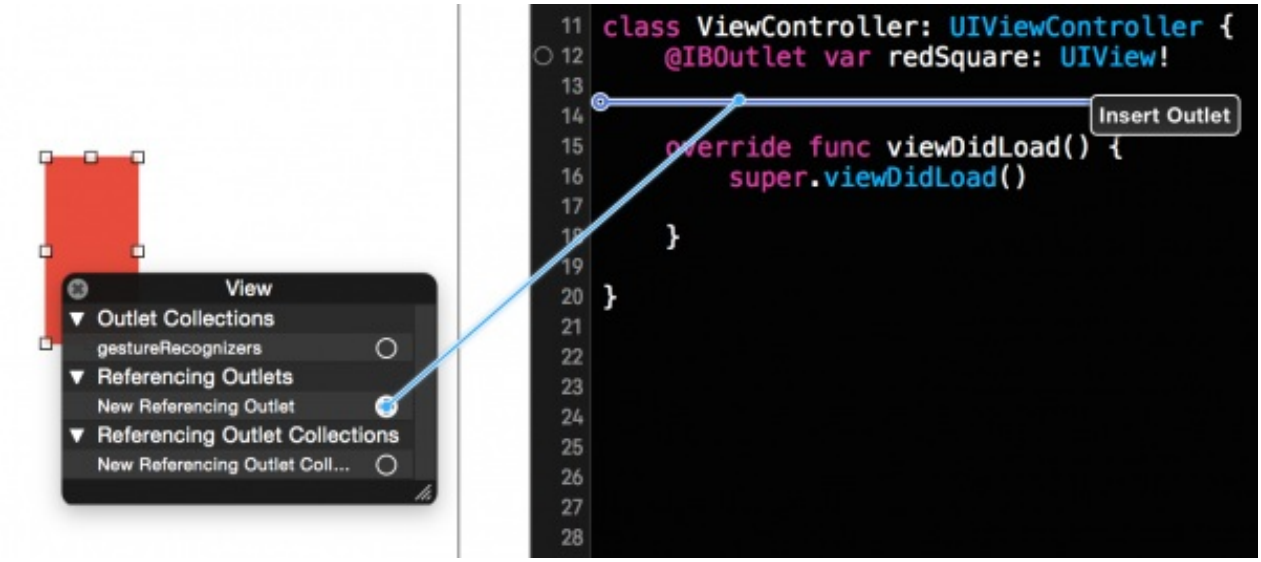
If you followed with all the steps you should see an orange view instead of the red one. You can change any view attribute at runtime and the changes should immediately be visible.



There is a shorter version for adding an outlet:

Open the `Assistant Editor` by pressing the button with two circles.

Right Click on the view you want to add an outlet to. Then drag an drop the outlet directly in the code of the view controller.



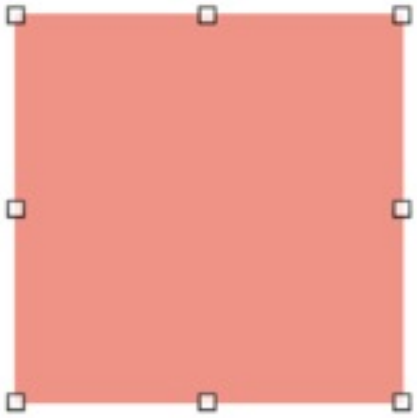
Give it a name and set the type if necessary. That will write the `@IBOutlet` code for you.

Useful view properties

So far we learned how to change the size, position and background color of a view. We can customise the appearance and behaviour of views by changing other properties.

alpha

If the value is smaller than `1`, the view and all its subviews will become transparent.



transform

You can apply affine transformations to a view.

```
view.transform = rotateAndShrink
```

I'm not going to go into detail on this topic. I will only mention two transformations that you will likely use.

Scale

The scale transform has two factors - one for the x axis and another for the y axis.

```
let scale = CGAffineTransformMakeScale(0.5, 0.5)
```

> The above transformation makes the view half its size

Rotate

The rotate transform rotates the view by a number of degrees. The value given to `CGAffineTransformMakeRotation` is expressed in radians. A positive value means a counterclockwise rotation and a negative value means a clockwise rotation.

```
let rotate = CGAffineTransformMakeRotation(CGFloat(M_PI_2 / 2))
```

```
```

> The above transformation rotates the view by 45 degrees counterclockwise

Combining two transformations

You can combine two transformations into one using `CGAffineTransformConcat`. The order does not matter.

```
let transformation = CGAffineTransformConcat(scale, rotate)
```

tag

The `tag` property is exactly what its name says. You can label a view with an `Int` value. The default value is 0. This is useful when you have multiple views with the same functionality (ex. the number buttons from a calculator app - you can use the tag to extract the digit and all the logic is the same).

hidden

Sometimes you don't want to show a view, but you want it to remain in the view hierarchy. Set the value of `hidden` to `true` to hide a view. **All touch events will be ignored while the view is hidden.**

bounds

`bounds` represents the frame of the view in its own coordinate system, in other words, a frame with the same size and origin in `.zero`.

When is this useful?

Good question!

I keep on seeing this pattern. You decide to add a background image in a screen.

```
let backgroundFrame = CGRect(x: 0,
                             y:0,
                             width: view.frame.width,
                             height: view.frame.height)
let backgroundImage =
    BackgroundImage(frame: backgroundFrame)
```

Why not?

```
let backgroundImage =
    BackgroundImage(frame: view.bounds)
```

> Note: The class `BackgroundImage` does not exist. We will discuss images in another chapter.

center

It gives you the coordinate of the view's center in the coordinate system of its superview.
Maybe I like math too much, but if you like designing custom animations or controls, you will definitely like this property.

The Layer

The view uses a `CALayer` object to render its components. It can be accessed by calling the `layer` property.

While the layer is used to draw the view, it also has a few visual attributes that you can use to customise the view. The most common ones are:

borderWidth

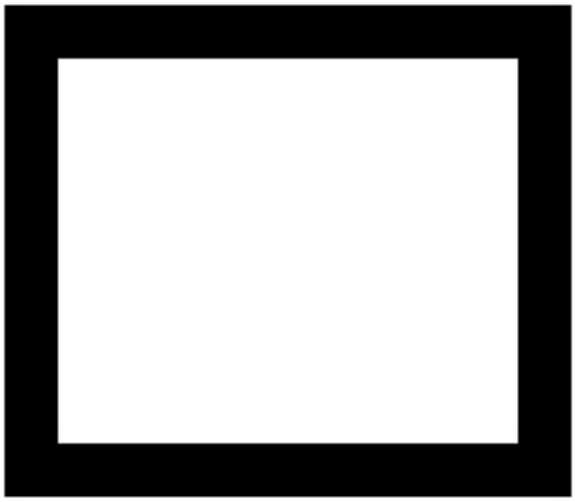
You can add a border to the drawing area by setting the `borderWidth` property to a value greater than 0.

With border `1.0`:

```
view.layer.borderWidth = 1.0
```

With border `10.0`:

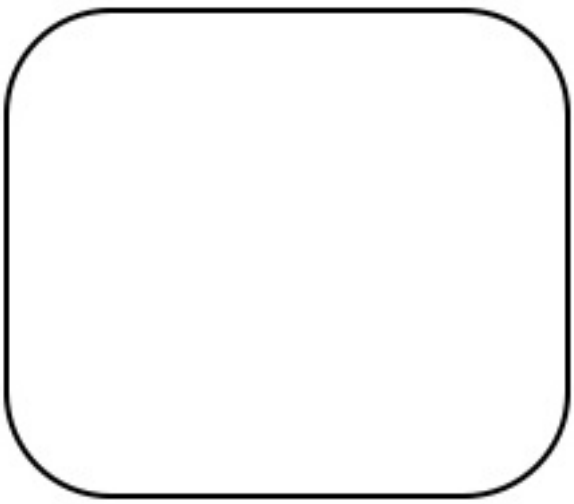
```
view.layer.borderWidth = 10.0
```



cornerRadius

Ever wondered how those rounded corners are made? Search no more! `cornerRadius` is what you are looking for!

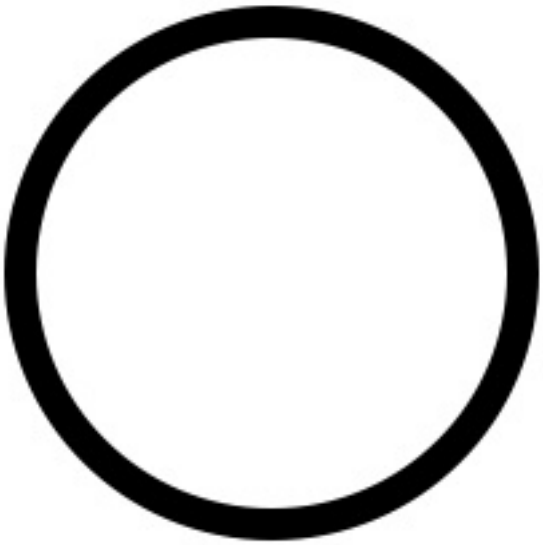
```
view.layer.cornerRadius = 40.0
```



If you have a square view and you set `cornerRadius` to half the width you

will get a circle.

```
view.layer.cornerRadius = view.frame.width / 2.0
```



borderColor

You can set the color of the border by setting the `borderColor` property to a `GCColor`. You can convert any `UIColor` to a `CGColor` by calling its `CGColor` property. (ex: `UIColor.blackColor().CGColor`)

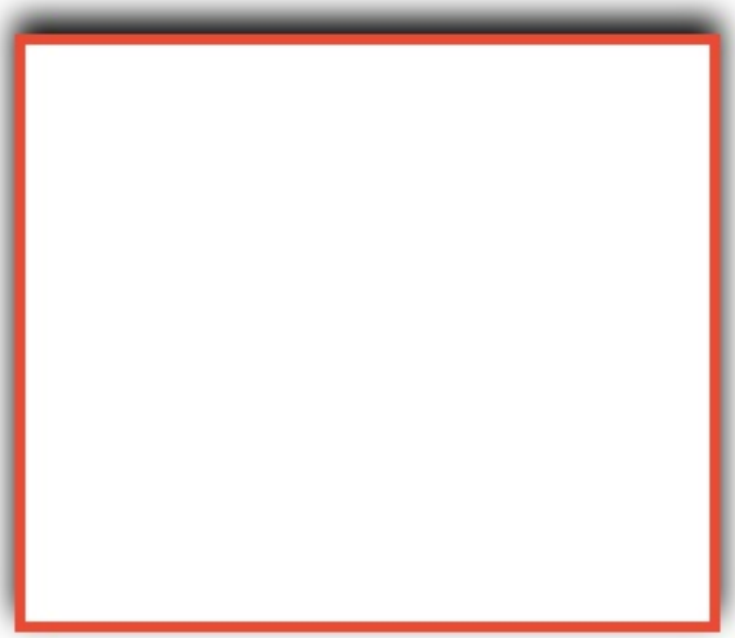
```
view.layer.borderColor = UIColor.blueColor().CGColor
```

Shadows

shadowOpacity

To enable shadows on a view you will need to set the `shadowOpacity` property to a value greater the `0`. The default value is `0`.

Doing so on a view will give you this effect.



This is because shadows can be customised from other properties that have a default value.

shadowOffset

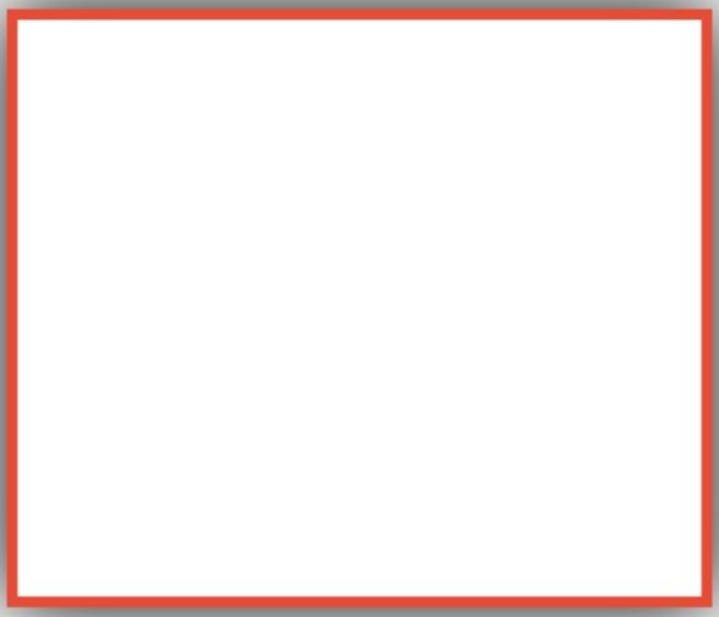
The shadow can be offset to give different effects based on the supposed *light* might be. The reason the default shadow was so high is that the default value of `shadowOffset` is `(0, -3)`.

Here you can see the shadow with offset `(0, 0)`:

shadowRadius

The shadow shows in a band round the layer. The width of that band is called `shadowRadus`. The default value is `3.0`.

Here is how the shadow looks at radius `10.0`:



shadowColor

You can make the shadow have any color you want. I personally like purple shadows - they are the best. No really - stick to black or gray.

IBDesignable

You can expose more customisation points in Interface Builder by making certain properties `IBInspectable` and the class `IBDesignable`.

Marking a class `IBDesignable` tells interface builder to run the code from that view and render it instead of showing a simulated version.

Marking a property as `IBInspectable` tells interface builder to show a field for that property.

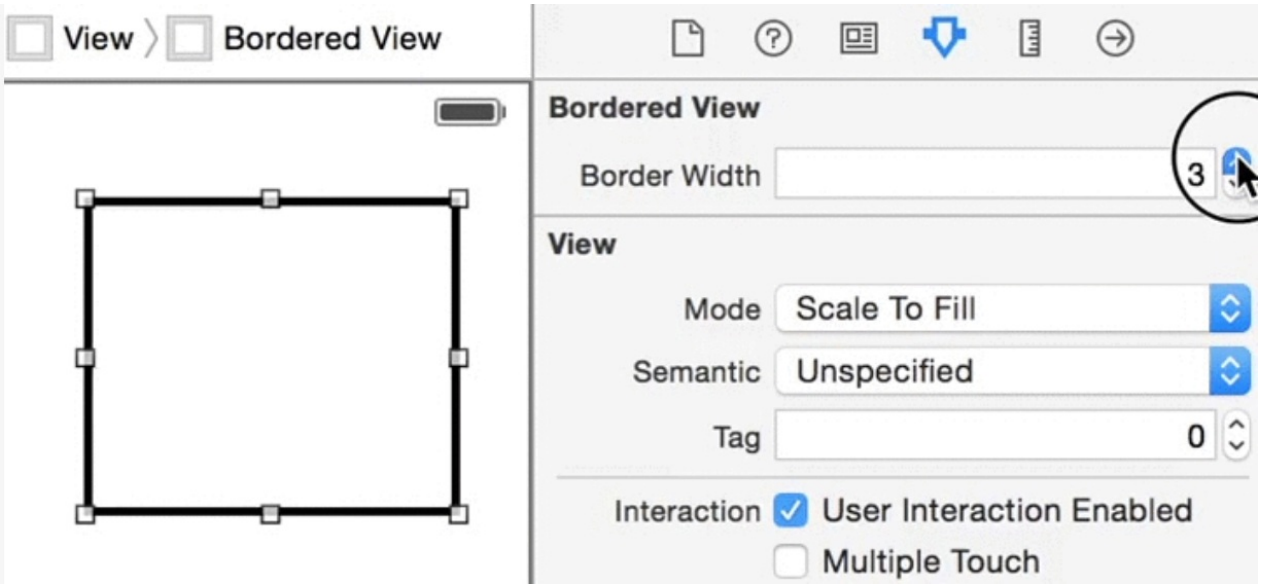
Here is a simple example of a view that exposes the `borderWidth` layer property.

```
@IBDesignable
class BorderedView: UIView {
    @IBInspectable var borderWidth: CGFloat = 0 {
        didSet {
            layer.borderWidth = borderWidth
        }
    }
}
```

To use it. Add a `UIView` in your screen. Select it. Then go to the `Indentity Inspector`.

Change it's class to `BorderedView`.

Then go to the `Attributes Inspector`. You should see the `Border Width` field:



> Learn more about **IBDesignable** from our tutorial: How to make awesome UI components

Conclusions

Although `UIView` might seem small, it's actually the blood and skin of iOS. Knowing how to use it is critical if you don't want to reinvent the wheel. There are a lot of customisation points for views, by combining them you can avoid writing extra code and make better apps.

Apple encourages developers to subclass `UIView` when they need to make a visual component that requires user interaction, but only when the standard systems views do not solve your problem. In the next chapter we are going to start learning how to use some of them.

Exercises

1. Target

Write the code to make a target :)

Solution

```
let width = 20
let padding = 20

let red = UIColor(red: 192.0/255.0, green: 57.0/255.0, blue: 43.0/255
let blue = UIColor(red: 41.0/255.0, green: 0.5, blue: 185/255.0, alph
let colors = [red, blue, red]

var l = (colors.count * (width + padding) - padding) * 2
var parent = view

for color in colors {
    let frame = CGRect(x: padding + width, y: padding + width, width:
    let circle = UIView(frame: frame)

    circle.backgroundColor = UIColor.whiteColor()
    circle.layer.borderColor = color.CGColor
    circle.layer.borderWidth = CGFloat(width)
    circle.layer.cornerRadius = CGFloat(l / 2)

    parent.addSubview(circle)

    parent = circle
    l -= 2 * (width + padding)
}
```

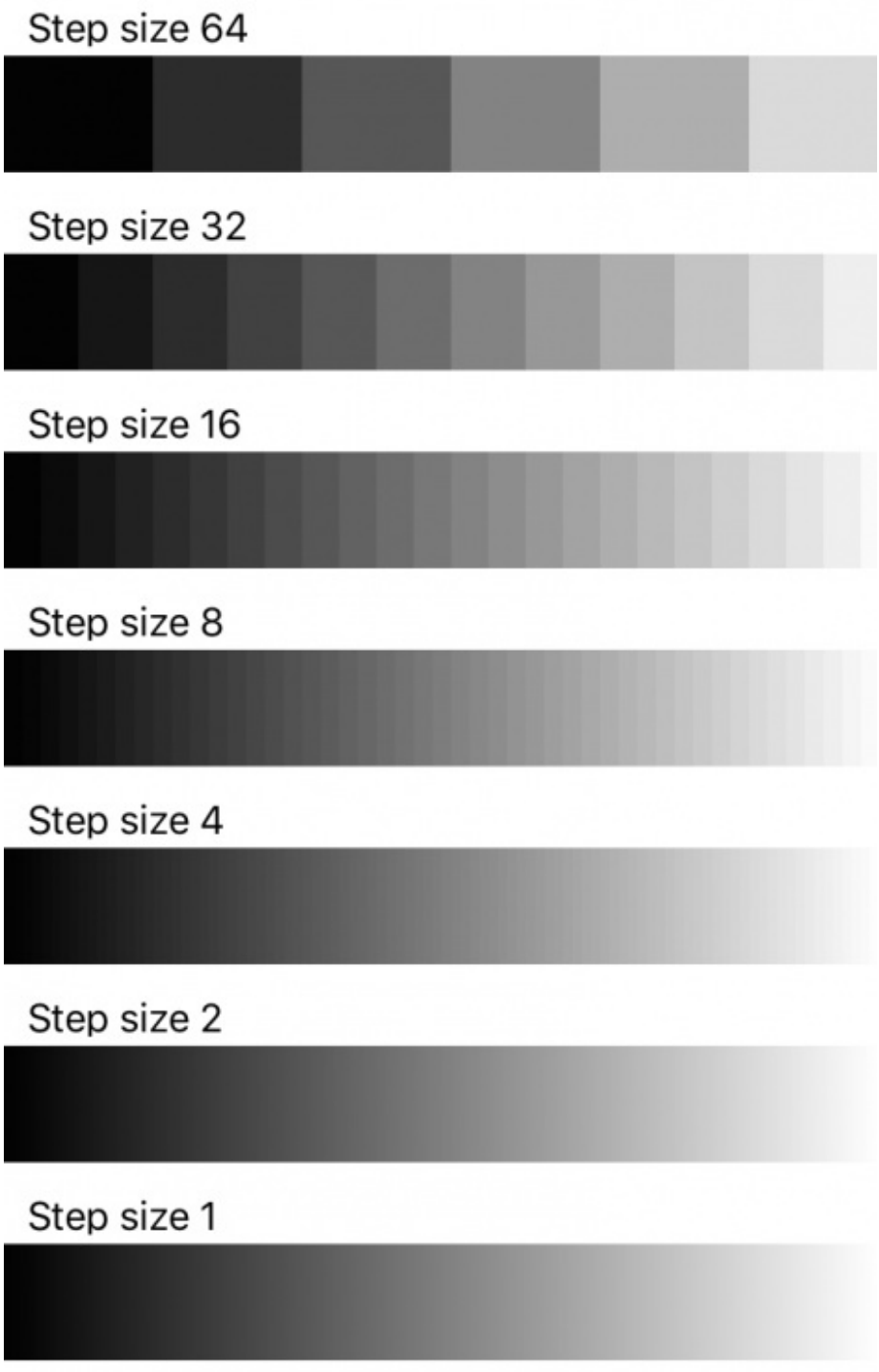
2. Gradient

Make a gradient that goes from black to white using only `UIViews`.

You can use `UIColor(white:alpha:)` to create the different shades of gray. The `white` parameter should have a value between `0.0` *(black)* and `1.0` *(white)*.

Hint

Try adding views in a line and change their background color. If you make them thin enough, it will look like a gradient.



Solution

```
let height: CGFloat = 50
let lineWidth: CGFloat = 1
```

```
let width = view.frame.width

var x: CGFloat = 0.0

while x < width {
    let frame = CGRect(x: x, y: 50, width: lineWidth, height: height)
    let line = UIView(frame: frame)

    line.backgroundColor = UIColor(white: x / width, alpha: 1)
    view.addSubview(line)

    x += stepSize
}
```

> Warning: this is not a practical approach! In real life you would use an image background that can stretch or CAGradientLayer.

3. UberView

Make a view that exposes the `borderWidth`, `borderColor` and `cornerRadius` layer properties.

Solution

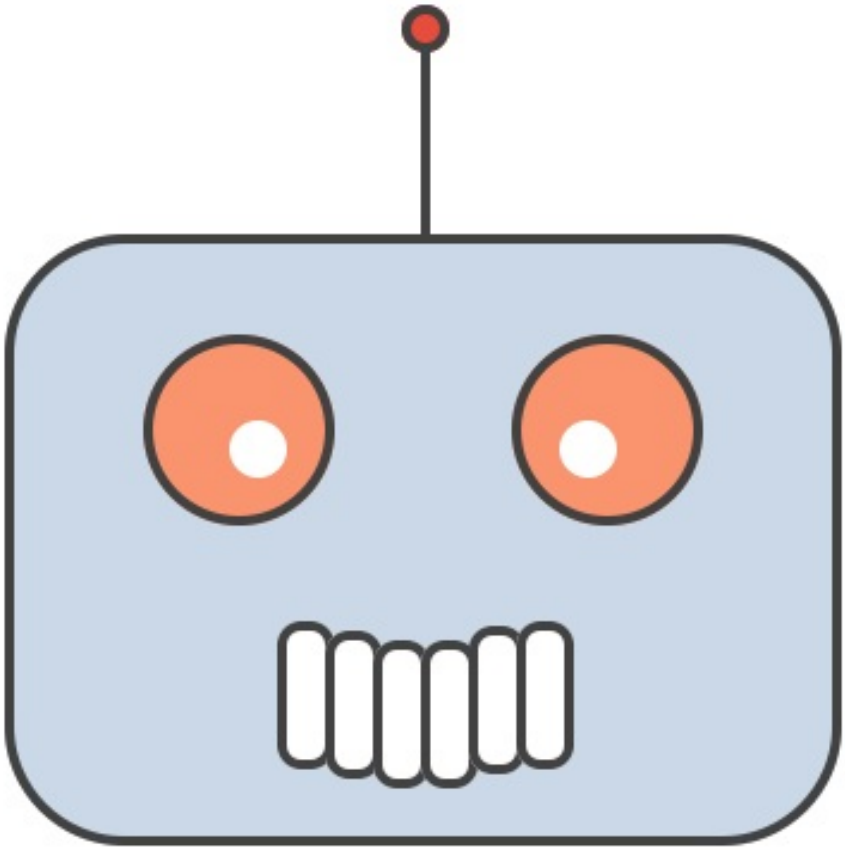
```
@IBDesignable
class UberView: UIView {
    @IBInspectable var cornerRadius: CGFloat = 0 {
        didSet {
            layer.cornerRadius = cornerRadius
        }
    }

    @IBInspectable var borderWidth: CGFloat = 0 {
        didSet {
            layer.borderWidth = borderWidth
        }
    }

    @IBInspectable var borderColor: UIColor = UIColor.blackColor() {
        didSet {
```

```
                layer.borderColor = borderColor.CGColor
            }
        }
    }
```

4. Robot

Use `UberView` to design a robot.

Here's mine:



Working with Colors

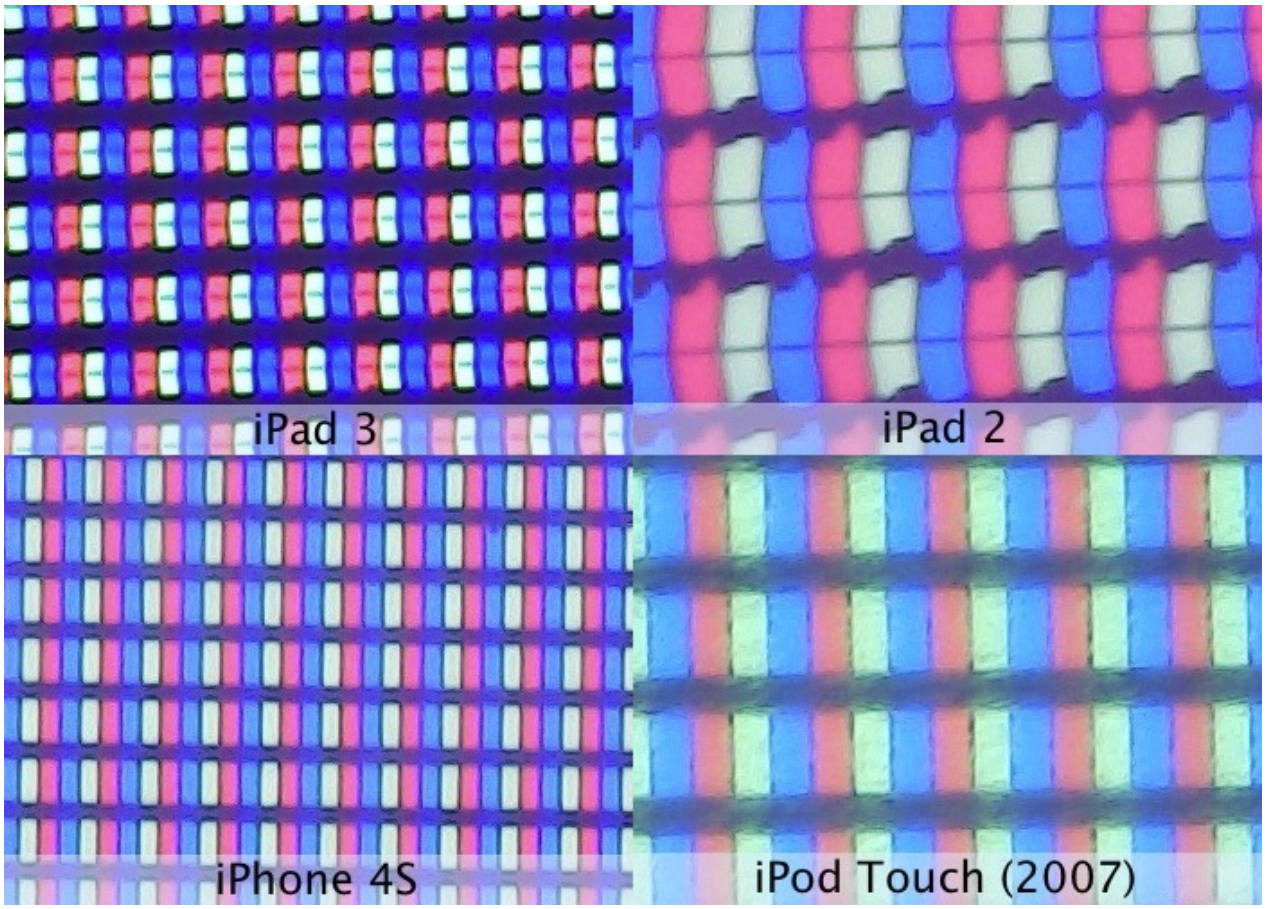
What will you learn

- How to represent colors in code
- How to create a new one using just numbers
- How to combine two color
- How to get lighter or darker shades

What is color to a computer?

The color of a pixel is described by the amount of red, green and blue light it emits.

In case you didn't already know, there are thousands of tiny colored lights/LEDs on the screen that you are using to read this article. Under a microscope, your screen looks like this:



The computer only knows how much electricity to give to each one of the three leds that represent a pixel. So if a pixel is red, the green and blue lights get low or no electricity, and the red one gets a lot.

RGB Color Model

The **RGB** color model (stands for **R**ed **G**reen **B**lue) is an additive color model in which red, green and blue light are added together in various ways to reproduce a broad spectrum of colors.

You might remember from school that red, blue and yellow are primary colors. And that is still true!

But there are two different things:
1) how colors combine if they **reflect light**
2) how colors combine if they **emit light**

When you use paint, light (either from the sun or from another source) hits the paint and then reflects into your eyes. When you use pixels on a screen, light is emitted from the screen and then hits your retina :)

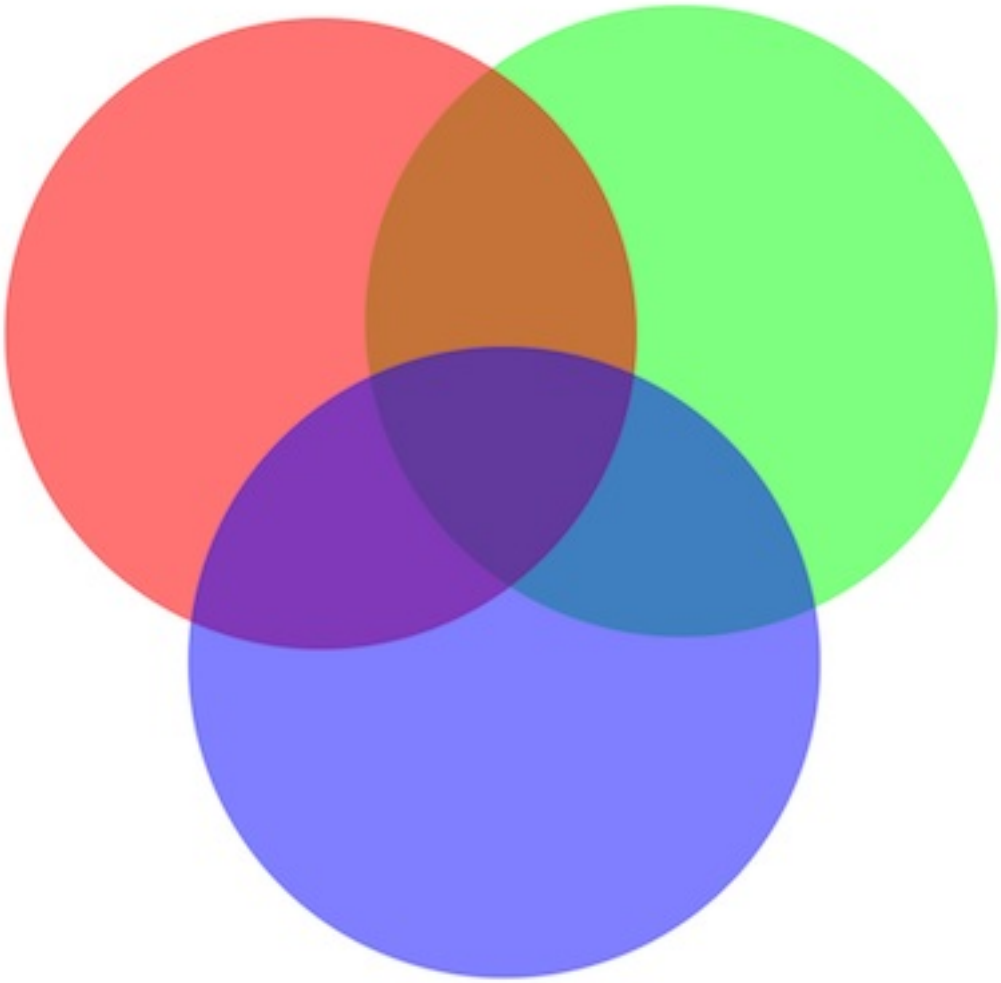
To encode a color in RGB you need three numbers between `0` and `255` or between `0.0` and `1.0`. If you use the floating point representation it will still get converted to the discrete representation from `0` to `255` where `0` means the light is not turned on for that color and `255` that it's turned on at the maximum level.

One this to keep in mind is that any display only shows colors in RGB.

RGBA Color Space

RGBA(stands for red green blue alpha) extends the **RGB** color model with an extra layer of information for opacity. This is the color we actually use in our software. But if the screen does not have an alpha LED, what does it do with that information?

The alpha channel describes how much light can pass trough. So if the opacity is set to `100%` the color will get shown on the screen exactly as the RGB part of the model describes it. If alpha is `0%` then the pixel is fully transparent.



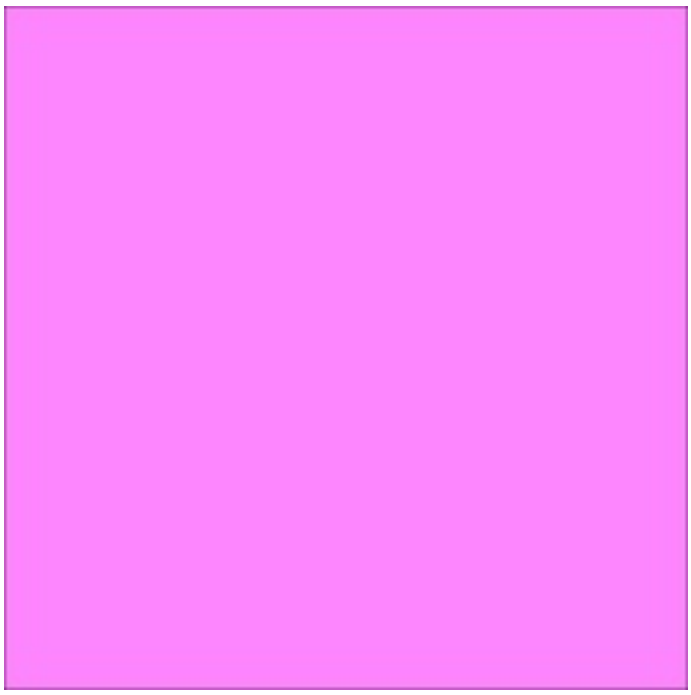
> 3 circles - each one has a primary color and `50%` opacity.

The inventors named alpha after the Greek letter in the classic linear interpolation formula `(1 - α) * A + α * B`.

Let's turn some lights on!

In iOS, colors are represented by the `UIColor` class. To create a new color you have to pass the values of the color components encoded as the floating points:

```
let color = UIColor(red: 1, green: 0.5, blue: 1, alpha: 1)
```

That will give this pinkish color: 

Making all values `1` will give white. And if you make the rgb part `0` and alpha `1` you get black.

How to combine colors?

A simple way to combine colors is by using the linear interpolation formula mentioned before:

```
func lerp(from a: CGFloat, to b: CGFloat, alpha: CGFloat) -> CGFloat
    return (1 - alpha) * a + alpha * b
}
```

`lerp` stands for linear interpolation and is the usual name this function is given. If you ever saw some code with a lot of lerping, now you know what that meant :)

If you want to learn more about linear interpolation, you can try the lessons from Khan Academy or if you are lazy like me you can watch this amazing talk by Steven Wittens - he has more cool videos on his website.

Let's lerp some colors:

The first thing we need to do is get the color components back from a `UIColor`. Unfortunately there are no `red`, `green`, `blue`, `alpha` properties:

```
extension UIColor {
    func components() ->
        (red:CGFloat, green:CGFloat,
            blue:CGFloat, alpha:CGFloat) {
        var r: CGFloat = 0
        var g: CGFloat = 0
        var b: CGFloat = 0
        var a: CGFloat = 0

        getRed(&r, green: &g, blue: &b, alpha: &a)

        return (r, g, b, a)
    }
}
```

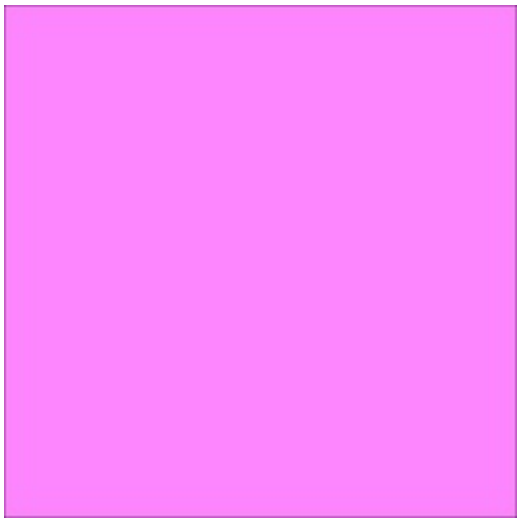
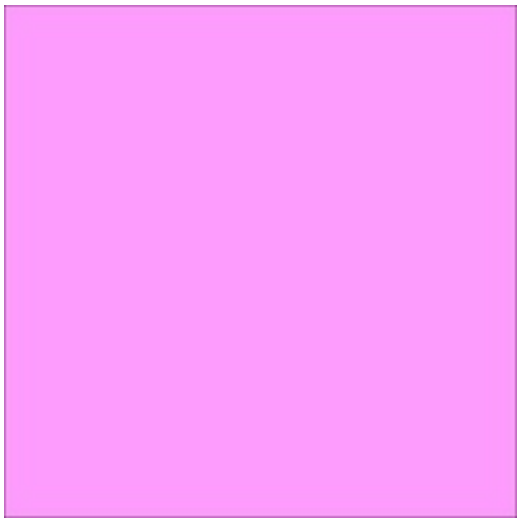
Now we can lerp colors:

```
extension UIColor {
  ...

    func combine(with color: UIColor, amount: CGFloat) -> UIColor {
        let fromComponents = components()

        let toComponents = color.components()

        let redAmount = lerp(from: fromComponents.red,
                             to: toComponents.red,
                          alpha: amount)
        let greenAmount = lerp(from: fromComponents.green,
                               to: toComponents.green,
                            alpha: amount)
        let blueAmount = lerp(from: fromComponents.blue,
                              to: toComponents.blue,
                           alpha: amount)
```

```
        let color = UIColor(red: redAmount,
                            green: greenAmount,
                            blue: blueAmount,
                            alpha: 1)

        return color
    }
}
```

To make a lighter shade of that pink we made before, we can call:

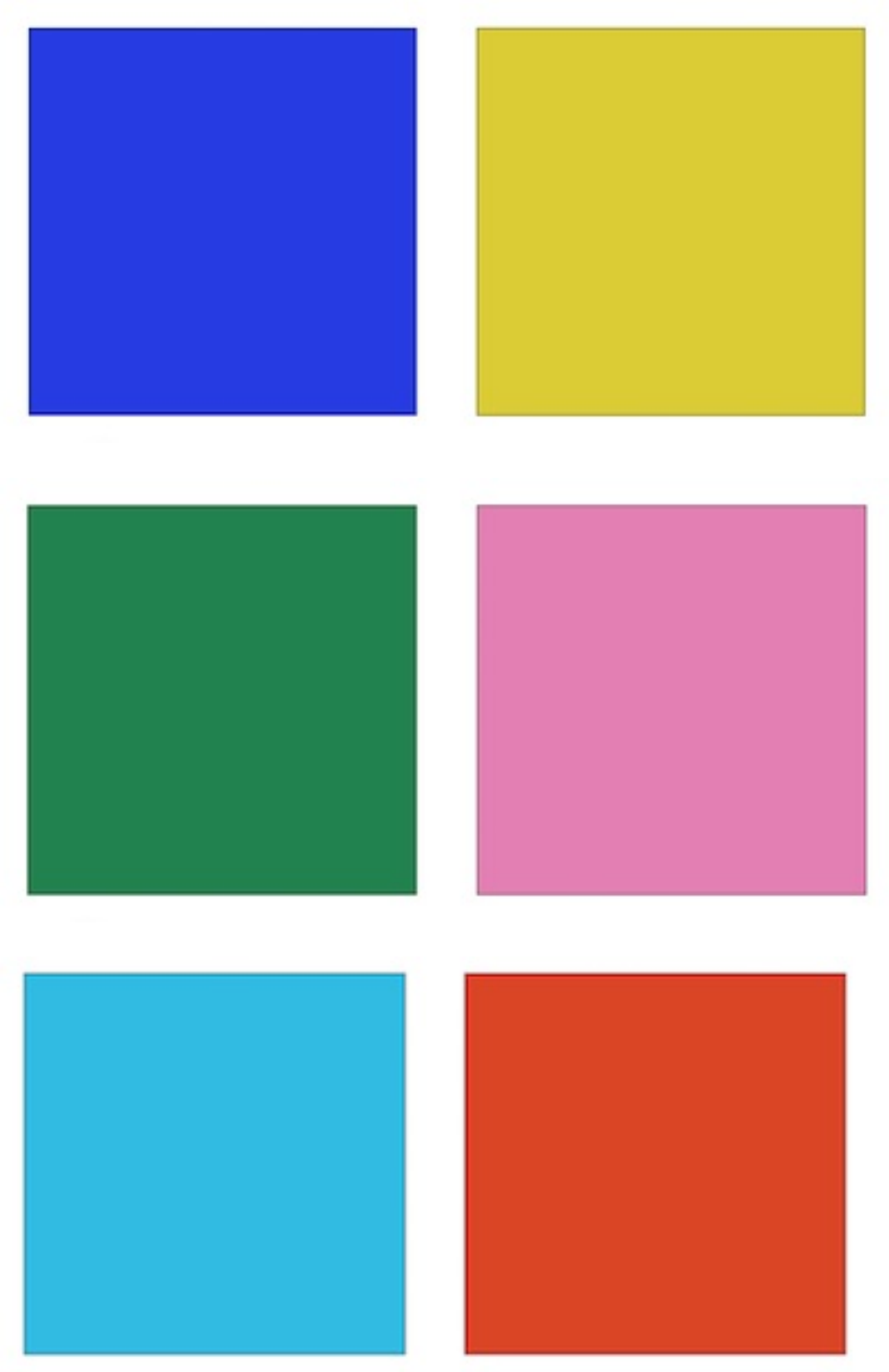
```
color.combine(with: .white, amount: 0.2)
```

Before	After
	

Complementary Colors

Complementary colors are pairs of colors which, when combined, cancel each other out. This means that when combined, they produce a grey-scale color like white or black. When placed next to each other, they create the strongest contrast for those particular two colors.

Complementary colors also look good together:

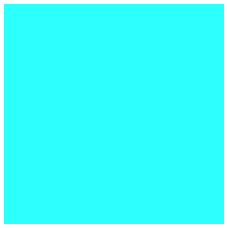


Contrast for colors means how much the color are different. To get the complementary of a color you need to reverse the intensity of each color channel:

```
extension UIColor {
    var complementary: UIColor {
        let (r, g, b, a) = components()

        return UIColor(red: 1 - r,
                       green: 1 - g,
                        blue: 1 - b,
                       alpha: a)
    }
}
```

You can use it like this:

```
let redComplementary = UIColor.red.complementary
```



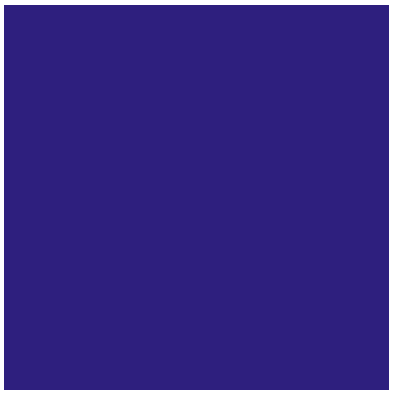
HSB Color Model

There is another way you can use to represent colors: the HSB Color Model. It stands for Hue, Saturation and Brightness.

Hue is a bit harder to explain without to much info, consider that it represents the wavelength of the color, or how far along the rainbow the color is. Brightness is the amount of light the image has - or how big are the numbers that represent it. Saturation measures the intensity of a color, as you desaturate a color it will become gray - the gray set by the brightness part of the model.

> from left to right colors vary on hue and from top to bottom on brightness

```
let color = UIColor(hue: 0.7,
             saturation: 0.8,
             brightness: 0.5,
                  alpha: 1.0)
```



Don't use `.red`

One of the most common mistakes among developers is sticking to the system's color palette. `.red` is `(255, 0, 0)` - that color does't exist

anyware in nature except on developer screens!

Stop using it in projects that other people will see. Use a set of colors you find online. There are thousands of sites providing free collections of pretty colors.

Zen bits

Make an extension named `Colors.swift` and use it in all your projects. People will have a better opinion about your work. Really, if there's one thing you take from this book, this would be a good one.

```
extension UIColor {
  class var myGreen: UIColor {
    return UIColor(red: 0.15, green: 0.68, blue: 0.38, alpha: 1)
  }

  class var myBlue: UIColor {
    return UIColor(red: 0.16, green: 0.50, blue: 0.73, alpha: 1)
  }

  class var myOrange: UIColor {
    return UIColor(red: 0.90, green: 0.49, blue: 0.13, alpha: 1)
  }

  class var myRed: UIColor {
    return UIColor(red: 0.75, green: 0.22, blue: 0.17, alpha: 1)
  }

  ...
}
```

Conclusion

It's easy and fun to create and combine colors, and with a bit of common

sense, you can make pretty stuff :)

Exercises

Color Gradient

Make a gradient from `.myRed` to `.myBlue` using only views.

```
func makeGradient() {
    let width = Int(view.frame.width)

    for i in 0...width {
        let frame = CGRect(x: CGFloat(i),
                           y: 0,
                       width: 1,
                      height: view.frame.height)

        let t = CGFloat(i) / view.frame.height

        let line = UIView(frame: frame)

        line.backgroundColor =
            UIColor.myRed.combine(with: .myBlue,
                                  amount: t)

        view.addSubview(line)
    }
}
```

More Colors!

Add `light`, `lighter`, `darker` and `dark` versions of the colors from `Colors.swift`:

- `light` should be 1 part white 4 parts the color
- `lighter` should be 1 part white 9 parts the color
- `dark` should be 1 part black 9 parts the color
- `darker` should be 1 part black 4 parts the color

```
extension UIColor {
    func light() -> UIColor {
        return combine(with: .white, amount: 0.2)
    }

    func lighter() -> UIColor {
        return combine(with: .white, amount: 0.1)
    }

    func dark() -> UIColor {
        return combine(with: .black, amount: 0.2)
    }

    func darker() -> UIColor {
        return combine(with: .black, amount: 0.1)
    }
}

extension UIColor {
    class var myLightGreen: UIColor {
        return myGreen.light()
    }

    class var myLighterGreen: UIColor {
        return myGreen.lighter()
    }

    class var myDarkGreen: UIColor {
        return myGreen.dark()
    }

    class var myDarkerGreen : UIColor {
        return myGreen.darker()
    }

    class var myLightBlue: UIColor {
```

```
        return myBlue.light()
    }

    class var myLighterBlue: UIColor {
        return myBlue.lighter()
    }

    class var myDarkBlue: UIColor {
        return myBlue.dark()
    }

    class var myDarkerBlue : UIColor {
        return myBlue.darker()
    }

    class var myLightOrange: UIColor {
        return myOrange.light()
    }

    class var myLighterOrange: UIColor {
        return myOrange.lighter()
    }

    class var myDarkOrange: UIColor {
        return myOrange.dark()
    }

    class var myDarkerOrange : UIColor {
        return myOrange.darker()
    }

    class var myLightRed: UIColor {
        return myRed.light()
    }

    class var myLighterRed: UIColor {
        return myRed.lighter()
    }

    class var myDarkRed: UIColor {
        return myRed.dark()
    }

    class var myDarkerRed : UIColor {
        return myRed.darker()
    }
}
```

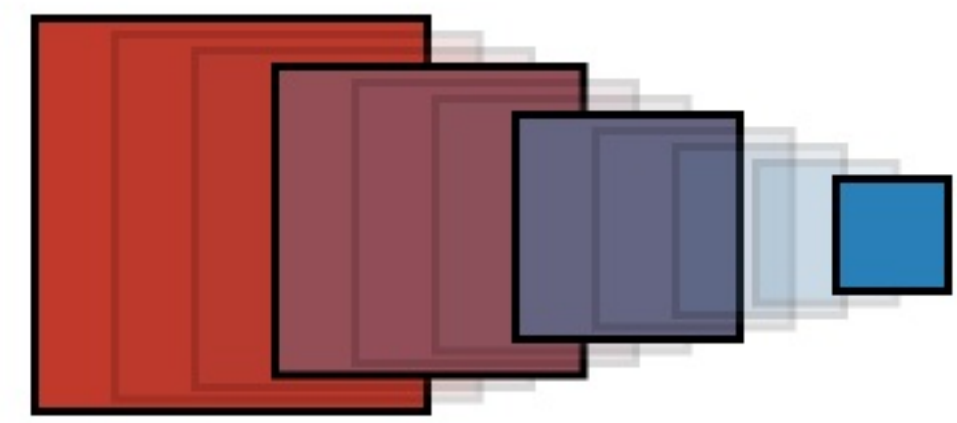
You should end up with these colors:

```
func showShades() {
    let width: CGFloat = 20
    let spacing: CGFloat = 5

    for (i, color) in UIColor.allShades.enumerated() {
        let frame = CGRect(x: spacing + CGFloat(i)*(width+spacing),
                           y: 0,
                           width: width,
                           height: view.frame.height)

        let line = UIView(frame: frame)

        line.backgroundColor = color

        view.addSubview(line)
    }
}
```

Animations

While changing view properties at runtime is useful, it doesn't look pretty. Most changes look better if they are animated from one state to another. `UIView` provides a very simple API for that. Do whatever changes you want to do to the view, but in a closure and set a duration. All changes on any animable property will be beautifully animated.

The `animate(withDuration:_:)` function will update the view and redraw it during the animation. You can see the different stages of an animation in the image below:



Implementing animation could not be any simpler. Most `UIView` properties are animable. This means that a change on that property can be smoothly transitioned from one state to another in an animation. This is the complete list of animable properties:

- frame
- bounds
- center
- alpha
- backgroundColor

- transform
- contentStretch

The simplest way to make an animation is by using the `animate(withDuration:animation:)` method. To make the animation above, we need to change the frame of the view and its background color from red to blue:

```
UIView.animate(withDuration: 3) {
    redView.backgroundColor = .blue
    redView.frame = newFrame
}
```

Customizing animations

The `UIView` animation API provides 4 methods:

- `animate(withDuration:animations:)`

  Animates the changes to one or more views over a period of time.

- `animate(withDuration:animations:completion:)`

  Animates the changes to one or more views over a period of time after which it calls the completion handler.

- `animate(withDuration:delay:options:animations:completion:)`

  Animates the changes to one or more views over a period of time using a set of `options` after which it calls the completion handler.

- `animate(withDuration:delay:dampingRatio:velocity:options:animations:completion:)`

  Performs `animations` using a timing curve described by the motion of

a spring. When `dampingRatio` is 1, the animation will smoothly decelerate to its final model values without oscillating. Damping ratios less than 1 will oscillate more and more before coming to a complete stop. You can use the initial spring `velocity` to specify how fast the object at the end of the simulated spring was moving before it was attached.

UIViewAnimationOptions

`.allowUserInteraction`

By default user interaction is turned off for views while they are animated. In case you want to animate a view and still have user interaction enabled use this option.

`.repeat`

Repeats the animation indefinitely. If you animate a change to the `backgroundColor` of a view, you will see the animation go from the original color to the new one and then jumping back to the original color and repeating the animation.

`.autoreverse`

Performs the animation forward one and then backwards. If you animate a change to the `backgroundColor` of a view, you will see the animation going from the original color to the new one and then animating back to the original color, then jumping to the new color. Because of this the `.autoreverse` option should be used combined with the `.repeat` option.

`.beginFromCurrentState`

If this key is not present, any animations affecting the current view(s) are allowed to finish before the new animation is started. If there are no animations on the current view(s) this key has no effect.

`.curveEaseIn`

Will cause the animation to begin slowly and then speed up in the end

`.curveEaseOut`

Will cause the animation to begin quickly and then slow down in the end

`.curveEaseInOut`

Will cause the animation to begin slowly and then speed up until the middle and then slow down in the end

> For more options and information check out the documentation.

Practice Time!

Download the starter project.

>  Starter Project

It has one controller with a red view inside:

`ViewController` has a `square` `@IBOutlet` to that view.

Perform the following animations in order:

- animate to the `backgroundColor` of `square` to `.myBlue`. The starter project contains the `Colors.swift` file.
- make square fade out then fade back in.
- make it two times as big but still centred in the same location
- rotate it 90 degrees
- make it spin forever

Solution

```
class ViewController: UIViewController {
    @IBOutlet var square: UIView!

    override func viewDidLoad() {
        super.viewDidLoad()

        let originalSize = square.frame.size
        let doubleSize =
            CGSize(width: originalSize.width * 2,
                   height: originalSize.height * 2)

        setSquare(color: .myBlue) {
         self.setSquare(alpha: 0) {
```

```
            self.setSquare(alpha: 1) {
             self.setSeqare(size: doubleSize) {
              self.setSeqare(size: originalSize) {
               self.rotateSquare() }}}}}
    }

    func setSquare(color: UIColor,
                   completion: @escaping () -> ()) {
        UIView.animate(withDuration: 0.4,
                       animations: {
            self.square.backgroundColor = color
          }, completion: { _ in completion() })
    }

    func setSquare(alpha: CGFloat,
                   completion: @escaping () -> ()) {
        UIView.animate(withDuration: 0.4,
                       animations: {
            self.square.alpha = alpha
        }, completion: { _ in completion() })
    }

    func setSeqare(size: CGSize,
                   completion: @escaping () -> ()) {
        UIView.animate(withDuration: 0.4,
                       animations: {
                let oldCenter = self.square.center
                self.square.frame.size = size
                self.square.center = oldCenter
        }, completion: { _ in completion() })
    }

    func rotateSquare() {
        UIView.animate(withDuration: 0.4,
                       delay: 0,
                     options: [.repeat],
                  animations: {
            self.square.transform =
                CGAffineTransform(rotationAngle:CGFloat(M_PI_2))
            }, completion: nil)
    }
}
```

Touch Events and Gesture Recognizers

When the iPhone came out it was magical! It was the first piece of technology that looked like it came from a science fiction movie. That tiny rectangle that you could touch changed the world we live in.

Tap! Swipe. Zoom. Scroll. Touch is at the heart of iOS. In this chapter we are going to take a brief overview at the how touch events are handled by iOS and how to detect gestures.

Touch interfaces are so intuitive you don't really need to learn that much to use them. My son is an one year old internet user. He picks his own cartoons or songs.

UIResponder

Because all you see are `UIViews`, it was an easy choice to let them handle touch events. You can respond to user interaction in three main ways: 1) by overriding `UIResponder` methods
2) by attaching gesture recognizers to a view
3) by taking a hitTest

`UIView` inherits from `UIResponder`, which is the base class that defines a responder object. A responder is an object that responds to user interaction events. `UIApplication` also inherits from `UIResponder`.

> This is how a responder object looks like when it's doing his job.

There are two general kinds of events: touch events and motion events. For now we are going to focus on the first ones.

Let's see what happens when you put your finger on the screen!

After all the hardware magic, `UIApplication` gets event objects, it passes them to its `UIWindow`, who then sends them to the view that has been touched and down the chain of responders.

That view is found by going over the view hierarchy and performing a hit test. `UIView` has a method called `point(inside:with:)` - the first parameter is a point and the second one is a touch event - the function should return `true` if the point is inside its bounds.

`UIWindow` calls the `hitTest` method on the view of the current view controller. It performs the hit test recursively on all subviews. The hit test returns the farthest view in the hierarchy that passed the hit test. To hit test a view you need to call the `point(inside:with:)` method with the current touch location and event. If that returns `true`, the view passed the hit test.

The hit test for a view will prefer to return a subview who passed the test instead of itself. So if you have a container view that has your the rest of your UI and controls, the events will get to them instead of the container.

> 
>
> **Quick tip**
>
> If you need to make an irregular shaped button you are going to need to override `point(inside:with:)`. An if you represent the button shape by a bezier path you can use `CGPathContainsPoint(_:)` to detect if the touch was inside of the path or not (source).

The chain of responders is a list of objects that inherit from `UIResponder`. The first one is `UIApplication`. The one you care the most about is `UIViewController`, your view controller.

For touch events, `UIResponder` has four main methods that get called so it can react to them:

- `touchesBegan(_:withEvent:)`
- `touchesMoved(_:withEvent:)`
- `touchesEnded(_:withEvent:)`
- `touchesCancelled(_:withEvent:)`

Mini Project

Enough theory. Let's have some fun with `UIResponder`!

We are going to make a small app that makes a view appears when you touch the screen.

Step 1: Create a new Xcode Project

Create a new Xcode project. Name it `Touches` and make sure the language is set to `Swift`.

Step 2: Override `touchesBegan`

Open `ViewController.swift` and override `touchesBegan`. You don't need to know the function definition by hand. Click below inside the `ViewController` class but not inside of a method and start typing `touchesBegan`, when the autocomplete option is touchesBegun hit enter. That should generate this code:

```
class ViewController: UIViewController {

    ...

    override func touchesBegan(_ touches: Set<UITouch>,
        with event: UIEvent?) {

    }
}
```

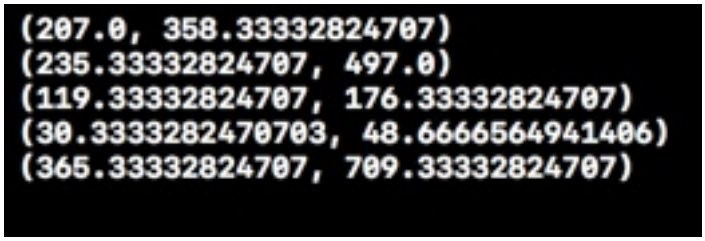
Every time you tap on the screen that method gets called. It receives two parameters: a set of touches and an `UIEvent` that contains the set of touches and more information about the event.

For each finger touching the screen, iOS generates a `UITouch` object. For most cases you only have one finger touching the screen. Let's make the app print the location of the tap.

First we need to get a touch object from `touches` and after that we can call the `location(in:)` method on it to get a `CGPoint` that represent the location of the touch relative to a view. We are going to pass the view controller's view `view`:

```
class ViewController: UIViewController {

    ...

    override func touchesBegan(_ touches: Set<UITouch>,
        with event: UIEvent?) {
        if let touch = touches.first {
            let location = touch.location(in: view)
            print(location)
        }
    }
}
```

Run the app and give it a few taps. The console should open and print the location of each tap:

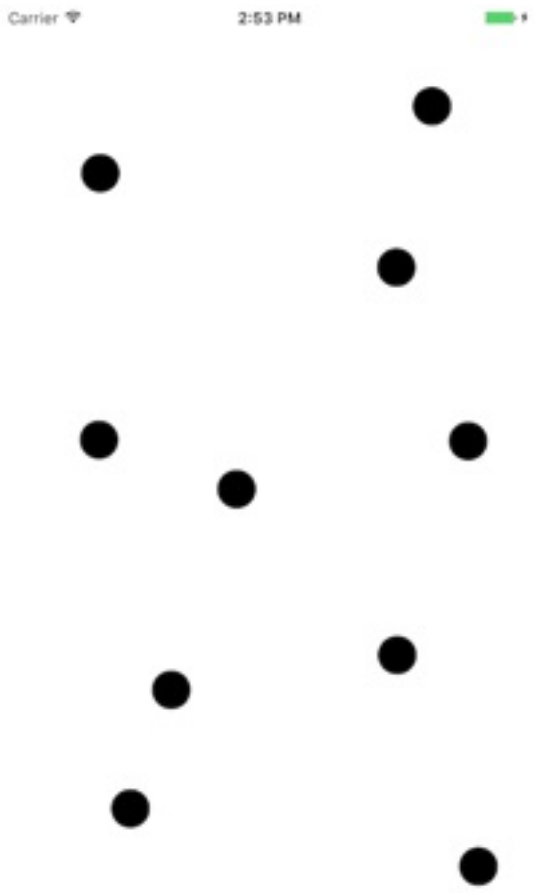


Step 3: Show a View

Now add a black circle with the center in the touch location:

```
class ViewController: UIViewController {

    ...

    override func touchesBegan(_ touches: Set<UITouch>,
        with event: UIEvent?) {
        if let touch = touches.first {
            let location = touch.location(in: view)

            // makes a frame for a view that is 30 by 30
            let touchViewFrame =
                CGRect(x: 0, y: 0, width: 30, height: 30)
            // creates a view with that frame
            let touchView = UIView(frame: touchViewFrame)

            // make the view black
            touchView.backgroundColor = .black

            // make the view look like a circle
            touchView.layer.cornerRadius = touchViewFrame.width / 2

            // move the views center in the location of the touch
            touchView.center = location

            // show the touchView
            view.addSubview(touchView)
        }
    }
}
```

Run the app. After a few taps it should look like this:



Step 4: Update the view

When you move your finger, the view doesn't follow your finger. Let's solve that. First thing you need to do is to keep a reference to `touchView` in the viewController.

```
class ViewController: UIViewController {
    var touchView: UIView!

    ...
}
```

And remove the `let` before `tocuhView` when you create it in `touchesBegan`:

```
class ViewController: UIViewController {
```

```
    ...

    override func touchesBegan(_ touches: Set<UITouch>,
        with event: UIEvent?) {
        if let touch = touches.first {
            ...
            let touchView = UIView(frame: touchViewFrame)
            ...
        }
    }
}
```

Override `touchesMoved` - as a start let's copy paste the code from above. That will create a new circle when the touch moves:

Now the app behaves like a crude drawing app. This is not a practical example because it will shortly add hundreds of views to the screen and that will make the interface slow (and finally it will break). Also, it doesn't handle the case when you move your finger fast on the screen, adding only

intermediate dots instead of a continuous line.

Remove the copied code after you get `location` from the touch event, except for the line that sets the `center` of `touchView`:

```
class ViewController: UIViewController {

    ...

    override func touchesBegan(_ touches: Set<UITouch>,
        with event: UIEvent?) {
        if let touch = touches.first {
            let location = touch.location(in: view)

            touchView.center = location
        }
    }
}
```

If you run the project now, the black circle will follow you finger.

Step 5: Remove the touch view

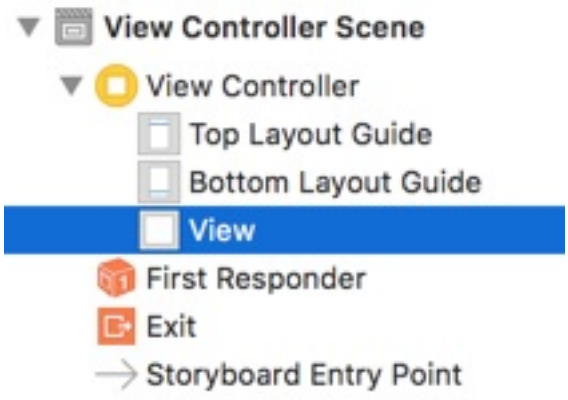
To remove the `touchView` when the touch finishes, you have to override two methods: `touchedEnded` for the case when the user lift his finger, and `touchesCanceled` - this gets called when someone from chain of responders cancel a touch event - this might happen when you animate a view that has an active touch on it, or when you have a gesture recognizer attached to a view.

```
class ViewController: UIViewController {
    ...

    override func touchesCancelled(_ touches: Set<UITouch>,
        with event: UIEvent?) {
        touchView.removeFromSuperview()
    }

    override func touchesEnded(_ touches: Set<UITouch>,
```

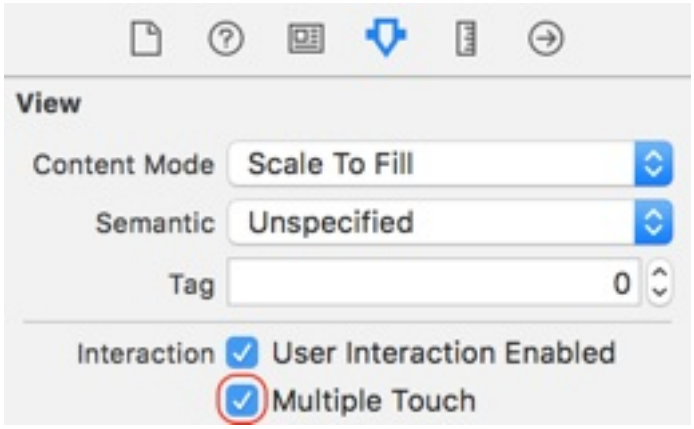
```
            with event: UIEvent?) {
            touchView.removeFromSuperview()
        }
    }
```

Step 6: Handle multiple touches

First thing you need to do for that is to enable multi-touch on `view`. Open `Main.storyboard` and select the view of the view controller:



In the `Attributes Inspector` enable `Multiple Touch`:



Notice that there's an option to enable or disable user interaction on a view. If `isUserInteractionEnabled` is set to `false`, the view will not receive any touch events. Most system's controls/views have this set to `true`, but some have it set to `false` by default, for example images, so you won't get any touch events on them until you set it to `true`.

Now if you touch the screen with multiple finger you will receive more than one even in `touches`.

We need a variable number of "properties" to hold all the black circles. Naming them `touchView1`, `touchView2`, ... isn't a good idea.

If you look at the definition of `Set`, you will notice that it only accepts `Hashable` elements:

```
public struct Set<Element : Hashable> : ... {
    ...
}
```

That means you can get the `hashValue` of a touch and use it as a key in a `Dictionary` that will hold the touch views.

So remove the `touchView` property and add a new one called `touchViews`:

```
class ViewController: UIViewController {
    var touchViews: [Int:UIView] = [:]

    ...
}
```

Now that we have more than one touch instead of getting the first from the set we are going to iterate over the set. Good news, only two lines of code will change in each `UIResponder` method.

Lets start with `touchesBegan`. The first one will be replace with a `for in` statement:

```
override func touchesBegan(_ touches: Set<UITouch>,
        with event: UIEvent?) {
        for touch in touches {
            ...
        }
```

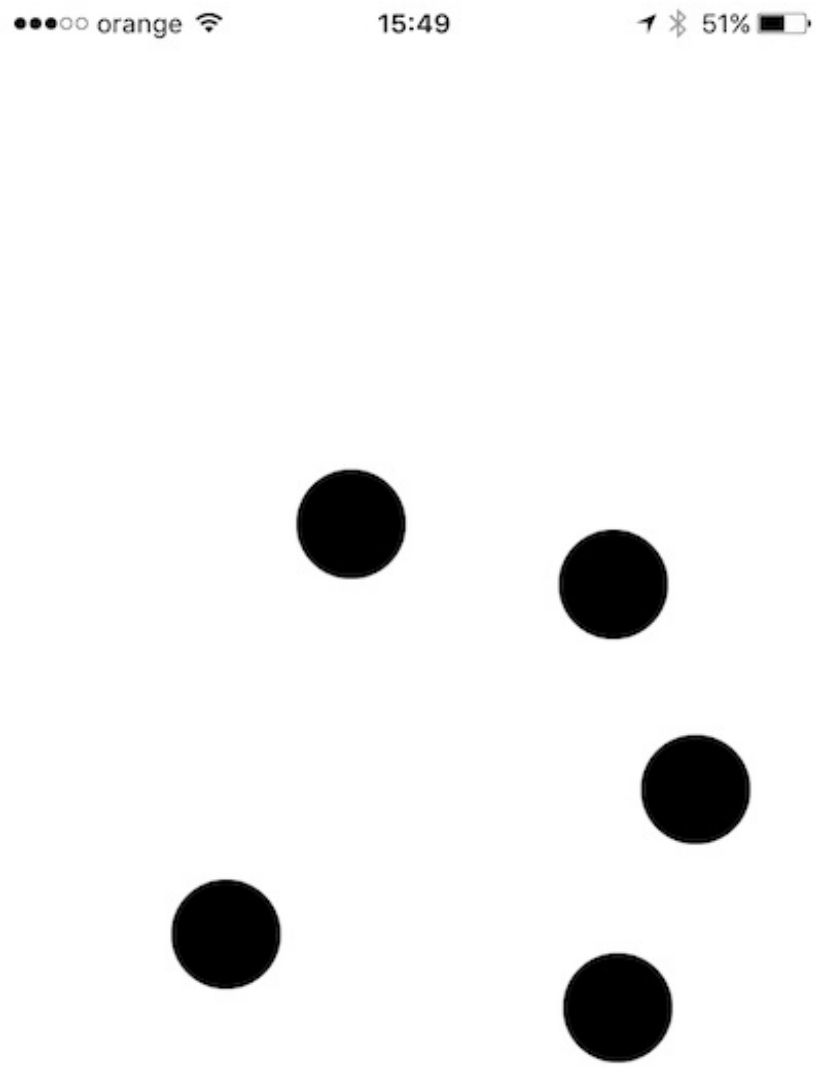
```
}
```

The next thing we have to change is how we create and store the touch view.

Instead of setting the property to a new `UIView` instance, we are going to put the `let` back in front of `touchView`. Then we are going to save the `hashValue` of the current `touch` and store `touchView` in `touchViews`. In the end you should end up with something like this:

```
class ViewController: UIViewController {
    ...

    override func touchesBegan(_ touches: Set<UITouch>,
        with event: UIEvent?) {
        for touch in touches {
            let location = touch.location(in: view)
            let touchViewFrame =
                CGRect(x: 0, y: 0, width: 30, height: 30)

            let hash = touch.hashValue
            let touchView = UIView(frame: touchViewFrame)

            touchView.layer.cornerRadius = touchViewFrame.width / 2
            touchView.center = location
            touchView.backgroundColor = .black

            view.addSubview(touchView)
            touchViews[hash] = touchView
        }
    }
}
```

Comment out the code from the other methods and try it out:



To make them update we need to change the way we get the touch view:

```
class ViewController: UIViewController {
    ...

    override func touchesMoved(_ touches: Set<UITouch>,
        with event: UIEvent?) {
       for touch in touches {
            let location = touch.location(in: view)
```

```
            let hash = touch.hashValue
            if let touchView = touchViews[hash] {
                touchView.center = location
            }
        }
    }
}
```

And in `touchesEnded` and `touchesCancelled` you need to get the respective views from the touch events, not just remove the last one added:

```
class ViewController: UIViewController {
    ...

    override func touchesCancelled(_ touches: Set<UITouch>,
        with event: UIEvent?) {
        for touch in touches {
            let hash = touch.hashValue
            if let touchView = touchViews[hash] {
                touchView.removeFromSuperview()
            }
        }
    }

    override func touchesEnded(_ touches: Set<UITouch>,
        with event: UIEvent?) {
        for touch in touches {
            let hash = touch.hashValue
            if let touchView = touchViews[hash] {
                touchView.removeFromSuperview()
            }
        }
    }
}
```

Although you probably won't use these methods to often, or ever, it's important that you at least know that they exist.

Gesture Recognizers

Handling more complicated interactions just by observing the different touch locations seems almost impossible, thankfully this is already implemented by Apple and available in `UIKit`. As you've guessed, I'm talking about Gesture Recognizers.

The standard library includes gestures recognisers for the following gestures:

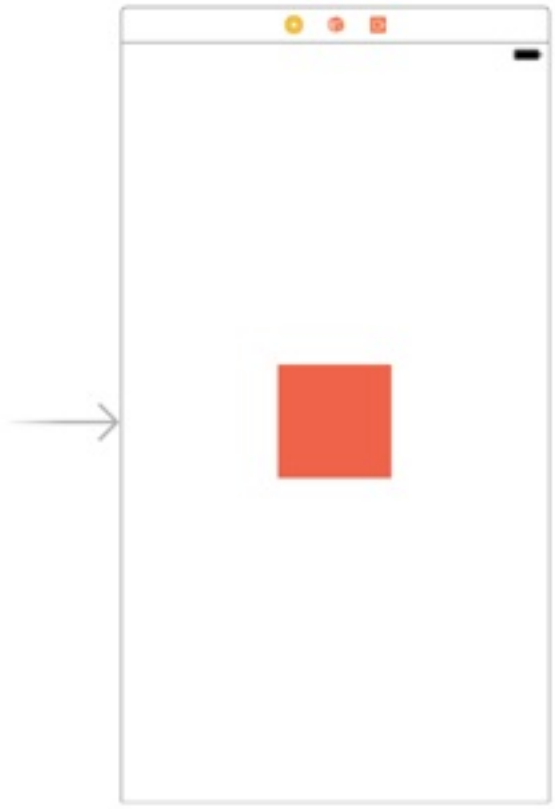
- Tap - can detect taps - you can customise the number of taps or fingers needed to activate the gesture
- Pan - pan simply means moving across the view
- Swipe - similar to the pan gesture, except a swipe is a swift gesture.
- Pinch - this is the gesture used to zoom in/out
- Long Press - detects a long press - you can customize the minimum duration and the number of fingers required
- Rotation - detects a rotation - this is a gesture with two fingers - you can extract the angle and speed of the gesture
- Screen Edge - detects a swipe from the edge of the screen

Adding a Gesture Recognizer with Interface Builder

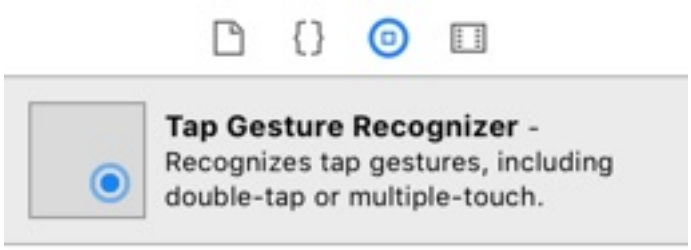
If you want to follow along, download the starter project:

>  Starter Project

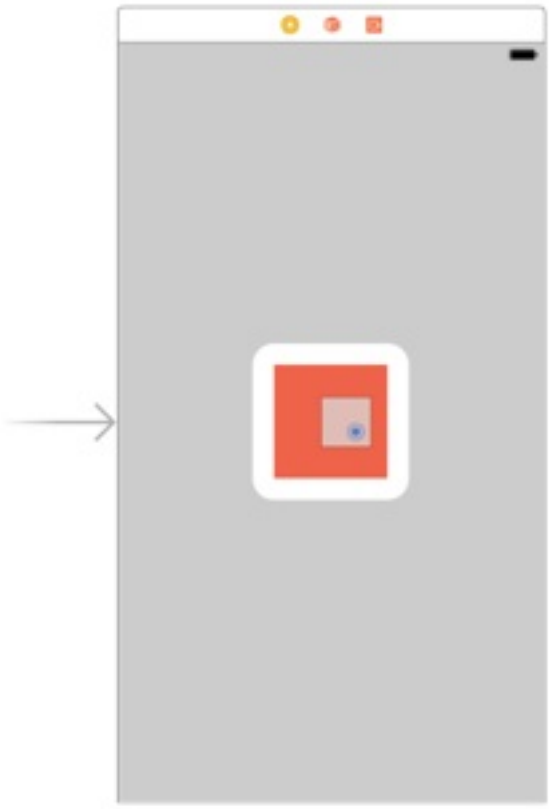
The first thing we need is the view to react to touches:



Search for Tap Gesture Recognizer in the `Components Library` pane:

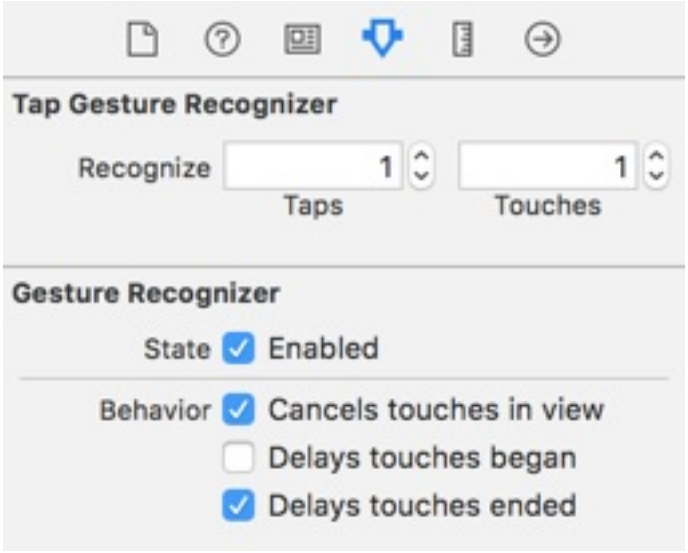


Drag and drop a `Tap Gesture Recognizer` on top of the view:



This will add a tap gesture recognizer in the object hierachy:

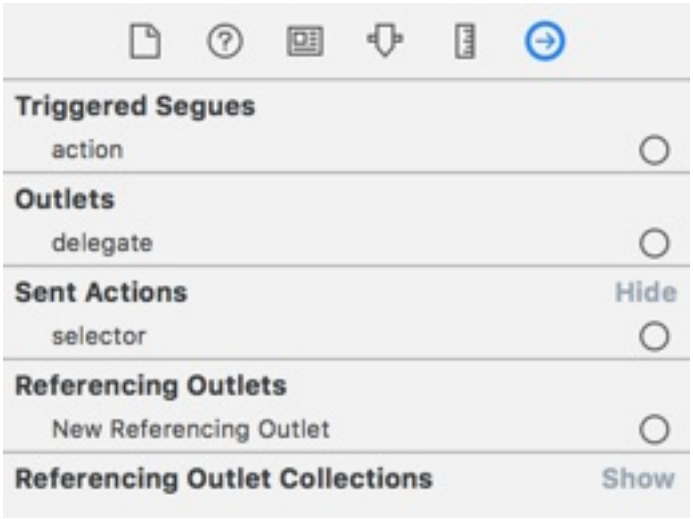
Click on it and open the `Attributes Inspector`. There you will see the customisation points of the tap gesture recognizer:



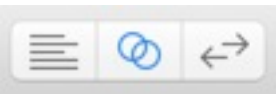
The most used customization point is the minimum number of taps and touches require to activate it. A double tap gesture recognizer is a tap gesture recognizer with `numberOfTapsRequired` property set to `2`.

Another thing you can do is to disable the gesture recognizer and enable it programatically later in your code.

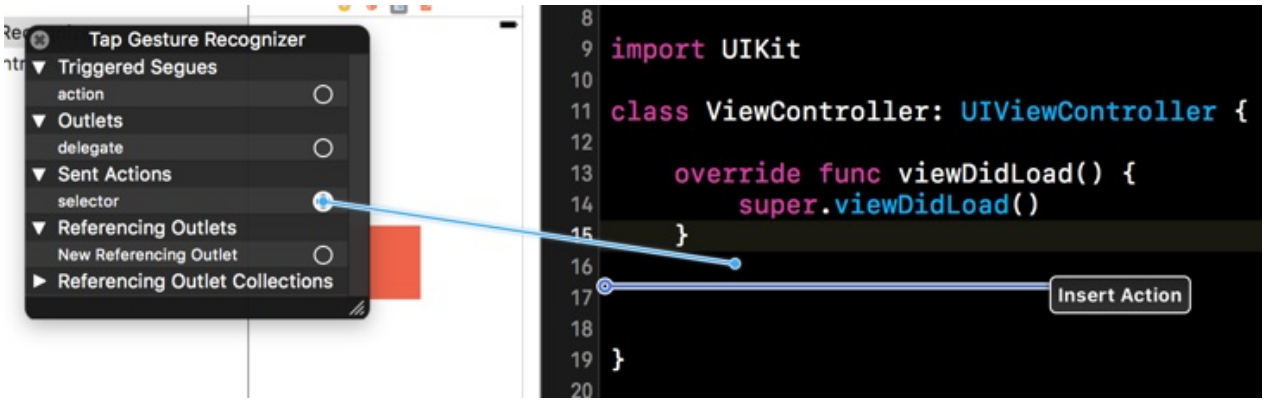
The gesture recognizer is all setup at this point. The only problem is that you don't get notified when a tap occurred. Open the `Connections Inspector`:



Open the assistant editor by pressing the *"two circles"* button



Connect the `selector` connection with a point inside `ViewController` after `viewDidLoad`. Another way you could do this is by right clicking on the `Tap Gesture Recognizer` object and then proceed to do the same.



That should open a popover menu:

Call it `didTap` and change the type from `Any` to `UITapGestureRecognizer` before clicking on `Connect`.

This generates a method that will be called after a each tap event:

```
class ViewController: UIViewController {
    ...

    @IBAction func didTap(_ sender: UITapGestureRecognizer) {
        print("did tap")
    }
}
```

>  **Zen bits**
>
> I've added a print statement to test if the connection worked. Always do this when adding a new connection. Test them, then write the rest of the code. Remember that a bug gets harder to solve the more time passes since it was created.

The process is pretty much the same for all gesture recognizers. But I didn't mention a thing about gesture recognizers so far; they have a `state`.

Some gestures are more complicated than a tap. Let's take for example the pan gesture: when you put a finger on the screen the gesture is not activated, but still possible, when you move your finger a bit the gesture is recognized or it started, when you move it a bit more the gesture changes its location and after you take you finger off the screen the gesture ended. If you followed that you can almost guess the main states of a gesture recognizer:

- `.possible` - when the gesture recognizer received events, the gesture is still possible but not yet recognized
- `.began` - when the gesture is recognized
- `.changed` - when it received more touched after it's recognized and still valid
- `.ended` - when the gesture stopped receiving events
- `.canceled` - when then touches responsible from triggering the gesture get cancelled
- `.failed` - when the gesture was not recognized; after this the gesture state will be reset to `.possible`.

The `selector` you set in `Interface Builder` will get called every time the gesture recognizer state changes. That mans that, for example, if you use a swipe gesture recognizer you will only want to handle the case where the

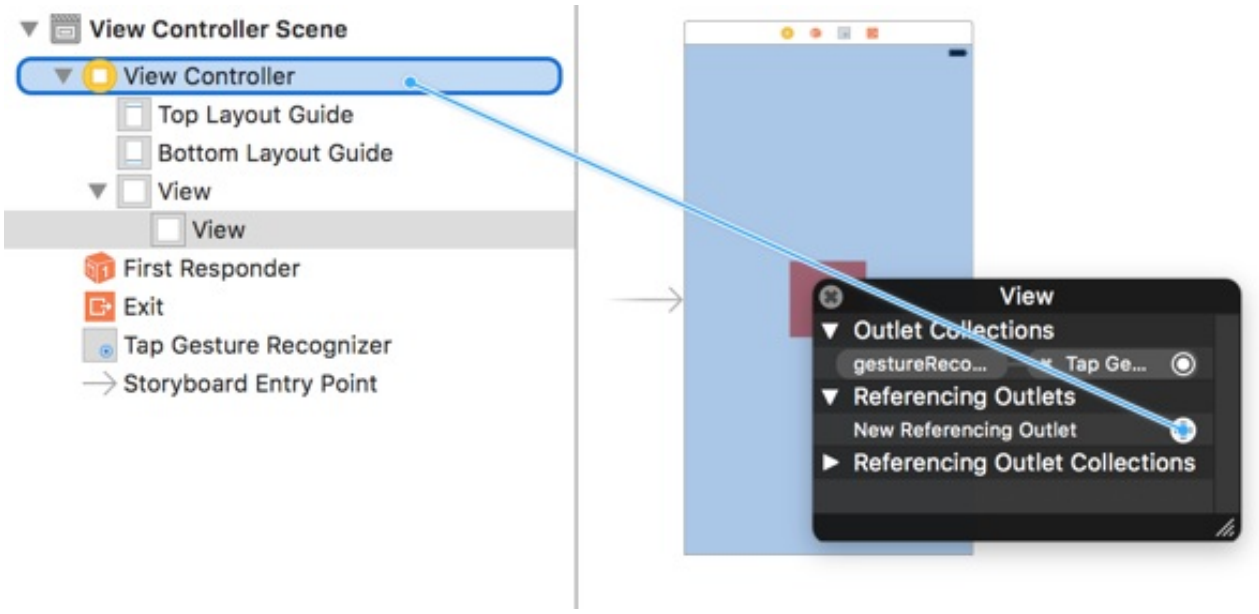
state is `.ended`. And for a tap gesture recognizer, a single touch event is enough to set the state to `.ended`, so that why you only get one call to the `didTap(_:)` method.

Lets continue expliring different recognizers! Add a referencing outlet for the red view:

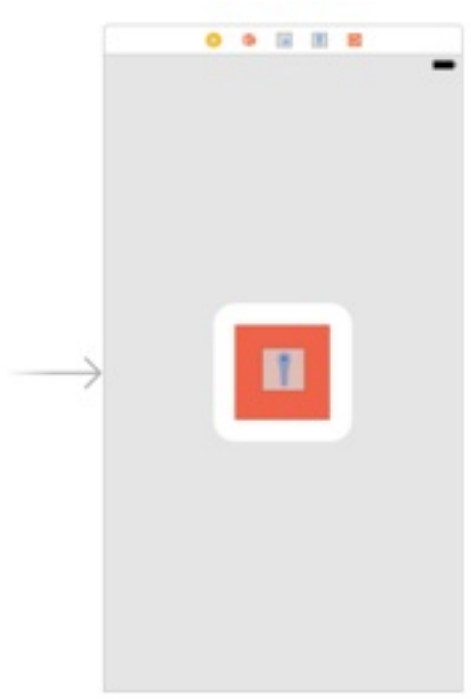
In add a `redView` property in `ViewController.swift` :

```
class ViewController: UIViewController {
    @IBOutlet var redView: UIView!

    ...
}
```

Connect in from `Interface Builder` :



Search for `Pan Gesture Recognizer` in the `Components Library` and drag one on `redView`.



Open the `Assitand Editor` and create the `didPan` method:

```
class ViewController: UIViewController {
    ...

   @IBAction func didPan(_ sender: UIPanGestureRecognizer) {
       print("did pan..")
   }
}
```

Run the app. You should see multiple `did pan..` message when moving over the view.

Now for a bit of magic. Lets translate the view to follow you finger:

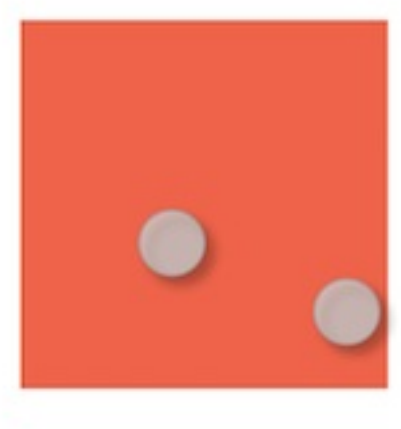
```
class ViewController: UIViewController {
    ...

    @IBAction func didPan(_ sender: UIPanGestureRecognizer) {
        // gets the translation from the recognizer
        let translation = sender.translation(in: redView)

        // moved the view
        redView.frame.origin.x += translation.x
```

```
            redView.frame.origin.y += translation.y

            // resets the offset in the recognizer
            sender.setTranslation(.zero, in: redView)
        }
    }
```

And now to handle scaling the view with the help of a pinch gesture recognizer:

```
class ViewController: UIViewController {
    ...

    @IBAction func didPinch(_ sender: UIPinchGestureRecognizer) {
        // get scale
        let scale = sender.scale

        // remember where the view was centered
        let oldCenter = redView.center

        // scale the view
        redView.frame.size.width *= scale
        redView.frame.size.height *= scale
        // that change the center of the view

        // move it back to where it was
        redView.center = oldCenter

        // reset scale
        sender.scale = 1.0
    }
}
```

If you want to test this in the simulator you will need to hold `Option` (`⌥`) to show two cursors:



If you move your mouse the points will move symmetrical to the point between them. It's a bit hard to control but you get used to it. So `Option` (`⌥`) + moving is a pinch or rotate gesture.

If you want to move both points to another part of the screen you need to hold both `Option` (`⌥`) and `Shift` (`⇧`) and then move the cursor on the screen. Don't let go of `Option` (`⌥`) because that will reset the position.

Adding a Gesture Recognizer from code

Just like views have `subviews`, they also have `gestureRecognizers`. To add a gesture recognizer to a view, you need to create it and then call the `addGestureRecognizer(_:)` method on the view.

Let's continue our project by adding the rotation gesture recognizer. When doing this in code, the steps are a bit reversed. First you need to create the callback:

```
class ViewController: UIViewController {
    ...
```

```
    func didRotate(_ sender: UIRotationGestureRecognizer) {
        print("did rotate")
    }
}
```

Then create an `UIRotationGestureRecognizer` instance in `viewDidLoad` and attach it to `redView`:

```
class ViewController: UIViewController {
    ...

    override func viewDidLoad() {
        super.viewDidLoad()

        let rotationGestureRecognizer =
            UIRotationGestureRecognizer(target: self,
                    action: #selector(didRotate(_:)))

        redView.addGestureRecognizer(rotationGestureRecognizer)
    }
}
```

When creating a gesture recognizer, you need to provide two arguments: an object that will be notified when the recognizer changes its state and a `selector` (method) that will be called on that object in order to notify it.

Run the project. See if you can trigger the print call.

Let's change the color of the view when we make a rotation gesture:

```
class ViewController: UIViewController {
    ...

    func didRotate(_ sender: UIRotationGestureRecognizer) {
        // get the rotation angle in radians
        let angle = sender.rotation

        // transform it in to a value that
        // ranges from -1 to 1
        let sine = sin(angle)
```

```
        // change from -1..1 to 0..1
        let value = (sine + 1) / 2

        // create a color with that value
        let color = UIColor(hue: value,
                            saturation: 0.7,
                            brightness: 0.7,
                            alpha: 1.0)

        redView.backgroundColor = color
    }
}
```

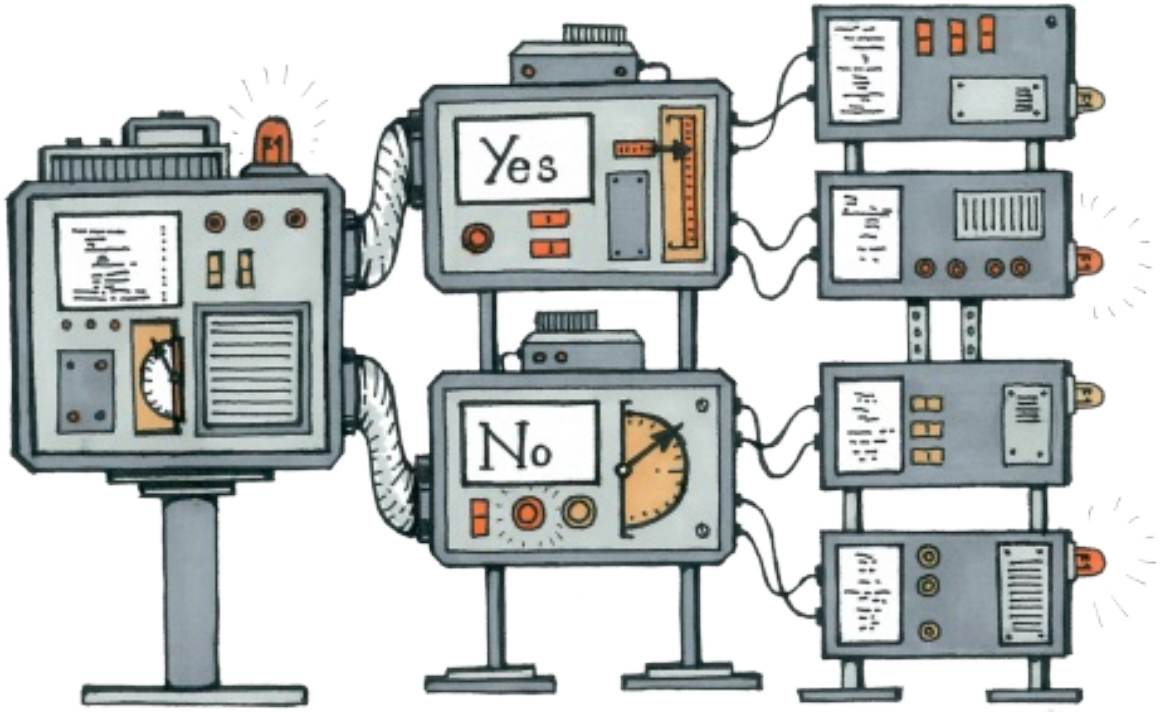
Practice Time!

Make a custom view called `TouchableView` that has the behaviour for move, scale and change color. Add a touchable view on the screen on a double tap. Change the shape from a square to a circle on double tap.

Draw something!

> Solution

Simple iOS Controls



A big part of making an app is the process of designing and implementing the user interface. In this chapter we are going to look at the most common UI components from iOS and the design patterns used by them. They are very customisable, so take your time to learn every know they have. Knowing what you can do with them will save you a lot of time and headaches.

When needed, we are also going to take a short look at the Objective-C roots of iOS and the system classes/interfaces needed for a specific task.

Start a new project and open the storyboard. Search for the components mentioned in this chapter and play around with them. For each component there are a few challenges that should help you practice using and customising UI controls.

Because almost all customization points are accessible from Interface Builder the customization part of each component will focus on how you achieve that effect in code. Take the time to explore what each option from the `Attributes Inspector` does for all controls.

iOS Human Interface Guidelines

Designing and implementing user interfaces is hard! Not everybody is an artist, but there are a few tricks you could learn that will dramatically improve your results - things like consistency, less is more and never make a button smaller than a 30 point square... My favorite collections of that kind of tricks is Apple's iOS Human Interface Guidelines. It outlines how apps should be designed for iOS. HIG is the common way of referring to it.

It has all you ever wanted to know about what makes a good app (or a good user experience). I highly recommend that you read it and refer to it when taking design decisions with your team. Good app developers also know a bit about design. Remember that people will judge your app by the way it looks first and only after on what it can do.

There is a lot to read in the HIG. But for now start with this page UIComponents: Controls. It shows most of the iOS controls and explains when and how to use them. If you didn't design an app or website before, I bet you didn't think like that about labels or buttons.

Interface Builder vs Code

You can implement UIs with Interface Builder or directly from code. Most apps use a mix of both and some use only programatic interfaces. I personally don't think that using only code is a good idea. Interface Builder has evolved a lot over the years, therefore, with the addition of `IBDesignable` you can create new views and controls and extend Interface Builder. So you can use code for interfaces and you can also use them in Interface Builder if you put in a little bit of effort.

It takes a few seconds to understand a screen with 20 components if you look at a storyboard, on the other hand if you would see the code for that screen you would need a few minutes to figure out which line is responsible for which part is which - even if it's your code!

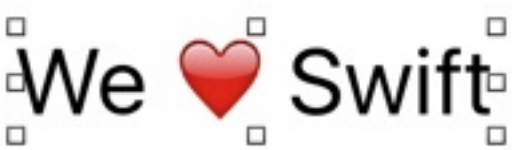
> 
> **Zen Bits**
>
> *Your job is to solve problems by writing the minimum amount of code. Always consider the case when you write 0!*

UILabel

Labels show small amounts of text, either to name a part of the screen, or to display information. Almost everywhere you see only few words on a screen you can bet that it's a label, it may also be included in other controls, for example: the title of a button is a label.

Code example

```
let label = UILabel(frame:
    CGRect(x: 10, y: 10, width: 200, height: 32))
// changing the font
label.font = .systemFont(ofSize: 30)
// changing the text
label.text = "We ❤ Swift"
```

The results should look like this: 

Customization

By changing a few properties, you can achieve all kinds of effects:

> **Note:** Because all iOS controls inherit from `UIView` you can customise any `UIView` property on all components.

Font

Fonts are loaded using the `UIFont` class. Each font has an unique identifier. To create a font object you need to set the name of the font and its size:

```
let hugeLightFont =
    UIFont (name: "HelveticaNeue-UltraLight",
            size: 30)
```

> 
> **Quick tip**
>
> You can find a complete visual list of all the system fonts on iosfonts.com.

To change the font of a label, set the `font` property:

```
label.font = hugeLightFont
```

System font

As an alternative, you can use the system font:

```
let font = UIFont.systemFont(ofSize: 14)
```

System font refers to the default font used for standard interface items. So when you add a label, its font is `System Font - 15`. Since iOS 9, the system font is San Francisco and before that, it was Helvetica Neue.

Text Color

Similar to the way you can customise the background color by setting the `backgroundColor` property, you can change the color of the text by setting the `textColor` property:

```
title.textColor = .white
```

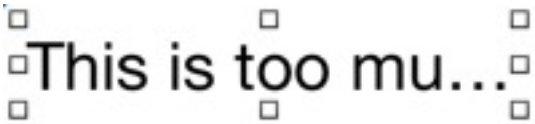
Line break mode

The line break mode option sets the way a label handles the case when it is trying to show more text than fits inside.

You can set the `lineBreakMode` to one of these options:

- `.byTruncatingTail` - this is the default option. The line is displayed so that the beginning fits in the container and the missing text at the end of the line is indicated by an ellipsis glyph `…`:

```
label.lineBreakMode = .byTruncatingTail
```



- `.byTruncatingMiddle` - the line is displayed so that the beginning and end fit in the container and the missing text in the middle is indicated by an ellipsis glyph:

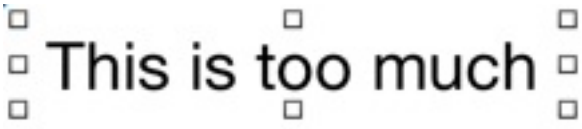
```
label.lineBreakMode = .byTruncatingMiddle
```

- `.byTruncatingHead` - the line is displayed so that the end fits in the container and the missing text at the beginning of the line is indicated by an ellipsis glyph:

```
label.lineBreakMode = .byTruncatingHead
```

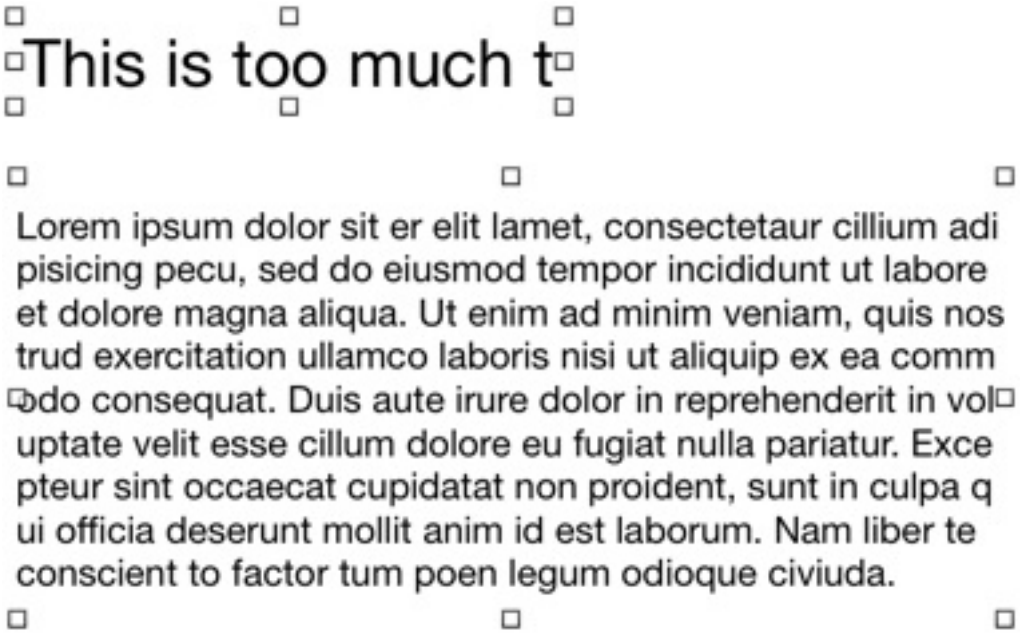
- `.byWordWrapping` - wrapping occurs at word boundaries, unless the word itself doesn't fit into a single line:

```
label.lineBreakMode = .byWordWrapping
```



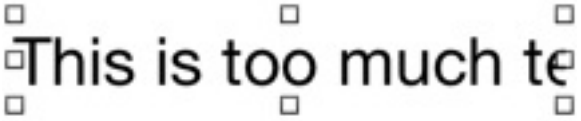
- `.byCharWrapping` - wrapping occurs before the first character that doesn't fit:

```
label.lineBreakMode = .byCharWrapping
```



- `.byClipping` - text is not drawn past the edge of the label:

```
label.lineBreakMode = .byClipping
```



Number of lines

By default labels show 1 line of text. You can make labels display text on multiple lines, but only do so if you are still using it as a label. Otherwise use a text view.


```
label.numberOfLines = 2
```

If you set `numberOfLines` to `0` the label will display as many lines as the `text` has but not more than it can fit in its size.

Text Alignment

There are five options for text alignment, described by the `NSTextAlignment` enum:

- `.left` : Align text along the left edge. This is the default text alignment option.

  ```
  label.textAlignment = .left
  ```

- `.center` : Align text equally along both sides of the center line.

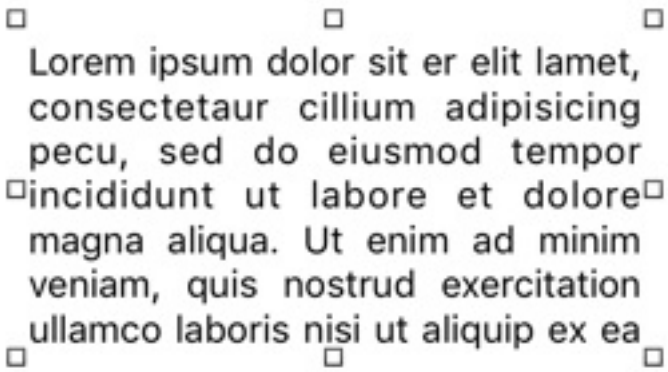
  ```
  label.textAlignment = .center
  ```

- `.right` : Align text along the right edge.

```
label.textAlignment = .right
```

- `.justified` : Fully justify the text so that the last line in a paragraph is natural aligned. This is only used for label that have more than one line of text.

```
label.textAlignment = .justified
```



- `.natural` : Use the default alignment associated with the current localization of the app. The default alignment for left-to-right scripts is `.left`, and the default alignment for right-to-left scripts is `.right`.

Attributed Strings

An attributed string is a special kind of string that can have different attributes set for parts of it. For example, a part of the text can have a different color or font size.

In the example above the label has a title and a subtitle. The title has a bigger font size than the subtitle. The heart is red and has a font that doesn't replace the unicode characters with one of Apple's emojis.

First, let's create an attributed string with the title and subtitle in it. To create an attributed string we need a "normal" one.

```
let title = "iOS Spellbook"
let subtitle = "We ❤ Swift"

let string = "\(title)\n\(subtitle)" as NSString
let attributedString =
    NSMutableAttributedString(string: string as String)

label.attributedText = attributedString
```

> **Obs:** `\n` represents a newline character. Inside strings you can write special characters by using the escape character `\`. Other examples include: `\t` for tab, `\"` for double quotes and `\\` for backslash `\`.

> **Obs:** `string` has a `NSString` reference to the text of the label. We are going to use it to make `rangeOfString` queries so we can decorate different parts of the text.

Now we can add different attributes for different parts of the string. First of all, we make the title portion of the text bigger:

```
...

// title
let titleAttributes = [
    NSFontAttributeName: UIFont(name: "HelveticaNeue",
                                size: 32)!]

let titleRange = string.range(of: title)

attributedString.addAttributes(titleAttributes,
                range: titleRange)

// you must add the atributes before you set the
// attributedText property because it makes a copy
label.attributedText = attributedString
```

And the subtitle font and size:

```
let subtitleAttributes = [
    NSFontAttributeName :
        UIFont(name: "HelveticaNeue", size: 20)!
]

let subtitleRange = string.range(of: subtitle)

attributedString.addAttributes(subtitleAttributes,
```

And finally, let's make the heart flat by changing the font to a dingbat and the color red.

```
let heartColor = UIColor(red: 0.91, green: 0.30, blue: 0.24, alpha:1)
let heartAttributes = [NSForegroundColorAttributeName: heartColor,
     NSFontAttributeName: UIFont(name: "ZapfDingbatsITC", size: 20)!]
let heartRange = string.range(of: "❤")
attributedString.addAttributes(heartAttributes, range: heartRange)
```

You can find the complete list of attributes in Apple's reference.

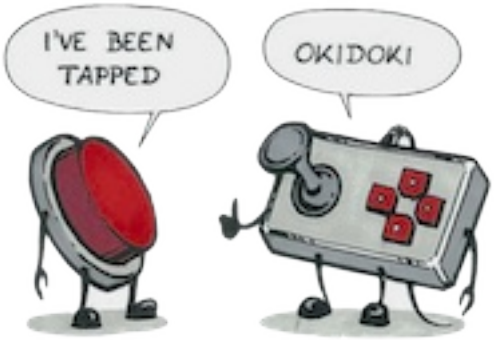
UIButton

Buttons are awesome! They make things happen when you need them to.

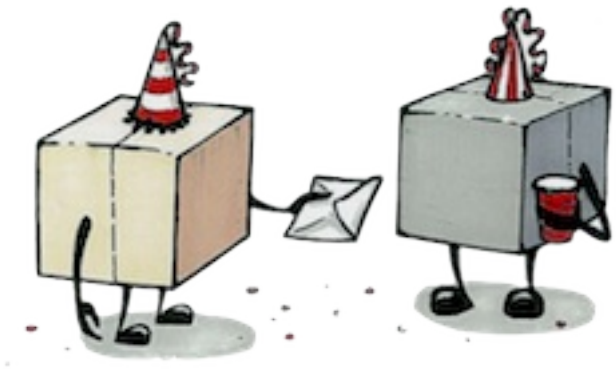
The button is also the simplest input you will use. If you had to use only buttons and labels to make apps, you could still make a lot of them. After we learn a bit about how buttons work and how to use them, we are going to make a few simple games to practice.

Target-Action

To do their magic, buttons implement a design pattern named *Target-Action*. I encourage you to take a look on Apple's explanation, but here is the short version:



And the general pattern looks like this:



Basically there's an event, and an object gives a message to another object. In this case the button is calling a method on the view controller when the `.touchUpInside` event is activated.

Selectors

In iOS the target-action pattern usually comes in the combo `target-selector`, where the `target` is an object that is a subclass of `NSObject` and a `selector` that can be used to send messages to the target.

Why a subclass of `NSObject`? What's that?

iOS was built using Objective-C. Although swift is compatible with Objective-C, swift objects are different. In Objective-C, classes don't store methods in the same way. The short version is this: swift uses an array and Objective-C uses a dictionary. This is one of the reasons Swift is faster.

Swift figures out what function it should call at compile time with a thing call `vtable`. Objective-C figures out what function to call at runtime with a thing called dynamic method dispatch. To do this it needs both an object and a selector.

A root class is a class that does not inherit from any other class. It defines the behaviour for the hierarchy below it. In Objective-C the root class is `NSObject` - which means that most iOS classes are NSObjects - that includes views and controllers. `NSObject` defines a lot of low level

functionality like object lifecycle. Another thing `NSObject` defines is the perform selector methods:

```
perform(_:)
perform(_:with:afterDelay:)
perform(_:with:afterDelay:inModes:)
...
performSelector(inBackground:with:)
```

A selector is an object that is used to invoke a method at runtime. So instead of writing `object.thisMethod()` you can say `object.perform(#selector(thisMethod))`. This provides a convenient interface for deciding what code to run at runtime.

Let's take an example:

```
class SelectorTest: NSObject {
    func noParameters() {
        print("no parameters")
    }

    func unnamedParameter(_ parameter: String) {
        print("parameter = \(parameter)")
    }

    func namedParameter(message: String) {
        print("message = \(message)")
    }

    func twoParameters(first: String, last: String) {
        print("fullName = \(first) \(last)")
    }
}
```

The methods are defined by:

- `SelectorTest.noParameters`
- `SelectorTest.unnamedParameter(_:)`

- `SelectorTest.namedParameter(message:)`
- `SelectorTest.twoParameters(first:last:)`

The first part of the selector is the name of the class followed by a dot `.` then by the method name, and if it has parameters, they are shown separated by a colon `:` inside of a pair of parentheses. Note that if a parameter has no external name it will be represented by an `_` like in the second case. In most cases the class of a selector can be inferred, so you can omit it.

To create a `Selector` object you can use the `#selector` expression. To call those methods on an instance of `SelectorTest`, we would do something like this:

```
// first create an instance
let instance = SelectorTest()

// no parameters
instance.perform(#selector(noParameters))
// prints: no parameters

// one unnamed parameter
instance.perform(#selector(unnamedParameter(_:)),
    with: "We ❤ Swift")
// prints: parameter = We ❤ Swift

// one named parameter
instance.perform(
    #selector(namedParameter(message:)),
    with: "iOS Spellbook")
// prints: message = iOS Spellbook

// two parameters
instance.perform(
    #selector(twoParameters(first:last:)),
    with: "Andrei",
    with: "Puni")
// prints: fullName = Andrei Puni
```

Putting it all together

By using these methods the `UIButton` class can call any method you tell it, to just by performing a `Selector` on a target. This way you can configure the button to call any method on any object.

Usually you connect buttons to the view controller that displays them, but that is not the only case. In code it looks something like this:

```
// setup is called by the view controller sometime after the interfac
fun setup() {
    // first we need to add a target to the button
    button.addTarget(controller,
                     action: #selector(didTapButton(_:)),
    forControlEvents: .touchUpInside)
}

// this will be called when the button is tapped
func didTapButton(_ button: UIButton) {
    print("the button was tapped!")
}
```

There are other button events but you will probably not going to use them. In case you where curious `.touchUpInside` refers to the fact that the touch started and ended inside the bounds of the button.

The selector has to be compatible with Objective-C, that means that the first parameter should be unnamed.

IBAction

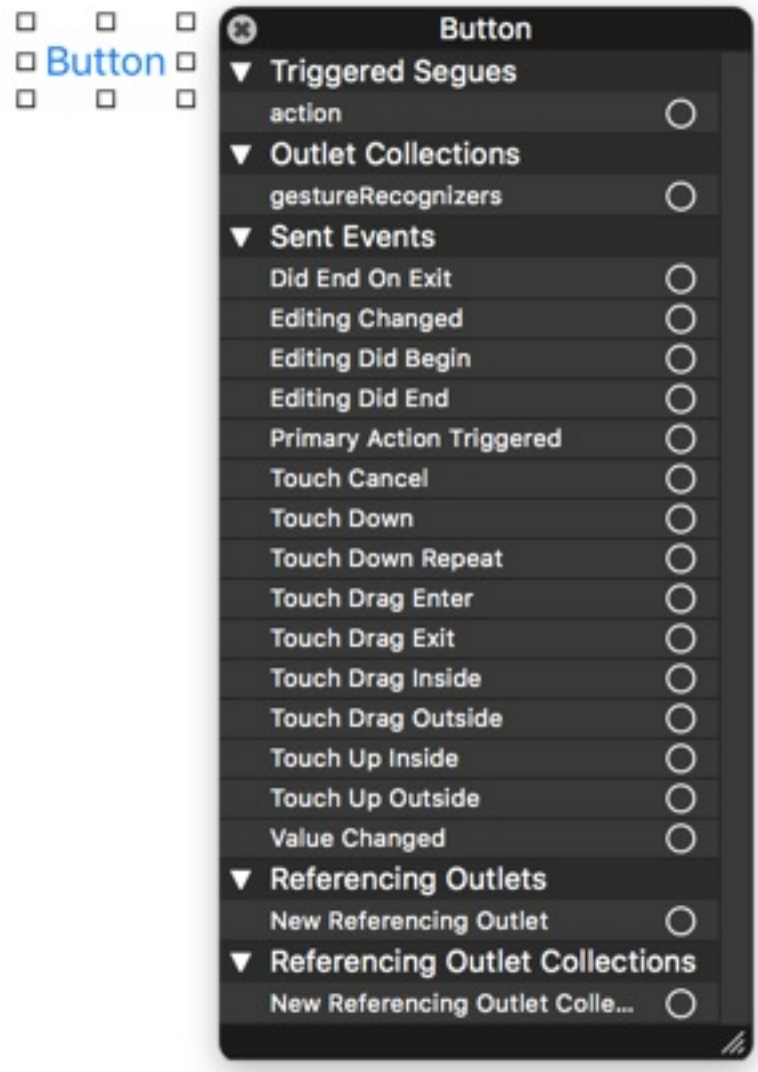
To make a method visible in interface builder, you have to mark it as an **`IBAction`**. You do that by writing **`@IBAction`** before **`func`**. After that it can be connected to an event sent by another object. Let's see how that would work on a button:

```
class ViewController: UIViewController {
```

```
    ...

    @IBAction func didTap(button: UIButton) {
        print("the button was tapped!")
    }
}
```

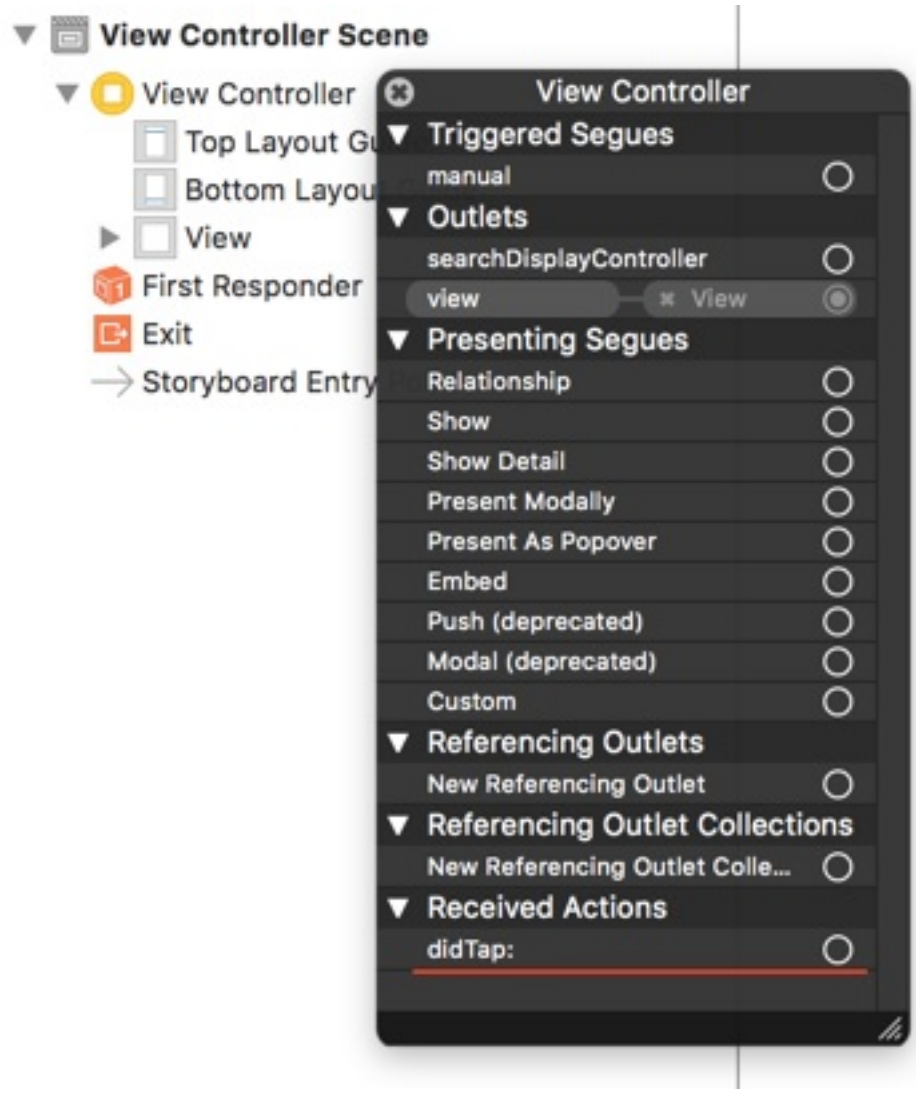
If you remember from the App Anatomy chapter, we discussed about the Objective-C preprocessor. `@IBAction` is also a preprocessor directive. It gets replaced with nothing. Its only purpose is to highlight a method as available for interface builder connections.

If you right click on a button in `Interface Builder` you will see a menu with all the events that it exposes:



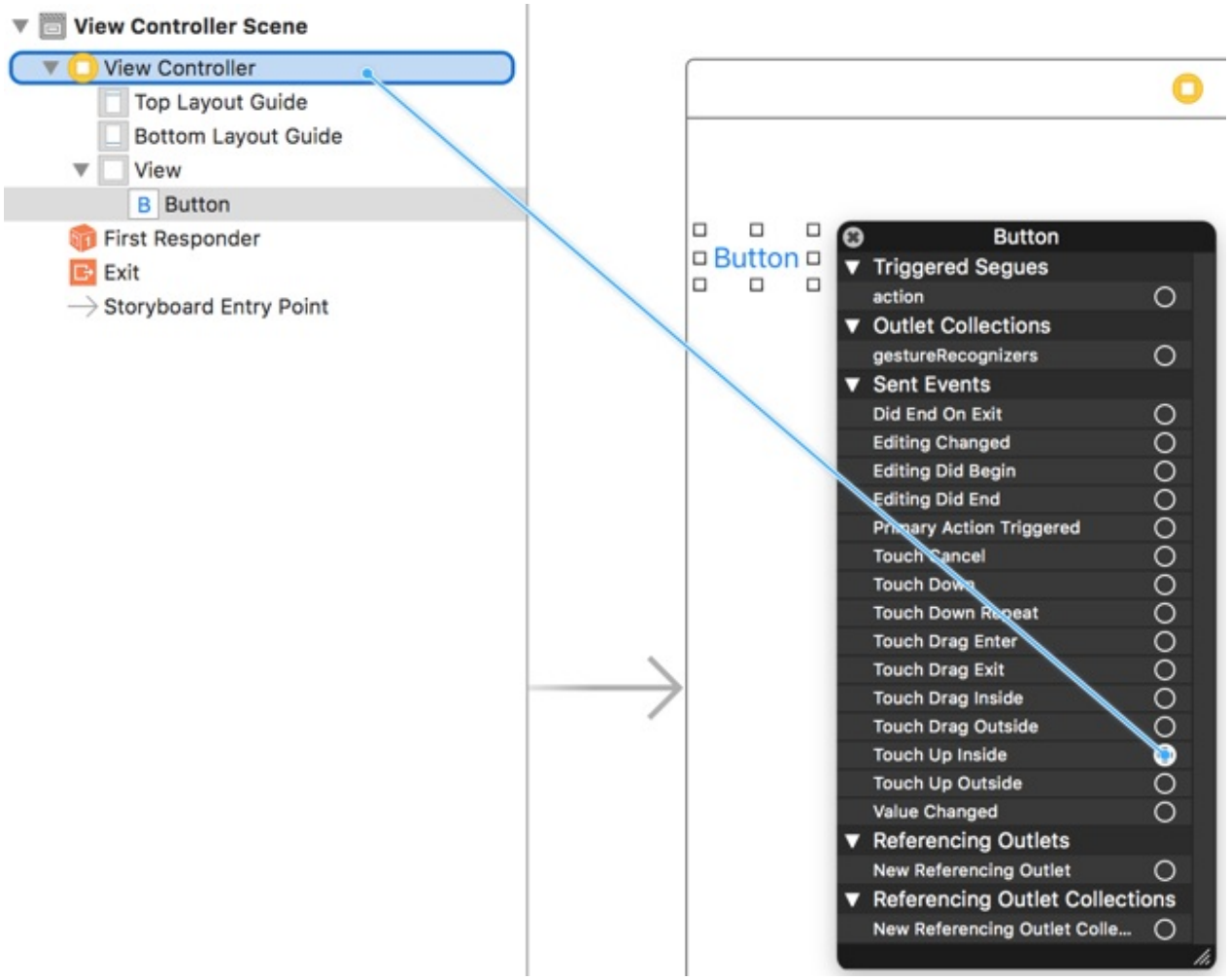
The button doesn't actually use all of them, the events are general and shared among all controls. `UIButton` inherits from `UIControl` - the class that defines the behaviour of an input view or control.

If you right click on `View Controller` you can see the `didTap` action in the bottom of the list:

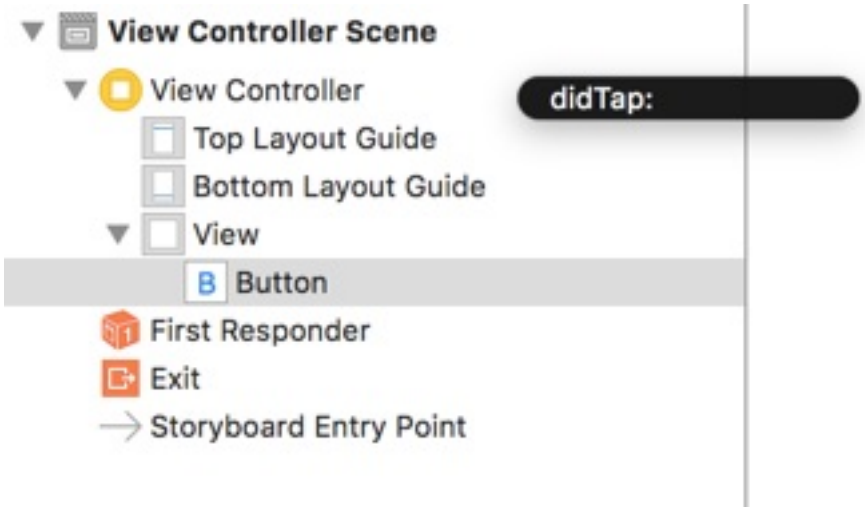


Now you can connect the didTap method to the `Touch Up Inside` event. You can do it from the controller to the button or vice versa:

Right click on the button, then click drag from the `Touch Up Inside` event to the `View Controller`:

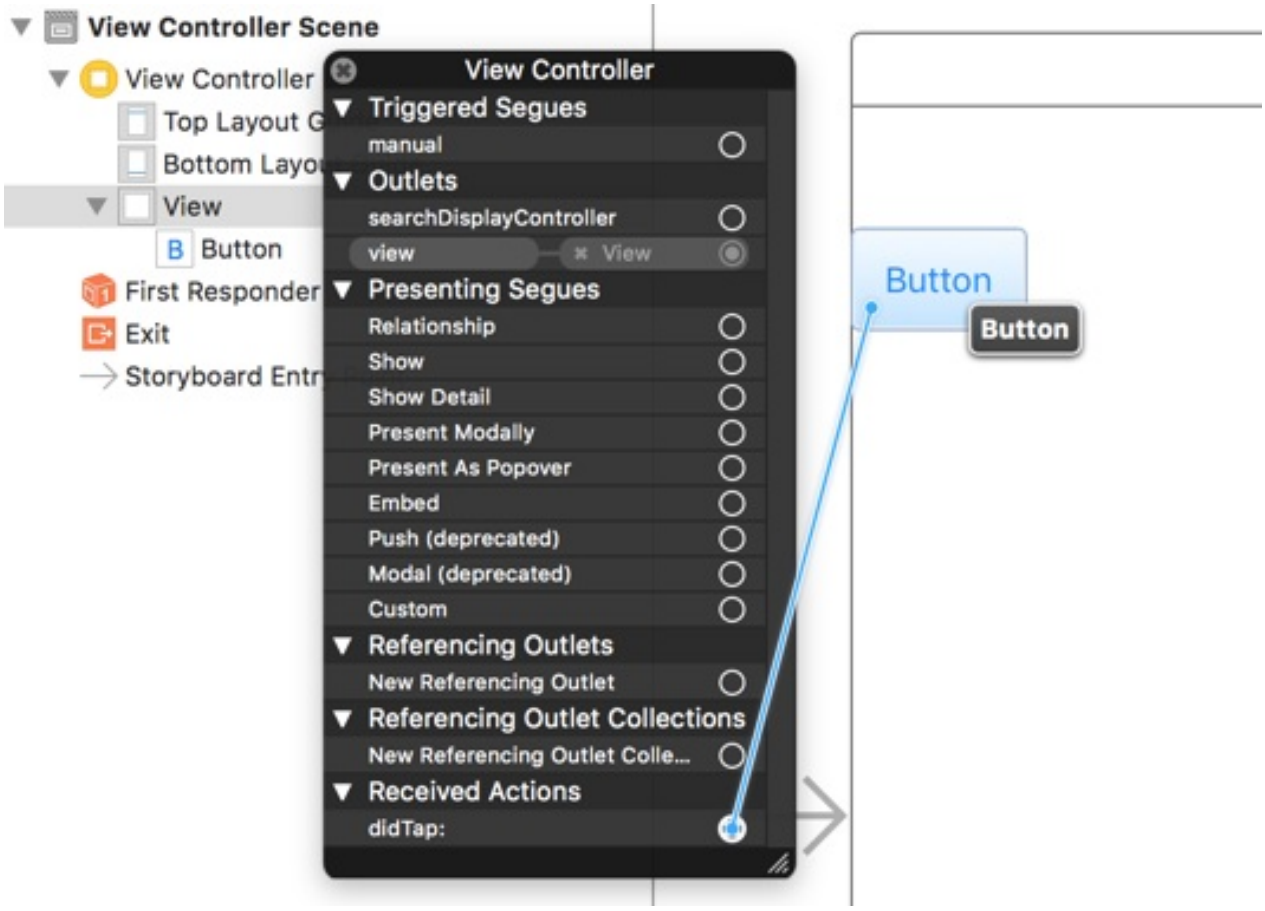


This should show the list of received actions:

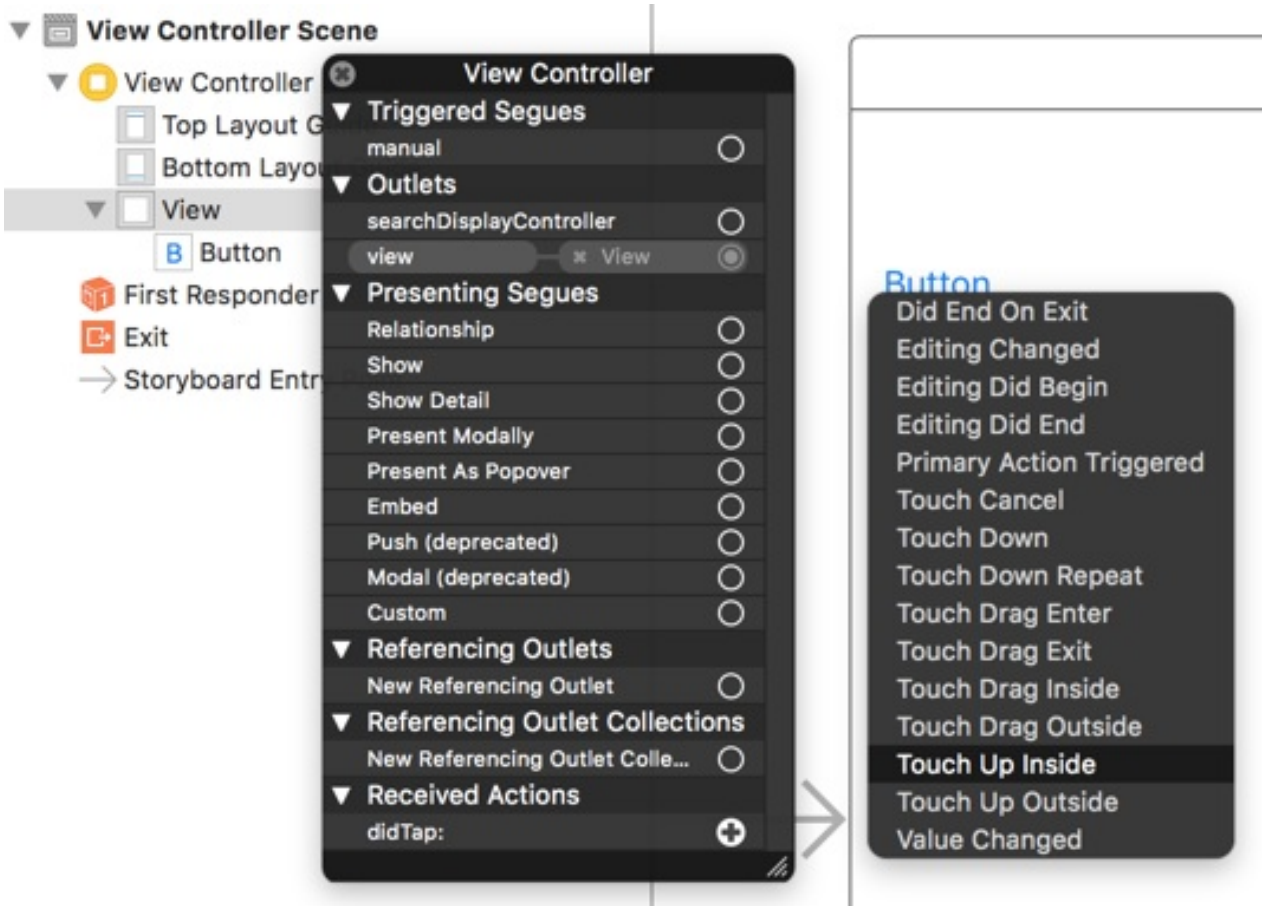


And if you click on `didTap:`, you will connect the two.

Another way you can do it is by starting from the action. Right click on `View Controller` and click drag from the `didTap:` action to the button:



This will expose the list of sent events:



Connect the two by clicking on `Touch Up Inside`.

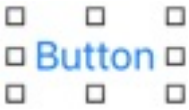
You can also make Xcode write the method for you if you use the `Assistant Editor` and connect the `Touch Up Inside` event with the view controller code.

Button States

A button can have five states: default, highlighted, focused, selected, and disabled.

You can configure some properties of the button for each state. This would allow you to make the font bold or a different color when the button is being pressed.

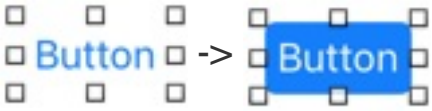
When the button first appears on the screen, it is in the default or normal state



When the user is touching the button but did not yet complete a tap, the button is in the highlighted state.

You can switch the state of the button to selected by setting the `selected` property to `true`:

```
button.selected = true
```



When the button is disabled, it's dimmed and doesn't display a highlight when tapped. You can disable a button by setting its `enabled` property to `false`:

```
button.enabled = false
```

The state of the button is represented by the `UIControlState` enum: `.normal`, `.highlighted`, `.focused`, `.selected` and `.disabled`.

Customization

Title

To change the title of a button you have to call `setTitle(:forState:)`. Notice that you have to specify for which state you want the title to be set. If you want the button to have the same title on all states then set it for the `.Normal` state and the button will inherit the title for all states:

```
button.setTitle("Tap Me!", forState: .normal)
```

To get the title for a state, you would call `title(for:)`:

```
button.title(for: .selected)
```

You can also use attributed strings:

```
// setting attributed title
button.setAttributedTitle(attributedTitle,
        forState: .normal)

// getting attributed title
button.attributedTitleForState(.normal)
```

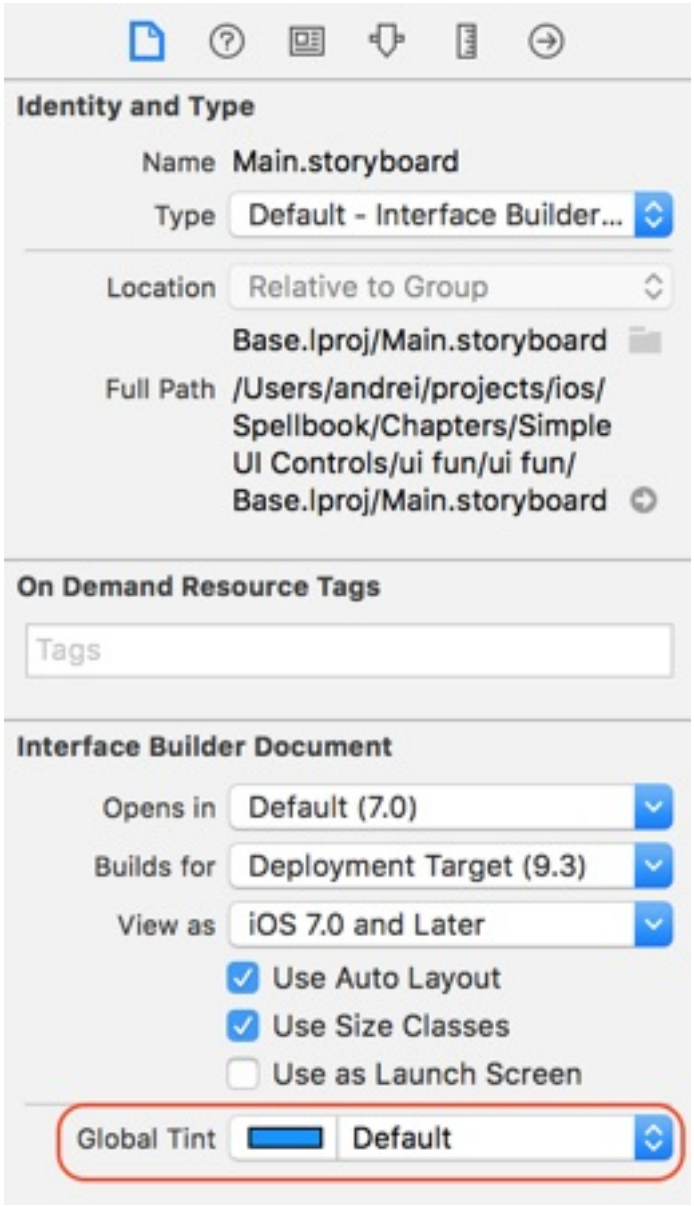
Tint Color

You can change the tint color by setting the `tintColor` property:

```
button.tintColor = .black
```

If you want to change the tint color for all the buttons from your app this is not the way to do it! Open the file inspector and look at the end of the

`Interface Builder Document` section:



You can change the global tint color from there and all components with the tint color set to default will have that color.

Image

In iOS images are represented by the `UIImage` class. You can create an image from a file (the image for a button) or directly from data (when you load cat memes from the internet). For the moment we are only going to load images from the `Resources` folder, so let's see how that works:

```
// create an image object
let image = UIImage(named: "swift-logo")

button.setImage(image, forState: .normal)
```

Same as with the title, if you set the image for the `.normal` state the rest will inherit it unless you set them to a different value.

Background Image

Same as you can update the image of a button, you can update the background by calling `setBackgroundImage(_:forState:)`:

```
let image = UIImage(named: "swift-logo")

button.setBackgroundImage(image, forState:.normal)
```

The main difference between the image of the button and its background image is positioning. If the image is small then it will be shown along the title. The background will always cover the button and will be rendered under the title.

System buttons

At the moment there are two system buttons you can use. There is nothing different about them, except for their appearance.

Add Contact

Used for buttons that represent an add new operation.

```
let button = UIButton(type: .contactAdd)
```

Info

Used to display extra information. There are two types of info buttons: one for a light background and one for a dark background.

```
let lightInfoButton = UIButton(type: .infoLight)
// or
let darkInfoButton = UIButton(type: .infoDark)
```

From what I've noticed the two look the same. Weird.

UITextField

A text filed is a label which can be edited. It can be configured to use different kinds of keyboards optimised for writing numbers, emails and so on.

Editing events

Whenever you interact with a text filed it generates an event, so the UI can react. There are two ways to receive editing events: you can subscribe using the `target-action` pattern or by providing a delegate to the text filed.

Target Action

`UITextFiled` is a subclass of `UIControl` which means that some events can be observed by using `addTarget` method. The relevant events for the text field are: `.EditingDidBegin`, `.EditingChanged`, `.EditingDidEnd` and `.EditingDidEndOnExit`. You can refer to all of them with `.AllEditingEvents`. You can also subscribe to all editing events with just one call to `addTarget`:

```
class ViewController: UIViewController {
    @IBOutlet var textField: UITextField!

    override func viewDidLoad() {
        super.viewDidLoad()
        textField.addTarget(self,
                  action: #selector(allEvents(_:)),
       forControlEvents: .allEditingEvents)
    }

    func allEvents(_ textField: UITextField) {
        print(textField.text!)
    }
}
```

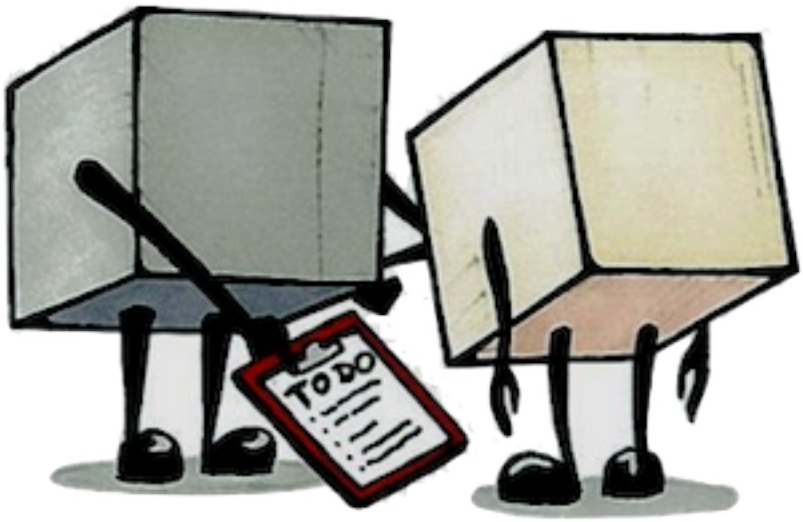
The target action should take as argument the control that was sent by the event, in this case the text field.

One of the problems with this method is that you can't customize the behaviour of the text by rejecting certain characters or deciding if editing should stop or not.

Text Filed Delegate

In case you don't know or forgot what the delegate pattern is, here is a quick reminder: **Delegation is a design pattern where an object hands out the responsibility of accomplishing a task to another object**.

Whoa, that was a mouthful! Here's an image to help you visualize:



In the case of the text field delegate, it receives editing events and sometimes its asked to decide about the text view behaviour.

The way this pattern is implemented is by defining a protocol that describes the responsibilities of the delegate. So for an object to be a text field delegate it has to implement to the `UITextFieldDelegate` protocol.

All the methods of `UITextFieldDelegate` are optional so you can use only the ones you need.

Some of the methods from the protocol just inform the delegate that a certain event has occurred:

```
textFieldDidBeginEditing(_:)
textFieldDidEndEditing(_:)
```

And some ask the delegate if a certain action should be taken or not by retuning a `Bool`:

```
textFieldShouldBeginEditing(_:)
textFieldShouldEndEditing(_:)
textFieldShouldClear(_:)
textFieldShouldReturn(_:)
textField(_:shouldChangeCharactersIn:replacementString:)
```

Notice that all event related methods have the form `sender` + `did` + `event` and all the rest have `sender` + `should` + `action`. This convention keeps the system code easy to understand or discover.

> 
>
> **Quick tip**
>
> You don't need to know the list of methods from the delegate protocols of any control! It's enough to remember that they have one. They usually have the name of the control + delegate so its easy to deduce. Write it in Xcode and `Command` (`⌘`) + click on it. That will take you to the definition of the protocol.

I usually make an extension for each protocol my view controller implements to keep the code clean and separated:

```
extension ViewController: UITextFieldDelegate {
    func textFieldShouldReturn(_ textField: UITextField) -> Bool {
        if textField.text!.characters.count > 0 {
            textField.resignFirstResponder()
            return true
        } else {
            return false
        }
    }
}
```

The `textFieldShouldReturn` method is called when the user presses the return key. If the delegate returns `true`, the text filed will stop editing. This is probably the most common reason people implement the delegate protocol.

You can use the `shouldChangeCharactersInRange` delegate method to reject certain inputs and force a certain value to be introduced. For example you can force a text field to accept only numbers, and more than that, only ones that don't start with 0.

```
extension ViewController: UITextFieldDelegate {
        func textField(_ textField: UITextField,
         shouldChangeCharactersIn range: NSRange,
               replacementString string: String) -> Bool {
        // simulate the result after the edit
        let text = (textField.text! as NSString)

        let result =
            text.replacingCharacters(in: range,
                                     with: string)

        // check if the result has only digits
        let onlyDigits =
            result.characters.reduce(true) {
                hasNonNumber, character in
           hasNonNumber &&
             "0123456789".characters.contains(character)
        }

        // check if first digit is not 0
```

```
            let firstDigitIsNotZero = result.characters.first != "0"

            return onlyDigits && firstDigitIsNotZero
        }
    }
```

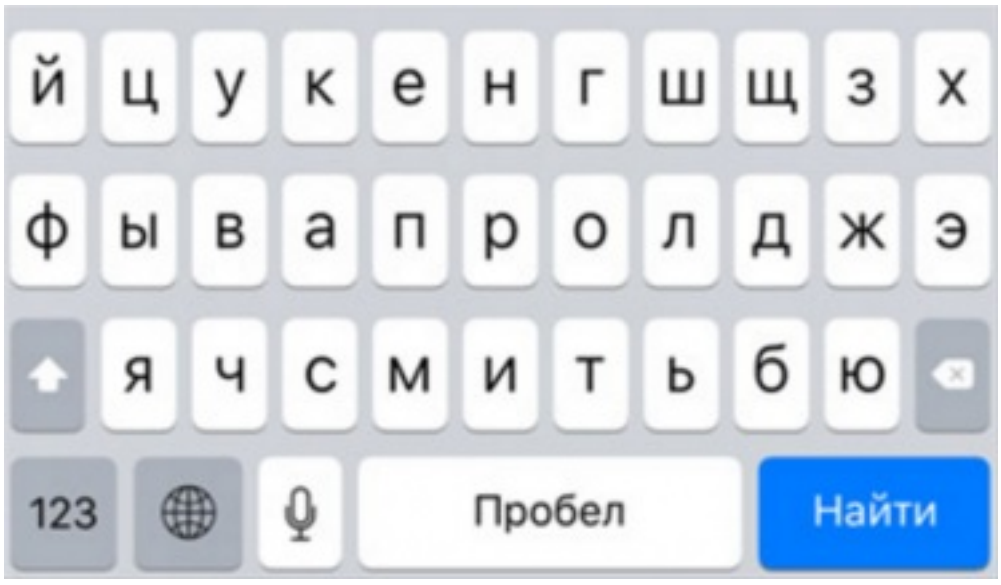
Handling the Keyboard

Keyboard types

There are several system keyboard layouts optimised for different kinds of input:

Default

The default keyboard is the general text input. Depending on the language of the phone, the keyboard might change. For example if the language is set to Russian, the keyboard would look like this:



ASCII Keyboard

A keyboard which can enter ASCII characters.

```
textField.keyboardType = .asciiCapable
```

Numbers and punctuation

You can access this keyboard by pressing the bottom left button on system's keyboard optimised for words.

```
textField.keyboardType = .numbersAndPunctuation
```

URL

A keyboard optimised for entering URL. It's similar to the default keyboard but it has the space key replaced with `.`, `/` and `.com`.

```
textField.keyboardType = .URL
```

Email

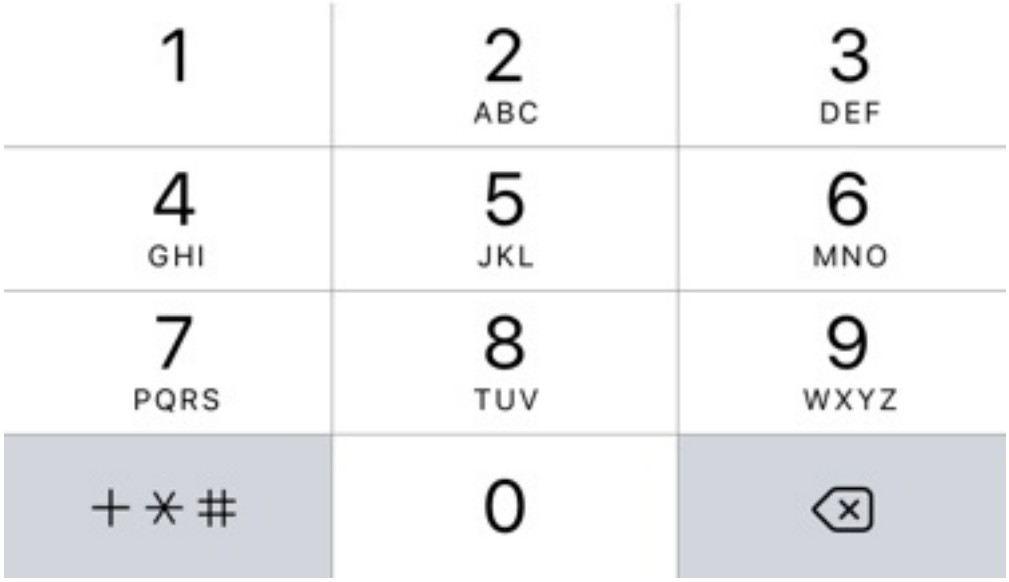
Similar to the URL keyboard this one replaces the space key with `@` and `.`. And also leaves a small space key, so the user is able to enter multiple emails.

```
textField.keyboardType = .emailAddress
```

Phone numbers

A keyboard that mimics the keyboard of what people used to call phones. This keyboard is used by the contacts app.

```
textField.keyboardType = .phonePad
```



Web search

The web search keyboard adds a `.` in the bottom right corner next to space and replaces the return button with `Go`

Twitter

Believe it or not, iOS has a keyboard optimised for writing tweets! It replaces the return key of the default keyboard with `@` and `#`.

Opening the keyboard

Sometimes you want to have the keyboard opened without user interaction. Like in a sign up form, because you want the user to have to take as few steps as possible. You can do this by telling the text field to become the first responder:

```
textField.becomeFirstResponder()
```

Dismissing the keyboard

To dismiss the keyboard you have to tell the text field to resign as the first responder:

```
textField.resignFirstResponder()
```

In case you don't have a reference to the textField, you can tell the view that contains the text field - or another input - to end editing and dismiss the

keyboard:

```
view.endEditing(true)
```

This is really convenient when you have may inputs on the screen because you don't have to change your code if you change the interface.

The text field is not a silver bullet!

The text field is the most used component for text input. It is also the most misused component! There are a lot of cases where text input is not necessary because the number of valid inputs is small or you can reduce them without losing functionality or there is a better control for that kind of input (ex. RGB values).

Let's take for example apartment listings. In a search form there is usually a field for the number of rooms:

Instead of a text field you can use a segmented control:

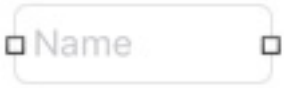
Before you throw a text field in front of the user, ask yourself if you can replace it with another control that will do the job better that entering text.

Customization

Placeholder

When the text file is empty it displays a placeholder string to signal what

should be written inside it. When the text field becomes the first responder, the placeholder disappears.



Border

Depending on the style of your app you might want to use a different kind of border than the default one:

Rounded

Displays a rounded-style border for the text field. This is the default option.

```
textField.borderStyle = .roundedRect
```


Line

Displays a thin rectangle around the text field.

```
textField.borderStyle = .line
```

Bezel

Displays a bezel-style border for the text field. This style is typically used for standard data-entry fields.

```
textField.borderStyle = .bezel
```

None

The text field does not display a border.

```
textField.borderStyle = .none
```

Background Image

You can change the way a text field looks by changing it background image. The image will be stretched to fit the size of the text field.

So if you use an image that looks like this:


```
textField.background = UIImage(named: "JustALine")
```

You will get a text field that looks like this:

Secure Input

For passwords or other sensitive information you should user secure text

fields. To do so set the `secureTextEntry` property to `true`:

```
textField.secureTextEntry = true
```

UITextView

The text view is a scrollable multi-line text container that also supports editing. The purpose of the text view is to display large amounts of text like comments of the body of an email.

Links

A cool thing that you can do with a text view and can't with a label or text field, is to let the user open links.

There are two ways to highlight links in a text view:

1. By enabling automatic detection of link, addresses, phone numbers or events/dates
2. Using attributed strings

By default iOS will give safari any website. Some links have a different format and will be handled by an app that supports it. For example opening `tel://0800123123` will call `0800123123` and `sms://0800123123` will open the Messages app in the new message screen. If you want to see a complete list of URL schemas, browse this page.

Delegate

Most of the methods are similar, if not the same, as the `UITextFieldDelegate` and they handle basic editing events.

Another thing the delegate lets you customize is the behaviour of your app when the users opens an URL. Instead of opening the application responsible for that URL, you can load the content in a web view or trigger an alternative action.

```
extension ViewController: UITextViewDelegate {

    func textView(_ textView: UITextView,
        in characterRange: NSRange,
        interaction: UITextItemInteraction) -> Bool {

        openWebViewAt(URL)

        return false
    }
}
```

Handling Keyboard

The keyboard is controlled the same way as the text field is. Make the text view the first responder to show the keyboard and resign it to dismiss the keyboard.

Customization

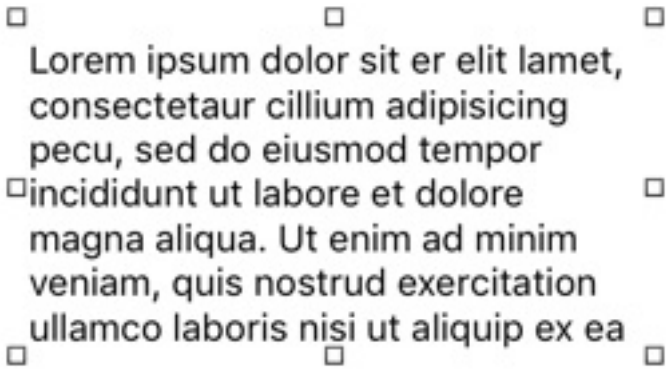
Text Alignment

You can change the text alignment by setting the `textAlignment` property.

Left

The text is left aligned. This is the default option.

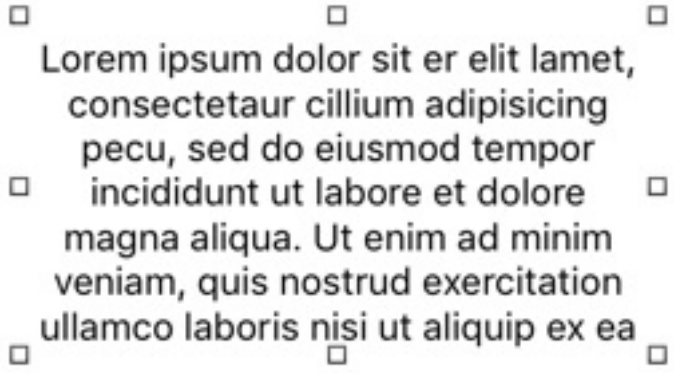
```
textView.textAlignment = .left
```



Center

Text is centered.

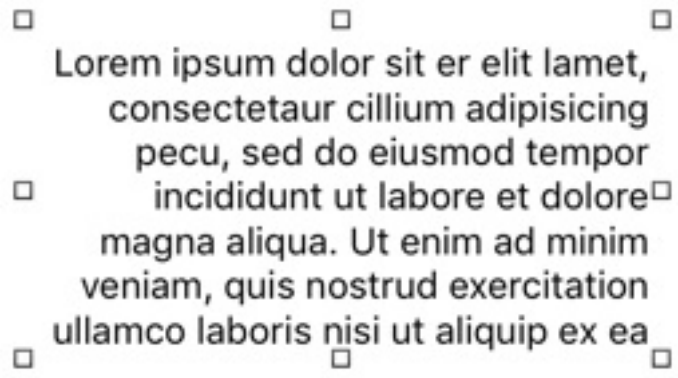
```
textView.textAlignment = .center
```



Right

Text is right aligned.

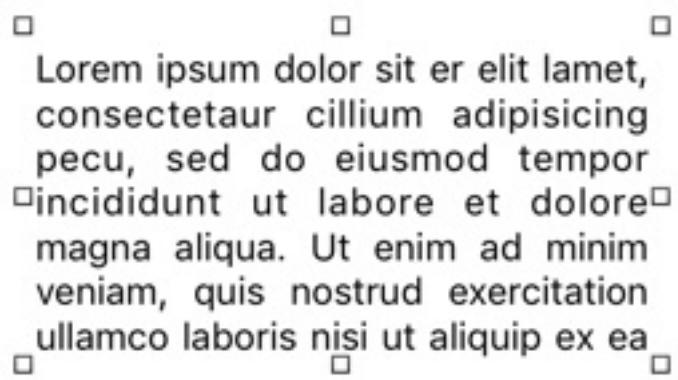
```
textView.textAlignment = .right
```



Justified

Fully-justified. The last line in a paragraph is natural-aligned.

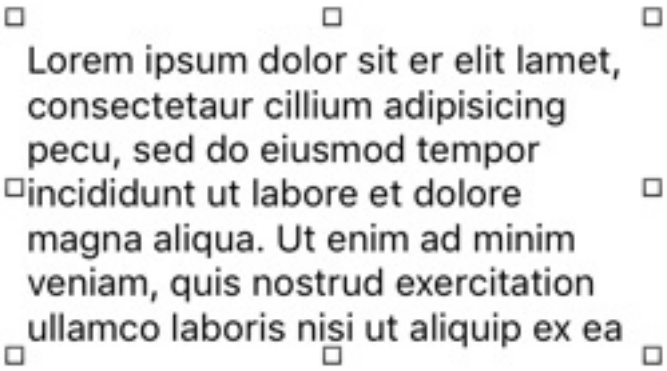
```
textView.textAlignment = .justified
```



Natural

The default alignment associated with the current localization of the app.

```
textView.textAlignment = .natural
```



Scrolling

By default a text view is scrollable. You can disable scrolling by setting the `scrollEnabled` property to `false`:

```
textView.scrollEnabled = false
```

Editing

You can enable or disable the editing behaviour of the text view by setting the `editing` property.

```
textView.editable = false
```

Attributed Text

Like the label, the text view can display attributed text.

UIImageView

`UIImageView` is the view that displays a single `UIImage`. The image can be set using the `image` property.

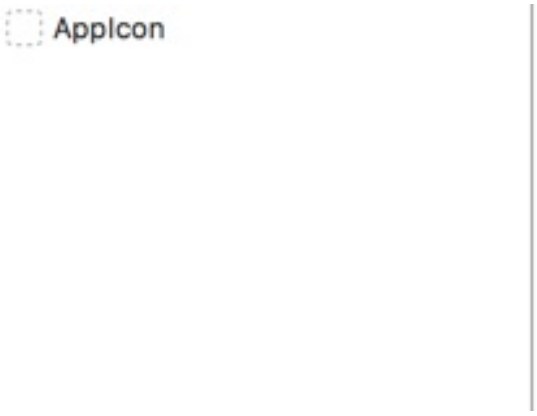
```
let image = UIImage(named: "heart")
let imageView = UIImageView(image: image)
```

By default `UIImageView` will change its size to fit the image unless you set its size.

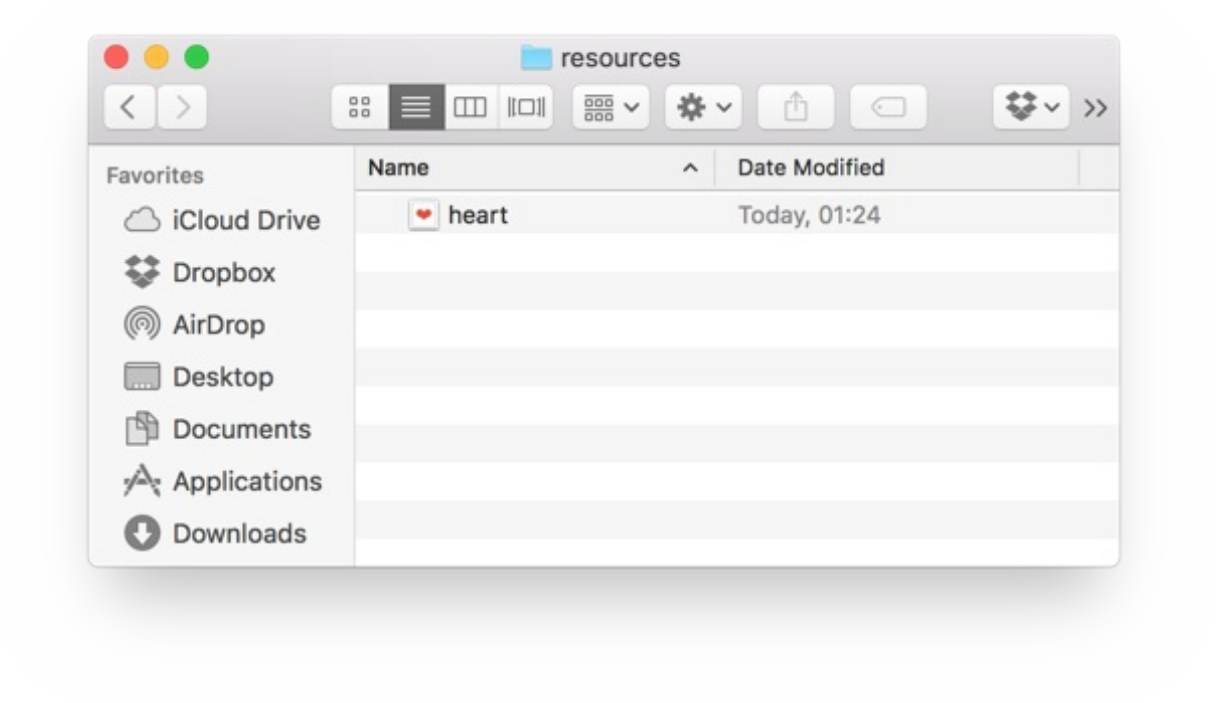
Adding images in Xcode

You can add images along you swift files but that gets messy pretty fast. The new way of handling resources on iOS is by using Asset Catalogs.

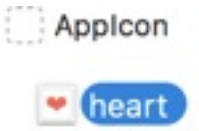
Open `Assets.xcassets`:



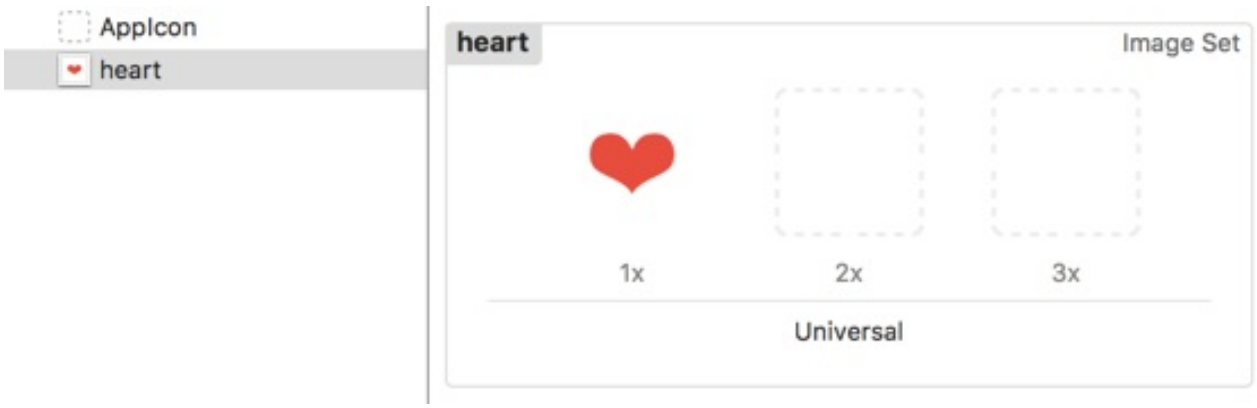
Now open finder and go to the folder that contains the resources you want to add:



Click and hold the resource and start dragging it in the direction where `AppIcon` used to be in Xcode. Don't let go of the file! Hit `Command` + `tab` (`⌘` + `tab`) to switch back to Xcode and drop the file somewhere under the icon image set.

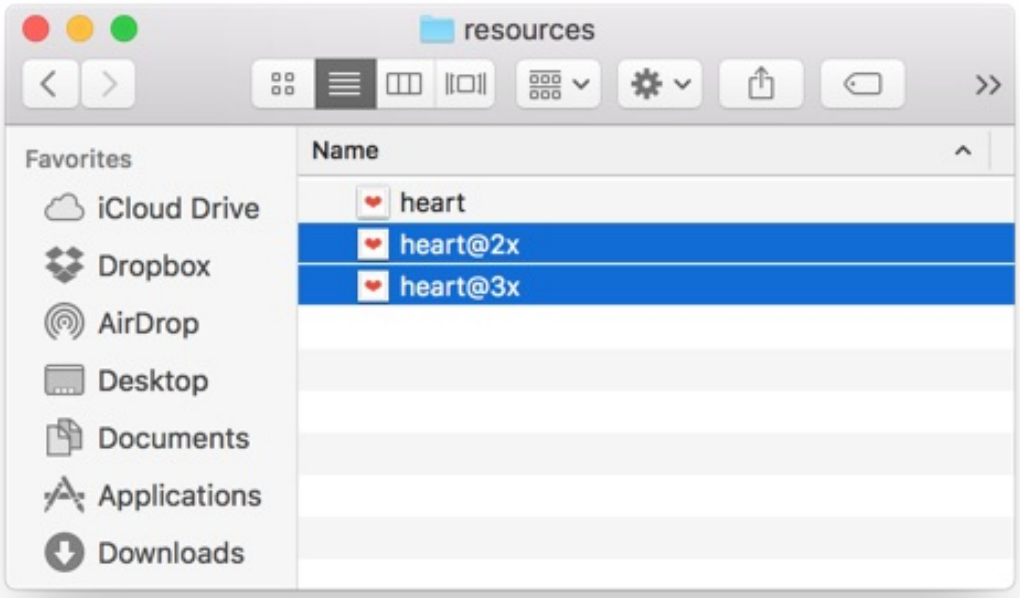


That will create an image set for you resource.

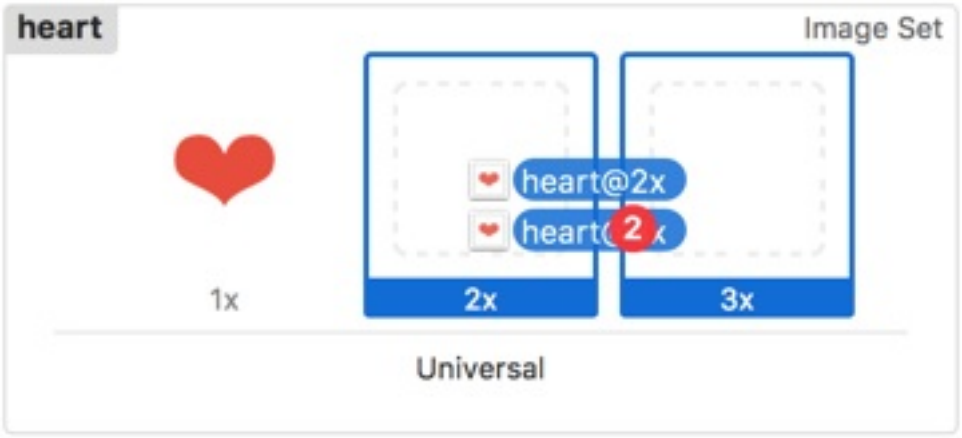


An image set is a collection of images that are the same image at different resolutions. This comes from the fact that iOS devices have retina screens with different pixel densities.

To specify the pixel density, you have to add the suffix `@2x` for 2 by 2 pixels per point and `@3x` for 3 by 3 pixels per point. You can find the complete list of specification for naming your resources here.



Select the image set you want to add more resources to, after that go in finder and drag and drop the resources inside it:



After that the image set should be complete:

Content Mode

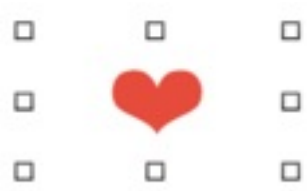
The content mode of an image specifies how the content should be drawn.

scaleToFill

The default value for `contentMode` is `.scaleToFill` which scales the image so that it fits in the bounds of the image view. Notice that this may distort the image.

scaleAspectFit

The option to scale the content to fit the size of the view by maintaining the aspect ratio. Any remaining area of the view's bounds is transparent.



scaleAspectFill

The option to scale the content to fill the size of the view. Some portion of the content may be clipped to fill the view's bounds.

redraw

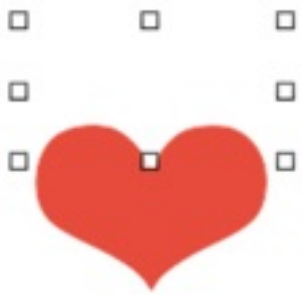
The option to redisplay the view when the bounds change by invoking the `setNeedsDisplay()` method. Also distorts the image.

center

The option to center the content in the view's bounds, keeping the proportions the same.

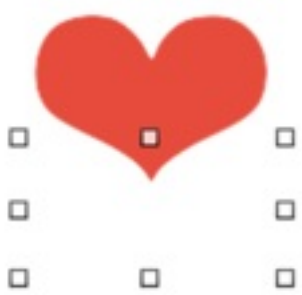
top

The option to center the content aligned at the top in the view’s bounds.



bottom

The option to center the content aligned at the bottom in the view’s bounds.



left

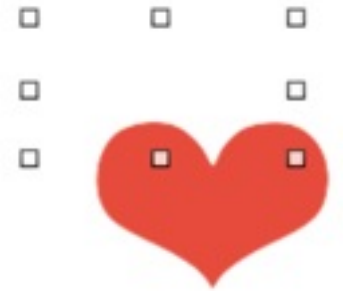
The option to align the content on the left of the view.

right

The option to align the content on the right of the view.

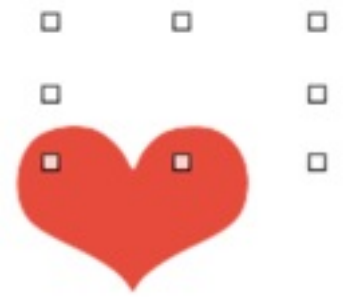
topLeft

The option to align the content in the top-left corner of the view.



topRight

The option to align the content in the top-right corner of the view.

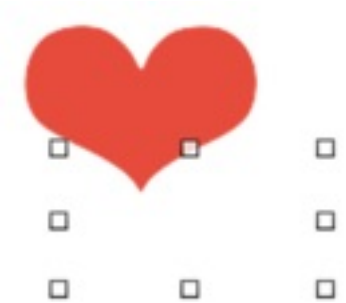


bottomLeft

The option to align the content in the bottom-left corner of the view.

bottomRight

The option to align the content in the bottom-right corner of the view.



User Interaction

By default the `isUserInteractionEnable` view property is set to `false` on image views. If you want to use gesture recognizers on images, either set the property to `true`, or add the image view inside a container view.

UISwitch

The `UISwitch` is a control that represents a boolean value that can be flipped on/off. It holds its state inside the `isOn` property.

Events

You get notified of events by connecting to the `valueChanged` control event.

```
@IBAction func didFlipSwitch(_ switch: UISwitch) {
    print(switch.isOn)
}
```

Customization

onTintColor

By setting the `onTintColor` property you can change the color of the active state:

```
mySwitch.onTintColor = .purple
```

thumbTintColor

By setting the `onTintColor` property you can change the color of the knob:

```
mySwitch.thumbTintColor = .purple
```

UISegmentControl

`UISegmentedControl` is a horizontal control made of multiple segments, each segment functioning as a button. Each segment can show a title and an image and has a state, either selected or not. Only one segment can be selected at once. When a segment is selected, the one that was selected before (if any) will be unselected. This control is used when the user has to select between multiple things of the same kind.

Events

You get notified of events by connecting to the `valueChanged` control event. You can get the selected index from the `selectedSegmentIndex` property - the segments are indexed starting with 0. If you need to get the title of a segment you can do so by using the `titleForSegment(at:)` method.

```
@IBAction didSelectSegment(
    _ segmentControl: UISegmentControl) {
    let index = segmentControl.selectedSegmentIndex
    let title = segmentControl.titleForSegment(at: index)

    print("selected segment at \(index)")
    print("title: \(title)")
}
```

UISlider

The `UISlider` control represents a value that can range over an interval.

Events

You get notified of events by connecting to the `valueChanged` control event. The value of the slider is held in the `value` property

```
@IBAction func didMoveSlider(_ slider: UISlider) {
    print(slider.value)
}
```

Customization

minimumValue

Use this property to set the value that the leading end of the slider represents. If you change the value of this property, and the current value of the slider is below the new minimum, the slider adjusts the value property to match the new minimum. If you set the minimum value to a value larger than the maximum, the slider updates the maximum value to equal the minimum. The default value of this property is `0.0`.

maximumValue

Use this property to set the value that the trailing end of the slider represents. If you change the value of this property, and the current value of

the slider is above the new maximum, the slider adjusts the value property to match the new maximum. If you set the maximum value to a value smaller than the minimum, the slider updates the minimum value to equal the maximum. The default value of this property is `1.0`.

- task priority (merge color red + green)
- color = red slider.value + green (1-slider.value)

Thumb Image

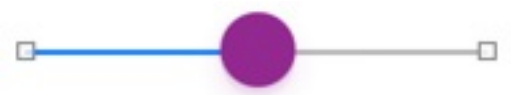
You can change the thumb image by using the `setThumbImage(_:for:)` method:

```
slider.setThumbImage(coolThumb, for: .normal)
```

Thumb Tint

You can change the thumb tint by setting the `thumbTintColor` property:

```
slider.thumbTintColor = .purple
```



Min/Max Track Tints

```
slider.minimumTrackTintColor = .red
slider.maximumTrackTintColor = .black
```

Exercises

Tap me if you can

Make an app with a single view controller that displays a round red button. Set the button title to `Tap Me!`.

Make the button move to a random position every 2 seconds. You can do this with the help of `perform(_:with:afterDelay:)`.

Solution

Color Tap

Make an app with a single view controller that displays 6 colored buttons and two labels. One of the labels is the score label and the other one is the challengeLabel.
Every round the label is set a random color and text. The text is the name of a random color. The user scores `1` point if he touches the button with the color the text says and `-10` otherwise. If the user does not select a color in

3 seconds he should be penalized 1 point and the color label should randomize.

Make the phone vibrate every time the user makes a wrong move:

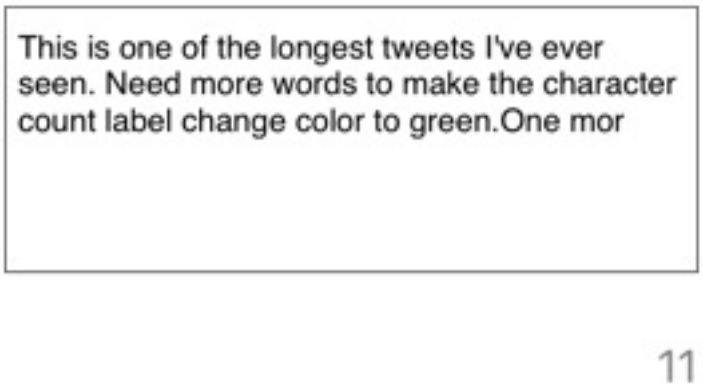
```
AudioServicesPlayAlertSound(
    SystemSoundID(kSystemSoundID_Vibrate))
```

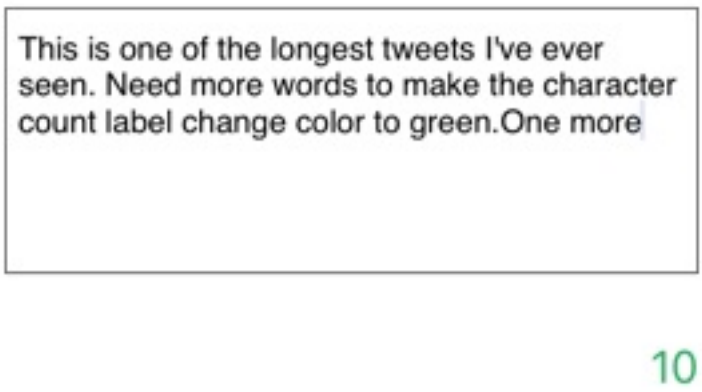
You will need to import the the AVFoundation framework. Click on the project, then on target, then select the Build Phases tab. In the `Link Binary With Framework` category press the "+" button and search for AVFoundation.

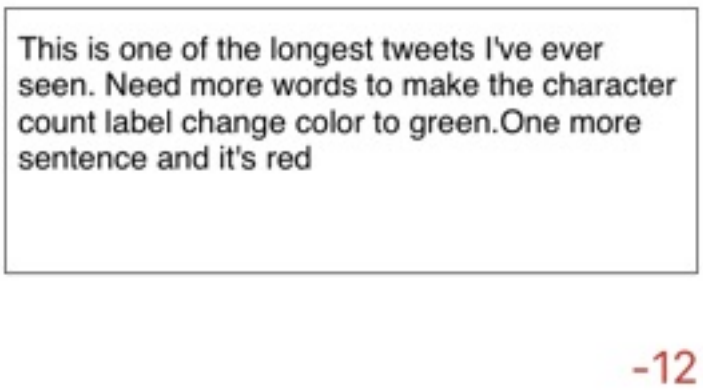
Solution

Tweet Screen

Make a tweet screen usint `UITextView`. A label should indicate the number of characters they have left until 140. The label should have the text color gray if the number of empty spaces is above `10`, green is it's under `10` and above equal to `0`, and red if there are more than 140 characters in the tweet.



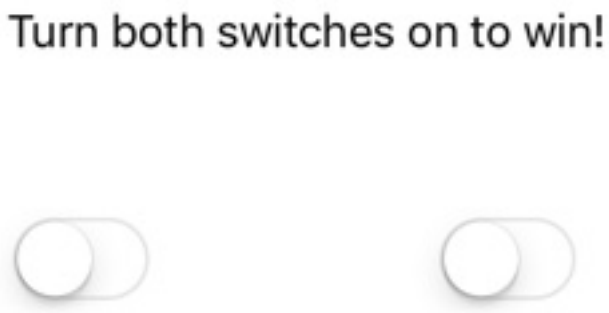




Solution

Impossible Game

Make an app with a single screen that has two switches and a label with the message "Turn both switches on to win!". Initially both switches are off. When both switches are on, flip the switch that was changed before.



Solution

Alpha slider

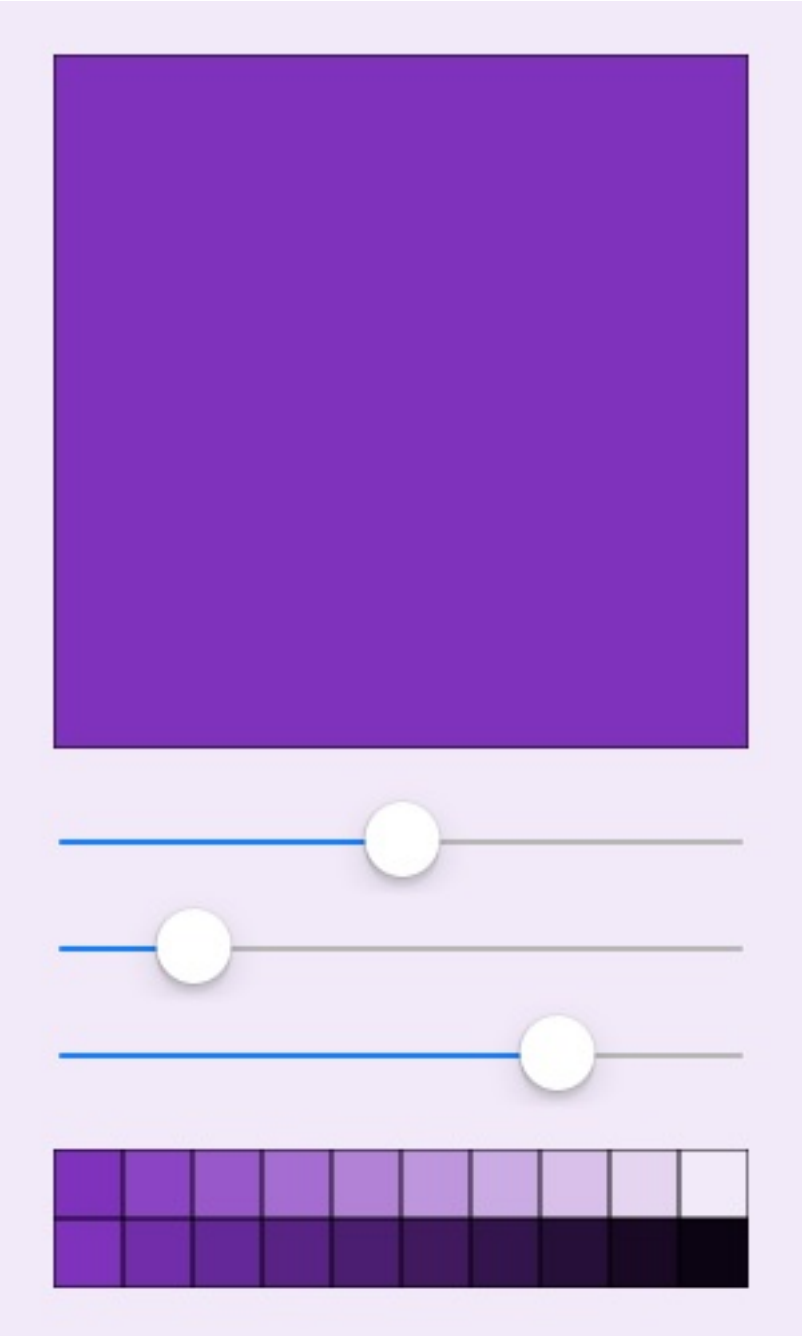
Bind the value of a slider to the alpha value of a view.

Solution

Color Picker

Make a color picker that also show lighter and darker shades so you can

browse the color space more efficiently:



Step 1: Create a new Playground

Open Xcode and create a new Playground, set the platform to iOS.

Step 2: Create a View Controller

Create a subclass of `UIViewController` and initialize it's view with:

```
class ViewController: UIViewController {
    override func loadView() {
        let width: CGFloat = 300
        let height: CGFloat = 500

        let screenSize = CGSize(width: width,
                                height: height)
        let frame =
            CGRect(origin: .zero,
                   size: screenSize)

        view = UIView(frame: frame)
        view.backgroundColor = .white
    }
}
```

Step 3: Show the view in the Playground

The first thing we need to do is to import the `PlaygroundSupport` module:

```
import PlaygroundSupport

class ColorPicker: UIViewController {
    ...
}
```

Create a instance of `ColorPicker`

```
let colorPicker = ColorPicker()
```

Set it as the `liveView` of the current Playground page:

```
PlaygroundPage.current.needsIndefiniteExecution = true
PlaygroundPage.current.liveView = colorPicker.view
```

Open the Assistand Editor by pressing `Command` + `Option` + `Return` (`⌘` +

⌥ + ↩). You should be able to see a white rectangle in the right side of the playground window.

Step 4: Basic UI

We need one sliders for each color component:

```
class ColorPicker: UIViewController {
    var redSlider: UISlider!
    var greenSlider: UISlider!
    var blueSlider: UISlider!

    func loadView() {
        ...

        let sliderSize =
            CGSize(width: width - 2 * padding,
                   height: 30)

        let redSliderFrame =
            CGRect(origin: CGPoint(x: padding, y: 300),
                   size: sliderSize)

        redSlider = UISlider(frame: redSliderFrame)
        redSlider.addTarget(self,
                        action: #selector(didMoveSlider(_:)),
                           for: .valueChanged)

        view.addSubview(redSlider)

        let greenSliderFrame =
            CGRect(origin: CGPoint(x: padding, y: 340),
                   size: sliderSize)

        greenSlider = UISlider(frame: greenSliderFrame)
        greenSlider.addTarget(self,
                              action: #selector(didMoveSlider(_:)),
                              for: .valueChanged)

        view.addSubview(greenSlider)


        let blueSliderFrame =
            CGRect(origin: CGPoint(x: padding, y: 380),
```

```
                    size: sliderSize)

        blueSlider = UISlider(frame: blueSliderFrame)
        blueSlider.addTarget(self,
                             action: #selector(didMoveSlider(_:)),
                             for: .valueChanged)

        view.addSubview(blueSlider)
    }

    func didMoveSlider(_ slider: UISlider) {

    }
}
```

Note that all sliders call the `didMoveSlider(_:)` method when their value changes. That's because we have to do the same thing if any of them changes - update the color on the screen.

And if we mentioned that let's make a view to display our color:

```
class ColorPicker: UIViewController {
    ...

    var colorView: UIView!

    func loadView() {
        ...

        let colorViewFrame =
            CGRect(x: padding, y: padding,
               width: width - 2 * padding,
              height: width - 2 * padding)
        colorView = UIView(frame: colorViewFrame)
        colorView.backgroundColor = .white
        colorView.addBorder()
    }
}
```

The last line ads a border around the view so we can see the view even if it's white.

```
extension UIView {
    func addBorder() {
        layer.borderWidth = 0.5
    }
}
```

At this point the UI should look like this:

Step 5: Update the color

```
class ColorPicker: UIViewController {
    ...

    func updateColor() {
        let redAmount = CGFloat(redSlider.value)
        let greenAmount = CGFloat(greenSlider.value)
        let blueAmount = CGFloat(blueSlider.value)

        let color = UIColor(red: redAmount,
                            green: greenAmount,
                            blue: blueAmount,
                            alpha: 1)

        colorView.backgroundColor = color
    }

    func didMoveSlider(_ slider: UISlider) {
        updateColor()
    }
}
```

`colorView` should update when you move a slider.

Step 6: Show shades

```
class ColorPicker: UIViewController {
    ...

    let shadesCount = 10

    var lightShades: [UIView] = []
    var darkShades: [UIView] = []

    override func loadView() {
        ...

        let totalWidth = width - 2 * padding
        let shadeSize = totalWidth / CGFloat(shadesCount)

        for i in 0..<shadesCount {
            // light shade
            let lightShadeFrame =
                CGRect(x: padding + CGFloat(i) * shadeSize,
                       y: 430,
```

```
                        width: shadeSize,
                        height: shadeSize)
            let lightShade = UIView(frame: lightShadeFrame)
            lightShade.backgroundColor = .white
            lightShade.addBorder()

            lightShades.append(lightShade)
            view.addSubview(lightShade)

            // dark shade
            let darkShadeFrame =
                CGRect(x: padding + CGFloat(i) * shadeSize,
                        y: 430 + shadeSize,
                        width: shadeSize,
                        height: shadeSize)
            let darkShade = UIView(frame: darkShadeFrame)
            darkShade.backgroundColor = .white
            darkShade.addBorder()

            darkShades.append(darkShade)
            view.addSubview(darkShade)
        }
    }
}
```

Update shades when the color changes:

```
class ColorPicker: UIViewController {
    ...

    func updateColor() {
        let redAmount = CGFloat(redSlider.value)
        let greenAmount = CGFloat(greenSlider.value)
        let blueAmount = CGFloat(blueSlider.value)

        let color = UIColor(red: redAmount,
                            green: greenAmount,
                            blue: blueAmount,
                            alpha: 1)

        colorView.backgroundColor = color

        for i in 0..<shadesCount {
            let t = CGFloat(i) / CGFloat(shadesCount)
```

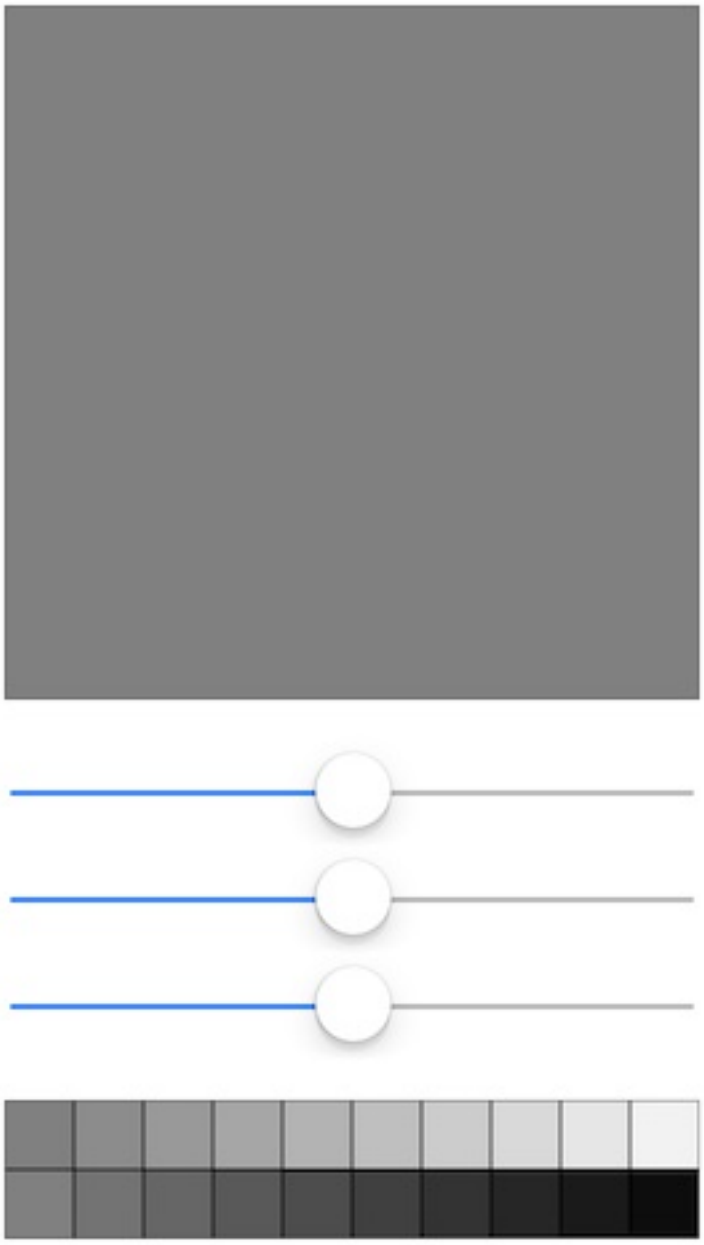
```
            let lightShade = lightShades[i]
            lightShade.backgroundColor =
                color.combine(with: .white, amount: t)

            let darkShade = darkShades[i]
            darkShade.backgroundColor =
                color.combine(with: .black, amount: t)
        }
    }
}
```

Step 7: Initalize Sliders

Set the value of each slider to `0.5` so we can test if the shades are displayed correctly:

```
class ColorPicker: UIViewController {
    ...

    override func viewDidLoad() {
        super.viewDidLoad()

        redSlider.value = 0.5
        greenSlider.value = 0.5
        blueSlider.value = 0.5

        updateColor()
    }
}
```

The screen should look like this:

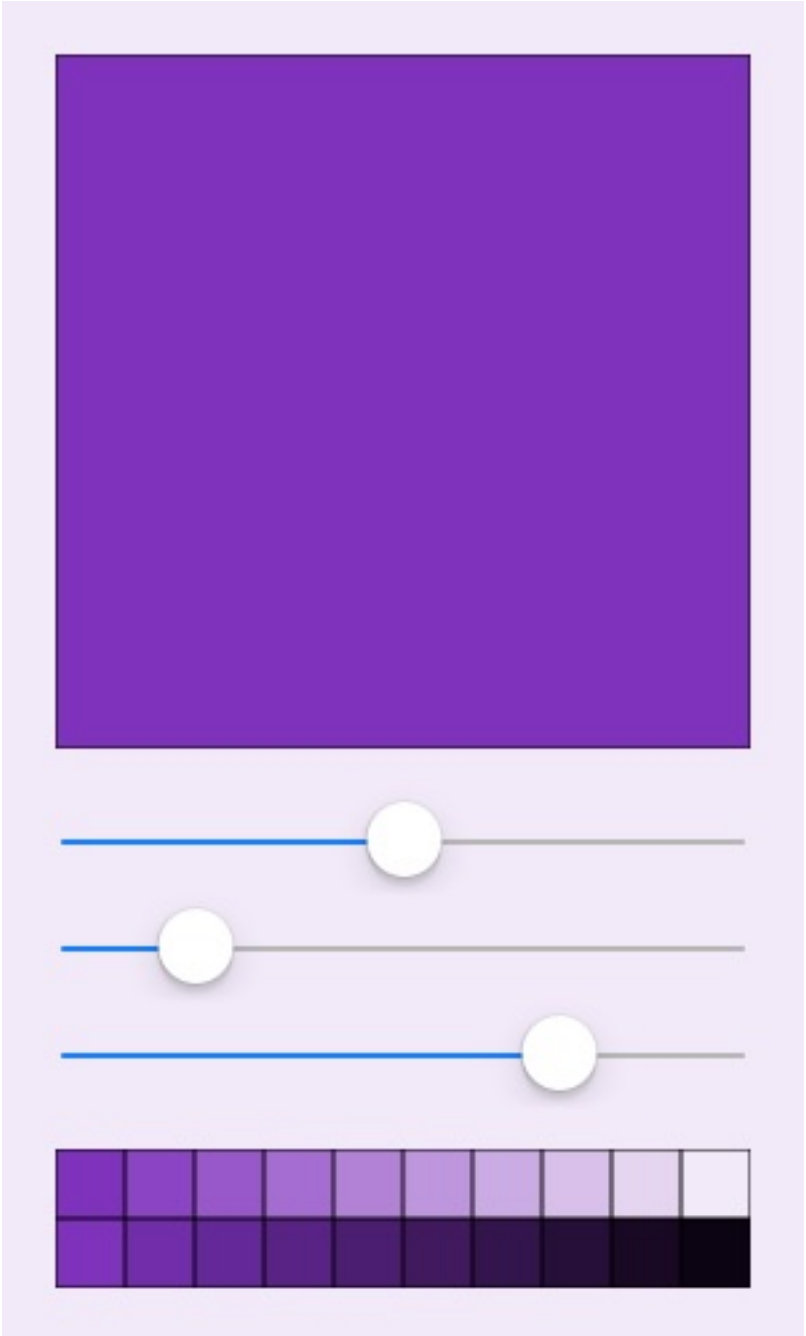


Step 8: A finishing touch

Set the background color of the `ColorPicker` to a really white version of the selected color:

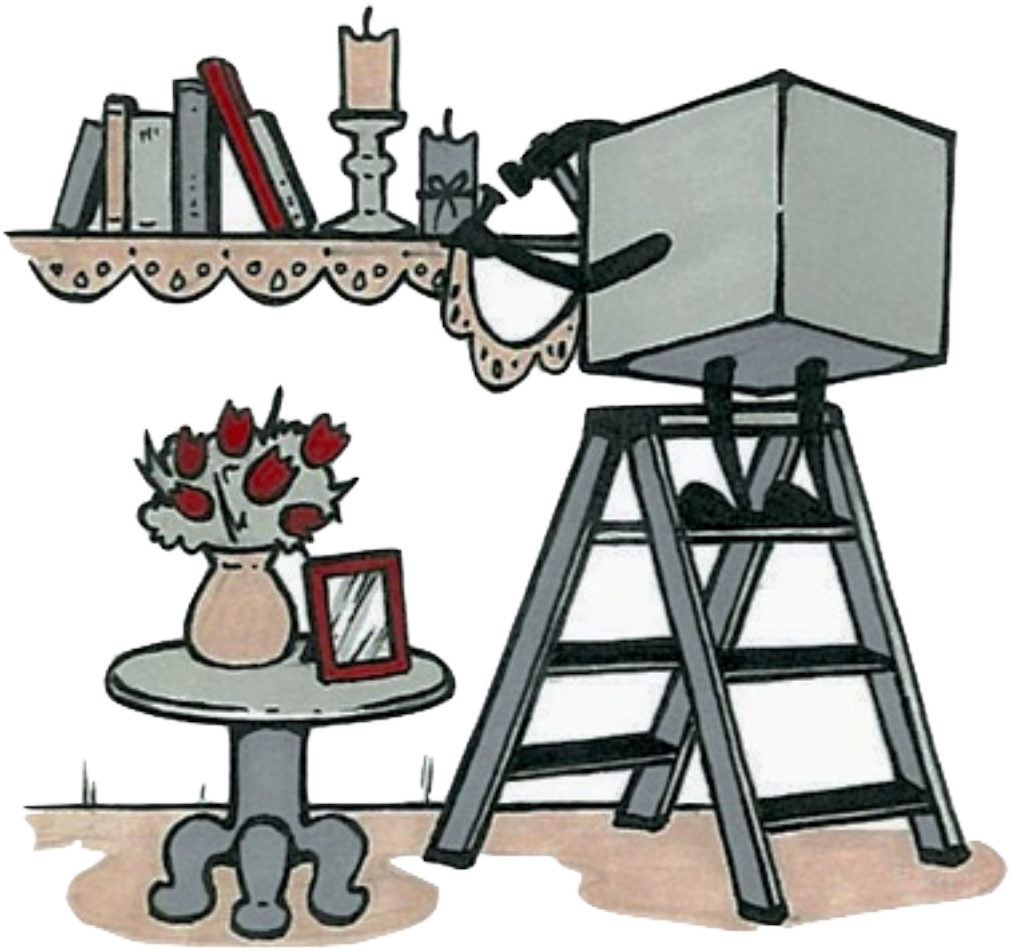
```
class ColorPicker: UIViewController {
    ...

    func updateColor() {
        ...
```

```
            view.backgroundColor =
                color.combine(with: .white, amount: 0.9)
        }
    }
```



Download the complete Playground

Auto Layout



A few years ago you might have gotten away without Auto Layout. Now there are a couple of different iPhone sizes and on iPad you can run two apps at the same time. Auto Layout is the way to go!

This chapter will cover the basics of Auto Layout so that you can start using it in your apps. We are going to go in depth on a few examples so that we cover all the basic types of relations you can have. In the end there are a couple of exercises to practice what you've learned.

What is Auto Layout?

Auto Layout is a system that makes it easy to support multiple screen sizes with one interface, by making your interface react to changes. It does this by solving a set of layout **constraints** which describe the interface.

Why use Auto Layout?

Auto Layout makes layout code much simpler to write and maintain, also in the most cases you don't even have to write code. That means less time writing code and debugging it.

It's also easier to understand constraints than the other approaches because in your head you use words not math.

Which one do you understand faster?

```
rightImage.x = leftImage.x + leftImage.width + padding
```

vs:

`rightImage` is `padding` points to the right of `leftImage`

Another reason might be that it is the default on iOS. With only a few exceptions, Auto Layout is the right tool for the job.

Layout constraints

Layout constraints are descriptions of the mathematical properties and relations between views. The `NSLayoutConstraint` class is used to create constraints on both iOS and Mac. All constraints have a coefficient and a constant.

There are several types of constraints:

- size constraints - *ex. an image should have a width of 200*
- alignment constraints - *a label should be centred vertically on the screen*
- spacing constraints - *space between two labels or between a view and the margin of the screen*

What's the purpose of constraints?

In the end by solving the given set of constraints, Auto Layout will **determine the frame** for each view in your screen. So each view should have constraints that will help determine its `width`, `height`, `x` and `y` position. There are a few exceptions but we are going to get into them later.

Relations to parent

A view can change its size and location based on its superview. But that does not work the other way around.

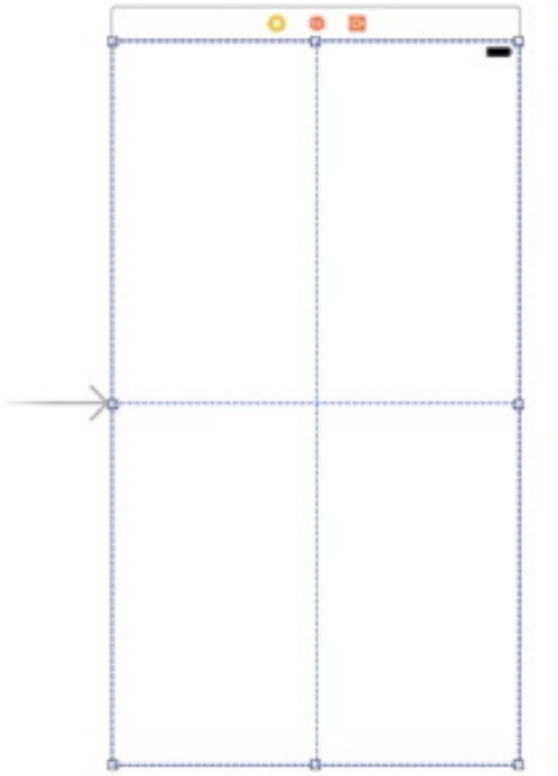
One of the most common things you will do is making a view fill its parent.

Let's have some fun! :)

Download the started project and open `Main.storyboard`.

Starter Project

Find the initial view controller and add a view from the `Object Library`. Resize the view so it fills the screen. You should see some helper lines when you have the correct size.

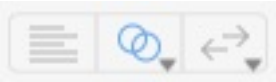


> The initial view controller is signaled using an arrow

Give it a background color so you can see it. I made it red.

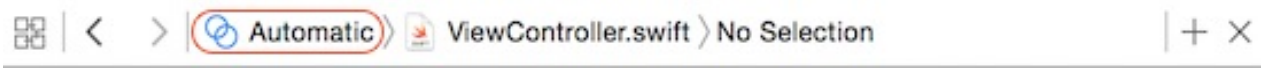
Ok that how it looks on an iPhone 7... how will this interface look like on other devices?

Xcode has a tool for this! Open the `Assistant Editor` from the segmented control in the top right corner of Xcode.

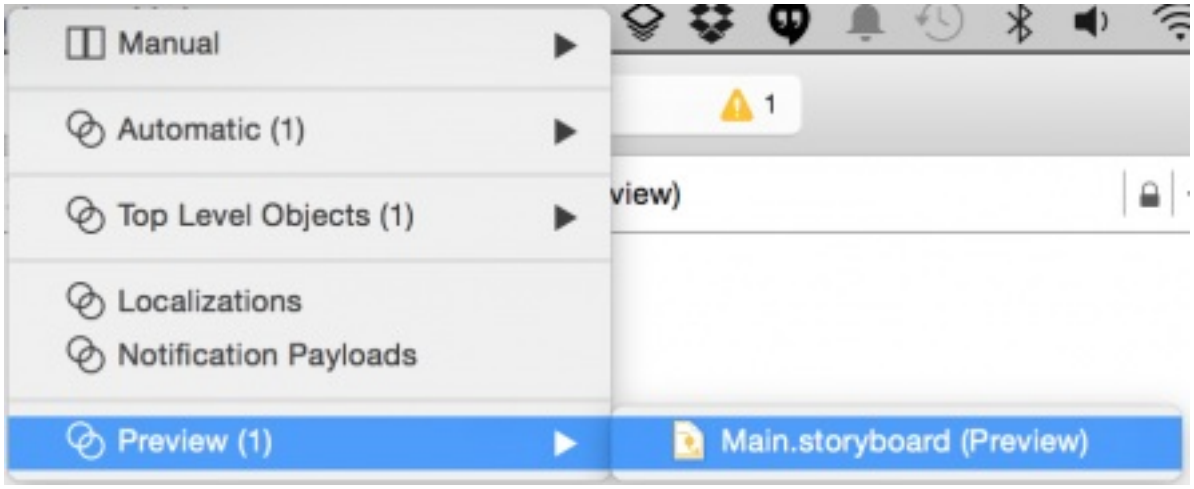


The default will open the swift code of the view controller you where editing.

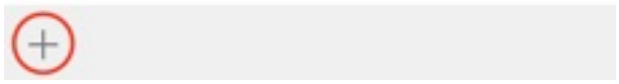
In the top of the `Assistant Editor` you will find this menu. Click on Automatic.



You will see this menu.

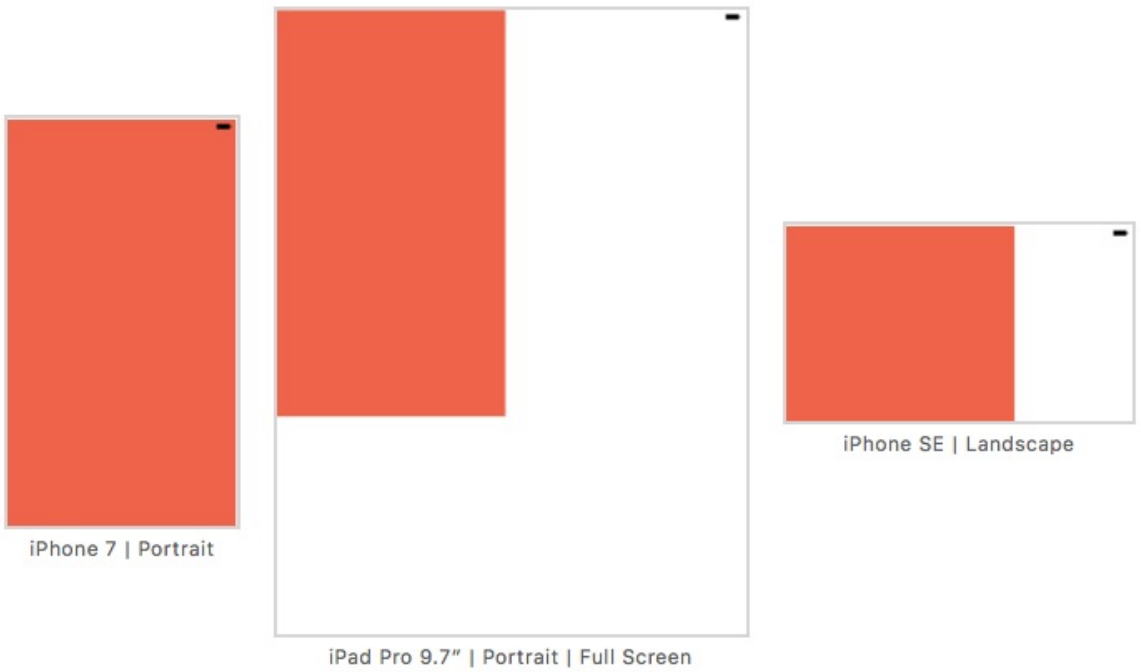


Select `Preview > Main.storyboard` to see a preview of your interface.



You can add more devices from the `+` button in the lower left corner.

Add a few devices.



Now we see that our interface has a few problems in landscape mode and on iPad.

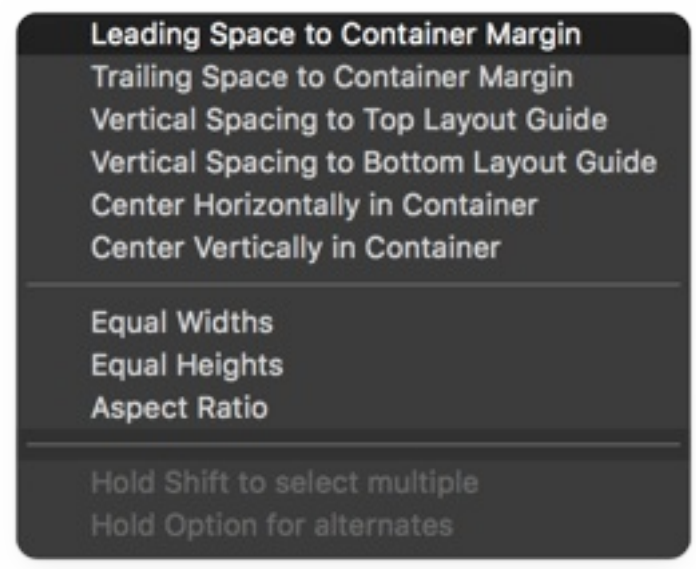
If you feel like you need more space when working on your interface with `Preview` opened, you can close the `Navigator` from the control on the top

right corner of Xcode.

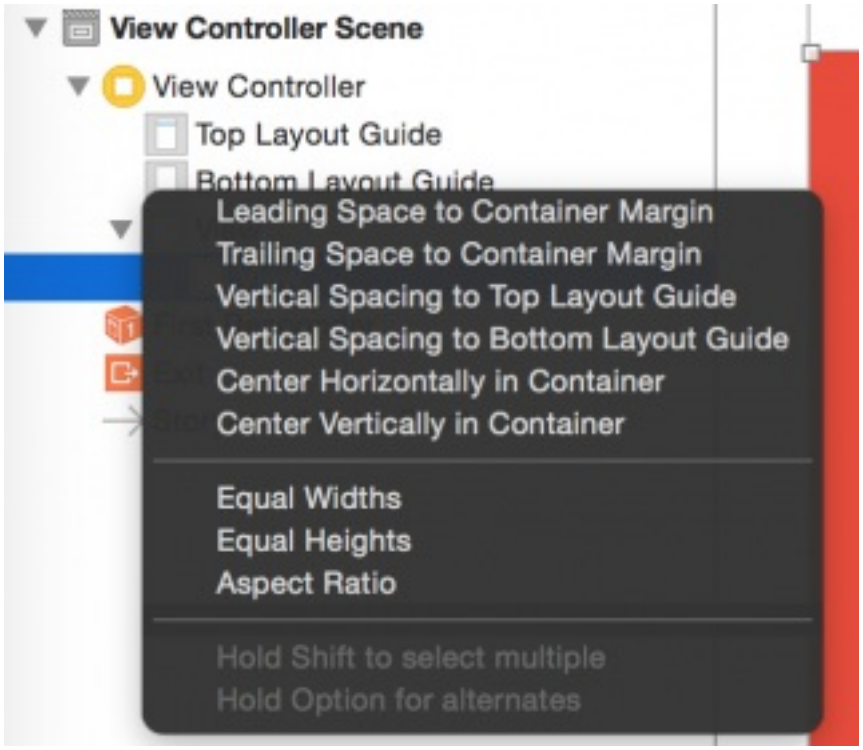
OK. Now that you have enough room, let's start adding some constraints.

The easiest way to add a constraint is by control dragging. Open the `Document Outline` in `Interface Builder` from the button on the lower left corner.

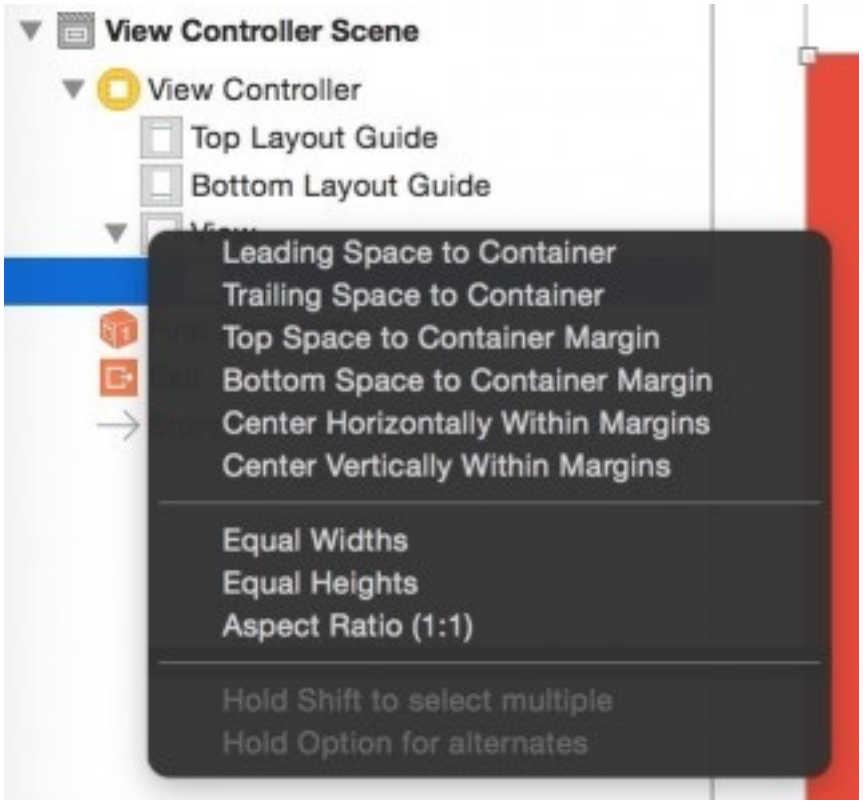
Hold the control key `^` and then click drag from the red view to its superview.



You should see a list of relations you can set.

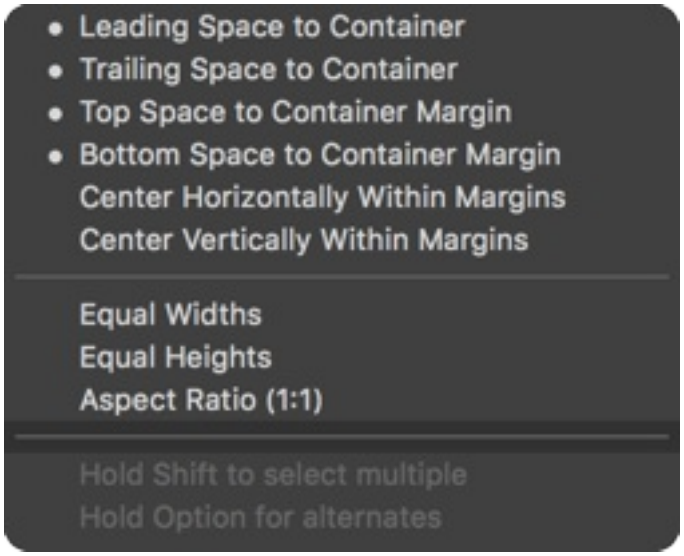


Hold alt ⌥ to change the available constraint options. All the constraints are the same. The only thing that differs is whether they are relative to the margin or not. We are going to cover margins later in this tutorial.

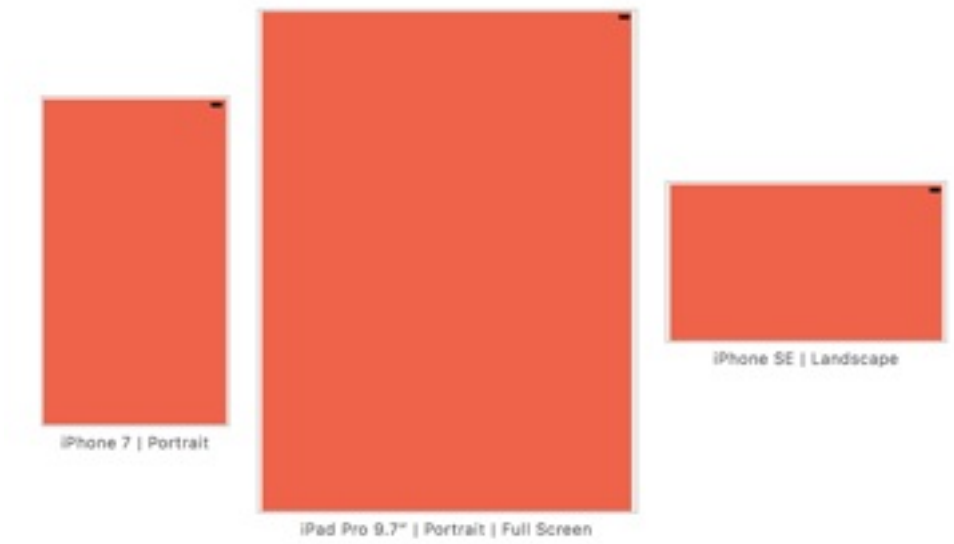


Add the following constraints:

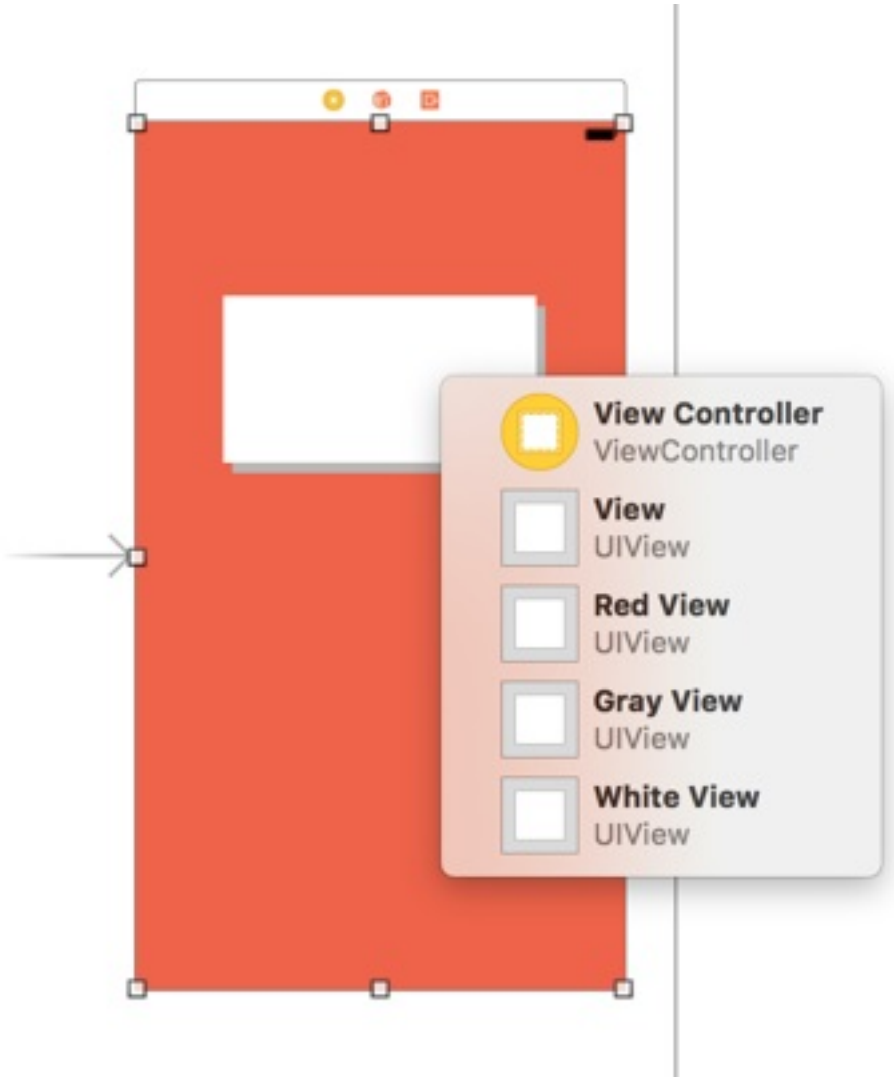
- Leading Space to Container
- Trailing Space to Container
- Top Space to Container Margin
- Bottom Space to Container Margin



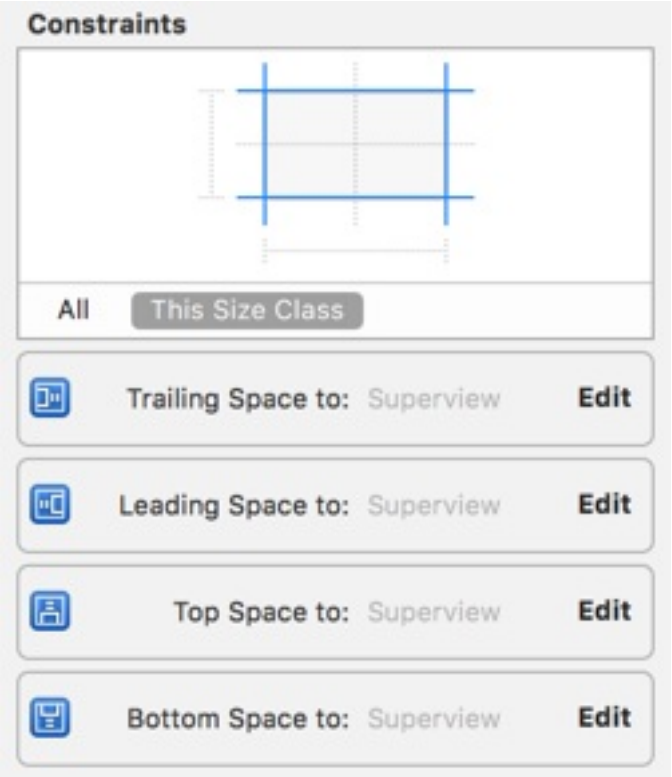
Now the interface should work on all devices.



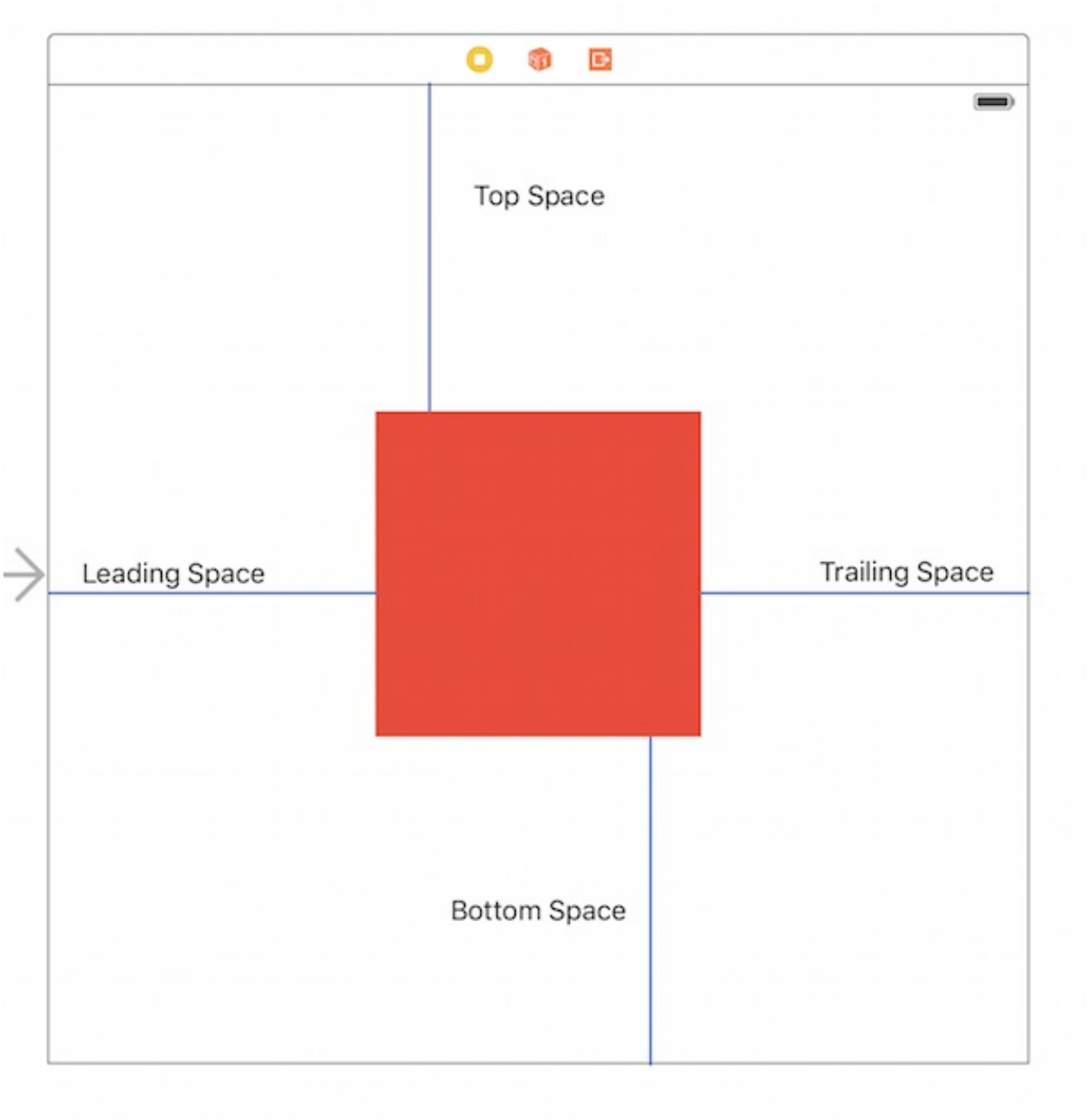
Quick tip

> You can control-drag directly on the interface. I recommended using the `Document Outline` to have better precision. You will notice that when you control+click drag on the interface, there will be different options based on the direction you are dragging in - it will take you to vertical or horizontal properties. If you can't click on a view because there is more than one at the same location, you can shift+control click on any point and you will see the list of views that are located at that point.
>
> 
>

You can inspect the constraints you added from the `Size Inspector`.



What's *Leading/Trailing Space to Container*?

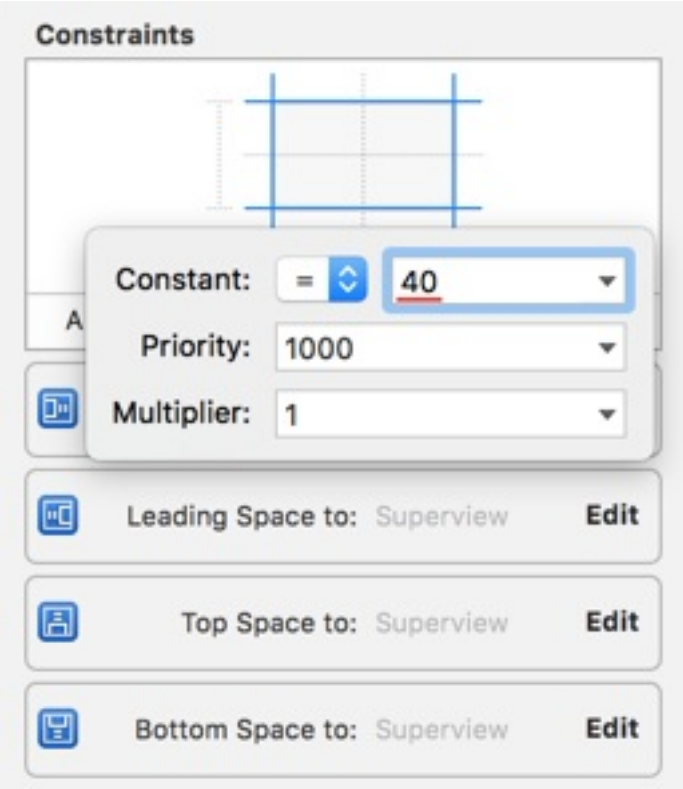


It's the space between the view and its superview. `Leading` is on the left. `Trailing` is on the right. In the example we did before, these were all set to `0`.

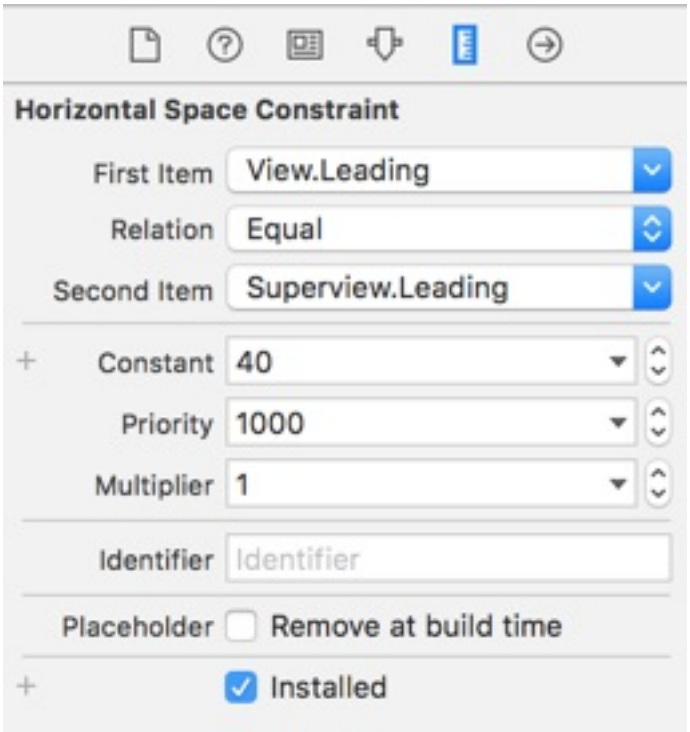
How can I change that? What if I want to leave some empty space?

Select the leading space constraint. You can find it by selecting the view and going to the `Size Inspector`. Click on the `Edit` button and set the constant to `40`. You will notice that the view will resize according to the new constraint. This only happens when you edit a constraint. When you

change the frame of the view you will get a layout warning or error.



Double click on the constraint and take a closer look.



Before we mentioned that constraints describe the mathematical properties and relations between views.

Can you guess what relation the Leading Space constraint describes?

```
View.Leading = Superview.Leading + 40
```

One thing we need to understand is that the constant will change its sign when we reverse the first and second item in order to maintain the same relation.

If we reverse the order we get:

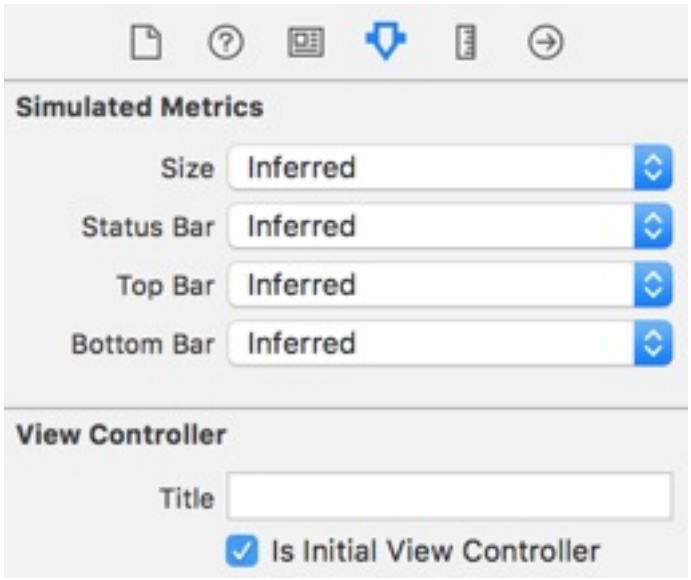
```
Superview.Leading = View.Leading - 40
```

Ok. Let try another one :)

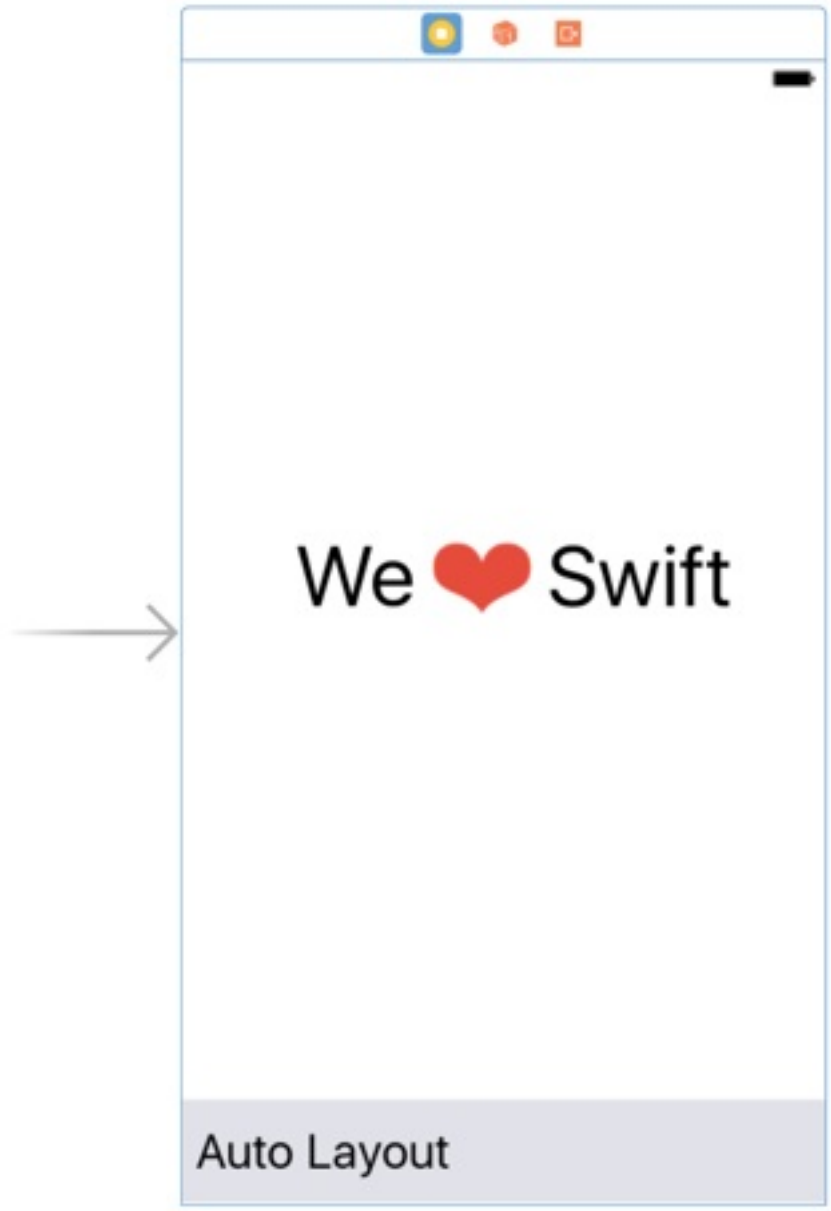
In the storyboard there should be another screen. Make that one the initial view controller.

Select the view controller object in the `Document Outline` pane. Or by clicking the ViewController symbol:

Check the `Is Initial View Controller` checkbox in the `Attributes Inspector`.

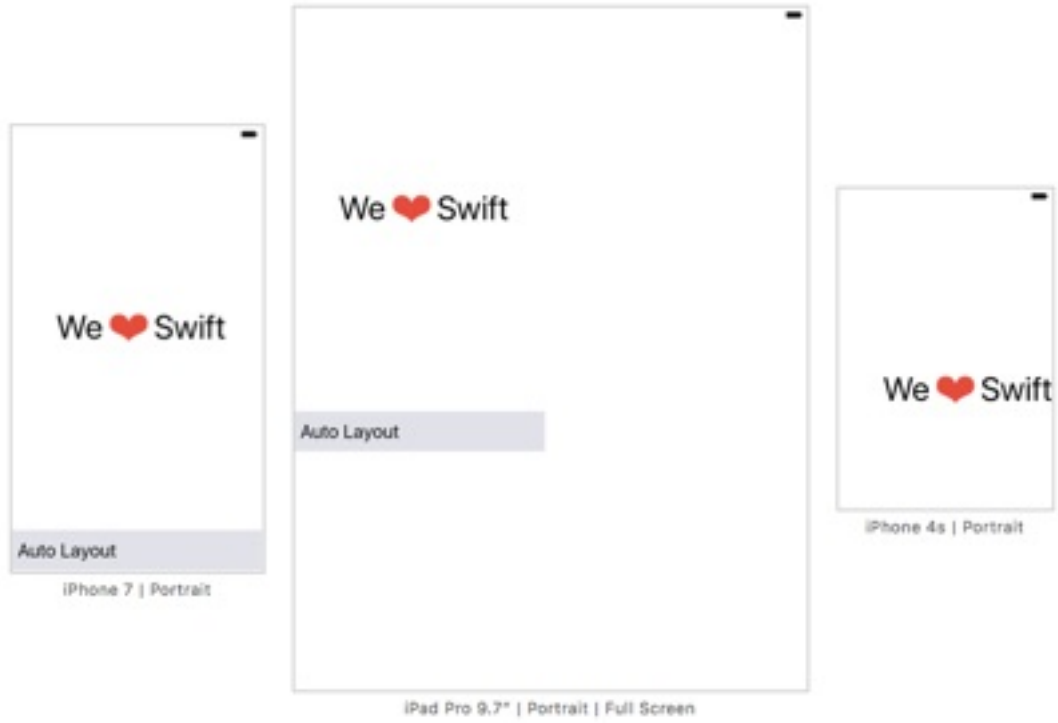


The arrow should point to the other screen now.



Our first goal is to make the `BottomContainer` stick to the bottom of the screen and fill up the width of the screen. `TitleLabel` should fit inside.

Open `Preview` to see how this interface will behave.

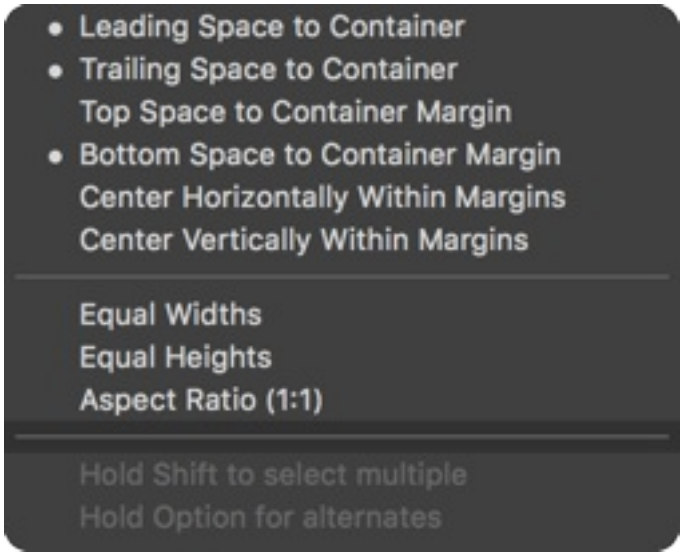


Not even close. Let's handle the container first :)

Can you guess which constraints are we going to use?

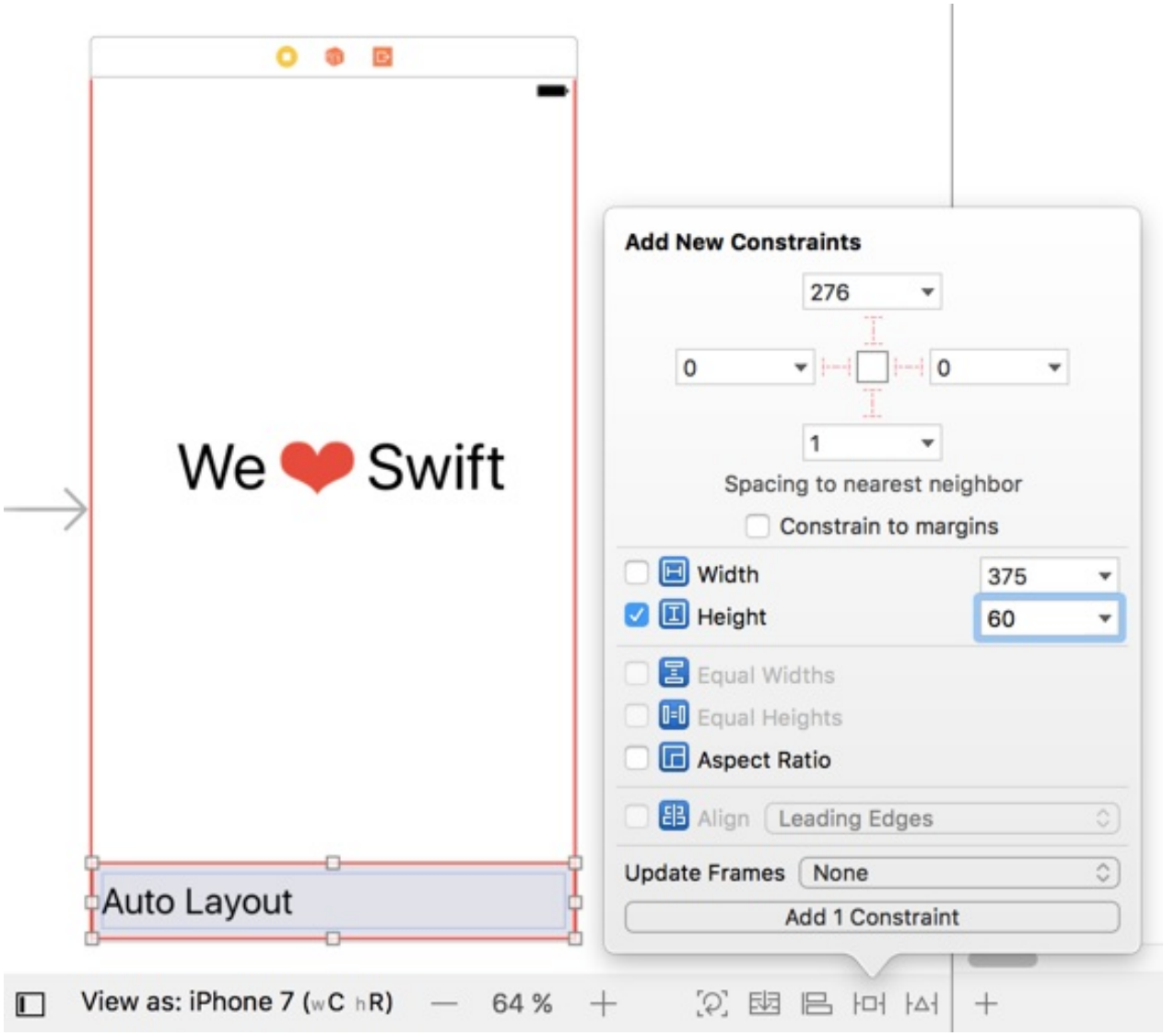
Making the container stick to the bottom and fill up the width of the screen is pretty similar to making it fill the screen. Instead of adding a Top Space constraint we are going to add one for the height of the container.

First add the constraints for leading, trailing and bottom space. Remember to hold alt `⌥` to change the options.



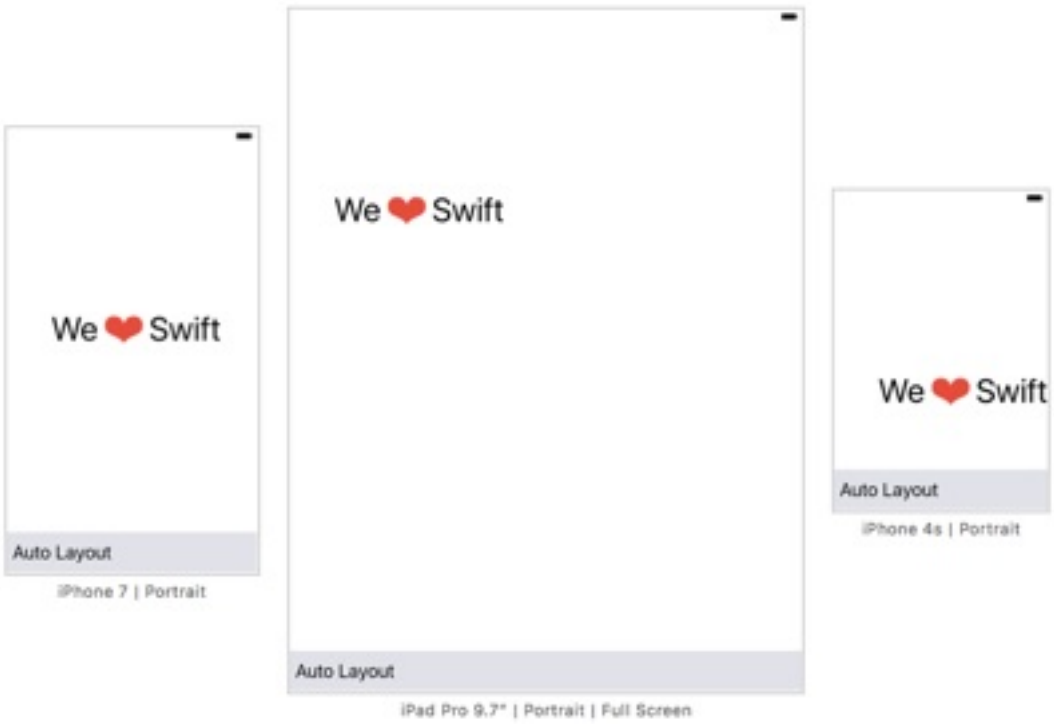
Before we added constraints that represented relations between two views. In order to set the height of the container we only need to `Pin` the value of a property.

Select `BottomContainer` and click on the `Pin` button. Check `Height` and then add the constraint.



> Notice the red borders. They signal a missing constraint or an error. After adding the height constraint on, they should be replaced by blue borders - which means that the view has enough constraints and that they are satisfied. If a view is misaligned, a yellow dotted frame will show the position and size indicated by its constraints.

If we look at the preview now we see that we are getting closer to our goal.

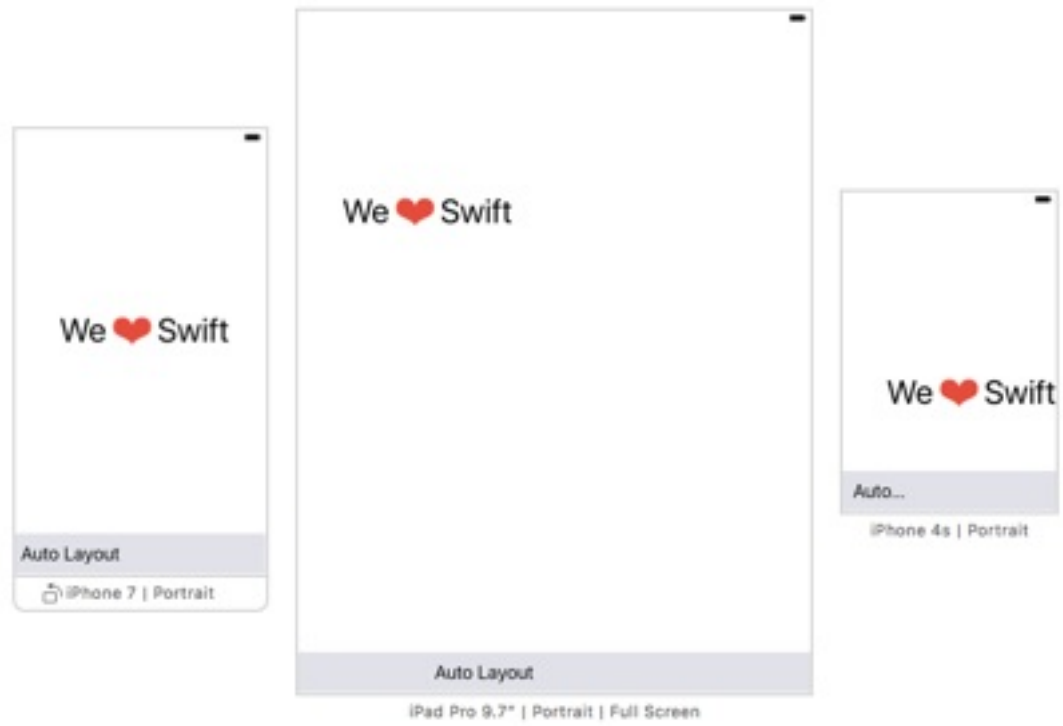


Inequalities

Let's make the `TitleLabel` fit inside the container. Add constraints for leading, trailing, and top space from `TitleLabel` to its container:

You only need 3 constraint because the label gets its height from the font size.

If we look at the preview now we see that we are getting close:



The label fits inside the container. But there is not enough room for the text on smaller screens. We don't want all that empty space. Instead we want the trailing space between the label and container to be **at least** 20 points.

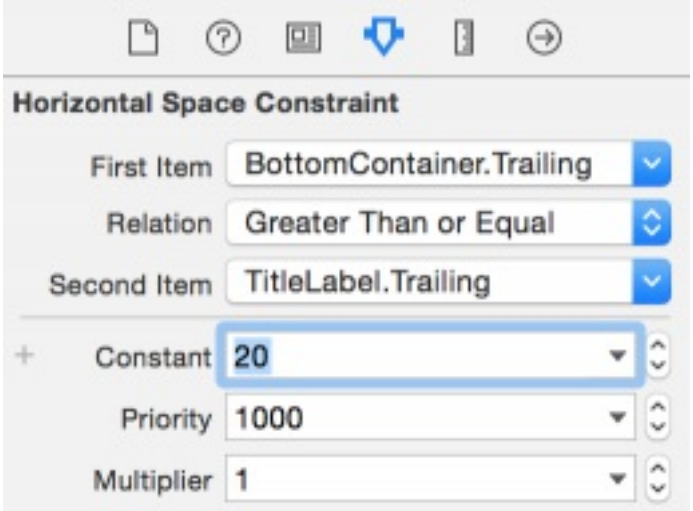
Constrains don't only express equality relations. You can also say that a certain attribute should be greater than or less than another one.

By default when you add a constraint in `Interface Builder` it creates a constraint with an equality relation.

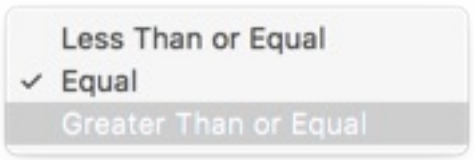
Let's change the trailing constraint to an inequality. First select the constraint.

Go to the `Attributes Inspector` and change the relation from `Equal` to `Greater Than or Equal` and the constant to `20`. You might have the `First Item` and `Second Item` reversed. In that case set the relation to `Less Than`

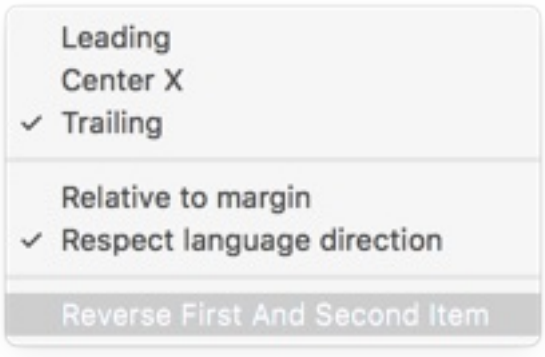
`or Equal` and the constant to `-20` or reverse the order from the `First/Second Item` dropdown.



> Constraint after update



> dropdown shown when changing relation type

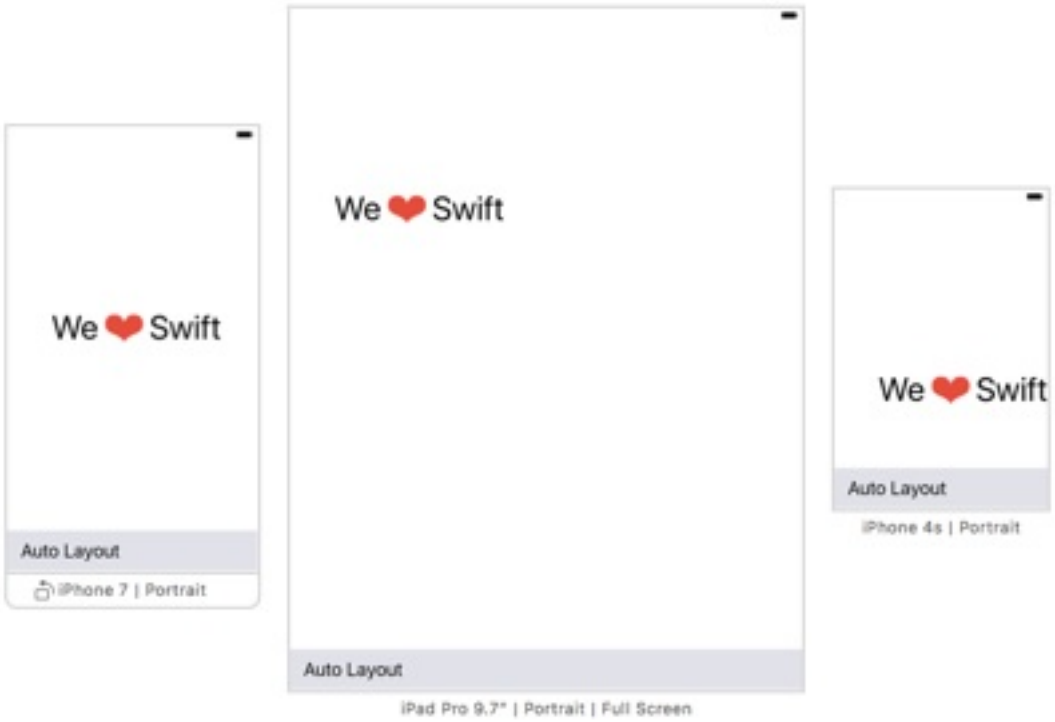


> dropdown shown when clicking on the First Item. The last option swaps items.

If you look at your constraint now you see that it shows a `≥` sign to indicate

that constraint is an inequality.

Now the title will display properly on all devices.



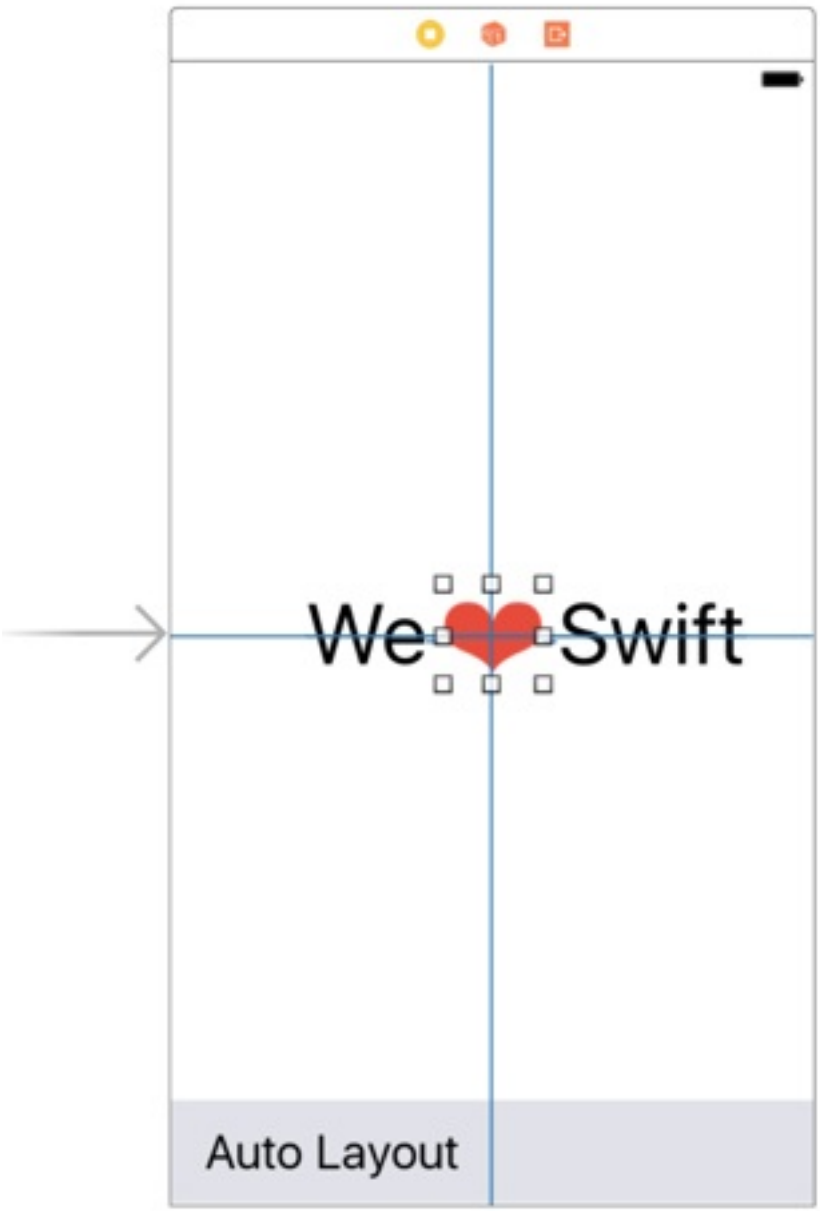
What other relations can be created?

We can add constraints on the layout attributes of view.

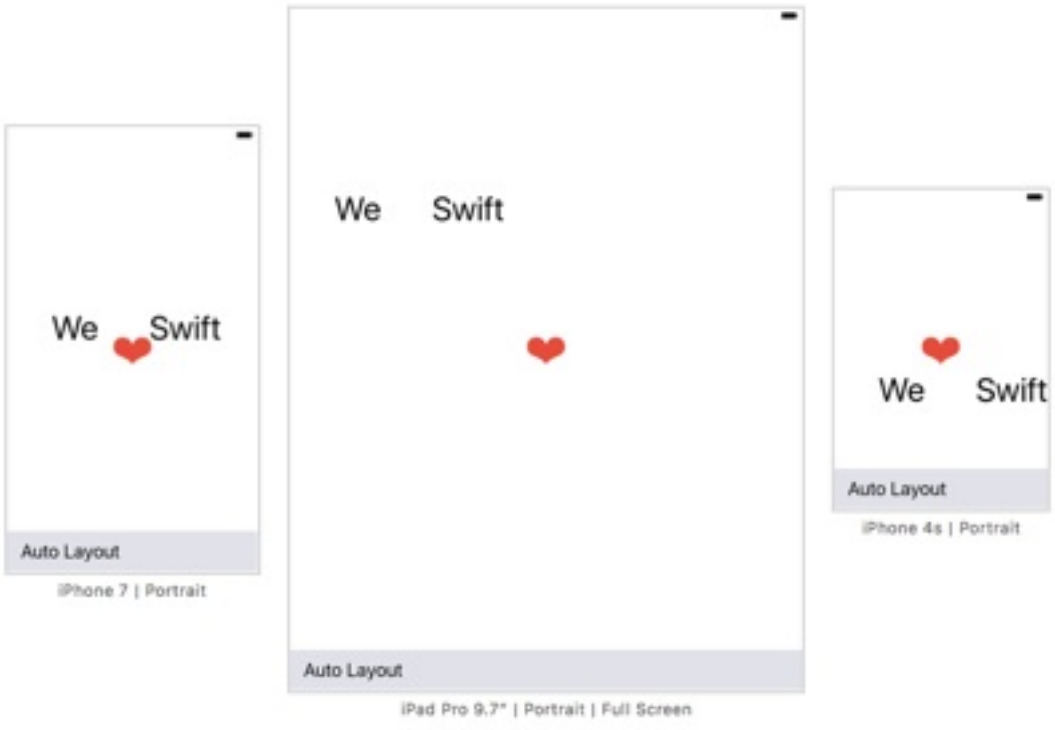
- **size attributes**: width and height
- **alignment attributes**: leading, trailing, top, bottom, left, right centerX and centerY. Left and Right are the same as Leading and Trailing in left-to-right languages - in right-to-left they are reversed.

Relations to sibling

Center the heart using a center horizontally and a center vertically constraint.



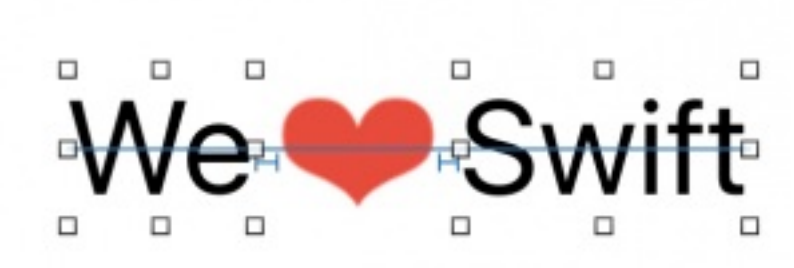
You can see in preview that the heart will show in the center of the screen on all devices. Click on the `Update Frames` button from the bottom bar if you see a yellow indicator.



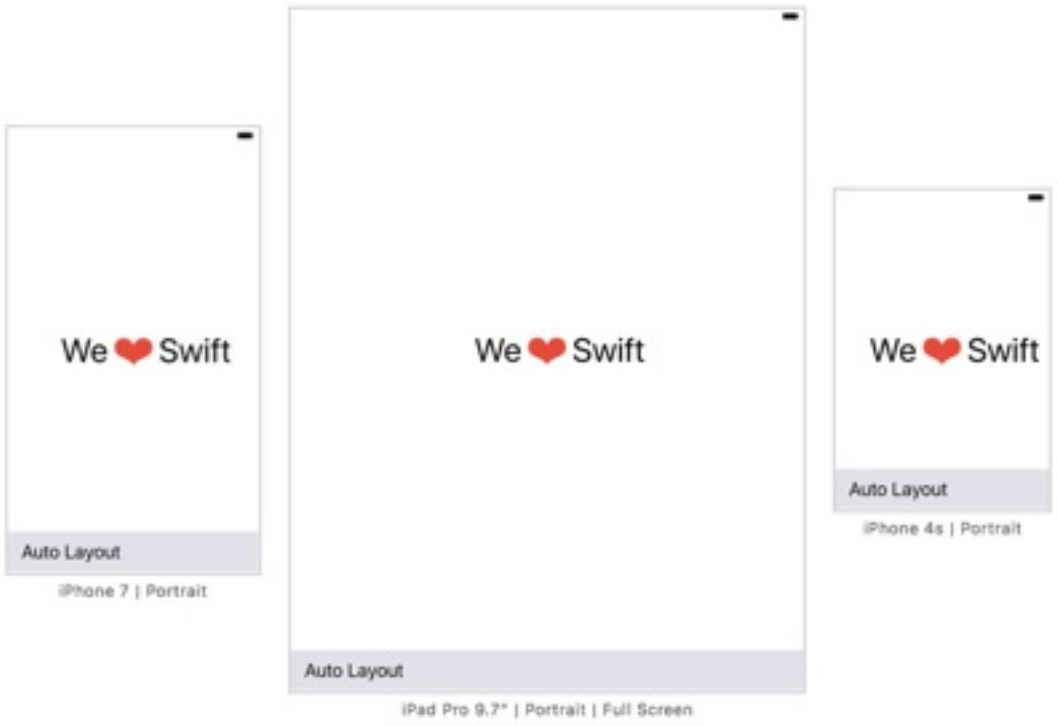
We want to make the `We` and `Swift` labels to be on the same level and height as the heart, with a bit of spacing in between. We can do that! With Auto Layout you can add constraints between views that have a common superview (directly or indirectly).

Move the labels so that they are at the same level with the heart. Control drag from each label to the heart and add a horizontal spacing constraint to set the spacing between the label and the heart.

You will also need an align center vertically constraint to make the label be on the same level with the heart. Add an equal height constraint to maintain proportions.



Now the labels will be on the same level with the heart.



Aspect Ratio

The heart logo a bit too small on an iPad, right?

We can fix that with an *aspect ratio constraint*.

What's that?

So far when we made constraints, we've only used a coefficient of `1` so we ignored it to make things easier to understand. The complete relation that a constraint represents looks like this:

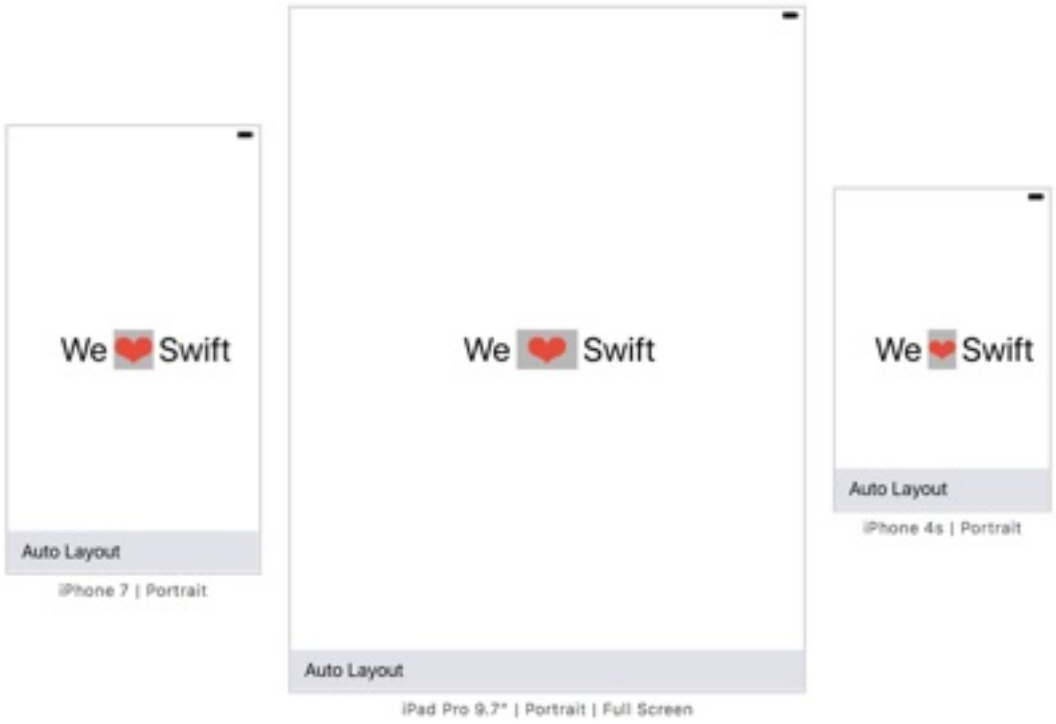
```
FirstItem.attribute1 = SecondItem.attribute2 * coefficient + constant
```

Control-drag from the heart to the screen and select the `Aspect Ratio`

constraint. Go to the `Size Inspector` and double click on it.

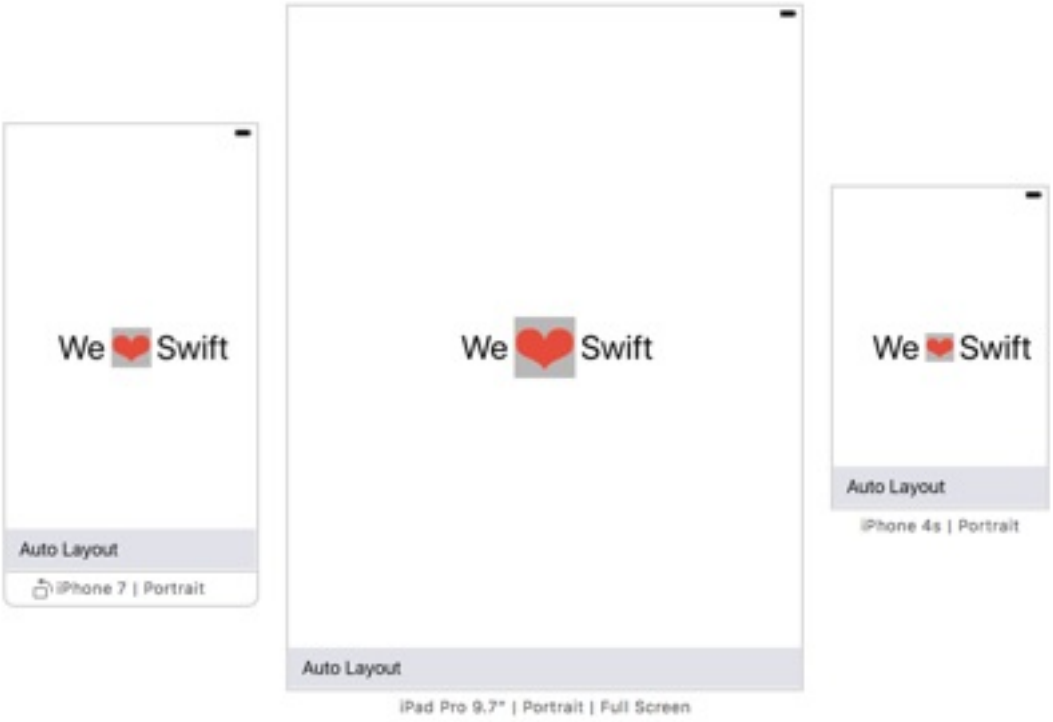
Make sure you have `Heart.width` as the first item and `Superview.width` as the second one. Depending on the direction you dragged you might have gotten different combinations of width and height.

You can see that the constant for this relation is `0` and the coefficient is `60:667` or about `eleven times smaller`. `Interface Builder` lets you express ratios in a more human friendly way - fractions! Because when you see `0.09375` you don't immediately think - "Oh! That's `30/320` ". Fractions are way more intuitive.



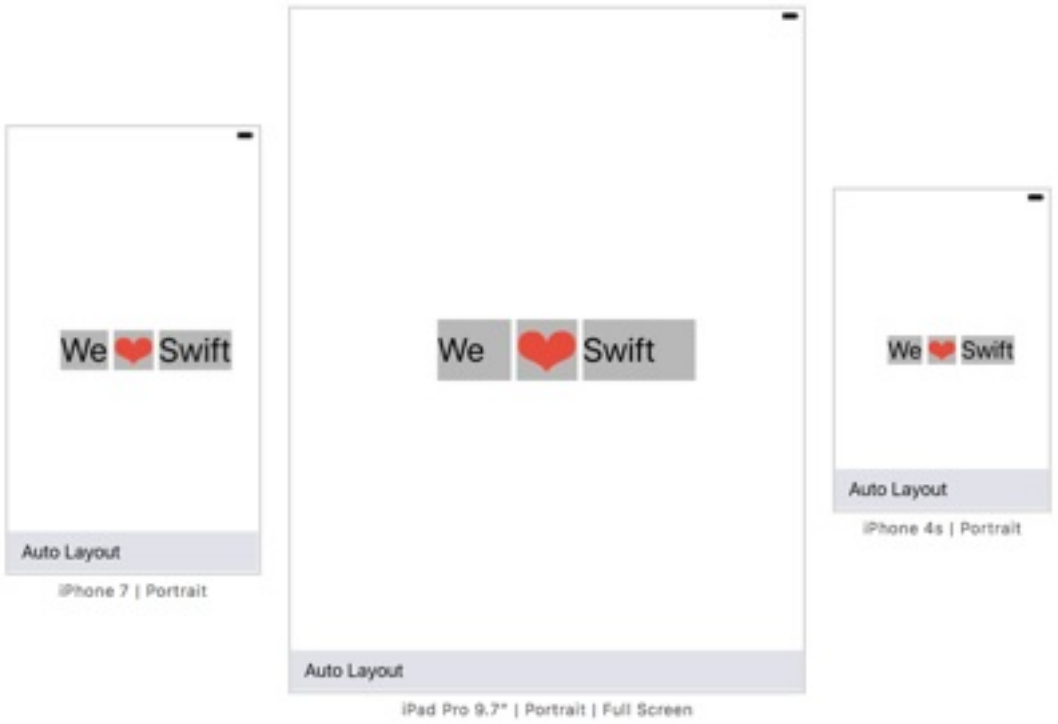
> The gray background was added to make the aspect ratio clear

As you can see, this didn't make the heart bigger - it only made it wider. To make it *grow* we need another constraint. This time from the width of the heart to its height with a ratio of `1:1` - which will make the height of the heart equal to its width, like a square. Control-drag from the heart to itself and select the `Aspect Ratio` constraint.

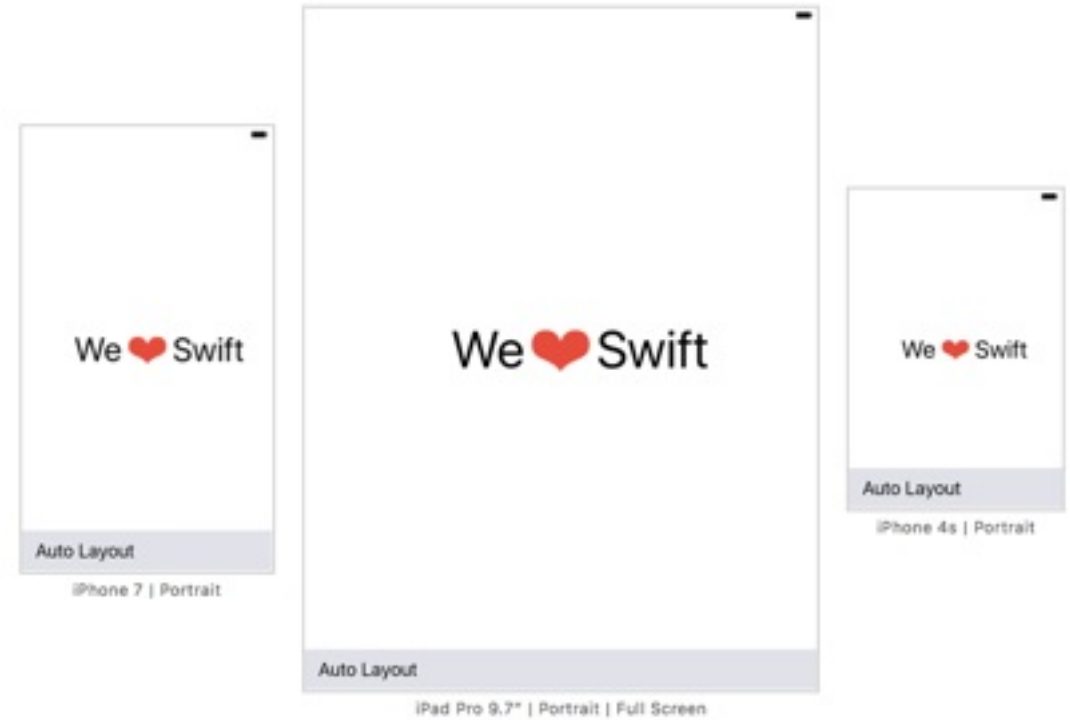


> The gray background was added to make the aspect ratio clear

To make the text be proportional with the heart, we need to add two constraints for each label. One from the label to itself, so it will maintain its aspect ratio. And another one to the heart to fix the height.



One last thing to notice here. On the iPad the text won't grow but the size of the label will. You can fix this by setting the font size to something bigger - like `100`.



Stack View

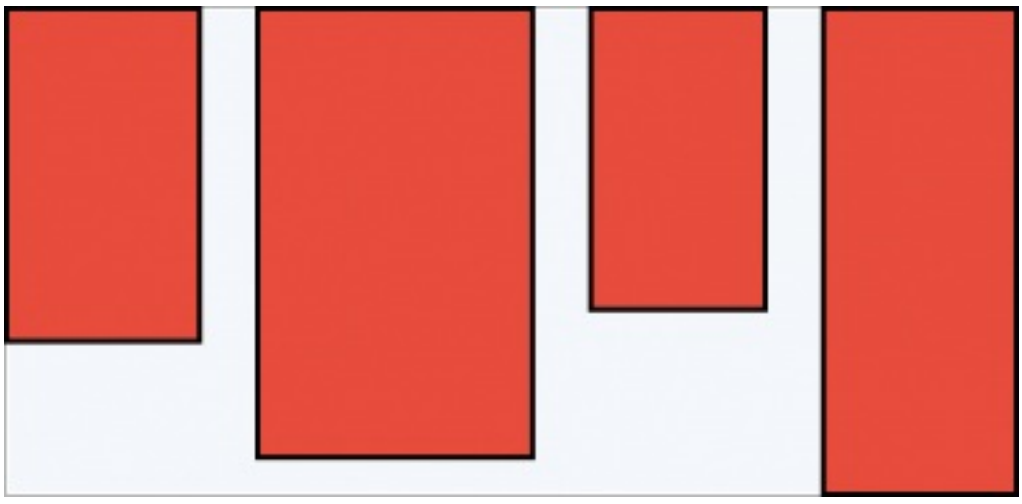
So far so good. We made the We Swift logo with an image and two labels. This approach has a small problem, you can clearly see on small devices that the logo, as a whole, is not actually centred inside the view.

Any idea on how to solve this?

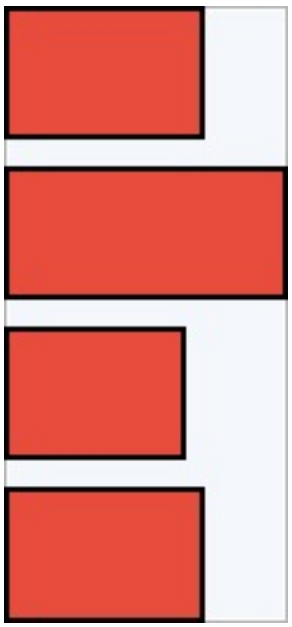
One way would be to move the components into a container and center that container. Starting with iOS 9 we have a special container that makes this kind of layout easy to implement - UIStackView. `UIStackView` uses Auto Layout to arrange its subviews, one after the other, horizontally or vertically. You can customise the spacing, alignment and distribution of subviews.

The alignment style refers to the way the stack view arranges its subviews perpendicular to its axis. Here are a couple of examples of alignment:

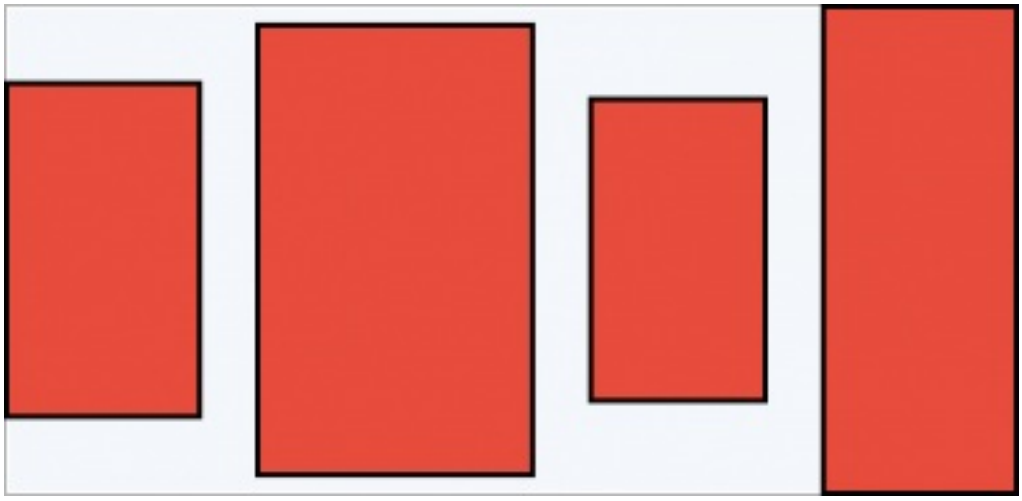
Top

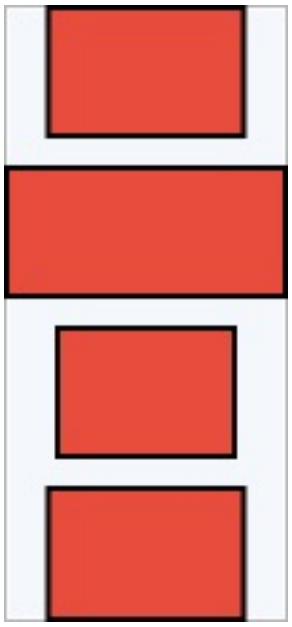


Leading

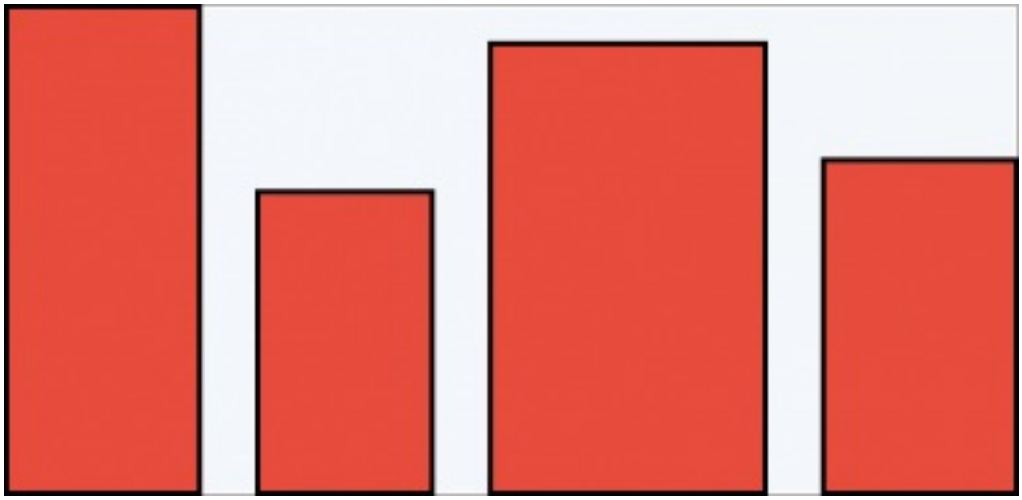


Center

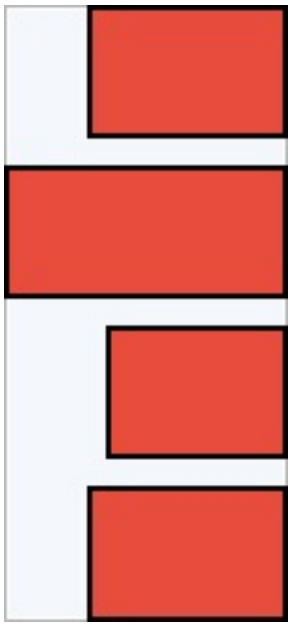




Bottom



Trailing



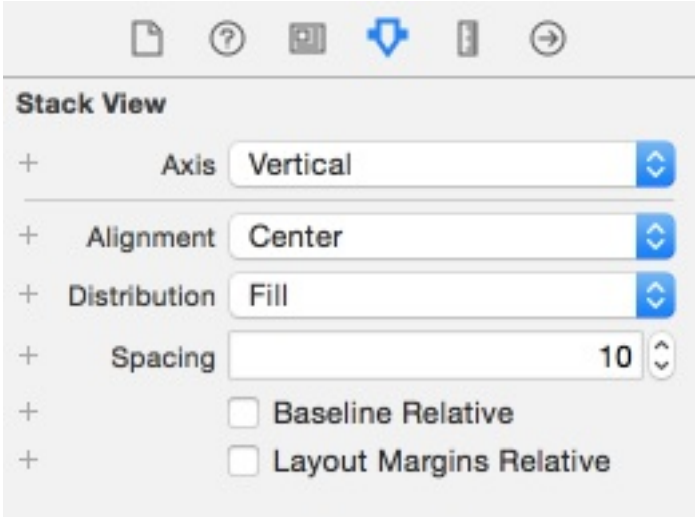
Let's make the main menu for a game using a stack view!

Add a few buttons for the different options from the menu and select all of them.

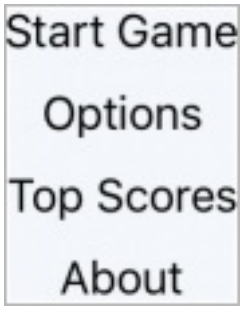
Find the `Stack` button in the lower right corner of `Interface Builder` and press it.

A stack view should wrap the buttons.

Select the stack view and open the `Attribute Inspector`.



By default the spacing is `0` and the alignment is fill. Change the spacing to `10` and the alignment to `Center`.

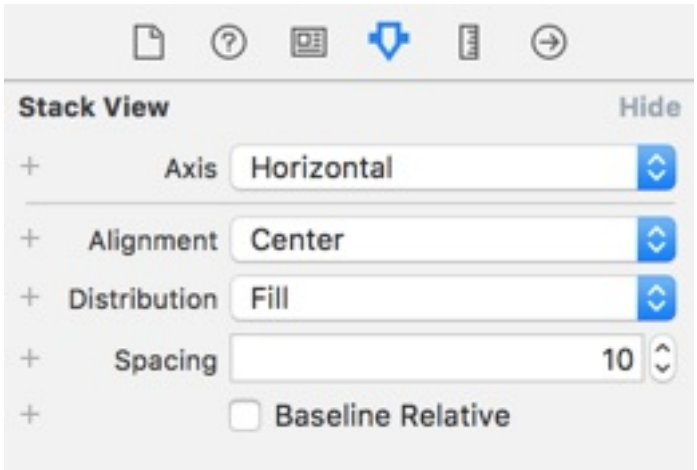


Now let's get back to fixing the We Swift logo.

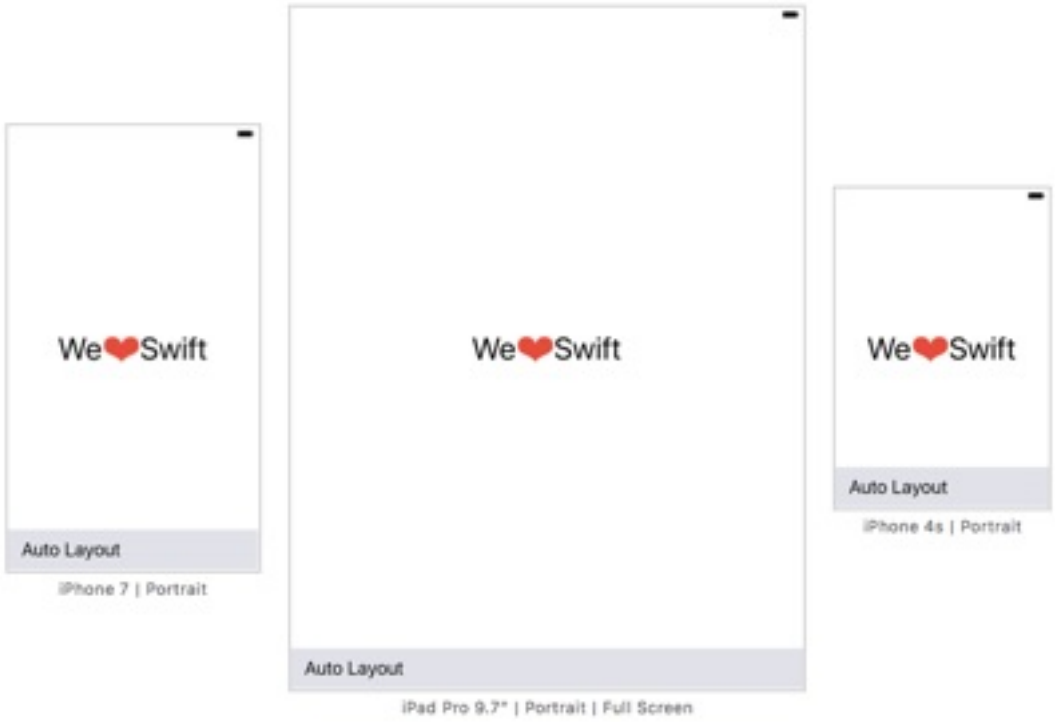
Remove the align center constraints from the heart and the horizontal spacing ones between the heart and each label. You can do this by selecting the constraint and the hitting the delete key.

Select the labels and the heart at the same time and then click on the `Stack` button.

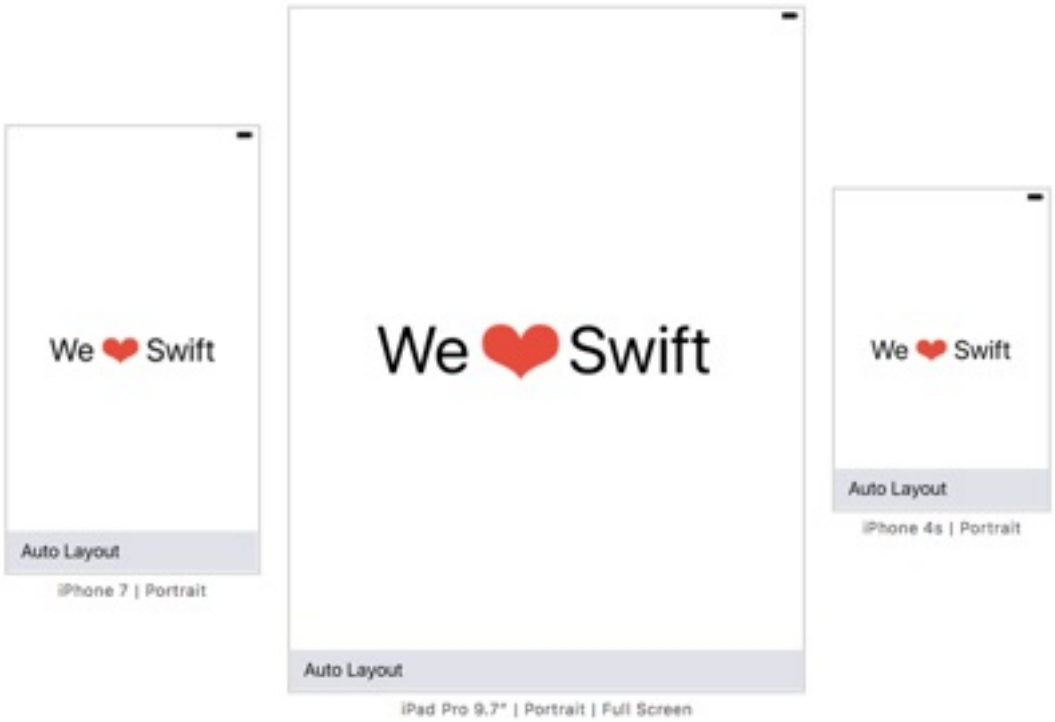
Select the stack view and set the alignment to `Center` and spacing to `10` points.



Add the align center constraints to the stack view. And one more constraint to make the width of the stack view less than the width of the screen.



One more constraint between the stack view and the view for aspect ratio and we are done:



Intrinsic size

You noticed that when you change the font size of a label, the size of the label changes. The same happens when you change the text it contains.

How does it do that?

You guessed it. The label uses some math magic to determine its size and then tells Auto Layout how big it is.

To add support for this on your own view, all you need to do is to override the `intrinsicContentSize()` method. If you do not have an intrinsic size for any dimension you can return `UIViewNoIntrinsicMetric`. The default implementation of `intrinsicContentSize()` returns `UIViewNoIntrinsicMetric` for all dimensions. Some components, like the `UISwitch`, have an intrinsic size for both dimensions, other just for one - like a text field - the height is fixed but the width isn't.

In case your content changes, you can notify Auto Layout to update your view by calling `invalidateIntrinsicContentSize()` on it.

Margins

Sometimes you want to add interface component close to the side of a container, but not starting from the edge. The size of that space will depend on the screen size. On an iPad you will want to leave more space than on an iPhone. Fortunately iOS has a standard margin size defined for each device. To arrange a view relative to the margin of its container, all you need to do is make the constraint relative to the margin.

The default margin on iOS is `{top: 8, left: 16, bottom: 8, right: 16}` on iPhone and `{top: 8, left: 20, bottom: 8, right: 20}` on iPad. To change this you need to set the `layoutMargins` property in code.

```
    container.layoutMargins = UIEdgeInsets(top: 0, left: 20, bottom: 1
```

Why are margins useful?

By using them you can adjust the placement of multiple views inside your container.

Constraints from code

So far we only worked with `Interface Builder` when adding constraints. You can do it from code but in most cases this won't be needed.

There are three ways to create constraints from code:

- create a `NSLayoutConstraint` and set its properties
- use the visual format language to create multiple constraints at once.
- use a library - my favorite is Cartography, it uses operator overloading to make describing layout easy and intuitive and it has over 5000 stars on github.

I'm going to add the same constraints from code in the three different ways. The result will look like this:

NSLayoutConstraint

```
let leadingConstraint =
    NSLayoutConstraint(item: self.redView,
                       attribute: .leading,
                       relatedBy: .equal,
                       toItem: self.view,
                       attribute: .leading,
```

```
                        multiplier: 1,
                        constant: 50)

let trailingConstraint =
    NSLayoutConstraint(item: self.redView,
                        attribute: .trailing,
                        relatedBy: .equal,
                        toItem: self.view,
                        attribute: .trailing,
                        multiplier: 1,
                        constant: -50)

let topConstraint =
    NSLayoutConstraint(item: self.redView,
                        attribute: .top,
                        relatedBy: .equal,
                        toItem: self.view,
                        attribute: .top,
                        multiplier: 1,
                        constant: 50)

let bottomConstraint =
    NSLayoutConstraint(item: self.redView,
                        attribute: .bottom,
                        relatedBy: .equal,
                        toItem: self.view,
                        attribute: .bottom,
                        multiplier: 1,
                        constant: -50)


let constraints = [
    leadingConstraint,
    trailingConstraint,
    topConstraint,
    bottomConstraint
]

self.view.addConstraints(constraints)
```

Visual Format

```
let bindings =
    Dictionary(dictionaryLiteral: ("redView", self.redView))
```

```
let horizontalConstraints =
    NSLayoutConstraint.constraints(withVisualFormat:
        "H:|-50-[redView]-50-|",
        options: [],
        metrics: nil,
        views: bindings)
self.view.addConstraints(horizontalConstraints)

let verticalConstraints =
    NSLayoutConstraint.constraints(withVisualFormat:
        "V:|-50-[redView]-50-|",
        options: [],
        metrics: nil,
        views: bindings)
self.view.addConstraints(verticalConstraints)
```

For more about Visual Format language read this tutorial.

Cartography

```
layout(redView) { view in
    view.edges == inset(view.superview!.edges, 50, 50, 50, 50)
}
```

You can read more about Cartography and how to use it on github.

Animations and Auto Layout

When Auto Layout is enabled you cannot change the view frames directly. Changes on frame, bounds or center, will have no effect. Changes to other properties will not be affected by Auto Layout so they can be animated.

To change the frame of a view that has constraints, you can change the `constant` or `multiplier` of any constraint then call `layoutIfNeeded()` on that view. If you click on a layout constraint in `Interface Builder` you can connect a referencing outlet from the `Connections Inspector`. See the

example below:

```
class ProfileViewController : UIViewController {
    @IBOutlet var profilePictureHeightConstraint: NSLayoutConstraint!
    @IBOutlet var profilePicture: UIImageView!

    ...

    func makeProfilePictureTaller() {
        UIView.animate(withDuration: 0.4) { {
      // change constraint
      self.profilePictureHeightConstraint.constant = 100
      // this will change the frame
      self.profilePicture.layoutIfNeeded()
        }
    }
}
```

Another way to change the frame of a view using Auto Layout, is by changing the constraints that affect that view:

```
UIView.animate(withDuration: 0.4) {
    self.article.removeConstraint(articleHeightConstraint)
    self.view.addConstraint(articleEqualHeightConstraint)
    self.article.layoutIfNeeded()
}
```

Useful Shortcuts

control-drag from any view to add a constraint on it. After you click, a menu will appear. If you press `Option` (`⌥`), the options will change - making them relative or not to the margin.

`⌘` + `⌥` + `=` : rearranges all the selected views based on their constraints, or all views in the current screen, if none are selected.

`^` + `⇧` + `click` : shows you a list of all the objects that are under the cursor in `Interface Builder`.

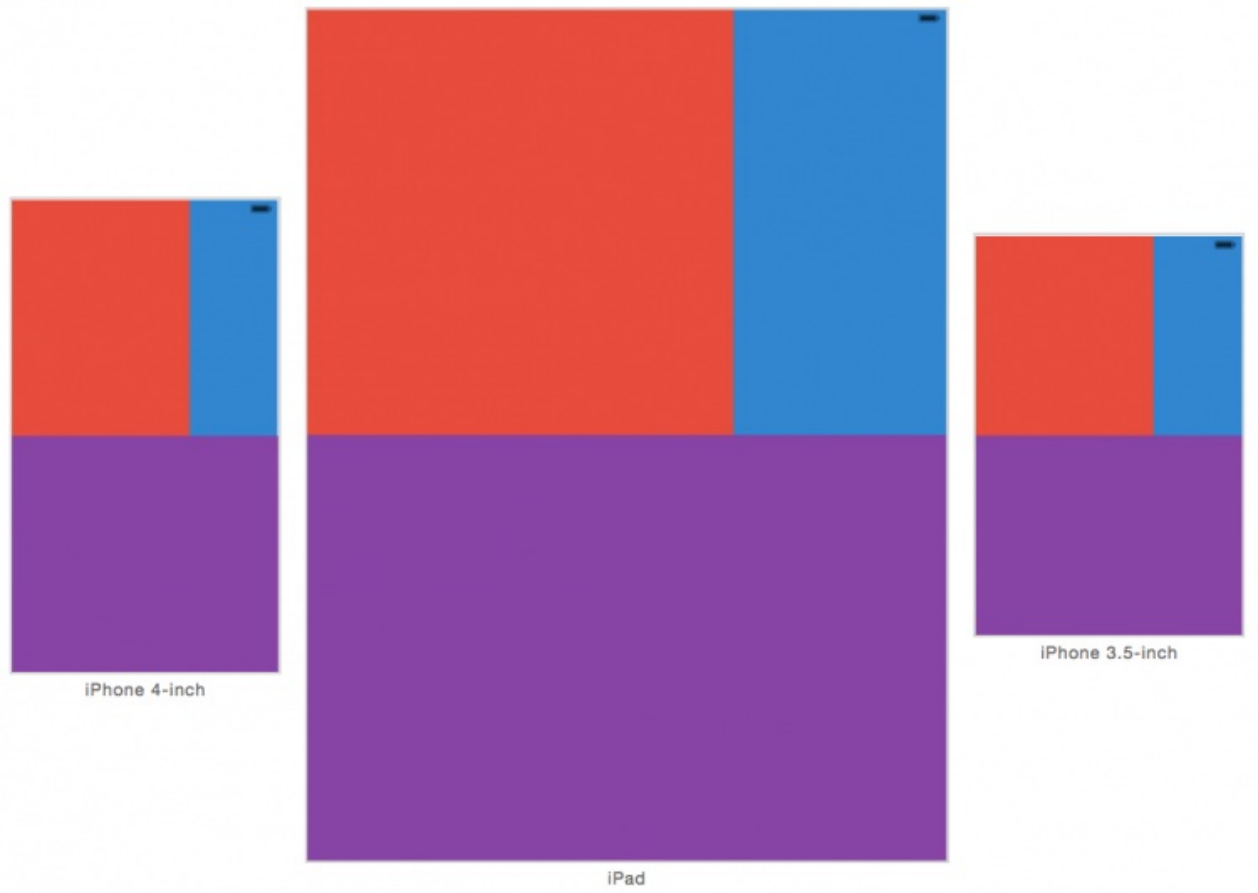
Conclusion

Auto Layout is a powerful tool - and the default layout engine for iOS. Although, most cases can be solved directly from Interface Builder. It's a good idea to understand what happens under the hood.

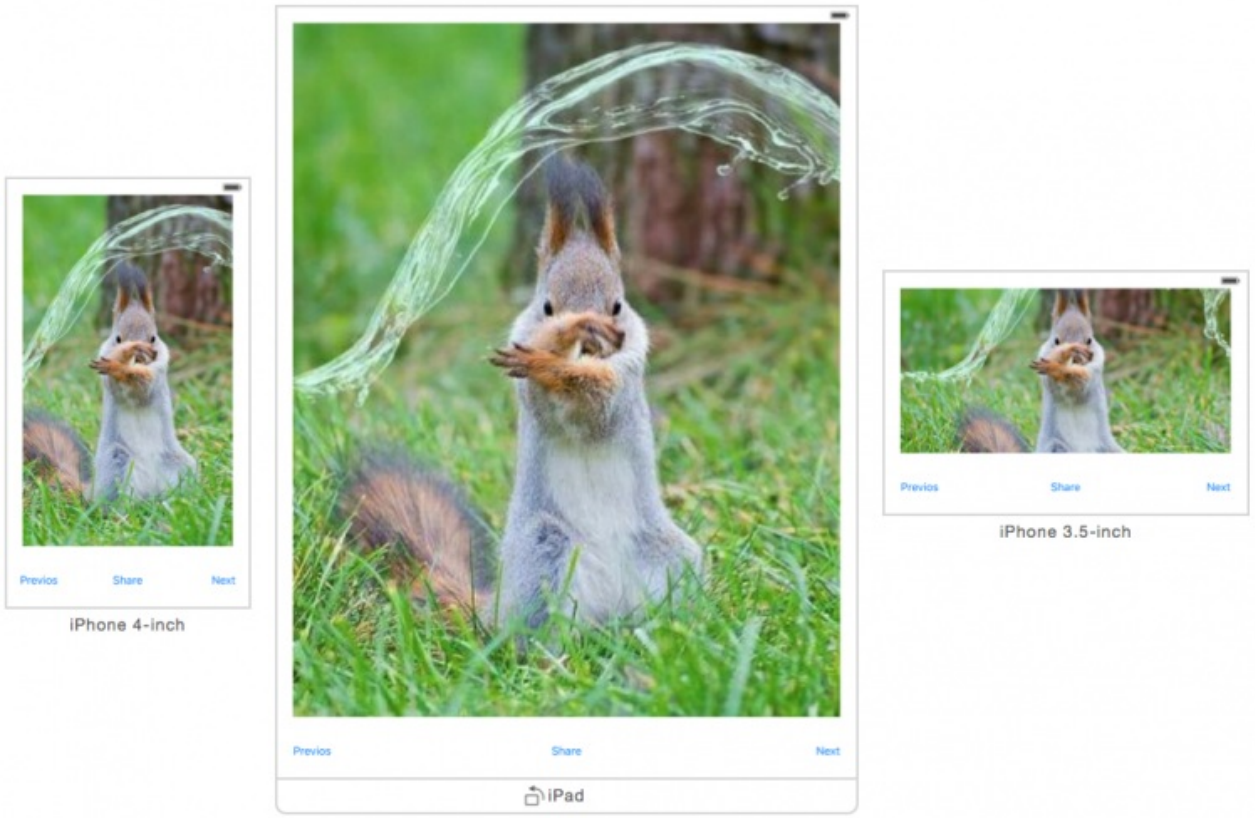
Exercises

All exercises will give you a preview of an interface on a couple of devices. Your task will be to recreate them based on the instructions.

1) The red view should have a width equal to `2/3` of the width of the screen and half the height. The blue view should have a `1/3` of the width of the screen. And the purple view should have half the height and the same width as the screen.

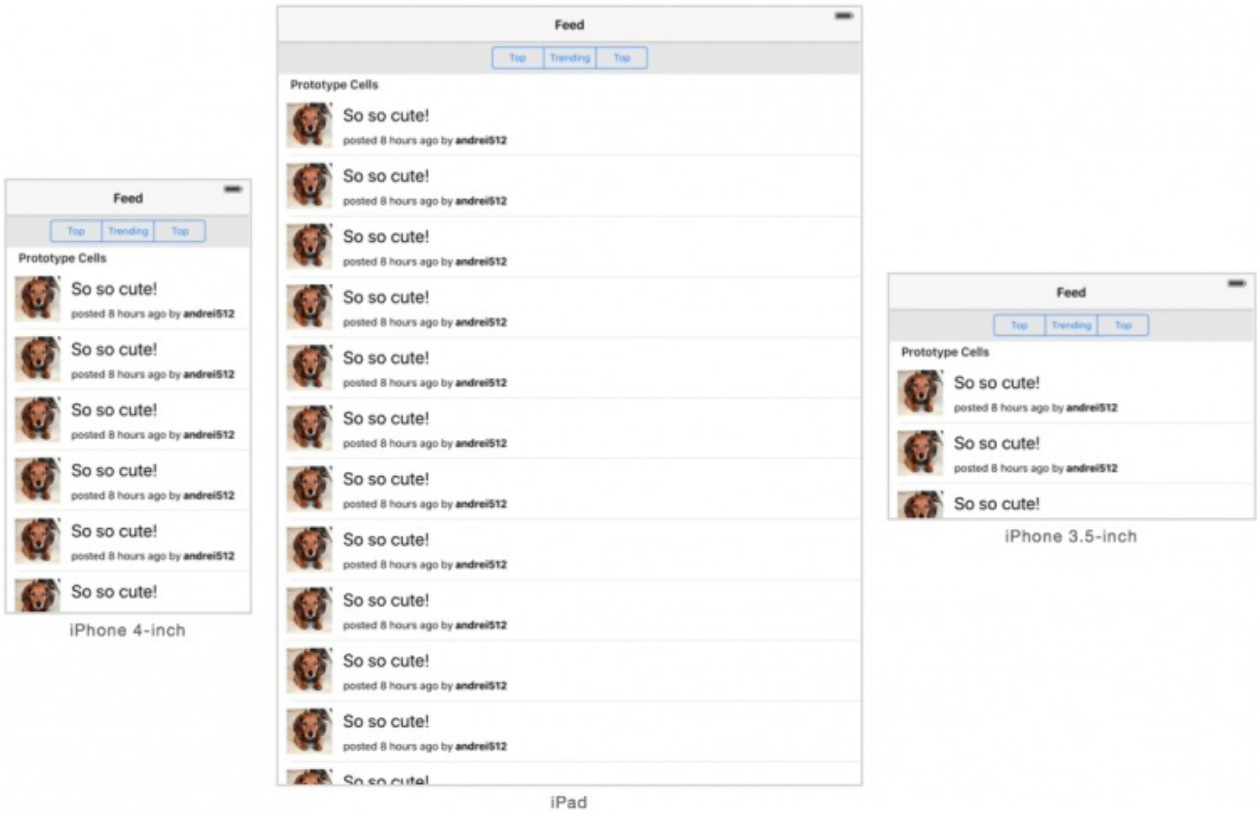


2) This is the view for a random cute image viewer. It has an image view that fills most of the screen and three buttons.



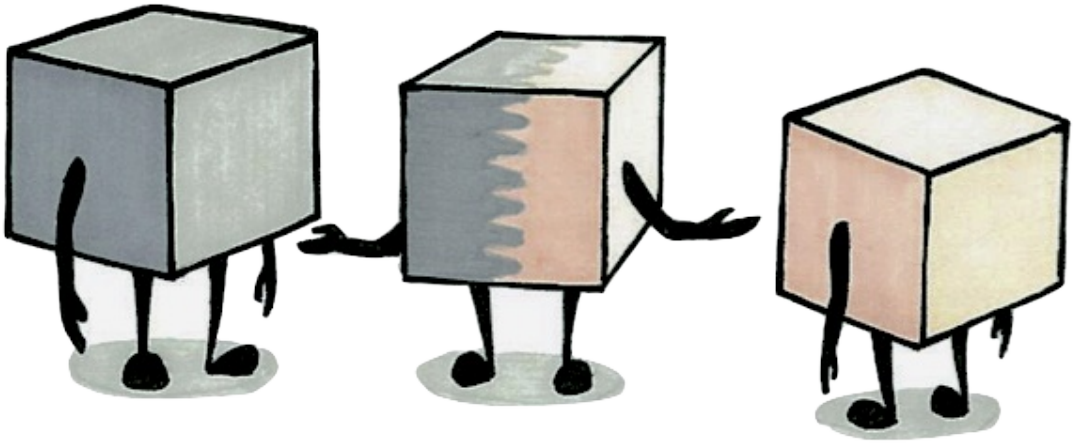
> you can find the kung fu squirrel in the starter project in the asset catalog

3) Under the navigation bar there should be a container. There should be a segmented control centered inside that container. The rest of the screen is taken by a table view.


Feed
Top
Trending
Top
Prototype Cells
So so cute!
posted 8 hours ago by andrei512
iPhone 4-inch
iPad
iPhone 3.5-inch

View Controller and Controller Hierarchy

iOS uses the Model-View-Controller design pattern.



This pattern has two main ideas:

1. Each object has one of three roles: model, view or controller.
    - a **View** is something you can see on the screen - all views are subclasses of `UIView`.
    - a **Model** is a kind of data your app handles. For example an item in a to do list, a location on the map, a user profile, etc.
    - a **Controller** is an object that displays one or more views corresponding to one or more models - all controllers are subclasses of `UIViewController`.
2. Not all kinds of objects talk to each other:
    - **Models** can talk directly with **Controllers**, but not with **Views**. For example a news feed can tell the controller it has new items to display.
    - **Views** can talk directly with **Controllers**, but not with **Models**. Any

`UIControl` (button, slider, text field, switch) can send events to the **Controller** that displays it.
- **Views** and **Models** can exchange messages indirectly by talking with **Controllers**. For example when a button has been tapped the controller is usually notified about the event.

UIViewController

You may have noticed that the class for a controller in iOS is `UIViewController` and not `UIController`. Why? Well, that's because controllers in iOS are a specific kind of controller, one that has a corresponding view which it handles. You can access the view by using the `view` property. That view represents what the controller will display.

Because we know that controllers have a corresponding view, we can implement a default behaviour for all of them. For example presenting a view controller on the screen or handling orientation changes. This allows developers to create a consistent experience across apps.

That means that a lot of the hard work is already done and can be used on all view controllers. To harness this power you will need to understand a few things:

1. The View Controller Lifecycle
2. View Controller events
3. The Navigation Stack

View Controller Lifecycle

The lifecycle of any object involves these 3 steps:

1. The object get created

2. The object is used
3. After the object is no longer needed, it gets destroyed - by that I mean removed from memory

View controllers are similar in the sense that they have these three stages, but each one has some specific things to look out for.

The general story of the view controller goes something like this: somewhere in an app, our hero, the view controller, gets created by another object. At this point the view controller has no view. The creator shows him on the screen and a view gets created for our view controller. After a while, another view controller might be presented on the screen, blocking the view of our view controller - something like a camera screen. That screen is dismissed and our view controller is visible again. After a (short) while our hero gets removed from the screen and dies. The end!

LoadView

When you show a view controller, the first thing that it will do is to create its view, if it didn't before. There are three ways of doing this:

1. In code - this is handled by the `loadView` method.
2. Using a storyboard - if your app uses storyboard, you will ask the storyboard to create the view controller and the storyboard .
3. Using a nib file - similar to how the storyboard works

The thing you need to remember here is that all options in the end call the `loadView` method. You should never call `loadView` in your code directly. The controller automatically calls `loadView` the first time its `view` property is called and its value is `nil`.

```
class ViewController: UIViewController {
    override func loadView() {
        let width: CGFloat = 300
        let height: CGFloat = 500
```

```
    let screenSize =
      CGSize(width: width, height: height)

        let frame = CGRect(origin: .zero,
                           size: screenSize)

        view = UIView(frame: frame)
        view.backgroundColor = .white
  }
}
```

Quick tip

If you want to load the view before it's shown on the screen, you can simply call the `view` property.

ViewDidLoad

You've guessed it! `viewDidLoad` is the method that gets called after `loadView` finishes loading the view. This is a great place to continue your interface setup.

```
class ViewController: UIViewController {
  override func viewDidLoad() {
    super.viewDidLoad()

    // add your custom logic here
  }
}
```

ViewWillAppear

Gets called after the view is created and before the view is shown on the screen. At this point, the view will still have its original size and won't match

the size of the screen.

```
class ViewController: UIViewController {
  override func viewWillAppear(_ animated: Bool) {
    print("hello!")
  }
}
```

ViewDidAppear

Gets called after the view is shown on the screen. At this point, the view size will match the size of the screen.

```
class ViewController: UIViewController {
  override func viewDidAppear(_ animated: Bool) {
    print("look at my view!")
  }
}
```

ViewWillDisappear

`viewWillDisappear` gets called before the view will disappeared from the screen. This can happen for three main reasons:

- it has been removed from the view hierarchy
- another view controller is presented
- the app is going in the background

```
class ViewController: UIViewController {
  override func viewWillDisappear(_ animated: Bool) {
    print("bye!")
  }
}
```

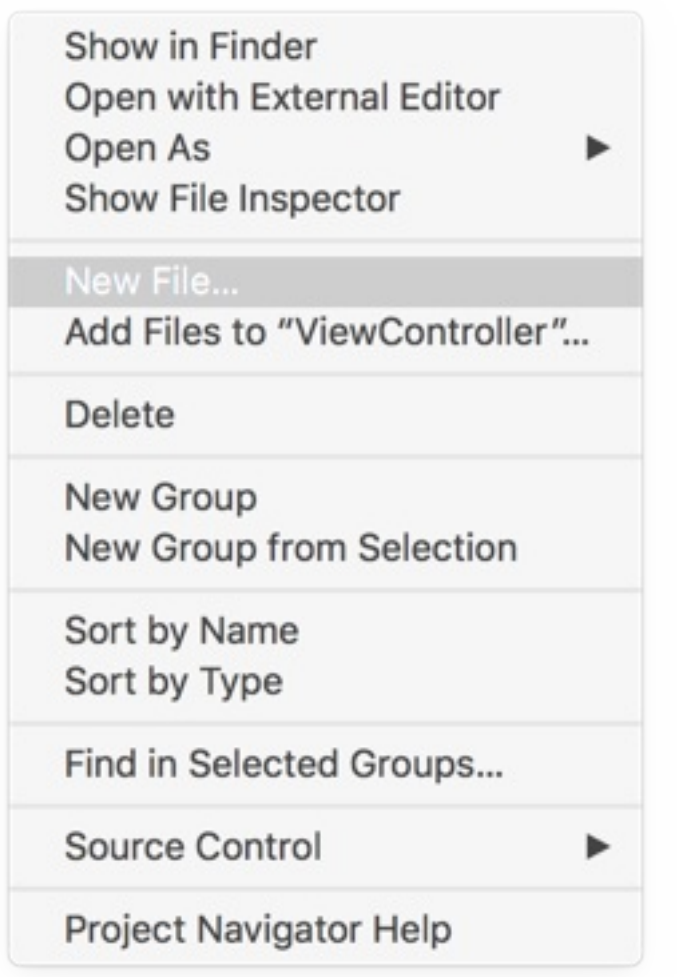
ViewDidDisappear

`viewWillDisappear` gets called after the view disappeared from the screen.

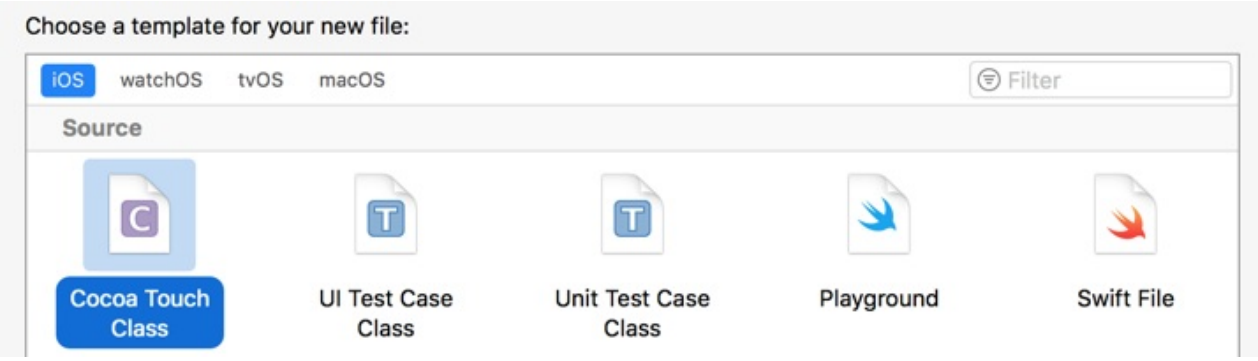
```
class ViewController: UIViewController {
  override func viewDidDisappear(_ animated: Bool) {
    print("I'm going to miss you!")
  }
}
```

Creating a New Controller

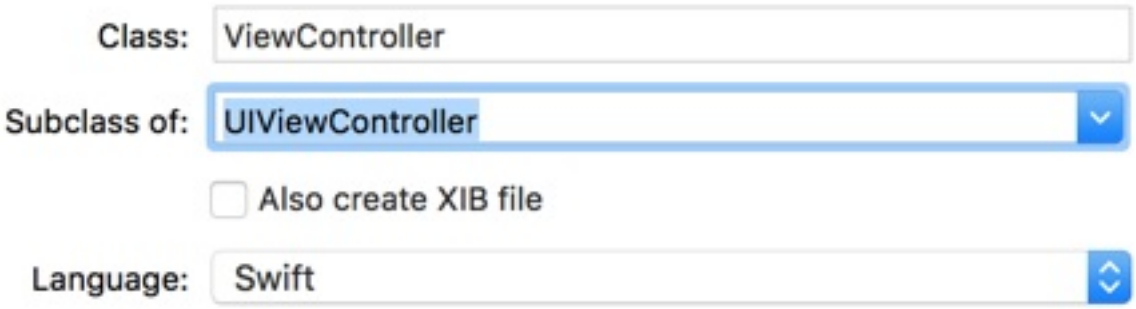
To create a new `UIViewController` subclass, right click on the main group from you project and select `New File`:



Select the `Cocoa Touch Class` template:

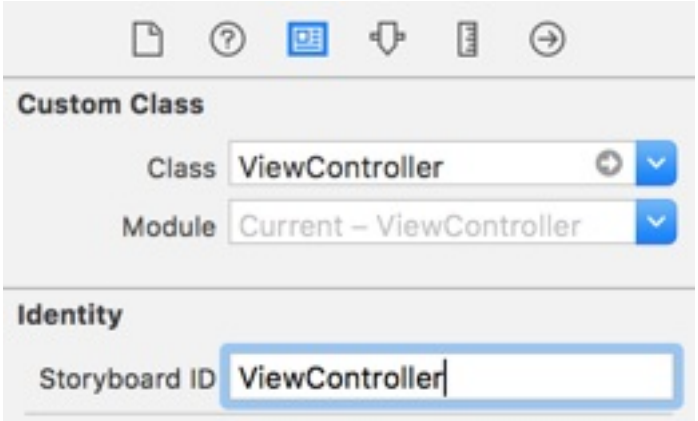


Change the class to `UIViewController` and name it:



This will create a swift file with the `UIViewController` subclass representing that controller. To use it you need to add it to the storyboard. Drag and drop a `View Controller Object` from the `Component Library` next to your other view controller(s):

Go to the `Identity Inspector` and change the class from `UIViewController` to the one you just created and set the storyboard ID - I usually use the name of the class for the storyboard ID.



You can make a separate file for this enum. You can instantiate a new view controller by using the storyboard object:

```
let secondary = storyboard?.instantiateViewController(
   withIdentifier: "AnotherViewController")
```

Zen bits

To make your code less error prone, it's a good practice to keep all your strings grouped in enums:

```
enum Screen: String {
    case Main = "ViewController"
    case Secondary = "AnotherViewController"
}

...

let secondary = storyboard?.instantiateViewController(
   withIdentifier: Screen.Secondary.rawValue)
```

This will help you avoid making typos and needing to write long strings without autocomplete :)

Presenting View Controllers

The easiest way to show a view controller on the screen is by presenting it from another one:

```
guard let secondary =
  storyboard?.instantiateViewController(
    withIdentifier:
      Screen.Secondary.rawValue) else {
    return
}

present(secondary, animated: true)
```

When the screen is done with its job, it can be dismissed by calling the `dismiss` method:

```
secondary.dismiss(animated: true)
```

> the usual case is that the secondary controller calls the dismiss method on `self`

Mini Project

Step 1: Create a New Project

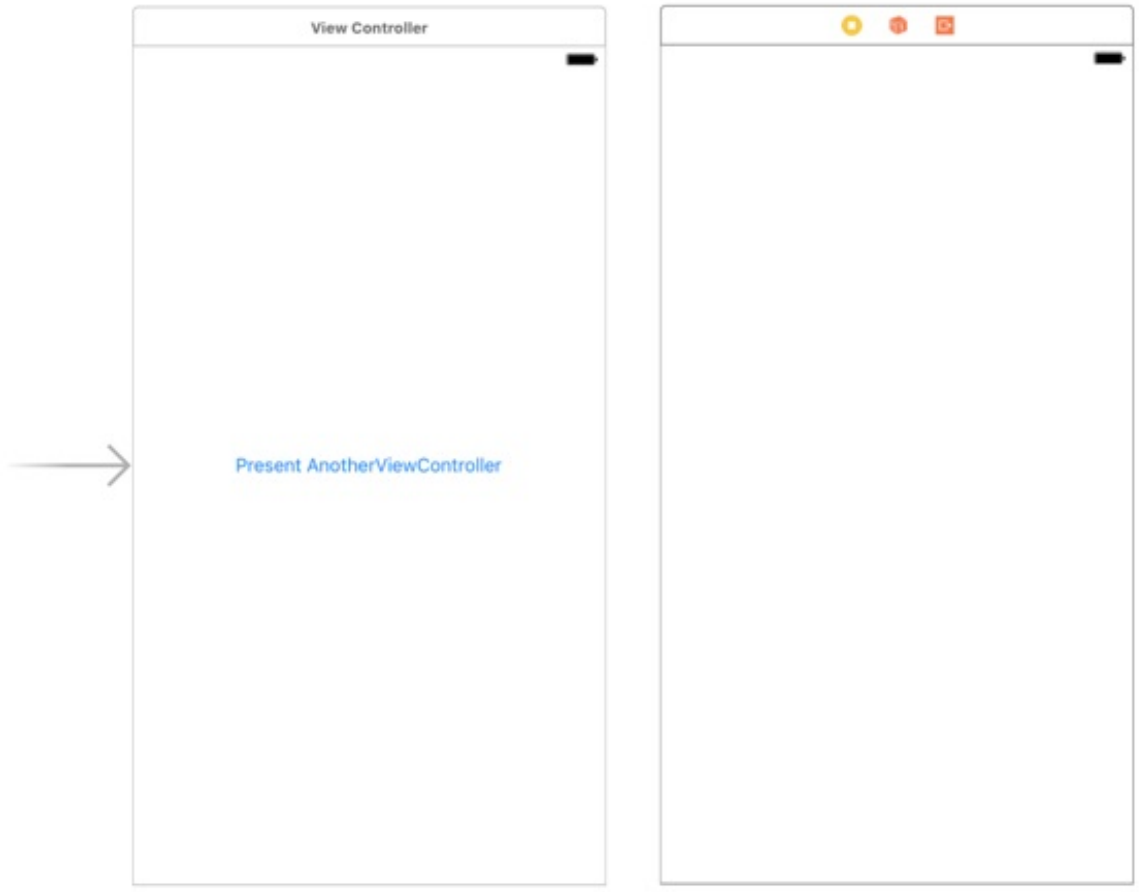
Create a new Xcode project with the `Single View Application` template.

Step 2: Create Another View Controller

Add a new view controller named `AnotherViewController` to the project. This involves creating the class and adding a view controller object in the storyboard with the class and storyboard ID `AnotherViewController`.

Step 3: Present AnotherViewController

Add a button to `ViewController` with the title `Present AnotherViewController`. Center it on the screen.



Connect the `.touchUpInside` event with a new `@IBAction` called `didTapPresent`.

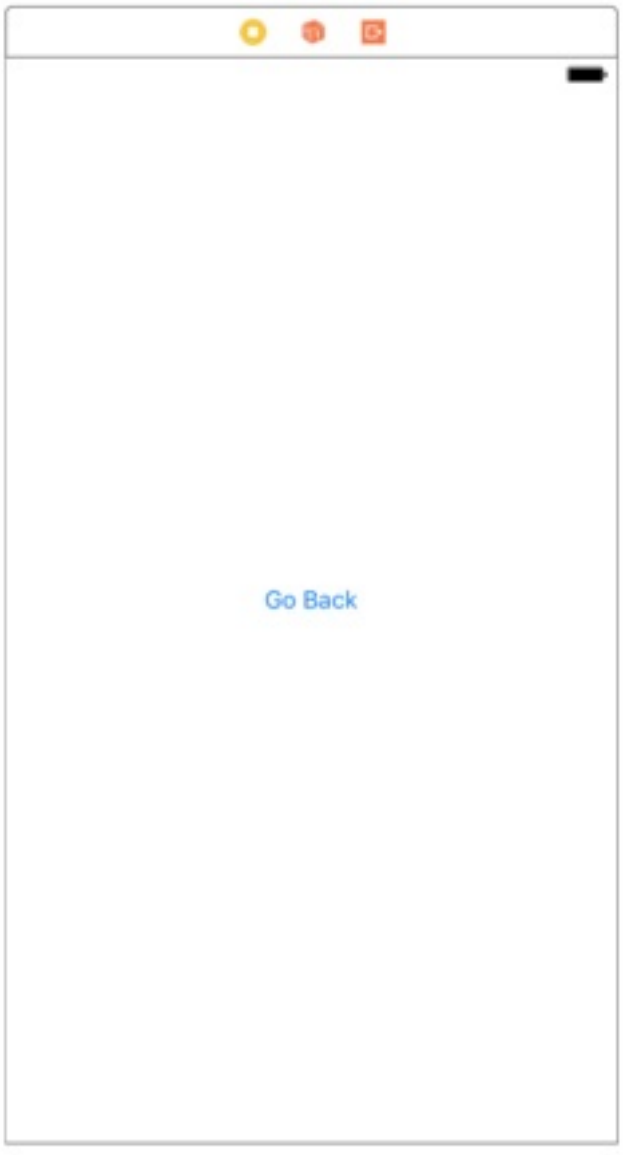
In `didTapPresent` create an instance of `AnotherViewController` and present it:

```
class ViewController: UIViewController {

    @IBAction func didTapPresent(_ sender: Any) {
        guard let secondary =
          storyboard?.instantiateViewController(
            withIdentifier:
              Screen.Secondary.rawValue) else {
            return
        }

        present(secondary, animated: true)
    }
}
```

Run the project now. When you tap on the button you should see an empty screen.

Step 4: Dismiss AnotherViewController

Add a button with the title `Go Back` in `AnotherViewController`. Connect the `.touchUpInside` event to the `didTapGoBack` method and call the `dismiss(_)` method in it:




```
class AnotherViewController: UIViewController {

    @IBAction func didTapDismiss(_ sender: Any) {
        dismiss(animated: true)
```

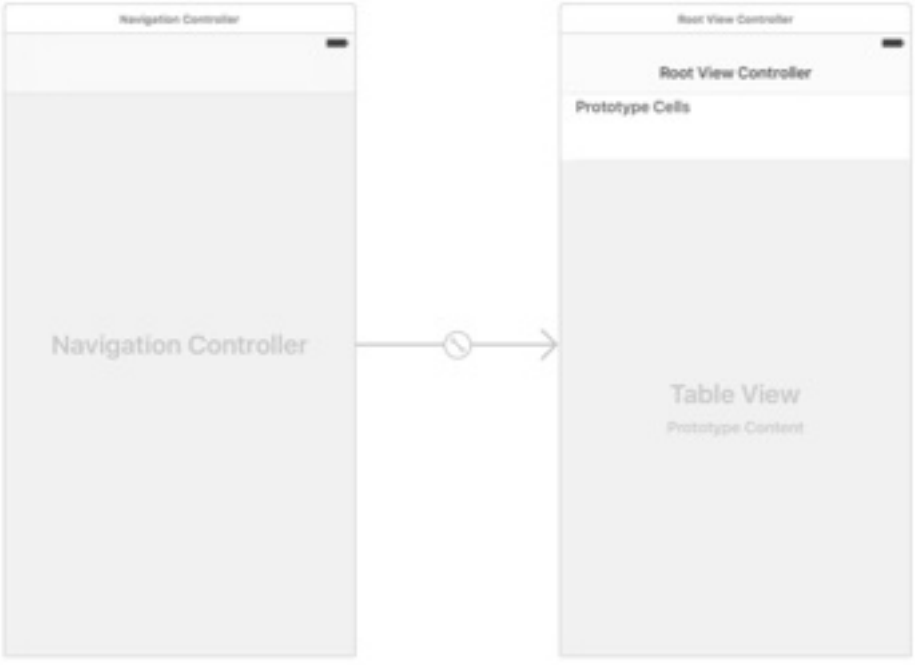
```
    }

}
```

Run the app now. You should be able to switch back and forth between controllers.

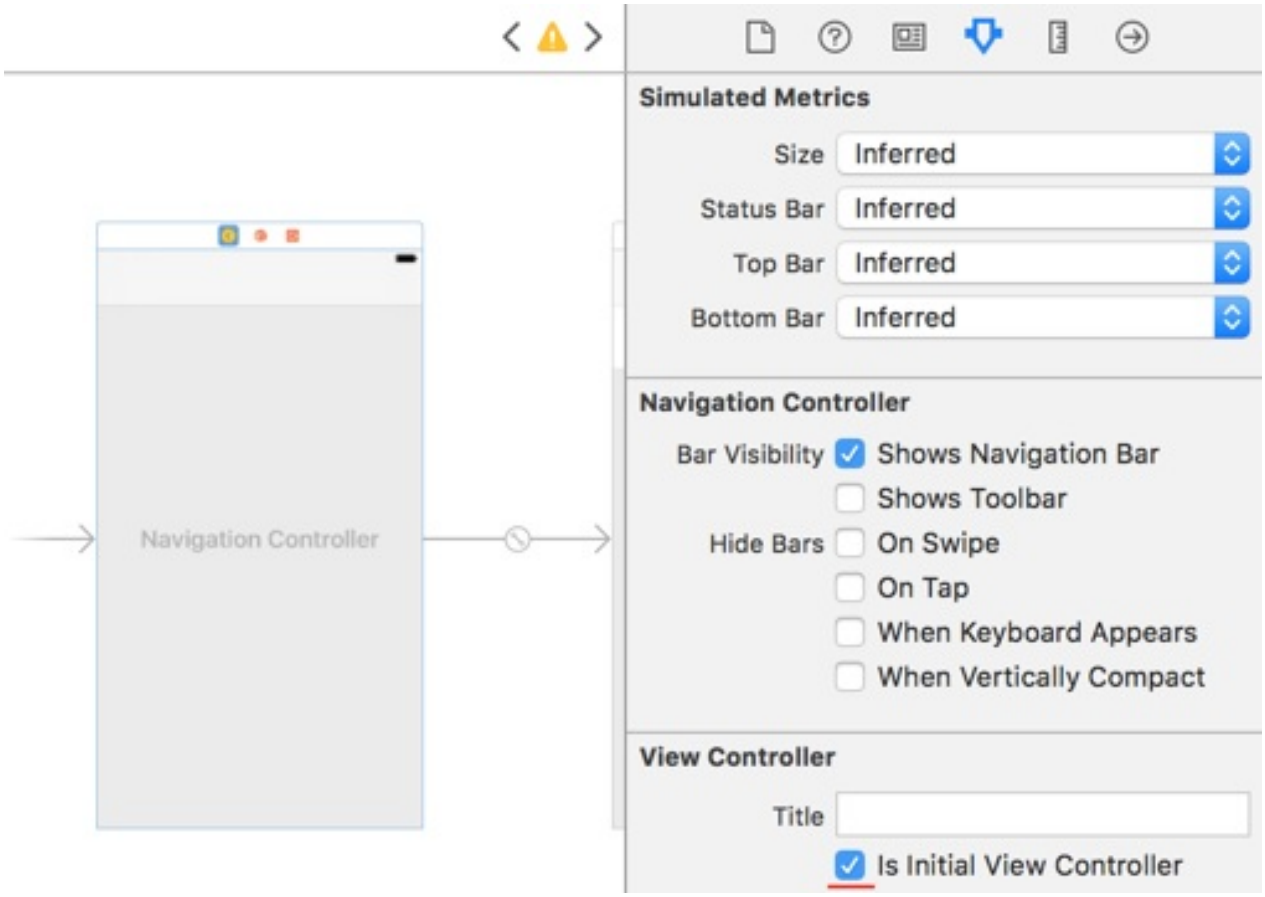
UINavigationController

You cannot present another modal from a controller that was presented modally. To do that you need to use a navigation controller. That's a view controller that manages a stack of controllers that are displayed.

The first thing you need to do, in order to use a navigation controller, is to add it to your storyboard. Drag and drop a `Navigation Controller` from the `Component Library`:



Make it the `Initial View Controller`:



Set your previous initial view controller as the root view controller of the navigation controller:

Remove the other view controller created by the `Navigation Controller Object` template or if you haven't setup you initial controller use that `View Controller Object`. You do this by connecting the `rootViewController` outlet with the desired controller object .

You can access the navigation controller by calling the `navigationController` property on any view controller from the navigation stack.

Push

To push a view controller on the navigation stack you need to call `pushViewController` method on the navigation controller instance:

```
class ViewController: UIViewController {
    func showSecondary() {
        guard let secondary =
            storyboard?.instantiateViewController(
              withIdentifier:
                Screen.Secondary.rawValue) else {
                return
        }

    navigationController?.
      pushViewController(secondary,
            animated: true)
    }
}
```

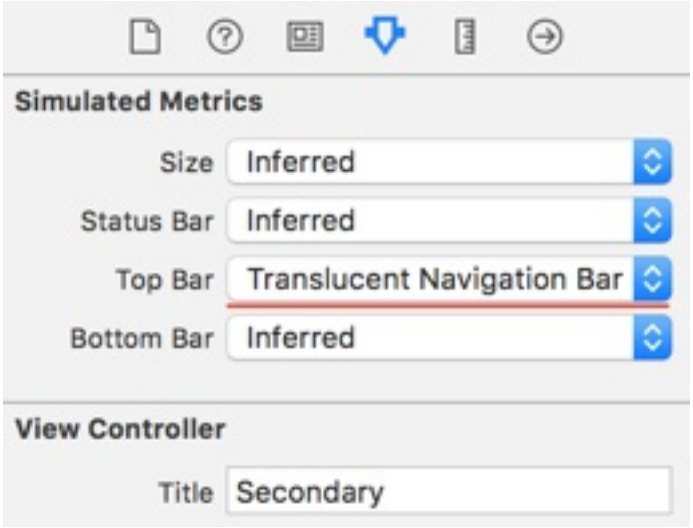
Pop

To push a view controller on the navigation stack you need to call `popViewController` method on the navigation controller instance:

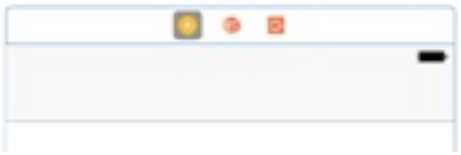
```
class ViewController: UIViewController {
    func popViewController() {
    navigationController?.
      popViewController(animated: true)
    }
}
```

The Navigation Bar

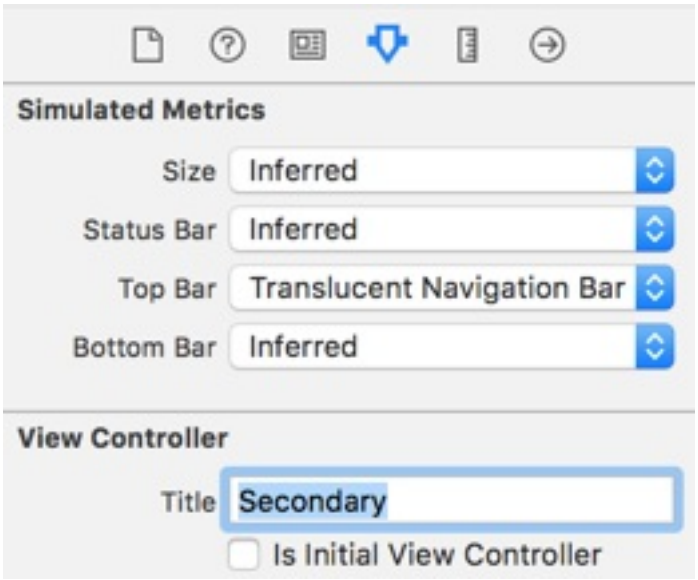
To make the navigation bar visible on a view controller object, you need to set the `Top Bar` simulated metric:



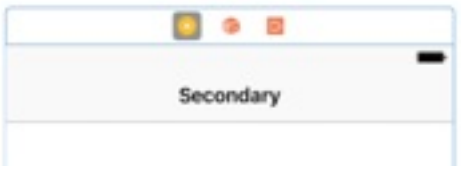
This will make a navigation bar to appear on top of the view controller:



You can change the title from the `Attributes Inspector` or by setting the `title` property on the view controller:




```
secondary.title = "Secondary"
```



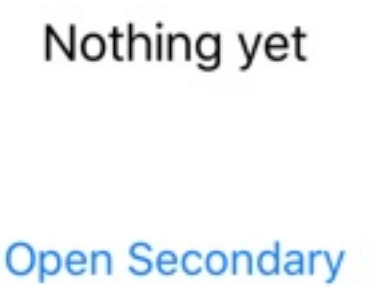
To hide the navigation bar you need to set `isNavigationBarHidden` property to true.

Exercises

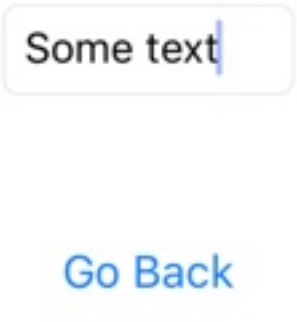
Pass a reference

Present controller and pass give it a reference of a label from the initial view. The second screen has an input field and a `Go Back` button. Display the inputed text in the second screen in the label from the initial screen.

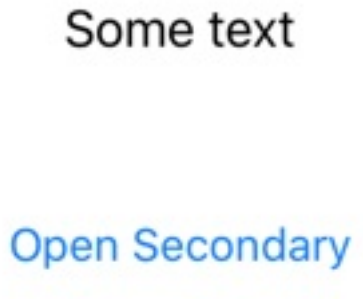
When you open the app the label says `Nothing yet`:



After that you enter some text:



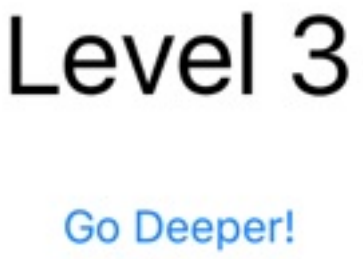
And go back:



Solution

Inception

Make an app with a single view controller class. That controller shows a label in the center indicating the deepness of the controller instance. Below the label there's a button with the title `Go Deeper`. Every time you go deeper, you push another instance of the same controller on the stack and the level should increase by one, the initial level is 1.

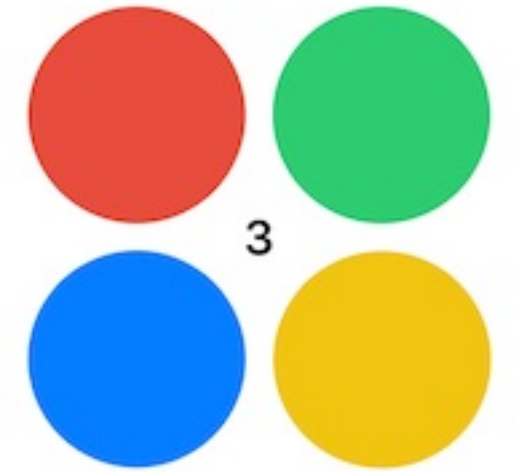


Solution

Memory Game

The game shows 4 colored circles. Each circle makes a particular sound when it is pressed or activated by the game. A round in the game consists of the app lighting up one or more buttons in a random order, after which the player must reproduce that order by pressing the buttons. As the game progresses, the number of buttons to be pressed increases by one each round, the first part of the sequence remains the same.

So the app might generate the sequence `[.red]`, if the user taps the red circle, the app will add another color to the sequence, making it `[.red, .green]`. After another succesfull round the sequence can become `[.red, .green, .red]`. The sequence resets when the player makes a wrong move.



You can generate tones by using system sounds, the phone pad sounds are perfect for this game:

```
enum Feedback: Int {
    case pad0 = 1200
    case pad1 = 1201
    case pad2 = 1202
    case pad3 = 1203
```

```
      case pad4 = 1204
      case pad5 = 1205
      case pad6 = 1206
      case pad7 = 1207
      case pad8 = 1208
      case pad9 = 1209

    func play() {
          if self != .None {
              let systemSoundID = SystemSoundID(self.rawValue)
              AudioServicesPlaySystemSound(systemSoundID)
          }
      }
}
```

You will need to import the `AVFoundation` framework in order to work with system sounds.

Solution