Unit 3

Navigation And Workflows

After the first two units, you should be ready to work toward more complex interfaces and interactions. You've learned the basics of Xcode, the development environment for building iOS apps, and you've had a chance to try out Interface Builder, a visual tool for creating user interfaces. After building a simple project, you went on to learn something about the Swift language, including fundamental programming concepts that enabled you to finish up the Apple Pie game. And you now have a functioning Keynote prototype of your own app idea.

In this unit, you'll learn about an important Swift feature for working with optional data. You'll also learn how to use multiple scenes, views, and controls to build simple workflows.

By the end of this unit, you should feel comfortable using Interface Builder and storyboards to build the user interface for apps with multiple views. You'll also test, validate, and plan how you can iterate on your app idea.

Swift Lessons

- Optionals
- Type Casting and Inspection
- Guard
- Constant and Variable Scope
- Enumerations

SDK Lessons

- Segues and Navigation Controllers
- Tab Bar Controllers
- View Controller Life Cycle
- Simple Workflows

What You'll Design

The App Design Workbook will guide you through testing, validating, and iterating on your app prototype.

What You'll Build

Quiz is a simple app that guides the user through a personality quiz and displays the results.

Lesson 3.1

Prepare to Test Your App

You have been working hard to get the look and feel of your app just right. But that's just one part of a much longer process. It often takes a lot of time to design an app that hits home for a user, and you may find that your first idea or design doesn't pan out as expected. Good design is about learning from users, reflecting on feedback, and iterating on your ideas. Testing your prototype will help you understand whether your ideas and assumptions are correct.

In this lesson, you'll architect your tests and create a plan to execute them.

What You'll Learn

- How to develop a plan for how and what users will test in your app prototype
- How to design a script to guide users through testing your app prototype

Related Resources

- WWDC 2018 Intentional Design
- WWDC 2018 Presenting Design Work
- WWDC 2019 What's New in iOS Design

Guide
Architect Your Testing

You've defined your app's goals; how will you determine whether you've achieved them? You've implemented a prototype; how do you expect it to be used? You'll define tests that answer those questions, and you'll also take a step back to think about setting expectations—yours and your users'.

Work through the Architect section of the App Design Workbook. By the end of this stage, you'll have a plan that you can use to write your test scripts.

Script Your Tests

Now that you've planned your testing, it's time to focus on the details.

Use the Script section of the App Design Workbook to complete a set of test scripts. You'll define the flow of your tests to keep the user engaged and oriented, dig into the kinds of questions each test can answer, and prepare for the unexpected.

Lesson 3.2

Optionals

One of the greatest strengths of Swift is its ability to read code and quickly understand data. When a function may or may not return data, Swift forces you to deal properly with both possible scenarios.

Swift uses unique syntax, called optionals, to handle this sort of case. In this lesson, you'll learn to use optionals to properly handle situations when data may or may not exist.

What You'll Learn

- How to create variables or constants that may not have a value
- How to check if a variable or constant contains a value
- How to create safe, clean code using optional binding
- How to create functions and initializers that return optionals
- When to use implicitly unwrapped optionals

Vocabulary

- **failable initializer**
- **force-unwrap**
- **implicitly unwrapped optional**
- **nested optional**
- **`nil`**
- **optional**
- **optional binding**
- **optional chaining**

Related Resources

- Swift Programming Language Guide: Optionals

Nil

Optionals are useful in situations when a value may or may not be present. An optional represents two possibilities: Either there *is* a value and you can use it, or there *isn't* a value at all.

Imagine you're building an app for a bookstore that lists books for sale. You have a model object `Book` type that has properties for `name` and `publicationYear`.

```
struct Book {
  var name: String
  var publicationYear: Int
}

let firstDickens = Book(name: "A Christmas Carol",
   publicationYear: 1843)
let secondDickens = Book(name: "David Copperfield",
   publicationYear: 1849)
let thirdDickens = Book(name: "A Tale of Two Cities",
   publicationYear: 1859)

let books = [firstDickens, secondDickens, thirdDickens]
```

So far, so good. Now imagine you're building a screen that shows a list of books that have been announced but haven't yet been published.

How might you initialize those books without a publish date? What do you assign to `publicationYear`?

```
let unannouncedBook = Book(name: "Rebels and Lions",
publicationYear: 0)
```

Zero isn't accurate, because that would mean the book is over 2,000 years old.

```
let unannouncedBook = Book(name: "Rebels and Lions",
publicationYear: 2019)
```

The current year or even next year isn't accurate either, because it *may* be released two years from now. There's no known launch date.

`nil` represents the absence of a value, or *nothing*. Because there is no `publicationYear` yet, `publicationYear` should be `nil`.

```
let unannouncedBook = Book(name: "Rebels and Lions",
publicationYear: nil)
```

That looks better, but the compiler throws an error. All instance properties must be set during initialization, and you can't pass `nil` to the `publicationYear` parameter because it expects an `Int` value.

Optionals solve this problem by providing a wrapper around a value that may exist. You can think of an optional as a box that, when opened, will contain either an instance of the expected type or nothing at all (`nil`).

Every type has a matching optional type, which you declare by adding a `?` after the original type name.

In this case, you need to update the type annotation on `publicationYear` property to `Int?`, an `Int` optional.

```
struct Book {
  var name: String
  var publicationYear: Int?
}

let firstDickens = Book(name: "A Christmas Carol",
   publicationYear: 1843)
let secondDickens = Book(name: "David Copperfield",
   publicationYear: 1849)
let thirdDickens = Book(name: "A Tale of Two Cities",
   publicationYear: 1859)

let books = [firstDickens, secondDickens, thirdDickens]

let unannouncedBook = Book(name: "Rebels and Lions",
publicationYear: nil)
```

Specifying The Type Of An Optional

Note that you can't create an optional *without specifying the type*. Consider what would happen if you tried to let Swift infer the type.

In this example, type inference will assume your variable is non-optional:

```
var serverResponseCode = 404 // Int, not Int?
```

In this example, type inference doesn't have any information to determine the data's type if the data *isn't* `nil`:

```
var serverResponseCode = nil // Error, no type specified when
   not `nil`
```

For these reasons, in most cases you'll need to use type annotation to specify the type when creating an optional variable or constant. Take a look at the following correct approaches to an `Int?` optional:

```
var serverResponseCode: Int? = 404 // Set to 404, but could be
   `nil` later

var serverResponseCode: Int? = nil // Set to `nil`, but could
   hold an `Int` later
```


Working With Optional Values

How do you determine whether or not an optional contains a value? How do you access the value? You could begin by comparing the optional to `nil` using an `if` statement. If the value is not `nil`, you can unwrap the value using the `force-unwrap` operator, `!`.

```
if publicationYear != nil {
  let actualYear = publicationYear!
  print(actualYear)
}
```

If you skipped the comparison of the optional to `nil` and you force-unwrapped an optional that doesn't contain a value, the code will generate an error and crash when you try to run it.

```
let unwrappedYear = publicationYear! // runtime error
print(unwrappedNumber)
```

It seems like good practice to compare an optional to `nil` before attempting to use the contained value, but it also feels redundant. As you'll recall, safety and clarity are primary design goals of Swift, so concise syntax is provided for safely using an optional's value if it has one, and avoiding errors if it doesn't.

Optional binding unwraps the optional and, if it contains a value, assigns the value to a constant as a non-optional type, making it safe to work with. This approach eliminates the need to continue working with the ambiguity of whether or not you are working with a value or with `nil`.

The syntax for optional binding looks like this:

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

If `someOptional` has a value, the value is assigned to `constantName` and is available only within the braces.

Take a look at how optional binding works on the previous `Book` example:

```
if let unwrappedPublicationYear = book.publicationYear {
  print("The book was published in \
  (unwrappedPublicationYear)")
}
```

Just like other `if` statements, you can add an `else` clause.

```
if let unwrappedPublicationYear = book.publicationYear {
  print("The book was published in \(unwrappedPublicationYear)")
}
else {
  print("The book does not have an official publication date.")
}
```


Functions And Optionals

Swift comes with many functions that return optional values.

Consider the example where you have a `String` whose value is set to `"123"`. You can see here that `string` could be converted into an `Int`.

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

But what if the string can't be converted?

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

Swift infers `possibleNumber` to be an `Int?` type because the initializer for `Int` that takes a `String` as a parameter may or may not be able to successfully convert the `String` into an `Int`. If `string` can be converted into an `Int`, `possibleNumber` will hold that value. If it can't, `possibleNumber` will be `nil`.

If you want to write a function that accepts an optional as an argument, simply update the type in the parameter list. Consider this print function that accepts a `firstName`, `middleName`, and `lastName`, but allows for `middleName` to be `nil` since not everyone uses a middle name.

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

The same is true for a function that returns an optional: Just update the return type. For example, a website URL returns the text from that page. The returned text is optional because the URL may not work or may not return any text.

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

Failable Initializers

Any initializer that might return `nil` is called a failable initializer. Earlier in this lesson, you saw how the `Int` initializer attempted to create an `Int` from a `String` and returned `nil` if it was unable to convert the `String`.

For greater control and safety, you may want to create your own failable initializers and define the logic for returning an instance, or `nil`.

Consider the following definition for a `Toddler`:

```
struct Toddler {
  var name: String
  var monthsOld: Int
}
```

In this example, every `Toddler` must be given a name, as well as an age in months. However, you might not want to create an instance of a toddler if the child is younger than 12 months or older than 36 months. To provide this flexibility, you can use `init?` to define a failable initializer. The question mark (`?`) tells Swift that this initializer may return `nil` and that it should return an instance of type `Toddler?`.

Within the body of the initializer, you can check whether the given age is less than 12 or greater than 36. If either one is true, the initializer returns `nil` instead of assigning a value to `monthsOld`:

```
struct Toddler {
  var name: String
  var monthsOld: Int

  init?(name: String, monthsOld: Int) {
    if monthsOld < 12 || monthsOld > 36 {
      return nil
    } else {
      self.name = name
      self.monthsOld = monthsOld
    }
  }
}
```

When you use the failable initializer to create `Toddler` instances, an optional will always be returned. To safely unwrap the value before proceeding to use it, you can use optional binding:

```
let toddler = Toddler(name: "Joanna", monthsOld: 14)

if let myToddler = toddler {
  print("\(myToddler.name) is \(myToddler.monthsOld) months old")
} else {
  print("The age you specified for the toddler is not between 1
  and 3 yrs of age")
}
```

Optional Chaining

It's also possible for an optional value to have optional properties, which you might think of as a box within a box. These are called nested optionals.

In the following example, note that every `Person` has an `age` and may have a `residence`. A `Residence` may have an `address`, and not every `Address` has an `apartmentNumber`.

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

struct Residence {
  var address: Address?
}

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


Unwrapping nested optionals can require a lot of code. In the following example, you're checking an individual's address to find out if he/she lives in an apartment. To do this for a given `Person` object, you must unwrap the `residence` optional, the `address` optional, and the `apartmentNumber` optional. Using if-let syntax, you'd have to do quite a bit of conditional unwrapping:

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

But there's a better way to do this. Rather than assign a name to every optional, you can conditionally unwrap each property using a construct known as *optional chaining*. If the person has a residence, the address has an apartment number, and if that apartment number can be converted to an integer, then you can refer to the number using `theApartmentNumber`, as seen here:

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

When chaining together optionals, a `?` appears before each optional in the chain.

If a `nil` value breaks the chain at any point, the `if let` statement fails. As a result, no value is assigned to the constant, and the code inside of the braces never executes. If none of the values are `nil`, the code inside of the braces executes and the constant has a value.

Implicitly Unwrapped Optionals

An object cannot be initialized until all of its non-optional properties are given a value. But in some cases, particularly with iOS development, some properties are `nil` for just a moment until the value can be specified after initialization. For example, you've used Interface Builder to create outlets so that you can access a particular piece of the interface in code.

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

If you were the developer of this class, you'd know that anytime a `ViewController` is created and presented to the user, there will always be a `label` on the screen, because you added it in the storyboard. But in iOS development, the storyboard elements aren't connected to their corresponding outlets until *after* initialization takes place. Therefore, `label` must be allowed to be `nil` for a brief period.

What about using a regular optional, `UILabel?`, for the type? The standard optional will require the if-let syntax to constantly unwrap the value, providing a safety mechanism for data that may not exist. But you *know* that `label` will have a value after the storyboard connects the outlets, so unwrapping an optional that isn't really "optional" feels cumbersome.

To get around this issue, Swift provides syntax for an implicitly unwrapped optional, using the exclamation mark `!` instead of the question mark `?`. As the name suggests, this type of optional unwraps automatically, whenever it's used in code. This allows you to use `label` as though it weren't an optional, while allowing the `ViewController` to be initialized without it.

Implicitly unwrapped optionals should be used in one special case: when you need to initialize an object without supplying the value, but you know you'll be giving the object a value before any other code tries to access it. It might seem convenient to overuse implicitly unwrapped optionals to save yourself from using `if let` syntax, but by doing so you'd remove an important safety feature from the language. Thus, many Swift developers consider too many `!`'s in code a red flag. If you try to access the value of an implicitly unwrapped optional and the value is `nil`, your program crashes.

Navigation and Workflows

Lab

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

Review Questions

Question 1 of 4

Which of the following declares a double with a value of 4.2 that can be set to `nil` at a later date?

- **A.** `let height: Double? = 4.2`
- **B.** `var height: Double = 4.2`
- **C.** `var height: Double? = 4.2`
- **D.** `var height: Double? = nil`

Check Answer

Lesson 3.3

Type Casting and Inspection

Whenever you work with data, the type plays a crucial role. For example, if a function returns an `Int`, you know you can use its value in a mathematical expression. But what if the type information isn't very specific and you need to inspect the data more closely to determine how to use it?

In this lesson, you'll learn why some data can only be expressed using a broader type and how you can test for specific kinds of data before using it.

What You'll Learn

- How to mix values of different types into the same collection
- How to check the specific type of value within a heterogeneous collection
- How to downcast an object to a particular type before accessing its properties and methods

Vocabulary

- **`as!`**
- **`as?`**
- **`Any`**
- **`AnyObject`**
- **conditional cast**
- **downcast**
- **type casting**
- **type inspection**

Related Resources

Swift Programming Language Guide: Type Casting

Type Casting

Suppose Brad's job is to visit the homes of his clients and to take care of their pets. When he arrives at each location, he performs different tasks depending on the type of animal. If the pet's a dog, he takes it for a walk. If it's a cat, he changes the litter box. And if it's a bird, he cleans the cage.

In Swift, a function's declaration determines the type of data to be returned. Since the function can't return a `Dog`, a `Cat`, and a `Bird`, the best it can do is return the parent type of all three, `Animal`.

```
func getClientPet() -> Animal {
  // returns the pet
}

let pet = getClientPet() // pet is of type Animal
```

Unfortunately, this type is too broad to be useful to Brad. Without knowing the specific animal type, he could accidentally try to walk the bird, clean the dog's litter box, or clean the cat's cage. Consider the following valid functions with `Dog`, `Cat`, and `Bird` parameters.

```
func walk(dog: Dog) {
  print("Walking \(dog.name)")
}

func cleanLitterBox(cat: Cat) {
  print("Cleaning the \(cat.boxSize) litter box'")
}
func cleanCage(bird: Bird) {
  print("Removing the \(bird.featherColor) feathers at the
    bottom of the cage'")
```

Brad needs to be able to access a version of `pet` that's one of the subclasses of `Animal`. You can use the `as?` operator to try and downcast the value to a more specific type and store it in a new constant. This operation is known as a *conditional cast*, because it casts the instance to the specified type if it's possible to do so. Use `if-let` syntax to check the conditions before converting the type:

```
let pets = allClientAnimals()

for pet in pets {
if let dog = pet as? Dog {
    walk(dog: dog)
  } else if let cat = pet as? Cat {
    cleanLitterBox(cat: cat)
  } else if let bird = pet as? Bird {
    cleanCage(bird: bird)
  }
}
```

Now Brad can be sure he's walking dogs, cleaning kitty litter boxes, and cleaning birdcages.

There's also a forced form of the type cast operator: `as!`. This version will force the downcast to the specified type. But if you specify an incorrect type, it will crash your program, just as it does with force-unwrapping an optional.

Imagine Alan has one pet, a dog, and he goes to pick it up from the pet store. A `fetchPet(for customer: String)` function may return an `Animal` type.

```
let alansDog = fetchPet(for: "Alan")
// alansDog is inferred as the `Animal` type
```


When you *know* that the returned object will be a more specific type, you can use the `as!` operator to cast the value immediately.

```
let alansDog = fetchPet(for: "Alan") as! Dog
// alansDog is inferred as the `Dog` type
```

When you begin to build apps, you'll discover that when you work with `UIKit`, the APIs can return very generic objects such as `UIViewController`. But as the developer of your application, you know what the specific type should be. For example, if pressing a button on the view of `FirstViewController` always presents a `SecondViewController`, you can force the downcast of `destination` to `SecondViewController` within the `prepare(for:sender:)` function. This function is called whenever you present a new view controller using storyboard segues.

```
class SecondViewController: UIViewController {
  var names: [String]?
}

func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  let secondVC = segue.destination as! SecondViewController
  secondVC.names = ["Peter", "Jamie", "Tricia"]
}
```

Use `as!` only when you are certain that the specific type is correct.

Any

You've learned that arrays, by default, are set to handle a specific type, such as an `Int`, `String`, or `Animal`. Homogenous collections are simpler to work with because you know the type of every instance within them.

But what if you want to work with nonspecific types? Swift provides two special types: `Any` and `AnyObject`. As the name implies, `Any` can represent an instance of any type: strings, doubles, functions, or whatever. `AnyObject` can represent an instance of any class in Swift but not a structure.

Here's an example of an array that can hold any type of instance, or `[Any]`:

```
var items: [Any] = [5, "Bill", 6.7, Dog()]
```

Because the array above can include anything, there's no way to guarantee the type of any item. For example, if you used `items[0]` to access the first item in the following array, it will return the value `5` with the nonspecific `Any` type. To go ahead and use the value of `firstItem` as an `Int`, you'd use the `as?` operator:

```
var items: [Any] = [5, "Bill", 6.7, true]
if let firstItem = items[0] as? Int {
  print(firstItem + 4) // 9
}
```

Although `Any` can be used just like any other type, it's always better to be specific about the types you expect to work with. Before using `Any` or `AnyObject` in your code, make sure you need the behavior and capabilities these special types provide.

Navigation and Workflows

Lab

Open and complete the exercises in `Lab—Type Casting.playground`.

Review Questions

Question 1 of 3

If a variable can be set to any given structure, what's the variable's type?

- **A.** `is`
- **B.** `as?`
- **C.** `Any`
- **D.** `AnyObject`

Check Answer

Lesson 3.4

Guard

Most bugs reside in complex code. The simpler your code is to read, the easier it is to spot potential bugs.

In this lesson, you'll learn to use `guard` to better manage control flow. You'll learn to handle invalid and special-case values up front, rather than weaving them throughout your programming logic.

What You'll Learn

- How to use `guard` to write more readable code
- How to build a function that guards against invalid arguments

Vocabulary

- **`guard`**
- **`guard-let`**

Related Resources

- Swift Programming Language Guide: Early Exit

As you start to work on more complex apps, you'll need to write functions that depend on a series of `true` conditions before executing. But the more conditions in your code, the harder it is to read—and debug—especially when the `if` statement is your only form of control flow.

You've learned that each `if` statement evaluates a Boolean expression. If the result is `true`, the code defined in the statement is executed; otherwise, it's skipped. But what if you have multiple `if` statements nested within one another? Your code begins to form what programmers call the "pyramid of doom":

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

What's so problematic about this example? With each `if` statement, the code is moving farther and farther from the beginning of each line, making the code hard to read. And each `else` statement moves farther and farther from its corresponding `if` statement, so it's difficult to tell how they match up. You can rework this logic to reduce the pyramid effect while still using `if` statements:

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

    if invitedGuests.isEmpty {
        print("It's just a family party.")
        return
    }

    if cakeCandlesLit == false {
        print("The cake's candles haven't been lit.")
        return
    }

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

But you can go a step further using the `guard` statement to clearly communicate to the person reading this code that specific conditions must be met before proceeding.


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

  guard !invitedGuests.isEmpty 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!")
}
```

A `guard`'s `else` block is executed only if the expression evaluates to `false`. This is the opposite of the `if` statement that executes the block if the expression evaluates to `true`.

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

// true: execute some code
```

With this design, you can write a function that returns early if it can't complete its task. A `guard` statement requires you to exit the scope of the function using the `return` keyword in the `else` case. By eliminating all the unwanted conditions using `guard` statements and calling `return`, you can move conditional checks to the top of the function and put the code you expect to run at the bottom. The expected code is no longer surrounded by unexpected conditions. This also clearly communicates to the reader the author's intent and reasoning for the conditional check—as opposed to using `if` statements, which don't require a `return`.

Here are two versions of a `divide` function, one using an `if` statement and one using a `guard` statement. Since you can't divide by zero, each version of the function prints the result if the divisor is not zero.

```
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)
}
```

The code using `guard` does an early `return` if the divisor passed in is 0. By the time it reaches the line that does the actual division, it has already removed any erroneous parameters.

The `if` example above could also be written similarly to the one using `guard` syntax by reversing the condition in the check. However, it's considered better practice and more readable to use `guard` in these cases.

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


Guard with Optionals

In an earlier lesson, you learned about optionals and that a function should perform work if an optional contains a value. When you unwrap an optional using `if-let` syntax to bind it to a constant, the constant is valid within the braces.

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

Instead, you can use `guard let` to bind the value within an optional to a constant that's accessible outside the braces.

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

Note the placement of the `else` statement after the condition. This placement signifies that `guard` is checking for a `true` or nonoptional condition. When the value you're attempting to unwrap is `nil`, the code within the `else` block is executed—otherwise execution continues to the line after the closing brace and the unwrapped value is available to use.

Both `if let` and `guard let` let you unwrap multiple optionals in one statement. However, doing so with a `guard let` makes all unwrapped values available throughout the rest of the function, rather than only within the control flow braces.

```
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.")
}
```

Using `guard` statements to move conditional code is one way to improve the readability of your programs. Throughout this course, you've looked over plenty of code that others have written. You've probably discovered that it's much easier to understand what's going on if you can quickly locate the core functionality—rather than search for it in a sea of validation code.

Navigation and Workflows

Lab

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

Review Questions

Question 1 of 2

What is the purpose of the `guard` statement? Check all that apply.

- **A.** To simplify control flow and communicate intent
- **B.** To eliminate invalid parameters early on
- **C.** To perform work that cannot be done with an `if` statement
- **D.** All of the above

Check Answer

Lesson 3.5

Constant and Variable Scope

As you write larger, more complex programs, you'll need to pay attention to where you declare your constants and variables. What's the optimal placement in your code? If you declare every variable at the top, you may find your code is more difficult to read and much more difficult to debug.

In this lesson, you'll learn to write well-structured code that's easy to read. You'll do this by properly scoping your constants and variables.

What You'll Learn

- How to differentiate between global and local scope
- How to create variables and functions in global and local scope
- How to re-use variable names using variable shadowing

Vocabulary

- **global scope**
- **local scope**
- **scope**
- **variable shadowing**

Each constant and variable lives within some sort of scope, a place where it's visible and accessible. There are two different levels of scope: global and local. Any variable declared in global scope is called a global variable, and a variable declared in local scope is a local variable.

Global scope refers to code that's available from anywhere in your program. For example, when you begin declaring variables inside an Xcode playground, you're declaring them in global scope. After you define a variable on one line, it's available to each line after it. Whenever you finish typing into a playground, the code is executed line by line, beginning from this global scope.

Whenever you add a pair of curly braces (`{ }`)—whether for a structure, class, function, `if` statement, `for` loop, or more—the area within the braces defines a new local scope. Any constant or variable that's declared within the braces is defined in that local scope and isn't accessible by any other scope.

Consider the following block of code:

```
var age = 55

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

print(age)
printMyAge()
```

Console Output:

```
55
My age: 55
```

Notice that the variable `age` is defined at the top of the code and not inside a control flow structure or function. This means it's in global scope and can be accessed throughout the program. The function `printMyAge` is able to reference `age`, even though it wasn't passed in as a parameter. Similarly, the function `printMyAge` isn't defined *within* a structure or class, so it's in global scope and is therefore accessible by the last line in the code.

Now look at another block of code:

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

printBottleCount()
print(bottleCount)
```

The variable `bottleCount` is defined within a function, `printBottleCount`, which has its own local scope between the braces. So `bottleCount` is in local scope and is only accessible by the contents of the function, inside the braces. The last line in the code will throw an error, because it's unable to find a variable named `bottleCount`.

Consider one more example:

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

printTenNames()
```

In the code above, `name` is a local variable and is available to anything defined within the same scope. It's also available within an even smaller local scope: the `for` loop on the next line. The variable `index`, while defined inside the function, was defined inside the loop, which can be thought of as a more narrowly defined subsection of the function's scope. `index` is therefore only accessible inside the loop. Because `print(index)` occurs just outside the loop, it produces an error.

Navigation and Workflows

Variable Shadowing

This next example defines a variable called `points` in two different locations: within the function's local scope and within the `for` loop's local scope. This is called variable shadowing. It's valid Swift code, but it might not be obvious what will happen when the code is executed.

```
func printComplexScope() {
    let points = 100
    print(points)

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

    print(points)
}

printComplexScope()
```

Console Output:

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

First, `points` is declared and set to 100. This value is printed on the next line. Within the `for` loop, another `points` is declared, this one with a value of 200. The second `points` completely shadows the function-scoped variable, which means that it renders the first `points` inaccessible. Any reference to `points` will access the one closest to the same scope. So when the `print` statement within the loop is called, it will print a value of 200 five times. After the loop is finished, the print statement will print the only `points` variable that it can access: the one with a value of 100.

To avoid unnecessary confusion in this particular example, you might advocate changing the name of the inner `points` variable. And you would probably be correct. However, there are a few cases when variable shadowing can be useful. Imagine having an optional `name` string and you want to use `if let` syntax to do some work with its value. Rather than coming up with a new variable name, like `unwrappedName`, you can reuse `name` within the scope of the `if let` braces:

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

if let name = name {
  // name is a local `String` that shadows the global `String?`
  of the same name
  print("My name is \(name)")
}
```

You can also use variable shadowing to simplify naming unwrapped optionals from a `guard` statement.

```
func exclaim(name: String?) {
  if let name = name {
    // Inside the braces, `name` is the unwrapped `String`
    Value
    print("Exclaim function was passed: \(name)")
  }
}

func exclaim(name: String?) {
  guard let name = name else { return }
  // name: `String?` is no longer accessible, only name: `String`
  print("Exclaim function was passed: \(name)")
}
```

Shadowing the optional with the unwrapped value is common in Swift code. Be sure that you can read and understand this common pattern.

Shadowing And Initializers

You can take advantage of your knowledge of variable shadowing to create clean, easy-to-read initializers. Suppose you want to create an instance of a `Person` by passing in a name and age as its two parameters. You'll also assume that every `Person` instance has both `name` and `age` properties:

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

let tim = Person(name: "Tim", age: 35)
print(tim.name)
print(tim.age)
```

Console Output:

```
Tim
35
```

As you're writing the initializer, you'll want to keep it as simple and logical as possible: assigning the `name` parameter to the `name` property and assigning the `age` parameter to the `age` property.

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

Since `name` and `age` are the names of parameters within the function scope, they shadow the `name` and `age` properties defined within the `Person` scope. You can place the keyword `self` in front of the property name to reference the property specifically—and to avoid any confusion that variable shadowing may cause to the compiler and the reader. This syntax makes it very clear that the `name` and `age` properties are set to the `name` and `age` parameters passed into the initializer.

Lab

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

Review Questions

Question 1 of 3

What is the result of the following block of code?

```
let sum = 99
func computeSum(scores: [Int]) -> Int {
  var sum = 0
  for score in scores {
    sum += score
  }
  return sum
}

computeSum(scores: [70, 30, 9])
```

- **A.** Compiler error; `sum` cannot be defined twice in the same scope.
- **B.** 99
- **C.** 109
- **D.** 0

Check Answer

Lesson 3.6

Enumerations

As a programmer, you'll work with many situations that require you to assign values from a limited number of options. Imagine you're writing a program that allows passengers to select a seat from three options: window, middle, and aisle. In Swift, you'd do this with an enumeration.

An enumeration, or `enum`, is a special Swift type that allows you to represent a named set of options. In this lesson, you'll learn when enumerations are commonly used, how to define an enumeration, and how to work with enumerations using `switch` statements.

What You'll Learn

- Why enumerations are a useful tool
- How to define simple enumerations
- How to define enumerations with raw values
- How to work with enumerations using the `switch` statement

Vocabulary

- **`case`**
- **`default`**
- **`enum`**
- **enumeration**

Related Resources

- Swift Programming Language Guide: Enumerations

Enumerations define a common type for a group of related values.

Consider the directions on a compass app: north, east, south, and west. The app can help orient the user toward any of those four directions. The needle on a compass always points to the north, but the heading of the compass moves as the user moves. The heading helps the user determine which direction they're facing.

You define a new enumeration using the keyword `enum`. The code below defines an `enum` for tracking the direction on a compass:

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

The `enum` defines the type, and the `case` options define the available values allowed with the type. It's best practice to capitalize the name of the enumeration and to lowercase the `case` options.

You can also define the available cases, separated by commas, on a single line:

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


Once you've defined the enumeration, you can start using it like any other type in Swift. Just specify the enumeration type along with the value:

```
var compassHeading = CompassPoint.west
// The compiler assigns `compassHeading` as a `CompassPoint` by
   type inference.

var compassHeading: CompassPoint = .west
// The compiler assigns `compassHeading` as a `CompassPoint`
   because of the type annotation. The value can then be assigned
   with dot notation.
```

Now that the type of `compassHeading` is set, you can change the value to another compass point using the shorter dot notation:

```
compassHeading = .north
```

Control Flow

In the control flow lesson, you learned how to use `if` statements and `switch` statements to respond to `Bool` values. You can use the same control flow logic when working with different cases of an enumeration.

Consider the code below that prints a different sentence based on which `CompassPoint` is set to the `compassHeading` constant:

```
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")
}
let compassHeading: CompassPoint = .west

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


Type Safety Benefits

Enumerations are especially important in Swift because they allow you to represent information, such as strings or numbers, in a type-safe way.

Imagine a set of data that represents movies of specific genres. Before learning about enumerations, you may have defined a simple movie structure like this:

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

Given that definition, you would use a `String` when setting the genre:

```
let movie = Movie(name: "Wolfwalkers", releaseYear: 2020,
genre: "Aminated")
```

Do you notice a problem in this initializer?

Many Swift developers would say that `genre` is "stringly typed" instead of "strongly typed." What they're referencing is the fact that `genre` is prone to all the errors that `String` values face—and one them is incorrect spelling. Imagine you wrote code to fetch all the movies in the "Animated" genre. *Wolfwalkers* would be missing.

As a better practice, you could assign `genre` a value from an enumeration called `Genre`.

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

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

let movie = Movie(name: "Wolfwalkers", releaseYear: 2020,
genre: .animated)
```

This code is much less error-prone. The compiler enforces safety by requiring you to choose a case from the `Genre` enumeration when you initialize a new movie.

Enumerations are an extremely powerful tool in Swift. You'll use them any time you want to add type safety where you might otherwise use strings or numbers. You'll continue to learn more advanced features of enumerations as you work with more complex data.

Navigation and Workflows

Lab

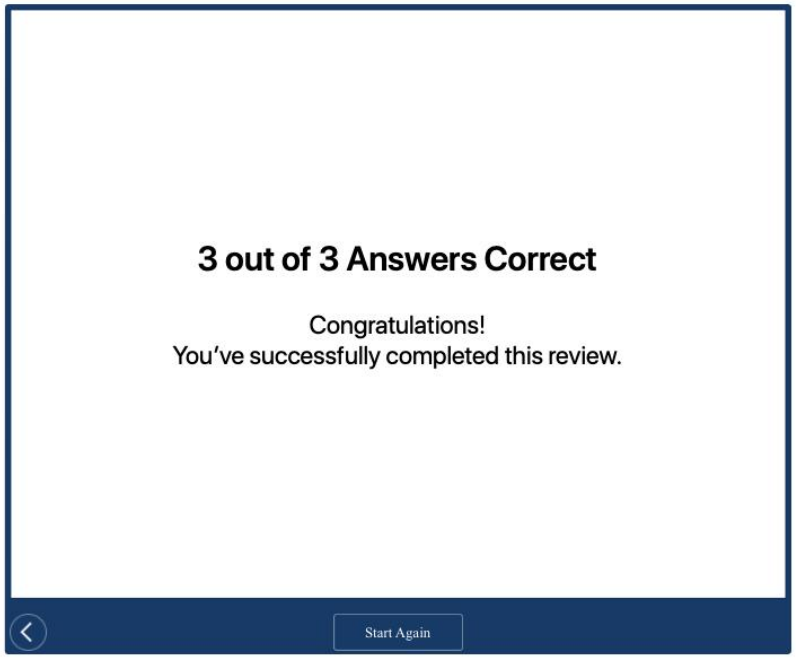
Open and complete the exercises in `Lab—Enumerations.playground`.

Connect To Design

Open your App Design Workbook and review the Prototype section for your app (or review the prototype itself). Can you find places where your app could use an enumeration? Add comments to the Prototype section or in a new blank slide at the end of the document.

In the workbook's Go Green app example, the developer might provide a list of possible types of recycling that a user could choose from. Everywhere the types of recycling get used, an enumeration would make sure each choice is accounted for.

Review Questions



Lesson 3.7

Segues and Navigation Controllers

You've already learned that view controllers manage different scenes within an app. But as your apps grow in complexity, you'll find you need different scenes using different view controllers to display information. You'll also need to transition between different scenes to allow the user to navigate the app.

In this lesson, you'll learn how to use segues to transition from one view controller to another, how to define relationships between view controllers, and how navigation controllers can help you manage scenes that display related or hierarchical content.

What You'll Learn

- How to move from one view controller to another
- How to add and customize a navigation controller
- How to pass information from one view controller to another

Vocabulary

- **bar button**
- **modal presentation**
- **modal segue**
- **navigation bar**
- **navigation controller**
- **pop**
- **push**
- **root view controller**
- **segue**
- **Show segue**
- **unwind segue**

Related Resources

- Showing and Hiding View Controllers
- API Reference: UINavigationController
- API Reference: UINavigationBar

Take a look at the most commonly used apps on your phone. Do any of them have one screen that always looks the same? Probably not. Most apps have many scenes for displaying different types of information. Each of these scenes is backed by a separate view controller instance or class.

Your job as a developer is to allow users to move easily from one scene to another. You can use Interface Builder to add segues, or transitions, between different scenes. You can also create special relationships between scenes with related content by including them in a navigation controller.

After you learn about the different types of segues, you'll learn how to create segues between different scenes in a navigation controller. Once you've mastered working with segues and navigation controllers, you'll be able to build more complex interfaces and navigation hierarchies—which, in turn, will equip you to build more powerful apps.

Segues

A segue defines a transition from one view controller to another. It often begins when the user taps a button or table row, and it ends when a new view controller is presented. Similar to creating outlets and actions, you define segues in Interface Builder by connecting the start and end points, clicking and dragging from one scene to another. You can also trigger segues programmatically.

In addition to the transition, a segue also defines the presentation method of the view controller. One common method is a modal presentation, which places a new view controller on top of the previous one. On smaller screens, a modal presentation will always appear at full screen. To adapt the UI for larger devices, you can customize a modal presentation to appear as a popover, a form sheet, or a full-screen presentation.

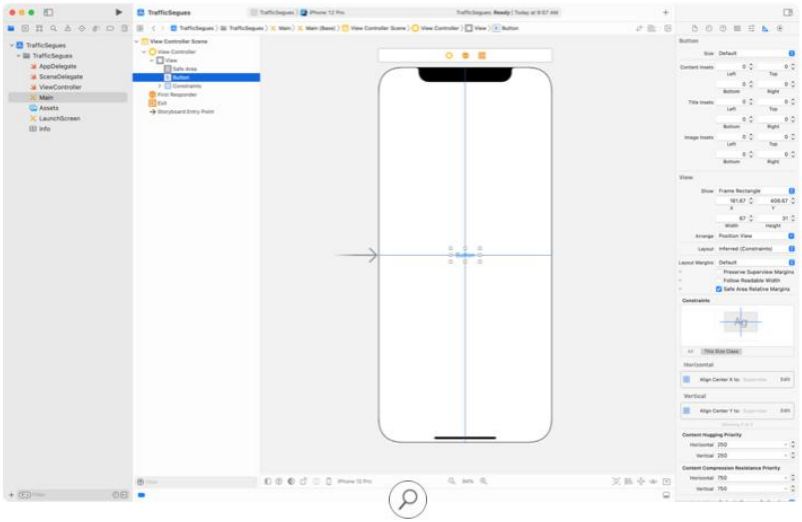
When learning about navigation controllers later in this lesson, you'll also learn about the push transition, which animates a new view controller from right to left onto the screen.

When a new view controller is presented modally, you can use an unwind segue to allow the user to dismiss the new view controller and return to the previous one.

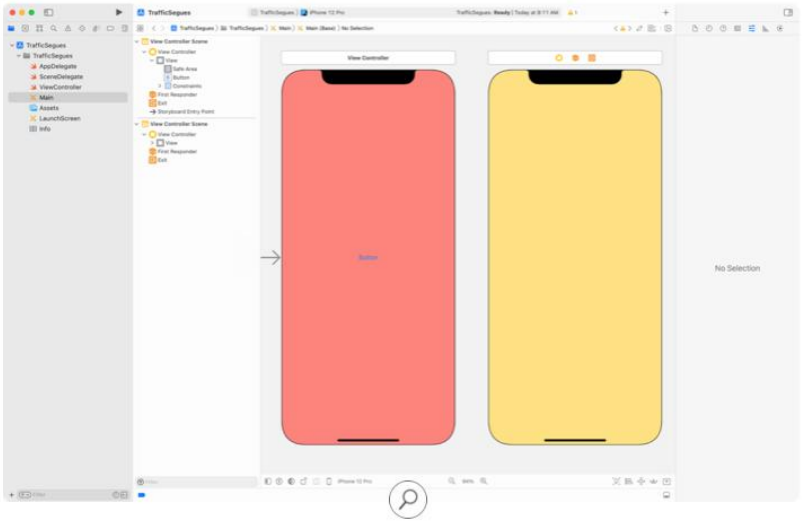
Create Triggered Segues

To practice transitioning between view controllers, you'll create a simple app that cycles through the different colors of a traffic light. Start by creating a new Xcode project using the iOS App template. Name the project "TrafficSegues." When creating the project, make sure the interface option is set to Storyboard. Select the `Main` storyboard in the Project navigator to open your project in Interface Builder.

Navigation and Workflows

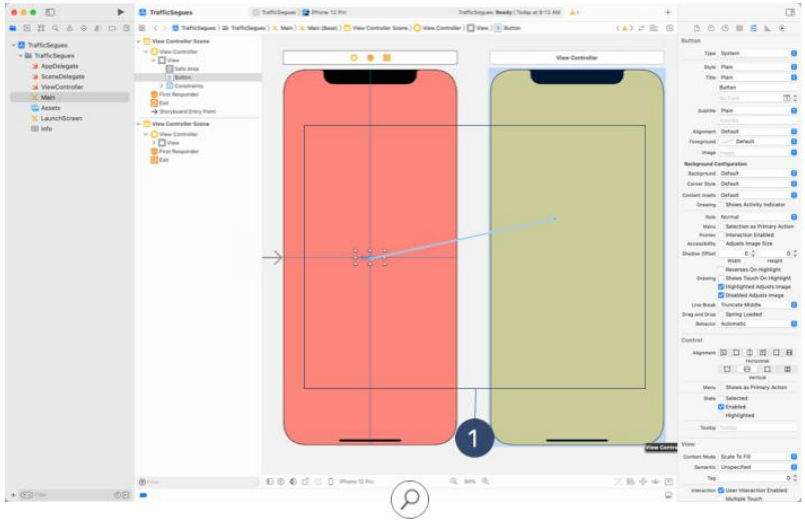


Add a `UIButton` to the center of the view, using the alignment guides to help position it. Click the Align button and select "Horizontally in Container" and "Vertically in Container" to create two constraints that center the button for all screen sizes.

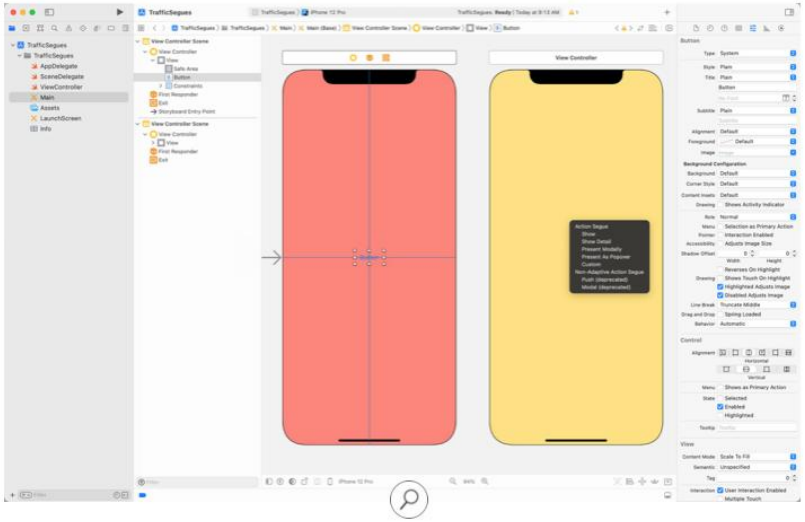


Locate View Controller in the Object library, and drag this object onto the canvas, positioning it to the right of the first view controller. Using the Attributes inspector, give the left view controller a red background and the right view controller a yellow background. (The screenshots use lighter shades of color to provide additional clarity.)

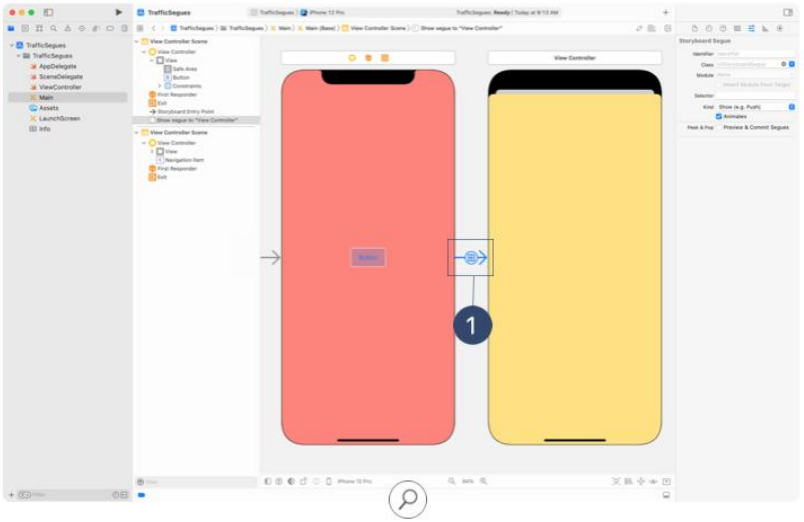
Navigation and Workflows



Imagine you want to transition to the yellow view controller when the user taps your `UIButton` in the red view controller. Holding down the Control key, select the button and drag the pointer to the second view controller. This action should highlight the yellow view controller, indicating it's a valid end point for the segue. [1]



When you release the mouse or trackpad button, you'll see a popover that allows you to specify the presentation method of the segue. There are multiple segues to choose from, but focus your attention on "Present Modally" and "Show." "Present Modally" will display the yellow view controller over the red, using a bottom-to-top sliding animation. You'll see this animation in the Mail app when you begin writing a new email, or in Contacts when you choose to create a new contact.

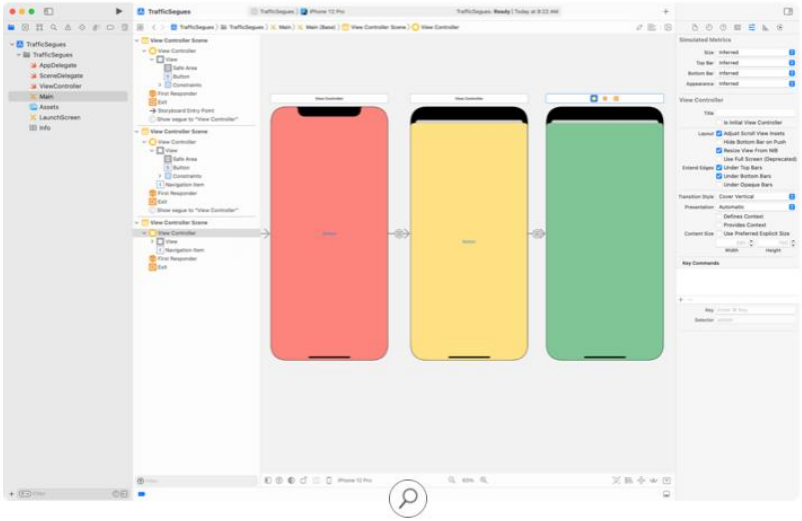


The Show segue also presents modally until a navigation controller is added to the storyboard scene. You'll add one of these later in the lesson. For now, select Show. An arrow appears from the red view controller to the yellow view controller indicating the segue. [1]

Build and run your app. When you click your `UIButton`, you should see the yellow view animate, from the bottom up, over the top of the red view.

Now add a third view controller, positioning it to the right of the yellow view. Set its background color to green. As in the previous steps, add a `UIButton` to the yellow view, create centering constraints, then **Control-drag** from the button to the green view controller and define a Show segue.

When you build and run your app, tapping the button on the the red view controller will modally present the yellow view controller, and tapping the button on the yellow view controller will modally present the green view controller.



Note that the red view controller is of type `ViewController`. You didn't assign a class to the yellow and green view controllers, so they'll be generic `UIViewController` instances. This distinction will be important when you implement the unwind segue.

Navigation and Workflows

Unwind Segue

You've just created a short sequence of segues. Although the user can swipe down to dismiss these views, it's best practice to always include a button to dismiss modal views as well. To do this, you need to create an unwind segue. Whereas a segue transitions to another scene, an unwind segue transitions *from* the current scene to return to a previously displayed scene.

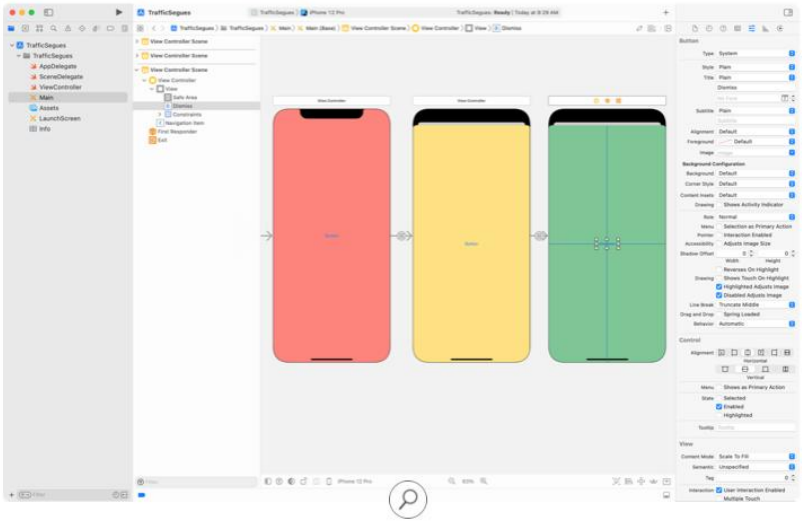
To begin, select `ViewController` in the Project navigator and add the following method just below the `viewDidLoad()` function:

```
@IBAction func unwindToRed(unwindSegue: UIStoryboardSegue) {

}
```

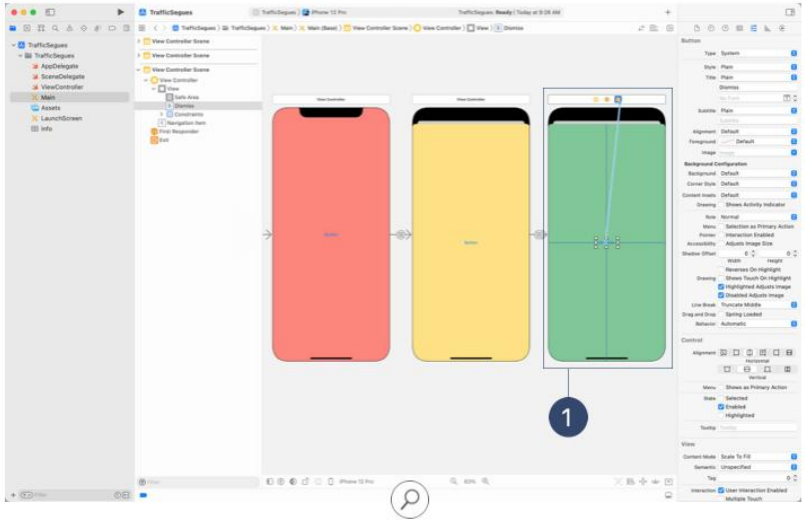
You can name the method anything you like, but it must take `UIStoryboardSegue` as its only parameter.

Unwind segues can be tricky to understand at first glance. By adding a function that takes a `UIStoryboardSegue` as a parameter to any scene's view controller definition, you're telling Interface Builder that the scene is a valid destination for an unwind segue.

In this lesson, the method you just added doesn't contain any code, but it can be used to pass information from the end point of the segue back to the source view controller.



Check out how that works. Back in the `Main` storyboard, add a button to the center of the green view. Update the button's text to read "Dismiss."



Control-drag the Dismiss button to the Exit object at the top of the view controller scene. [1] When you release the mouse or trackpad button, a popover appears, listing all available destinations for unwinding. In this case, there's only one option: `unwindToRedWithUnwindSegue`, which matches the method signature you placed in the definition of `` `ViewController` ``. Go ahead and select it.

Build and run your app. When you click the Dismiss button, it should unwind all the way back to the red view controller.

Navigation Controllers

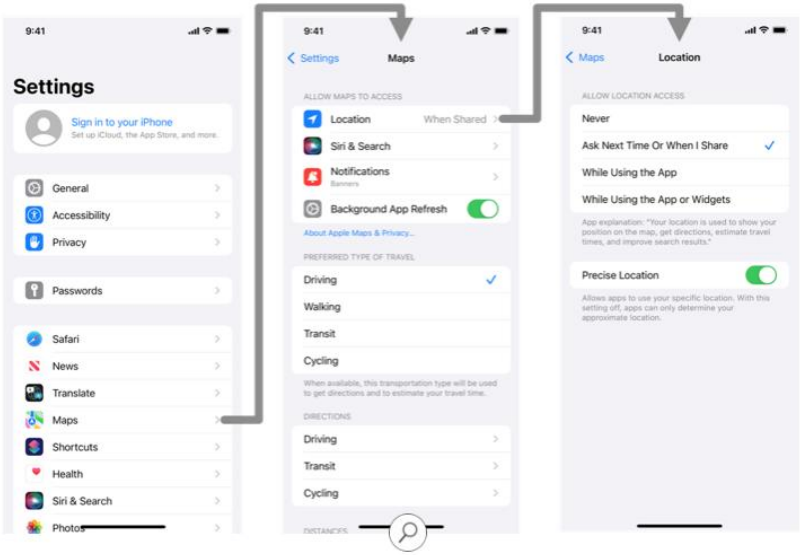
Modal segues are the preferred method of transitioning from one context to another within your app. For example, in the iOS Mail app, tapping the Compose button transitions from reading messages to writing messages. The Cancel button is always available if the user chooses to return to a previous context.

However, some situations require a segue from one view controller to a related view controller. For example, when the user taps a cell in Settings, a new view controller animates from right to left to cover the screen, visually adding to the stack of displayed view controllers. Adding a new view controller to the top of a stack is called pushing onto the stack. The following video illustrates the default animation of a push transition as compared to a modal transition in the Settings and Shortcuts apps.

Navigation and Workflows

Tapping the Back button in the top-left corner or swiping back dismisses the top view and returns to the next highest view controller, animating from left to right. Dismissing a view controller from the top of the stack is known as popping off of the stack.

Navigation controllers manage the stack of view controllers and provide the animations when navigating between related views.



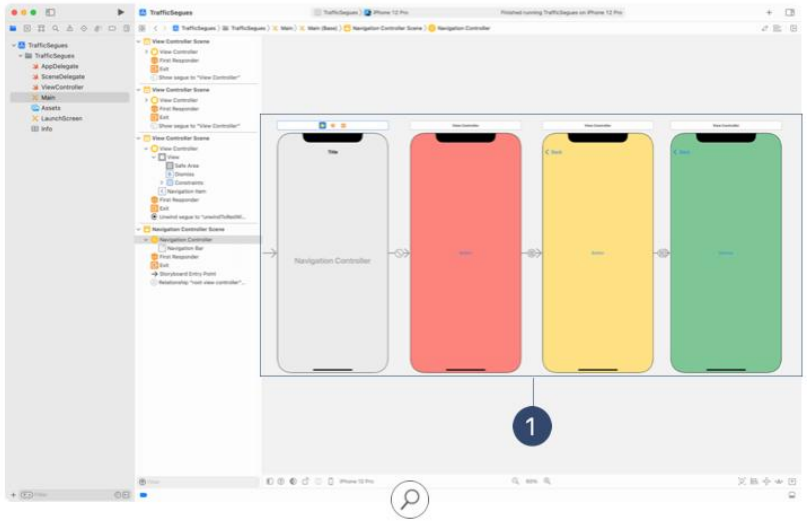
This push-and-pop structure is like washing a stack of dirty plates. After you wash and dry each plate, you place it in the cabinet. The first plate you wash will be at the bottom of the stack, and the last plate will be on the top. Later, when you grab a clean plate from the cabinet, you'll be grabbing the last plate you washed.

Now imagine that each plate is a view controller being pushed onto the screen. As you continue to push new view controllers, the first one—known as the root—moves farther down in the stack. Multiple taps of the Back button will eventually return to the root view controller, at which point the Back button goes away. Every navigation controller has a root view controller.

Another way to think about a navigation controller is that it mirrors a hierarchical data structure. In the case of Mail, the list of accounts (the root) gives you the ability to tap to reveal the account's folders. Tapping on each folder reveals its messages. Settings is similar. There's a list of setting categories (the root) one of which may be selected to reveal the settings or subcategories within, each time traveling deeper into the hierarchy. For the example above, from the root, General is selected and then Accessibility within that. Pushing a view controller onto the stack delves deeper into the hierarchy and popping a view controller from the stack travels back up the hierarchy to the root.

Back in your TrafficSegues project, you can use a navigation controller to manage the red, yellow, and green screens. Red will push to yellow, and yellow will push to green.

To add a navigation controller into your scene, select the red view controller. Next, click the Embed In button in the bottom toolbar and select Navigation Controller. Alternatively, go to the Xcode menu bar and choose **Editor > Embed In > Navigation Controller**.



Either of these methods will place a navigation controller at the beginning of the scene and set the red view controller as its root.

Build and run your app to see what's changed. You may notice a few important differences:

- The Show segues between the red, yellow, and green view controllers have adapted to Show (Push), rather than Present Modally. [1] (This is a key feature of the Show segue: It adapts the presentation method depending on whether it's used within a navigation controller or independently.)
- The Dismiss button still unwinds back to the red view controller, but it does so by popping off view controllers rather than dismissing them.
- At the top of each view is a transparent navigation bar, which provides space for the Back button as well as for a title and additional buttons.
- The Document Outline now includes a Navigation Controller Scene, which includes a Navigation Bar.

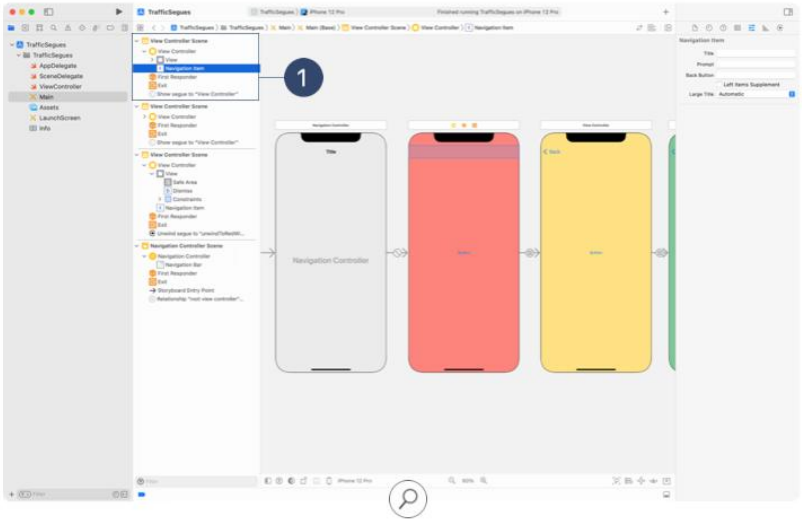
Navigation Bar

One of the most obvious features of a navigation controller is the navigation bar, which appears at the top of the screen. A navigation bar may display a title and/or button items.

Select the navigation bar in the Document Outline, and check out the Attributes inspector to see what properties you can customize, such as the bar's tint color, title color, and title font. (You might also choose to modify these properties in code.) View the documentation for `UINavigationBar` for a complete list of customizable properties.

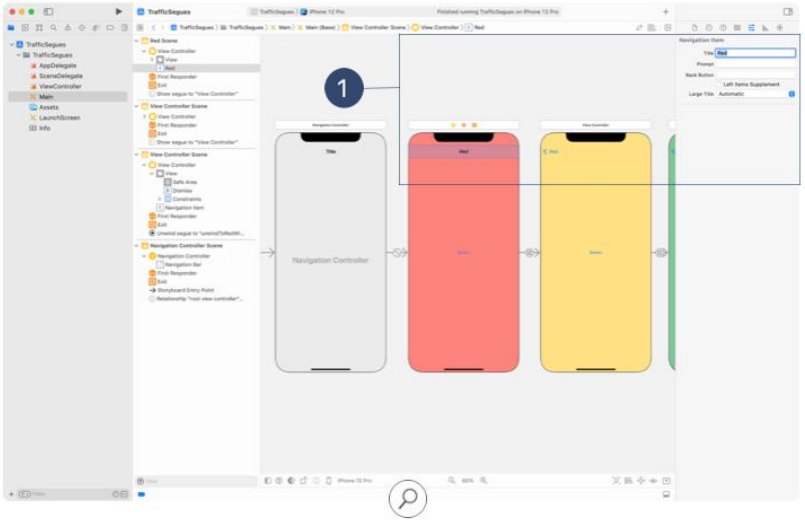
Navigation Item

Every `UIViewController` has a `navigationItem` that you can use to customize its navigation bar. [1] When you added the navigation controller in the earlier step, Interface Builder automatically added a navigation item to the root (red) view controller.

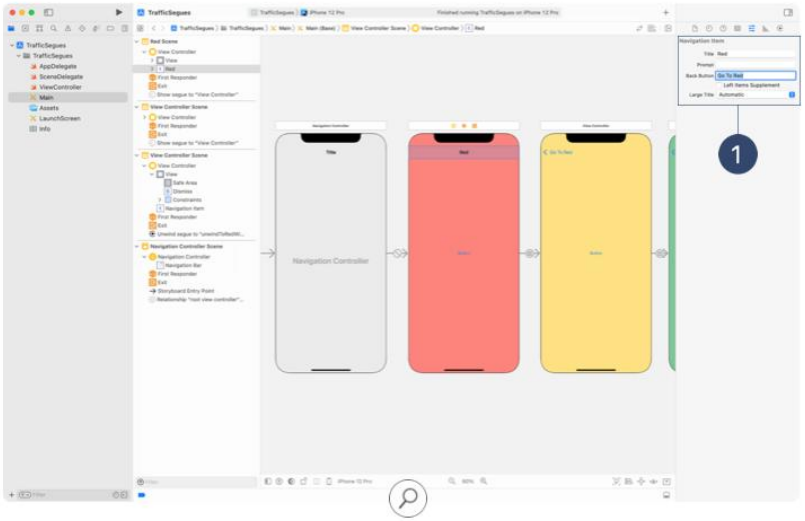


Navigation and Workflows

In the Document Outline, select the navigation item for the red view controller, and open the Attributes inspector. Enter "Red" in the Title attribute. [1]



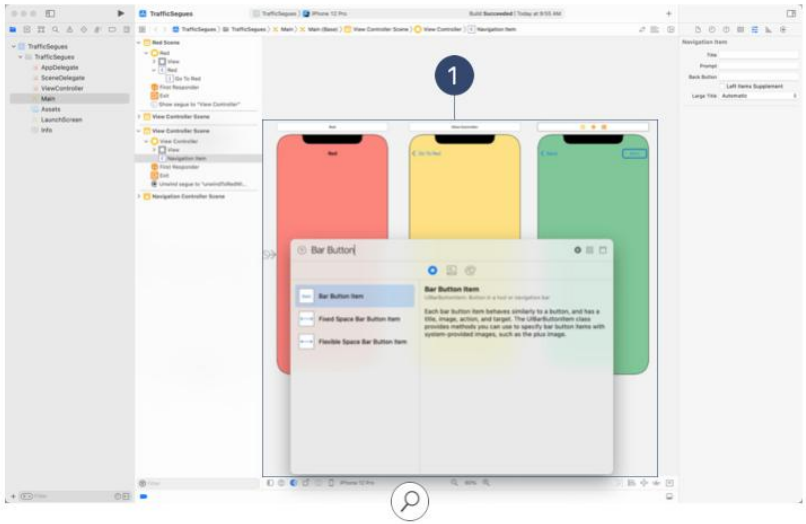
Build and run the app. When you push to the yellow screen, you'll notice that the Back button now displays "Red." How did that happen? The Back button used the title of the preceding view controller as its text. However, if the preceding view controller doesn't have a title, the Back button simply displays "Back." If you want the Back button to use other text, like "Go To Red," you can enter it in the Back Button field in the Attributes inspector of the red view controller's navigation item. 1



Under certain circumstances, Interface Builder will add a navigation item to view controllers. In the event it hasn't, you can add one yourself by finding a Navigation Item in the Object library, and dragging it on top of your view controller.

Set the titles of the yellow and green view controllers by selecting their Navigation Item and using the Attributes inspector to create "Yellow" and "Green" titles.

In addition to titles and Back buttons, navigation items can include a special type of button, known as a bar button, which can appear on either navigation bars or toolbars. Find the Bar Button Item in the Object library, and place one in the top-right corner of the green view controller's navigation bar. 1

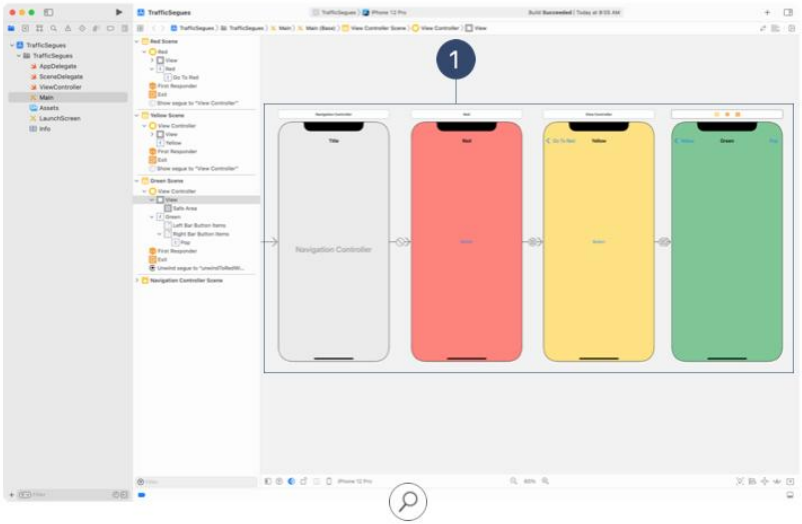


Click the bar button you just added, and open the Attributes inspector. In the System Item popup menu, you'll see commonly used button choices, such as Add, Save, and Cancel. Choose one or two to see the button text change. Play with the Style and Tint attributes as well. The Bar Item properties allow you to customize your button further. For example, you can use the Image field to replace a text title with an image or icon.

For now, go ahead and update the Title property of the bar item to "Pop." (Notice that this update changes the System Item option to Custom.)

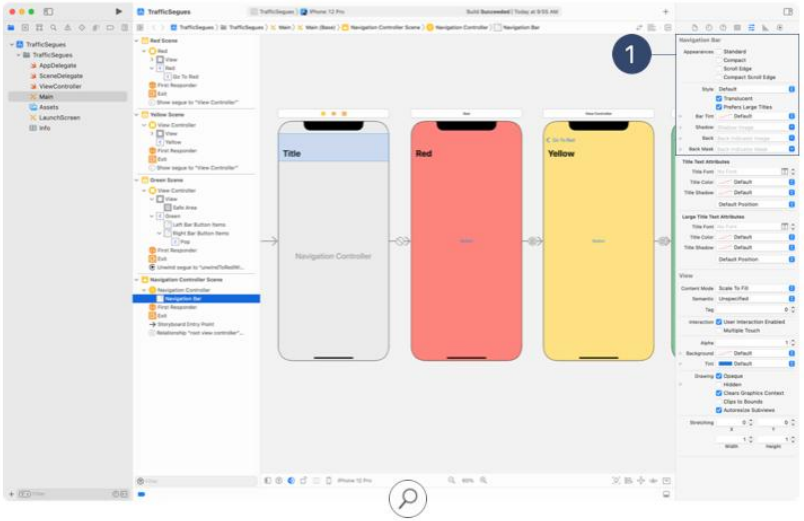
Next, you can wire this new bar button to the unwind segue. Refer to the same steps you used to connect the Dismiss button. Once you're done, you can delete the Dismiss button.

Your storyboard now has all its segues. [1] Build and run your app to see the titles and the button you just added. Notice that clicking the Pop button returns you to the red view controller.

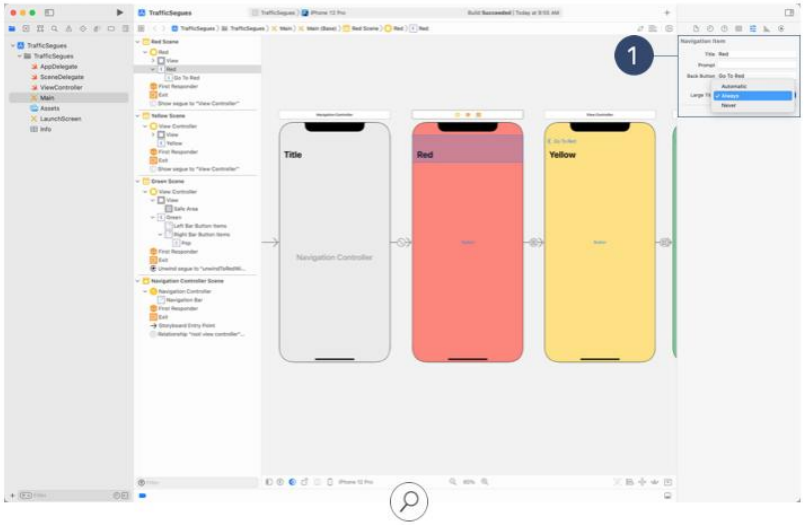


Large Titles

You may have noticed in system apps like Settings that the navigation bar title on the primary screen appears much larger than in subsequent scenes. This large title can be added to your own apps by selecting the navigation bar in Interface Builder, then checking the "Prefers Large Titles" box in the Attributes inspector. [1]



Note that all of your titles, not just the title of the first view controller, have now adopted a large title. You can adjust which view controllers use the large title option by selecting the navigation item in question and selecting one of the options from the Large Title dropdown in the Attributes inspector. [1]

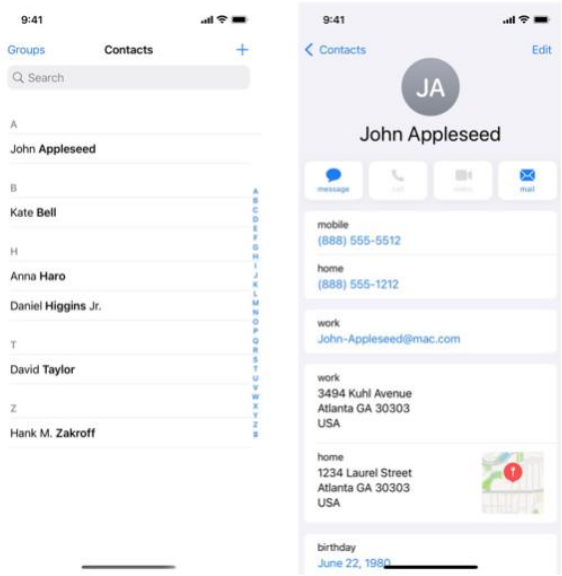


The Always option will ensure that the title of that navigation item will always be large; the Never option will ensure that the title of that navigation item will never be large; and the Automatic option will adopt the behavior of the previous view controller in the navigation stack. Unless you have a specific reason to do otherwise, a good first practice is to have your root view controller adopt a large title, and subsequent view controllers adopt smaller titles.

Navigation and Workflows

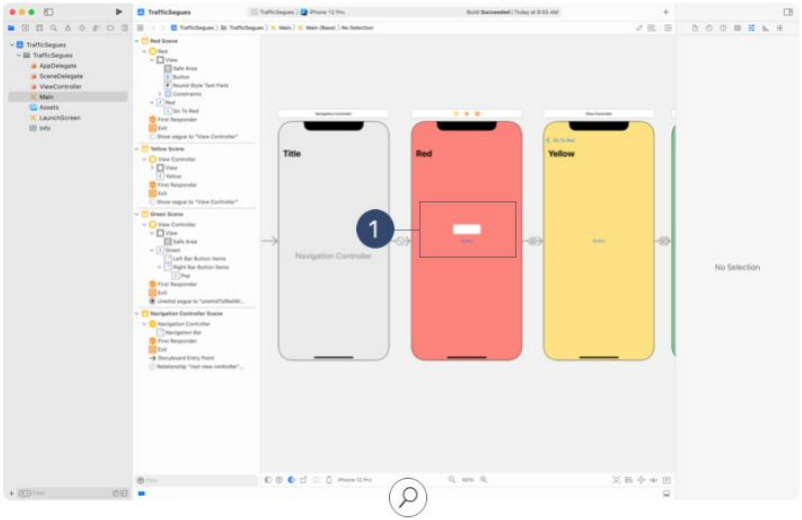
Pass Information

In many apps, you'll need to pass information from one view controller to another before a segue takes place. For example, when you tap a name in the Contacts app, details about the contact need to be relayed to the Information screen before it's presented.



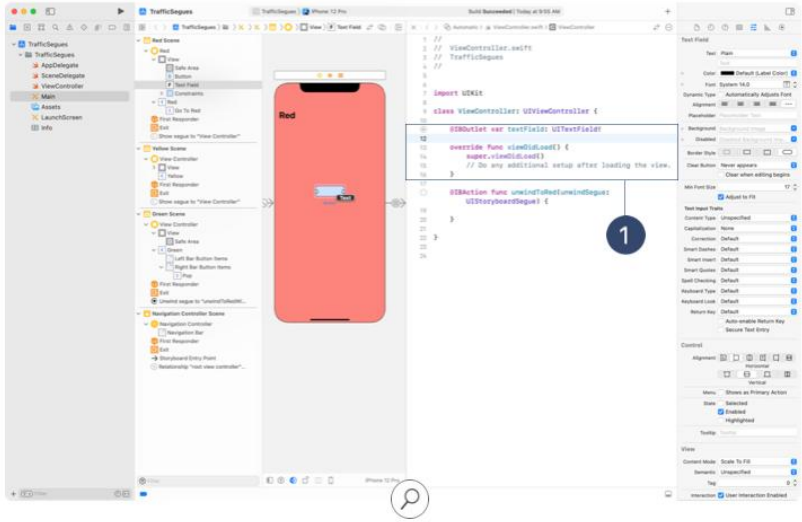
To learn how this works, you'll update the red view controller to include a text field in addition to the button. When the button is pressed, triggering the segue, Interface Builder will use any text in the text field as the title for the yellow view controller's navigation item.

Begin by dragging a text field from the Object library onto the red view controller, positioning it just above the button. For this example, don't worry about adding constraints. [1]



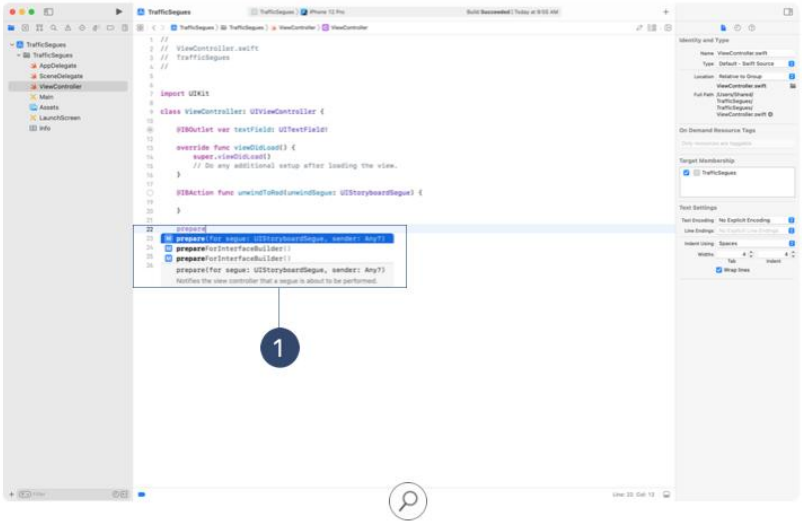
Navigation and Workflows

Create an outlet for the text field, naming it "textField." [1] You'll need to refer to the text field in code so that you can access the text and update the destination's title accordingly.



Now you'll set up the segue to pass the text from the text field to the yellow view controller.

Every `UIViewController` has a method, `prepare(for:sender:)`, which is called before a segue from the view controller takes place. Begin typing "prepare" near the bottom of the `ViewController` definition, but before the closing bracket. Xcode will offer to help you complete the method name. Choose `prepare(for segue: UIStoryboardSegue, sender: Any?)` and press the Return key to add the method. [1]



The first argument of this method is the segue itself. A segue contains a few properties that help to pass information across it:

- `identifier`—The name of the segue, which differentiates it from other segues. You can set this property in Interface Builder using the Attributes inspector.
- `destination`—The view controller that will be displayed once the segue is complete. While the value is a `UIViewController`, you may need to downcast it to a particular `UIViewController` subclass in order to access properties accessible only on that subclass.

Since there's only one segue on the red view controller, the `identifier` isn't needed. And since your goal is to update the `title` property of a navigation item and every `UIViewController` has this property, there's no need for the downcast. So the code to set the title of the destination's navigation item is fairly straightforward:

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

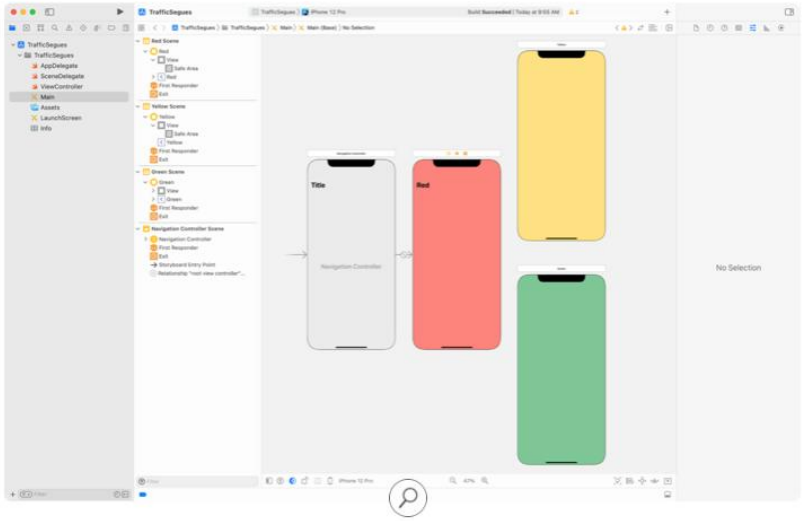
Build and run your app. Then enter a short string into the text field. When you press the button, the method you just added will update the title on the yellow screen.

Writing the code for passing data between scenes can be a little tricky at first. But you'll discover that passing data between screens is extremely useful and will allow you to write very flexible code. You'll continue working with passing information with segues in the Quiz project.

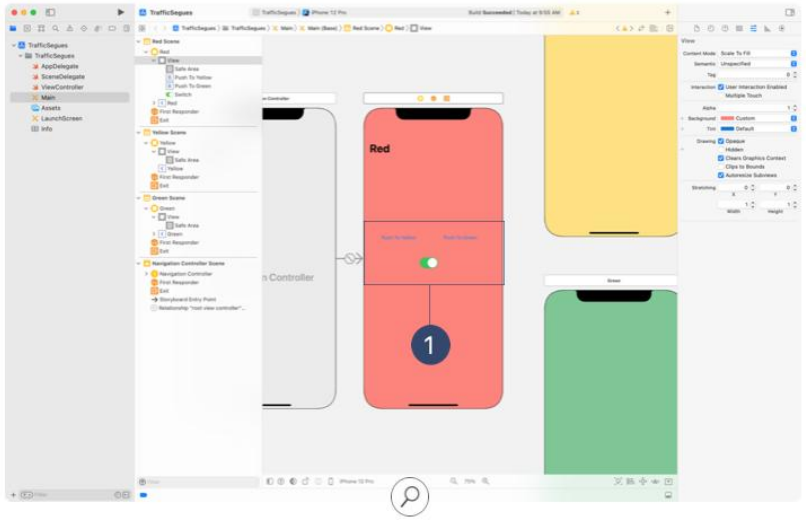
Create Programmatic Segues

Sometimes you'll need to use some logic to determine whether or not to perform a segue. Every segue that you've created at this point has involved dragging from a button to a view controller. When you create a segue this way, it will *always* be performed when the button is pressed. In this section, you'll define a few generic segues between view controllers and decide programmatically whether they should be performed.

Before you begin, restore your application to a clean slate. Remove all existing segues, controls, and labels from your storyboard. You also need to remove any outlets you've created, as well as the `prepare(for:sender:)` and `unwindToRed(segue:)` methods. Your app should look like this when you're ready:

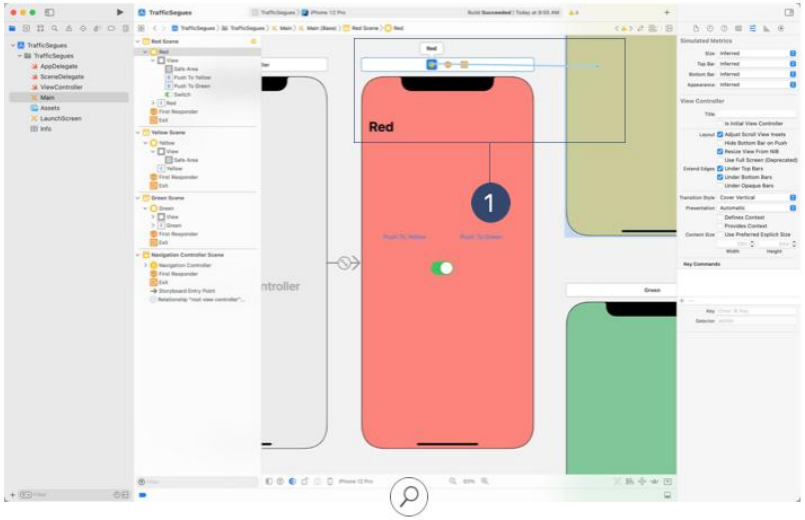


Navigation and Workflows



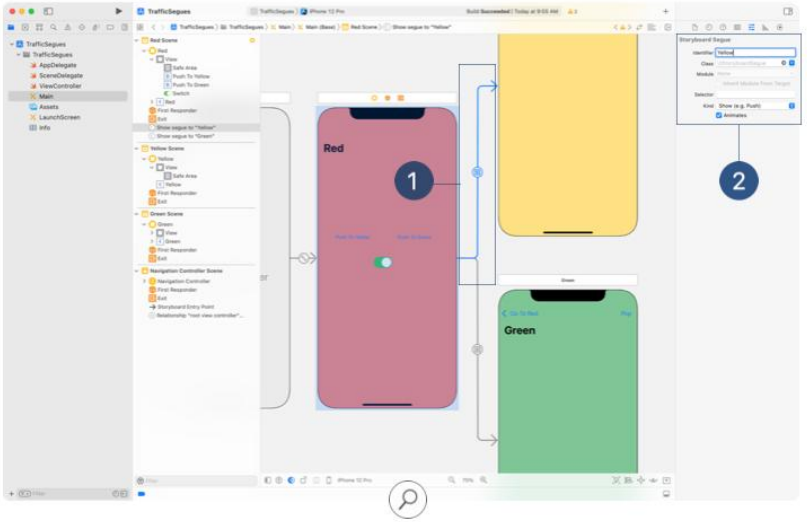
In this example, you'll place two buttons on the red view controller. One button will segue to the yellow screen, and the other will segue to green. The UI will also include a `UISwitch` that will determine whether or not the segues will be performed. If the switch is enabled, the segue can be performed; otherwise, no segue will take place.

Begin by placing two buttons and a switch from the Object library onto the red view. Update the title of one button to read "Push to Yellow" and the other to "Push to Green." Place the switch below the buttons, as shown. [1] In this example, don't be too concerned with creating constraints to position the views for every screen size and orientation.



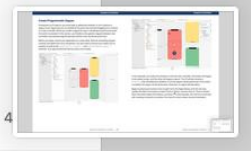
Rather than **Control-dragging** from the buttons to the view controllers, you'll want to **Control-drag** from the red view controller to the yellow and green view controllers. By doing so, you're defining two generic segues: one that moves from red to yellow, and another that moves from red to green. **Control-drag** from the view controller icon at the top of the red screen to the yellow view controller and create a Show segue, then repeat this step for the green view controller. [1]

In order to perform the segues in code, you'll need to give each segue a valid identifier string. Select the segue itself by tapping on the segue line. [1] In the Attributes inspector, name one identifier "Yellow" and the other "Green." [2]



You'll need to check the status of the `UISwitch` in code to determine whether or not to perform the segue. In order to do this, you'll need to create an outlet for the switch. Open the assistant editor and **Control-drag** from the switch to a valid location within the `ViewController` definition.

```
@IBOutlet var segueSwitch: UISwitch!
```



Create an action for each button by **Control-dragging** from the button to a valid location within the `ViewController` definition.

```
@IBAction func yellowButtonTapped(_ sender: Any) {

}

@IBAction func greenButtonTapped(_ sender: Any) {

}
```

To perform a segue programmatically, there's a method that exists on view controllers named `performSegue(withIdentifier:sender:)`. The first parameter for this method takes a `String`, which corresponds to the identifier that you assigned the segues in the Attributes inspector. The `sender` parameter is additional information that you can supply to the segue regarding which control triggered the segue, but it's not needed in this example and can be set to `nil`.

Call `performSegue(withIdentifier:, sender:)` in each method only if the switch is set to the "On" position.

```
@IBAction func yellowButtonTapped(_ sender: Any) {
    if segueSwitch.isOn {
        performSegue(withIdentifier: "Yellow", sender: nil)
    }
}

@IBAction func greenButtonTapped(_ sender: Any) {
    if segueSwitch.isOn {
        performSegue(withIdentifier: "Green", sender: nil)
    }
}
```


Build and run your application. When you press each button, the corresponding action will be triggered. Each action checks the status of the switch, and performs the appropriate segue if the switch is enabled.

Recall that the `identifier` property of the `UIStoryboardSegue` passed to `prepare(for:sender:)` tells you the name of the segue that's about to be performed. Examine the API documentation for `shouldPerformSegue(withIdentifier:sender:)`. How could you move the check of `segueSwitch.isOn` to `shouldPerformSegue(withIdentifier:sender:)` rather than inside each button's action?

Lab—Login

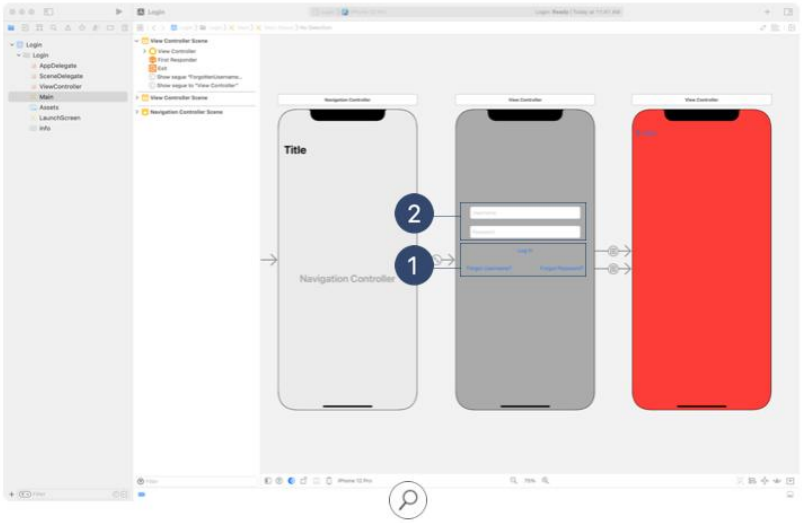
The objective of this lab is to create a login screen that passes a username between view controllers. You'll use view controllers, a navigation controller, and segues to create both the login screen and a simple landing screen that displays in its title either the username or text related to a forgotten username or password.

The instructions in this lab won't cover constraints and Auto Layout. But as you work through the steps, take some time to ensure that your views adapt to different screen sizes and orientations.

Create a new project called "Login" using the iOS App template.

Step 1

Create Your Storyboard Scenes With Simple Segues



- Using the image on the preceding page, configure your storyboard to have two view controllers, the first one for the login screen and the second for the landing screen.
- Add two text fields and three buttons to the login screen, also following the image. Change the text of the first button to say "Log In," the second to say "Forgot Username," and the third to say "Forgot Password." [1]
- For the two text fields, use the Attributes inspector to set placeholder text to "Username" and "Password." [2] To hide text that's entered in the password text field, select the Secure Text Entry checkbox at the bottom of the Attributes inspector.
- From the login button to the landing screen, create a Show segue.
- Using the Attributes inspector for the landing screen view controller, find the Presentation option and choose Full Screen. This presents the view as a full screen rather than as a card that's dismissible by swiping down. You want the user to feel they've entered your app—the login screen shouldn't be lingering in the background.
- Run the app. Check that you can segue from the login screen to the landing screen. Since you haven't set up a way to pass information from the login, your landing screen will be blank.

Step 2

Add Navigation Controller And Prepare For Segue

- Embed your login view controller in a navigation controller and add a title to the navigation item. Change its Large Title option to Always. Select the navigation controller's navigation bar and check Prefers Large Titles.
- Select the navigation item for the landing screen. Change its Large Title option to Always.
- In the `ViewController` file add the `prepare(for:sender:)` method. Feel free to rely on autocomplete to properly override the method.
- Create an outlet from your username text field to the `ViewController` file.
- In `prepare(for:sender:)`, set the title of the destination view controller's navigation item to the text from the username text field.
- Run the app and ensure that what you type in the username text field appears in the title of the landing screen when you tap the login button.

Step 3
Add Programmatic Segues

- Create a segue from the login view controller (not the login button) to the landing view controller. Be sure to give the segue a descriptive identifier.
- Create an outlet from each of the two remaining buttons (the Forgot Username button and the Forgot Password button), and give them descriptive names like "forgotUserNameButton."
- Create an action from each of the buttons.
- Within each action, call `performSegue(withIdentifier:sender:)`, passing the identifier of the most recently created segue. Instead of setting `sender` to `nil`, set `sender` to be the button that was tapped. For example, if the segue identifier is `ForgottenUsernameOrPassword` then the inside of the button's action might look as follows:

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

- Earlier in this lesson, you learned that the `prepare(for:sender:)` gives you access to the identifier of the segue that was called. Now that you have two possible identifiers, you need to use control flow statements to pass different information to the landing screen based on which segue was called. If the login segue was called, you want the landing screen's title to be the username. However, if the Forgot Password button was tapped, you want the title to read "Forgot Password." Similarly, tapping the Forgot Username button will change the title to "Forgot Username." Before reading on, take a minute to see if you can do this on your own using `segue.identifier, sender`, downcasting, and `if` statements.
- It's OK if you didn't get it on your own. This is new material. If you want to pass a different title based on which button was tapped, the body of your `prepare(for:sender:)` method might look like the following:

Navigation and Workflows

```
guard let sender = sender as? UIButton else {return}

if sender == forgotPasswordButton {
    segue.destination.navigationItem.title = "Forgot Password"
} else if sender == forgotUsernameButton {
    segue.destination.navigationItem.title = "Forgot Username"
} else {
    segue.destination.navigationItem.title = usernameTextField.text
}
```

- The code above casts the sender as a `UIButton`, which in this case always succeeds. Why is that? The login segue is specifically triggered by a button, and the `ForgottenUsernameOrPassword` segue is triggered by the method `performSegue(withIdentifier:sender:)`, where you passed in the corresponding button as the `sender`. After that, an `if` statement checks whether `sender` was `forgotPasswordButton` and sets the title accordingly. If `sender` wasn't `forgotPasswordButton`, another `if` statement checks whether `sender` was `forgotUserNameButton` and sets the title accordingly. If `sender` wasn't `forgotPasswordButton`, there's only one remaining case—when the login button was tapped—which sets the title to the username.

Great job! You are well on your way to making useful apps! Be sure to save your work in your projects folder.

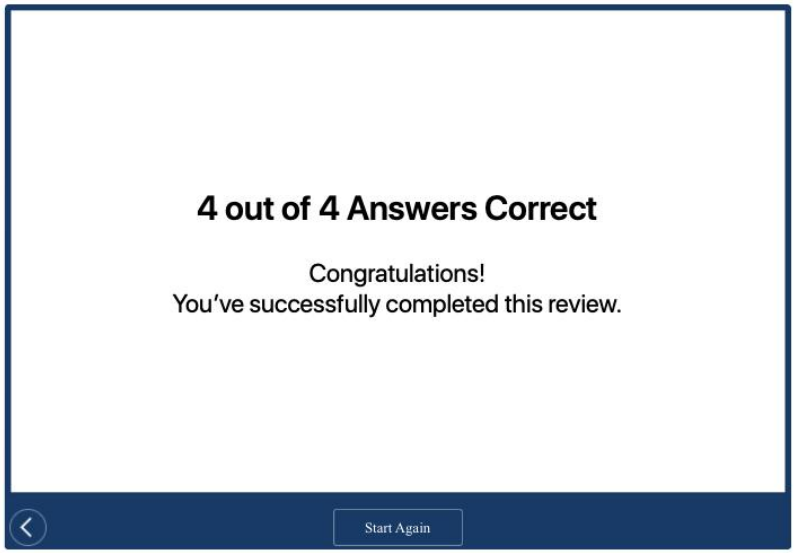
Connect To Design

In your App Design Workbook, reflect on your need for segues and different kinds of view controllers. Will you need navigation controllers to display related or hierarchical content? Make comments in the Prototype section or in a new blank slide at the end of the document.

In the workbook's Go Green app example, a navigation controller could be used to manage the views in the different actions a user would want to do in the app, like navigating from the achievements summary screen to the detail for one achievement.

Navigation and Workflows

Review Questions



Lesson 3.8

Tab Bar Controllers

In the last lesson, you learned how to navigate from one view controller to another. But as you add features to an app, you may realize that drilling up and down with a navigation controller just doesn't cut it. It may be time to flatten out your view controller hierarchy.

In this lesson, you'll use tab bar controllers to organize different kinds of information or different modes of operation. Tab bar controllers are key to navigating between view controllers, allowing you to comfortably pack more functionality into a single app.

What You'll Learn

- How to appropriately use a tab bar controller
- How to add a tab bar controller
- How to add view controllers to the tab bar controller
- How to customize tab bar items

Vocabulary

- **badge**
- **flat hierarchy**
- **system item**
- **tab bar**
- **tab bar controller**
- **tab bar item**

Related Resources

- iOS Human Interface Guidelines: Tab Bars
- API Reference: UITabBarController
- API Reference: UITabBar
- API Reference: UITabBarItem
- API Reference: UIViewController tabBarItem

A tab bar controller allows you to arrange your app according to distinct modes or sections. For example, the Clock app is divided into five modes: World Clock, Alarm, Bedtime, Stopwatch, and Timer.

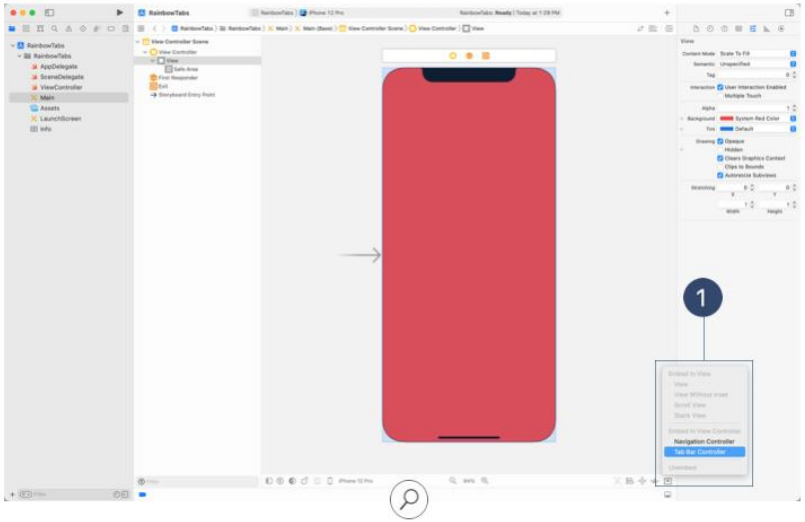
As you'd expect, a tab bar interface features a tab bar view, which runs along the bottom o the app's screen. Each tab can contain its own independent navigation hierarchy, with the tab bar controller coordinating the navigation between the different view hierarchies. The tab bar distinguishes the currently selected tab with a different-colored icon and title.

Add A Tab Bar Controller

To practice building a tab bar interface, you'll create a simple app that navigates between several root view controllers. (This could be a good template for future projects that also use a tab bar controller.)

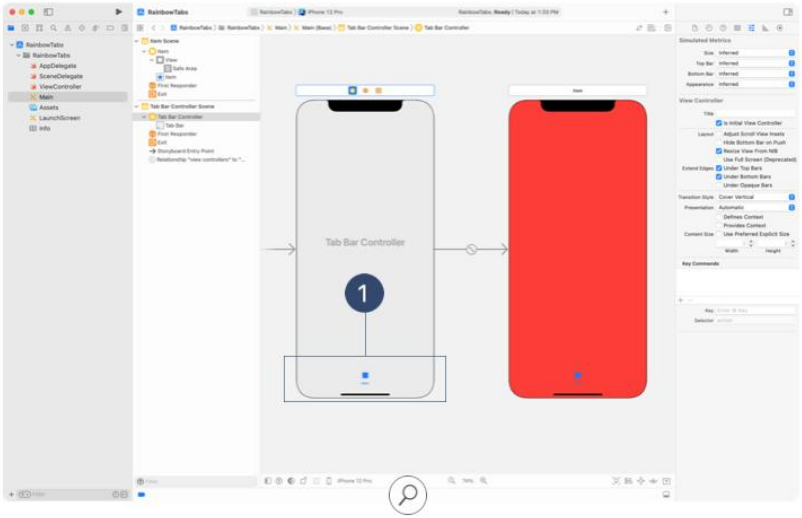
Start by creating a new project named `RainbowTabs` using the iOS App template. Open the `Main` storyboard and, in the Document Outline, select View under View Controller. Using the Attributes inspector, set the view's background color to System Red.

Next, you'll create the tab bar controller. With the red view selected, click the Embed In button in the bottom toolbar and select Tab Bar Controller. [1] Alternatively, go to the Xcode menu bar and choose **Editor > Embed In > Tab Bar Controller**.



Navigation and Workflows

This action places a tab bar controller at the beginning of the scene.

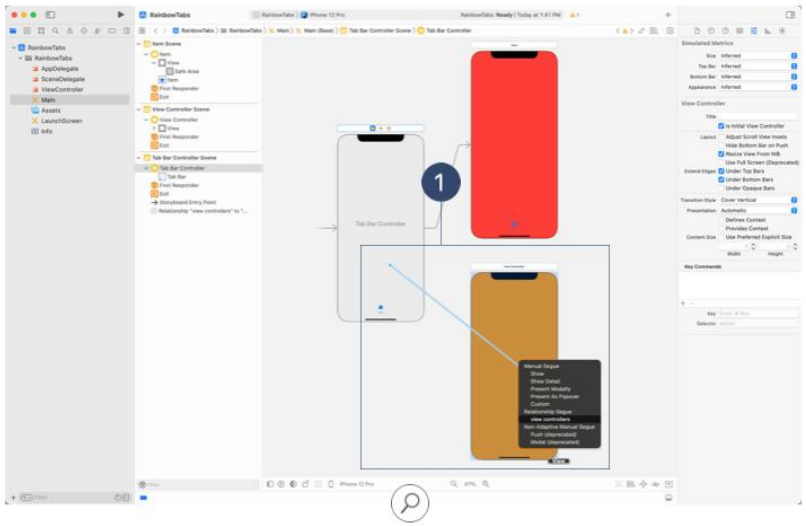


The tab bar controller maintains a list of tabs through its `viewControllers` property, an array of the root view controllers displayed by the tab bar interface.

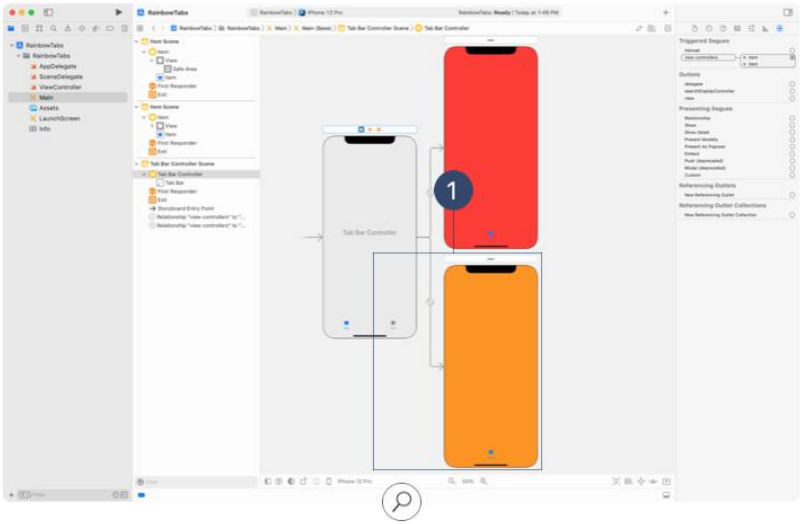
That last step added the red view controller to the tab bar controller's array of root view controllers. For each root view controller, there's an associated `UITabBarItem` instance. You now have a tab bar with one tab bar item. [1]

Add Tabs

To add another tab bar item, select View Controller in the Object library, and drag it onto th canvas. Give the view controller an orange background. Next, you'll need to add the new view controller into the `viewControllers` array. **Control-drag** from the tab bar controller to the orange view controller, and release the mouse or trackpad button. In the popover, you can see "view controllers" listed under Relationship Segue. Choose this option. [1]



Navigation and Workflows

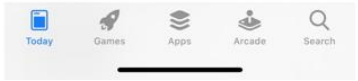


You should now see a second tab bar item on the tab bar controller. [1]

Build and run your app. Notice that you can switch between the two view controllers by selecting a different tab bar item.

Tab Bar Items

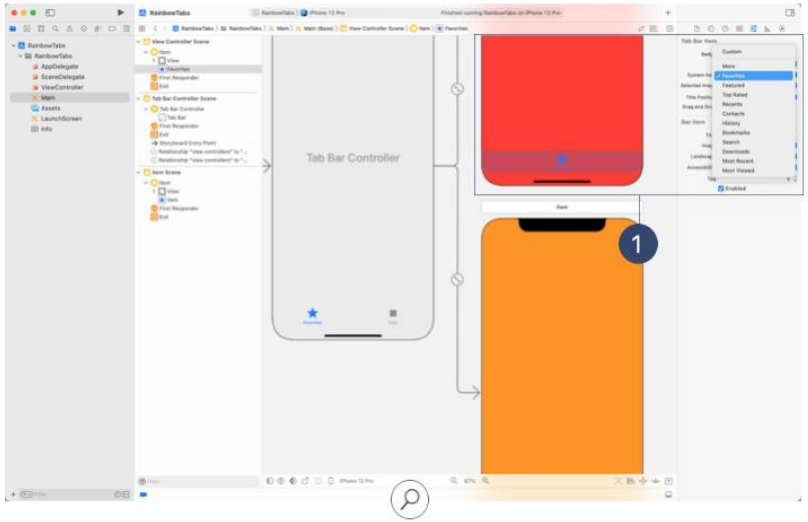
A tab bar item consists of two visual elements: an image and a label. The iOS SDK provides several iOS-style icons paired with system-defined text, referred to as system items. One example is the search icon that you see in the App Store tab bar.



Here's a complete list of available system item title with their corresponding icons:

Title	Image
More	
Favorites	
Featured	
Top Rated	
Recents	
Contacts	
History	
Bookmarks	
Search	
Downloads	
Most Recent	
Most Viewed	

In your RainbowTabs project, select the tab bar item in the red view controller, and open the Attributes inspector. Choose any of the system items from the System Item pop-up menu. [1] Notice how the tab bar item adjusts to your different selections.



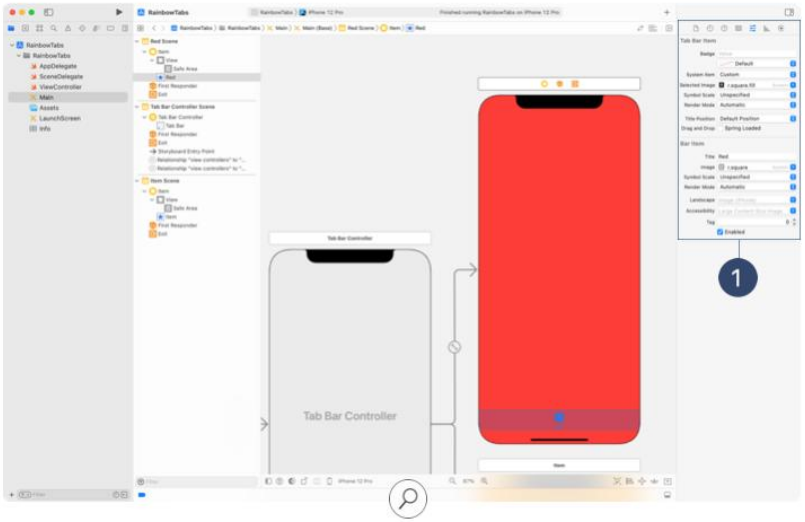
Now change the device orientation in Interface Builder to landscape using the "Orientation" button. Notice how the tab bar item's title now appears to the right of the icon. When in portrait, tab bar items will be displayed with the icon just above the item title. In landscape, the icon and title will be displayed side by side. When on an iPhone in landscape, the tab bar will be thinner and have smaller images if you provide a smaller image to the `.landscapeImagePhone` property. That way less content is obscured by the tab bar.

Customize Tab Bar Items

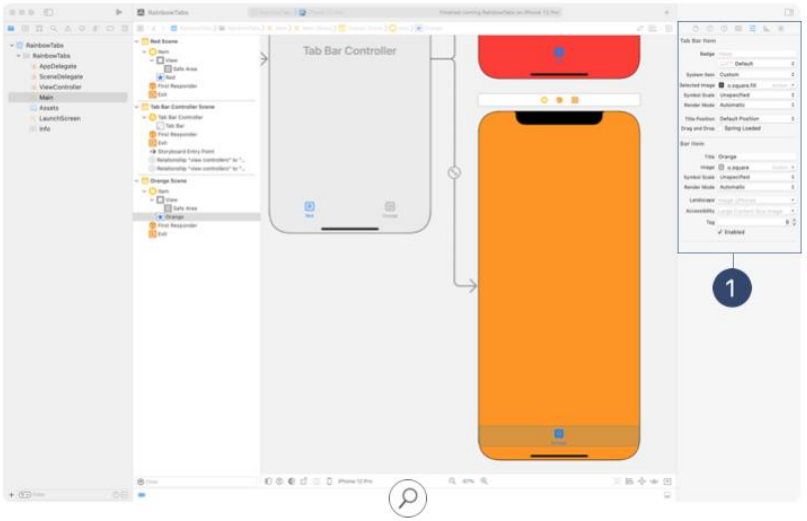
But maybe the system items don't make sense for your app. You can use the Attributes inspector to customize an item's label and its image, for both unselected and selected states. When providing icons of any kind for an iOS app, you should review the Human Interface Guidelines for Custom Icons to ensure that you're designing your icons in the right format and size.

Before designing your own icons, consider using SF Symbols. These vector icons work great at all sizes, and you have lots to choose from. You can browse the full set of symbols using the SF Symbols app. You can also type a common symbol name—for example, "circle"—into an Image field in the Attributes inspector to see what's available.

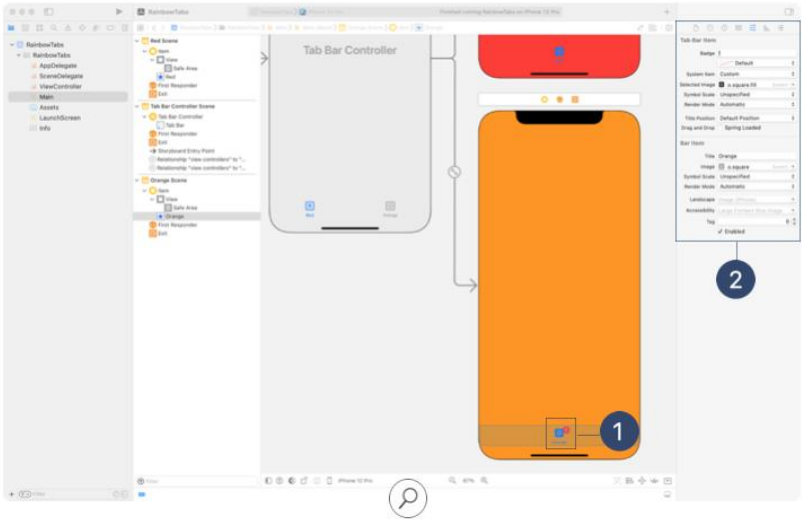
With the red view's Tab Bar Item still selected, type "Red" in the Title field and type "r.square" into the Image field. From the menu, choose the matching symbol. The image is provided through SF Symbols as denoted by the "System" label next to the arrow. [1] You can also customize the Selected Image attribute to distinguish the tab item's selected state ("r.square.fill" may be a good choice).



Go ahead and make changes to the tab bar item for the orange view controller. [1]

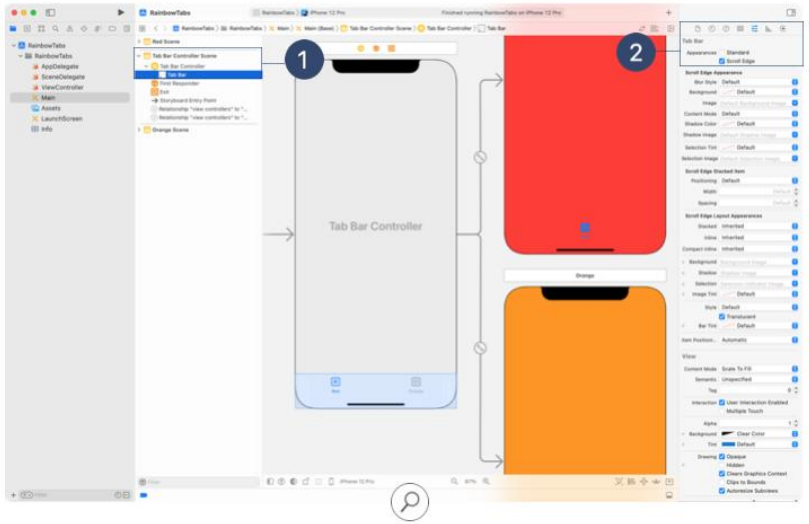


What if you want your tab bar item to indicate that new information is available for that view or mode? You can add a small red marker with white text, known as a badge, to the tab bar item. [1] Use the Badge attribute to enter information. [2]



Configuring The TabBar

The background of the views in this application are a solid color and the tab bar will be presented as transparent in your app. This may make it difficult to see the tab bar items or the red badges. The tab bar supports an appearance option that will show the tab bar with a blur background at all times. Click on the Tab Bar in the Tab Bar Controller [1] and click the checkbox next to Scroll Edge [2] in the Appearances section of the Attributes inspector.



Programmatic Customization

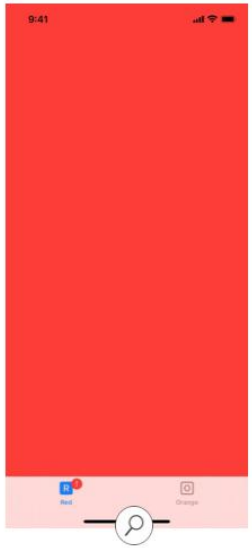
Storyboards are ideal for setting up initial, or default, view scenes; but they don't allow you to make runtime adjustments using the Attributes inspector. That's OK. You can accomplish any of these customizations in code.

For example, imagine you want to alert your user that new information is available. Your app would have to update the badge at runtime. To assign a badge in code, set the `badgeValue` property to a non-nil string. You can access your view controller's `UITabBarItem` instance through its `tabBarItem` property. For more explanation, you can reference the UIViewController Documentation.

In `ViewController`, insert the following line to the `viewDidLoad()` function:

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

Run your app in Simulator. You'll notice the red tab item now has a badge.



The badge draws your user's attention to that tab. After they've viewed the new information, the badge is no longer necessary. To remove the badge, assign a `nil` value to the `badgeValue` property in a `viewWillDisappear(_:)` method.

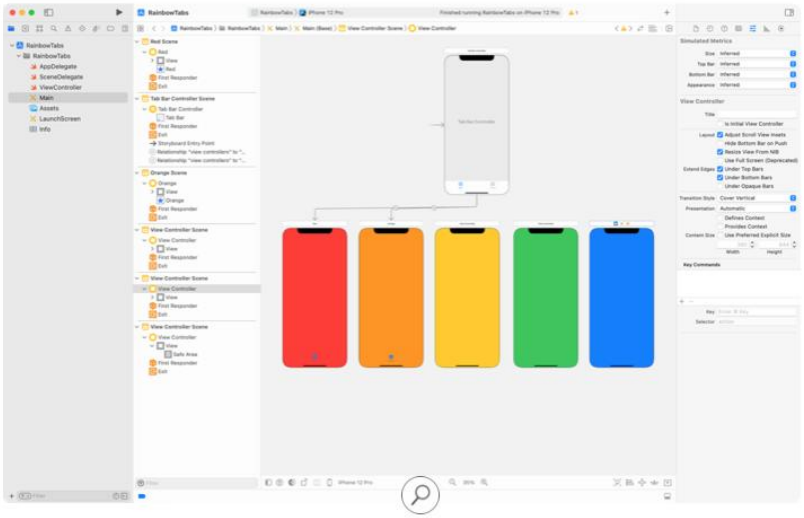
```
override func viewWillDisappear(_ animated: Bool) {
    super.viewWillDisappear(animated)

    tabBarItem.badgeValue = nil
}
```

Build and run the app again, when you navigate to the Orange tab and you will see the badge on the red tab disappear.

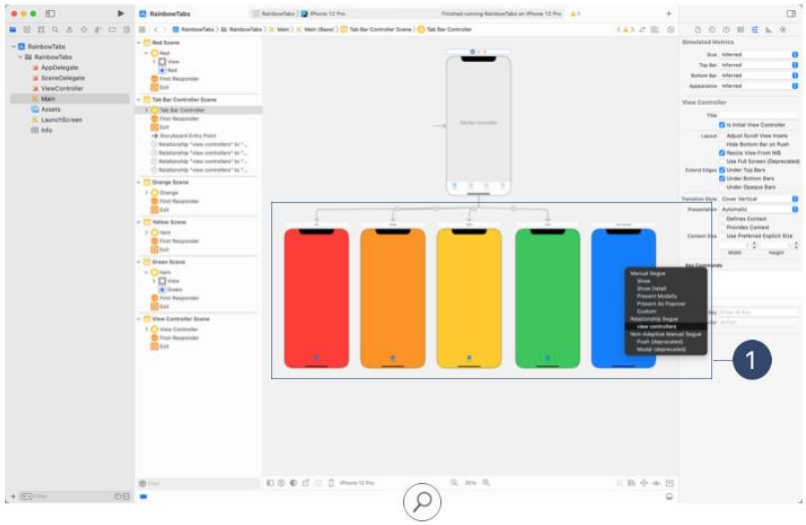
Even More Tab Items

You've probably noticed that the current version of your project doesn't live up to its "RainbowTabs" name. Try adding three more colors in three more tab items. A tab bar will display all tab bar items as long as there's enough horizontal space.

Select View Controller in the Object library, and add three of them to the canvas. Set the background color of one view controller to yellow, another to green, and the third to blue. If you find it helpful, you can reposition the view controllers on the canvas to match the order on the tab bar.

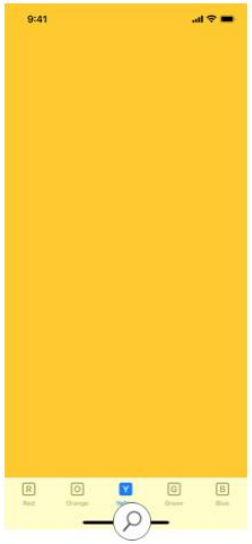


Add each of these view controllers to the `viewControllers` property of the tab bar controller. [1]



You can update the new tab bar items with images from the assets catalog or from SFSymbols.

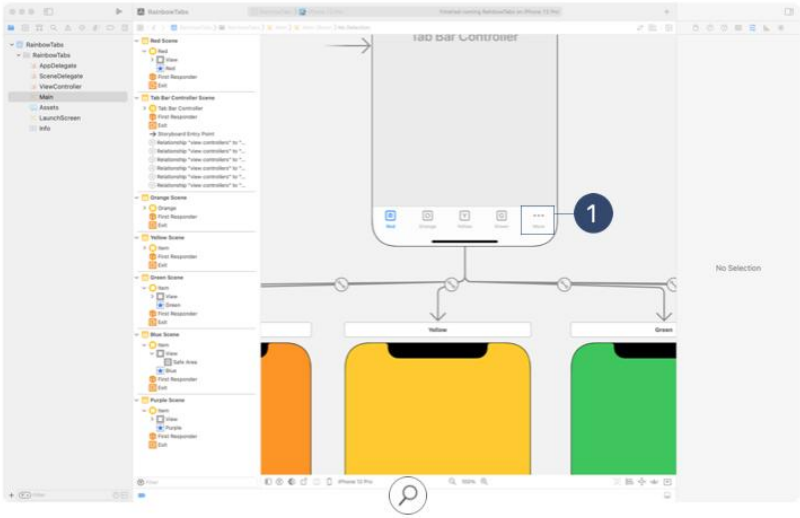
Run your app in Simulator on an iPhone device. With five items on the tab bar, there isn't much space left. What do you think will happen if you add a final purple view controller to the tab bar controller?



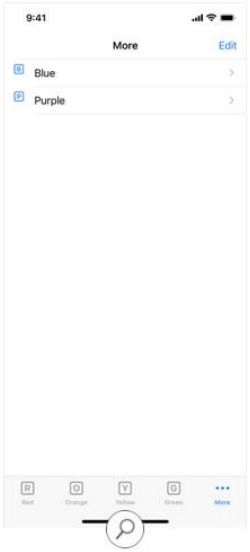
Navigation and Workflows

Drag another view controller onto the canvas, update its background color to purple, and add it to the tab bar controller's array of view controllers. You'll notice, in your tab bar controller scene, the fifth tab is replaced with a More tab item. [1]

Run your app and check it out.



What just happened? Whenever you add more view controllers than the tab bar can display, the tab bar controller inserts a special view controller, known as the More view controller. This view controller lists the omitted view controllers in a table, which can expand to accommodate any number of items.

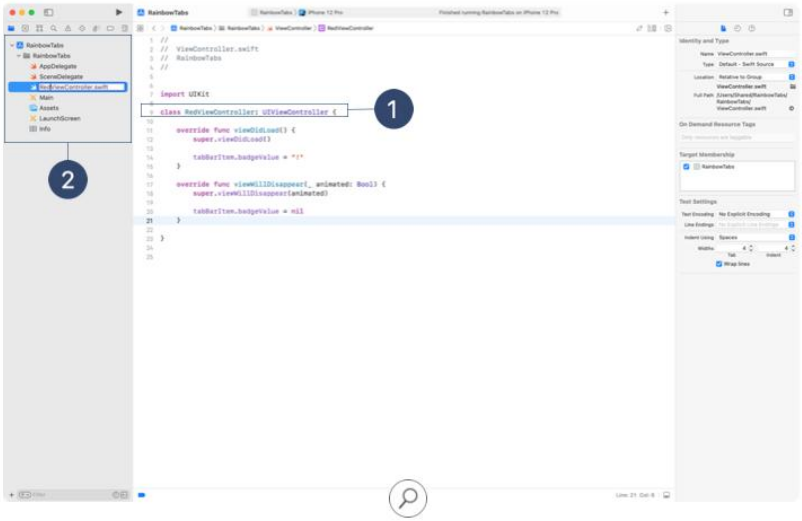


The More view controller is unusual. It can't be customized or selected. It doesn't appear among the view controllers managed by the tab bar controller. It appears when needed and is otherwise separate from the rest of your content.

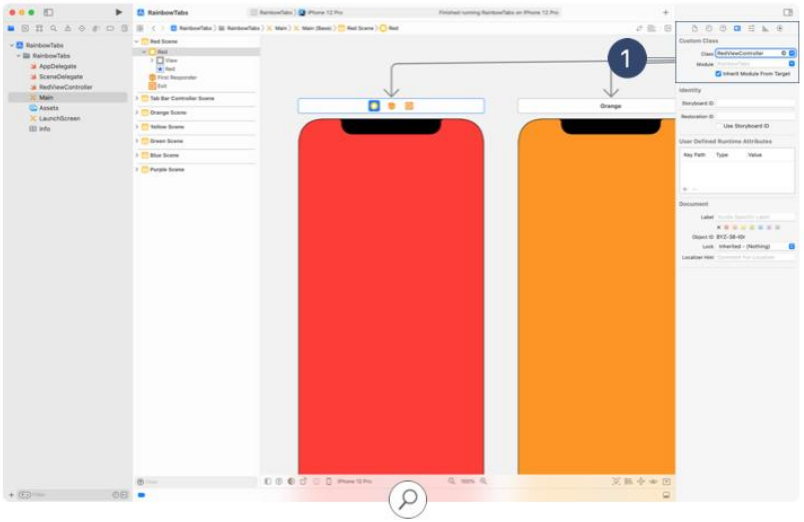
The More view controller can be quite useful for displaying additional tab items, but also consider that a More tab requires more time and effort from the user. A much better practice is to plan your app carefully so that you include only essential tabs—the minimum number necessary for your app and its information hierarchy. For iPhone apps, five is generally considered the maximum; for iPad-only apps, you can add a few more.

Link The Tabs To Code

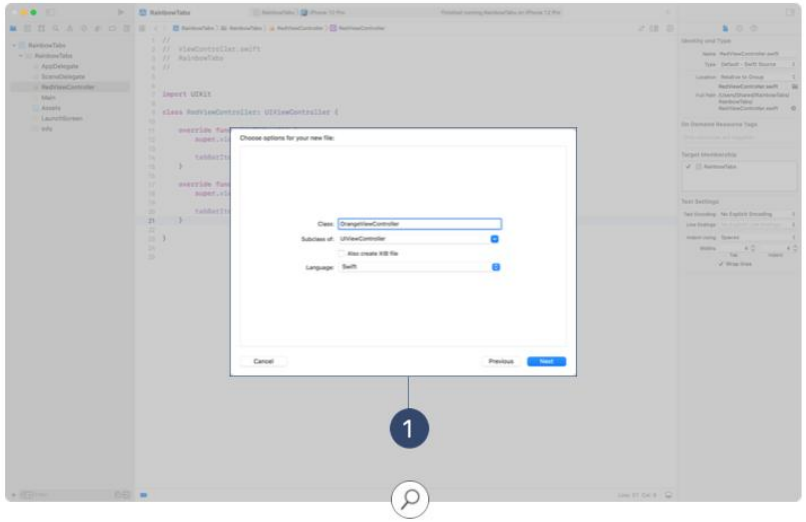
Currently, only the controller with the red view can be customized with code, because it's the only controller that uses a `UIViewController` subclass. That subclass is called `ViewController`, and it was created as part of the iOS App template. At some point, you're going to want to add new functionality into the other view controllers, such as fetching data or managing a user account. Regardless of the view controller's responsibilities, there is a high chance you'll need to use code in order to perform a task. This means you'll need additional `UIViewController` subclasses, one for each of the different colored screens.



Begin by renaming the existing `ViewController` class to something more descriptive, such as `RedViewController`. [1] Update the class definition, then rename the `ViewController` file to `RedViewController`. [2] The filename does not have to match the class name, but it makes the class definition easier to locate in the future for yourself or other members of your team.



You'll need to update the red view's controller class from `ViewController` to `RedViewController` as well, since the `ViewController` class no longer exists. Highlight the red view's controller in the Document Outline, then use the Identity inspector to set the custom class to the new name. [1]



Now you're ready to add new view controllers to your project. Select **File -> New -> File** (**Command-N**) from the Xcode menubar. Select "Cocoa Touch Class" as your starting template, then click Next. Set the subclass of your new class to `UIViewController`, then give your class a new name, `OrangeViewController`.[1] It is convention to append "ViewController" to the end of your class name so it's clear what type of object you're subclassing. Click Next, then Create, to finalize the subclass creation.

Now that you have a new view controller to work with, you can update the controller of the orange view to `OrangeViewController`. Open the `Main` storyboard and select the orange view's controller in the Document Outline. Use the Identity inspector to set the custom class to `OrangeViewController`.Repeat the steps of creating a new `UIViewController` subclass for every tab, and assign each view controller a unique custom class.

Challenge

Change the tab bar controller to use three navigation controllers as its `viewControllers`. Each navigation controller's root view controller should be one of the colored view controllers.

Lab—About Me

Objective

The objective of this lab is to use a tab bar controller to display different modes of information or operation. You'll create an app that displays distinct types of information about yourself in separate tabs. This app is similar to the "Hello" app you created earlier—feel free to use some of the same images and information.

Create a new project called "AboutMe" using the iOS App template. As you go through the steps below, remember that this app is—like the name implies—all about you, so make it personal..

Step 1

Set Up Your View Controllers

- Drag at least three view controllers from the Object library. Each view controller will represent a facet of your life. The example app uses one view controller for bio, one for family, and one for hobbies.
- Give each view controller a background color. Or if you want to get fancy, drag an image view to cover the entire view controller and set a background image.
- On each view controller, drag out labels to provide text about yourself, your family, your hobbies, or whatever personal info you've decided to include. Do you want to include some photos? Drag out image views and set their images. Be sure to add constraints that will keep your views arranged consistently on different screen sizes and orientations.

Step 2

Set Up Your Tab Bar Controller

- Select all your view controllers and embed them in a tab bar controller.
- Give each tab a title that fits the personal info you're adding in that section.
- Set the tab icon on each tab. You may use the assets provided, use SFSymbols, or find your own icons.
- Run the app. Check that the tab titles and icons appear.

Congratulations! You've made an app that tells a little bit about yourself. Be sure to save it to your projects folder.

Navigation and Workflows

Connect To Design

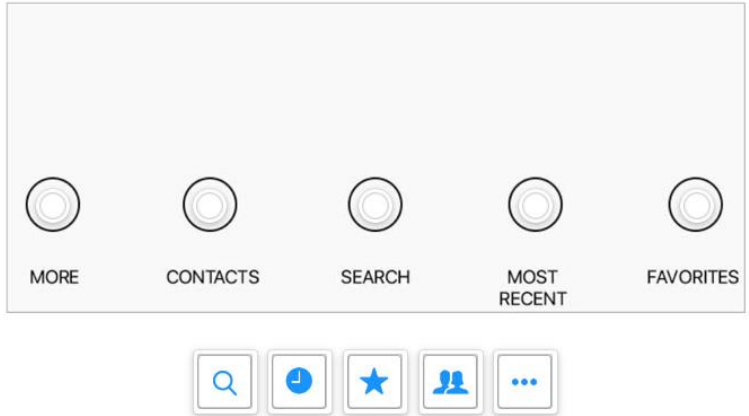
Open up your App Design Workbook and review the Prototype section for your app (or review the prototype itself). Did you plan for, or do you need, a tab bar controller? Add comments to the Prototype section or in a new blank slide at the end of the document.

In the workbook's Go Green app example, a tab bar would be used to toggle between the "My Forest", log, challenges, achievements, and emporium views.

Review Questions

Question 1 of 4

Match the system item image with its title.



Lesson 3.9

View Controller Life Cycle

Now that you've learned the basics of Interface Builder, you know that view controllers are the foundation of your app's internal structure. Every app has at least one view controller, and most apps have several.

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

What You'll Learn

- Appropriate times to perform work within the view controller life cycle
- How to add and remove views from the view hierarchy

Vocabulary

- implementation
- `override`
- state

Related Resources

- Developer Documentation: Displaying and Managing Views with a View Controller
- API Reference: UIViewController

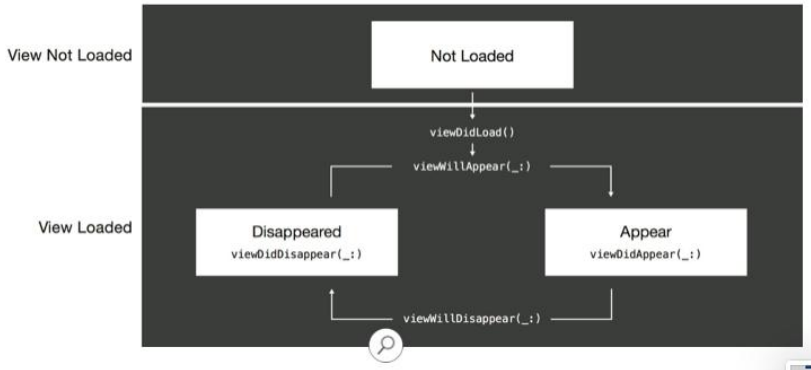
When you've finished setting up views and view hierarchy, you'll need to move on to writing the logic behind your app. Your view controller classes are responsible for displaying your user's data and handling user interactions. In the Light project, you used a `UIViewController` subclass to manage the button tapping events through actions, and updated the view's background color accordingly. A view controller also controls the creation of its views, handles events based on the state of the view as it progresses through its life cycle, and disposes of its views when they're no longer needed.

View Controller Life Cycle

In iOS, view controllers can be found in one of several different states:

- View not loaded
- View appearing
- View appeared
- View disappearing
- View disappeared

As the view transitions from one state to another, iOS calls SDK-defined methods, which you can implement in your code. In the figure below, you can see the name of each method that gets called when a state transition occurs.



You probably noticed a pattern in the method names. After the view is loaded, the methods come in pairs: "will" and "did." (Compare the names of the `willSet` and `didSet` property observers for variables.) This standard Apple design pattern allows you to write code before and after the named event occurs. However, it's important to note that it's possible to have a "will" callback without the corresponding "did" callback. For example, `viewWillDisappear(_:)` could be called without `viewDidDisappear(_:)` ever happening.

While every app is different, you'll learn the guidelines for each of the view controller life cycle methods and the specific types of tasks associated with them.

To gain tangible experience with the view controller life cycle, you'll build an app that prints messages to the console describing each life cycle method as it is called. As you learn more about different life cycle methods, you'll add to the project.

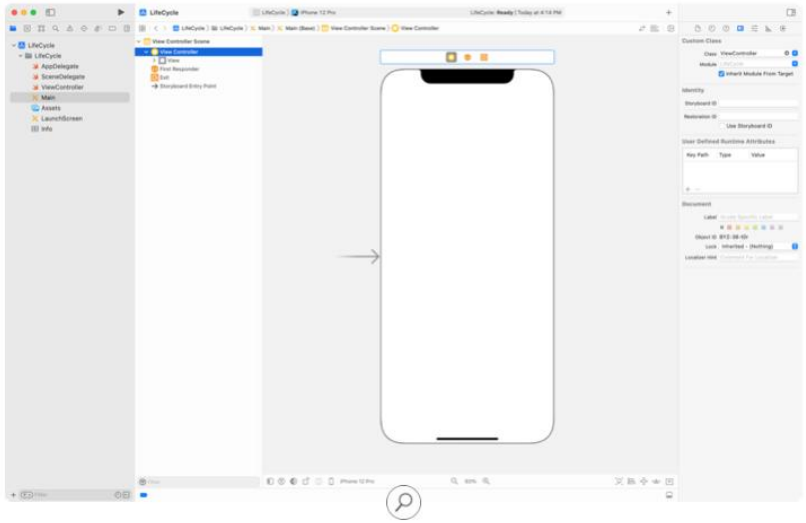
In Xcode, create a new project using the iOS App template. Name the project "LifeCycle."

View Did Load

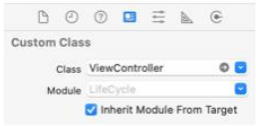
After you've instantiated a view controller, whether from a storyboard or programmatically, the view controller will load the view into memory. This process creates the views that the controller will manage.

After a view controller has finished loading its particular views, its aptly named `viewDidLoad()` function is called, giving the controller a chance to perform work that depends on the view being loaded and ready. For example, in Light, `updateUI()` was called to sync the background color of the view with the `lightOn` state. Other types of setup tasks to perform in `viewDidLoad()` include additional initialization of views, network requests, and database access.

To add custom implementations of the view controller life cycle methods, you'll need to know the class associated with the view controller scene in question. In LifeCycle, open the `Main` storyboard to reveal its view controller. Select the view controller and use the Identity inspector to reveal its subclass.



In the Class field, you'll see a white arrow inside a gray circle—which provides you with a shortcut to the class. Click the arrow to open the associated file, `ViewController`.

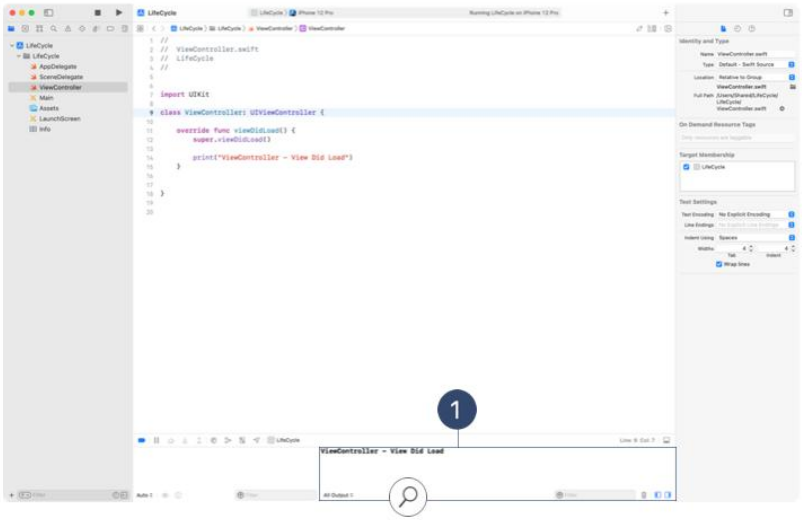


Add the following to the end of the `viewDidLoad()` function:

```
print("ViewController - View Did Load")
```

This line will print to the console the string "ViewController - View Did Load" any time `viewDidLoad()` is called.

Build and run your app. In the console pane, you should see your message after the view loads. [1]



View Event Management

Some units of work may only be performed one time—for example, updating a label's font text, or color. For those tasks, `viewDidLoad()` is the most appropriate place to do it.

For work that will be performed multiple times, your app can rely on view event notifications. When the visibility of its views changes, a view controller will automatically call its life cycle methods—allowing you to respond to the change in view state.

These methods include:

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

Take a look at the documentation. You may notice that each of these methods requires you to call the superclass's version at some point in your implementation. One way to understand why this needs to happen is to think of the following: when you write your custom view controller subclass and want to use life cycle methods (including `viewDidLoad()`), you will get an error if you do not use the `override` keyword. This is because the superclass already contains definitions for these methods. Using the `override` keyword overrides the implementation in the superclass, `UIViewController` in this case, and allows your implementation of this method to be executed instead of the superclass's implementation.

However, since you don't know the implementation of `UIViewController`'s life cycle methods, you could be creating unexpected issues by not letting the superclass's code run. `UIViewController` could be doing important work that your app needs to function properly. To fix this, you can explicitly call the superclass's version of the method using the `super` keyword. Now both your code and `UIViewController`'s code will run.

Generally, the call to the superclass's implementation will be the first line of your overridden method.

Example:

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

View Will Appear and View Did Appear

After `viewDidLoad()`, the next method in the view controller life cycle is `viewWillAppear(_:)`. This is called right before the view appears on the screen. This is an excellent place to add work that needs to be performed before the view is displayed (and every time it's displayed) to the user. For example, if your view displays information relative to the user's location, you may want to request the location in `viewWillAppear(_:)`. That way, the view can be updated to take advantage of the new location. Other tasks include: starting network requests, refreshing or updating views (such as the status bar, navigation bar, or table views), and adjusting to new screen orientations.

As you'd expect, `viewDidAppear(_:)` is called after the view appears on the screen. If your work needs to be performed each time the view appears—but may require more than a couple of seconds—you'll want to place it in `viewDidAppear(_:)`. This way, your view will display quickly as your function continues to execute.

Use the `viewDidAppear(_:)` method for starting an animation or for other long-running code, such as fetching data.

To continue exploring the life cycle, override the function `viewWillAppear(_:)`.

```
override func viewWillAppear(_ animated: Bool) {

}
```

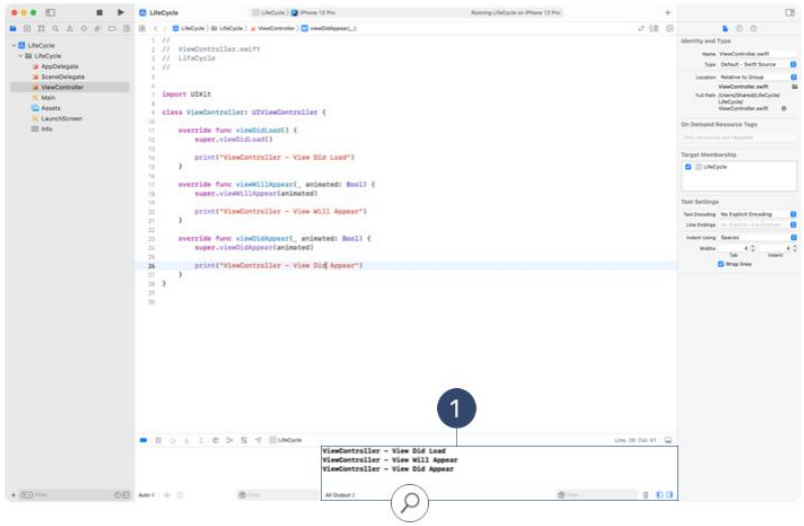
Because you're writing your own custom implementation of `viewWillAppear(_:)`, remember to call the superclass version of `viewWillAppear(_:)`. Because the call to `viewWillAppear(_:)` requires the `animated` property, you can pass along the parameter given to the subclass.

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)
}
```

Next, add a print statement so you can check the order of the life cycle methods:

```
print("ViewController – View Will Appear")
```

Now also add a similar print statement to `viewDidAppear(_:)`.



Build and run your app. You should see three statements printed in your console. [1] Refer to the view controller state diagram above, and see if you can trace the view's transitions.

View Will Disappear and View Did Disappear

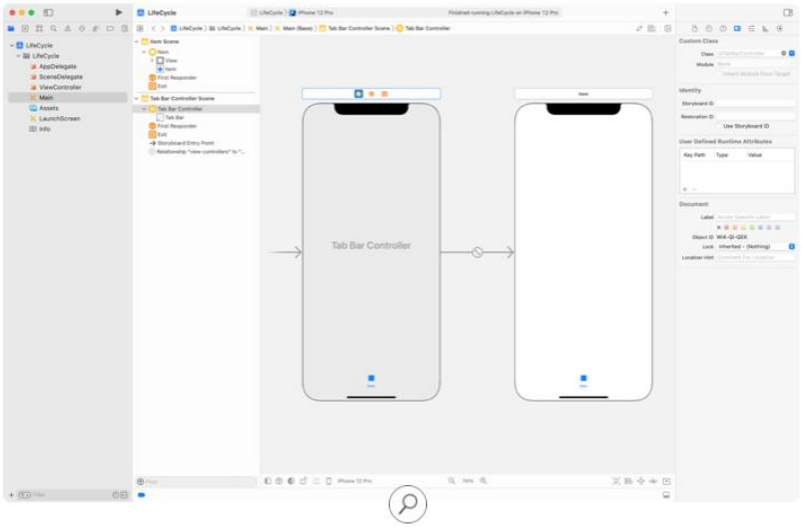
You've probably already guessed that `viewWillDisappear(_:)` is called before the view disappears from the screen. This method executes when the user navigates away from the screen by tapping the back button, switching tabs, or presenting or dismissing a modal screen. You can use the `viewWillDisappear(_:)` method for saving edits, hiding the keyboard, or canceling network requests.

The last method in the life cycle is `viewDidDisappear(_:)`, which is called after the view disappears from the screen—typically after the user has navigated to a new view. If this method executes, it is certain the view has disappeared. As such, this method gives you an opportunity to stop services related to the view, for example, playing audio or removing notification observers.

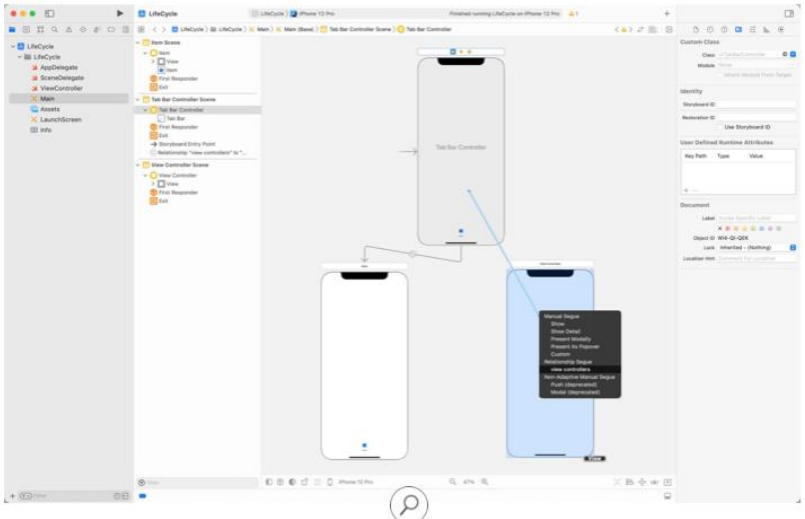
Now take a moment to return to the LifeCycle project. In your `ViewController` class, add and override the "will" and "did" functions for the disappear view event. Add print statements so you can see the life cycle events.

For your view to disappear, you'll need to add a second view controller. Otherwise, there'd be no way to navigate *from* the current view. But first, you'll add a tab bar controller to handle navigation.

Navigation and Workflows

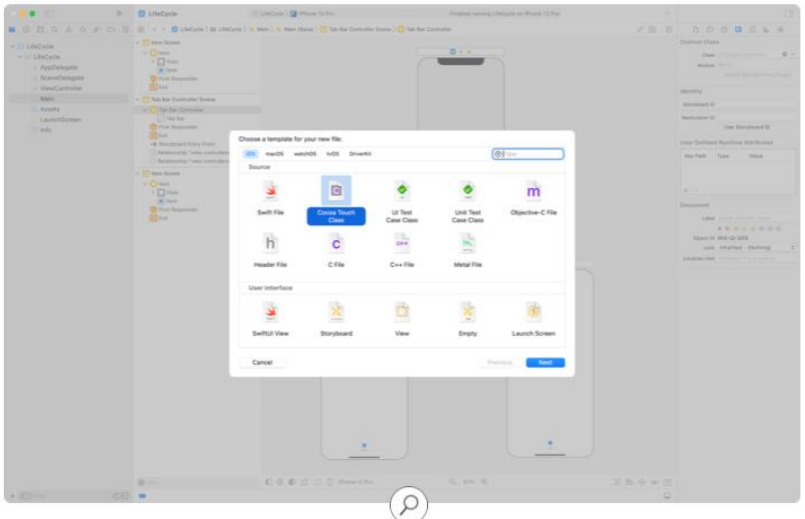


Open the `Main` storyboard. Select the `ViewController` scene and embed it in a tab bar controller.

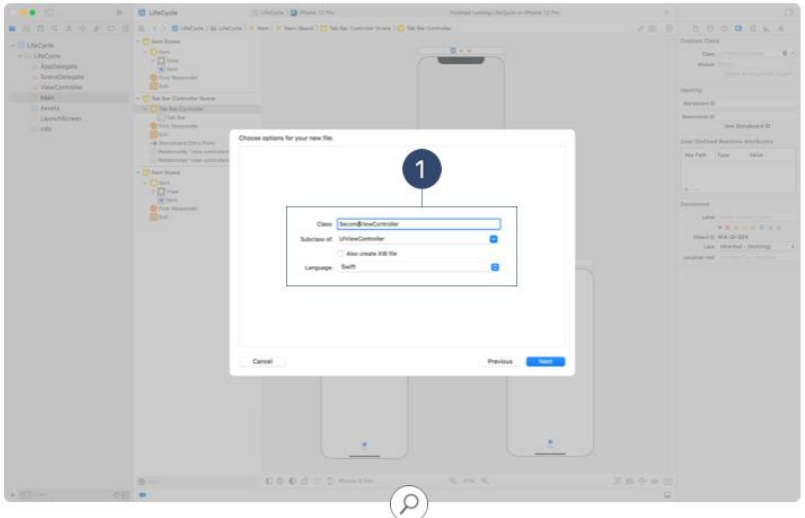


Next, drag a view controller from the Object library, and add it as a second tab in the tab bar controller. Feel free to organize the storyboard and update the background color of the view controllers.

Navigation and Workflows

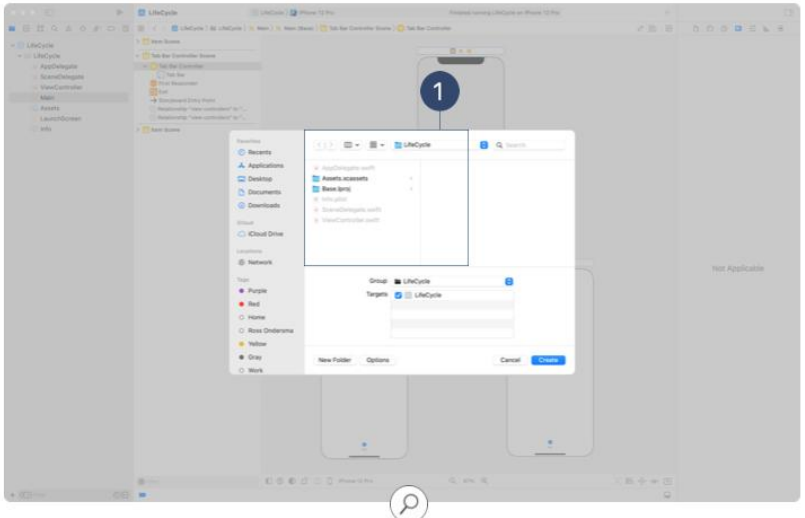


To write custom code for the new view controller, you'll have to add a new file. From the Xcode menu bar, choose **File > New > File** (or **Command-N**). Then select Cocoa Touch Class and click Next.

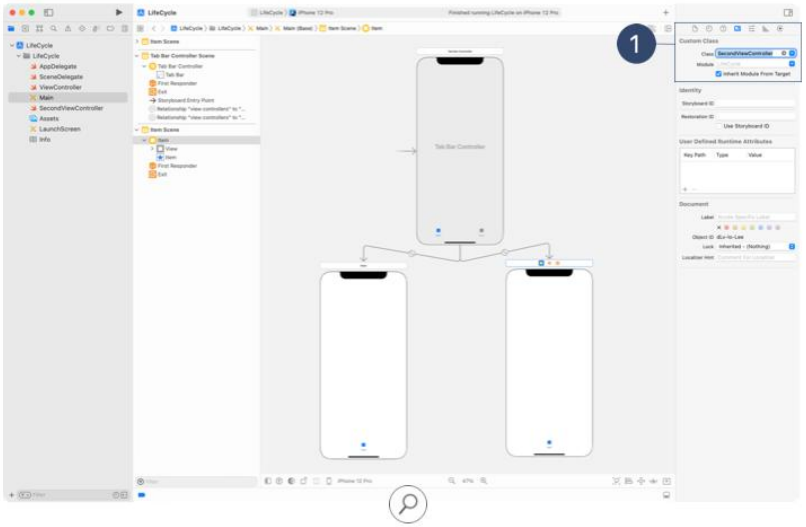


Name the new class “SecondViewController” and make sure the Subclass field is set to `UIViewController`.[1]

Navigation and Workflows



Add the new file to the LifeCycle folder.[1]



Next, you'll need to tell the storyboard the identity of your new view controller. Go back to the `Main` storyboard and select the second view controller. Open the Identity inspector and update the Class to `SecondViewController`. Make sure to press Return to confirm this change.[1]

Navigation and Workflows

To see how iOS transitions between the view controllers, you'll use the life cycle methods of `SecondViewController` as well. Go back to your `SecondViewController` class file, and add the same five life cycle methods that you have in `ViewController`. Don't forget to call the `super` implementation. Add print statements to each event. Consider something like this: "SecondViewController - View Did Load."

Build and run your app. Without clicking anything, you should see that the `ViewController` instance has loaded and both appear methods have been called.

But why don't you see any prints from `SecondViewController`? The storyboard initialized the `SecondViewController`, so what's happening? Why hasn't the view loaded? The view controller won't load its view until the view needs to be displayed to the user. And that's a good thing. By delaying the loading, the app is conserving memory —which is a limited resource in a mobile environment.

To load the `SecondViewController` instance, return to Simulator and click the tab for the second view controller. At this point, you should see the following print statements in this order:

```
ViewController – View Did Load
ViewController – View Will Appear
ViewController – View Did Appear
SecondViewController – View Did Load
SecondViewController – View Will Appear
ViewController – View Will Disappear
ViewController – View Did Disappear
SecondViewController – View Did Appear
```

Switching back to the first tab, the order of the print statements is:

```
ViewController – View Will Appear
SecondViewController – View Will Disappear
SecondViewController – View Did Disappear
ViewController – View Did Appear
```

The order in which these functions are called helps to explain how view controller's views are added and removed from the view hierarchy:

- If a view controller's view is to be added to the hierarchy, `UIKit` first ensures that the view has been loaded. If not, it loads the view and triggers `viewDidLoad()`.
- Before adding the view controller's view to the hierarchy, `UIKit` triggers `viewWillAppear(_:)`.
- View controller views that will no longer be displayed are then removed, and those view controller's `viewWillDisappear(_:)` and `viewDidDisappear(_:)` methods are called.
- Finally, `UIKit` displays the new view and triggers `viewDidAppear(_:)`.

As you can see, there are many uses for the view controller life cycle methods. Each one is like a "notification" telling your code that the view event has taken place. Using this guide, you, as the developer, will figure out how best to take advantage of each of the methods for the particular task at hand.

Challenge

Draw the state diagram from memory.

Lab—Order Of Events

Objective

The objective of this lab is to further your understanding of the view's life cycle. You will create an app that adds to the text of a label based on the events in the view controller life cycle.

Create a new project called "OrderOfEvents" using the iOS App template.

Step 1

Create New View Controller Subclasses

- Drag out two more view controllers from the Object library, and place them next to the existing view controller in the storyboard. The first view controller will simply be a starting point for the app that has a button to segue to the next screen. The second view controller will have a label that displays the order of different view controller life cycle events. The third view controller will provide a way to navigate away from the second view controller.
- Create two new files, one for each of the new view controllers, by choosing **File > New > File** from the Xcode menu bar (or press **Command-N**), then choosing Cocoa Touch Class. Name the files "MiddleViewController" and "LastViewController." Check that they're both subclasses of `UIViewController`.
- One at a time, select the middle and last view controllers in the storyboard and link them to the files you just created, using the Identity inspector to set their class to `MiddleViewController` and `LastViewController`.
- Note that the original view controller in your storyboard (the one provided by the project template) already has a corresponding `ViewController` file and its class is properly assigned in the Identity inspector.

Step 2
Set Up Your Storyboard

- Embed the first view controller in a navigation controller.
- On this view controller, add a button that says: "Show me the life cycle." Create a Show segue from the button to the middle view controller.
- Add a button to the bottom of the middle view controller. Create a Show segue from this button to the last view controller. Since no information is being passed between view controllers, you don't need to worry about setting the segue's identifier.
- Add a label to the top of the middle view controller. Replace its text with: "Nothing has happened yet." Since this label will need more lines later on, use the Attributes inspector to set Lines to 0. This attribute will allow the label to expand as needed.
- Add a label to the last view controller, and replace its text with: "Go back and see if anything happened."

Step 3
Update The Label Based On The Life Cycle Event

- In the storyboard, create an outlet from the label in the middle view controller to `MiddleViewController`.
- Add a variable property of type `Int` just below the outlet for your label. Call it "eventNumber" and set it equal to `1`. At the end of each life cycle event, your code will add 1 to this property—numbering events as they're added to the label.
- In `MiddleViewController`, add a method to update your label for a given event. Use conditional binding to unwrap the existing text in the label. Set the label text equal to what was already there, plus a statement about the life-cycle event that just occurred, then update `eventNumber`. This statement should use `eventNumber` to keep track of the order of events. Your code might look as follows:

```
func addEvent(from: String) {
    if let existingText = label.text {
        label.text = "\(existingText)\nEvent number \
(eventNumber) was \(from)"
        eventNumber += 1
    }
}
```


- What's happening? The above code unwraps the text in the label—which you need to do because the value is optional. Next, it adds a newline (`\n`) to the label text (starting a new line), a description of the event that just occurred, and its event number. The code increments `eventNumber` by one, so that the next time `eventNumber` is accessed, it will describe the next event number in the sequence.
- In `MiddleViewController`, implement all five life-cycle methods you learned in this lesson: `viewDidLoad()`, `viewWillAppear(_:)`, `viewDidAppear(_:)`, `viewWillDisappear(_:)`, and `viewDidDisappear(_:)`.

```
override func viewDidLoad() {
    super.viewDidLoad()
    addEvent(from: "viewDidLoad")
}
```

- When you've finished filling out the body of all five life cycle methods, run the app and navigate from screen to screen. What happens when you navigate to the last view controller and back to the middle view controller? Do all the life cycle events happen again? If not, why not? What about when you navigate all the way back to the initial view controller and then to the middle view controller? Do all the life cycle events happen again? Why is this different from navigating from the last to the middle view controller?
- Great job! You've made an app that displays the order of view controller life cycle events as they happen. Be sure to save it to your project folder.

Review Questions

Question 1 of 5

Of the life cycle methods in this lesson, which one will execute first?

- **A.** `viewWillAppear(_:)`
- **B.** `viewDidAppear(_:)`
- **C.** `viewDidLoad()`
- **D.** `viewDidDisappear(_:)`
- **E.** `viewWillDisappear(_:)`

Check Answer

Lesson 3.10

Building Simple Workflows

Apps are more than a collection of views and controls. A great app is beautiful, approachable, engaging, powerful, and simple to use.

So far in this unit, you learned about many of the `UIKit` tools that help you build common user interfaces and respond to user interactions. In this lesson, you'll tie those concepts together to design simple workflows and familiar navigation hierarchies.

What You'll Learn

- How to break down a feature into an intuitive workflow
- How to design a navigation hierarchy for an app
- Where to learn more about interface conventions

Vocabulary

- **navigation hierarchy**
- **workflow**

Related Resources

- iOS Human Interface Guidelines

You've learned that `UIKit` is the foundation for building iOS apps. Its conventions drive how iOS users interact with their devices, the operating system, and most of the apps they use every day. And it's because of these conventions that many iOS apps today feel familiar to long-time users.

The iOS Human Interface Guidelines resource defines some of the best practices for building intuitive workflows and familiar navigation hierarchies. As an app developer, you have the opportunity to put these practices to excellent use.

Design Principles

Equipped with a wide variety of views and controls in your toolset, you're already able to build many features into your apps. But to build an app that's both simple and powerful requires mastery of the Human Interface Guidelines—an understanding of when, where, and how to use UI objects in a familiar way.

The first page of the Human Interface Guidelines includes a list of six characteristics to keep in mind as you design your apps.

Aesthetic Integrity

Your app's appearance and behavior should make sense for its goals and purpose. For example, an app that helps people perform a serious task can keep them focused by using subtle graphics, standard controls, and predictable interactions. On the other end of the spectrum, an entertaining app, such as a game, can sport a captivating appearance that promises fun and excitement, while encouraging discovery.

Consistency

A consistent app incorporates features and behaviors in ways that people expect. Whenever possible, use system-provided interface elements, well-known icons, standard text styles, and uniform terminology—all available in `UIKit`—to deliver a familiar experience.

As you design specific features, explore how other apps have solved similar problems. For example, if you're designing a workflow for completing an order in an e-commerce app, take a look at the checkout flow of popular e-commerce apps. You shouldn't blatantly copy other apps, but you can use good examples as a starting point for your design.

Direct Manipulation

Through direct manipulation of onscreen content, users can see the immediate, visible results of their actions—which both adds engagement and facilitates understanding. When your app presents content that users can change or adjust, consider providing a way to manipulate the content with a gesture or by rotating the device. Many photo editing apps, for example, use swipe gestures to adjust settings or apply effects.

Feedback

Users should never wonder if an app responded to their actions. Make sure your app provides perceptible feedback—in the form of alerts, animations, or other confirmations—to let users know what's going on. If your app is loading data from the internet, display the network activity indicator. If the action triggered by a button isn't available, dim the button.

For some good examples, take a look at the built-in iOS apps. In what ways do they acknowledge user actions? You might notice that interactive elements are highlighted briefly when tapped, progress indicators communicate the status of long-running operations, and animation and sound help clarify the results of actions.

Metaphors

Users catch on faster when an app's objects and actions are metaphors for familiar experiences—whether rooted in the real or digital world. Have you ever wondered why the home screen on a computer is called the Desktop? Early versions of graphical interfaces used the Desktop metaphor to help users feel at home with this new computing paradigm. Folders represented directories, documents looked like paper, and the trash can was for deleting files.

Metaphors work even better in iOS because people interact physically with things on the screen. iOS interfaces use distinct visual layers and realistic motion to convey hierarchy and depth. Users can move views out of the way to discover information. They drag and swipe content. They toggle switches, move sliders, and scroll through picker values. They even flick through pages of books and magazines.

User Control

No matter how awesome an app is, it doesn't take control. It might suggest a course of action or warn about dangerous consequences, but the user should always get to make the important decisions. As you design your apps, try to give users decision-making power without bombarding them with alerts or options at every juncture. An app can make people feel in control by keeping interactive elements familiar and predictable, confirming destructive actions, and making it easy to cancel operations—even when they're already underway. The best apps find the correct balance between enabling users and avoiding unwanted outcomes.

Human Interface Guidelines

The Human Interface Guidelines document is your best resource for planning and designing apps. Take a moment to read through a handful of sections to get a feel for the type of information it covers.

Here are a few great sections to check out:

- Data Entry details best practices for longer input screens or forms for collecting information from the user.
- Color talks about using color to help people understand how to interact with interface elements.
- Tables gives specific guidelines for presenting information in a table view.

As a new developer, you'll be more successful if you follow everything in the Human Interface Guidelines. But as you gain experience, you may recognize use cases that call for something outside the box. There are many great apps that extend beyond iOS conventions. As the saying goes, learn the rules like a pro, so you can break them like an artist.

Navigation and Workflows

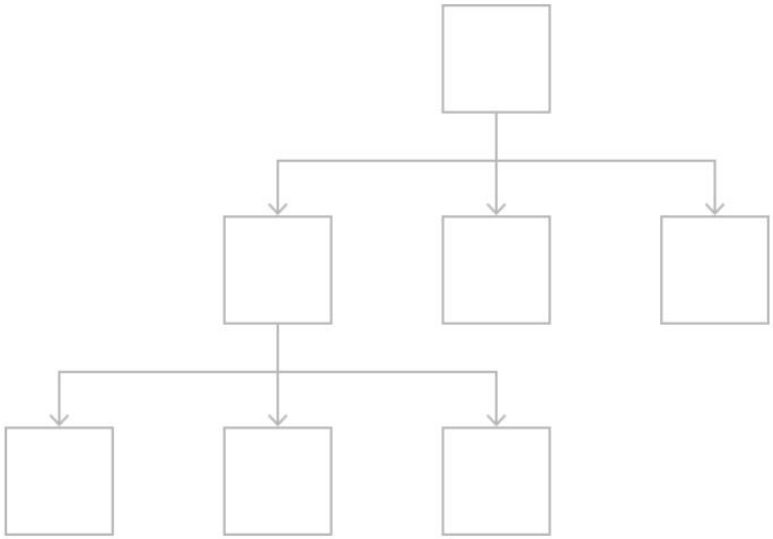
Navigation Hierarchy

Your job as a developer is to implement navigation in a way that supports the purpose of your app without distracting the user. Navigation should feel natural and familiar, and shouldn't dominate the interface or draw focus away from the content.

In iOS, there are three main styles of navigation: hierarchical, flat, and content-driven or experience-driven.

Hierarchical Navigation

In this style, the user makes one choice per screen until reaching a destination. To navigate to another destination, they must retrace their steps or start over from the beginning and make different choices. Settings and Mail use a hierarchical navigation style.



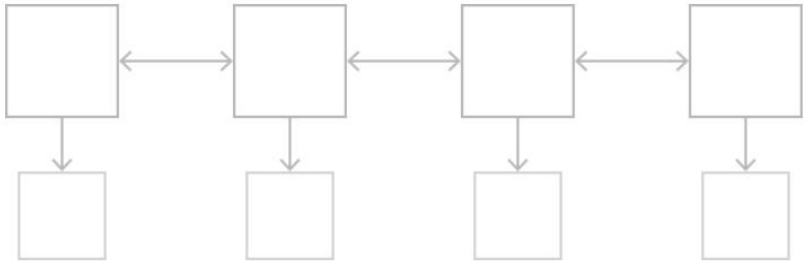
For this style, you'll typically use navigation controllers to enable navigation through a series of hierarchical screens. When a new screen is displayed, a Back button on the leading end of the navigation bar allows an easy return. (To give the user context, the bar may display the title of the current screen.)

Consider the Settings app. Users open Settings to view or update preferences, which are grouped into categories. These are hierarchical relationships: Each category contains subcategories or individual settings that the user can change.

The interface is designed using the same hierarchy. When you open the Settings app, you're presented with a list of categories. Choose a category and you see a list of subcategories or individual preferences. When you want to return to a parent list higher in the hierarchy, you can use the Back button to return to the previous view.

Flat Navigation

In a flat navigation design, users switch between multiple content categories. Music and the App Store use this approach.



For flat navigation, you'll typically use a tab bar controller to organize information at the app level. A tab bar is a good way to provide access to several workflows or distinct modes of operation. You might also use a page controller.

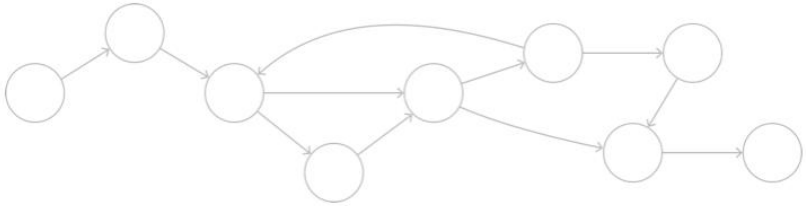
Consider the App Store, which displays five categories of tasks, each presented by a tab: curated Today view, browse games, browse apps, Apple Arcade, or search.

Some apps combine multiple navigation styles. For example, the App Store uses a tab bar controller to separate tasks in a flat style but relies on hierarchical navigation within each tab.

Navigation and Workflows

Content-Driven, or Experience-Driven, Navigation

In this style, the user moves freely through content, or the content itself may define the navigation. Games, books, and other immersive apps generally use this type of navigation.



In this course, you won't be building apps that rely on content-driven navigation.

Navigation Design Guidelines

It's important that users always know where they are in your app and how to get to their next destination. Regardless of navigation style, it's essential that the path through content is logical, predictable, and easy to follow. A general guideline is to give people only one path, or navigation style, on each screen. If they need to see a screen in multiple contexts, or places in your app, consider using a modal view to complete the task.

As you think about the navigation hierarchy of your app, here are a few guidelines to follow:

Design an information structure that makes it fast and easy to get to content. Organize your information in a way that requires a minimum number of taps, swipes, and screens.

Use standard navigation components. Whenever possible, use standard navigation controls, such as tab bars, segmented controls, table views, collection views, and split views. Users are already familiar with these controls and will intuitively know how to get around in your app.

Use a navigation bar to traverse a hierarchy of data. The navigation bar's title can display the user's current position in the hierarchy, and the Back button makes it easy to return to the previous position.

Use a tab bar to present peer categories of content or functionality. A tab bar lets people quickly and easily switch between categories or modes of operation, regardless of their current location.

Use the proper transition style. Whenever you're transitioning to a new screen to display more detail about an item, you should use a right-to-left push transition used by a Show segue within a navigation controller. If the user is switching contexts—from displaying contacts to adding a new contact, for example—use a modal transition.

Example Workflow

If you were to build an alarm and stopwatch app, it would be useful to plan out the features and workflow of the app before sitting down to build it. A good way to do this is with a simple flowchart that details the purpose of each screen, the mode of navigating from one screen to another, and the general navigation hierarchy of the app.

For this example, assume the app needs the following features:

- Display alarms
- Toggle alarms on and off
- Create alarms
- Edit alarms
- Change the sound of alarms
- Basic stopwatch functionality (display, start, lap, stop, reset)

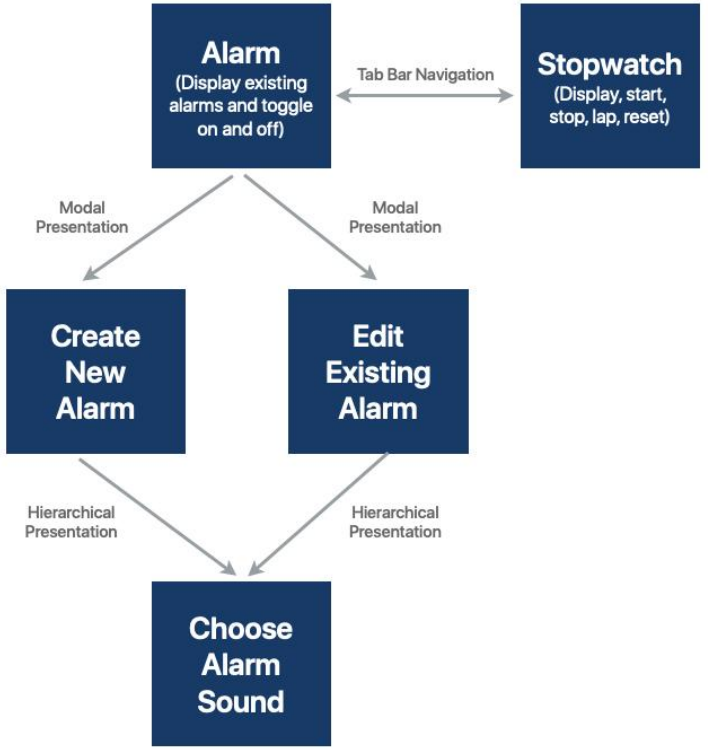
Looking at the planned features, you can see that there are two separate modes in the app: alarm and stopwatch. A tab bar controller would provide a flat navigation design that would allow the user to clearly see both distinct features.

Because the alarm and stopwatch tabs are separate, each can focus on the workflows necessary to run its own feature. The stopwatch functionality can easily exist on a single view. So, the stopwatch tab doesn't need any more than its own tab.

Managing alarms may require more than one screen. For example, it may be useful to have additional screens to view, create, and edit different alarms.

Navigation and Workflows

For this, you can build a view to list all the alarms and present different modal views when the user selects a preexisting alarm or needs to create a new one. The flowchart below illustrates the navigation hierarchy and overall workflow of this hypothetical app.



Lab—Pizza Delivery

Objective

The objective of this lab is to practice designing simple and intuitive workflows and navigation hierarchies. You'll use what you've learned about design principles to plan out the navigation hierarchy and flow for a hypothetical pizza delivery app.

Step 1

Place Your Order

- Take a look at a few popular e-commerce apps to get a good idea of purchase and checkout processes.
- Whether in Keynote, in Pages, or on paper, create a flowchart that outlines the navigation hierarchy for your pizza delivery app. The ordering process should allow the user to view the menu, order a pizza, create a custom pizza, and proceed to their cart.

Step 2

Check Out

- In addition to thinking through the ordering process, you'll need to think about the checkout process. Again, look at some of your favorite e-commerce websites or apps to see how they handle checking out. You don't have to actually buy anything—just go far enough that you get a good sense of navigation hierarchy and workflow.
- Once a customer has finished selecting or creating their pizza in your pizza delivery app, what will they do next? Design the navigation hierarchy for checking out, and add it to your flowchart.
- Determine how the app will flow from ordering the pizza to the checkout process. Can a customer leave the ordering navigation hierarchy and still view their cart? Is the cart accessible only after going through the ordering process? Design what you think is best for the user—and for a successful pizza business.

Step 3

Other Useful Features

- Are there other steps you think might be essential to a pizza delivery app? If so, think of the simplest way to include them and add them to your flowchart.

Congratulations! This sort of practice will help you create user-friendly workflows as you start designing your own apps. If you created your flowchart on paper, take a picture of it so you can save it to your project folder. If you created it on the computer, be sure to save the file.

Review Questions

Question 1 of 4

Which navigation style does the Mail app use?

- **A.** Hierarchical
- **B.** Flat
- **C.** Content-driven

Check Answer

Guided Project: Personality Quiz

In this unit, you learned about the mechanisms provided by the `UIKit` framework for managing the flow of your app. Previously, you learned how to manage the position and size of views and controls using Auto Layout and stack views. Now you'll combine that knowledge to build an app.

For this guided project, you'll create a personality quiz. Maybe you've seen this type of game before. Players are presented with a lighthearted topic and answer questions that align them to a particular outcome. Here are some examples of personality quiz topics:

- What animal are you?
- What country should you visit next?
- Which Apple product are you?

There are no correct answers to quiz questions. Answers are purely subjective, and their results don't even have to be logical. For example, suppose you design a quiz called "What country should you visit next?" You could pose the question "What is your favorite color?" and decide that the answer "green" aligns to Italy and not to France. Other questions and answers might make more sense. If the player prefers sushi over pasta, you could award points for Japan instead of for Italy.

This guided project will use "Which animal are you?" as the quiz topic. The four possible outcomes are: dog, cat, rabbit, and turtle. But if you prefer to choose your own topic and outcomes, go ahead. As long as you're following the steps in the project, any topic is fine. You'll learn more if you're having a good time.

Part One
Project Planning

Rather than diving directly into Interface Builder or Swift, it's important to think about the goals and requirements of your personality quiz. Who's your target audience? Maybe you have a topic in mind, but what features will the quiz include? What models and views will you need to accomplish those features?

If you try to dig into writing code before you've thought through those questions, you'll probably end up rewriting or throwing away a lot of work. Spend some time now to plan your project and consider how best to approach it.

Features

What features are required to produce a fun personality quiz? Since this is probably your first app of this type, keep the list short. At a minimum, the app should accomplish three main goals:

1. Introduce the player to the quiz.
2. Present questions and answers.
3. Display the results.

Workflow

With those three goals in mind, try to imagine your app as a series of related views. Each feature is very distinct from the other two, so each deserves its own screen. Some type of input will move the player from one feature to the next. For example, you can include a "Begin Quiz" button to move the player from the introduction to the questions, and answering the final question can transition to displaying the results. For the purposes of this project, you'll need at least three view controllers—one to present each of the three features.

Did you also think about presenting a new view controller for each question? That's not actually necessary. In earlier lessons, you learned about two methods of presenting view controllers—modally or with a push onto a navigation stack. A modally presented screen typically includes a way to be dismissed, and any new view controller you push on a navigation controller has a Back button in the upper-left corner. In either case, there's always an implied method of dismissal.

Navigation and Workflows

What would happen if you had separate view controllers for each question? Imagine a player is on the ninth question and wants to return to the third question. If you had a view controller for every question, they would have to dismiss six view controllers to go back. That's ok, but then they'd have to answer the questions in between—all over again—to return to the ninth question.

In this project, you'll update a single screen that can present all your questions, rather than presenting questions on separate screens.

Views

Depending on your quiz topic and the questions you want to ask, your personality quiz may need different input controls. Consider the following questions:

- Which food do you like the most?
- Which of the following foods do you like?
- How much do you like this particular food?

The first question is a multiple-choice question, where only one answer is valid. For this question, you could use a button for each food. The second question can have zero or more answers. You could use switches so that the player can select as many foods as they like, as well as a button to submit their choices. The third question might involve a 1-to-10 scale using a slider.

As you can see, the type of question dictates the controls that will be displayed onscreen. With a single view controller for all the questions and answers in your quiz, you'll want to display only the buttons, switches, slider, and/or labels that match the current question. The simplest and best way to do this is to group the controls within a view that corresponds to the question type. The appropriate view—for single-answer, multiple-answer, or ranged response—can then be shown or hidden.

Models

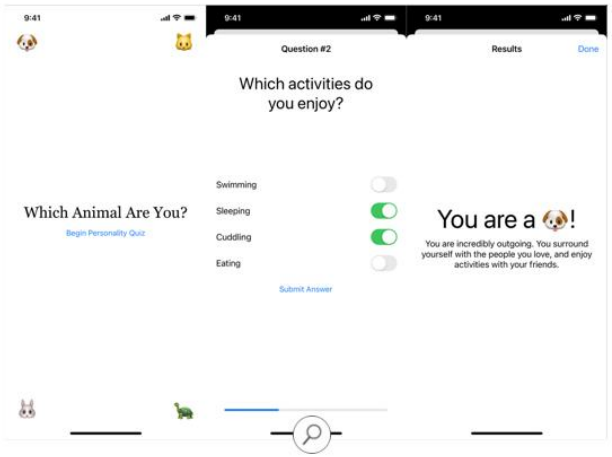
What type of data do you need for a personality quiz? At first, you might consider creating an array of strings to hold your questions, but where would you store the answers? A better idea would be to create a `Question` struct that holds a collection of answers. The collection of answers also needs to be more than an array of strings, because each answer will correspond to one of the quiz outcomes. At the very least, you'll want to include a `Question` struct, an `Answer` struct, and some sort of result type.

Consider the result type. In a personality quiz, an answer can correspond to only one outcome. For example, in the animal quiz, the result is never a dog *and* a cat. It's one or the other. This is the perfect use-case for an enumeration. In the same way that a `Direction` enumeration might have `north`, `south`, `east`, and `west` cases, the `AnimalType` can be `dog`, `cat`, `rabbit`, or `turtle`. To review the details of using an `enum`, visit the Enumerations lesson presented earlier in this unit.

Quick Overview

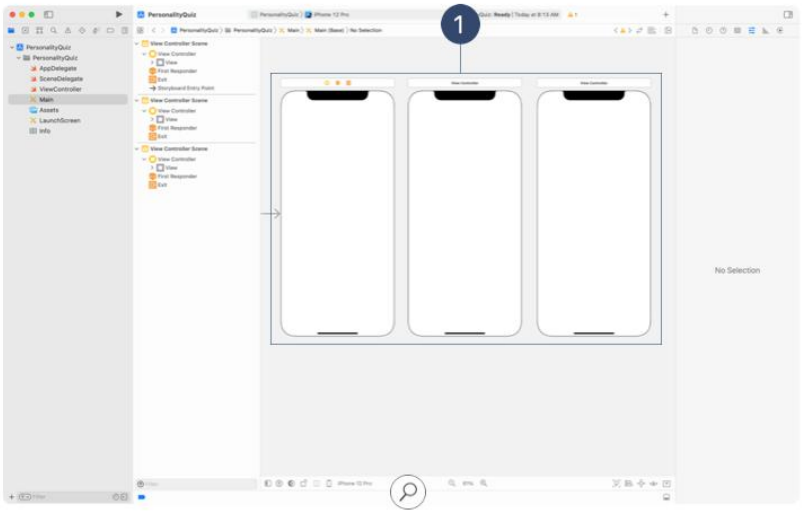
Now that you've analyzed which components you'll need, it's probably easier to see how the project will come together. You'll use three view controllers for your quiz:

- The first is an introduction screen with information about the quiz and a button to begin.
- The second view controller displays a question and several answers, and manages the responses. This view controller is refreshed for each question, and depending on what kind of question you ask, the right controls will be displayed.
- The third view controller tallies up the answers and presents the final outcome. This result can be dismissed, allowing another player to start the quiz from the first view controller.



Part Two
Project Setup

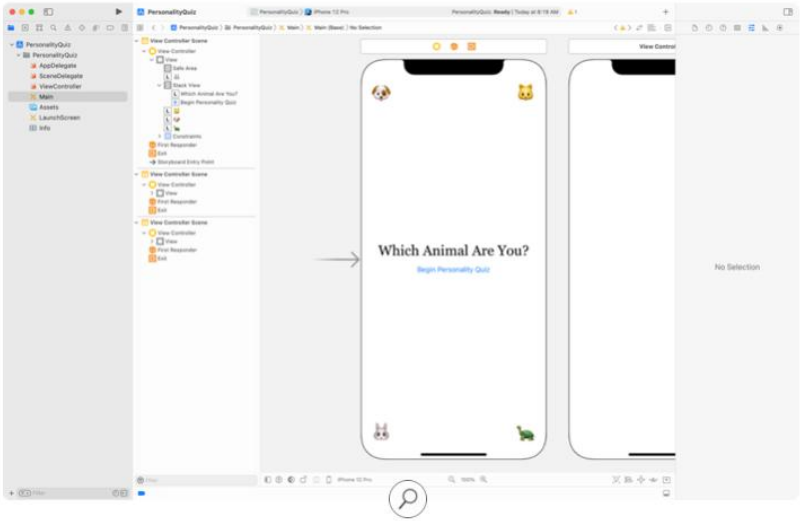
Create a new project using the iOS App template. Name it "PersonalityQuiz," and open the `Main` storyboard. The storyboard already contains one view controller, but you'll need two more. Drag two view controllers from the Object library onto the canvas, and position all three in a horizontal row. [1]



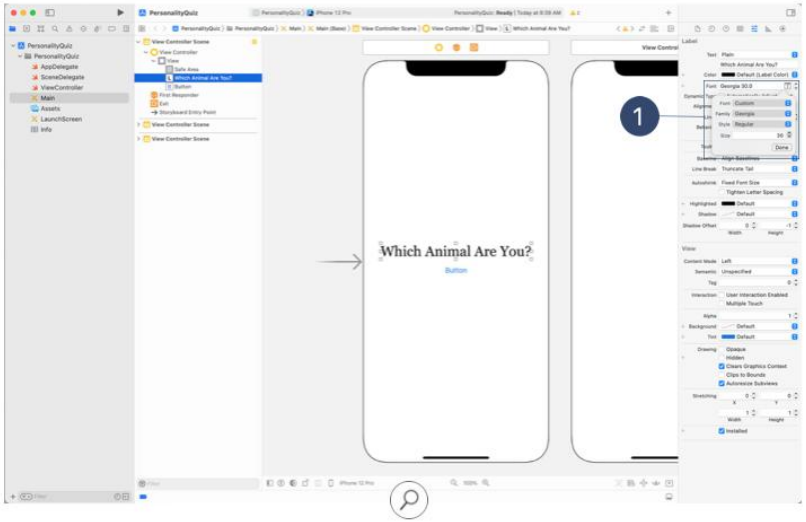
Create the Introduction Screen

The first view controller will invite the player to take your quiz. At a minimum, it needs to include a label that introduces the quiz and a button to begin. Beyond these simple requirements, the design of the screen is entirely up to you.

As an example, take a look at the introduction screen below, then follow the instructions to learn how to build it. You can use the same screen, or something similar, as long as the topic of the quiz is clear to the player.

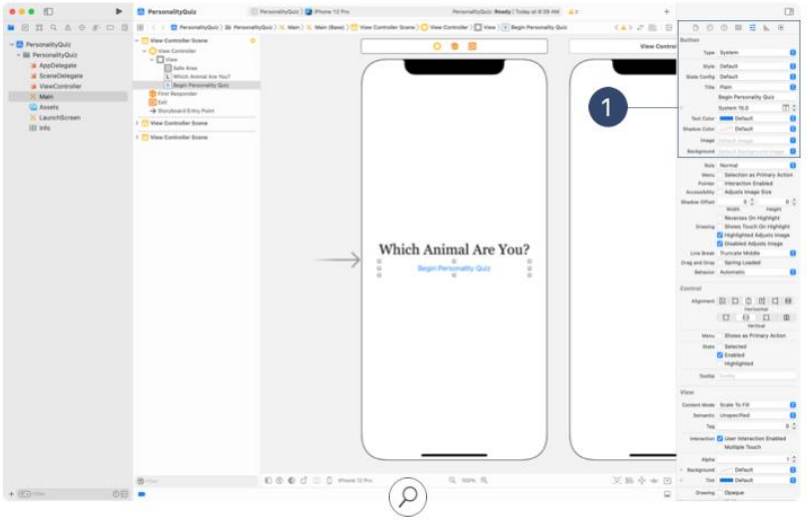


Add a label from the Object library onto the view controller. Then add a button just below the label. With the label selected, use the Attributes inspector and set the alignment for the label to centered. Now change the label's text, text color, and font. In this screenshot, the text reads "Which Animal Are You?" using the Georgia font Regular 30.0. [1]



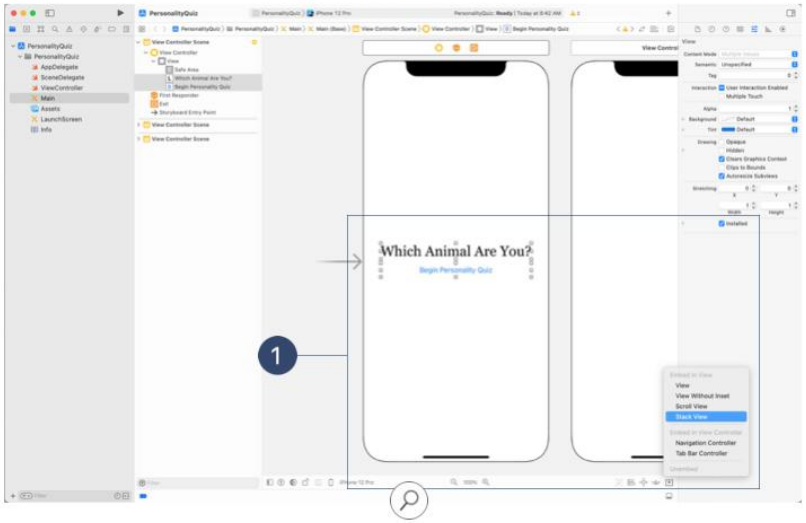
Navigation and Workflows

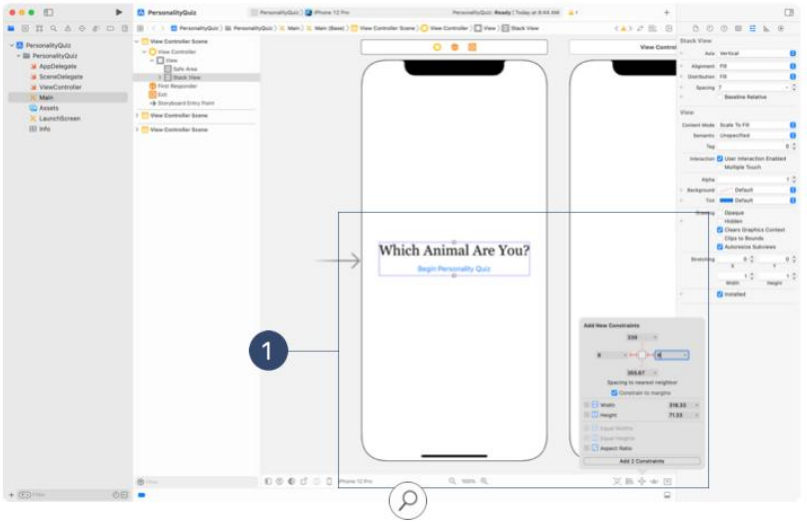
Select the button and update the title to read "Begin Personality Quiz" using the System font 15.0. [1]



Whenever you have multiple items in a horizontal row or a vertical column, it's a good idea to use a stack view. This approach will reduce the number of constraints you'll need to create and manage.

Highlight the label and button, then click the Embed In button and select Stack View. [1]

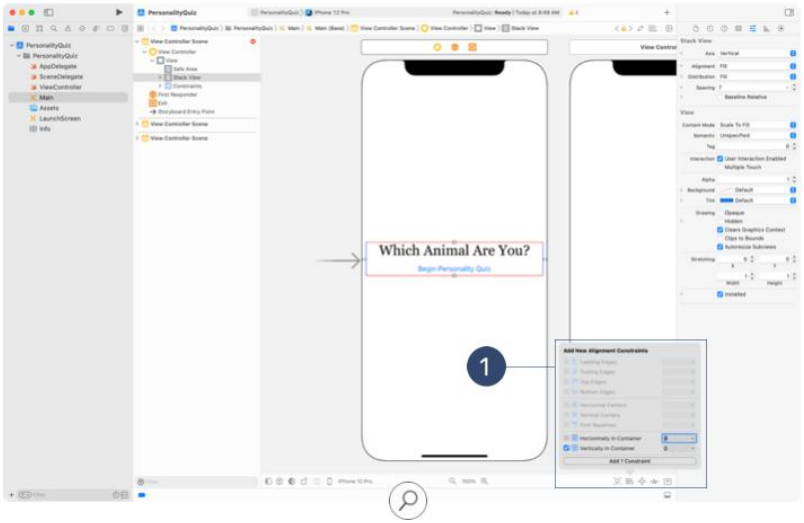




With the stack still selected, use the Attributes inspector to check that the Axis is set to Vertical, and set both the Alignment and Distribution to Fill. These settings ensure that elements in the stack are positioned vertically and that they fill all available space along the stack view's axis.

The label and button should be centered both horizontally and vertically. Although you can create two constraints to do that, the label is too close to the screen's horizontal bounds. Use leading and trailing constraints to ensure that it doesn't run off the screen no matter the device size.

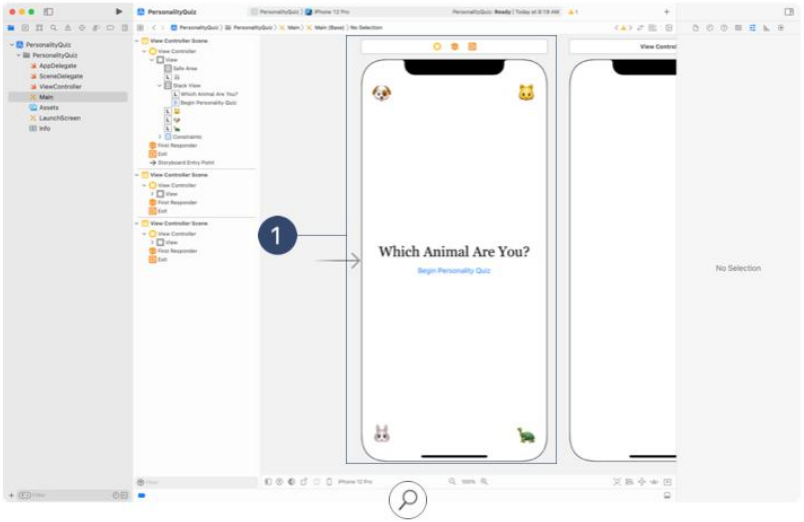
Use the Add Constraint tool to create leading and trailing constraints 8 points from each side. [1]



Next, use the Align tool to add a constraint that centers the stack view vertically. Click "Add 1 Constraint." [1]

Navigation and Workflows

That's all that's necessary for an introduction screen, but it's a little boring. What are the possible outcomes? For this topic, it would be fun to add an emoji for each animal (dog, cat, rabbit, and turtle) and position them in the four corners of the view. [1] If there aren't any emoji that fit your particular quiz topic, consider using images in place of emoji text.

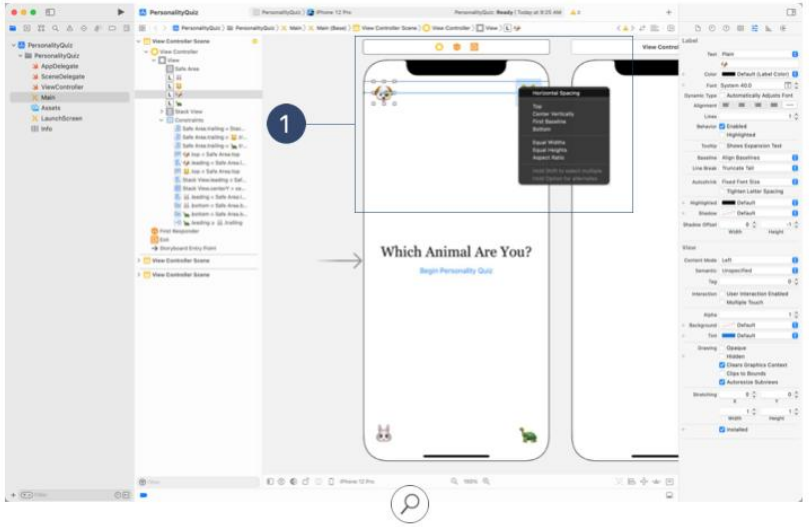


If you haven't already, drag four more labels from the Object library onto the view. Replace the text with the emoji for each animal. To bring up the emoji picker, highlight the label text in the Attributes inspector, then press **Control-Command-Space**. Enlarge all the emoji by setting the Font to System 40.0. Finally, use the blue layout guides to place each label in a corner with the recommended margins.

To hold your emoji in their respective corners on all screen sizes, you'll need to add two constraints to each label. Begin by selecting the top-left label and clicking the Add New Constraints button. Enable the top and leading constraints. If you used the layout guides, the top constraint should have a value of 0 and the leading constraint should have a value of 20. Add these two constraints.

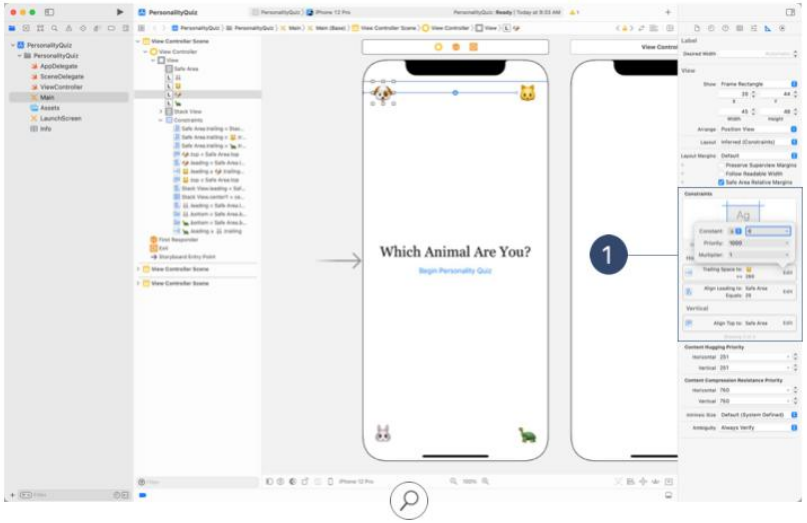
The position of your top-left emoji is all set. Now repeat the steps for the other three labels, using the appropriate edges to create constraints. Check out the four Add New Constraints tool values, from left to right in the following diagram, for the dog, cat, rabbit and turtle labels.

Xcode isn't satisfied with only these constraints. It produces a warning notifying you that trailing and leading constraints are missing, which may cause overlapping with other views. The reason is that labels often have dynamic values or sizes. For this interface design, you're fine leaving it as is. However, it's best practice to satisfy the compiler and resolve all warnings.



The 🐶 and 🐰 need trailing constraints whereas the 🐱 and 🦕 need leading constraints. You can use several options to meet the requirements of your interface's design. The quickest is to add a horizontal space between the 🐶 and 🐱 as well as the 🐰 and 🦕. Set the constraints to ≥ 0 so that they remain in place no matter the screen size.

Control-Drag from the 🐶 to the 🐱 and select Horizontal Spacing. [1]



With the constraint added, navigate to the Size inspector and locate the "Trailing Space to: 🐱" constraint. Click Edit and set Constant to ≥ 0.[1] Repeat the process for the 🐰 and 🦕.

Along the way, after positioning and adding constraints, you might notice some yellow warnings. That's OK. Click the Update Frames button near the bottom of Interface Builder to readjust the position and size of your views to match the constraints you've created.

Create the Question and Answer Screen

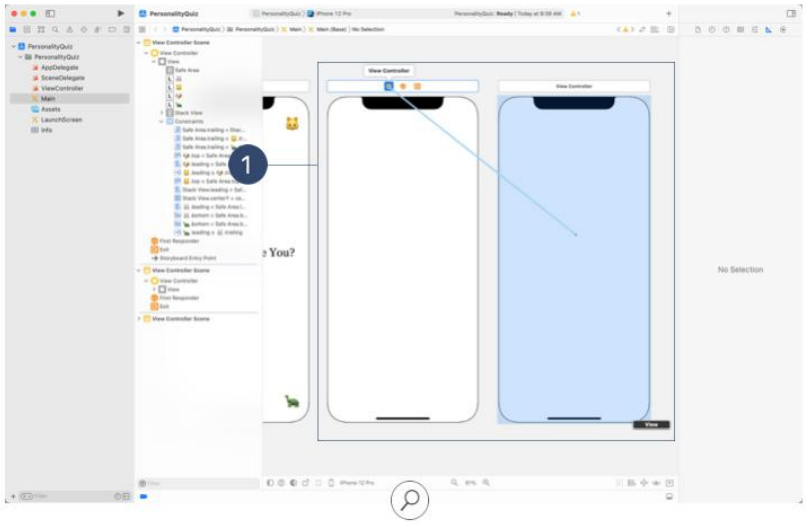
The second view controller will display each question, one at a time, along with input controls that allow the player to respond. The controls you use—buttons, switches, or sliders—need to make sense for the questions and answers in your quiz. You'll think through the controls a bit later in the project.

After the player has answered a question, your app will need to make a decision:

- If there's another question in the quiz, update the labels and controls in the view controller accordingly.
- If there are no more questions, display the results in a new view controller.

How will the app know what to do? You'll need to create some logic that determines whether or not to make a segue after receiving an answer to the current question. If you were to create a segue by **Control-dragging** from the input control to the next view controller, it would force the segue to take place whenever the player interacts with the control.

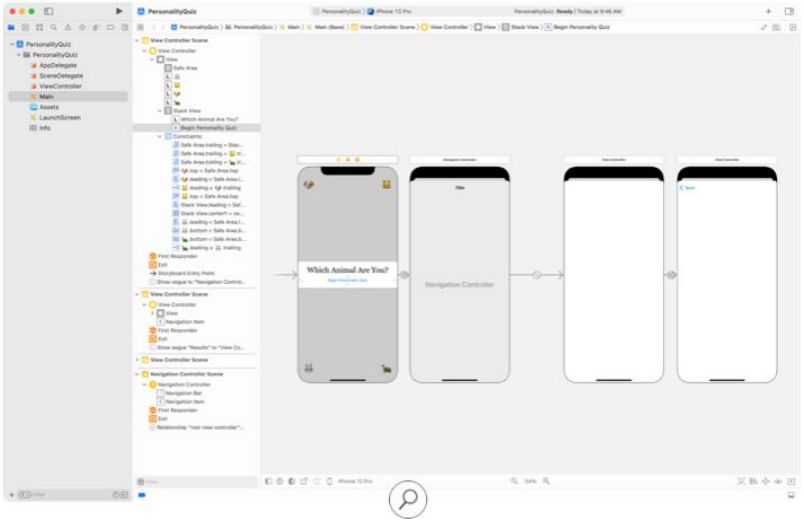
Instead, you can invoke a segue programmatically between the second and third view controllers. **Control-drag** from the view controller icon, above the second view controller's view, to the third view controller,[1] and create a Show segue. Highlight the segue in the storyboard. Then use the Attributes inspector to give it an identifier string, "Results."



As you've already learned, when you embed the root view controller in a navigation controller, the Show segue will adapt from a modal presentation to a right-to-left push. In your quiz, when the player has answered the last question, you can push to the final view controller to display the results.

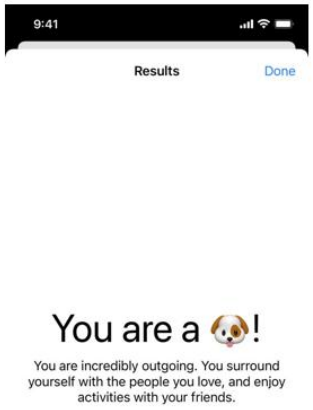
Should all three view controllers be contained in a navigation controller? Remember that a modal presentation is the right choice whenever the *context* of your app will change. And it's a context shift when the player transitions from the introduction screen to the question screen. That means that the first view controller should modally present a view controller contained in a navigation controller. Select the second view controller, then click the Embed In button and choose Navigation Controller.

Create a Show segue from the first view controller's button to the navigation controller. This gives the user the ability to start the quiz.

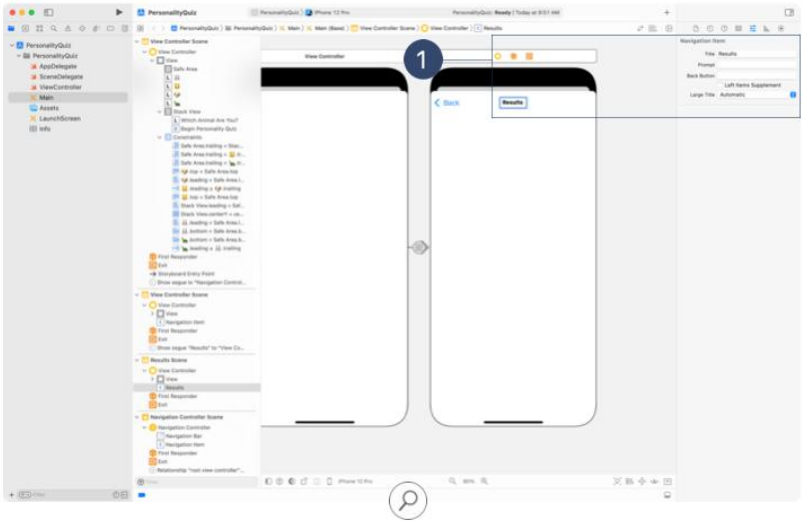


Create the Results Screen

After the player has finished the quiz, the app will display the results, along with a short description. The player should have some way to dismiss the results and return to the introduction screen so that another player can take the quiz. Here's an example of this interface:

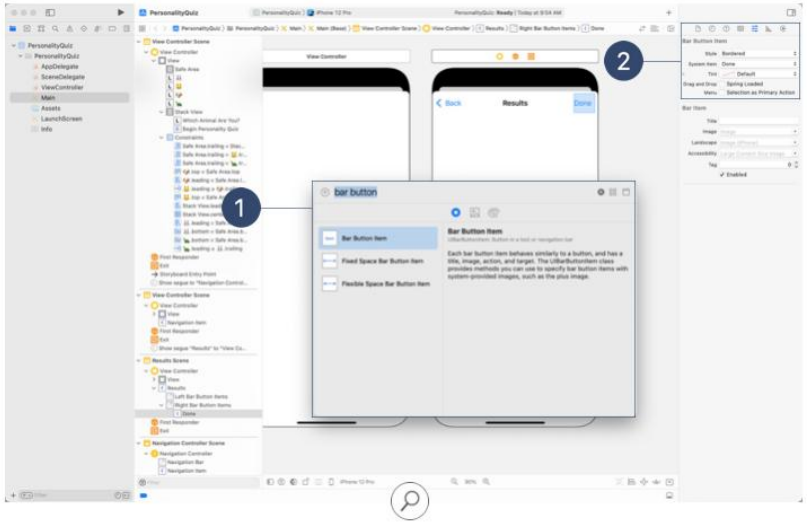


Now that you've embedded the results screen in a navigation controller, a navigation bar is available for placing titles and buttons—as long as the view controller has a navigation item. You'll add that now.

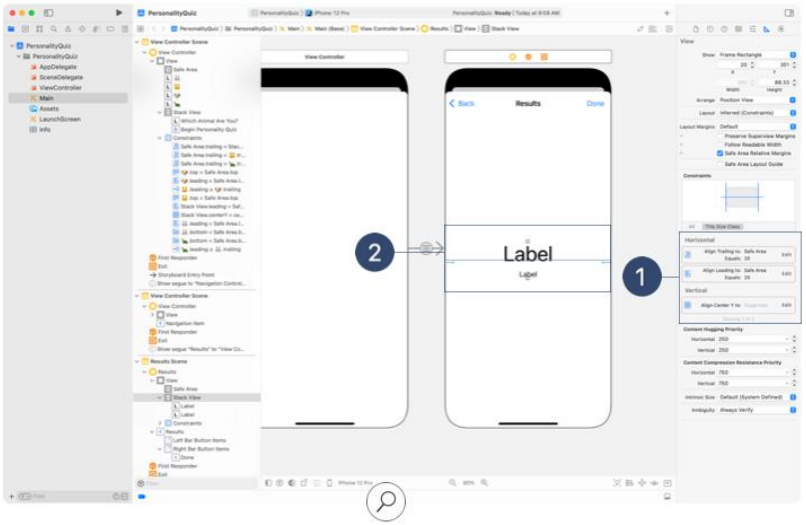


Update the final view controller's title to "Results" in the Attributes inspector for its Navigation Item, or simply double-click in the navigation item. [1]

Add a Done button that dismisses the results and returns to the introduction screen. Drag a bar button item from the Object library [1] to the right side of the navigation item. In the Attributes inspector, update the bar button's System Item attribute to Done, which also automatically changes the text and font of the button. [2]



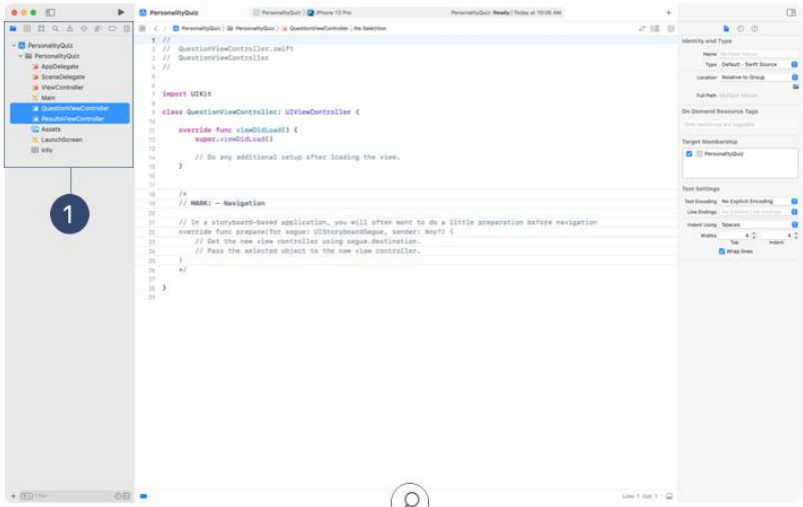
By now you're a pro at positioning stack views with constraints. To replicate the results screen, create a vertical stack view with two labels. Use the Align tool to center the stack vertically, and use the Add New Constraints tool to set the leading and trailing edges to 20. [1] Use System Font 50.0 for the first label and System Font 17.0 for the second label. Set Alignment for both labels to centered. The second label will describe more about the results and will likely run to multiple lines, so you'll want to set the Lines attribute to 0 and the Line Break to Word Wrap. Your scene should end up with two centered labels, the top one larger than the one below it. [2]



Create Descriptive Subclasses

Now that you have three view controllers in your storyboard scene, you'll need three `UIViewController` subclasses in code. Create a new file by choosing **File > New > File** from the Xcode menu bar. Select Cocoa Touch Class as your starting template, then choose UIViewController from the Subclass pop-up menu. This choice will automatically append "ViewController" to the class name, making the object's type clear to other developers. Name the class "QuestionViewController" and click Next. The Group pop-up menu should list a folder that matches the name of the project, PersonalityQuiz. Choose it and click Create.

Repeat these steps to create a second class, naming it "ResultsViewController." When you're finished, you'll see two new files in the Project navigator for your quiz. [1]



You might also notice that the Project navigator lists a subclass of `UIViewController` called "ViewController." The iOS App template automatically assigned this name to your app's first view controller. To be more descriptive, click the filename and change it to "IntroductionViewController." Next open the file and change the class name to "IntroductionViewController," then close the file.

Now your project has three descriptively named `UIViewController` subclasses. Reopen the `Main` storyboard. One at a time, select each view controller and use the Identity inspector to assign it the appropriate custom class. The first view controller will be `IntroductionViewController`, followed by the `QuestionViewController` and `ResultsViewController`.

Navigation and Workflows

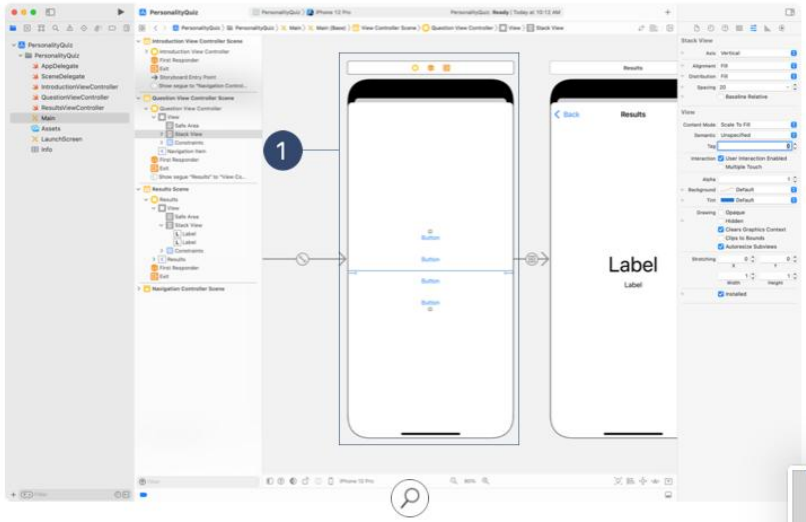
Part Three
Create Questions And Answers

During the project planning phase, you considered three different question types: single-answer, multiple-answer, and ranged response. Now take some time to come up with your list of questions. Think about how you can reword each question to fit one of the three categories.

Single-Answer Questions

Suppose you ask "Which food do you like the most?" The answer might include a list of four foods, and the player must pick one. What kind of control would you use? A simple approach would be to present a button for each answer, organized in a vertical stack view.

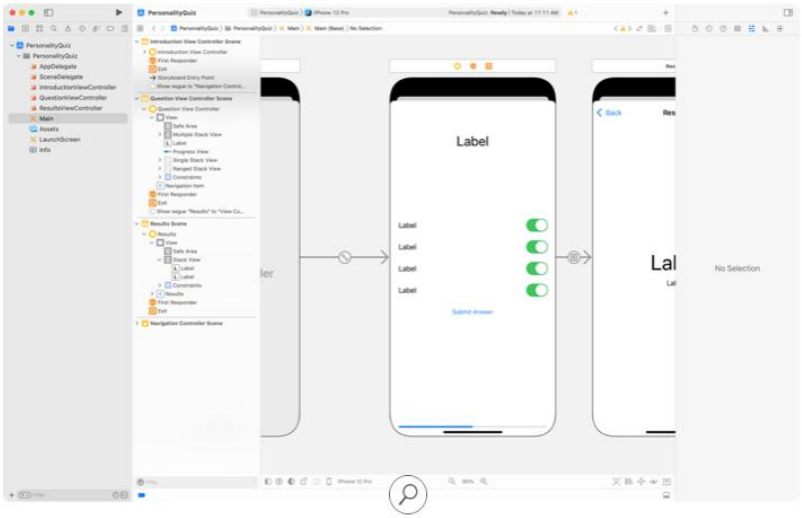
Begin by dragging a vertical stack view from the Object library to the `QuestionViewController`. Now add four buttons to the stack view. Use the Align tool to center the stack vertically within the view, then use the Add New Constraints tool to set its leading and trailing edges to 20 pixels. Add space between the buttons by setting Spacing to 20 in the Attributes inspector. If necessary, click the Update Frames button to reposition the stack based on the constraints you've created. [1]



Of course, the button titles will change based on the answers you provide. You'll update them later, when you move on to coding the quiz.

Multiple-Answer Questions

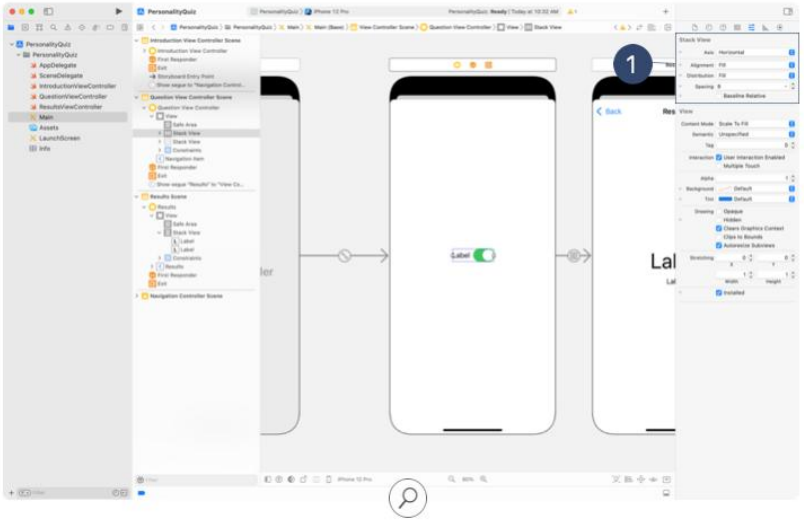
The question "Which of the following foods do you like?" suggests that the player can choose multiple answers. Rather than using buttons for the answers, it would make more sense to create pairs of labels and switches—so the player can switch on all positive answers. When the player has made their selections, they can tap a button to submit the answers and move on to the next question.



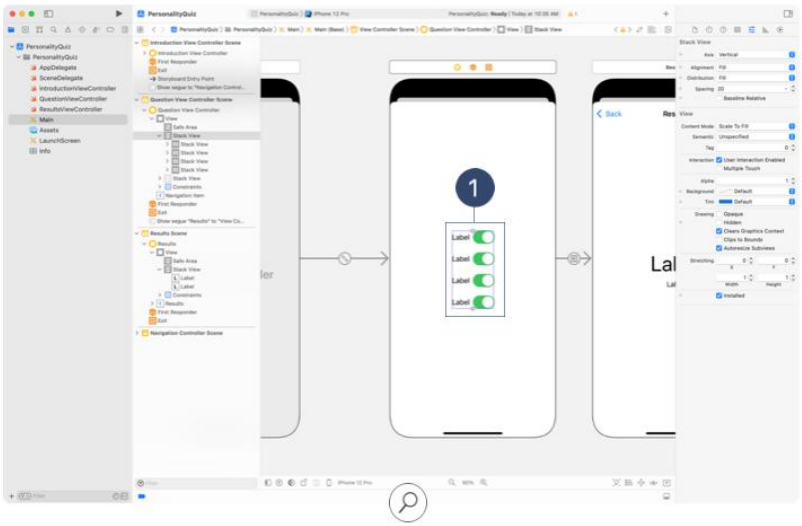
Before you begin, you might want to hide the single-answer stack view. Select the stack in the storyboard or the outline, then open the Attributes inspector, and deselect Installed at the bottom of the pane.

The switch UI is not much different from the button UI. Each label-and-switch pair can be held in a horizontal stack view. And just like the single-answer question, the rows can be held in a vertical stack view.

Begin by adding a label and a switch from the Object library. Highlight both of them, then click the Embed In button and choose Stack View. In the Attributes inspector for the stack view, make sure that the Axis is set to Horizontal and that Alignment and Distribution are both set to Fill.[1]

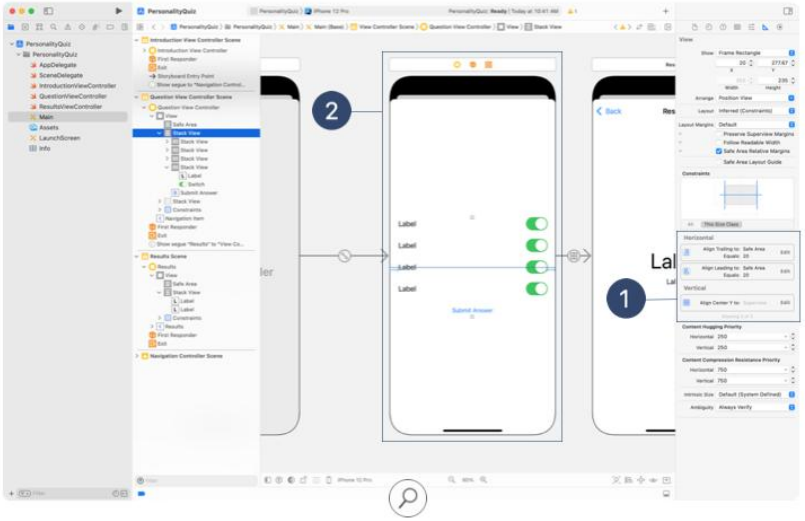


Select the stack view, then copy (**Command-C**) and paste (**Command-V**) it to add three copies to the view. Now select all four horizontal stacks, and click the Embed In button and choose Stack View to place them in another stack view. In the Attributes inspector for this new stack view, set Axis to Vertical, Alignment and Distribution to Fill, and Spacing between elements to 20. [1]



Navigation and Workflows

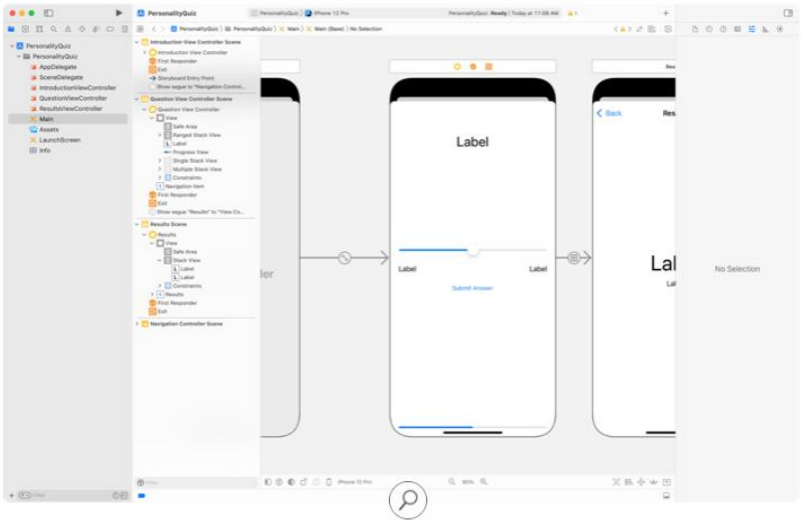
Add a button to the bottom of the stack view, and set its title to "Submit Answer." Finally, use the Align tool to center the stack vertically within the view, then use the Add New Constraints tool to set the leading and trailing edges to 20 pixels from each margin. [1] If necessary, use the Update Frames button to reposition the frames based on the constraints you just created. [2]



Navigation and Workflows

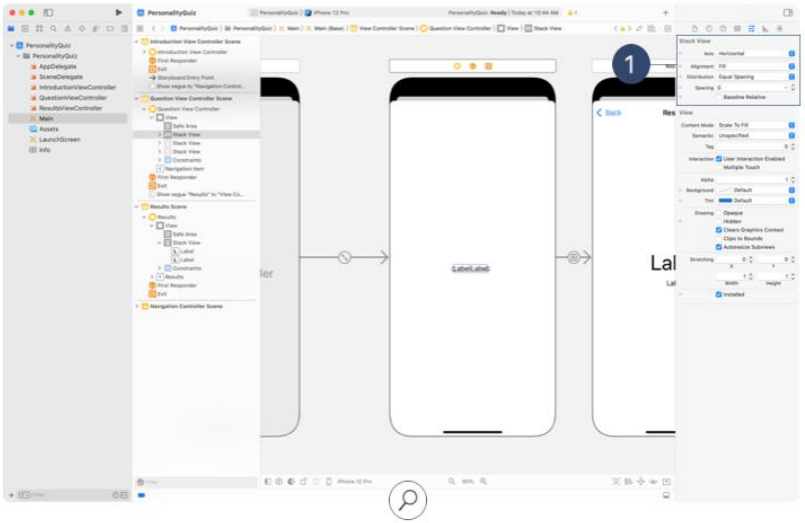
Range Questions

The third type of question might follow this format: "How much do you like this particular food?" You could probably think of a way to use a button or a switch for the answer, but the player might have a better experience if their choice feels a little more freeform. To allow the player a range of answers, you could use a slider as the input control, with a label on either end of the slider.

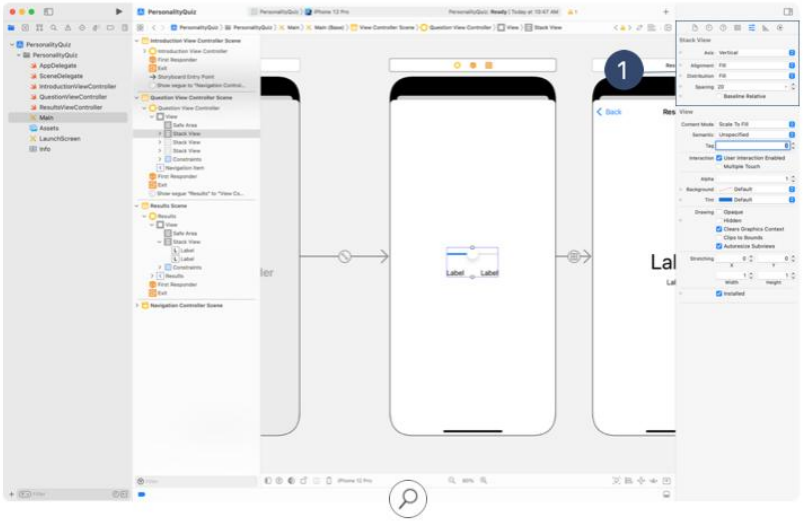


To make things easier, you might want to hide the multiple-answer stack, just as you did earlier with the single-answer stack. Select it in the storyboard, open the Attributes inspector, and deselect Installed at the bottom of the pane.

You can use stack views to create this interface without having to define very many constraints—similar to the switch approach. Begin by adding two labels to the canvas from the Object library, then select both, click the Embed In button and choose Stack View. In the Attributes inspector, check that the Axis is set to Horizontal, Alignment is set to Fill, and Distribution is set to Equal Spacing. [1]

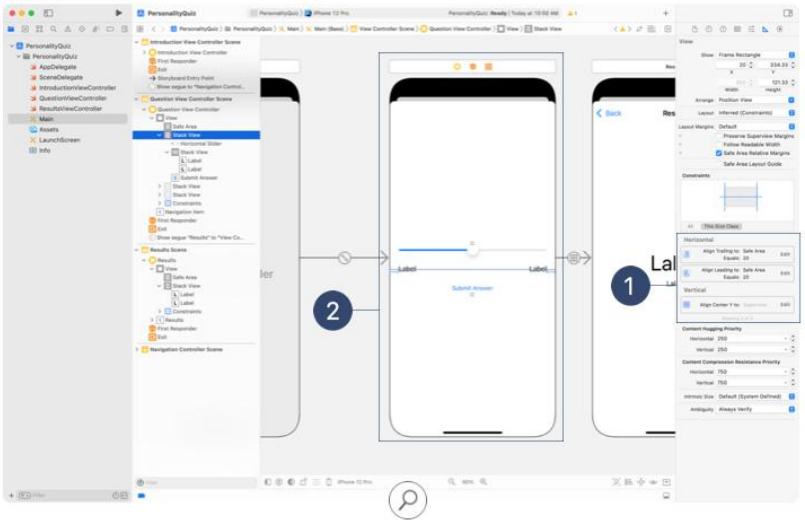


Next, drag a slider from the Object library onto the canvas. Select the slider and the horizontal stack, then click the Embed In button and choose Stack View. In the Attributes inspector, verify that the Axis is set to Vertical, the Alignment and Distribution are both set to Fill, and the Spacing is set to 20. [1]



Navigation and Workflows

Add a button to the bottom of the stack, and set its title to "Submit Answer." Use the Align tool to center the stack vertically within the view, then use the Add New Constraints tool to align the leading and trailing edges with 20 pixels of spacing to each margin. [1] If necessary, use the Update Frames button to reposition the frames based on the constraints you've created. [2]

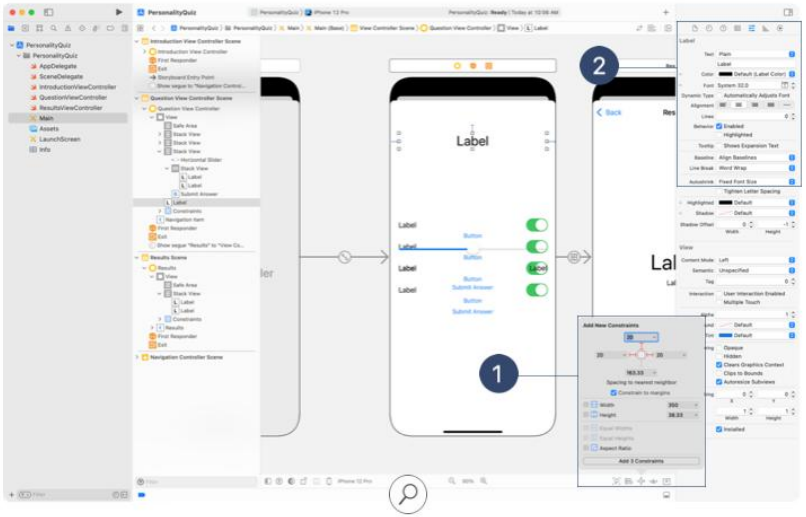


Before you move on, you'll need to re-enable the stack views that you uninstalled during the building process. In the Document Outline, select each stack view, then select the Installed checkbox in the Attributes inspector.

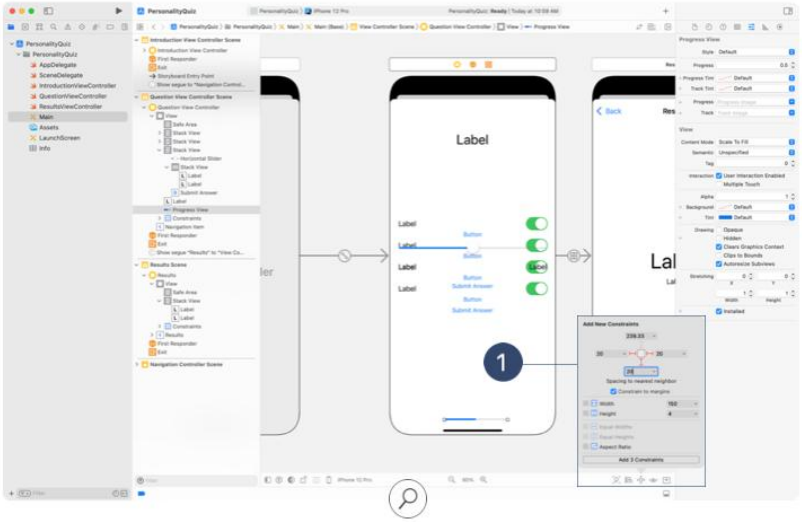
Navigation and Workflows

Question Label and Progress

No matter what kind of question you ask your players, you need to display it in a label. Add a label to the top of the view. Use the Add New Constraints tool to position the label 20 pixels below the navigation bar and 20 pixels from the leading and trailing margins. [1] In the Attributes inspector, set the text alignment to center and the font to System Font 32.0. Set the Lines attributes to 0 to give the label the ability to use as many lines as needed. Change the Line Break attribute to Word Wrap. [2]



Players often like to know how far along they are in the quiz. Search for "Progress View" in the Object library, and add it to the view. Use the Add New Constraints tool to position the progress view 20 pixels from the bottom and 20 pixels from the leading and trailing edge of the view. [1]



Part Four
Models And Outlets

So far in this lesson, you've designed the view controllers in the storyboard, and you've got three `UIViewController` subclasses ready to receive some code. Now it's time to create structures that hold the question data and to update the user interface based on the values of each question and its answers. Once the data has been laid out, you can update the user interface based on which question is being displayed.

Data Models

Create a new file named "Question" to house the model definitions. Use this file to define all the structures necessary for your personality quiz. You can create this file by choosing **File > New > File** (or **Command-N**) from the Xcode menu bar, then selecting "Swift file."

It's safe to assume that every `Question` will have text to represent the question itself, along with an array of `Answer` objects. Since your quiz can use three different types of input methods, you'll create an `enum` that describes the question's response type: single-answer, multiple-answer, or ranged response. An example of the structure is shown below:

```
struct Question {
    var text: String
    var type: ResponseType
    var answers: [Answer]
}

enum ResponseType {
    case single, multiple, ranged
}
```


Navigation and Workflows

Every answer corresponds to a result type. In the animal example, suppose you ask "Which of these foods do you like the most?" and the possible answers are: "Steak," "Fish," "Carrots," and "Corn." Each response corresponds to a dog, cat, rabbit, and turtle, respectively—and therefore, to a particular emoji. If the `answers` property was an array of strings, there wouldn't be a simple way to associate an answer with a particular result. Instead, an `Answer` struct will have a string to display to the player and a `type` property that ties the answer to a specific result.

Here's an example of the data:

```
struct Answer {
    var text: String
    var type: AnimalType
}

enum AnimalType: Character {
    case dog = "🐶", cat = "🐱", rabbit = "🐰", turtle = "🦎"
}
```

Typically at the end of a personality quiz, the player receives some text about the outcome of the quiz. Since you've already defined an `enum` to represent each personality type—or in this case, animal type—you could include a `definition` property that will be presented as a label on the results screen.

Here's an example of a definition for the animal types:

```
enum AnimalType: Character {
    case dog = "🐶", cat = "🐱", rabbit = "🐰", turtle = "🐢"

    var definition: String {
        switch self {
        case .dog:
            return "You are incredibly outgoing. You surround
            yourself with the people you love and enjoy
            activities with your friends."
        case .cat:
            return "Mischievous, yet mild-tempered, you enjoy
            doing things on your own terms."
        case .rabbit:
            return "You love everything that's soft. You are
            healthy and full of energy."
        case .turtle:
            return "You are wise beyond your years, and you
            focus on the details. Slow and steady wins the race."
        }
    }
}
```


Display Questions and Answers

The `QuestionViewController` will hold the array of `Question` objects in a property called `questions`. As you create the objects, you'll need to take special care with how many `Answer` objects you place in the `answers` property. When you built stack views for single- and multiple-answer responses, you created four buttons and four switches to represent a static number of possible answers. So any `Question` you create with a `type` property that's set to `single` or `multiple` must have exactly four `Answer` objects.

For the ranged response, you can get away with only two answers: the two ends of the slider. But it would be better to define four possible ranges so that the question can give points to each of the four outcomes. The collection of answers for a ranged response needs to be in some sort of order—from least likely to most likely, for example—so that you can accurately assign the answers to a result.

In the following example, the array is filled with a question of each response type: single-answer, multiple-answer, and ranged response:

```
var questions: [Question] = [
  Question(
    text: "Which food do you like the most?",
    type: .single,
    answers: [
      Answer(text: "Steak", type: .dog),
      Answer(text: "Fish", type: .cat),
      Answer(text: "Carrots", type: .rabbit),
      Answer(text: "Corn", type: .turtle)
    ]
  ),

  Question(
    text: "Which activities do you enjoy?",
    type: .multiple,
    answers: [
      Answer(text: "Swimming", type: .turtle),
      Answer(text: "Sleeping", type: .cat),
      Answer(text: "Cuddling", type: .rabbit),
      Answer(text: "Eating", type: .dog)
    ]
  ),

  Question(
    text: "How much do you enjoy car rides?",
    type: .ranged,
    answers: [
      Answer(text: "I dislike them", type: .cat),
      Answer(text: "I get a little nervous", type: .rabbit),
      Answer(text: "I barely notice them", type: .turtle),
      Answer(text: "I love them", type: .dog)
    ]
  )
]
```


Display Questions with the Right Controls

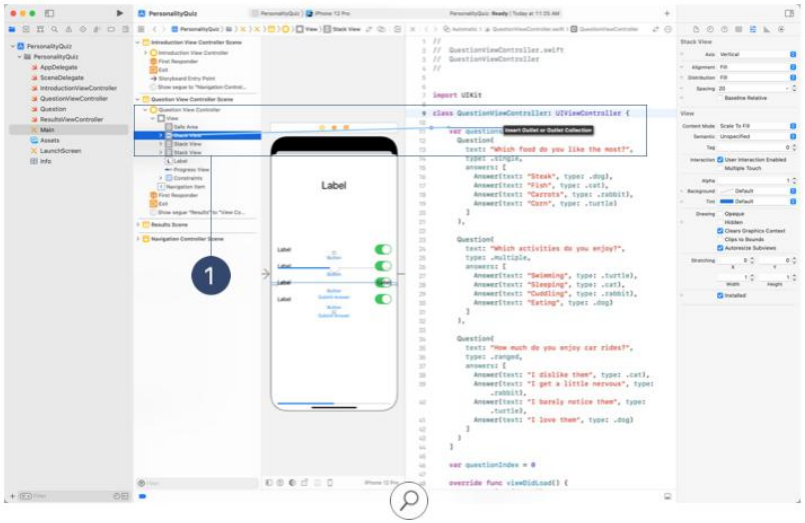
Now that you have a list of questions to draw from, you'll need to keep track of which ones your app has already displayed and to calculate when you've displayed them all. One technique is to use an integer as an index into the `questions` collection. This integer will start at 0 (the index of the first element in a collection), and you'll increment the value by 1 after the player answers each question.

Add a property called `questionIndex` to your `QuestionViewController`:

```
var questionIndex = 0
```

As the player moves from question to question, you'll need to show the correct stack view and to hide the other two. But before you can write code that changes the stack view's visibility, you'll need to create the necessary outlets and actions.

Navigation and Workflows



Open the `Main` storyboard and select `QuestionViewController`. Open an assistant editor to view `QuestionViewController` alongside the storyboard. **Control-drag** from the single-answer stack view to the definition of the `QuestionViewController` class, ❶ then release the mouse or trackpad to bring up the popover. Verify that the Connection type is set to Outlet, then enter "singleStackView" into the Name field and click Connect. Repeat these steps two more times, entering the names `multipleStackView` for the multiple-answer stack and `rangedStackView` for the ranged stack.

Next, create a reusable method, called `updateUI()`, that you can call before displaying each question to the player. You should call this method in `viewDidLoad()` to set the proper interface for the first question.

```
override func viewDidLoad() {
    super.viewDidLoad()
    updateUI()
}

func updateUI() {

}
```


The `updateUI()` method is responsible for updating a few key pieces of the interface, including the title in the navigation bar and the visibility of the stack views. You can use the `questionIndex` property to create a unique title—for example, "Question #4"—in the navigation item for each question. With the stack views, it's easiest if you hide all three stack views, then inspect the `type` property of the `Question` to determine which stack should be visible.

You can use the `questionIndex` property in conjunction with the `questions` collection to access the particular question:

```
func updateUI() {
    singleStackView.isHidden = true
    multipleStackView.isHidden = true
    rangedStackView.isHidden = true

    navigationItem.title = "Question #\(questionIndex + 1)"

    let currentQuestion = questions[questionIndex]

    switch currentQuestion.type {
    case .single:
        singleStackView.isHidden = false
    case .multiple:
        multipleStackView.isHidden = false
    case .ranged:
        rangedStackView.isHidden = false
    }
}
```

Build and run your app. If you've set everything up properly, the stack view that's visible should correspond to the first question you defined in the `questions` property. Try reordering the questions to test each interface.

Update the Buttons and Label Text

The interface on your question screen works, but you still need to update the button titles and label text. To make this happen, you'll need to create outlets for the labels and buttons associated with each stack view.

In addition to the outlets you created for the stack views, this screen requires 12 outlets for the controls and labels. There are four button outlets in the single-answer stack view, four label outlets in the multiple-answer stack view, and two label outlets in the ranged response stack view. You also have the label that displays the question text near the top of the screen, and the progress view near the bottom.

When you create a large number of outlets, it's important to use concise, easily recognizable variable names—and to keep the variable declarations organized near their corresponding stack view outlet. The code below provides an example of good variable names and outlet organization:

```
@IBOutlet var questionLabel: UILabel!

@IBOutlet var singleStackView: UIStackView!
@IBOutlet var singleButton1: UIButton!
@IBOutlet var singleButton2: UIButton!
@IBOutlet var singleButton3: UIButton!
@IBOutlet var singleButton4: UIButton!


@IBOutlet var multipleStackView: UIStackView!
@IBOutlet var multiLabel1: UILabel!
@IBOutlet var multiLabel2: UILabel!
@IBOutlet var multiLabel3: UILabel!
@IBOutlet var multiLabel4: UILabel!


@IBOutlet var rangedStackView: UIStackView!
@IBOutlet var rangedLabel1: UILabel!
@IBOutlet var rangedLabel2: UILabel!

@IBOutlet var questionProgressView: UIProgressView!
```


Navigation and Workflows

Go ahead and create the outlets above. Since the screen has so many controls overlapping one another, you'll probably find it's easier to **Control-drag** from the item in the Document Outline to the view controller definition, rather than **Control-dragging** from the item on the canvas.

With the outlets in place, you can update each of the controls in the `updateUI()` method. Two of the outlets, `questionLabel` and `questionProgressView`, will need to be updated with every new question. The label and button outlets need to be updated only if their related stack view will be displayed.

For the question label, assign its text to the current question string. For the progress view, calculate the percentage progress by dividing the `questionIndex` by the total number of questions.

```
func updateUI() {
    singleStackView.isHidden = true
    multipleStackView.isHidden = true
    rangedStackView.isHidden = true

    let currentQuestion = questions[questionIndex]
    let currentAnswers = currentQuestion.answers
    let totalProgress = Float(questionIndex) /
        Float(questions.count)

    navigationItem.title = "Question #\(questionIndex + 1)"
    questionLabel.text = currentQuestion.text
    questionProgressView.setProgress(totalProgress, animated:
        true)

    switch currentQuestion.type {
    case .single:
        singleStackView.isHidden = false
    case .multiple:
        multipleStackView.isHidden = false
    case .ranged:
        rangedStackView.isHidden = false
    }
}
```

To keep the `switch` statement concise, you can refactor the updates to stack specific controls into their own methods.

In the single-answer stack view, each button title corresponds to an answer. Use the `setTitle(_:for:)` method to update the title. The first button will use the first answer string, the second button will use the second answer string, and so on.

```
func updateSingleStack(using answers: [Answer]) {
    singleStackView.isHidden = false
    singleButton1.setTitle(answers[0].text, for: .normal)
    singleButton2.setTitle(answers[1].text, for: .normal)
    singleButton3.setTitle(answers[2].text, for: .normal)
    singleButton4.setTitle(answers[3].text, for: .normal)
}
```

Similarly, in the multiple-answer stack view, each label's text corresponds to an answer. Set the `text` property of the first label to the first answer string, and repeat for the other three labels.

```
func updateMultipleStack(using answers: [Answer]) {
    multipleStackView.isHidden = false
    multiLabel1.text = answers[0].text
    multiLabel2.text = answers[1].text
    multiLabel3.text = answers[2].text
    multiLabel4.text = answers[3].text
}
```

For the ranged response, you'll need to set up the stack view a bit differently. While there are only two labels to update, the quiz will work better if every question has four answers (even though only two answers are required).

Since the number of answers isn't guaranteed, it wouldn't be safe to index directly into the collection. For example, if you used `answers[3]` to access the fourth element of `answers`, but the collection contained only two `Answer` structs, the program would crash.

No matter how many answers you have for your ranged response question, the `first` and `last` properties of the collection allow you to safely access the two `Answer` structs that correspond to the labels.

```
func updateRangedStack(using answers: [Answer]) {
    rangedStackView.isHidden = false
    rangedLabel1.text = answers.first?.text
    rangedLabel2.text = answers.last?.text
}
```

With these three new methods defined, update the switch statement cases in `updateUI()` to call them.

```
switch currentQuestion.type {
case .single:
    updateSingleStack(using: currentAnswers)
case .multiple:
    updateMultipleStack(using: currentAnswers)
case .ranged:
    updateRangedStack(using: currentAnswers)
}
```

Build and run your app. The labels and buttons should all update to reflect the first question. Great work! If you see controls that aren't updating, use the Connections inspector for `QuestionViewController` to verify that you've created each `@IBOutlet` properly. Hover over every item in the list to check their outlets, and—if necessary—break any incorrect outlets.

Navigation and Workflows

Retrieve Answers with Actions

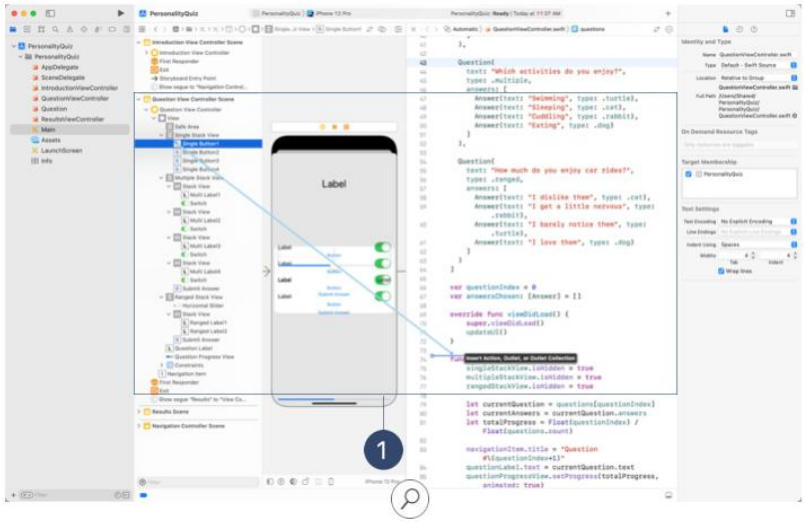
So far, you've generated a list of questions, and your app is displaying the correct user interface for the first question. That's an excellent start. In this section, you'll record the player's answers and move to the next question. When the player has answered every question in the collection, you'll present the results screen.

No matter which set of controls you use, you'll need to initialize an empty collection that can store the player's answers.

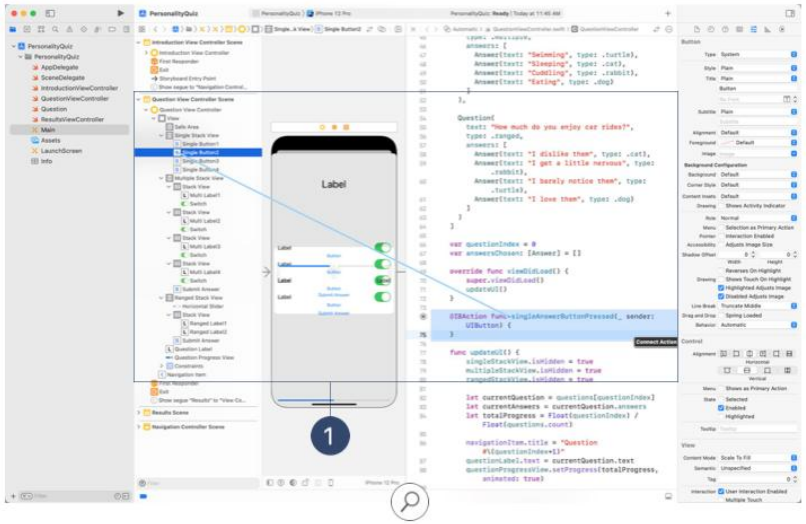
```
var answersChosen: [Answer] = []
```

In the single-answer stack view, you'll determine which outcome each tapped button corresponds to, append it to the collection, then move on to the next question. All four buttons will perform the same work, so you can create one `@IBAction` that any of the four buttons will call when tapped.

Begin by **Control-dragging** from the first button in the single-answer stack view to a space within the `QuestionViewController` class definition.[1] This is the same way you created an `@IBOutlet`, but this time you'll change the Connection type to Action. Name the method "singleAnswerButtonPressed" and change the Type from Any to UIButton—so that the `sender` parameter of the method will be of type `UIButton`.



Navigation and Workflows



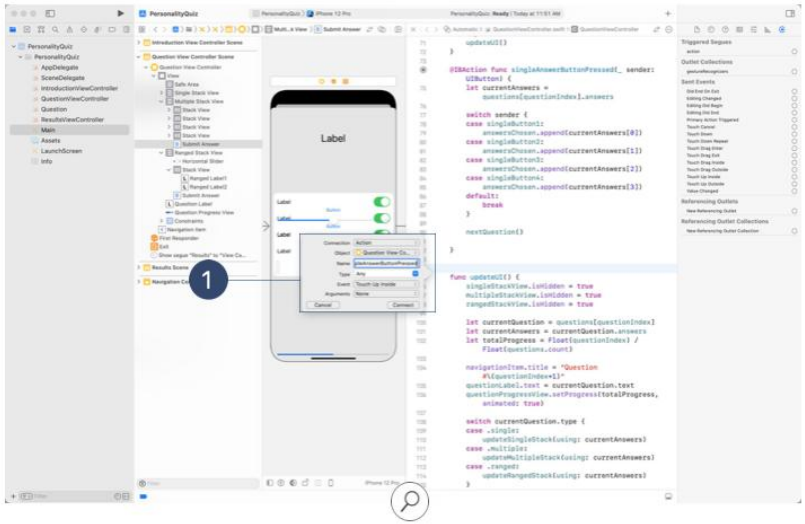
Connect the three remaining buttons in the single-answer stack view to this newly created `@IBAction`.[1]

Why do you need to update the type? You're tying the tap from multiple buttons to this one action, so you'll need to specify *which* button triggered the method. You can use an `if` statement and `==` to compare two `UIButton` objects, or you can use a `switch` statement. If the method was triggered using `singleButton1`, the app will know that the player selected the first answer.

Try filling in the action body with code that checks which button was tapped, then appends the chosen answer to the `answersChosen` array. Once you've finished adding the answer to the array, you'll need to go to the next question. Create a new method called `nextQuestion()` and leave it empty. You'll fill it in later. For now, just add a call to `nextQuestion()` at the end of `singleAnswerButtonPressed(_:)`

```
@IBAction func singleAnswerButtonPressed(_ sender: UIButton) {
    let currentAnswers = questions[questionIndex].answers

    switch sender {
    case singleButton1:
        answersChosen.append(currentAnswers[0])
    case singleButton2:
        answersChosen.append(currentAnswers[1])
    case singleButton3:
        answersChosen.append(currentAnswers[2])
    case singleButton4:
        answersChosen.append(currentAnswers[3])
    default:
        break
    }

    nextQuestion()
}
```


For the multiple-answer user interface, you'll determine which answers to add to the collection based on the switches the player has enabled. **Control-drag** from the Submit Answer button to code, and create an action with the name "multipleAnswerButtonPressed." Go ahead and change the Arguments attribute to None, since you don't need the button to determine which answers were chosen. [1]



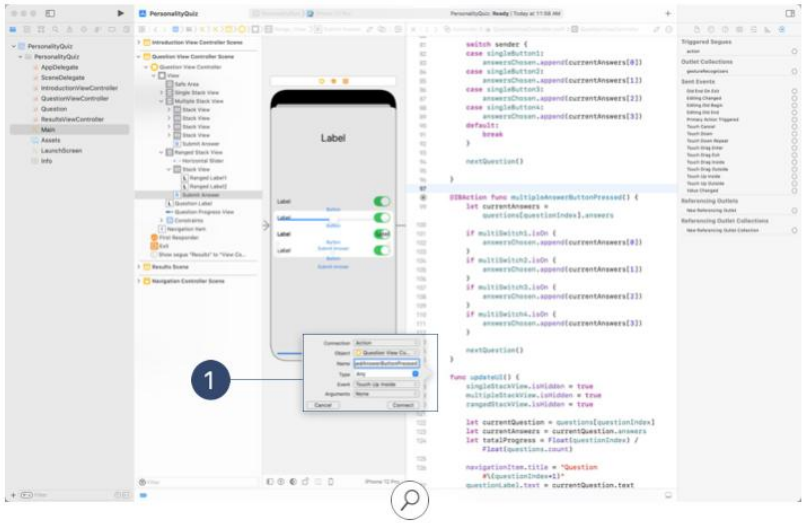
Next, create four outlets, one for each `UISwitch`, so that you can check which are enabled and then add those answers to the collection. **Control-drag** from each `UISwitch` in the Document Outline to code, and give each switch a name. To keep your code neat and organized, enter the code for each of these outlets near the label variables associated with the multiple-answer stack view.

```
@IBOutlet var multiSwitch1: UISwitch!
@IBOutlet var multiSwitch2: UISwitch!
@IBOutlet var multiSwitch3: UISwitch!
@IBOutlet var multiSwitch4: UISwitch!
```

If the first switch is enabled, you want to add the first answer. Unlike the method with single-answer questions, this method allows you to append as many as four answers per question.

```
@IBAction func multipleAnswerButtonPressed() {
    let currentAnswers = questions[questionIndex].answers

    if multiSwitch1.isOn {
        answersChosen.append(currentAnswers[0])
    }
    if multiSwitch2.isOn {
        answersChosen.append(currentAnswers[1])
    }
    if multiSwitch3.isOn {
        answersChosen.append(currentAnswers[2])
    }
    if multiSwitch4.isOn {
        answersChosen.append(currentAnswers[3])
    }

    nextQuestion()
}
```


For a ranged response question, you'll read the current position of the `UISlider` and use that value to determine which answer to add to the collection. **Control-drag** from the Submit Answer button to code, and create an action with the name "rangedAnswerButtonPressed." Again, change the Arguments attribute to None. [1]



Next, create an `@IBOutlet` for the `UISlider`. **Control-drag** from the slider in the Document Outline to code and give it a name. As you did in earlier steps, place the code for this outlet near the label variables associated with the ranged response stack view.

```
@IBOutlet var rangedSlider: UISlider!
```

Take a moment to think about how you can use the slider's value to correspond to four different answers. A slider's value ranges from 0 to 1, so a value between 0 and 0.25 could correspond to the first answer, and an answer between .75 and 1 could correspond to the final answer.

To convert a slider value to an array's index, use the equation `index = slider value * (number of answers - 1)` rounded to the nearest integer. This results in the following method implementation for `rangedAnswerButtonPressed`:

```
@IBAction func rangedAnswerButtonPressed() {
    let currentAnswers = questions[questionIndex].answers
    let index = Int(round(rangedSlider.value *
      Float(currentAnswers.count - 1)))

    answersChosen.append(currentAnswers[index])

    nextQuestion()
}
```

Pass Data to the Results View

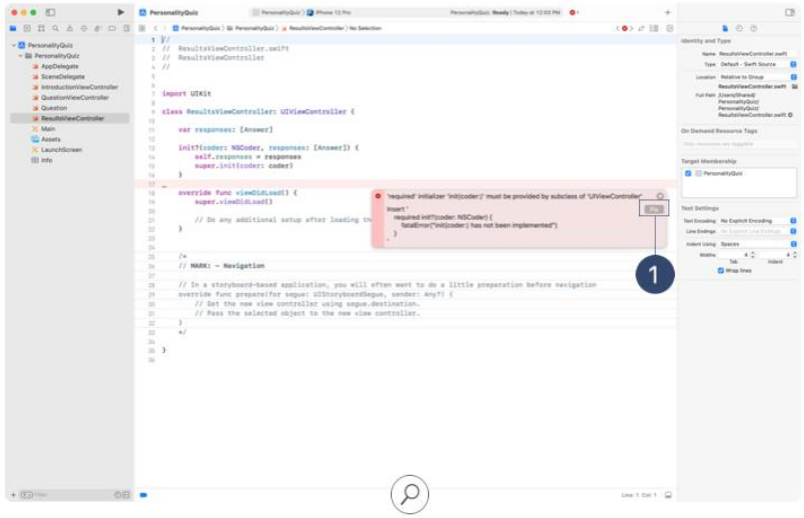
When it is time to show the results screen you will need to pass the user's responses to it. Open `ResultsViewController` and add a property to hold the user's responses.

```
var responses: [Answer]
```

This will cause Xcode to report a compilation error stating "Class 'ResultsViewController' has no initializers". This is because `responses` does not have an initial value nor is it eve assigned a value in an initializer. To resolve this you will create a custom initializer for `ResultsViewController` that takes `responses` as an argument–satisfying the compiler. Add the following initializer:

```
init?(coder: NSCoder, responses: [Answer]) {
    self.responses = responses
    super.init(coder: coder)
}
```

This new initializer takes two parameters, `coder` and `responses`. The `coder` parameter will be provided and is used by `UIKit` to initialize your view controller from the information defined in your Storyboard. The `responses` parameter will be supplied by you, when calling this initializer, and assigned to `self.responses` which you just added. Finally, the `super` initializer is called passing through `coder`.



Unfortunately all this work led to a new compilation error stating "`required` initializer `init(coder:)` must be provided by subclass of `UIViewController`." When you provide your own initializer, you must implement any `required` initializers that the superclass defines.

For more information, click the red symbol next to the error.

Xcode provides you with a "fix-it" for this problem. Click the "Fix" button to insert the suggested code fix.

```
required init?(coder: NSCoder) {
    fatalError("init(coder:) has not been implemented")
}
```

In this case, you're no longer interested in the provided required initializer because you'll be using your own. This implementation of the required initializer, if called, will now crash your application—you won't be calling it.

Respond to Answered Questions

Now that you've handled the user input for each question and can pass responses to `ResultsViewController`, it's time to implement the `nextQuestion()` method in `QuestionViewController`. For this, you increment the value of `questionIndex` by 1, then determine whether any remaining questions exist. If they do, you call `updateUI()` to update the title and display the proper stack view. The method uses the new value of `questionIndex` to display the next question. If no questions remain, it's time to present the results using the segue you created in the storyboard.

```
func nextQuestion() {
    questionIndex += 1

    if questionIndex < questions.count {
        updateUI()
    } else {
        performSegue(withIdentifier: "Results", sender: nil)
    }
}
```

Build and run your app, then test each of your input controls. Do you notice any bugs? One issue is that the interfaces for multiple-answer and ranged responses retain the answer values from the previous question of the same type. For example, if the player has moved a slider all the way to the left for one question, the next question that uses a slider starts with the slider all the way to the left.

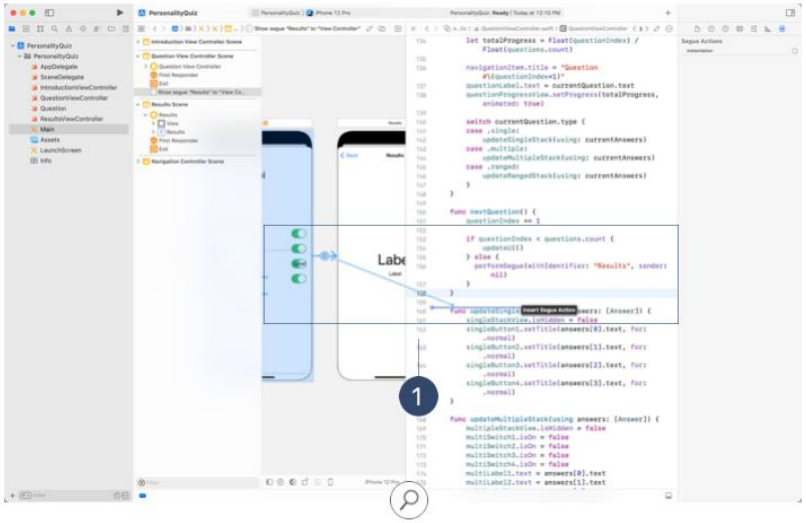
To resolve this problem, you can reset the positions of the switches and slider to logical defaults when the next question is displayed.

Update the `updateMultipleStack(using:)` and `updateRangedStack(using:)` methods to include code that resets the positions of their controls.

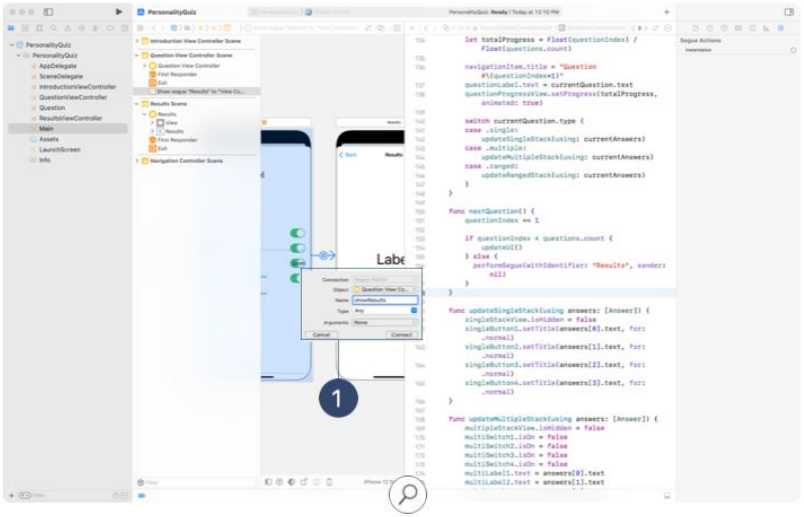
```
func updateMultipleStack(using answers: [Answer]) {
    multipleStackView.isHidden = false
    multiSwitch1.isOn = false
    multiSwitch2.isOn = false
    multiSwitch3.isOn = false
    multiSwitch4.isOn = false
    multiLabel1.text = answers[0].text
    multiLabel2.text = answers[1].text
    multiLabel3.text = answers[2].text
    multiLabel4.text = answers[3].text
}
func updateRangedStack(using answers: [Answer]) {
    rangedStackView.isHidden = false
    rangedSlider.setValue(0.5, animated: false)
    rangedLabel1.text = answers.first?.text
    rangedLabel2.text = answers.last?.text
}
```

The second, most glaring, issue is that you cannot proceed to the results view because the app crashes. When you call `performSegue(withIdentifier: "Results", sender: nil)`, `UIKit` is using the default required initializer for `ResultsViewController`. If you recall, you made that method crash when called. You need to tell your Storyboard what to call instead when this segue is triggered.

Locate the Results segue as identified by the arrow between the question and results controllers. **Control-drag** from the segue to `QuestionsViewController` to create an `@IBSegueAction`.[1]



Fill in `showResults` for the name and set Arguments to `None`, then click Connect.[1] This inserts a method similar to an `@IBAction` except that it's unique to segues. The method is set to return an optional `ResultsViewController`. It's your job within the implementation to initialize and return one. Using the custom initializer you created earlier, return a new instance of `ResultsViewController` passing in the provided `coder` and `answersChosen` from `self`.




```
@IBSegueAction func showResults(_ coder: NSCoder) ->
    ResultsViewController? {
     return ResultsViewController(coder: coder, responses:
        answersChosen)
}
```

Now when the Results segue is performed, this method is called and its return value is used to present the controller.

Part Five
Calculate And Display Results

You have a working user interface for multiple question types, and you're recording the player's answers. The final steps include calculating the results, presenting them, and dismissing the results screen so that the quiz is ready for a new player.

Calculate Answer Frequency

Now that the `ResultsViewController` has the player's responses, it's time to think about how to calculate the personality. What was the most common personality type among the selected answers? In this example, if the player gave two answers that corresponded closely to dog and only one for each other animal, the best result would be dog. If two or more animals are tied for the most answers, either one would be a valid result.

How can you tally up the results? You need to loop through each of the `Answer` structures in the `responses` property and calculate which `type` was most common in the collection. A good solution might be the dictionary data model, where the key is the response type and the value is the number of times a player has selected it as an answer—in effect a histogram. For example, if the player gave two answers that corresponded closely to dog and only one for each other animal, the dictionary looks like the following:

```
{
  cat : 1,
  dog : 2,
  rabbit : 1,
  turtle : 1
}
```


Navigation and Workflows

To begin, within `ResultsViewController` declare a method named `calculatePersonalityResult` that you can call in `viewDidLoad()`. Within that method, you calculate the frequency of each response, as in the code above. For the "Which animal are you?" quiz, you can use the following code:

```
override func viewDidLoad() {
    super.viewDidLoad()
    calculatePersonalityResult()
}

func calculatePersonalityResult() {
  let frequencyOfAnswers = responses.reduce(into: [:]) {
    (counts, answer) in
        counts[answer.type, default: 0] += 1
    }
}
```

When calculating the result, you don't need the entire `Answer` structure. You care about only the `type` property of each `Answer`. So, you can reduce `responses` into a new dictionary of type `[AnimalType: Int]`, where `Int` is the frequency a player selected the corresponding `AnimalType`.

Here's a breakdown of what's happening. `reduce(into:)` is a method on `Array` that iterates over each item, combining them into a single value using the code you provide. Just as a `for...in loop` executes its body once for each item in an array, `reduce(into:)` executes the code inside the following braces once per item. But the code inside the braces differs from the body of a loop. It's a closure that takes two parameters, which you see as `(counts, answer) in`. The first parameter is the item you're reducing into. The second parameter is the item from the collection for current iteration. (You can find more information about closures in the Swift Language Guide.)

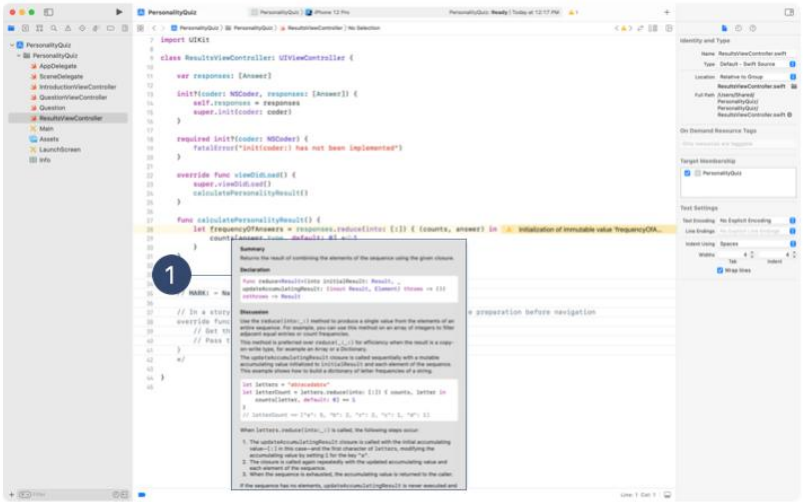
In this case, you're reducing `responses` into a dictionary that's initially empty—the argument passed for `into:`—and then incrementing the value in the dictionary that corresponds to the key for each response. Notice the `default:` parameter in the subscript to `counts`. This is a way to guard against missing keys in a dictionary. If the key doesn't exist, it's created and set to the value specified for `default`. By supplying a default value, you avoid the extra logic you'd otherwise need to handle the `nil` value a dictionary normally returns when a key doesn't exist.

The following is the equivalent code without using `default` subscript syntax:

```
let frequencyOfAnswers = responses.reduce(into:
    [AnimalType: Int]()) { (counts, answer) in
     if let existingCount = counts[answer.type] {
         counts[answer.type] = existingCount + 1
     } else {
         counts[answer.type] = 1
     }
}
```

If you don't specify a default value for a nonexistent key, you must unwrap the subscript result before attempting to increment it, and insert a new value if it doesn't exist yet. Also notice that you're now required to specify type information for the initial value for the `into:` parameter. When `0` was specified as the default value for `counts`, Swift was able to infer that values of the `counts` dictionary should be of type `Int`. (In both cases, Swift can infer the keys to be `AnimalType` because that's the type you're passing.)

For more details, **Option+Click** `reduce(into:)` to open quick help. You'll find an example usage that's exactly what you've just done—it's the ideal method for this task.



Determine the Most Frequent Answers

Now that you have a dictionary that knows the frequency of each response, it's possible to determine which value is the largest. You can use the Swift `sorted(by:)` method on a dictionary to place each key/value pair into an array, sorting the `value` properties in descending order.

```
let frequentAnswersSorted = frequencyOfAnswers.sorted(by:
{ (pair1, pair2) in
    return pair1.value > pair2.value
})

let mostCommonAnswer = frequentAnswersSorted.first!.key
```

How does this code work exactly? Assume your dictionary looks like the following:

```
{
  cat : 1,
  dog : 2,
  rabbit : 1,
  turtle : 1
}
```

The parameter you pass into `sorted(by:)` is a closure that takes any two key-value pairs. In the animal quiz, `pair1` might correspond to `cat : 1` and `pair2` might be `dog : 2`. Within the body of the closure, you need to return a Boolean value to indicate which of the pairs is larger. In the case of `return 1 > 2`, the Boolean value is `false` —so the method knows that `pair2` is larger than `pair1`.

When the method is finished, the array `frequentAnswersSorted` might look something like the following code. For key/value pairs that have the same value, there's no way to rank one over the other—so rabbit, turtle, and cat may be in a different order. But it doesn't matter, since you only care about the first element in the array.

```
[(dog, 2), (rabbit, 1), (turtle, 1), (cat, 1)]
```

You can simplify the closure using four techniques. First, you can use `$0` and `$1` as implicit parameter names without defining your own parameters, so you can remove `(pair1, pair2) in`. You can also refer to the elements of the pairs by number rather than by name, replacing the reference to `.value` with `1`. Next, you can eliminate the `return` statement for closures that contain just one expression. The closure automatically returns that value. Finally, you can use trailing closure syntax to remove the parentheses around the call to `sorted(by:)` and eliminate the argument label. The call to `reduce(into:)` above also used trailing closure syntax. That method has two parameters. You needed to pass the first argument using traditional function-call syntax inside parentheses. Then you supplied the last argument—the trailing closure—outside the parentheses.

Using these techniques, and retrieving the key of the first element of the result, you can simplify the code into one line:

```
let mostCommonAnswer = frequencyOfAnswers.sorted { $0.1 >
   $1.1 }.first!.key
```

View the Results

Now all that's left is to update the text of your labels to appropriate values inside `calculatePersonalityResult`. You'll need to add some outlets in `ResultsViewController` so that each label's text can be updated in code. Open the assistant editor and **Control-drag** from each label to a space within the `ResultsViewController` class definition. Give each label an appropriate name.

```
@IBOutlet var resultAnswerLabel: UILabel!
@IBOutlet var resultDefinitionLabel: UILabel!
```


Add the following code at the end of `calculatePersonalityResult` to update your labels with the data held in `mostCommonAnswer`:

```
resultAnswerLabel.text = "You are a \(mostCommonAnswer.rawValue)!"
resultDefinitionLabel.text = mostCommonAnswer.definition
```

Build and run your app, and you'll finally get to see your quiz results visually.

Restart the Quiz

In most personality quizzes, the player goes through all the questions only once. After the results have been displayed, players shouldn't have a way to go back and change previously answered questions to try and achieve a different outcome. Unfortunately, the Back button on the result screen implies that they can do that. To hide the Back button in the navigation bar, add the following line of code to the bottom of `viewDidLoad()` for `ResultsViewController`:

```
navigationItem.hidesBackButton = true
```

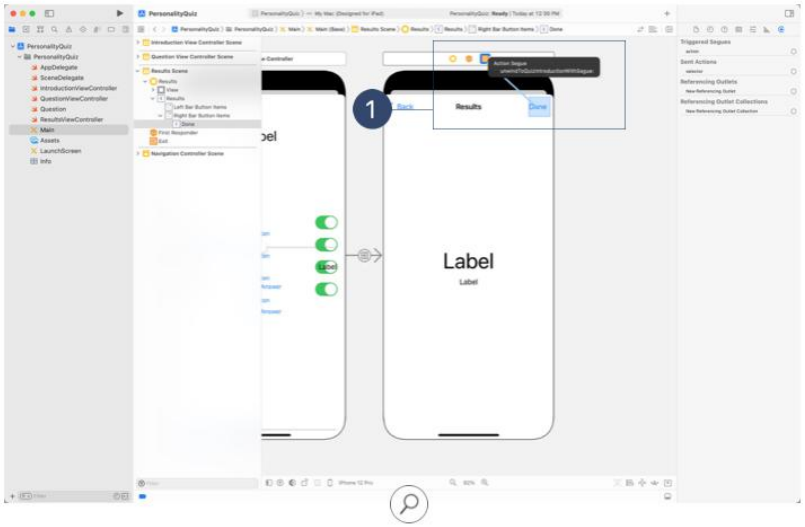
Instead of changing previous responses, the player should be able to dismiss the results and start with a clean slate. A tap of the Done button can return to the `IntroductionViewController`, making it very clear that the quiz is over. But at the moment, the Done button doesn't connect to any sort of action.

You'll need to create an unwind method in the first view controller. Add the following to the `IntroductionViewController` definition:

```
@IBAction func unwindToQuizIntroduction(segue:
UIStoryboardSegue) {

}
```

Since the app doesn't need to retain or pass back any information when the `ResultsViewController` is dismissed, you can leave the method body blank.

In the storyboard, **Control-drag** from the Done button to the Exit button at the top of the view controller. Select the `unwindToQuizIntroductionWithSegue` option that appears in the popover. [1]



Now when the player taps the Done button, the unwind segue will dismiss the view controllers that were created after the `IntroductionViewController` was displayed. This includes both the `QuestionViewController` and the `ResultsViewController`.

Wrap-Up

Congratulations on building a custom personality quiz app for iOS!

This project was intense, and there was a lot going on with the interface. If you struggled to create stack views, add constraints, create outlets, or modify the attributes of view and controls, take some time to practice working with Interface Builder. It's one of your most valuable tools for building apps—and will reduce a lot of code you'd have to write if you created the interface programmatically.

If you had any issues with the Swift code, remember that you can always refer to the Xcode documentation for additional information about methods and properties—or refer to the previous unit that covered Swift syntax and data types.

Stretch Goals

Would you like to extend your personality quiz further? Are there steps you'd like to improve on? Here are some ideas that will make the questions more dynamic and add replay value to your app:

- Allow the player to choose between multiple personality quizzes from the introduction screen.
- Randomize the order in which the questions are presented, as well as the order of the answers.
- Allow single-answer and multiple-answer questions to have a dynamic number of answers, rather than always four. Hint: Rather than creating the controls in Interface Builder, you'll need to add/remove labels and controls from stack views programmatically.

Lesson 3.11

Evaluate Your App

In this unit, you've had the chance to reflect on how code and tools can be used to build parts of your prototype, such as the use of tab bars. But now it's time to test the prototype as you planned and iterate on your idea. With each iteration, you can make the prototype more real, more detailed, and closer to the final product. Then the only step left is to actually build your app!

In this lesson, you'll test your prototype and then evaluate the feedback to draw conclusions about how to best improve on your app idea and design.

What You'll Learn

- How to select participants for testing your app prototype
- How to synthesize feedback from your user tests
- How to develop next design and development steps

Related Resources

- WWDC 2017 Love at First Launch
- WWDC 2019 Great Developer Habits

Guide
Prepare Your Test

The final step in preparing to test your app prototype is deciding who to test with. The quality of your data depends on the users you test with, so it's important to select them carefully. And you'll want to make sure that you're ready right at the start to provide each participant with an enjoyable experience.

Use the Prepare section of the App Design Workbook to finalize your plans for testing your app prototype. Then go for it—test the app prototype!

Validate

You'll have a lot of information to digest after testing your prototype. It's important to summarize and draw the correct conclusions from your testing data so that you know how to improve your app.

Work through the Validate section of the App Design Workbook. You'll start by formatting your data to make it digestible. Then you'll summarize your observations by discovering relationships between them. Then you'll zoom out to root causes and identify core issues.

Iterate

Look closely at your first prototype and you'll see a world-changing app beginning to take form. Now comes the critical phase of any design—revising and looking for opportunities to make improvements, large and small.

Complete the Iterate section of the App Design Workbook to use the conclusions from your analysis as a guide to reevaluate choices you made throughout your app design journey.

And then you are ready to do two things—rework your app design, and then try to build your app using all the coding skills and tools you have been learning.

Guide
Prepare Your Test

The final step in preparing to test your app prototype is deciding who to test with. The quality of your data depends on the users you test with, so it's important to select them carefully. And you'll want to make sure that you're ready right at the start to provide each participant with an enjoyable experience.

Use the Prepare section of the App Design Workbook to finalize your plans for testing your app prototype. Then go for it—test the app prototype!

Validate

You'll have a lot of information to digest after testing your prototype. It's important to summarize and draw the correct conclusions from your testing data so that you know how to improve your app.

Work through the Validate section of the App Design Workbook. You'll start by formatting your data to make it digestible. Then you'll summarize your observations by discovering relationships between them. Then you'll zoom out to root causes and identify core issues.

Iterate

Look closely at your first prototype and you'll see a world-changing app beginning to take form. Now comes the critical phase of any design—revising and looking for opportunities to make improvements, large and small.

Complete the Iterate section of the App Design Workbook to use the conclusions from your analysis as a guide to reevaluate choices you made throughout your app design journey.

And then you are ready to do two things—rework your app design, and then try to build your app using all the coding skills and tools you have been learning.

Summary

Nice work! This unit covered a *lot* of ground.

You've learned how to work with optional data in Swift. You've also learned about navigation hierarchies and how to build simple workflows using tabs and navigation stacks.

With your working knowledge of Xcode, Swift, and `UIKit`, there are many new apps that you're now capable of building. In the next unit, you'll up-level your `UIKit` experience by learning about table views, and you'll tie together all these skills to build an app that allows the user to view, enter, and save information.

Addi

- Learn
- Learn
- Learn
- Find to
  Apple
- Conne
- Learn
- Check
- Check

Additional Resources

- Learn more about teaching code and the Develop in Swift program.
- Learn more about Swift Playgrounds.
- Learn more about Swift.
- Find tools and resources for creating apps and accessories for Mac, iPhone, iPad, Apple Watch, and Apple TV.
- Connect with other programmers in the Apple Developer Forum.
- Learn more about submitting your app to the App Store.
- Check out Develop in Swift Explorations.
- Check out Develop in Swift Data Collections.