framework training
We love technology

Building iOS Apps in Swift

Course notes for part 1

May 2022

020 3137 3920 @FrameworkTrain

frameworktraining.co.uk

Session 1: Introduction to Swift

- Trainer and Delegate Introductions / Intro to course
- Foundations of Swift
- XCode Playgrounds
- Basics of the language: Types, Operators, Conditionals, Iterators, Strings, Arrays, Dictionaries
- Lab 1: Trying out the language basics

Welcome & Introductions

Chris Price

cjp@aber.ac.uk

framework training
business value through education

Building iOS Apps in Swift

- 10 day course
- Based around excellent material provided by Apple
- Expanded on in complex areas not covered well by Apple's course

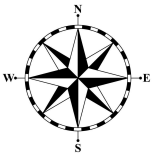
Introductions - me

- Course Instructor: Chris Price
  - Professor at Aberystwyth University, Wales
  - Teach courses in Software Engineering and in App Development in Swift
  - Been developing apps since 2009
  - Shipped apps to around 200,000 users in that time

Introductions - you

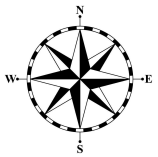
- Course attendees
  - You…
    - Name
    - Professional Position
    - Programming experience
    - Motivation and Expectations

Course Content



- Basics of Swift language
- Extra features of Swift needed for app building
- Enough about building iOS apps to build *most* self-contained apps
- Persistence
- Linking to internet systems, cloud services, libraries
- Notifications
- Reactive systems and concurrency
- Automated testing

How will the course work



- We will cover basics of Swift quickly, and try things out in Swift Playgrounds
- We will take more time over possibly unfamiliar features such as Optionals, Protocols and Closures
- We will build *many* small apps in Xcode, some together and some following instructions in Apple Books

Questions/comments?

?

Introduction to Swift and Playgrounds

Chris Price

Swift - a modern language: Safe

- Strong typing
- Compile-time checking as much as possible
- Ensures that things are initialised
- Makes switch statements cover all possible cases
- Makes clear you know what is included in an if statement
- Takes nil pointers seriously

Swift - a modern language: Fast

- Language that helps compiler to optimise
- Encourages the user to make their intentions clear so that compiler can optimise the code

Swift - a modern language: Expressive

- Doesn't make people write stuff the compiler should know:
- Implied type declaration where possible
- Implicit type name when type known (e.g. for enums)

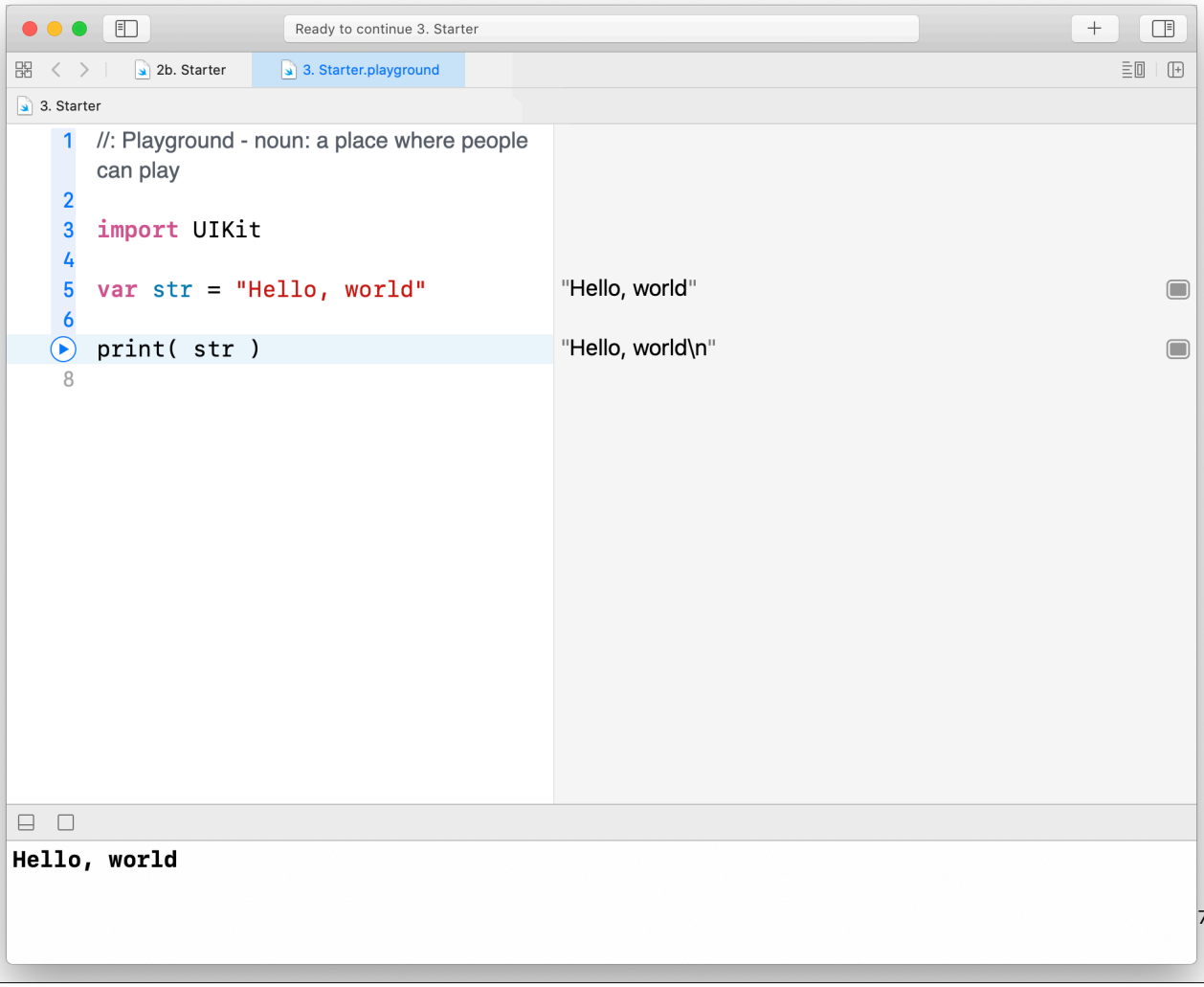
- Has the features you might expect in a modern language
- Powerful Collections
- Protocols
- Extensions
- Functional programming
- Ints, Doubles etc are first class items

Hello, world

```
print("Hello, world!")
```


Try making a playground

1. Open Xcode
2. Choose File > New > Playground
3. Select iOS, select the Blank template and click Next
4. Change str to "Hello, world!"
5. Hold cursor over line you want to compile to and a blue circle with triangle in it will show
6. Click on blue triangle to run all code up until there



Explore playgrounds and language basics

1. Try running "Starter Playground" included in Session 1 of the course
2. See how Playgrounds shows you the run time values
3. There is a 4 page summary of the basics of the language included - try cutting and pasting language examples into a Playground, and see what they do.
4. Run ExploreBasics Playground, typing in answers to the questions and seeing how the answers run

More on material in this session

- We have very briefly covered the basics of the language
- They are covered much more slowly in Develop in Swift Fundamentals:
  - Lesson 1.1: Swift Intro and Playgrounds
  - Lesson 1.2: Constants, variables, types
  - Lesson 1.3: Operators
  - Lesson 1.4: Control flow
  - Lesson 2.1: Strings
  - Lesson 2.5: Collections
  - Lesson 2.6: Loops

We will look in more detail at:

- Lesson 2.2: Functions
- Lesson 2.3: Structures
- Lesson 2.4: Classes
- Unit 3: Optionals / Guard / Scopes

The Swift Language - Brief Overview (1)

Chris Price v1.4, Sept 2019

Simple Variable Types

Int, Double, Bool

Can be implicitly declared:
let x = 42 // Int constant
var y = 3.1 // Double variable

Explicit declaration:
let z: Int = 42 // Int constant
let a: Double = 3 // Double constant
let cold: Bool = true //Boolean constant

Common Operators

Arithmetic Operators

+	Add	// e.g. y = a+b
–	Subtract	
*	Multiply	
/	Divide	
%	Remainder	

Arithmetic shorthand

// e.g. y += 21 same as y = y+21

+=	Add and assign
–=	Subtract and assign
*=	Multiply and assign
/=	Divide and assign
%=	Remainder and assign

Unary operators

-	Minus	//e.g. y = -x Gives negative of a number

Boolean tests – return true or false

<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
==	Equal
!=	Not equal
&&	Logical AND
||	Logical OR

Conditionals

```
if comparison1 {
  //called if comparison 1 is true
}
else if comparison2 {
   //called if comparison 2 is true
}
else {
   //called if nothing else is true
}
```

e.g.

```
let age = 27
if age >= 18 {
   print( "allowed to vote")
} else {
   print( "not old enough to vote")
}
```

Curly brackets are mandatory.
Round brackets around condition are optional.

Switch Statement

```
let age = 1
switch age {
case 0,1:
   print( "baby" )
case 2...4:
   print( "toddler" )
case 5..<18:
   print( "child" )
default:
   print( "adult" )
}
```

First applicable case is executed then exits. Must have a default unless all values are accounted for by other cases. Can switch on more than one value.

```
let num = 15
switch (num%3, num%5) {
   case (0, 0):
      print( "FizzBuzz" )
   case (0, _):
      print( "Fizz" )
   case (_, 0):
      print( "Buzz" )
   default:
      print( num )
}
```

The Swift Language - Brief Overview (2)

Chris Price v1.4, Sept 2019

Strings

Simple string expressions and printing out

```
let emptyString = ""
let greeting = "Hello"
let friendlyGreeting = greeting + ", friend"
print( friendlyGreeting ) // prints to console
```

String interpolation

```
let temp = "Boiling water is \(100*1.8 + 32)F"
```

String tests

```
if emptyString.isEmpty ...
if greeting.hasPrefix( "He" )...
if greeting.hasSuffix( "matey" )...
```

Equality tests whether made up of same characters in same order

```
if emptyString == ""     //this is true
if greeting == "hello"   // this is false as lower case
if "aardvark" < "apple"  // lexically less, so true
```

Useful string features

```
greeting.count // returns string length (5)
greeting.lowercased()  //returns "hello"
```

Manipulating characters in a string

```
let me = "Chris Price"
let space = me.firstIndex(of: " ") ?? me.endIndex
let name = me[..<space]
```

Arrays

```
//Declare empty array of strings
var myPets: [String]  =  [ ]
//Different way of doing same thing
var pets = [String]( )
myPets.count     // will be zero

myPets.append( "Chaz the Dog" )
myPets.append( "Dave the Goldfish" )
myPets.count    // will be two

//Declare constant array with one string in it
let yourPets = [ "Idris the Guinea Pig"]
// Combines the two string arrays into one string array
 var ourPets = myPets + yourPets
ourPets.count   // will be three

// Replace an element in a variable array
ourPets[0] = "Chaz the Bad Dog"
// Delete an element
let deadPet = ourPets.remove(at: 2)
ourPets.count    // now only two pets
// deadPet will contain the element removed

// can add extra element at specific position
ourPets.insert( "Bob the Capybara", at: 0)

// Initialise fixed size with default values
// Makes array with 200 values of 0.0
var readings = [Double]( repeating: 0.0, count: 200)
readings.removeLast()   // Can delete last element
readings.count              // will be 199
```

For Loops

for item in range { statements }

```
var sum = 0
for i in 1...5 { sum = sum + i }          // Gives the answer 15
sum = 0
for i in 1..<5 { sum = sum + i }          // Gives the answer 10
```

for item in collection { statements }

```
var listOfAttendees = [ "Bill", "Jane", "Jim", "Fred", "Ann" ]
for name in listOfAttendees{
   print( name )
}    // Will print each name in the list to console
```

Equivalent to 'for item in collection' using range

```
for i in 0..<listOfAttendees.count {
   print( listOfAttendees[i] )
}    // Will print each name in the list to console

// can get index and element by doing the following
for (index, name) in listOfAttendees.enumerated() {
   print( "\(index+1). \(name) " )
}
```

While Loops

while condition { statements }

e.g.

```
var i = 0
while (i < 100)
{
   print( "I told you so" )
   i++
}   // will print "I told you so 100 times
```

The Swift Language - Brief Overview (3)

Chris Price v1.4, Sept 2019

Functions

```
// function with no parameters or result
func printHello() {
  print( "Hello")
}
printHello()   // usage

// function with one parameter, no result
func printMessage(message: String) {
  print( message )
}
printMessage(message: "Well, hello")   // usage

// To not need to name parameter in usage,
// would need to define printMessage as follows
// func printMessage(_ message: String)

// function with two parameters, one result
func concat(s1: String, s2: String ) -> String {
   return s1 + " and " + s2
}
let billAndBen = concat(s1: "Bill", s2: "Ben")

// function with no params, tuple as result
func returnTuple() -> (String, Int) {
  return( "Page not found", 404 )
}
let result = returnTuple()
// String in result can be accessed as result.0
// Int in result can be accessed as result.1
```

Structures

```
struct Person{
  // properties
  let firstName: String
  let lastName: String

  // computed property
  var fullName: String {
    return firstName + " " + lastName
  }

  // Method - like function but on instance
   func printName(){
      print( fullName )
   }
}

//Create an instance of a Person
let newPerson = Person(firstName: "Jim",
                       lastName: "Jones")

//Call method for instance newPerson
newPerson.printName()

let oldPerson = newPerson
//When you assign a structure to a new variable,
//its value is COPIED (unlike classes).

// Structures have automatically generated
//initialisers, but you can also write your own.
```

Classes

```
//Class syntax would be identical for the Person
//class, but need to write an initialiser.
class Person{
  let firstName: String
  let lastName: String

   // Needs this initialiser
   init(fName: String, lName: String) {
      firstName = fName
      lastName = lName
   }

  var fullName: String {
      return firstName + " " + lastName
  }

   func printName(){
      print( fullName )
   }
}

let newPerson = Person(fName: "Jim",
                       lName: "Jones")

newPerson.printName()
let oldPerson = newPerson
//When you assign a class to a new variable,
//both variables point to the same structure
```

The Swift Language - Brief Overview (4)

Chris Price v1.4, Sept 2019

Class inheritance

```
// Can declared subclass of existing class
// Subclass inherits properties and methods
// Extra properties and methods can be declared,
// but extra properties must be initialised.
// Where methods are replaced by more specific
//versions, they need the qualifier 'overrides'

class Doctor: Person{
  let specialism: String
   // Needs this initialiser
   init(fName: String, lName: String, spec: String) {
      specialism = spec
      super.init(fName: fName,lName: lName)
   }

   override func printName(){
      print( "Dr " + fullName+" does "+specialism)
   }

   func seePatient( patientName: String ) {
      print( "Hullo, " + patientName
             + ", I'm Dr " + fullName )
   }
}
let doc = Doctor(fName: "Jane",
                 lName: "Jones", spec: "Pediatrics")
doc.printName()
doc.seePatient(patientName: "George")
```

Enumerations

```
enum TempType {
   case degF
   case degC
}
var tempType = TempType.degC
if tempType == .degF {
   print("In Fahrenheit")
} else {
   print("In centigrade")
}

// More complex example
enum DaysOfWeek {
   case monday, tuesday, wednesday,
          thursday, friday, saturday, sunday
}

var today = DaysOfWeek.monday
switch today{
case .saturday, .sunday:
   print( "Chill - it's the weekend" )
default:
   print( "Apply nose to grindstone" )
}
// Will print "Apply nose to grindstone"

today = .saturday
// We know type of today now,
// so don't need to say "DaysOfWeek"
```

Dictionaries

```
//Create a dictionary containing our pets
var pets = ["Dave": "Fish", "Chaz": "Dog"]
pets["Idris"] = "Pig"  //Adds Idris to pets
pets.isEmpty      // returns false
pets.count        // returns 3
pets["Idris"]     // returns " Pig"
pets["Dog"]       // returns nil

// Update value for Chaz
pets["Chaz"] = "Bad Dog"

// Loop over key/value pairs
for (name, animal) in pets {
   print( "\(name) is a \(animal)" )
}

//Delete Chaz from dictionary
pets["Chaz"] = nil // Deletes Chaz entry

// Loop over keys
for name in pets.keys {
   print( "\(name) is a pet" )
}

// Loop over values
for animal in pets.values {
   print( "We have a \(animal)" )
}
```

Session 2: Introduction to using Xcode

- Making first app / toolkit features
- MVC and linking code to Storyboards
- Using the documentation
- Introduction to UIKit
- *Lab 2: Building and running single screen apps on simulator and device*

- This will cover lessons 1.5 to 1.8 of Development in Swift Fundamentals

1

How do you make a multi-screen app?

Case Study: Crossflow Energy

At the core of Crossflow's MODULAR MICROGENERATION SYTEMS are Crossflow's SMART Wind Turbines integrated with market leading solar and battery technologies.

Turbine
Crossflow SMART TURBINE

PV Array
Flexi or Fixed Solar pV with ground and tower mounting options

Battery
Latest rack mounted battery technology providing consistent power supply

Generation or Grid
Other Energy Generation inputs optional

Model-View-Controller in Xcode

Changes to the model are dealt with by the controller

Controller
The code that links your views and your model

User actions on the view are dealt with by the controller

Controller updates the model when things change

Controller updates the view when things change

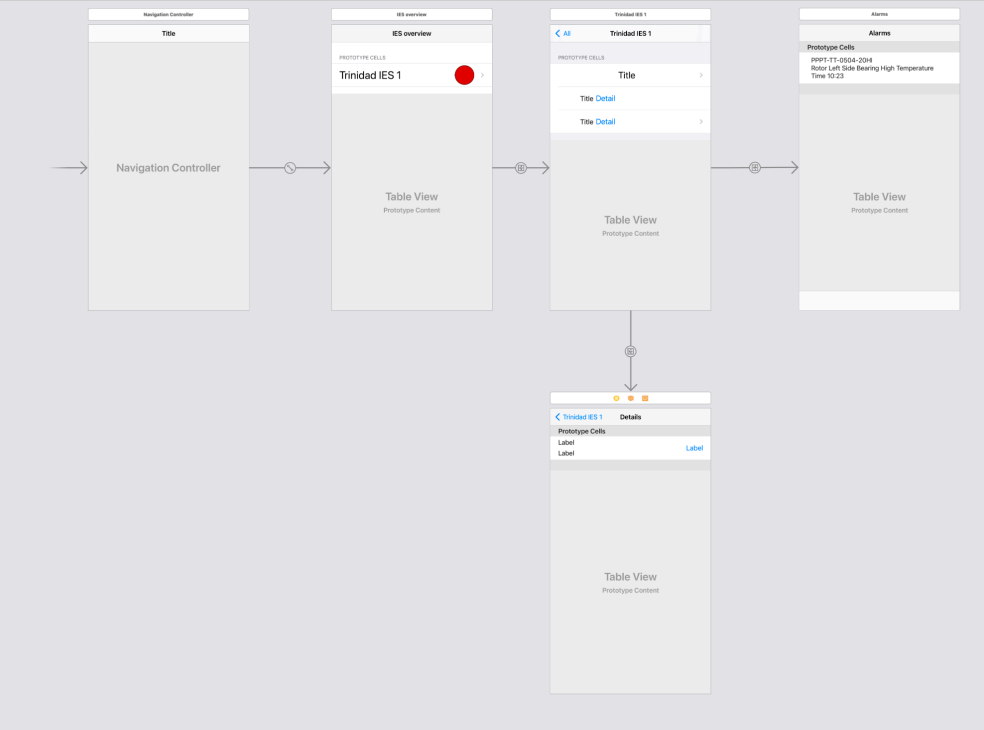
Model
The objects that hold / process the data for your app

View
Structure of what you see on the screen

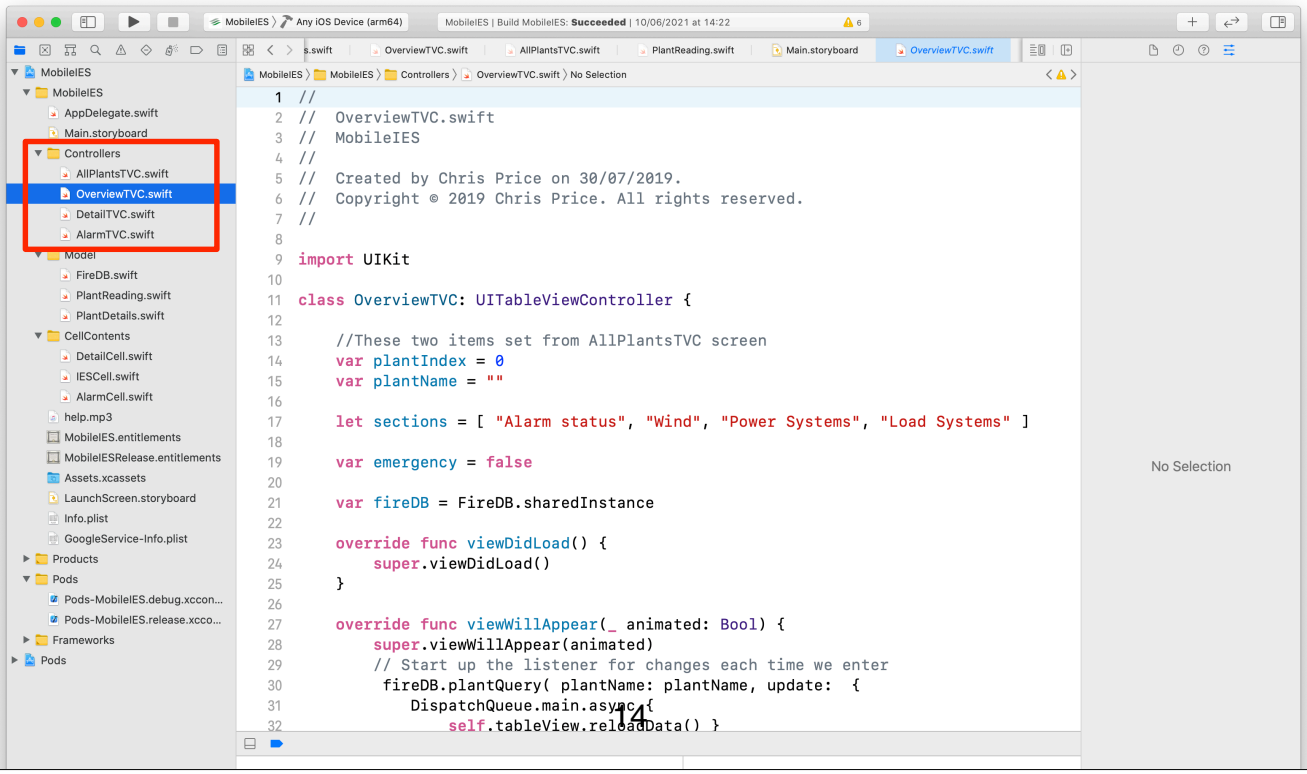
2

Views are made in Interface Builder

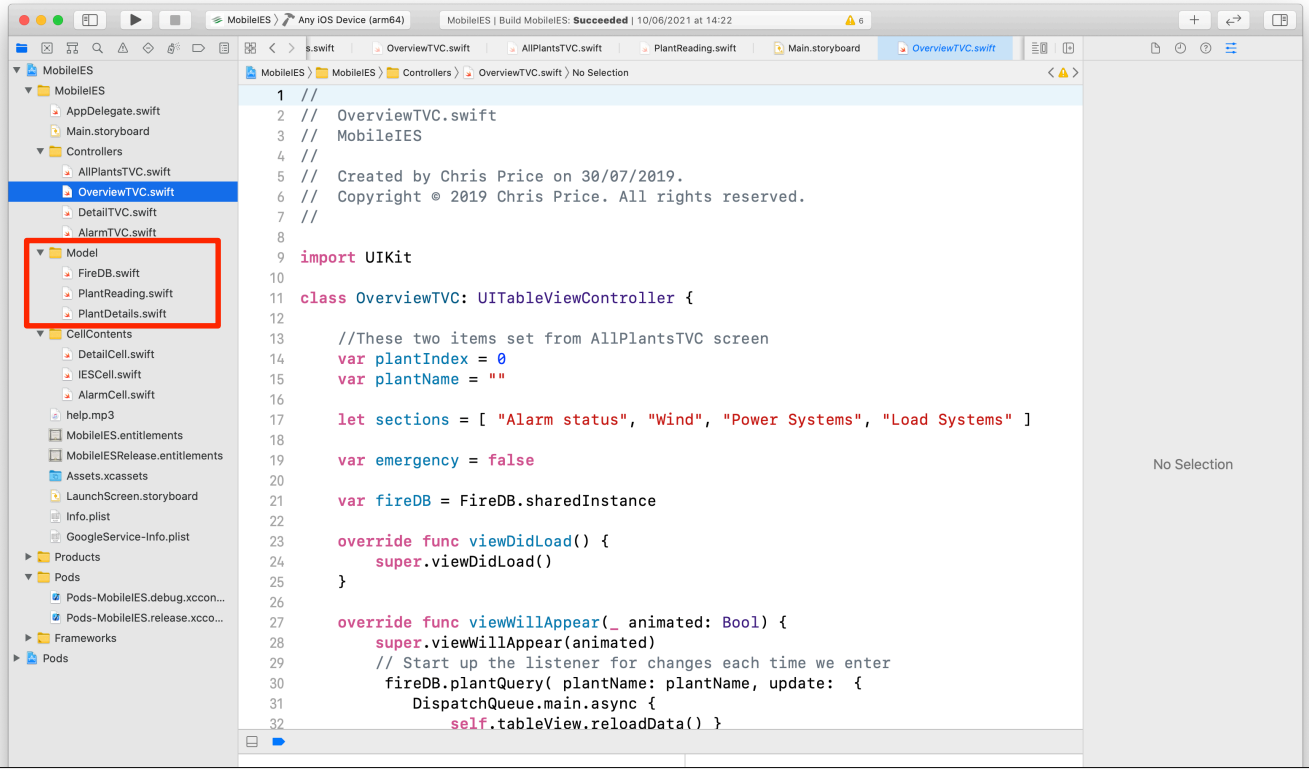
in simple case, one view for each screen



There will be view controller code for each of those screens



Model code will be a set of classes for holding and operating on data objects



Session 3: UIKit: Views and controls

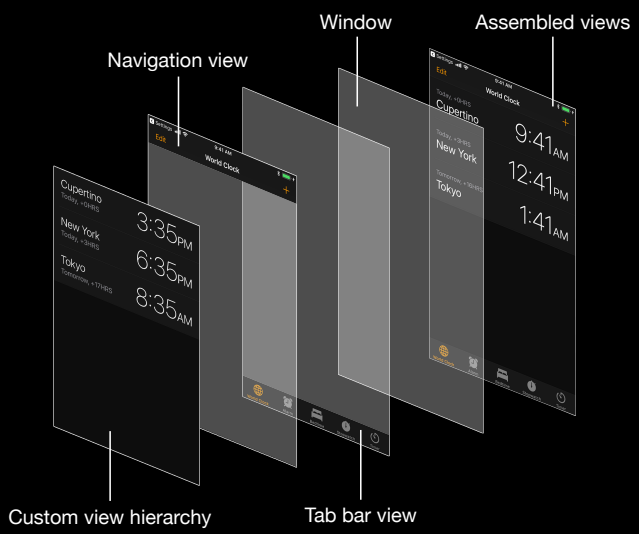
- Attributes of common views
- Adding controls to your app
- Adding more views
- *Lab 3: Building apps with more views and controls*

- This will cover lessons 2.8 to 2.10 of Development in Swift Fundamentals

1

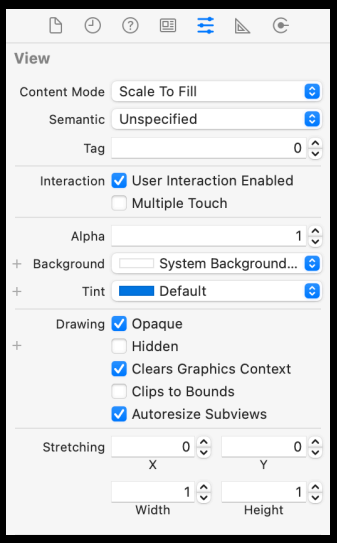
Unit 2—Lesson 8: Introduction to UIKit (p. 246 in book)

Common system views



Common system views
Configuration

Property	Description
Size	Width and height of the view
Position	Position of the view onscreen
Alpha	Transparency of the view
Background Color	Background color that will be displayed
Tag	Integer that you can use to identify view objects



Label (UILabel)

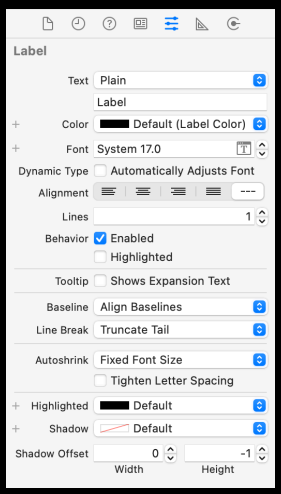
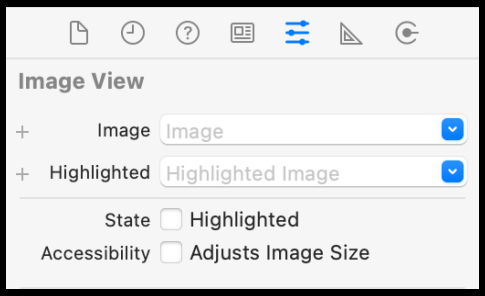
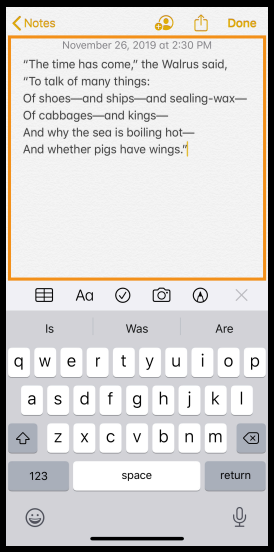
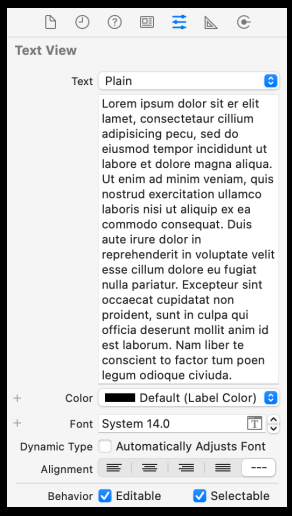


Image view (UIImageView)



Text view (UITextView)





Scroll view (UIScrollView)

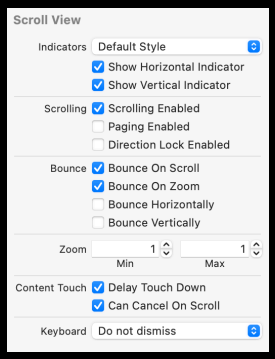
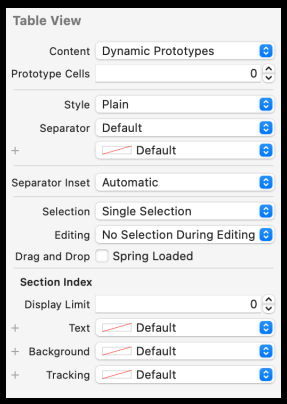
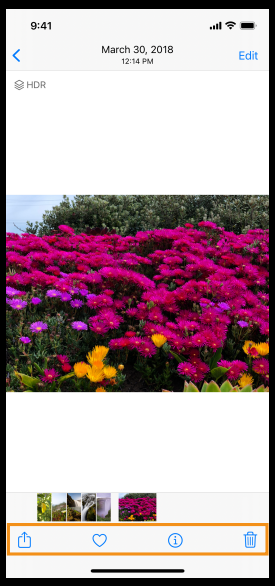
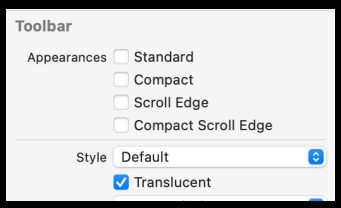


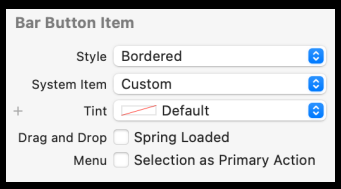
Table view (UITableView)



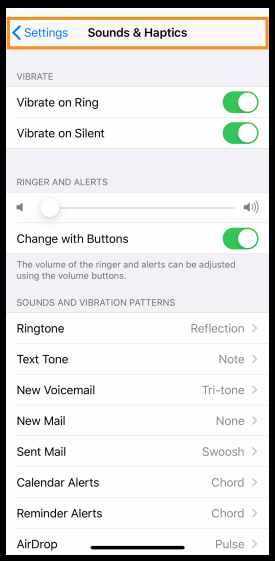
Toolbars (UIToolbar)

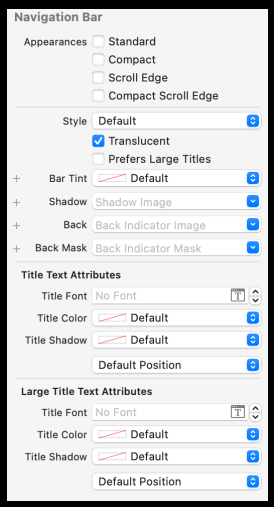




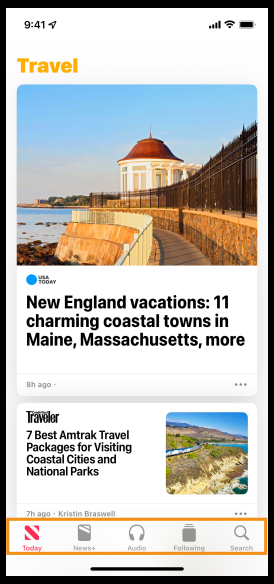


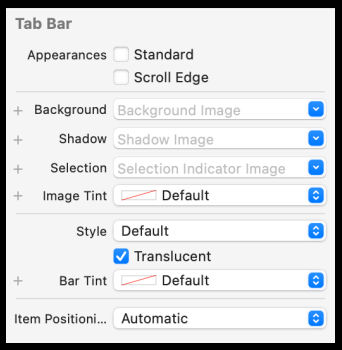
Navigation bars (UINavigationBar)





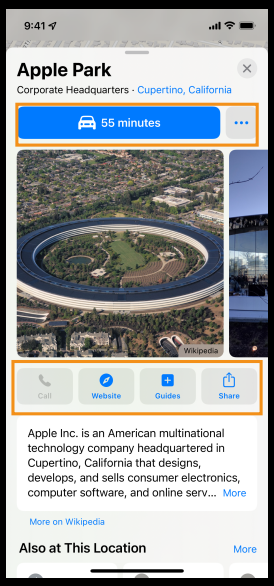
Tab bars (UITabBarController)

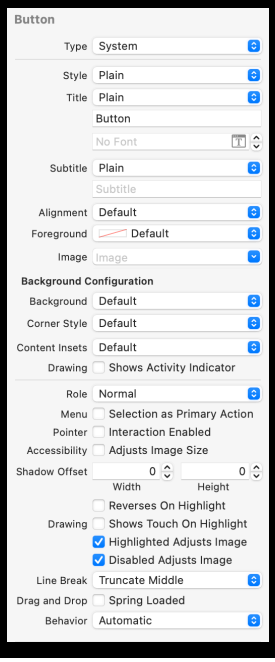




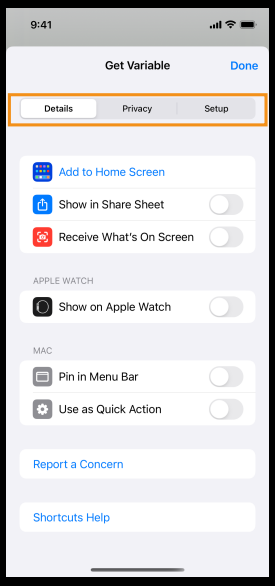
Controls

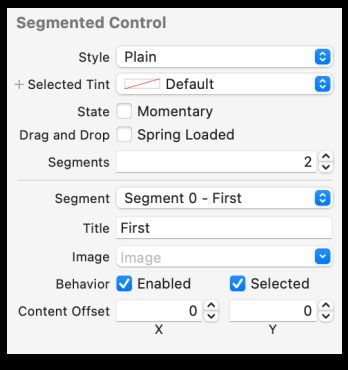
Buttons (UIButton)



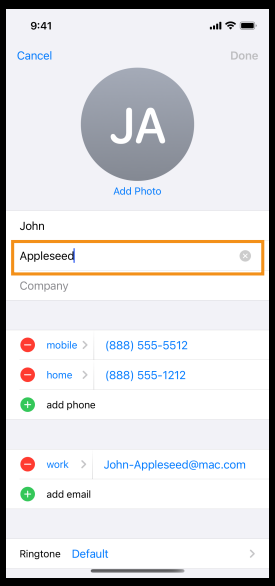


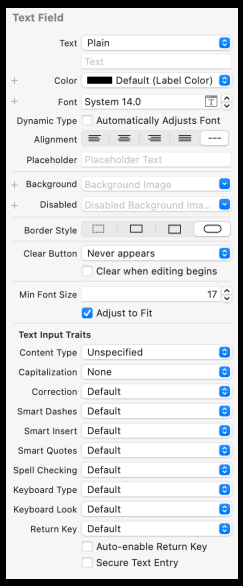
Segmented controls (UISegmentedControl)



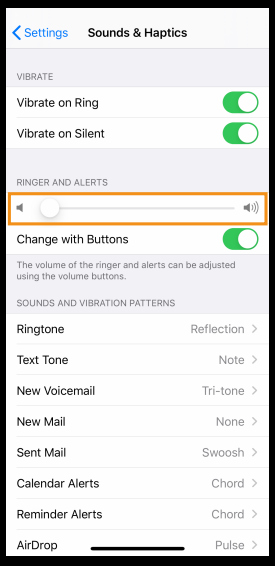


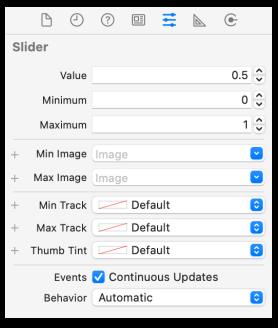
Text fields (UITextField)



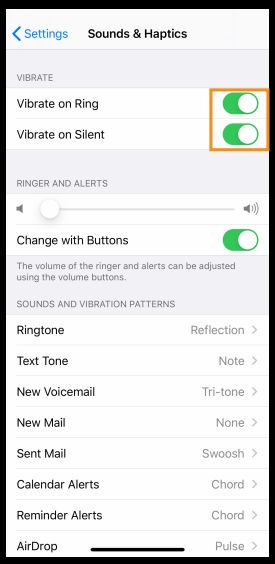


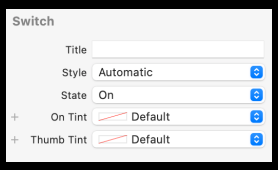
Sliders (UISlider)



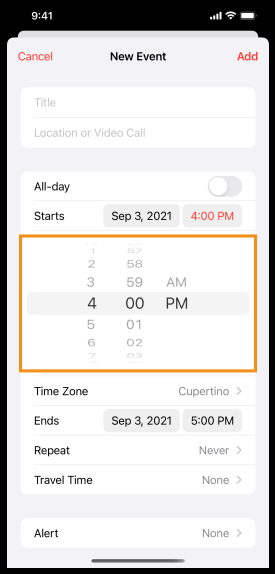


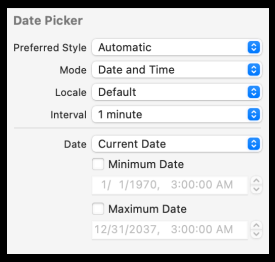
Switches (UISwitch)



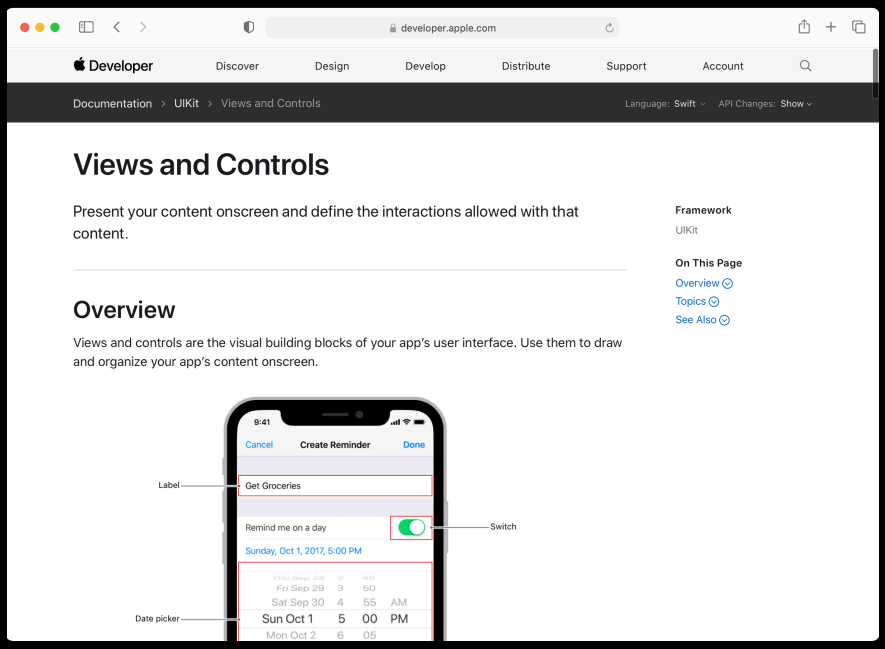


Date pickers (UIDatePicker)





UIKit Views and Controls Documentation



Unit 2—Lesson 7
Introduction to UIKit

Learn about some of the most commonly used interface elements in UIKit and where you can go to find out more.

Unit 2—Lesson 7
UIKit Survey

Identify different views and controls in some of the most common system apps on iOS

Look through Settings, Contacts, News, and Calendar

Use Pages to complete a survey of your findings

Session 4: Functions

- *Function parameters and results*
- *Argument labels*
- *Default values*
- *Lab 4: Writing functions*

- This covers lesson 2.3 of Development in Swift Fundamentals

1

Unit 2—Lesson 3: Functions

Functions

```
tieMyShoes()
```

```
makeBreakfast(food: "scrambled eggs", drink: "orange juice")
```

Functions
Defining a function

```
func functionName (parameters) -> ReturnType {
    // Body of the function
}
```

```
func displayPi() {
  print("3.1415926535")
}

displayPi()
```

```
3.1415926535
```

Parameters

```
func triple(value: Int) {
  let result = value * 3
  print("If you multiply \(value) by 3, you'll get \(result).")
}

triple(value: 10)
```

```
If you multiply 10 by 3, you'll get 30.
```

Parameters
Multiple parameters

```
func multiply(firstNumber: Int, secondNumber: Int) {
  let result = firstNumber * secondNumber
  print("The result is \(result).")
}

multiply(firstNumber: 10, secondNumber: 5)
```

```
The result is 50.
```

Return values

```
func multiply(firstNumber: Int, secondNumber: Int) -> Int {
  let result = firstNumber * secondNumber
  return result
}
```

Return values

```
func multiply(firstNumber: Int, secondNumber: Int) -> Int {
 return firstNumber * secondNumber
}
```

```
let myResult = multiply(firstNumber: 10, secondNumber: 5)
print("10 * 5 is \(myResult)")
```

```
print("10 * 5 is \(multiply(firstNumber: 10, secondNumber: 5))")
```

Argument labels

```
func sayHello(firstName: String) {
  print("Hello, \(firstName)!")
}

sayHello(firstName: "Amy")
```

Argument labels

```
func sayHello(to: String, and: String) {
  print("Hello \(to) and \(and)")
}

sayHello(to: "Luke", and: "Dave")
```

Argument labels
External names

```
func sayHello(to person: String, and anotherPerson: String) {
  print("Hello \(person) and \(anotherPerson)")
}

sayHello(to: "Luke", and: "Dave")
```

Argument labels
Omitting labels

```
print("Hello, world!")
```

```
func add(_ firstNumber: Int, to secondNumber: Int) -> Int {
    return firstNumber + secondNumber
}

let total = add(14, to: 6)
```

Default parameter values

```
func display(teamName: String, score: Int = 0) {
  print("\(teamName): \(score)")
}

display(teamName: "Wombats", score: 100)
display(teamName: "Wombats")
```

```
Wombats: 100
Wombats: 0
```

Unit 2—Lesson 2
Lab: Functions

There is a short Playground lab (called “Quick functions.playground”) in

Try that.

If you have problems with that, work through chapter 2.2 from the book, and try the function exercises in the Longer Functions playground in your own time.

Session 5: Prototyping an app interface

- Deciding what goes into the app
- Making the screens for a multi-screen app and joining them
- Trying out the app on potential users
- Taking Apple design considerations and HCI guidelines into account
- *Lab 5: Building an app prototype without code*

- This material not well covered in Apple coding books

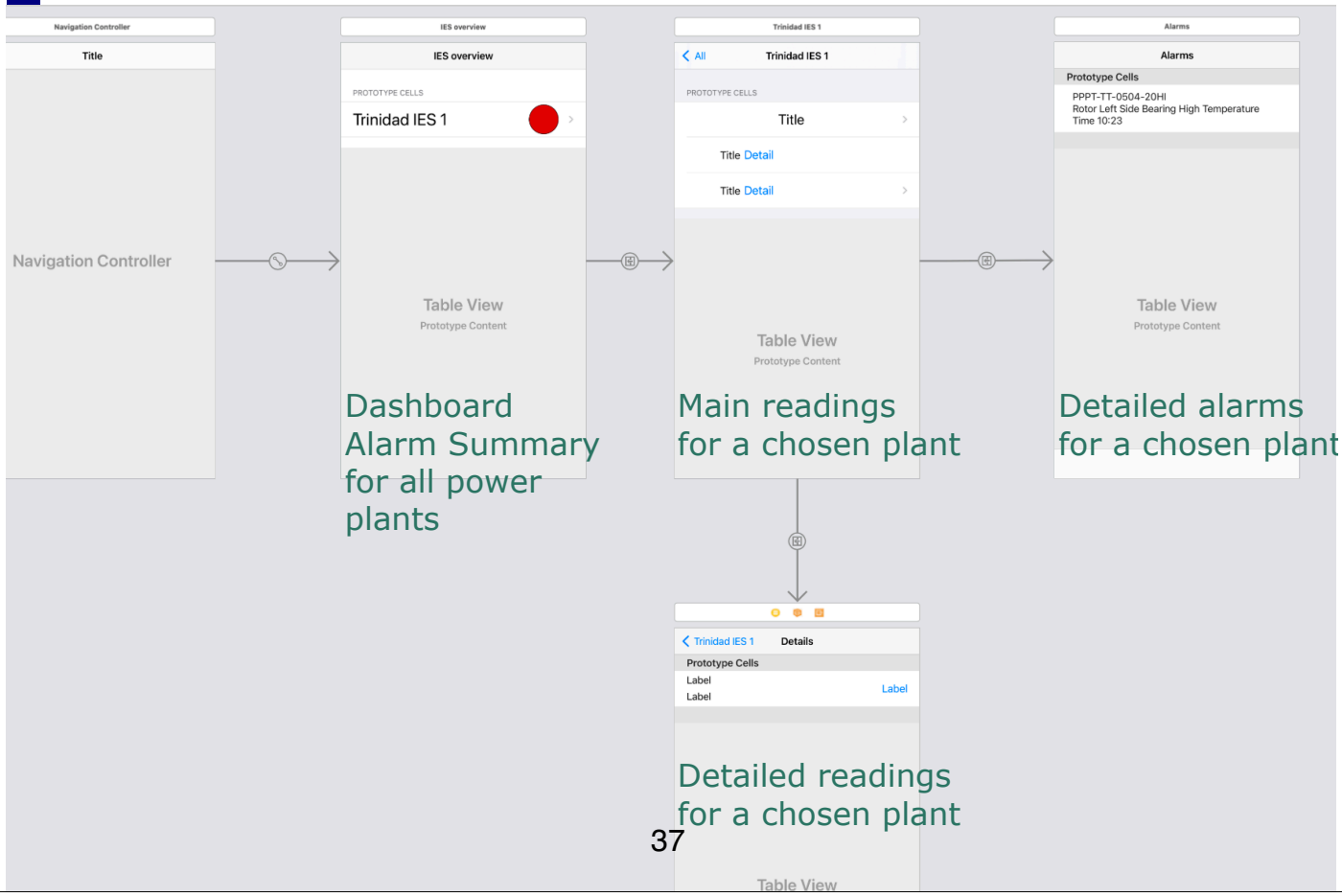
Party Talk

How do we turn an idea into an app

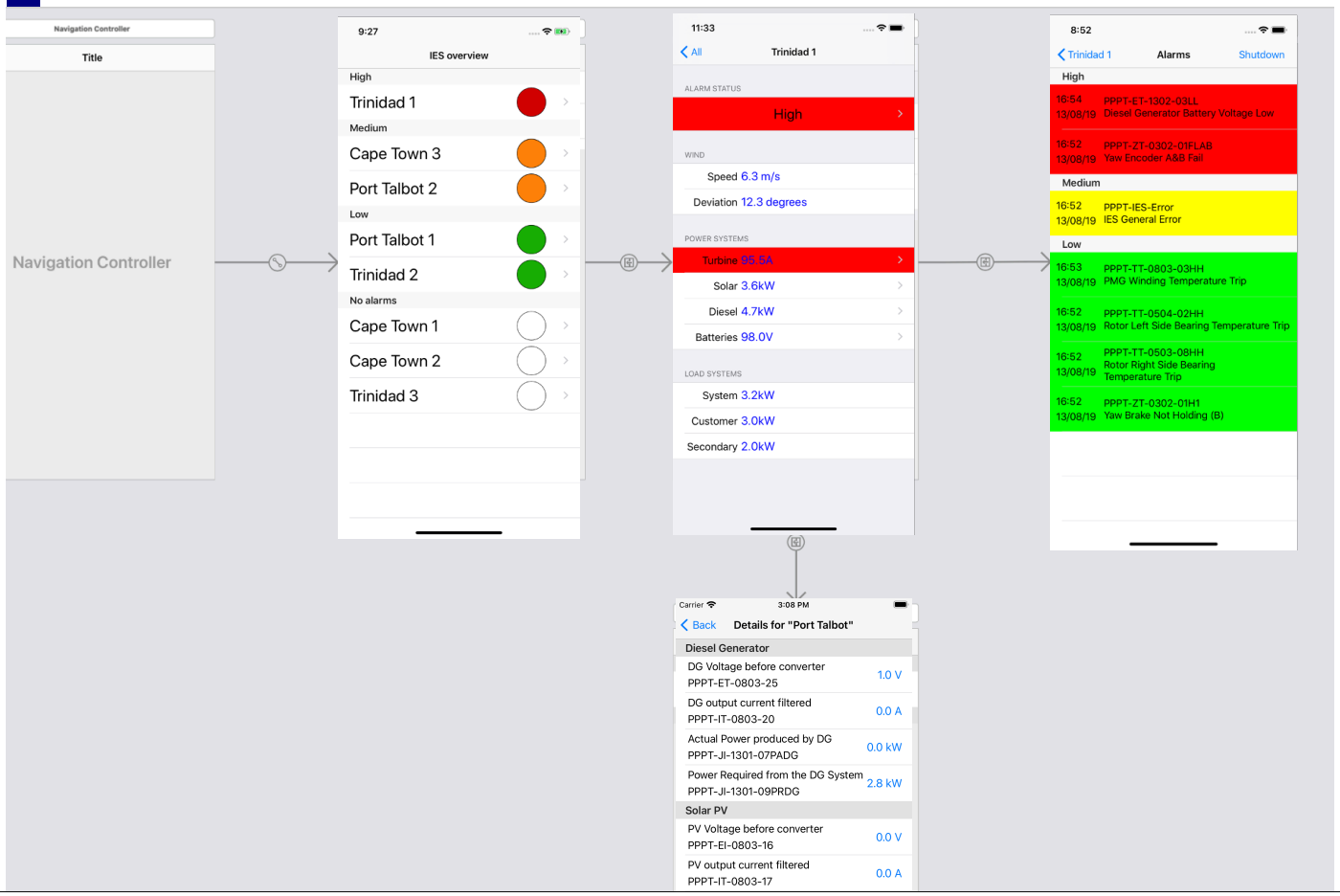
- Prototyping
- Xcode and Interface Builder can do this very efficiently and very effectively
- Build the views and add realistic data
- When we agree on the screens that will make up the app, we have made several steps towards having a real app

- Lets look at the Crossflow example again

4 types of screen in app



4 types of screen in app



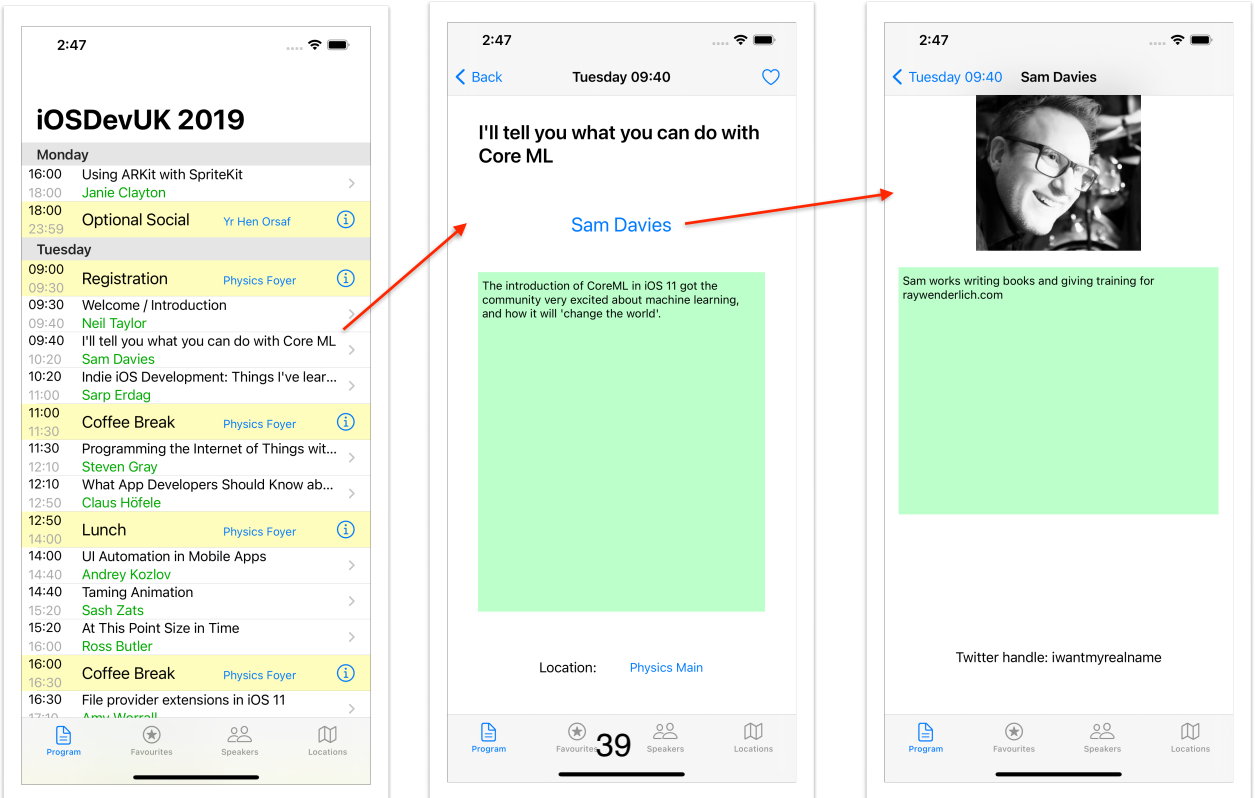
Having done this prototype

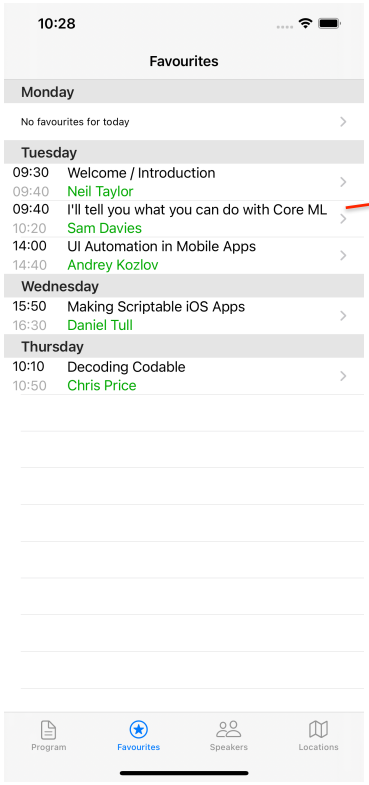
- Users can try out the prototype
- They can interact with the prototype as if it was the real app (but with static data)
- We can assess whether they have all the info they need in specific situations
- We can produce screenshots and use them in a design document
- We can make view controllers to fill out the app with its actual behaviour

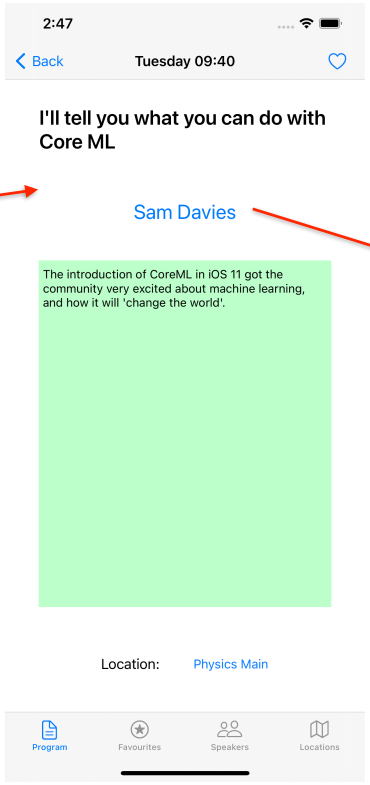
Conference Attendee App
A case study through the course

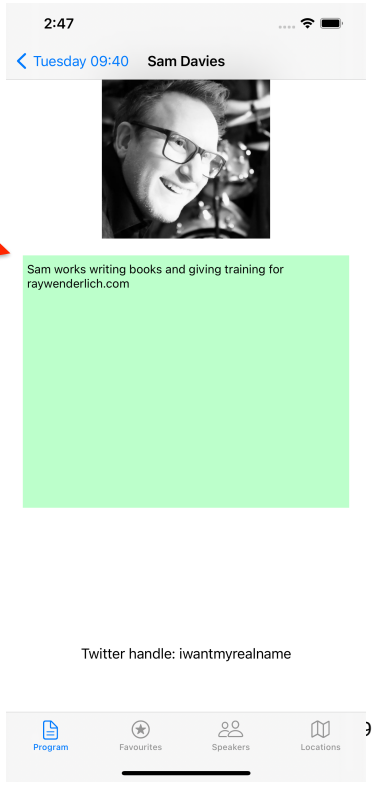
- I run the UK's largest iOS development conference each year (iosdevuk.com)
- We make an app each year so that the attendees can see:
  - the schedule of the conference
  - the details of talks
  - The details of speakers
  - Where talks, dinners etc are being held
  - And can bookmark what sessions they want to attend

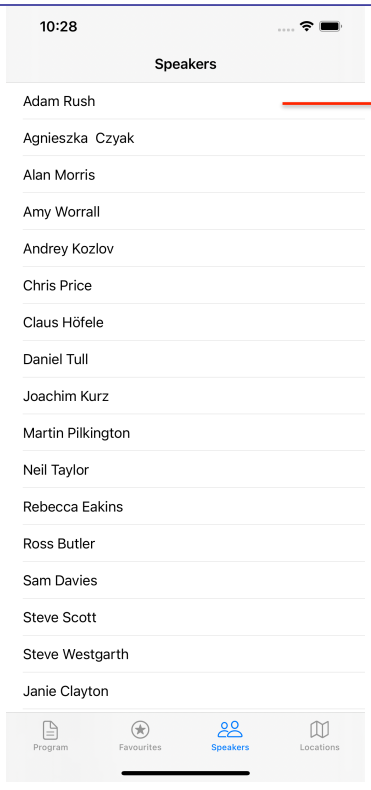
We are ready to make a non-functional prototype

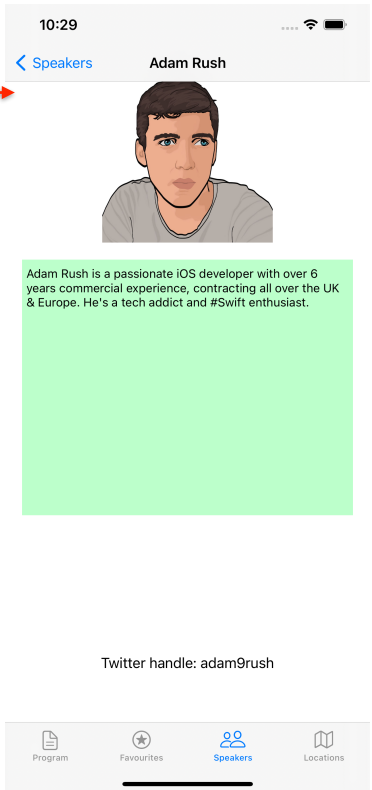



10:28
Favourites
Monday
No favourites for today
Tuesday
09:30 Welcome / Introduction
09:40 Neil Taylor
09:40 I'll tell you what you can do with Core ML
10:20 Sam Davies
14:00 UI Automation in Mobile Apps
14:40 Andrey Kozlov
Wednesday
15:50 Making Scriptable iOS Apps
16:30 Daniel Tull
Thursday
10:10 Decoding Codable
10:50 Chris Price
Program
Favourites
Speakers
Locations


2:47
Back
Tuesday 09:40
I'll tell you what you can do with Core ML
Sam Davies
The introduction of CoreML in iOS 11 got the community very excited about machine learning, and how it will 'change the world'.
Location:
Physics Main
Program
Favourites
Speakers
Locations

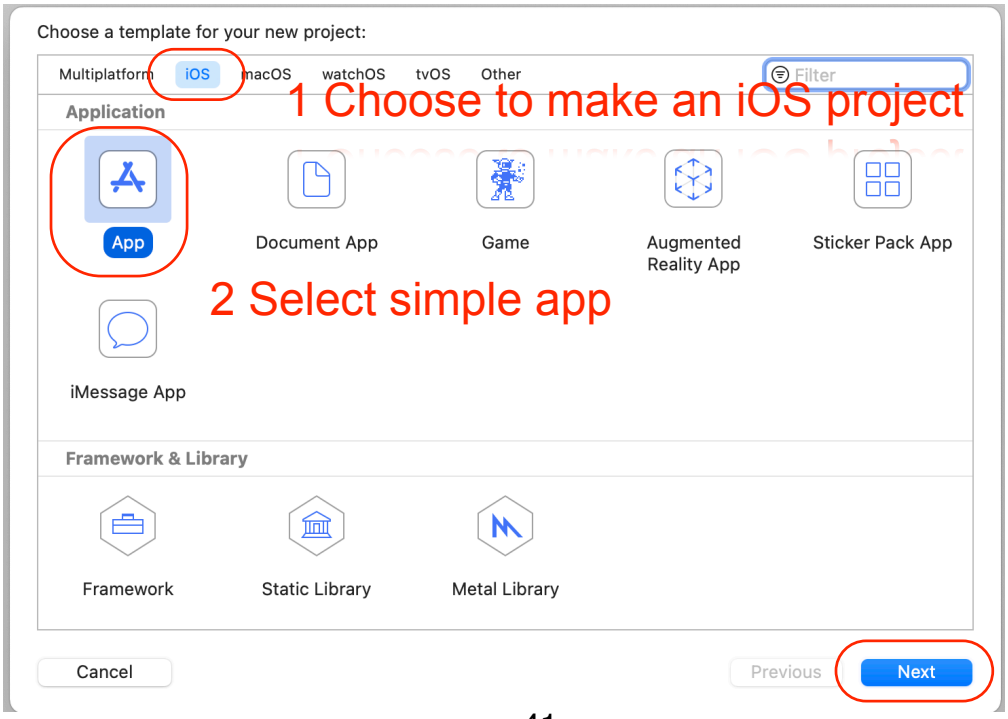

2:47
Tuesday 09:40
Sam Davies
Sam works writing books and giving training for raywenderlich.com
Twitter handle: iwantmyrealname
Program
Favourites
Speakers
Locations


10:28
Speakers
Adam Rush
Agnieszka Czyak
Alan Morris
Amy Worrall
Andrey Kozlov
Chris Price
Claus Höfele
Daniel Tull
Joachim Kurz
Martin Pilkington
Neil Taylor
Rebecca Eakins
Ross Butler
Sam Davies
Steve Scott
Steve Westgarth
Janie Clayton
Program
Favourites
Speakers
Locations


10:29
Speakers
Adam Rush
Adam Rush is a passionate iOS developer with over 6 years commercial experience, contracting all over the UK & Europe. He's a tech addict and #Swift enthusiast.
Twitter handle: adam9rush
Program
Favourites
Speakers
Locations

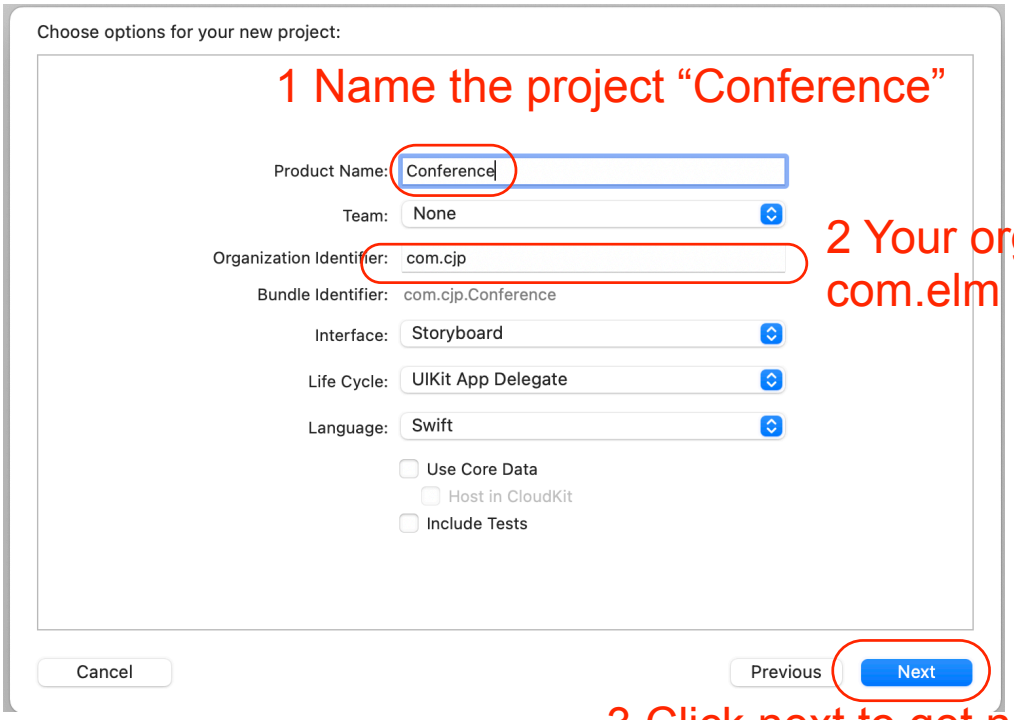
Making prototype app - Step 1

- In Xcode, choose File/New/Project
- Following screen will show:



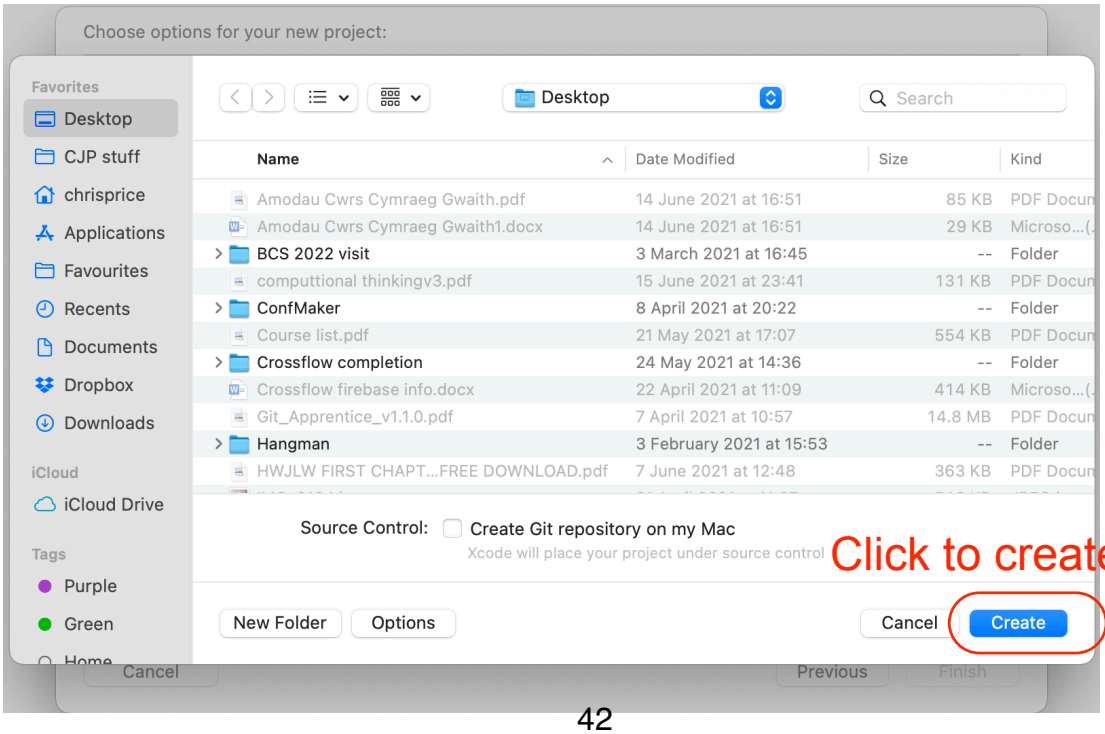
Step 2

- On this screen we fill in project details
- You need to check all details are right



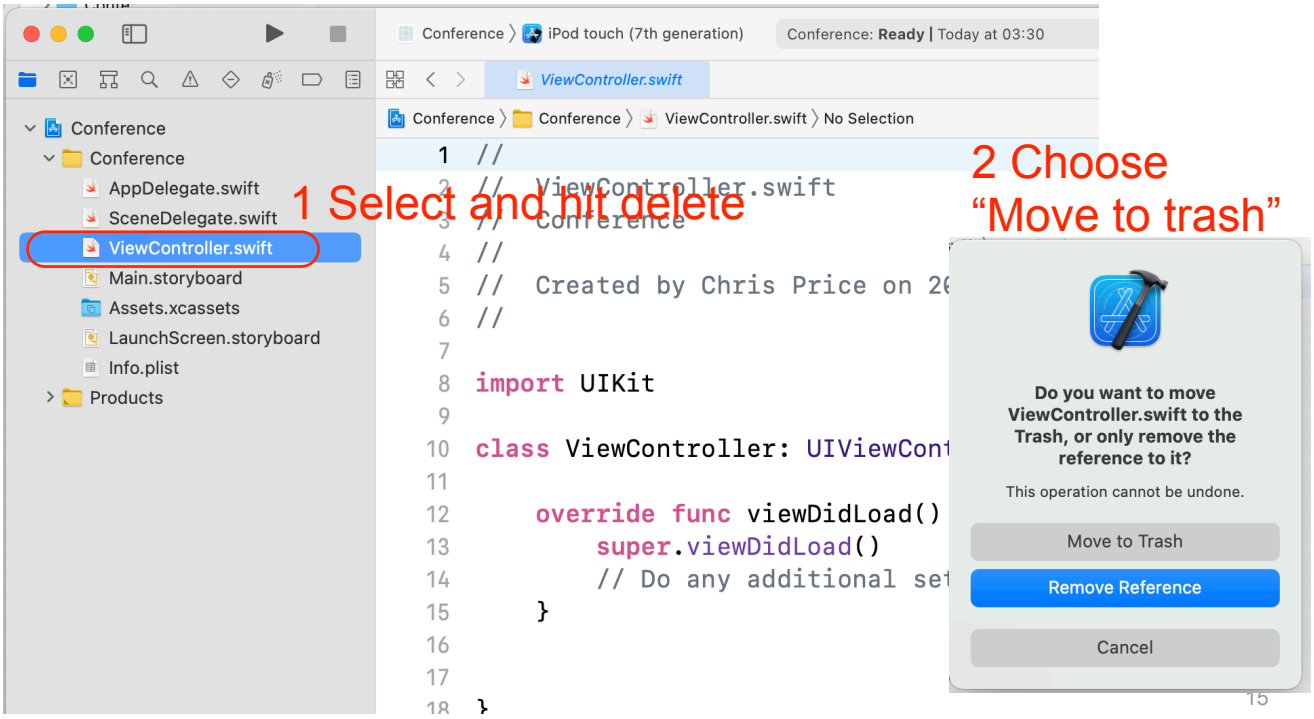
Step 3

- Final setup screen - we decide where we want to save the project, and whether we are using Git



Step 4

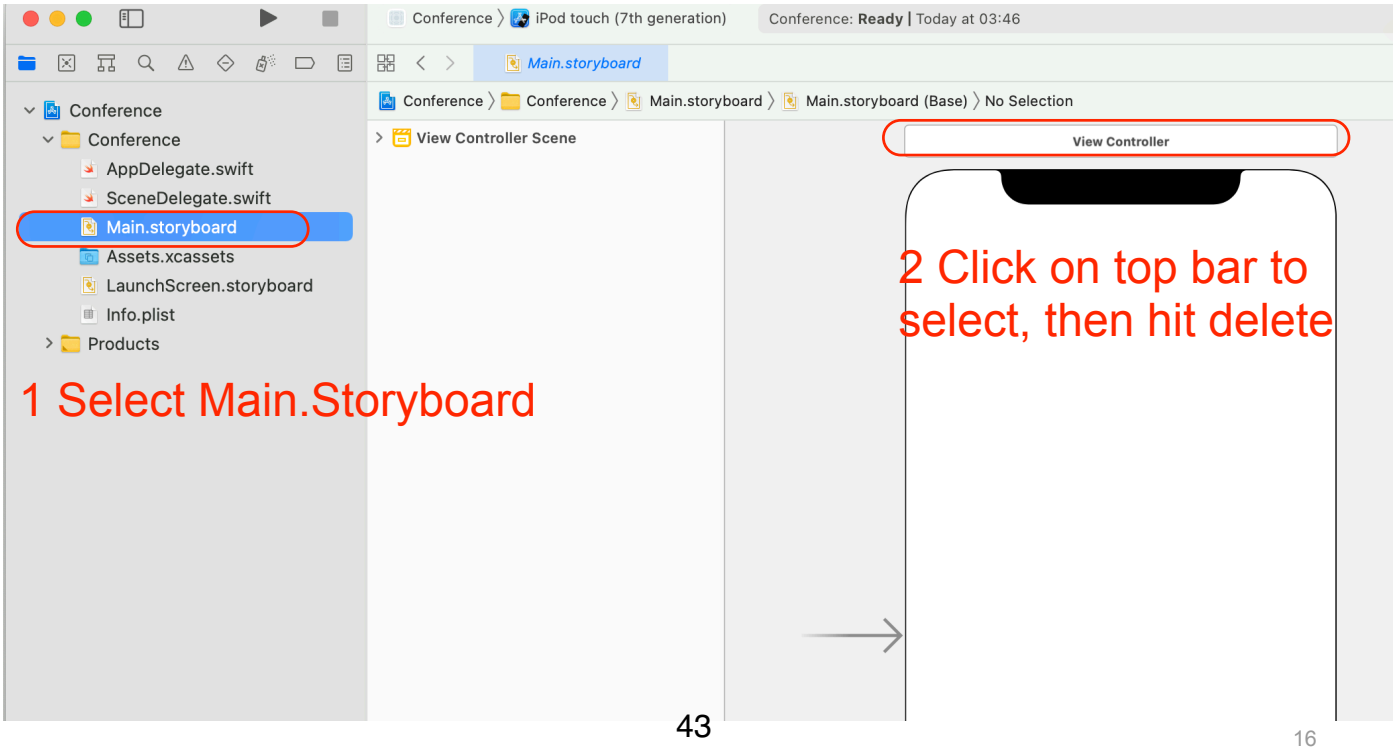
Delete ViewController.swift as we won't be using it.
•Select the file and hit delete it - choose “Move to Trash” at the prompt



15

Step 5

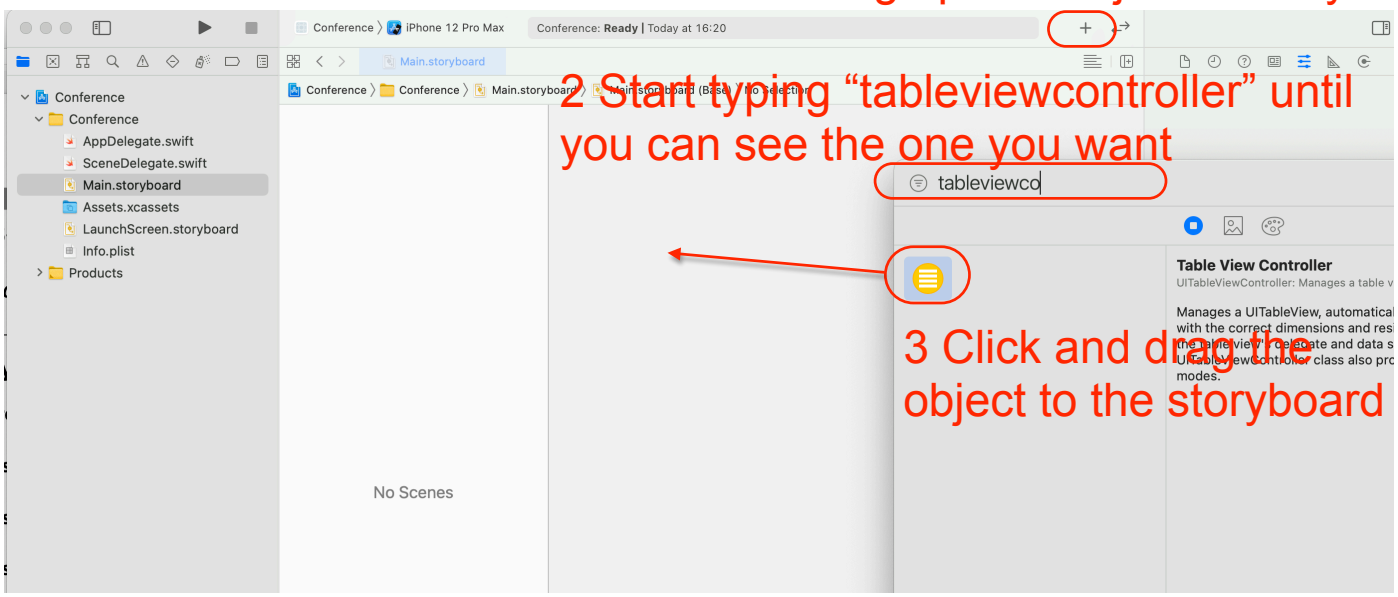
- Select Main.storyboard, then select the view controller shown in the storyboard, and hit delete



16

Step 6

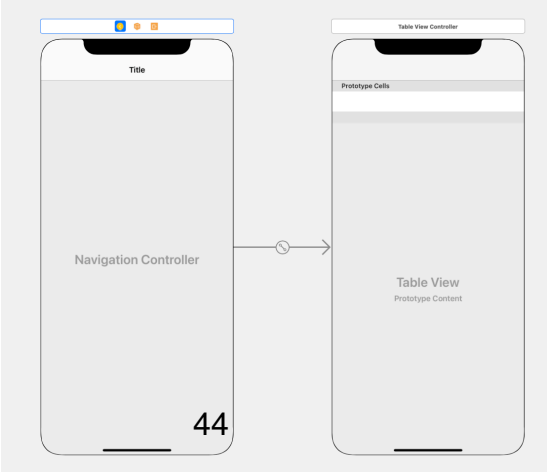
- Add a table view controller to the storyboard



Step 7

Embed the table view controller in a navigation controller

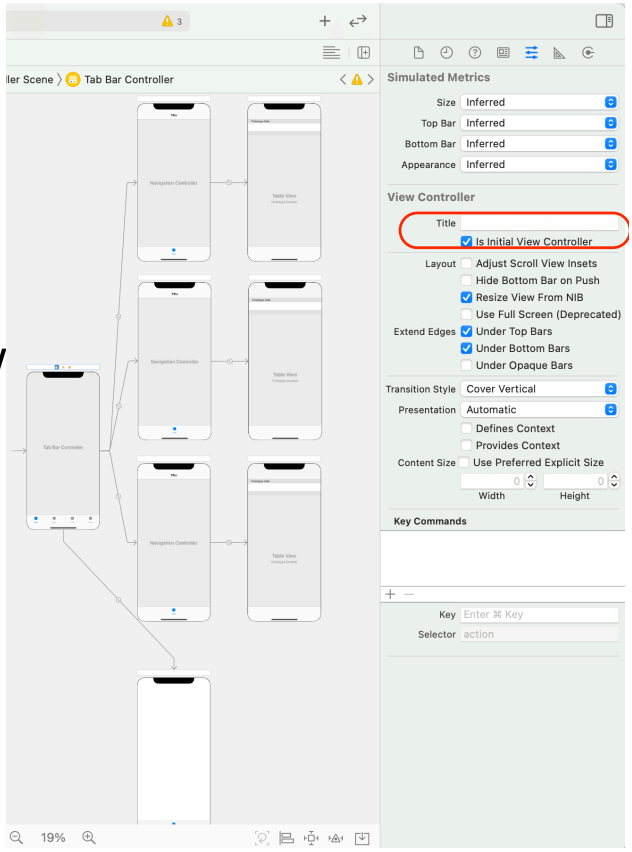
- Select the table view controller (by clicking the bar at the top in the storyboard)
- Choose menu Editor/Embed in/NavigationController
- It will then look like the screen below
- Make another two table controllers and embed each of them in a navigation controller



Step 8

Add another view controller, and make them tabs

- Add another view controller from the “+” object library
- Select all three navigation controllers and the simple view controller (click on storyboard and drag over them)
- Choose menu Editor/Embed in/TabController
- Select the tab controller and in the attribute editor on the right, select *Is InitialViewcontroller*
- This will make the tab appear when the app starts up

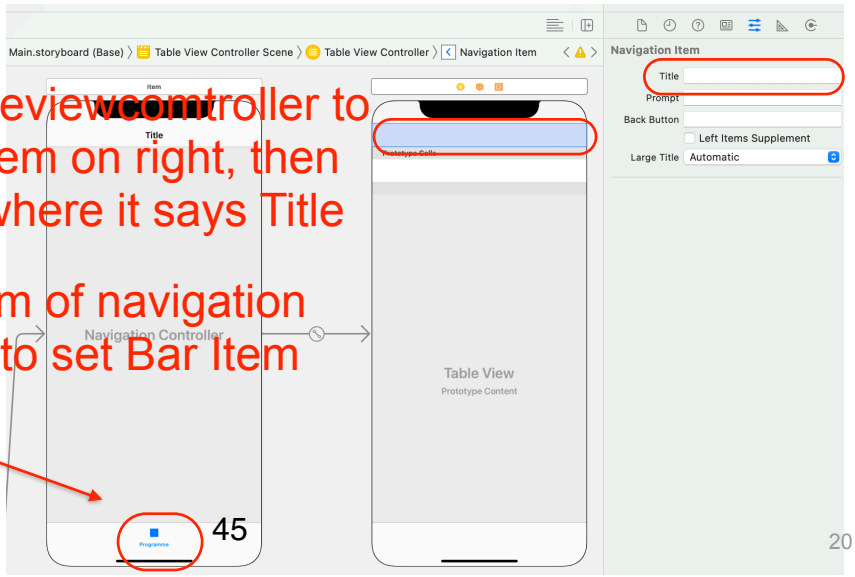


Step 9

- Click at the top of the first table view controller
- Insert the title “Programme”
- Click at the bottom of the associated navigation bar
- Type “Programme” there for the bar item. It has an associated image - select “doc.text”

1 Click at top of tableviewcontroller to get the navigation item on right, then type “Programme” where it says Title

2 Then click at bottom of navigation controller to be able to set Bar Item values

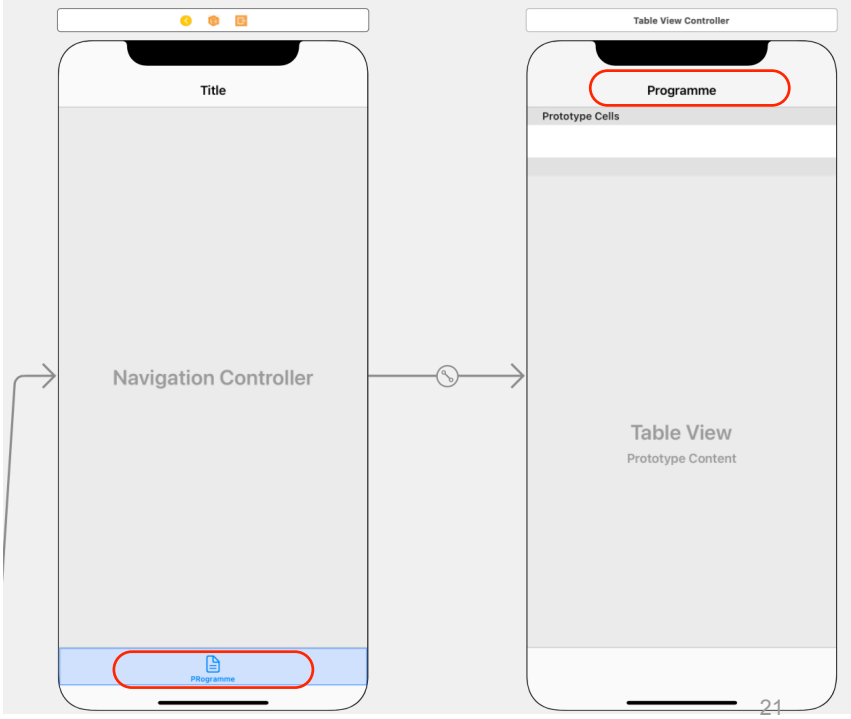


Step 10

- After step 9, the first navigation pair looks as shown

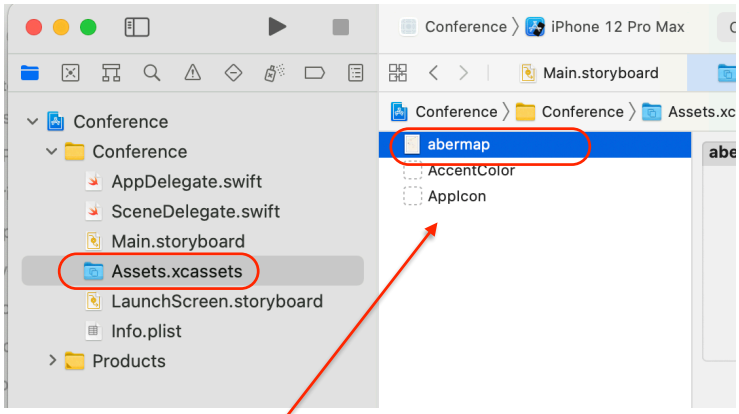
- Repeat for 2nd pair with Favourites and image star.circle

- Same for 3rd Table view controller with Speakers and image person.2

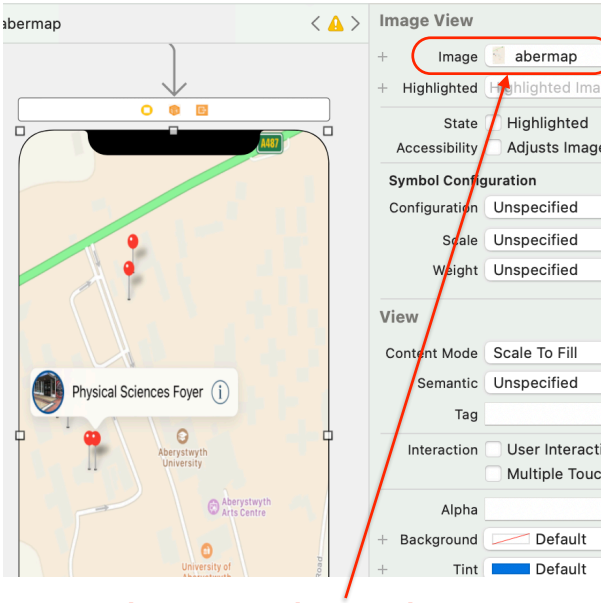


Step 11

- For the simple view controller, add the abermap.jpg provided in Session 5 to Assets, then add a image view from the object library to the view controller, and initialise it with the jpg



1 Drag abermap.jpg here



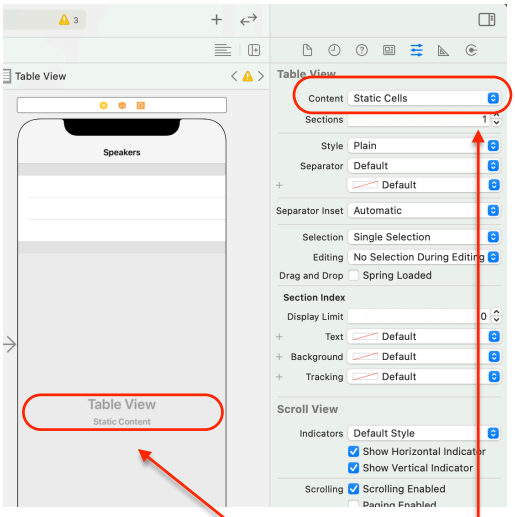
2 Having added image view, set it to abermap

Where are we?

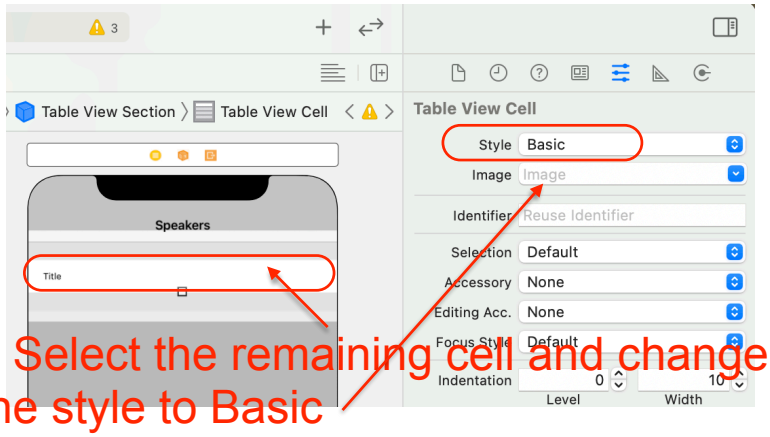
- If you run your app, it has four tabs - when you select each of the first three, you get an empty screen except for the header, and the fourth shows a map
- Next we will populate the Speaker tab with a list of Speakers, and link it to the details of a single speaker

Step 12

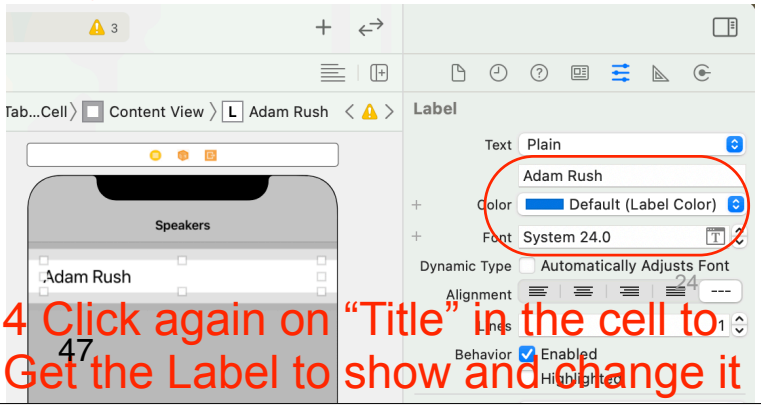
- We are going to add some static cells to the Speaker tableview controller



1 Select the Table View and set the Content to static cells
2 Click on two of the three cells created and hit delete



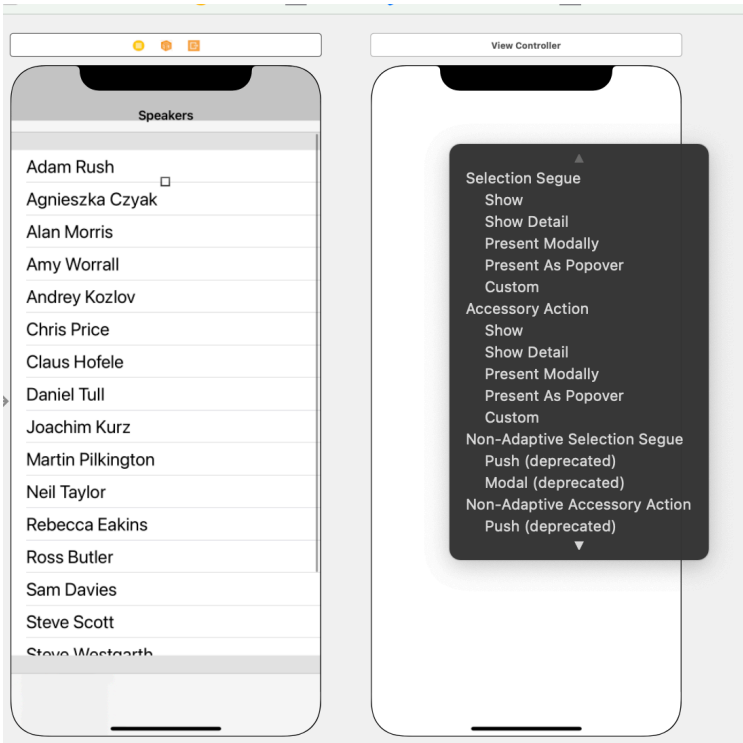
3 Select the remaining cell and change the style to Basic



4 Click again on "Title" in the cell to Get the Label to show and change it

Step 13

- Click off the table cell, then you can select it again
- You can then type command-d to duplicate the cell into a whole screen of cells, and put different names in
- Add another view controller to the right of the Speakers screen
- Click and drag from the Adam Rush cell to the new screen and choose Show on the pop up menu
- Click on the title space at the top of the new screen and add Adam Rush as a title



Step 14

- Add an image view, a text view and a label to the new screen
- Add the Adam Rush jpg to Assets, then select it to be shown on the image view
- Put some text about him in the text view
- Put his twitter handle in the label

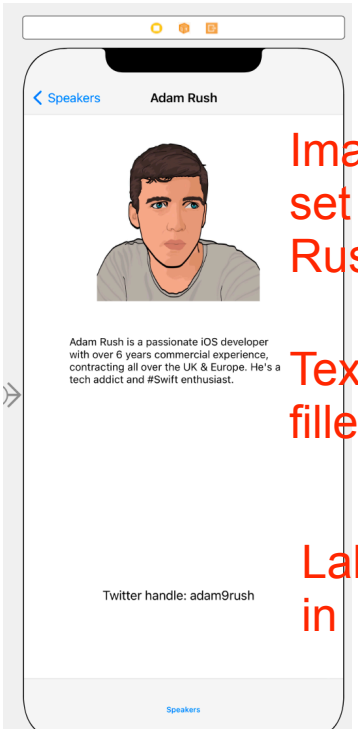


Image view, set to “Adam Rush.jpg”

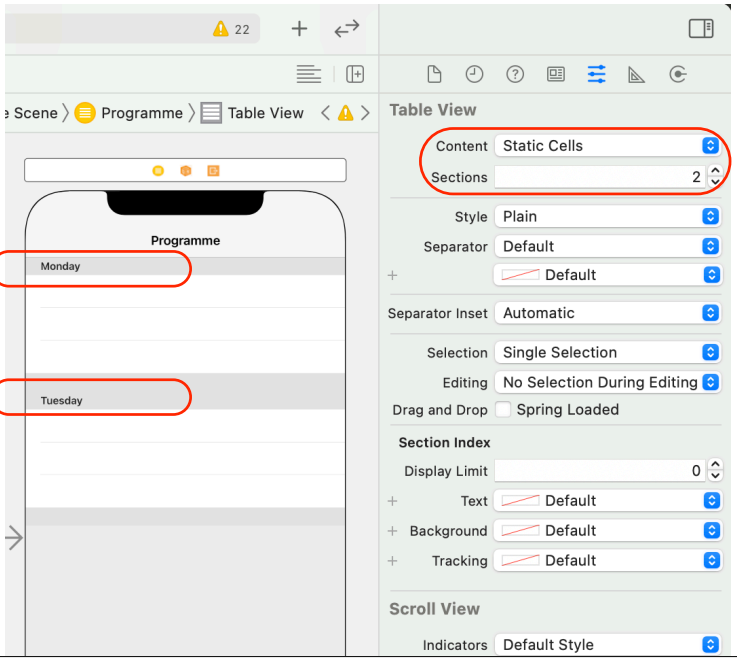
Text view, filled in

Label, filled in

- We now have the third and fourth tab choices looking good - next we will fill in the Programme

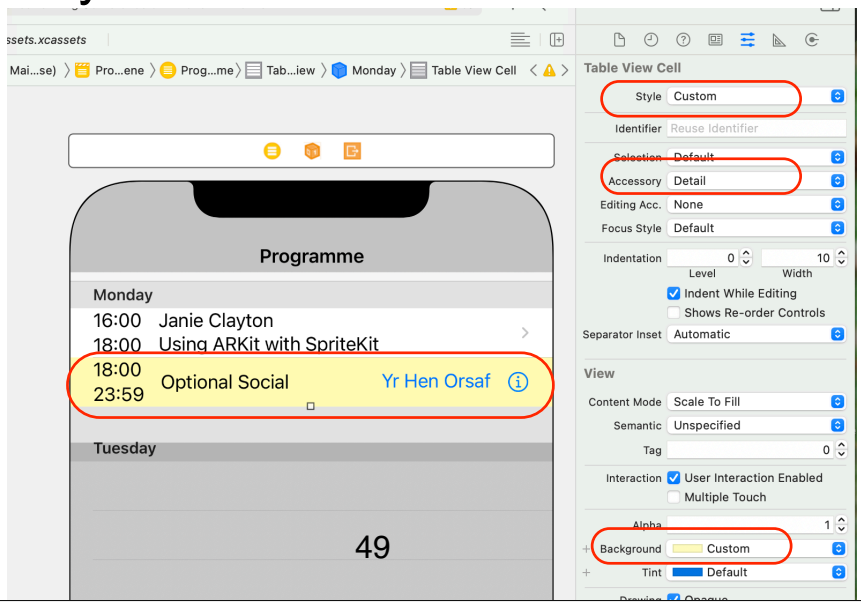
Step 15

- Make the Programme table view into static cells (as we did for Speakers in step 12)
- Change number of sections to 2
- Make section headers say “Monday” and “Tuesday”



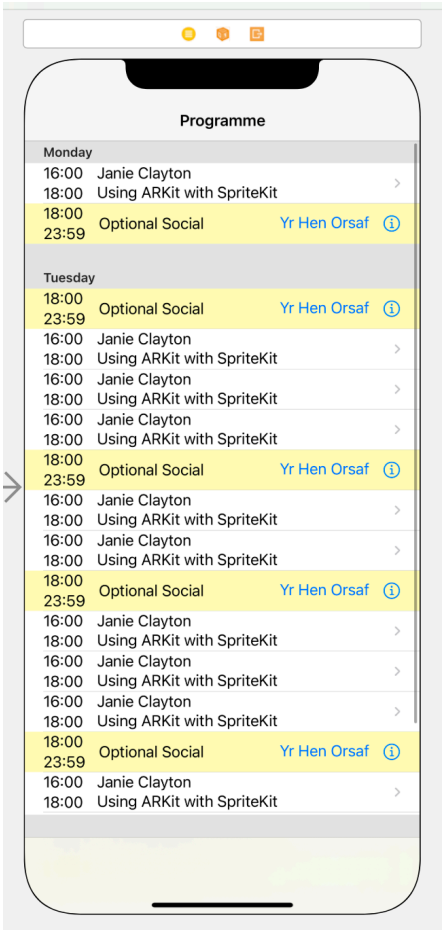
Step 16

- Add 4 labels to two types of custom table cell as shown (one cell for a talk slot, one for a break)
- Set background of break cell coloured yellow
- Add Accessory disclosure indicator to talk cell
- Add Accessory Detail to break cell



Step 17

- Having made the two types of cell, we can copy them, and paste many of them to make the correct structure for the Programme page
- We then need to change the data in the cells into representative data for our app



Finishing the app

- From here, there is nothing new to complete the prototype
- We add a talk screen with two labels, a textview and a button and we link the talk screen button to the speaker screen (or a copy of it with the correct data)
- We make the Favourites screen like the Programme screen but with less cells showing, and link it to the talk screen

We now have a complete set of app screens that we can try out on our users

We can gradually replace these fixed screens with real screens as we implement the app

30

Apple Human Interface guidelines

- Many of choices we have made might have been different on Android
- e.g. use of tabs and back buttons
- Apple have guidance for how to build apps on different platforms:
  - https://developer.apple.com/design/human-interface-guidelines/
- Worth reading to give apps native feel e.g. see navigation section for iOS

31

Session 6: Structures

- Creating structs and initialising them
- Properties and access
- Methods and using them
- Lab 6: Creating and using Structs

- This covers lesson 2.4 of Development in Swift Fundamentals

1

Unit 2—Lesson 4: Structures

Structures

```
struct Person {
  var name: String
}
```

Capitalize type names

Use lowercase for property names

Structures
Accessing property values

```
struct Person {
  var name: String
}

let person = Person(name: "Jasmine")
print(person.name)
```

```
Jasmine
```

Structures
Adding functionality

```
struct Person {
  var name: String

 func sayHello() {
    print("Hello there! My name is \(name)!")
  }

}

let person = Person(name: "Jasmine")
person.sayHello()
```

```
Hello there! My name is Jasmine!
```

Instances

```
struct Shirt {
  var size: String
  var color: String
}

let myShirt = Shirt(size: "XL", color: "blue")

let yourShirt = Shirt(size: "M", color: "red")
```

```
struct Car {
  var make: String
  var year: Int
  var color: String

  func startEngine() {...}

  func drive() {...}

  func park() {...}

  func steer(direction: Direction) {...}
}

let firstCar = Car(make: "Honda", year: 2010, color: "blue")
let secondCar = Car(make: "Ford", year: 2013, color: "black")

firstCar.startEngine()
firstCar.drive()
```

Initializers

```
let string = String.init() // ""
let integer = Int.init() // 0
let bool = Bool.init() // false
```

Initializers

```
var string = String() // ""
var integer = Int() // 0
var bool = Bool() // false
```

Initializers
Default values

```
struct Odometer {
  var count: Int = 0
}

let odometer = Odometer()
print(odometer.count)
```

```
0
```

Initializers
Memberwise initializers

```
let odometer = Odometer(count: 27000)
print(odometer.count)
```

```
27000
```

Initializers
Memberwise initializers

```
struct Person {
  var name: String
}
```

Initializers
Memberwise initializers

```
struct Person {
  var name: String

  func sayHello() {
    print("Hello there!")
  }
}

let person = Person(name: "Jasmine") // Memberwise initializer
```

```swift
struct Shirt {
  let size: String
  let color: String
}

let myShirt = Shirt(size: "XL", color: "blue") // Memberwise initializer


struct Car {
  let make: String
  let year: Int
  let color: String
}

let firstCar = Car(make: "Honda", year: 2010, color: "blue") // Memberwise initializer
```

Initializers
Custom initializers

```swift
struct Temperature {
  var celsius: Double
}

let temperature = Temperature(celsius: 30.0)
```

```swift
let fahrenheitValue = 98.6
let celsiusValue = (fahrenheitValue - 32) / 1.8

let newTemperature = Temperature(celsius: celsiusValue)
```

```swift
struct Temperature {
  var celsius: Double

  init(celsius: Double) {
    self.celsius = celsius
  }

  init(fahrenheit: Double) {
    celsius = (fahrenheit - 32) / 1.8
  }
}

let currentTemperature = Temperature(celsius: 18.5)
let boiling = Temperature(fahrenheit: 212.0)

print(currentTemperature.celsius)
print(boiling.celsius)

18.5
100.0
```

Unit 2—Lesson 3

Lab: Structures

Open and complete the exercises on page 1 of Lab – `Structures.playground`:

- Exercise - Structs, Instances, and Default Values

Instance methods

```
struct Size {
  var width: Double
  var height: Double

  func area() -> Double {
    return width * height
  }
}

var someSize = Size(width: 10.0, height: 5.5)

let area = someSize.area() // Area is assigned a value of 55.0
```

Mutating methods

```
struct Odometer {
  var count: Int = 0 // Assigns a default value to the 'count' property.
}
```

Need to

- Increment the mileage
- Reset the mileage

```
struct Odometer {
  var count: Int = 0 // Assigns a default value to the 'count' property.

  mutating func increment() {
    count += 1
  }

  mutating func increment(by amount: Int) {
    count += amount
  }

  mutating func reset() {
    count = 0
  }
}

var odometer = Odometer() // odometer.count defaults to 0
odometer.increment() // odometer.count is incremented to 1
odometer.increment(by: 15) // odometer.count is incremented to 16
odometer.reset() // odometer.count is reset to 0
```

Computed properties

```
struct Temperature {
  let celsius: Double
  let fahrenheit: Double
  let kelvin: Double
}

let temperature = Temperature(celsius: 0, fahrenheit: 32, kelvin: 273.15)
```

```
struct Temperature {
  var celsius: Double
  var fahrenheit: Double
  var kelvin: Double

  init(celsius: Double) {
    self.celsius = celsius
    fahrenheit = celsius * 1.8 + 32
    kelvin = celsius + 273.15
  }

  init(fahrenheit: Double) {
    self.fahrenheit = fahrenheit
    celsius = (fahrenheit - 32) / 1.8
    kelvin = celsius + 273.15
  }

  init(kelvin: Double) {
    self.kelvin = kelvin
    celsius = kelvin - 273.15
    fahrenheit = celsius * 1.8 + 32
  }
}

let currentTemperature = Temperature(celsius: 18.5)
let boiling = Temperature(fahrenheit: 212.0)
let freezing = Temperature(kelvin: 273.15)
```

```
struct Temperature {
  var celsius: Double

  var fahrenheit: Double {
    return celsius * 1.8 + 32
  }
}

let currentTemperature = Temperature(celsius: 0.0)
print(currentTemperature.fahrenheit)

32.0
```

Challenge
Add support for Kelvin

Modify the following to allow the temperature to be read as Kelvin

```
struct Temperature {
  let celsius: Double

  var fahrenheit: Double {
    return celsius * 1.8 + 32
  }

}
```

Hint: Temperature in Kelvin is Celsius + 273.15

```
struct Temperature {
  let celsius: Double

  var fahrenheit: Double {
    return celsius * 1.8 + 32
  }

  var kelvin: Double {
    return celsius + 273.15
  }

}

let currentTemperature = Temperature(celsius: 0.0)
print(currentTemperature.kelvin)

273.15
```

Property observers

```
struct StepCounter {
    var totalSteps: Int = 0 {
        willSet {
            print("About to set totalSteps to \(newValue)")
        }
        didSet {
            if totalSteps > oldValue  {
                print("Added \(totalSteps - oldValue) steps")
            }
        }
    }
}
```

Property observers

```
var stepCounter = StepCounter()
stepCounter.totalSteps = 40
stepCounter.totalSteps = 100
```

```
About to set totalSteps to 40
Added 40 steps
About to set totalSteps to 100
Added 60 steps
```

Type properties and methods

```
struct Temperature {
  static var boilingPoint = 100.0

  static func convertedFromFahrenheit(_ temperatureInFahrenheit: Double) -> Double {
    return(((temperatureInFahrenheit - 32) * 5) / 9)
  }

}

let boilingPoint = Temperature.boilingPoint

let currentTemperature = Temperature.convertedFromFahrenheit(99)

let biggestInt = Int.max
```

Copying

```
var someSize = Size(width: 250, height: 1000)
var anotherSize = someSize

someSize.width = 500

print(someSize.width)
print(anotherSize.width)
```

```
500
250
```

self

```
struct Car {
  var color: Color

  var description: String {
    return "This is a \(self.color) car."
  }
}
```

self
When not required

Not required when property or method names exist on the current object

```
struct Car {
  var color: Color

  var description: String {
    return "This is a \(color) car."
  }
}
```

self
When required

```
struct Temperature {
  var celsius: Double

  init(celsius: Double) {
    self.celsius = celsius
  }
}
```

Unit 2—Lesson 3
Lab: Structures

Open again and complete exercises on pages 3 and 5 of
`Lab - Structures.playground`

Session 7: Classes

- Inheritance in Swift
- Initializers
- Methods and properties
- Overriding methods and properties
- Difference between classes and structs
- When to use classes and structs
- Lab 7: Writing classes

- This covers lesson 2.5 of Development in Swift Fundamentals

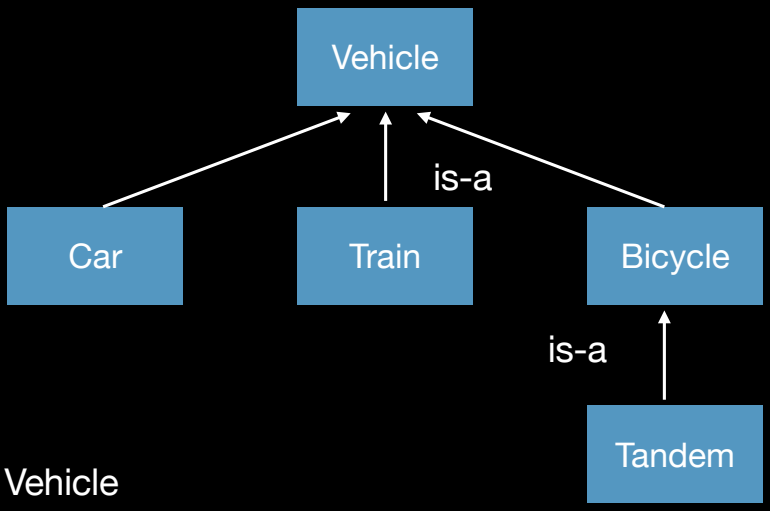
1

Unit 2—Lesson 5: Classes, Inheritance

```
class Person {
  let name: String

  init(name: String) {
    self.name = name
  }

  func sayHello() {
    print("Hello there!")
  }
}

let person = Person(name: "Jasmine")
print(person.name)
person.sayHello()

Jasmine
Hello there!
```

Inheritance

Base class



Car is Subclass of Vehicle

Vehicle is Superclass of Car

Inheritance
Defining a base class

```
class Vehicle {
   var currentSpeed = 0.0

   var description: String {
      return "traveling at \(currentSpeed) miles per hour"
   }

   func makeNoise() {
      // do nothing - an arbitrary vehicle doesn't necessarily make a noise
   }
}

let someVehicle = Vehicle()
print("Vehicle: \(someVehicle.description)")
```

```
Vehicle: traveling at 0.0 miles per hour
```

Inheritance
Create a subclass

```
class SomeSubclass: SomeSuperclass {
    // subclass definition goes here
}
```

```
class Bicycle: Vehicle {
    var hasBasket = false
}
```

Inheritance
Create a subclass

```
class Bicycle: Vehicle {
    var hasBasket = false
}

let bicycle = Bicycle()
bicycle.hasBasket = true

bicycle.currentSpeed = 15.0
print("Bicycle: \(bicycle.description)")
```

```
Bicycle: traveling at 15.0 miles per hour
```

Inheritance
Create a subclass

```
class Tandem: Bicycle {
    var currentNumberOfPassengers = 0
}
```

Inheritance
Create a subclass

```
class Tandem: Bicycle {
    var currentNumberOfPassengers = 0
}

let tandem = Tandem()
tandem.hasBasket = true
tandem.currentNumberOfPassengers = 2
tandem.currentSpeed = 22.0
print("Tandem: \(tandem.description)")
```

```
Tandem: traveling at 22.0 miles per hour
```

Inheritance
Override methods and properties

```
class Train: Vehicle {
    override func makeNoise() {
        print("Choo Choo!")
    }
}

let train = Train()
train.makeNoise()
```

```
Choo Choo!
```

Inheritance
Override methods and properties

```
class Car: Vehicle {
    var gear = 1
    override var description: String {
        return super.description + " in gear \(gear)"
    }
}
```

Inheritance
Override methods and properties

```
class Car: Vehicle {
    var gear = 1
    override var description: String {
        return super.description + " in gear \(gear)"
    }
}

let car = Car()
car.currentSpeed = 25.0
car.gear = 3
print("Car: \(car.description)")
```

```
Car: traveling at 25.0 miles per hour in gear 3
```

Inheritance
Override initializer

```
class Person {
  let name: String

  init(name: String) {
    self.name = name
  }
}

class Student: Person {
  var favoriteSubject: String
}
```

Class ‘Student’ has no initializers

Inheritance
Override initializer

```
class Person {
  let name: String

  init(name: String) {
    self.name = name
  }
}

class Student: Person {
  var favoriteSubject: String
  init(name: String, favoriteSubject: String) {
    self.favoriteSubject = favoriteSubject
    super.init(name: name)
  }
}
```

References

- When you create an instance of a class:
  - Swift returns the address of that instance
  - The returned address is assigned to the variable
- When you assign the address of an instance to multiple variables:
  - Each variable contains the same address
  - Update one instance, and all variables refer to the updated instance

```swift
class Person {
  let name: String
  var age: Int

  init(name: String, age: Int) {
    self.name = name
    self.age = age
  }
}

var jack = Person(name: "Jack", age: 24)
var myFriend = jack

jack.age += 1

print(jack.age)
print(myFriend.age)
```

```
25
25
```


```swift
struct Person {
  let name: String
  var age: Int
}

var jack = Person(name: "Jack", age: 24)
var myFriend = jack

jack.age += 1

print(jack.age)
print(myFriend.age)
```

```
25
24
```

Memberwise initializers

- Swift does not create memberwise initializers for classes
- Common practice is for developers to create their own for their defined classes

Class or structure?

- Start new types as structures
- Use a class:
  - When you're working with a framework that uses classes
  - When you want to refer to the same instance of a type in multiple places
  - When you want to model inheritance

Unit 2, Lesson 4

Lab: Classes.playground

Open and complete the exercises on pages 1 and 2 in Lab – `Classes.playground`

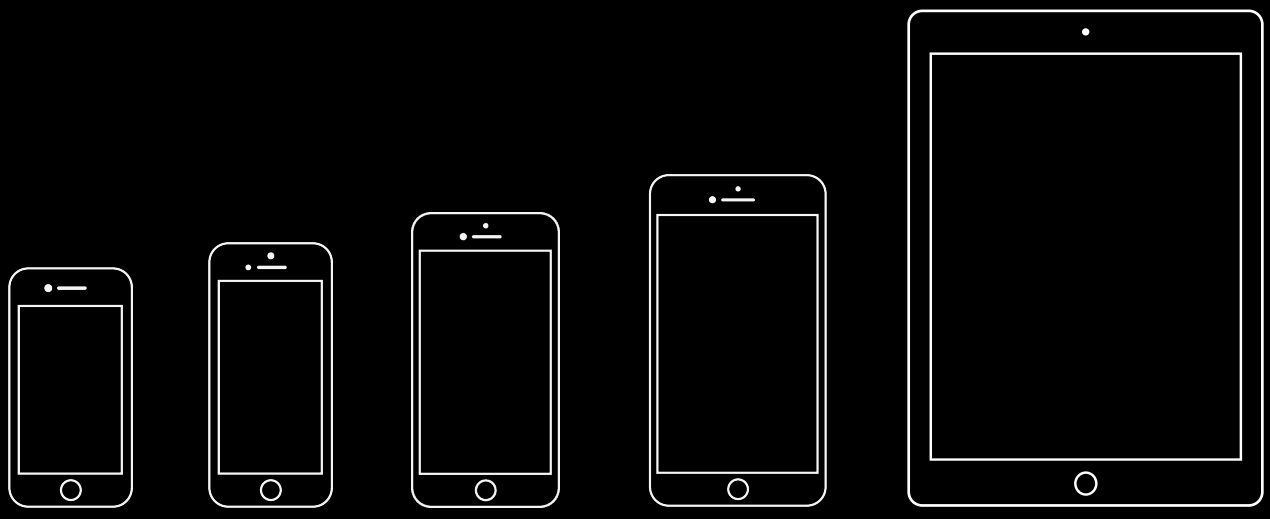
Session 8: Working with different screen sizes

- Autolayout
- Stack layout
- Lab 8: Fitting many items on different size screens
- Further exercises on Autolayout
- This covers lesson 2.11 of Development in Swift Fundamentals

1

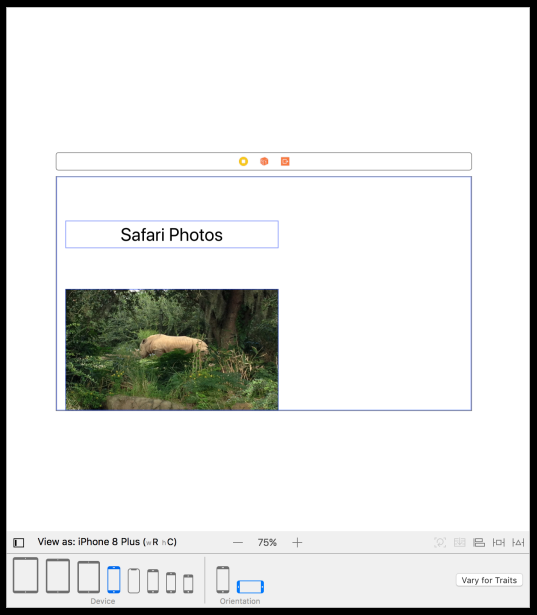
Lesson 2—11:
Auto Layout and Stack Views

Layout for multiple sizes

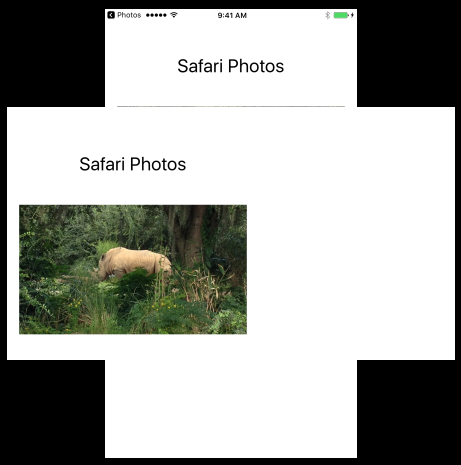


Interface Builder

View as:

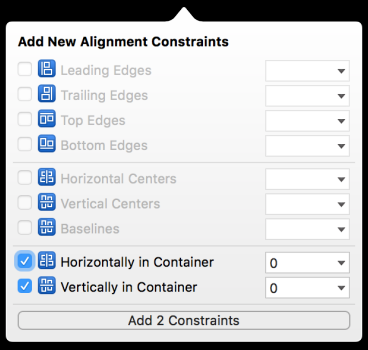


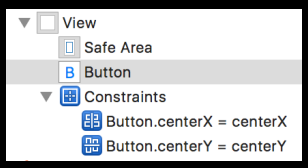
Why Auto Layout?

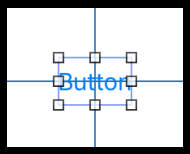


Create alignment constraints

Text

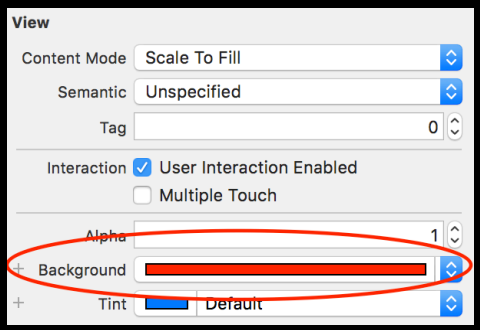


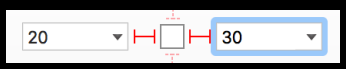




Create size constraints

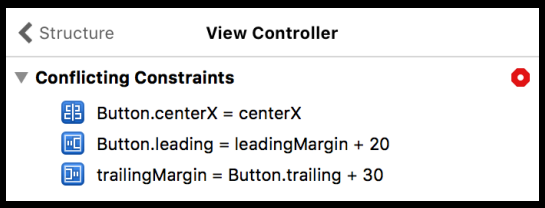
Text

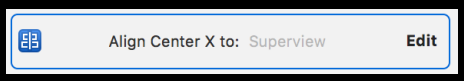




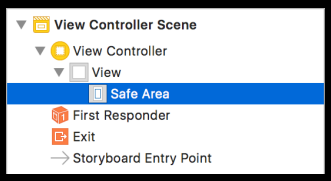
Resolve constraint issues

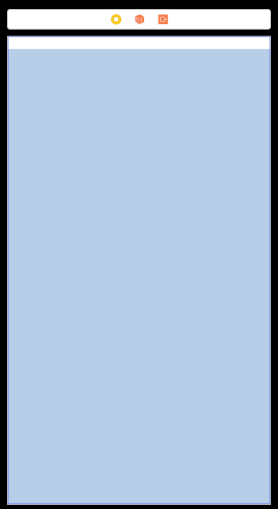
Text





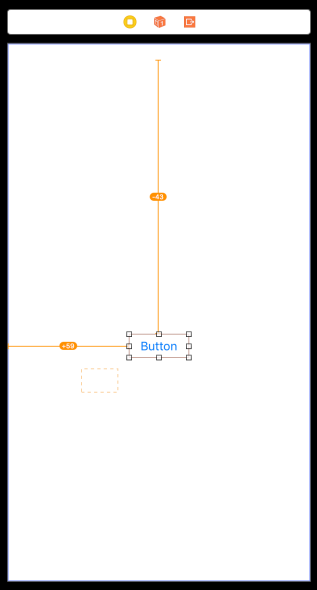
Safe area layout guide

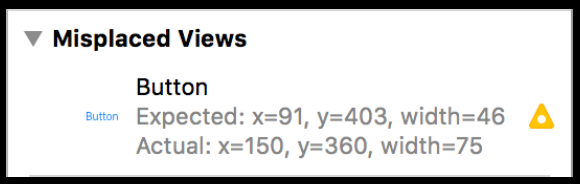




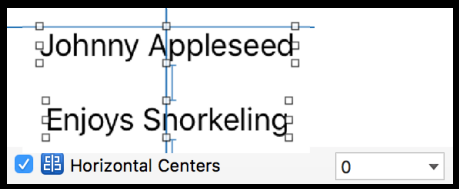
Resolve constraint warnings

Text



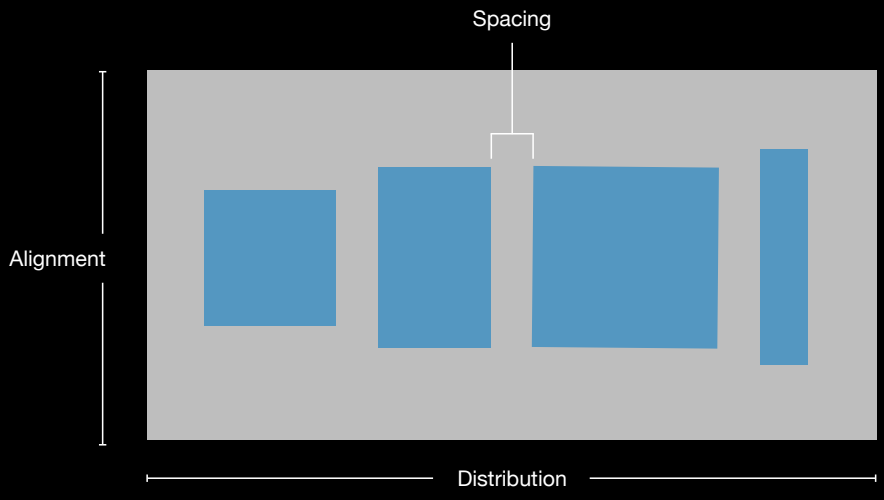


Constraints between siblings

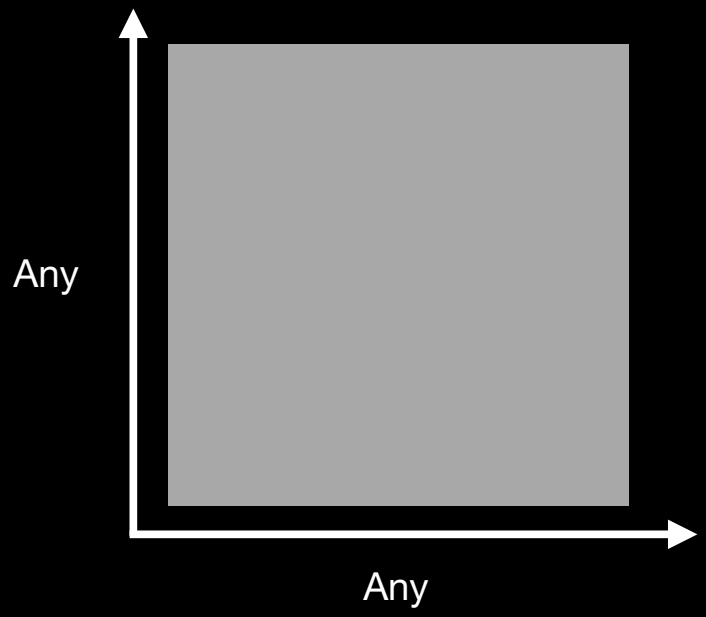


Stack views
Text

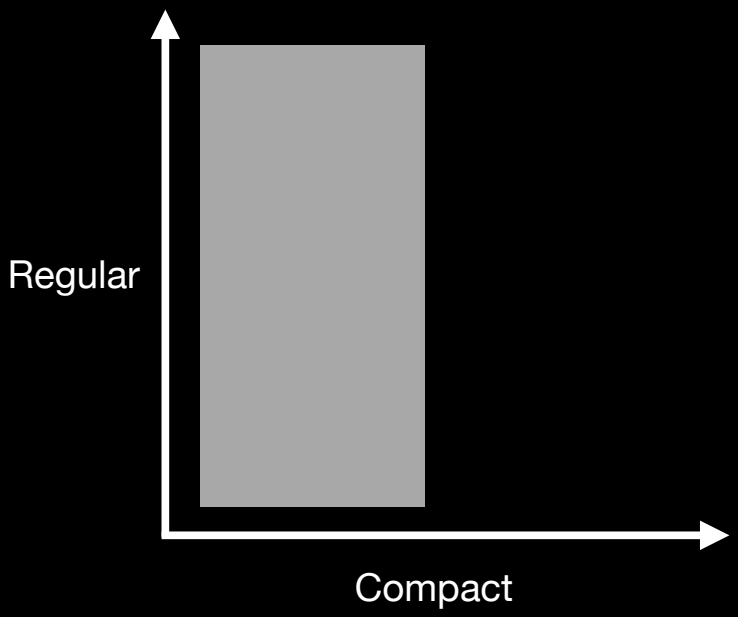
Stack views
Stack view attributes



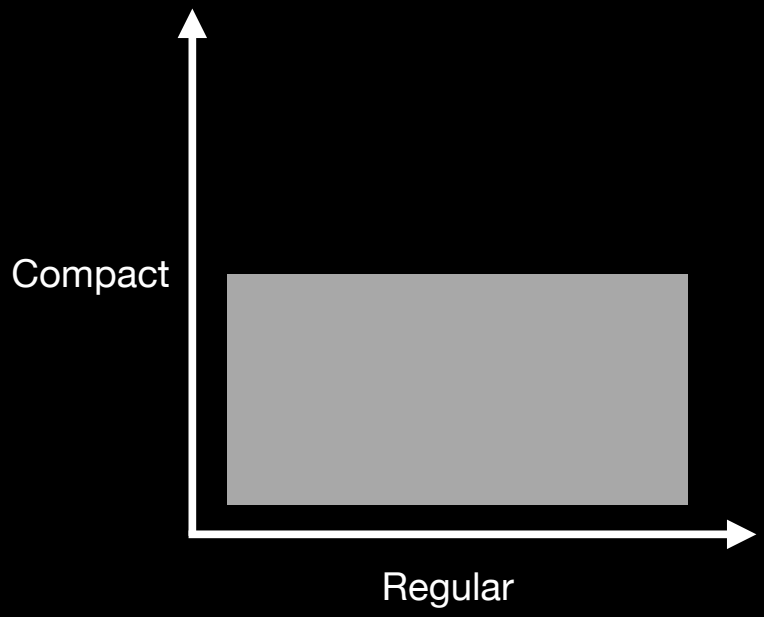
Size classes



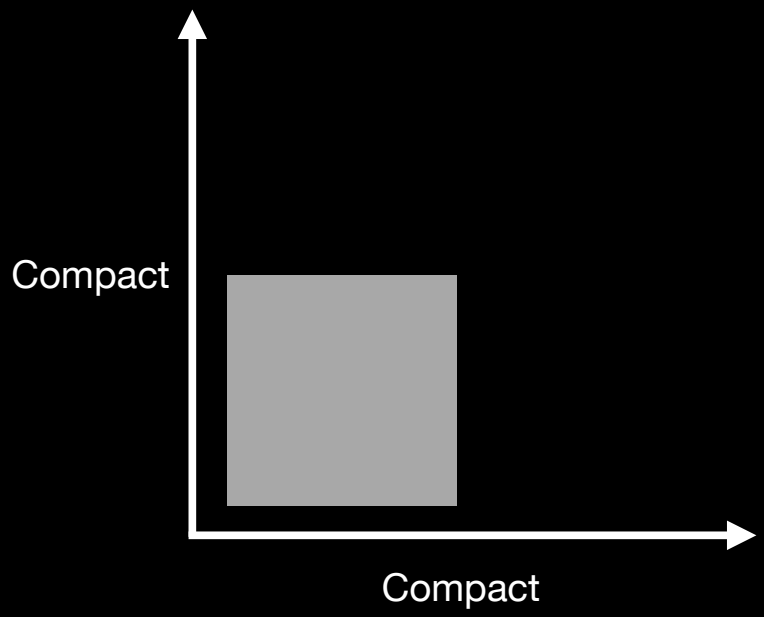
Size classes



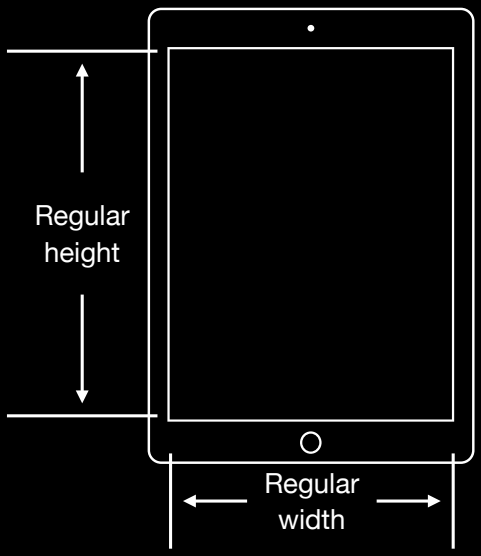
Size classes

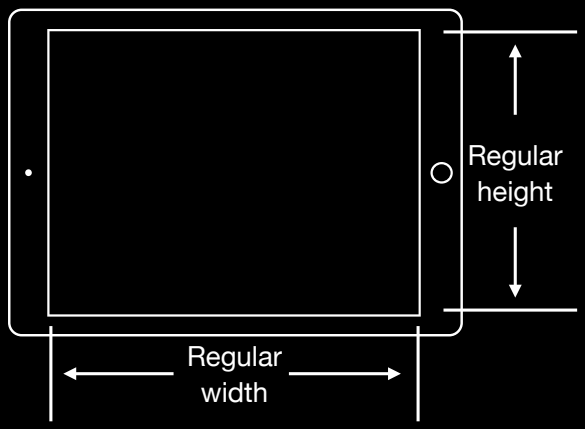


Size classes

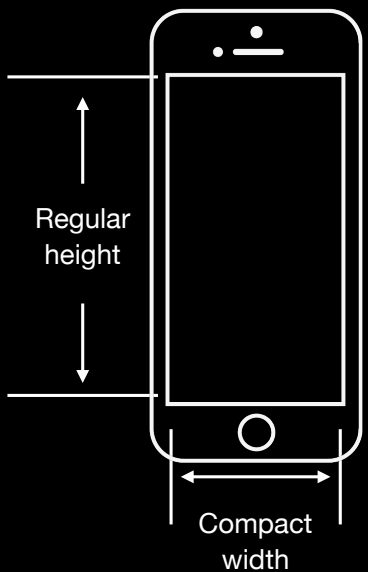


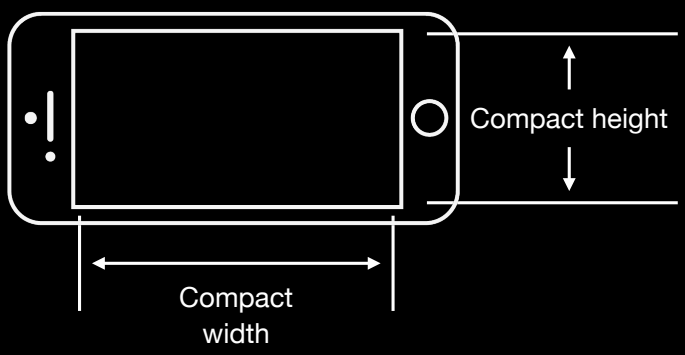
Size classes
iPad





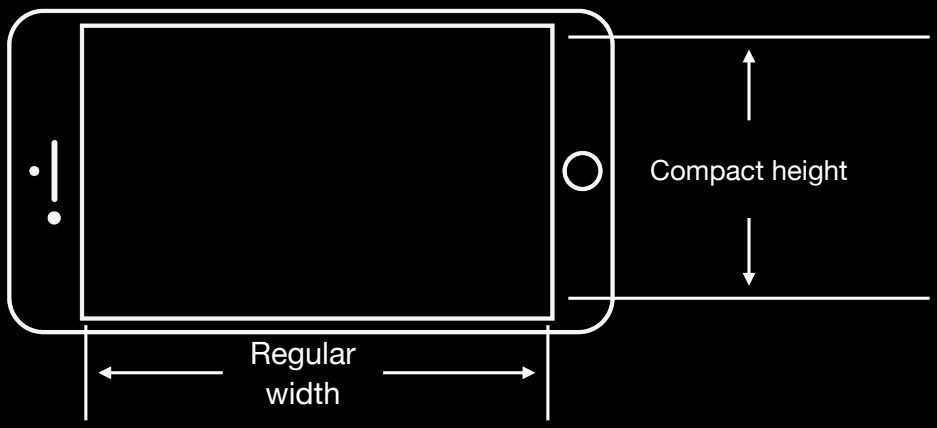
Size classes
iPhone





Size classes

iPhone 6 Plus, iPhone 7 Plus, and iPhone 8 Plus



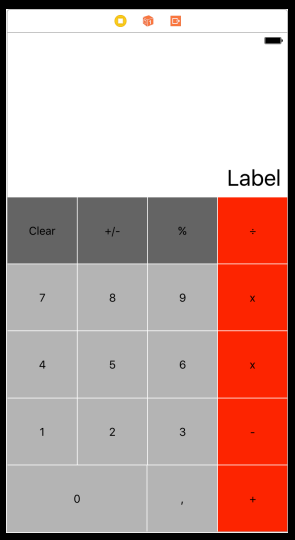
Unit 2—Lesson 10

Auto Layout and Stack Views

Learn the fundamentals of Auto Layout for building precisely designed user interfaces.

Unit 2—Lesson 10
Lab: Calculator

Use view objects, constraints, and stack views to create a simple calculator that maintains its layout on all device sizes.



Session 9: More language features
Dictionaries/Optionals / Guard / Scope / Enums

- Dictionaries/why we need optionals
- How to use them correctly
- Making code tidy with Guard
- Understanding scope of variables
- Lab 9: Optionals and Guard
- Enums and how to use them

- This covers lessons 2.6, 3.2, 3.4, 3.5, 3.6 of Development in Swift Fundamentals

1

Unit 2—Lesson 6: Collections

(We have done Arrays, so will just do Dictionaries, page 228)

Dictionaries

[key1 : value1, key2: value2, key3: value3]

```
var scores = ["Richard": 500, "Luke": 400, "Cheryl": 800]
```

```
var myDictionary = [String: Int]()
var myDictionary = Dictionary<String, Int>()
var myDictionary: [String: Int] = [:]
```

Add/remove/modify a dictionary
Adding or modifying

```
var scores = ["Richard": 500, "Luke": 400, "Cheryl": 800]

scores["Oli"] = 399

let oldValue = scores.updateValue(100, forKey: "Richard")
```

Add/remove/modify a dictionary
Adding or modifying

```
var scores = ["Richard": 500, "Luke": 400, "Cheryl": 800]

scores["Oli"] = 399

if let oldValue = scores.updateValue(100, forKey: "Richard") {
  print("Richard's old value was \(oldValue)")
}
```

```
Richard's old value was 500
```

Add/remove/modify a dictionary
Removing

```
var scores = ["Richard": 100, "Luke": 400, "Cheryl": 800]
scores["Richard"] = nil
print(scores)

if let oldValue = scores.removeValue(forKey: "Luke") {
  print("Luke's score was \(oldValue) before he stopped playing")
}
print(scores)
```

```
["Cheryl": 800, "Luke": 400]
Luke's score was 400 before he stopped playing
["Cheryl": 800]
```

Accessing a dictionary

```
var scores = ["Richard": 500, "Luke": 400, "Cheryl": 800]

let players = Array(scores.keys) //["Richard", "Luke", "Cheryl"]
let points = Array(scores.values) //[500, 400, 800]

if let myScore = scores["Luke"] {
  print(myScore)
}
```

```
400
```

```
if let henrysScore = scores["Henry"] {
  print(henrysScore)
}
```

Unit 2—Lesson 5
Lab: Collections

Open and complete exercise 3 (Dictionaries) in

`Lab - Collections.playground`

Unit 3—Lesson 1: Optionals

We just saw this stuff with Dictionaries
This is an example of Optionals

```
var scores = ["Richard": 500, "Luke": 400, "Cheryl": 800]

scores["Oli"] = 399

if let oldValue = scores.updateValue(100, forKey: "Richard") {
  print("Richard's old value was \(oldValue)")
}
```

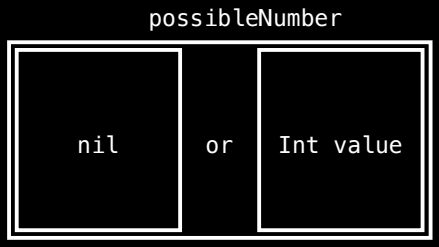
```
Richard's old value was 500
```

Functions and optionals
Return values

```
let numberString = "123"
let possibleNumber = Int(numberString)
```

```
let numberString = "Cynthia"
let possibleNumber = Int(numberString)
```

Type of Int(String) is an Optional Integer



Working with optional values
Force-unwrap - dangerous if value is nil

```
let possibleNumber = Int(numberString)
if possibleNumber != nil {
  let actualNumber = possibleNumber!
  print(actualNumber)
}

let unwrappedNumber = possibleNumber!
```

error: Execution was interrupted

Specifying the type of an optional

```
var serverResponseCode = 404
```

```
var serverResponseCode = nil
```

‘nil’ requires a contextual type

```
var serverResponseCode: Int? = 404
```

```
var serverResponseCode: Int? = nil
```

Working with optional values
Optional binding

```
if let constantName = someOptional {
  //constantName has been safely unwrapped for use within the braces.
}
```

```
if let possibleNumber = Int(numberString) {
  print("The value of the string was \(possibleNumber)")
}
else {
  print(“That was not a legal integer value”)
}
```

Functions and optionals
Defining

```
func printFullName(firstName: String, middleName: String?, lastName: String)

func textFromURL(url: URL) -> String?
```

Optional chaining

```
class Person {
  var age: Int
  var residence: Residence?
}

class Residence {
  var address: Address?
}

class Address {
  var buildingNumber: String?
  var streetName: String?
  var apartmentNumber: String?
}
```

Optional chaining

```
if let theResidence = person.residence {
  if let theAddress = theResidence.address {
    if let theApartmentNumber = theAddress.apartmentNumber {
      print("He/she lives in apartment number \(theApartmentNumber).")
    }
  }
}
```

Optional chaining

```
if let theApartmentNumber = person.residence?.address?.apartmentNumber {
  print("He/she lives in apartment number \(theApartmentNumber).")
}
```

Implicitly Unwrapped Optionals

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

Unwraps automatically

Should only be used when need to initialize an object without supplying the value and you'll be giving the object a value soon afterwards

Unit 3—Lesson 4: Guard

```
func singHappyBirthday() {
  if birthdayIsToday {
    if invitedGuests > 0 {
      if cakeCandlesLit {
        print("Happy Birthday to you!")
      } else {
        print("The cake candle's haven't been lit.")
      }
    } else {
      print("It's just a family party.")
    }
  } else {
    print("No one has a birthday today.")
  }
}
```

```
func singHappyBirthday() {
  guard birthdayIsToday else {
    print("No one has a birthday today.")
    return
  }

  guard invitedGuests > 0 else {
    print("It's just a family party.")
    return
  }

  guard cakeCandlesLit else {
    print("The cake's candles haven't been lit.")
    return
  }

  print("Happy Birthday to you!")
}
```

guard

```
guard condition else {
  //false: execute some code
}

//true: execute some code
```

guard

```
func divide(_ number: Double, by divisor: Double) {
  if divisor != 0.0 {
    let result = number / divisor
    print(result)
  }
}
```

```
func divide(_ number: Double, by divisor: Double) {
  guard divisor != 0.0 else { return }

  let result = number / divisor
  print(result)
}
```

guard with optionals

```
if let eggs = goose.eggs {
  print("The goose laid \(eggs.count) eggs.")
}
//`eggs` is not accessible here
```

```
guard let eggs = goose.eggs else { return }
//`eggs` is accessible hereafter
print("The goose laid \(eggs.count) eggs.")
```

guard with optionals

```
func processBook(title: String?, price: Double?, pages: Int?) {
  if let theTitle = title, let thePrice = price, let thePages = pages {
    print("\(theTitle) costs $\(thePrice) and has \(thePages) pages.")
  }
}
```

```
func processBook(title: String?, price: Double?, pages: Int?) {
  guard let theTitle = title, let thePrice = price, let thePages = pages else { return }
  print("\(theTitle) costs $\(thePrice) and has \(thePages) pages.")
}
```

Unit 3—Lesson 5:
Constant and Variable Scope

Scope

Global scope—Defined outside of a function

Local scope—Defined within braces (`{` `}`)

```
var globalVariable = true

if globalVariable {
 let localVariable = 7
}
```

Scope

```
var age = 55

func printMyAge() {
  print("My age: \(age)")
}

print(age)
printMyAge()
```

```
55
My age: 55
```

Scope

```
func printBottleCount() {
    let bottleCount = 99
    print(bottleCount)
}

printBottleCount()
print(bottleCount)
```

Use of unresolved identifier 'bottleCount'

Scope

```
func printTenNames() {
  var name = "Richard"
  for index in 1...10 {
    print("\(index): \(name)")
  }
  print(index)
  print(name)
}

printTenNames()
```

Use of unresolved identifier 'index'

Variable shadowing

```
let points = 100

for index in 1...3 {
  let points = 200
  print("Loop \(index): \(points+index)")
}
print(points)
```

```
Loop 1: 201
Loop 2: 202
Loop 3: 203
100
```

Variable shadowing

```
var name: String? = "Robert"

if let name = name {
 print("My name is \(name)")
}
```

Variable shadowing

```
func exclaim(name: String?) {
  if let name = name {
    print("Exclaim function was passed: \(name)")
  }
}
```

```
func exclaim(name: String?) {
  guard let name = name else { return }
  print("Exclaim function was passed: \(name)")
}
```

Shadowing and initializers

```
struct Person {
  var name: String
  var age: Int
}

let todd = Person(name: "Todd", age: 50)
print(todd.name)
print(todd.age)
```

```
Todd
50
```

Shadowing and initializers

```
struct Person {
  var name: String
  var age: Int

  init(name: String, age: Int) {
    self.name = name
    self.age = age
  }
}
```

Unit 3, Lesson 1
Lab: Optionals.playground

Open and complete the exercises in `Lab - Optionals.playground`

Unit 3—Lesson 3
Lab: Guard

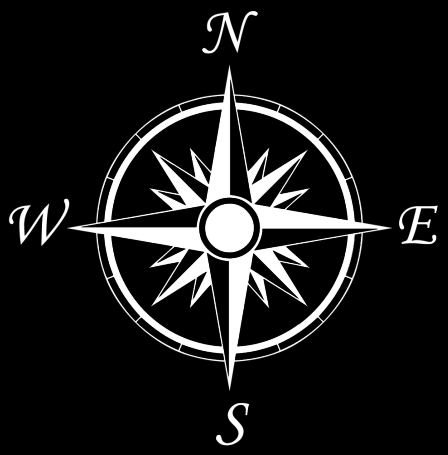
Open and complete the exercises in `Lab - Guard.playground`

Unit 3—Lesson 4
Lab: Constant and Variable Scope

Open and complete the exercises in `Lab - Scope.playground`

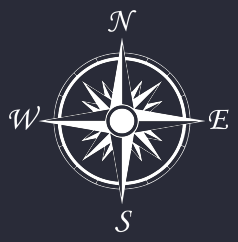
Unit 3—Lesson 6: Enumerations

Enumerations



Enumerations

```
enum CompassPoint {
  case north
  case east
  case south
  case west
}
```



Enumerations

```
enum CompassPoint {
  case north, east, south, west
}
```

```
var compassHeading = CompassPoint.west
```

```
var compassHeading: CompassPoint = .west
compassHeading = .north
```

Control flow

```
let compassHeading: CompassPoint = .west

switch compassHeading {
  case .north:
    print("I am heading north")
  case .east:
    print("I am heading east.")
  case .south:
    print("I am heading south")
  case .west:
    print("I am heading west")
}
```

Control flow

```
let compassHeading: CompassPoint = .west

if compassHeading == .west {
  print("I am heading west")
}
```

Type safety benefits

```
struct Movie {
  var name: String
  var releaseYear: Int?
  var genre: String
}

let movie = Movie(name: "Finding Dory", releaseYear: 2016, genre: "Aminated")
```

Type safety benefits

```
enum Genre {
  case animated, action, romance, documentary, biography, thriller
}

struct Movie {
  var name: String
  var releaseYear: Int?
  var genre: Genre
}

let movie = Movie(name: "Finding Dory", releaseYear: 2016, genre: .animated)
```

Unit 3—Lesson 5
Lab: Enumerations

Open and complete the exercises in `Lab - Enumerations.playground`

Session 10: Advanced navigation

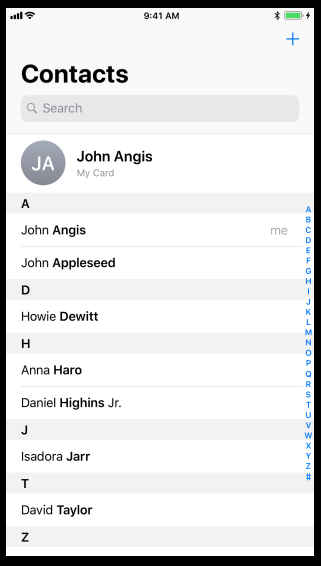
- Segues
- Navigation
- Tab bars
- Lifetime of an app
- *Lab 10: Making complex apps*

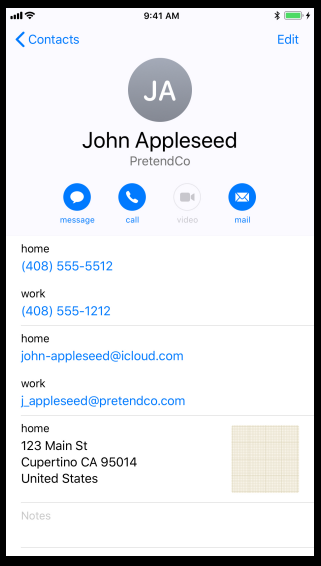
- This covers lessons 3.7, 3.8, 3.9 of Development in Swift Fundamentals

1

Unit 3—Lesson 7:
Segues and Navigation Controllers

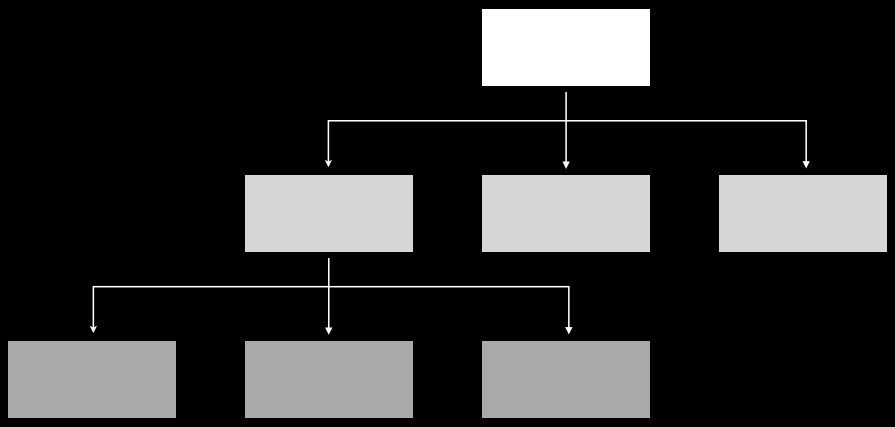
Segues and navigation controllers





Navigation hierarchy

Hierarchical



Segues (UIStoryboardSegue)

A `UIStoryboardSegue` object performs the visual transition between two view controllers

It is also used to prepare for the transition from one view controller to another

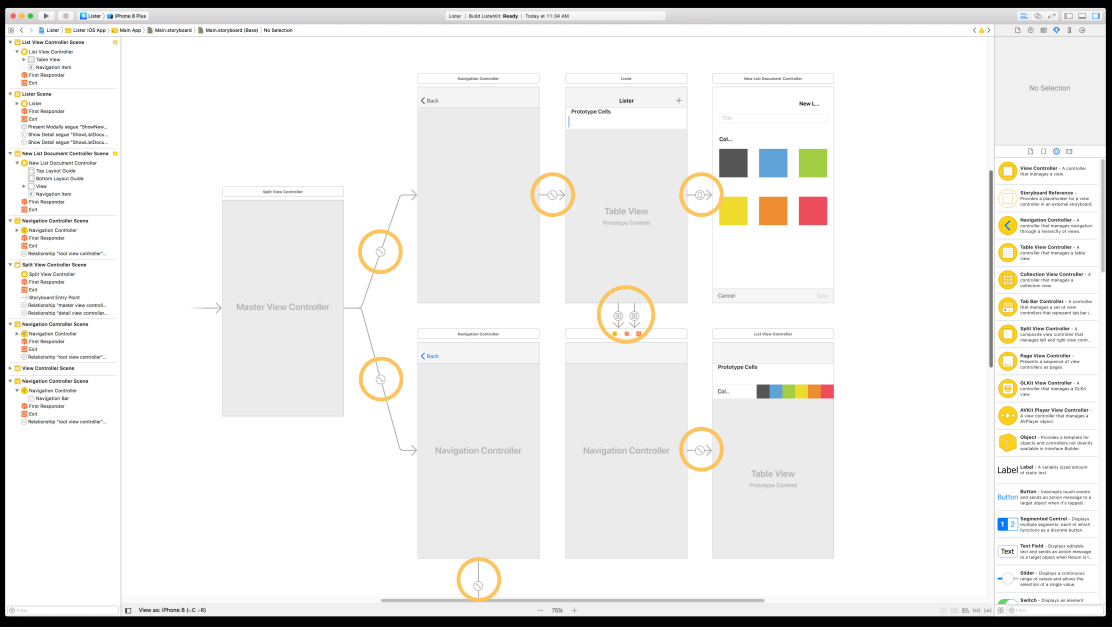
Segue objects contain information about the view controllers that are involved in a transition

When a segue is triggered, before the visual transition occurs, the storyboard runtime can call certain methods in the current view controller

Useful if you need to pass information forward

Segues (UIStoryboardSegue)

Segues between scenes



Segues (UIStoryboardSegue)

Unwind

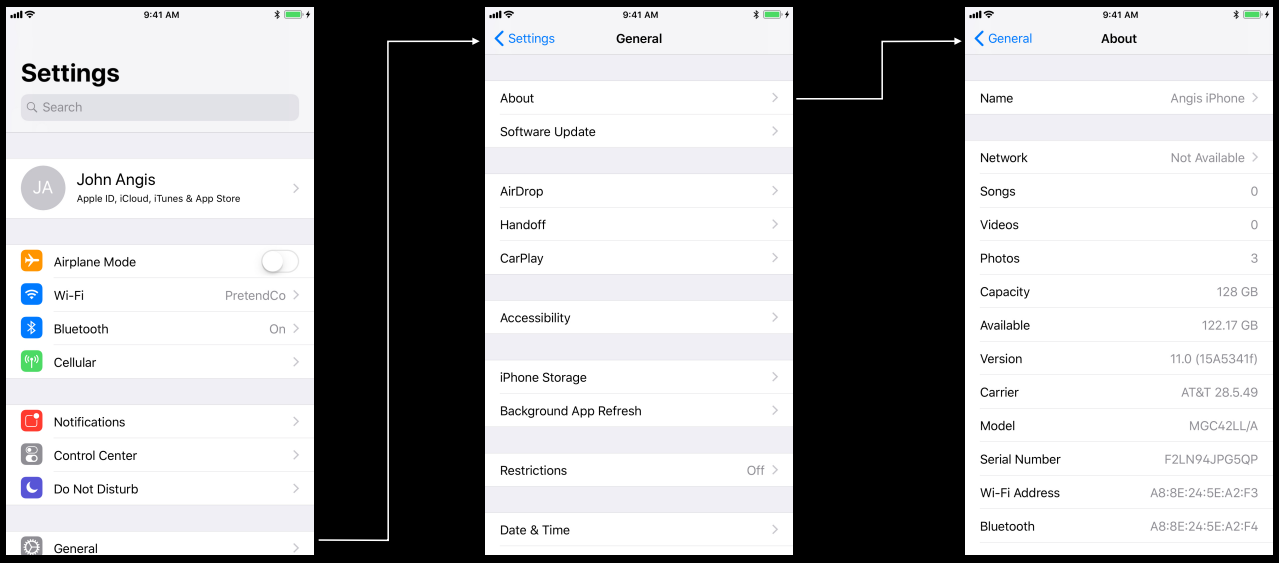
```
@IBAction func myUnwindFunction(unwindSegue: UIStoryboardSegue) {
}
```

Implement the returned method in the view controller you wish to return to

Doesn’t need to do anything apart from being implemented

Connect this to the view controller returning from

Navigation controller (UINavigationController)



Navigation controller

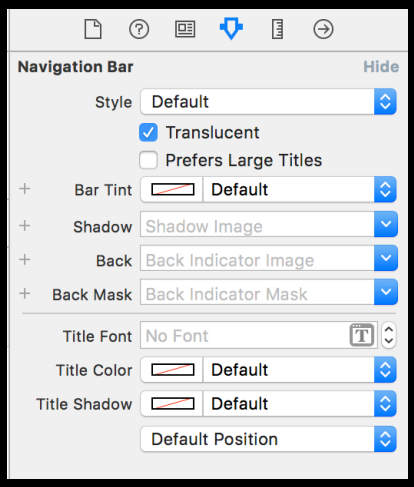
The top view controller's title

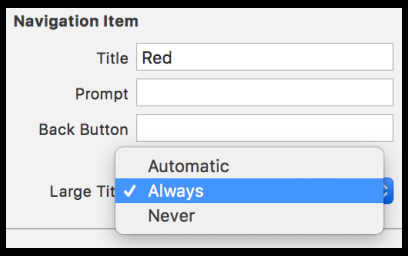
Back button

Navigation bar

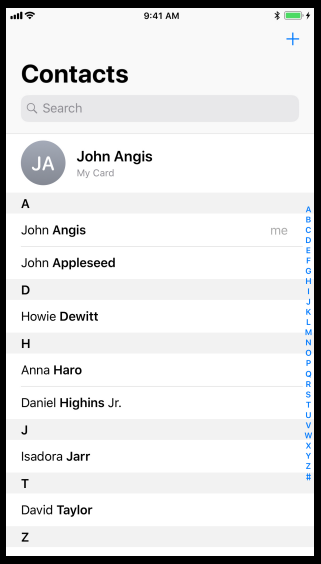
The top view controller's view

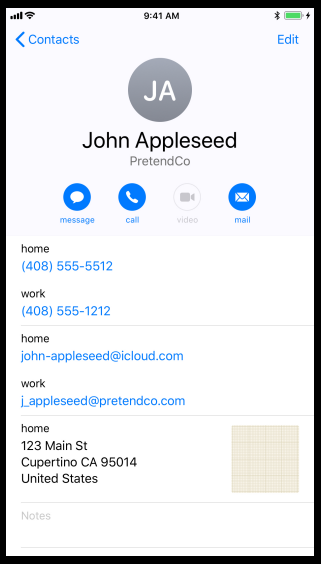
Navigation controller
Large titles





Pass information





Pass information

```
func prepare(for segue: UIStoryboardSegue, sender: Any?)
```

Segue properties

- `identifier`
- `destination`

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
   segue.destination.navigationItem.title = textField.text
}
```

Create programmatic segues

```
performSegue(withIdentifier:, sender:)
```

```
performSegue(withIdentifier: "ShowDetail", sender: nil)
```

Unit 3—Lesson 6
Segues and Navigation Controllers

Learn how to use segues to transition from one view controller to another

How to define relationships between view controllers

How navigation controllers can help you manage scenes that display related or hierarchical content

Unit 3—Lesson 6
Lab: Login

Create a login screen that will pass a user name between view controllers

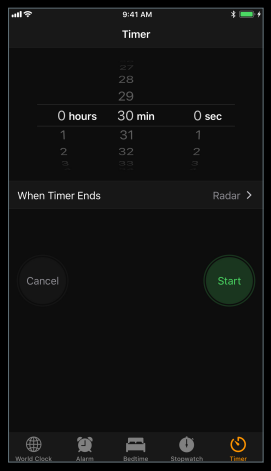
Use view controllers, a navigation controller, and segues to create both the login screen and a simple landing screen that will display in its title either the user name or text related to a forgotten user name or password

Unit 3—Lesson 8: Tab Bar Controllers

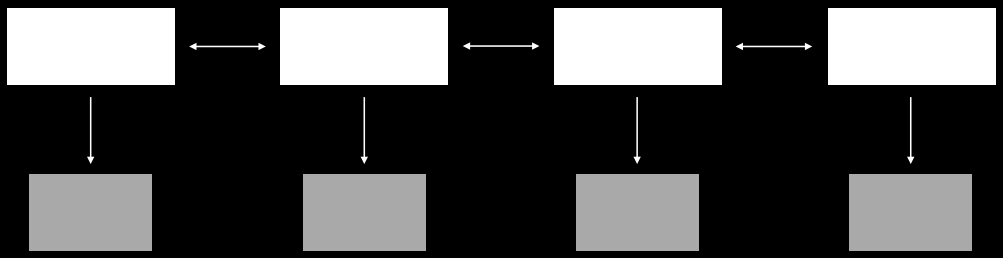
UITabBarController

A specialized view controller that manages a radio-style selection interface

A tab bar is displayed at the bottom of the view



Navigation hierarchy
Flat

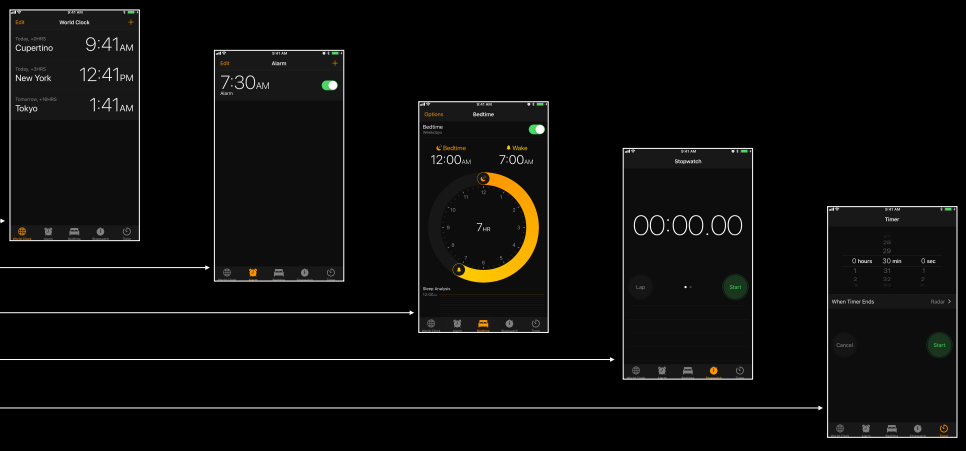


UITabBarController
Configuration

Tab Bar Controller

`viewControllers`

- World Clock Controller
- Alarm Controller
- Bedtime Controller
- Stopwatch Controller
- Timer Controller



Add a tab bar controller

Using a storyboard

Drag in a UITabBarController from the object library

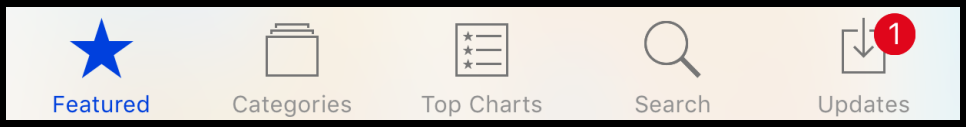
Add tabs

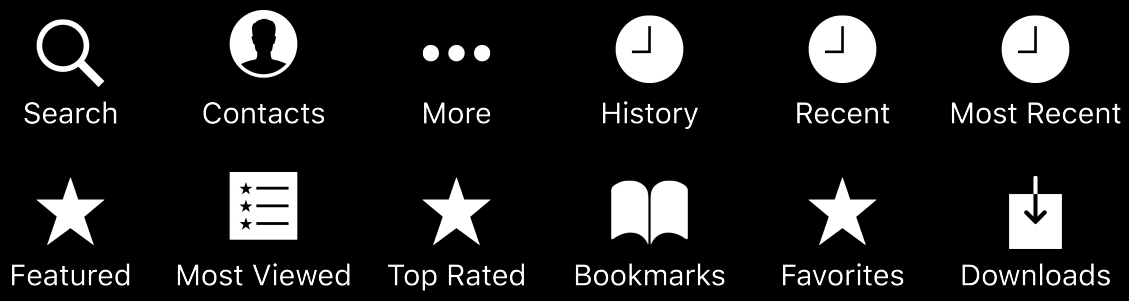
Drag a new view controller object onto the canvas

To create a segue, control-drag from the `UITabBarController` to the view controller

Select "view controllers" under Relationship Segue

UITabBarItem
Glyphish





Programmatic customization

```
tabBarItem.badgeValue = "!"
```

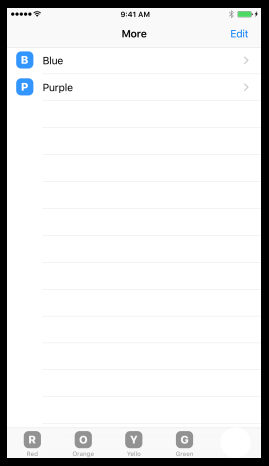

```
tabBarItem.badgeValue = nil
```

Even more tab items
More ! = Better

More view controller:

- Appears when needed
- Can’t be customized

If possible, plan app to avoid More



Unit 3—Lesson 7
Tab Bar Controllers

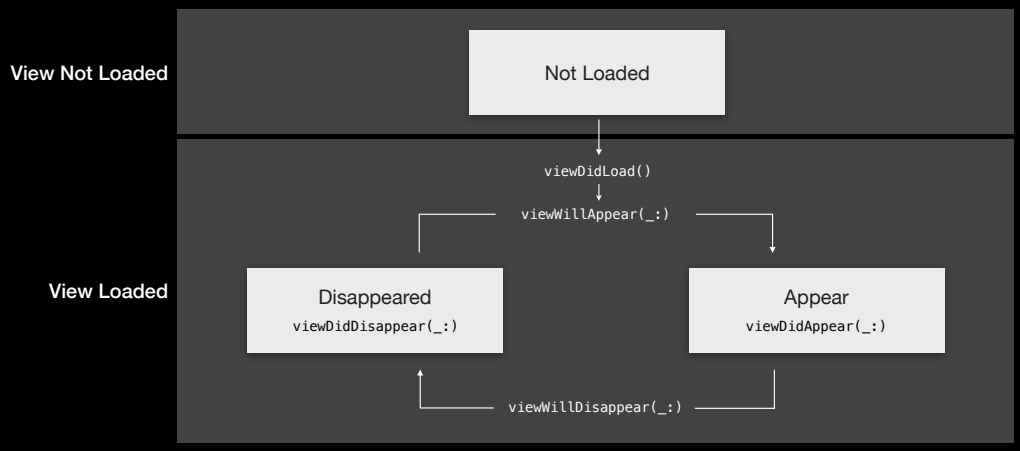
Learn how to use tab bar controllers to organize different kinds of information or different modes of operation.

Unit 3—Lesson 7
Lab: About Me

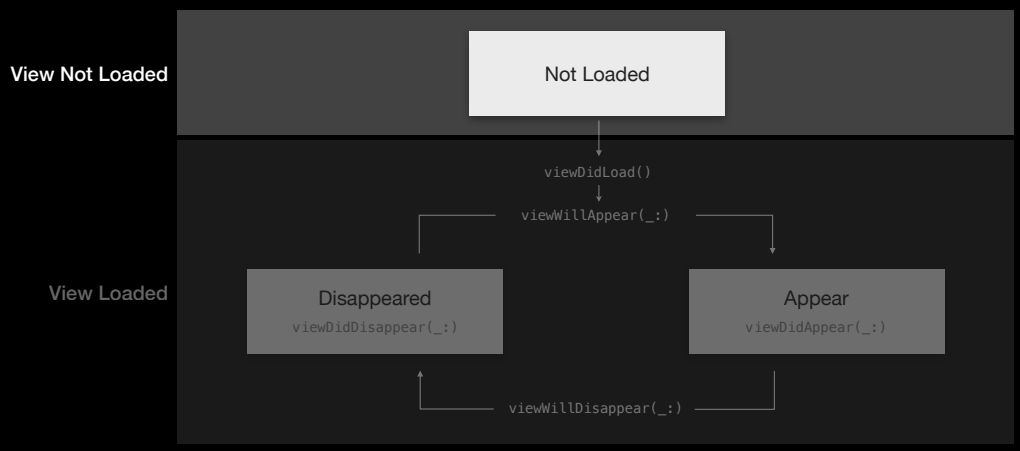
Create an app that displays distinct types of information about yourself in separate tabs.

Unit 3—Lesson 9:
View Controller Life Cycle

View controller life cycle



# View controller life cycle
viewDidLoad( )

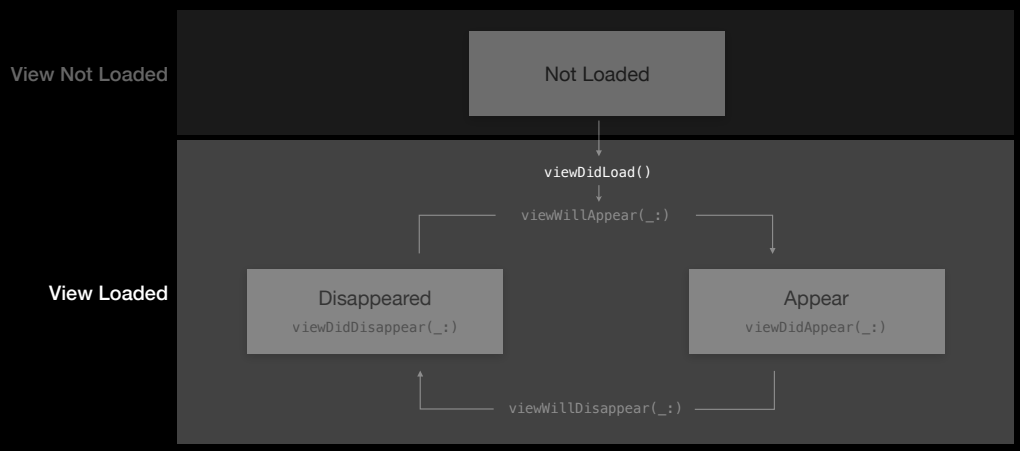


View event management

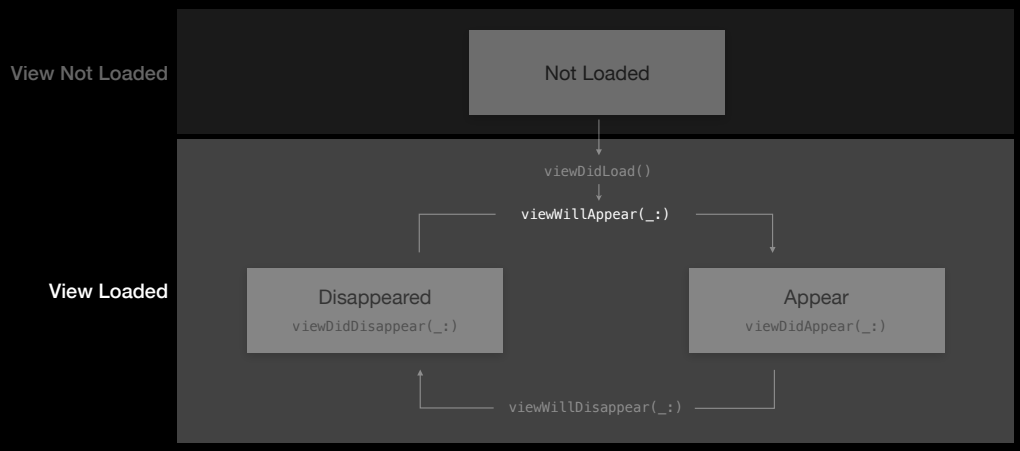
```
viewWillAppear(_:)
viewDidAppear(_:)
viewWillDisappear(_:)
viewDidDisappear(_:)
```

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)
    // Add your code here
}
```

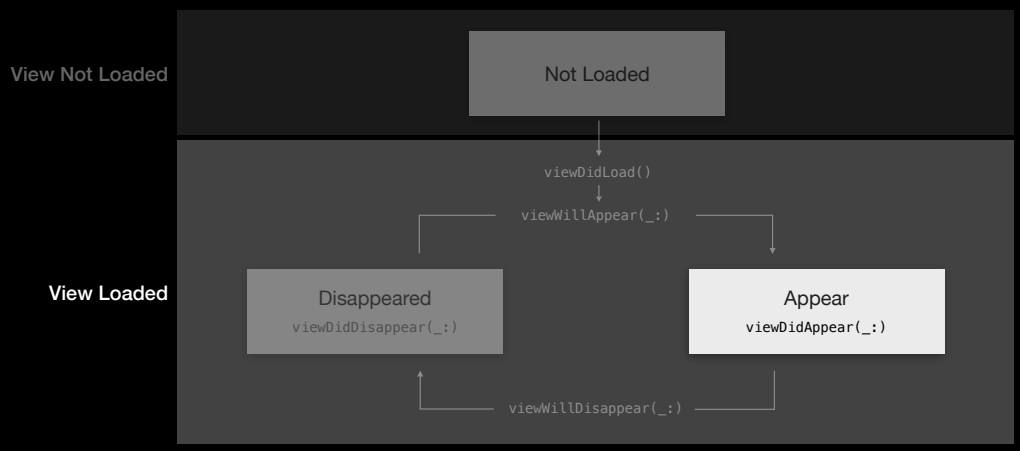
View event management
viewWillAppear(_:)



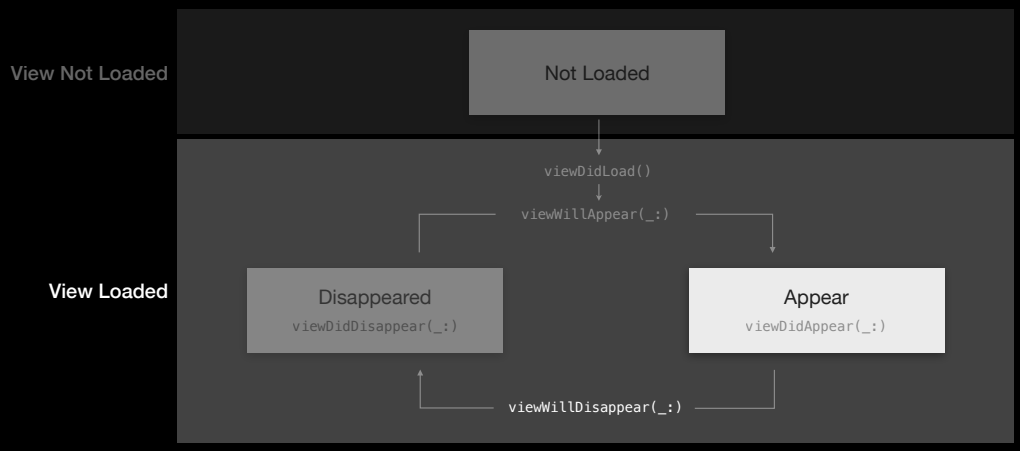
View event management
viewDidAppear(_:)



# View event management
viewWillDisappear(_:)



# View event management
viewDidDisappear(_:)



Unit 3—Lesson 8
View Controller Life Cycle

This lesson will explain more about the view controller life cycle so you can understand the infinite potential of this important class

Unit 3—Lesson 8
Lab: Order of Events

Further your understanding of the view's life cycle by creating an app that adds to a label’s text based on the events in the view controller life cycle