

Develop in Swift

Fundamentals

Introduction

Have you ever had an idea for an app and wondered how to make it happen? If so, this course was designed for you. You'll start by focusing on iOS development tools, basic programming concepts, and industry best practices. Building on this foundation, you'll work through practical exercises, creating apps from scratch, and building the mindset of an app developer.

You'll build three projects—a simple flashlight app that changes the background color of the screen, a fill-in-the-blanks word guessing game, and a personality quiz. Throughout the course, you'll have the chance to design, prototype, and test an app of your own. As you build up your coding skills, you'll make connections to how you can apply what you are learning to bring your app idea to life.

You can work through *Develop in Swift Fundamentals* on your own, or you may be in a class with a teacher. If you're on your own, we recommend completing every lesson, lab, and guided project, to make sure you're building all the skills. If you have a teacher guiding you, keep in mind that they may use different parts of the course in different ways.



This course was designed for students with no prior programming experience. Prior experience with languages other than Swift will definitely give you a boost when learning the basics. And if you already know something about Swift, Xcode, and iOS development, you might want to jump straight into the labs and guided projects to practice your skills.

Course Structure And Content

At the core of Develop in Swift Fundamentals are three progressively challenging guided projects, each preceded by multiple lessons that cover the concepts and skills required to build the app. App design lessons are integrated throughout the course that explore how to develop and iterate on your own app ideas as well as how to create a prototype that can serve as a compelling demo and launch your project toward a successful 1.0 release.

About the Lessons

This course features 33 lessons that help you learn a specific skill related to Swift or app development. Each lesson starts with a brief introduction to the concept, a set of learning objectives, new vocabulary terms, and references to documentation used to build the lesson. The body of the lesson includes concept explanations, sample code, and screencasts. At the end of each lesson, a lab and review questions allow you to apply the concepts you've just learned and check your understanding.

Since Develop in Swift Fundamentals covers three very different types of content—Swift, app development, and app design—you'll see three different approaches to the lessons.



Swift

Swift lessons focus on specific concepts, and the labs for these are presented in playgrounds – an interactive coding environment that lets you experiment with code and see the results immediately.



App development

App development lessons cover the Software Development Kit, or SDK. These lessons focus on building specific features for iOS apps, usually guiding you through a mini-project. The labs for these guide you to apply what you learned in a new scenario.



App design

App design lessons use the app design cycle to build design skills to turn an idea into an app prototype in Keynote. You'll refine and improve your ideas over time, connecting what you are learning about code to bring your app idea to life.

About the Projects

Each guided project includes a description of user-centered features, a project plan, and step-by-step instructions that lead to a fully functioning app. Through these guided projects - as well as through labs sprinkled throughout the course - you will be able to customize features according to your interests. At the same time, you'll be performing the kind of work you can expect in an app development workplace.



Light

The first project is **Light**, a simple flashlight app. You'll learn the basics of data, operators, and control flow in the Swift programming language. You'll also learn about Xcode, Interface Builder, building and running an app, debugging, and documentation.



Apple Pie

The second project is **Apple Pie**, a word-guessing game. You'll learn about Swift strings, functions, structures, collections, and loops. You'll also learn about UIKit, the system views and controls that make up a user interface, and how to display data using Auto Layout and stack views.



Personality Quiz

The third project is **Personality Quiz**, a personalized survey that reveals a fun response to the user. You'll learn how to build simple workflows and navigation hierarchies using navigation controllers, tab bar controllers, and segues. You'll also learn about optionals and enumerations, two powerful tools in Swift.

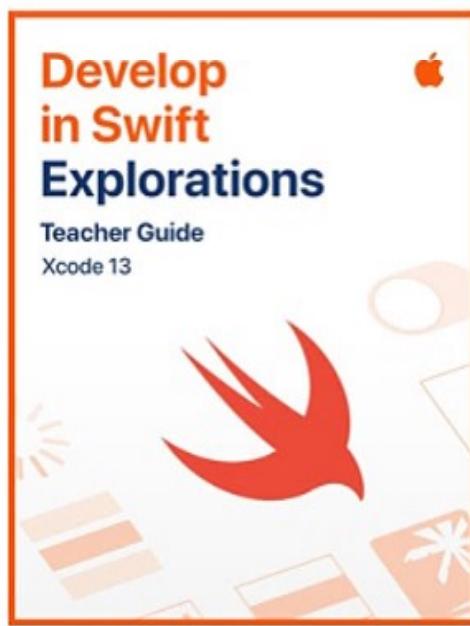
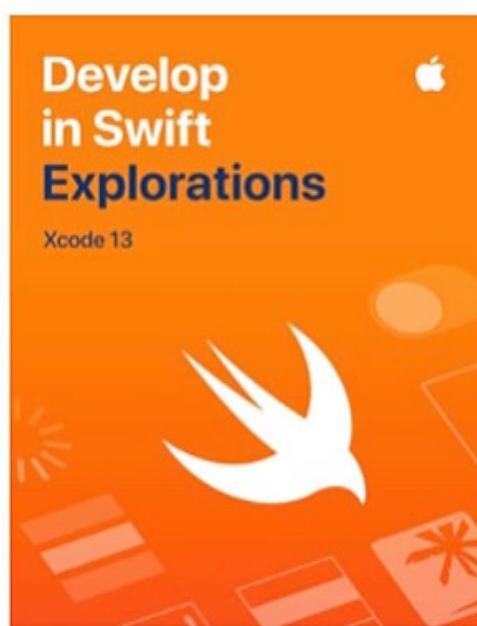
After you've built the guided projects, you'll learn how to design, prototype, and architect an app of your own.

Curriculum Pathway

Develop in Swift curriculum encourages students to solve real world challenges creatively through app development. Students build foundational knowledge with Explorations or Fundamentals courses then progress to more advanced concepts in Data Collections. All courses include free teacher guides to support educators—regardless of experience teaching Swift or other programming languages.

Explorations (One semester)

Students learn key computing concepts, building a solid foundation in programming with Swift. They'll learn about the impact of computing and apps on society, economies, and cultures while exploring iOS app development.



Unit 1: Values

Episode 1: The TV Club

Unit 2: Algorithms

Episode 2: The Viewing Party

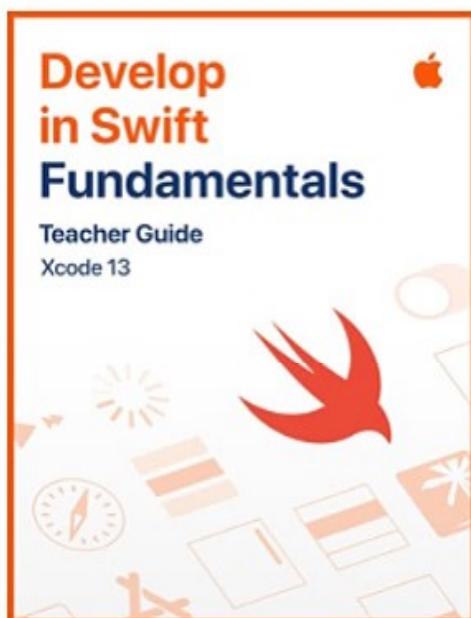
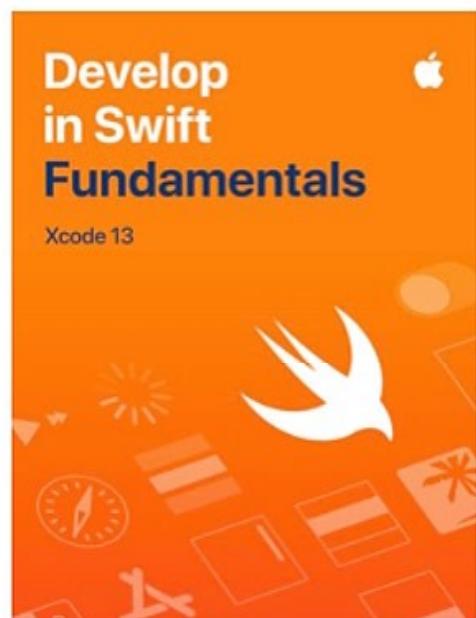
Unit 3: Organizing Data

Episode 3: Sharing Photos

Unit 4: Building Apps

Fundamentals (One semester)

Students build fundamental iOS app development skills with Swift. They'll master the core concepts and practices that Swift programmers use daily and build a basic fluency in Xcode source and UI editors. Students will be able to create iOS apps that adhere to standard practices, including the use of stock UI elements, layout techniques, and common navigation interfaces.



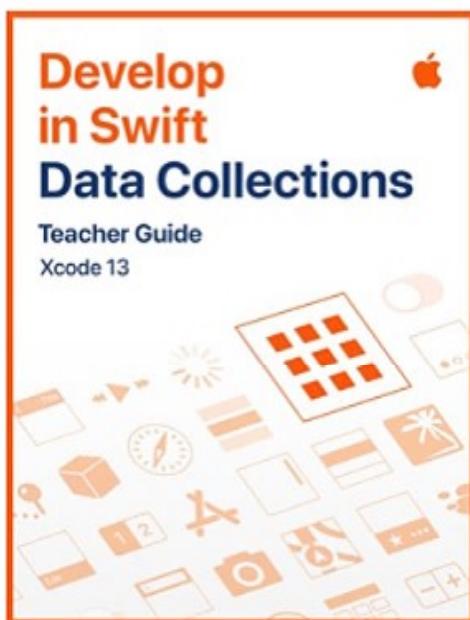
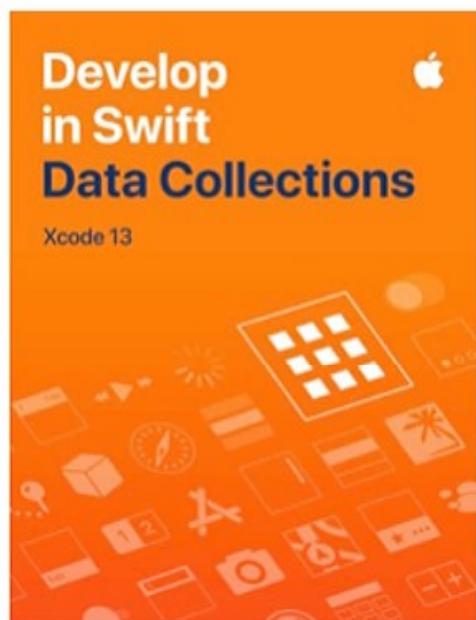
Unit 1: Getting Started with App Development

Unit 2: Introduction to UIKit

Unit 3: Navigation and Workflows

Data Collections (One semester)

Students expand on the knowledge and skills they developed in Fundamentals by extending their work in iOS app development, creating more complex and capable apps. They'll work with data from a server and explore new iOS APIs that allow for much richer app experiences—including displaying large collections of data in multiple formats.



Unit 1: Tables and Persistence

Unit 2: Working with the Web

Unit 3: Advanced Data Display

Set Up Your Learning Environment

Learning to build apps involves many tools and many resources. At any given time, you may have multiple projects and playgrounds open in Xcode—as well as this book, Xcode documentation, Safari, and some number of assets on your desktop. As you start to build apps, you’ll discover it’s important to keep your workspace organized.

It’s up to you how to navigate between applications. Some students like to use split-screen mode so they can keep all their tools in one single view. Others prefer to run each application (including this book) in full-screen mode and switch between applications as necessary.

To enter full-screen mode, click the green circle in the top left of the window or use the keyboard shortcut, **Control-Command-F**. You can then navigate between the full-screen applications using Mission Control, by swiping left or right with four fingers on the trackpad, or using the keyboard shortcuts, **Control-Left Arrow** and **Control-Right Arrow**.

Gather Your Materials

To complete the lessons in this guide, you’ll need the following:

- A Mac running macOS Big Sur or Monterey.
- Xcode 13, available on the Mac App Store.
- Project files for the course, which you can download here.

To access these materials in Xcode, you might need to enter the administrator name and password for your Mac.



Download student
materials

A Note About the App Design Workbook

As you work through the App Design Workbook, you'll find embedded coding activities focused on using SwiftUI in Swift Playgrounds. SwiftUI is another framework for building great apps. While this course focuses on the UIKit framework, some app developers choose to use a bit of both frameworks, since they have different strengths and weaknesses. While the App Design Workbook SwiftUI coding exercises are optional, if you choose to complete the activities they will help you expand your coding knowledge and skills.

A Word of Advice

Develop in Swift Fundamentals is designed to make Swift and iOS development approachable. But you will get stuck. All programmers get stuck.

Learning to program is hard. And building apps is hard. You'll feel discouraged when you can't get something to work just right. You'll feel frustrated when you've been stuck for hours on the same problem. And you may want to quit when you don't understand something.

But it gets easier. It turns into a puzzle. You'll experience a rush of adrenaline when you hit the Run button and your app works, especially after you've spent hours or days trying to get one little thing to work just right. You'll smile when you write code that runs perfectly on the first try. And you'll celebrate when your first app goes live on the App Store.

We're excited to see what you come up with.

Unit 1

Getting Started With App Development

Welcome to *Develop in Swift Fundamentals*. By learning the basics of the Swift programming language, you'll be on the fast track to developing your own apps.

This first unit introduces you to the basics of Swift, building modern mobile apps, iOS, Xcode, and other tools in the Xcode development environment. You'll also learn a bit about Interface Builder, a visual tool for crafting user interfaces.

After completing this unit, you'll be familiar with everything you need to tackle your first app, and you'll have defined an idea for your own app.



Swift Lessons

- Introduction to Swift and Playgrounds
- Constants, Variables, and Data Types
- Operators
- Control Flow



SDK Lessons

- Xcode
- Building, Running, and Debugging an App
- Documentation
- Interface Builder Basics

What You'll Design

The App Design Workbook will guide you through defining your own app idea for a specific audience.

What You'll Build

Light is a simple full-screen flashlight app, where the user taps the screen to toggle its color between black and white.

Lesson 1.1

Start Defining Your App

As an app developer, you're challenged with creating a thing that people didn't have before and maybe didn't even know they needed. To do this, before you even jump into coding you need a vision and purpose for your app that you can continually return to.

In this lesson, you'll start to define your app. An initial discovery process will help you identify the challenge you want to solve and understand your audience. Then you'll analyze how an app can tackle the challenge. You'll use Keynote templates in the App Design Workbook to track your app ideas.

What You'll Learn

- How to identify the challenge you want to solve
 - How to explore and gain a better understanding of your audience and their needs
 - How to analyze the causes of the problem and your solution compared to competitors
-

Related Resources

- [WWDC 2019 Designing Award Winning Apps and Games](#)
- [WWDC 2020 Design for Intelligence: Meet People Where They Are](#)
- [Human Interface Guidelines: Inclusion Overview](#)

Guide

App Design Workbook

Begin by opening the App Design Workbook and familiarizing yourself with the layout. Read the Welcome pages to learn about how to use the workbook, Keynote, and the other resources you will need. You'll use the workbook to document your app design process and ideas throughout the course, so save it in an easy-to-find place on your device. Also, remember that this course focuses on UIKit, so the SwiftUI coding exercises in the App Design Workbook are optional.

Use the App Design Workbook to complete the following exercises.

Discover

Think about your favorite app. What's its purpose? What problem(s) does it solve? Assuming you're thinking about a well-designed app, it's probably fairly easy to answer these questions. Many great apps focus on one particular type of user and one or two issues—and therefore have a very specific set of goals.

Use the Discover section of the App Design Workbook to begin identifying a challenge and the people it affects. By the end of this part of the workbook, you'll have a thorough understanding of a challenge and an insight into the people who would benefit from a solution.

Analyze

Now it's time to dig a little deeper. The more you understand the issue and audience, the more targeted the app will be—and the more likely it is that it will solve a problem that a group of users experiences.

Complete the Analyze section of the App Design Workbook. By the end of this stage, you'll have a clearer picture of the form your app might take. You'll look at the root causes of your users' challenges, and you'll use them to drive feature ideas that take advantage of key iOS capabilities while contrasting your ideas with existing apps.

Coming Up

As you build your coding knowledge in the rest of this unit, you'll regularly return to your app idea and think about how you might use what you are learning to make the app you care about come to life. Keep your App Design Workbook handy so that you can refer back to it and add notes to keep your app design ideas and plans moving forward.

Lesson 1.2

Introduction to Swift and Playgrounds

In this lesson, you'll learn about the origin of Swift and some of its basic syntax.

What You'll Learn

- Why Swift is a great language to learn
- How to use Xcode playgrounds to run Swift code

Vocabulary

- [console](#)
- [open source](#)
- [playground](#)
- [results sidebar](#)

Related Resources

- [Swift Programming Language Guide: About Swift](#)

A Little History

At the Apple Worldwide Developers Conference 2014, Apple introduced Swift as a modern language for writing apps for iOS and macOS. Apple now has new platforms, including watchOS and tvOS, that also use Swift as the primary programming language.

Since the 1990s, most developers have written applications for Apple platforms in Objective-C, a language built on top of the C programming language. Objective-C is more than 30 years old, and C is more than 40 years old. Both languages have served the software developer community well. They won't be going away in the foreseeable future.

However, Objective-C can be challenging to learn. Because technology has been advancing so fast in recent years, Apple saw the opportunity to create a more modern language that was easier to learn and easier to read, write, and maintain.

As you learn Swift, you may see the influence of its C and Objective-C heritage.

A Modern Language

What's a modern language? It's one that's safe, fast, and expressive. As Apple was designing Swift, those three primary goals were at the core of every decision. As you learn programming concepts in Swift, you'll come to appreciate how each decision points to safety, speed, and clarity.

Some of the features that make Swift a modern language include:

- Clean syntax, which makes code readable and easier to work with
- Optionals, a new way of expressing when a value may not exist
- Type inference, which speeds up development and allows the compiler to help identify common issues
- Type safety, which enforces code that's less likely to crash your program
- Automatic Reference Counting (ARC) for memory management, which automatically handles some of the deeper technical challenges of native programming
- Tuples and multiple return values, which allow smaller units of code to do more
- Generics, which help developers write code that can be used in multiple scenarios
- Fast and concise iteration over collections, making Swift a fast language
- Structs that support methods, extensions, and protocols, which allow Swift to optimize for memory use and speed while providing flexibility for developers
- Map, filter, reduce, and other functional programming patterns, which simplify code and streamline common actions that previously required multiple lines of code
- Powerful error handling, which helps the developer write fewer bugs and better handle scenarios that could cause apps to crash or perform unexpectedly

At the moment, you're probably not familiar with most of the concepts on the list. That's OK. For now, just realize that they make Swift a great language to use for building applications. As you progress through this course, you'll learn what each of these features means and how it's relevant to writing safe, fast, and expressive code.

A Safe Language

A number of features already mentioned make Swift a safe language by helping you write code that is less likely to crash your app. Computer programs are very literal, and so code written to handle one thing may not be capable of handling another. Type safety forces you to be explicit about the “type” of each object that you create, manipulate, and assign, and only lets you write code that the given object can handle. This prevents you from writing code that may crash if it is not designed to work with the “types” of objects you are referencing. Type inference similarly allows the compiler to infer the type of an object, thereby saving you time and again ensuring that the compiler can enforce proper rules regarding what operations and functions can be performed with each type.

Optionals are somewhat unique to Swift, and allow you to better express when a value may not exist. This helps you ensure that your code can handle both scenarios where values exist and scenarios where they do not. Swift also provides for sophisticated error handling, which as the name implies, allows you to write code that can handle errors gracefully and simply.

Open Source

In December 2015, Apple released the Swift language and supporting resources on [GitHub](#) as an open source project. Open source can mean a lot of things. The most important point to understand is that an open source language is developed in the open, with community input and support. Everyone is welcome to contribute or simply to follow along.

Open development means that Swift is evolving and improving. Over time, syntax may change and support will be added for more platforms, including delivering more Swift technology to Linux. Now that you’re learning Swift, you can be confident that your knowledge will become more and more valuable as the language continues to improve and as adoption grows across Apple platforms and beyond.

For more information about how to follow along or participate in building the Swift language, visit the project home page at [Swift.org](#).

Hello World

When you start with a new programming language, it's a time-honored tradition to build a "Hello, world!" app, one of the simplest programs you can write in any language. All this program does is print "Hello, world!" to the screen.

Swift code is written in plain text files with a `.swift` file extension. Each line in the file represents a statement, and a program is made up of one or more statements. These are the instructions you wish your app to run. Generally, code is executed starting at the top of the file and works its way to the bottom of the file.

As you'll learn, you can control whether specific sections of code are executed using control flow statements (`if`, `else`), how many times they're executed using loops (`for-in`, `while`), and how to use data that can be passed to different statements.

Some programs are made up of millions of statements spread across thousands and thousands of files. The compiler makes the code executable by combining all the files into a program.

But for now, let's keep it simple.

In Swift, the default file is called `main.swift`. Any Swift code included in the `main.swift` file will be executed from top to bottom.

So how would you write a "Hello, world!" app in Swift?

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

As you might guess from the code above, `print()` is a function (a smaller unit of code that performs a specific task) that takes a bit of text, called a `String`, and prints it to the console.

The console is a text-entry and display tool for system administration. Decades ago, all computer interactions were through consoles, or command line interfaces. Today we have graphical user interfaces for most tasks, but developers still use the console to perform many programming or administrative tasks. In fact, almost every system you use has a console behind the scenes, which you may be able to access even when you're using the graphical interface.

Terminal

How do you access the console? macOS comes with a console app called Terminal, and Swift comes with a tool called a REPL, which stands for Read, Eval, Print Loop. The REPL allows you to enter simple commands, evaluate them, and print the result.

Use the Swift REPL in the console to write your first “Hello, world!” program.

1. Open the Terminal application on your Mac. You can search “Terminal” in Spotlight or find the application in the system Applications/Utilities folder.
2. Enter the Swift REPL by typing `swift` and pressing Return.
3. Type the command `print("Hello, world!")` and press Return to execute it.

Note that “Hello, world!” was printed onscreen just below your `print` command. If you are entirely new to programming, this may be the first time you have ever provided a computer with written instructions for it to execute. Congratulations!

You can now exit the Swift REPL and Terminal by doing the following:

1. Type `:quit` and press Return to exit the Swift REPL.
2. Quit Terminal.

Playground

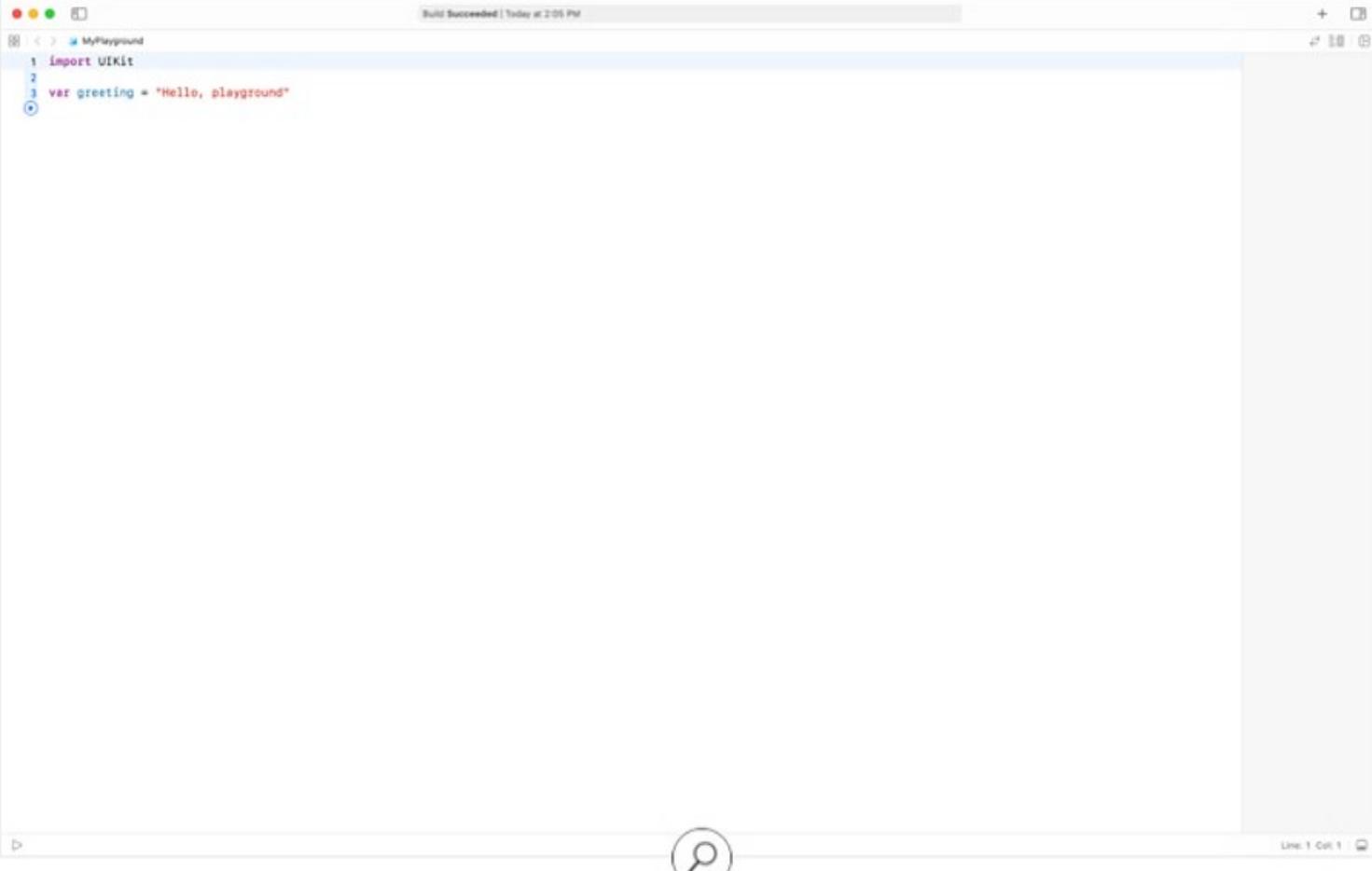
One of the most important things announced with Swift was playgrounds, a special Xcode document type that runs Swift code in a simple format with easily visible results.

Working in playgrounds is preferable to working in a REPL for many reasons. For starters, they make writing Swift code fun and simple. Using a more familiar interface, you can type in a line of code and the results appear immediately in the sidebar. If your code runs over a period of time, you can watch the progress in the timeline.

Playgrounds can also be extremely useful for Swift developers. They make it easy to manage more code in one place and to see multiple results in the results sidebar.

Developers often use playgrounds for prototyping their Swift code, then move it into an Xcode project after testing.

On a technical level, a playground is a file wrapper around a `main.swift` file. Every time you edit the code, the playground runs the results. You can include additional files and resources, which you'll do in a future activity.



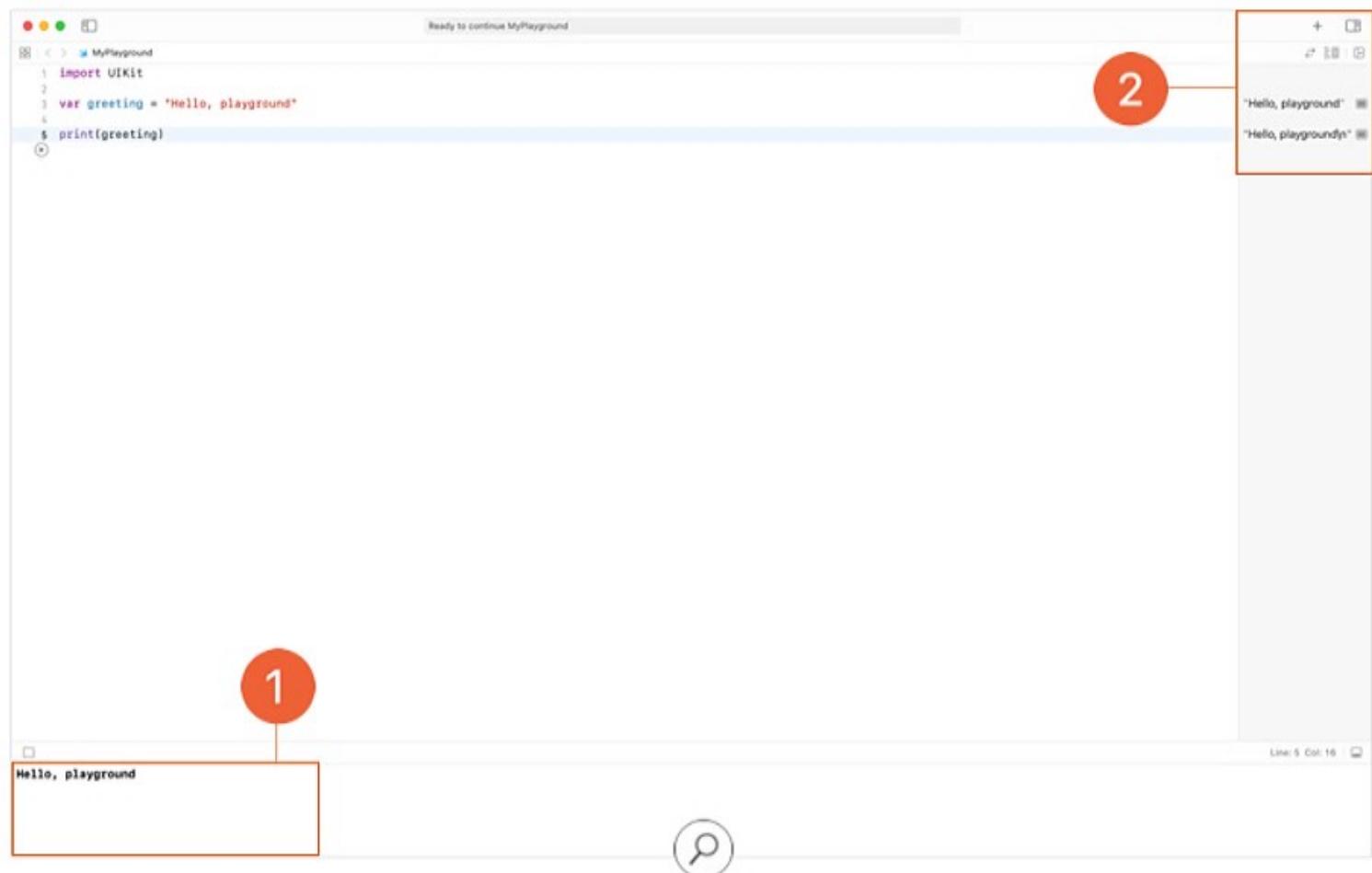
The screenshot shows the Xcode interface with a playground window titled "MyPlayground". The code editor contains the following Swift code:

```
import UIKit
var greeting = "Hello, playground"
```

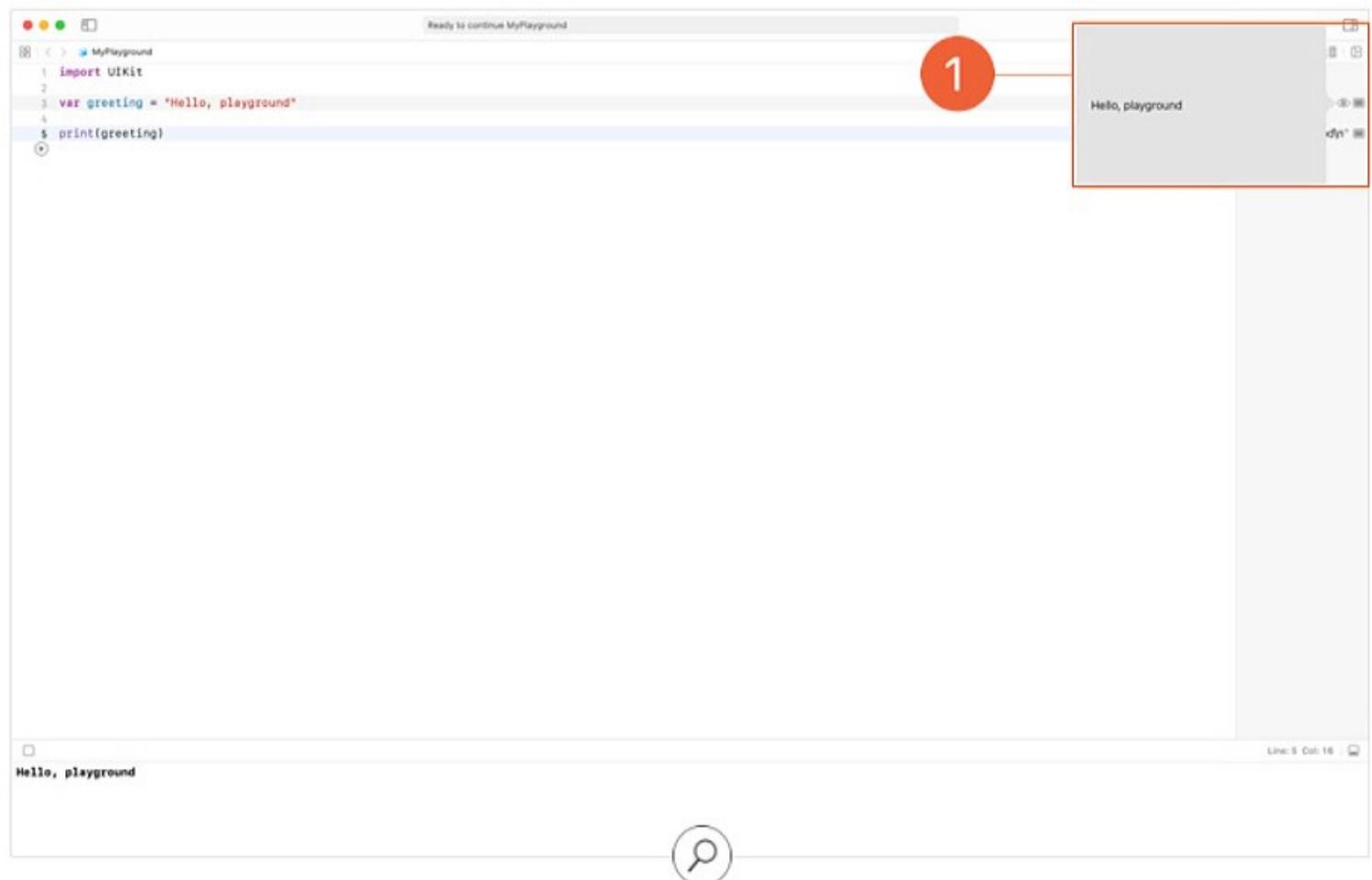
The status bar at the top right indicates "Build Succeeded | Today at 2:05 PM". A search bar with a magnifying glass icon is located at the bottom center of the window. The bottom right corner of the window displays "Line: 1 Col: 1".

To view printed results from your Swift code, you can open a debug console area in a playground. ①

Remember, every time you edit the Swift code, the code is run from top to bottom. For every expression, results are shown on the right side. ②



To view results that are “richer” than the plain text you see in the sidebar or debug area, you can use the Quick Look tool. ①

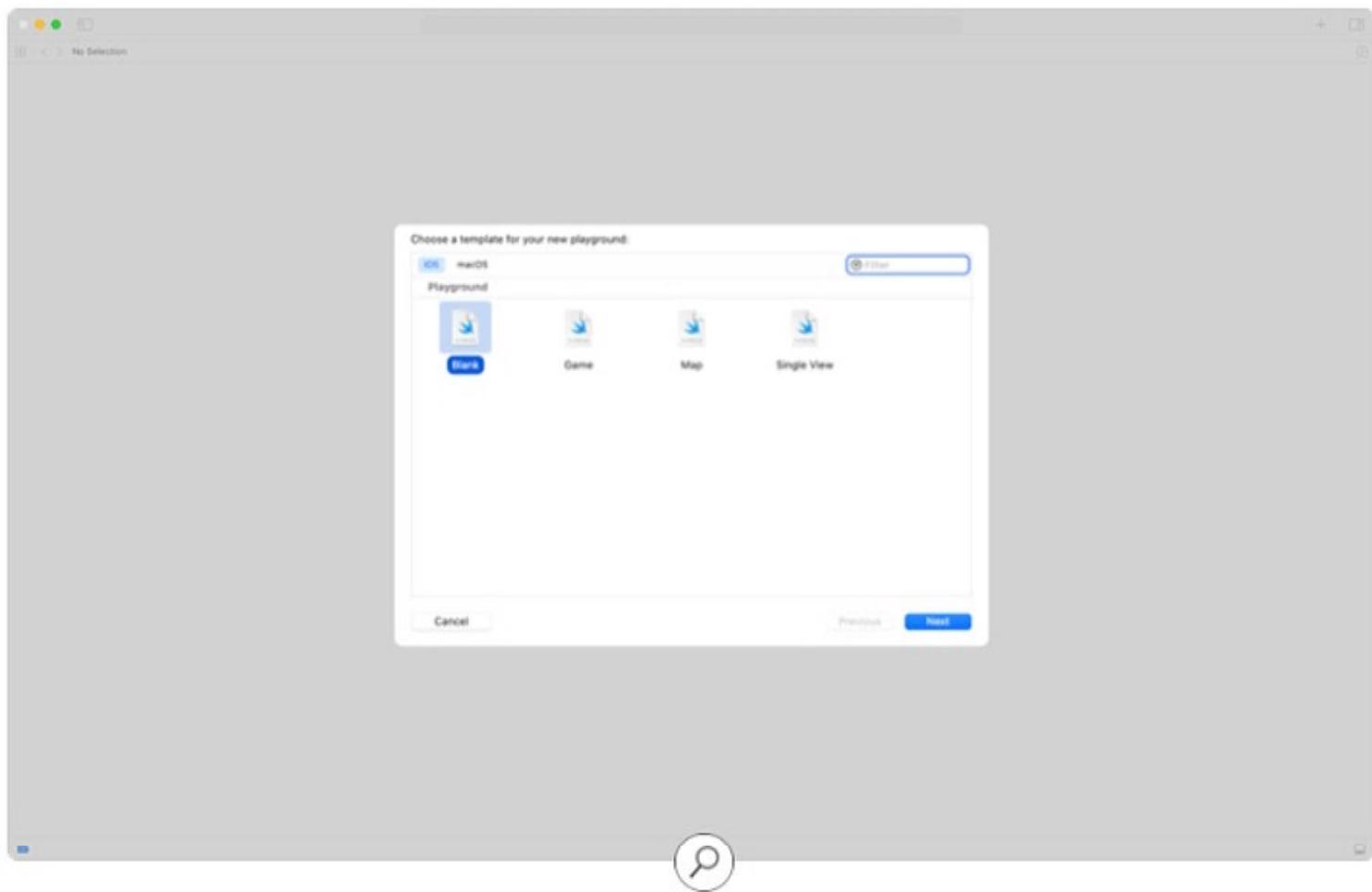


To view more details inline,^① you can use the Show Results button.^②

The screenshot shows the Xcode playground interface. In the top-left, there's a code editor window titled "MyPlayground" containing the following Swift code:

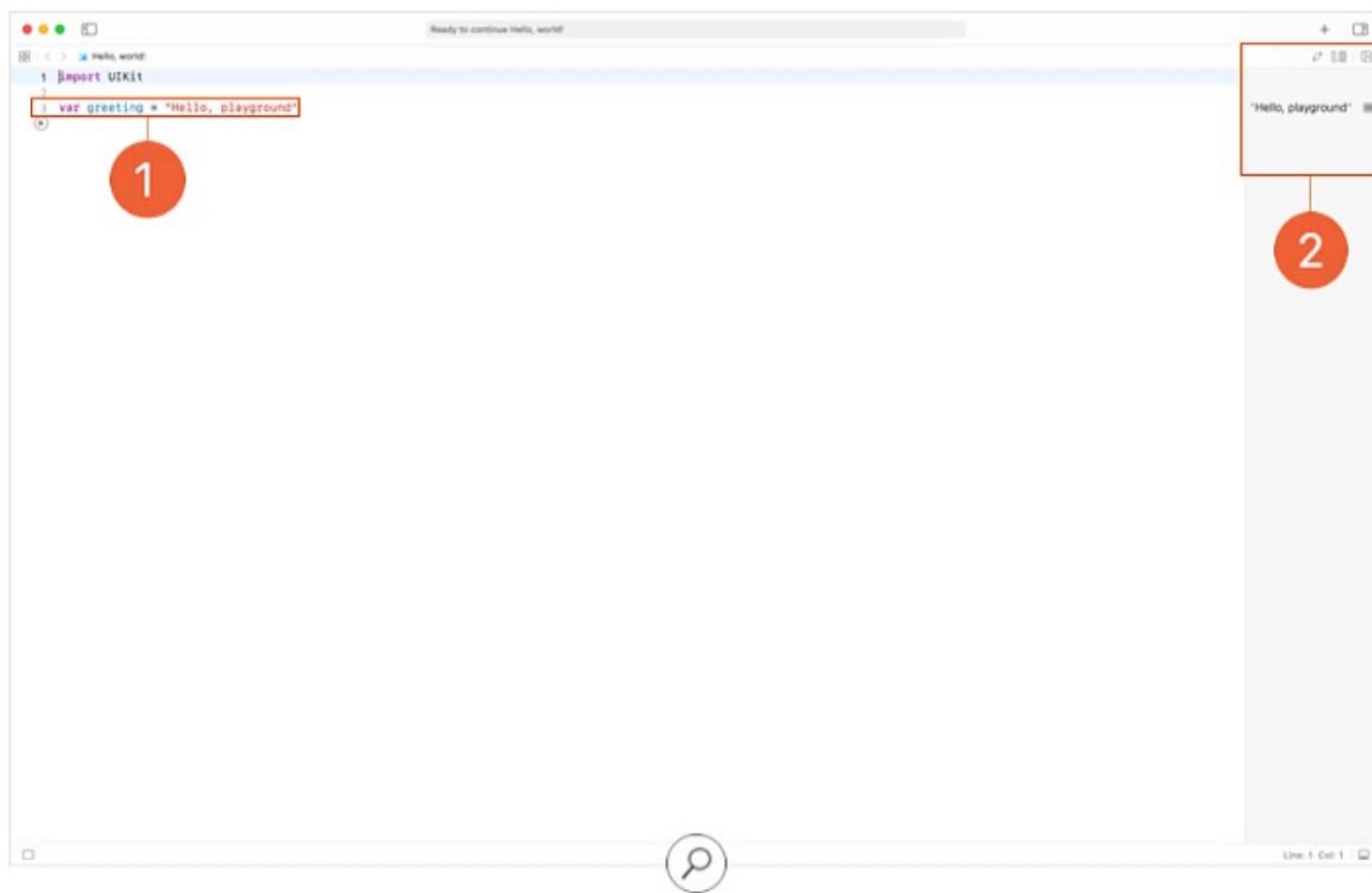
```
import UIKit
var greeting = "Hello, playground"
$ print(greeting)
Hello, playground
```

A red circle labeled "1" highlights the output "Hello, playground" in the code editor, which is enclosed in a red box. In the top-right corner of the Xcode interface, another red circle labeled "2" highlights the "Show Results" button, which is also enclosed in a red box. Below the code editor, a search bar with a magnifying glass icon is visible. At the bottom of the screen, the status bar displays "Hello, playground" and "Line: 5 Col: 16".



Now go and build your first “Hello, world!” using an Xcode playground.

1. Open Xcode.
2. Create a new playground file by choosing **File > New > Playground** from the menu bar.
3. Choose the iOS platform and the Blank template.
4. Name your playground “Hello, world!” and save it to your course resources folder.



Did you notice that the default playground comes with an `import UIKit` statement and the variable `var greeting = "Hello, playground"`? ^① Open the results sidebar (if it's not already displayed) and note that the string value is printed in the sidebar. ^②

The `import UIKit` statement allows you to use anything from the `UIKit` framework. This will become more important as you work through the labs in this course. You should typically leave the import statement in all of your playgrounds.

5. Earlier, using the Swift REPL, you wrote `print("Hello, world!")` to print "Hello, world!" to the console. Try that in the playground now. Start a new line and type `print("Hello, world!")`.

You should now see "Hello, world!" in the results sidebar and in the debug console.

What's going on in this playground? The code `var greeting = "Hello, playground"` in the playground file template created a variable named `greeting` and set it equal to a string of text containing "Hello, playground." Because the `greeting` value is a `String`, you can pass that value to the `print` function just like you did with "Hello, world!"

6. Update your code to print the `greeting` variable instead of your “Hello, world!” text by replacing “Hello, world!” with `greeting`.

Pause for just a moment and consider why this works. The `print` function takes a `String` parameter and prints it to the console. Both of these examples work because “Hello, world!” is a `String` and `greeting` is a variable that holds a `String`.

You’ll learn more about strings and variables in the next lesson.

Lab

Open and complete the exercises in [Lab—Introduction.playground](#).

Review Questions

Question 1 of 4

What are the three primary goals of Swift?

- A. Safe
- B. Accurate
- C. Fast
- D. Expressive

[Check Answer](#)



Lesson 1.3

Constants, Variables, and Data Types

Building apps, and programming in general, is all about working with data. As a developer, you'll need to understand how to handle and store data using clearly defined types.

In this lesson, you'll learn how constants and variables enable you to name pieces of data so they can be used later in your program. You'll learn to define constants for values that don't change and to define variables for values that do change.

You'll also learn about types, which are like blueprints for working with data. Different types can be used and stored in constants or variables. You'll learn what types are included in Swift and how Swift types can help you write better code.

What You'll Learn

- How to represent numbers, strings, and boolean values using native Swift data types
- When to use a constant and when to use a variable
- How to assign values to constants and variables
- How type inference helps you write clean code
- How type safety helps you write safe code

Vocabulary

- | | | |
|-----------------------------|----------------------------------|----------------------------|
| • Bool | • Int | • variable |
| • constant | • let | • var |
| • comment | • mutable | |
| • Double | • property | |
| • function | • type inference | |
| • immutable | • type safety | |

Related Resources

- [Swift Programming Language Guide: Constants and Variables](#)

Constants and variables associate a name (like `age` or `name`) with a value of a particular type (like the number `29` or the string “John”). You can then use the name to refer to that value many times in your program. Once it’s set, the value of a constant is immutable, which means that it *cannot* be changed. In contrast, the value of a variable is mutable, which means that it can be changed at any time.

By defining a constant or variable, you’re asking the compiler to allocate storage for the value in memory on the device that will run the program. You’re also asking it to associate the constant or variable name with the assigned value so you can access the value later.

Constants

When you want to name a value that won’t change during the lifetime of the program, you’ll use a constant.

You define constants in Swift using the `let` keyword.

```
let name = "John"
```

The code above creates a new constant called `name` and assigns the value “John” to the constant. If you want to access that value in later lines of code, you can use `name` to reference the constant directly. This quick reference is especially helpful when you have one value that you use many times in a program.

```
let name = "John"  
print(name)
```

This code would print “John” to the console.

Because `name` is a constant, you can’t give it a new value after assigning it. For example, the following code won’t run:

```
let name = "John"  
name = "James"
```

Variables

When you want to name a value that may change during the lifetime of the app, you'll use a variable.

You define variables using the `var` keyword.

```
var age = 29  
print(age)
```

The code above would print `29` to the console.

Because `age` is a variable, you can assign a new value in later lines of code. The following code compiles with no errors.

```
var age = 29  
age = 30  
print(age)
```

This code would print `30` to the console.

You can assign constants and variables from other constants and variables. This functionality is useful when you want to copy a value to one or more other variables.

```
let defaultScore = 100
var playerOneScore = defaultScore
var playerTwoScore = defaultScore

print(playerOneScore)
print(playerTwoScore)

playerOneScore = 200
print(playerOneScore)
```

Console Output:

```
100
100
200
```

Constant Or Variable?

You just learned to use a constant when the value won't change and a variable when the value might change.

But there's a nuance here that's worth learning. Even though certain values might be variable, you can represent them with a constant because they won't change during the lifetime of a *single execution* of the code.

Imagine you're calculating information about the distance traveled on a trip. The program can be reused to track many trips over time, but it tracks one trip at a time. How would you represent the following data in your code?

- **Starting location**—This is a GPS coordinate of where you started your trip. Once you begin tracking a trip in your program, the location won't change. You'll represent this value with a constant.
- **Destination**—This GPS coordinate is where you want to arrive. Your app can be used for many destinations, so you may think this would be represented with a variable. But once your program begins tracking a trip, the destination won't change. You'll represent this value with a constant.
- **Current location**—The GPS coordinate of your current location will change whenever you move. So you'll represent this value with a variable.
- **Distance traveled**—How far have you traveled from your starting point? This value changes as you move. You'll represent it with another variable.
- **Remaining distance**—How far must you travel to arrive at your destination? The remaining distance changes as you move. You'll represent this value with a variable.

Constants and variables perform very similar jobs. You may think it would be easier to use variables for everything and ignore constants altogether. Technically your code *could* work this way. So why should you use constants at all?

First, if you set a value to a constant, the compiler understands that it should never be changed. This means you won't be able to build or run your program if you try to change the constant's value. In this way, the compiler enforces safety.

Second, there are special optimizations that the compiler can make for constant values. When you use constants for values that won't change, the compiler can make low-level assumptions about how to store the value. These adjustments allow your program to execute faster.

Third, it's the idiomatic, or accepted, way to do things in Swift.

As you learn more about building software, you'll come to appreciate best practices for consistently writing clean, readable code. This is especially important when working on a team with other developers. You'll also discover that following best practices will make it easier for you to remember what your code does, especially when you want to make changes later on.

If you want to be a great developer, you'll need to follow shared patterns and conventions. This course will guide you toward the best practices for writing Swift.

Naming Constants And Variables

There are rules for naming constants or variables. The compiler enforces the rules, so you won't be able to build and run your program if you break them.

- Names can't contain mathematical symbols.
- Names can't contain spaces.
- Names can't begin with a number.

Other than these three rules, you can name a constant or variable whatever you like.

```
let π = 3.14159
let 一百 = 100
let 🎲 = 6
let mañana = "Tomorrow"
letanzahlDerBücher = 15 // numberOfBooks
```

In addition to the rules, there are some best practices for naming constants and variables:

- Constant and variable names should be clear and descriptive, making it easy to understand the code when you come back to it later. For example, `firstName` is better than `n` and `restaurantsNearCurrentCity` is better than `nearby`.
- To be clear and descriptive, you'll often want to use multiple words in a name. When you put two or more words together, the convention is to use *camel case*. Lowercase the first letter in the name, and then capitalize the first letter of each new word. You've probably noticed examples of this: `defaultScore` is the camel case treatment for combining `default` and `score`. Camel case is easier to read than all the words compressed together. For example, `defaultScore` is clearer than `defaultscore` and `restaurauntsNearCurrentCity` is clearer than `restaurantsnearcurrentcity`. Camel case also allows assistive devices to better delineate word boundaries.

Comments

Did you notice that the translation for `anzahlDerBücher` was written off to the side? As code becomes more complex, it can be useful to leave little notes for yourself, as well as other developers who might read your code. These *comments* are created by placing two forward slashes in front of the text. When your code is compiled, the comments will be ignored, so write as many as you find helpful.

```
// Setting pi to a rough estimate  
let π = 3.14
```

If you need multiple lines for your comment, you can also place as much text as you'd like between `/*` and `*/` and it will be ignored by the Swift compiler.

```
/* The digits of pi are infinite,  
so instead I chose a close approximation.*/  
let π = 3.14
```

Comments are most frequently used to explain difficult sections of code, provide copyright information at the top of a file, and to provide dating information (when the file was created and/or modified).

Types

Each constant or variable in Swift has a *type* that describes its kind of value. In the examples you've seen, `29` is an integer that's represented by the Swift type `Int`. Likewise, "John" is a character string represented by the Swift type `String`.

Swift includes many predefined types you can use to more easily write clean code. Whether your program needs to include numbers, strings, or Boolean (true or false) values, you can use types to represent a specific kind of information.

Here are a few of the most common types in Swift.

Name	Type	Purpose	Example
Integer	<code>Int</code>	Represents whole numbers, or integers	4
Double	<code>Double</code>	Represents numbers requiring decimal points, or real numbers	13.45
Boolean	<code>Bool</code>	Represents true or false values	<code>true</code>
String	<code>String</code>	Represents text	"Once upon a time..."

Swift also supports collection types, which group multiple values into a single constant or variable. One collection type is named an `Array`, which stores an ordered list of values. Another collection type is named a `Dictionary`, which has keys that help you look up specific values. You'll learn more about collections in a future lesson.

The Swift standard library includes all the types above, so you can always use them.

In addition, you can define your own types in Swift by creating a *type definition*. Consider a simple Person type:

```
struct Person {  
    let firstName: String  
    let lastName: String  
  
    func sayHello() {  
        print("Hello there! My name is \(firstName) \(lastName).")  
    }  
}
```

You can think about a type definition as a blueprint. A blueprint designates how to construct a building but isn't a building itself. However, you can create many buildings from a single blueprint.

A type definition declares the information it stores (properties) and its capabilities or actions (methods). In this case, a `Person` stores `String` information in two properties, `firstName` and `lastName`, and has one action, the method `sayHello()`. You're probably unfamiliar with all the syntax above, but that's OK.

The example above describes how the type `Person` should look and perform. When you create a type and assign it to a variable or constant, you're creating a version or *instance* of the type. Thus, an instance is a value. Consider the following code:

```
let aPerson = Person(firstName: "Jacob", lastName: "Edwards")
let anotherPerson = Person(firstName: "Candace", lastName:
"Salinas")

aPerson.sayHello()
anotherPerson.sayHello()
```

Console Output:

```
Hello there! My name is Jacob Edwards.
Hello there! My name is Candace Salinas.
```

This code creates two instances of the `Person` type. One instance represents a `Person` named Jacob Edwards, and the other instance represents a person named Candace Salinas.

You'll learn more about defining types in a future lesson.

Type Safety

Swift is considered a type-safe language. Type-safe languages encourage or require you to be clear about the types of values your code can work with. For example, if part of your code expects an `Int`, you can't pass it a `Double` or a `String`.

When compiling your code, Swift performs *type checking* on all your constants and variables and flags any mismatched types as errors. If you mismatch types, you can't run your program.

```
let playerName = "Julian"  
var playerScore = 1000  
var gameOver = false  
  
playerScore = playerName  
// Will be flagged for mismatched types, will not compile.
```

Because type safety enforces the type a constant or variable can hold, assigning a `String` to a variable that only holds an `Int` doesn't make sense. You'll learn how Swift figured out the types shortly.

Type safety also applies to variables, constants, and literal values that represent data that may seem compatible. For example, Swift recognizes `Int` and `Double` as completely different types, even though they both represent numbers.

```
var wholeNumber = 30  
var numberWithDecimals = 17.5  
  
wholeNumber = numberWithDecimals  
// Will be flagged for mismatched types, will not compile.
```

In the case above, both variables are numeric types, but `wholeNumber` will be an `Int` and `numberWithDecimals` will be a `Double`. In Swift, you can't assign a value of one type to a variable of another type.

When working with numbers, you may find that you need to assign a very large value to a variable or constant. This can be difficult to read, since it is hard to see how many zeros there are in 1000000000. Fortunately, Swift allows you to put underscores in your numbers as a way of formatting for easier reading.

```
var largeUglyNumber = 1000000000
var largePrettyNumber = 1_000_000_000
```

Type Inference

You may have noticed that you don't have to specify the type of value when you declare a constant or variable. This is called *type inference*. Swift uses type inference to make assumptions about the type based on the value assigned to the constant or variable.

```
let cityName = "San Francisco"
// "San Francisco" is obviously a `String`, so the compiler
automatically assigns the type of cityName to a `String`.

let pi = 3.1415927
// 3.1415927 is a number with decimal points, so the compiler
automatically assigns the type `pi` to a `Double`.
```

Once you assign a value to a constant or variable, the type is set and can't be changed. This is true even for variables. The *value* of a variable may change, but not its type.

You might find cases where it's useful, or even required, to explicitly specify the type of a constant or variable. This is referred to as a *type annotation*. To specify a type, add a colon (:), a space, and the type name following the constant or variable name.

```
let cityName: String = "San Francisco"
```

```
let pi: Double = 3.1415927
```

You can use a type annotation with different number types. The Swift compiler will adjust the value to match the type.

```
let number: Double = 3
print(number)
```

Console Output:

3.0

There are three common cases for using type annotation:

1. When you create a constant or variable but haven't yet assigned it a value.

```
let firstName: String  
//...  
firstName = "Layne"
```

2. When you create a constant or variable that could be inferred as more than one type.

```
let middleInitial: Character = "J"  
// "J" would be inferred as a `String`, but we want a  
`Character`  
  
var remainingDistance: Double = 30  
// `30` would be inferred as an `Int`, but the variable should  
support decimal numbers for accuracy as the number decreases.
```

3. When you write your own type definition.

```
struct Car {  
    var make: String  
    var model: String  
    var year: Int  
}
```

Required Values

Whenever you define a constant or variable, you must either specify a type using a type annotation or assign it a value that the compiler can use to infer the type.

```
var x  
// This would result in an error.
```

Even when you use a type annotation, your code can't work with a constant or variable if you haven't yet assigned it a value.

```
var x: Int  
print(x)  
// This would result in an error.
```

Once the value is assigned, the constant or variable becomes available.

```
var x: Int  
x = 10  
print(x)
```

Lab

Open and complete the exercises in [Lab—Constants and Variables.playground](#).

Connect To Design

Open your App Design Workbook and review the Define sections you have completed. Although you aren't to the planning phase yet, you can start to think about your app plan now. Add comments to the Define sections you have completed or in a new blank slide at the end of the document. Reflect on what kinds of information your app might need to use. Are there different types you anticipate using—numbers, strings, Boolean values?

Try categorizing the information as constants or variables. Which data will change as you use the app, and which data will remain constant?

In the workbook's Go Green app example, the app might need to use `String` variables to keep track of the items a user recycles. It might also use a `Double` constant to represent the equivalent of 1 pound of plastic in pounds of carbon dioxide.

Review Questions

Question 1 of 7

Which of the following values would be best represented with a constant?

- A. Player name
- B. Player level
- C. Player score
- D. Player location

[Check Answer](#)



Lesson 1.4

Operators

 Operators are the symbols that make your code work. You'll use them to perform actions like check, change, or combine values. There are operators for working with and comparing numbers, operators for checking `true` and `false` values, operators that help you simplify conditional assignments, and many more.

In this lesson, you'll learn about some of the operators in the Swift language, including basic math operators for addition (+), subtraction (-), multiplication (*), and division (/).

What You'll Learn

- How to do basic mathematic operations
 - How to add two numbers of different types together
 - How to find the remainder of a division operation
-

Vocabulary

- [compound assignment](#)
 - [operator](#)
-

Related Resources

- [Swift Programming Language Guide: Basic Operators](#)

In programming, an operator makes your code work by performing an action on the values to its left and right. When you're reading code and see an unfamiliar symbol, chances are it's an operator.

This lesson covers the most common operators.

Assign A Value

Use the `=` operator to assign a value. The name on the left is assigned the value on the right.

This code declares that "Luke" is your favorite person:

```
let favoritePerson = "Luke"
```

The `=` operator is also used to modify or reassign a value.

The following code declares a `shoeSize` variable and assigns `8` as its value. The value is then modified to `9`:

```
var shoeSize = 8
shoeSize = 9 // Reassigns shoeSize to 9
```

Basic Arithmetic

You can use the `+`, `-`, `*`, and `/` operators to perform basic math functionality.

```
var opponentScore = 3 * 8
// opponentScore has a value of 24
var myScore = 100 / 4
// myScore has a value of 25
```

You can use operators to perform arithmetic using the values of other variables:

```
var totalScore = opponentScore + myScore // totalScore has a  
value of 49
```

An operator can even reference the current variable, updating it to a new value:

```
myScore = myScore + 3 // Updates myScore to the current value  
plus 3
```

For decimal-point precision, you can do the same operations on Double values:

```
let totalDistance = 3.9  
var distanceTraveled = 1.2  
var remainingDistance = totalDistance - distanceTraveled  
// remainingDistance has a value of 2.7
```

When you use the division operator (/) on Int values, the result will be an Int value rounded down to the nearest whole number, because the Int type supports whole numbers:

```
let x = 51  
let y = 4  
let z = x / y // z has a value of 12
```

If you explicitly declare constants or variables as Double values, the results will include decimal values.

```
let x: Double = 51  
let y: Double = 4  
let z = x / y // z has a value of 12.75
```

Make sure to use Double values whenever your code requires decimal-point accuracy.

Compound Assignment

An earlier code snippet updated a variable by adding a number to itself:

```
var myScore = 10  
myScore = myScore + 3  
// Updates myScore to the current value plus 3
```

But there's a better way to modify a value that's already been assigned. You can use a compound assignment operator, which adds the `=` operator after an arithmetic operator.

```
myScore += 3 // Adds 3 to myScore  
myScore -= 5 // Subtracts 5 from myScore  
myScore *= 2 // Multiples myScore by 2  
myScore /= 2 // Divides myScore by 2
```

Compound assignment operators help you write cleaner, more concise code.

Remainder Operator

Use the remainder operator (`%`) to quickly calculate the remainder from the division of two `Int` values.

```
let dividend = 10  
let divisor = 3  
let quotient = dividend / divisor // quotient has a value of 3  
let remainder = dividend % divisor // remainder has a value of 1
```

You can calculate the remainder as `dividend - (quotient * divisor)`, but the remainder operator is easier, faster, cleaner, and more concise.

Order Of Operations

Mathematic operations always follow a specific order. Multiplication and division have priority over addition and subtraction, and parentheses have priority over all four.

Consider the following variables.

```
var x = 2  
var y = 3  
var z = 5
```

Then consider the following calculations:

```
x + y * z // Equals 17  
(x + y) * z // Equals 25
```

In the first line above, multiplication takes precedence over addition. In the second line, the parentheses get to do their work first.

The remainder operator has the same precedence as multiplication and division:

```
y + z % x // Equals 4  
(y + z) % x // Equals 0
```

Numeric Type Conversion

As you've learned, you can't mix and match number types when performing mathematical operations.

For example, the following will produce a compiler error, because `x` is an `Int` value and `y` is a `Double` value:

```
let x = 3
let y = 0.1415927
let pi = x + y
```

To enable the code to finish, you can create a new `Double` value from `x` by prefixing the type you want to convert it to:

```
let x = 3
let y = 0.1415927
let pi = Double(x) + y
```

In the revised code, `Double(x)` creates a new `Double` value from the `Int` value `x`, enabling the compiler to add it to `y` and assign the result to `pi`.

Lab

Open and complete the exercises in [Lab—Operators.playground](#).

Connect To Design

In your App Design Workbook, reflect on the kinds of calculations that might connect to the problem your users have. What kinds of mathematical calculations might your app need to make? Will a user need to know a total, percent, or other calculated value? Even though you don't have a full plan yet, add your ideas in comments in the Define sections you have completed or in a new blank slide at the end of the document.

In the workbook's Go Green app example, a user might want to know the percentage of recycling in their total trash, which can be calculated by the app using operations such as division and addition.

Review Questions

Question 1 of 7

What is the value of `myNumber` at the end of the following code?

```
let x = 2  
let y = 4  
let z = 6  
let myNumber = x + y + z
```

- A. 4
- B. 6
- C. 10
- D. 12

[Check Answer](#)



Lesson 1.5

Control Flow

In the “Introduction to Swift and Playgrounds” lesson, you learned that a Swift program is written in a `.swift` file that’s executed from top to bottom. You saw this for yourself as you worked in playground files. But large applications aren’t so simple. They don’t fit into one file, and they usually don’t run every line of code from top to bottom.

You will write code that makes decisions about what lines of code should be executed depending on results of previously executed code. This is called control flow.

In this lesson, you’ll learn how to use logical operators to check conditions and to use control flow statements (`if`, `if-else`, and `switch`) to choose what code will be executed as a result.

What You'll Learn

- How to use `if` and `else` statements to control what code is executed
 - How to use the logical operators NOT (`!`), AND (`&&`), and OR (`||`) to check if something is true or false
 - How to use a `switch` statement to control what code is executed
 - How to use the ternary conditional operator (`? :`) to conditionally assign different values to a constant or variable
-

Vocabulary

- | | |
|--|---|
| <ul style="list-style-type: none">• comparison operator• conditional• if statement• if-else statement | <ul style="list-style-type: none">• interval matching• logical operators• switch• ternary conditional operator |
|--|---|
-

Related Resources

- [Swift Programming Language Guide: Control Flow](#)

Think of an application you use that requires you to log in. If you launch the application and you're already logged in, the app displays your data. If you're not logged in, the app asks for your login credentials. If you enter a correct user name and password, you're logged in and the app displays your data. If you enter incorrect credentials, you're asked to enter the correct information.

This example describes one common interaction that requires multiple checks and runs code based on the results.

These checks are called conditional statements, and they're part of a broader concept called control flow. As a developer, you have control flow tools that check for certain conditions and execute different blocks of code based on those conditions.

Based on the outcome of a set of conditions, you can use a variety of statements to control what code is executed. These conditional statements are all examples of control flow.

Logical And Comparison Operators

Each `if` statement uses a logical or comparison operator to decide if something is `true` or `false`. The result determines whether to run the block of code or to skip it.

Here's a list of the most common logical and comparison operators:

Operator	Type	Description
<code>==</code>	Comparison	Two items must be equal.
<code>!=</code>	Comparison	The values must not be equal to each other.
<code>></code>	Comparison	Value on the left must be greater than the value on the right.
<code>>=</code>	Comparison	Value on the left must be greater than or equal to the value on the right.
<code><</code>	Comparison	Value on the left must be less than the value on the right.
<code><=</code>	Comparison	Value on the left must be less than or equal to the value on the right.
<code>&&</code>	Logical	AND—The conditional statement on the left <i>and</i> right must be <code>true</code> .
<code> </code>	Logical	OR—The conditional statement on the left <i>or</i> right must be <code>true</code> .
<code>!</code>	Logical	NOT—Returns the logical opposite of the conditional statement immediately following the operator

You can mix and match operators to create a Boolean value, or `Bool`. Boolean values are either `true` or `false` and you can combine them with an `if` statement to determine if code should be run or skipped.

If Statements

The most straightforward conditional statement is the `if` statement. An `if` statement says that “If this condition is true, then run this block of code.” If the condition isn’t `true`, the program skips the block of code.

In most cases, you’ll use an `if` statement to check simple conditions with only a few possible outcomes. Here’s an example:

```
let temperature = 100
if temperature >= 100 {
    print("The water is boiling.")
}
```

Console Output:

The water is boiling.

The `temperature` constant is equal to `100` and the `if` statement prints text if `temperature` is greater than or equal to `100`. Because the `if` statement resolves to `true`, the block of code accompanying the `if` statement is executed.

If-Else Statements

You just learned that an `if` statement runs a block of code if the condition is `true`. But what if the condition isn't `true`? By adding an `else clause` to an `if` statement, you can specify a block of code to execute if the condition is *not* `true`:

```
let temperature = 100
if temperature >= 100 {
    print("The water is boiling.")
} else {
    print("The water is not boiling.")
}
```

You can take this idea even further. By using `else if`, you can declare more blocks of code to run based on any number of conditions. The following code checks for the position of an athlete in a race and responds accordingly:

```
var finishPosition = 2

if finishPosition == 1 {
    print("Congratulations, you won the gold medal!")
} else if finishPosition == 2 {
    print("You came in second place, you won a silver medal!")
} else {
    print("You did not win a gold or silver medal.")
}
```

You can use many `else if` statements to account for any number of potential cases.

Boolean Values

You can assign the results of a logical comparison to a `Bool` constant or variable to check or access the value later. `Bool` values can only be `true` or `false`.

In the following code, the `Bool` statement determines that `number` doesn't qualify as `isSmallNumber`:

```
let number = 1000
let isSmallNumber = number < 10
// number is not less than 10, so isSmallNumber is assigned
  a `false` value
```

And here, the `Bool` statement determines that `currentSpeed` does qualify as `isSpeeding`.

```
let speedLimit = 65
let currentSpeed = 72
let isSpeeding = currentSpeed > speedLimit
// currentSpeed is greater than the speedLimit, so
  isSpeeding is assigned a `true` value
```

It's also possible to invert a `Bool` value using the logical NOT operator, which is represented with `!`.

```
var isSnowing = false

if !isSnowing {
  print("It is not snowing.")
}
```

Console Output:

It is not snowing.

In the same way, you can use the logical AND operator, represented by `&&`, to check if two or more conditions are `true`.

```
let temperature = 70

if temperature >= 65 && temperature <= 75 {
    print("The temperature is just right.")
} else if temperature < 65 {
    print("It is too cold.")
} else {
    print("It is too hot.")
}
```

Console Output:

The temperature is just right.

In the above code, the temperature meets both conditions: It's greater than or equal to 65 degrees and less than or equal to 75 degrees. It's just right.

You have yet another option: the logical OR operator, represented by `||`, to check if either one of two conditions is `true`.

```
var isPluggedIn = false
var hasBatteryPower = true

if isPluggedIn || hasBatteryPower {
    print("You can use your laptop.")
} else {
    print("😢")
}
```

Console Output:

You can use your laptop.

This example prints “You can use your laptop.” Even though `isPluggedIn` is `false`, `hasBatteryPower` is `true`, and the `||` OR operator has determined that one of the options is `true`.

Switch Statement

You’ve seen `if` statements and `if-else` statements that you can use to run certain blocks of code depending on certain conditions. But a long chain of `if`, `else if`, `else if ... else` statements can become messy and difficult to read after a small number of options. Swift has another tool for control flow known as a `switch` statement that’s optimal for working with many potential conditions.

A basic `switch` statement takes a value with multiple options and allows you to run separate code based on each option, or `case`. You can also provide a `default` case to specify a block of code that will run in all the cases you haven’t specifically defined.

Consider the code that prints a name for a vehicle based on its number of wheels:

```
let numberOfWheels = 2
switch numberOfWheels {
    case 0:
        print("Missing something?")
    case 1:
        print("Unicycle")
    case 2:
        print("Bicycle")
    case 3:
        print("Tricycle")
    case 4:
        print("Quadcycle")
    default:
        print("That's a lot of wheels!")
}
```

Given a constant `numberOfWheels`, the code provides a separate action if the value is `0, 1, 2, 3, or 4`. It also provides an action if `numberOfWheels` is anything else.

This code *could* be written as a nested `if-else` statement, but it would quickly become tricky to parse.

Any given `case` statement can also evaluate multiple conditions at once. The following code, for example, checks whether a character is a vowel or not:

```
let character = "z"

switch character {
    case "a", "e", "i", "o", "u":
        print("This character is a vowel.")
    default:
        print("This character is a consonant.")
}
```

When working with numbers, you can use interval matching to check for inclusion in a range. Take a look at the syntax in the following code snippet:

```
switch distance {
    case 0...9:
        print("Your destination is close.")
    case 10...99:
        print("Your destination is a medium distance from here.")
    case 100...999:
        print("Your destination is far from here.")
    default:
        print("Are you sure you want to travel this far?")
}
```

The `switch` operator is the right tool for control flow when you want to run different code based on many different conditions.

Ternary Conditional Operator

An interesting (and very common) use case for an `if` statement is to set a variable or return a value. If a certain condition is `true`, you want to set a variable to one value. If the condition is `false`, you want to set the variable to a different value.

Consider the following code. It checks if the value of one number is greater than the other and assigns the higher value to a `largest` variable:

```
var largest: Int
```

```
let a = 15
```

```
let b = 4
```

```
if a > b {  
    largest = a  
} else {  
    largest = b  
}
```

Because this situation is so common in programming, many languages include a special operator, known as a ternary conditional operator (`? :`), for writing more concise code. Swift programmers generally leave out the word “conditional” and refer to this as simply the “ternary operator.”

The ternary operator has three parts:

1. A question with a `true` or `false` answer.
2. A value if the answer to the question is `true`.
3. A value if the answer to the question is `false`.

Here's an example:

```
var largest: Int  
let a = 15  
let b = 4  
  
largest = a > b ? a : b
```

Take a close look at the last line of code. You can read it as "If `a > b`, assign `a` to the `largest` variable; otherwise, assign `b`." In this case, `a` is greater than `b`, so its value is assigned to `largest`.

It's never *required* that you use the ternary operator in Swift. But it's useful for assigning a value based on a condition, rather than resorting to a more complex `if-else` statement.

Tip

Swift provides a global function to find the largest value in an easier-to-read and more concise expression.

```
largest = max(a, b)
```

Lab

Open and complete the exercises in [Lab – Control Flow.playground](#).

Connect To Design

Open your App Design Workbook and either add comments to the Define sections you have completed or add a blank slide at the end of the document. Reflect on the choices a user might need to make when using your app. If you find yourself writing something like “If a user does X, then Y will happen,” then you’re starting to think about how your app will behave as the user interacts with it—and how control flow will make that possible.

In the workbook’s Go Green app example, when a user logs a new object, they must choose between trash and recycling, which affects the kind of log entry that’s created.

Review Questions

Question 1 of 10

Which of the following logical operators means "equals"?

- A.** =
- B.** ==
- C.** !=
- D.** >=

[Check Answer](#)



Lesson 1.6

Xcode

All great programmers have taken time to become comfortable with the IDE (integrated development environment) they're using to develop their apps. For iOS developers, Xcode is the cornerstone of that environment. At first, Xcode might seem like just another text editor. But you'll quickly learn it's an indispensable tool for compiling and debugging code, building user interfaces, reading documentation, submitting apps to the App Store, and so much more.

In this lesson, you'll learn to create a basic iOS app, which will give you a chance to become familiar with the Xcode interface and its capabilities.

What You'll Learn

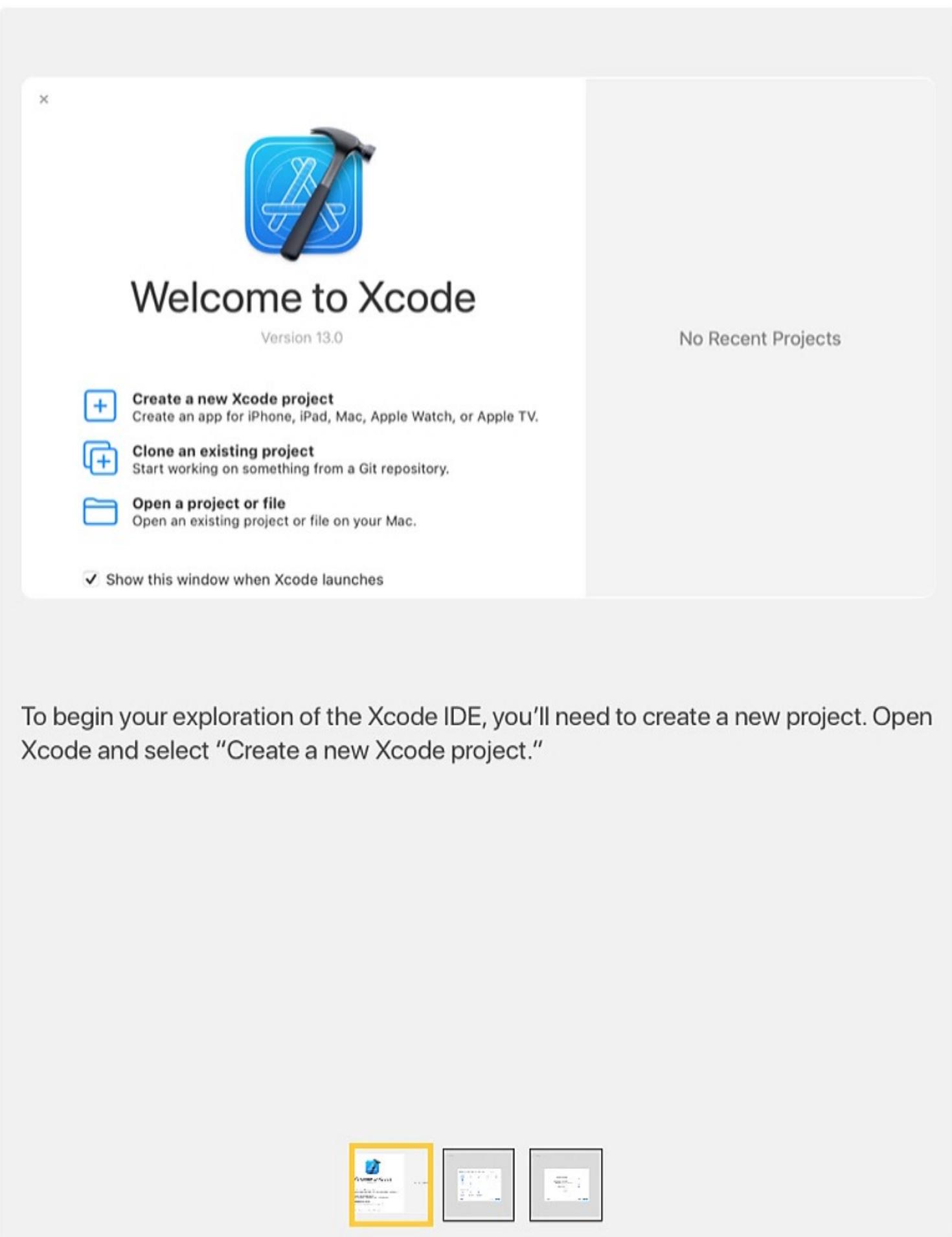
- How to navigate Xcode projects
 - How to use the Project navigator, debug area, assistant editor, and version editor
-

Vocabulary

- | | | |
|---|--------------------------------------|----------------------------------|
| • active scheme | • Git | • Storyboard |
| • assistant editor | • IDE | • target |
| • compiler | • project | • Inspector area |
| • console pane, or console area | • Project navigator | • variables view |
| • debug area | • Project template | • version editor |
| • executable apps | • push notifications | |
| | • standard editor | |

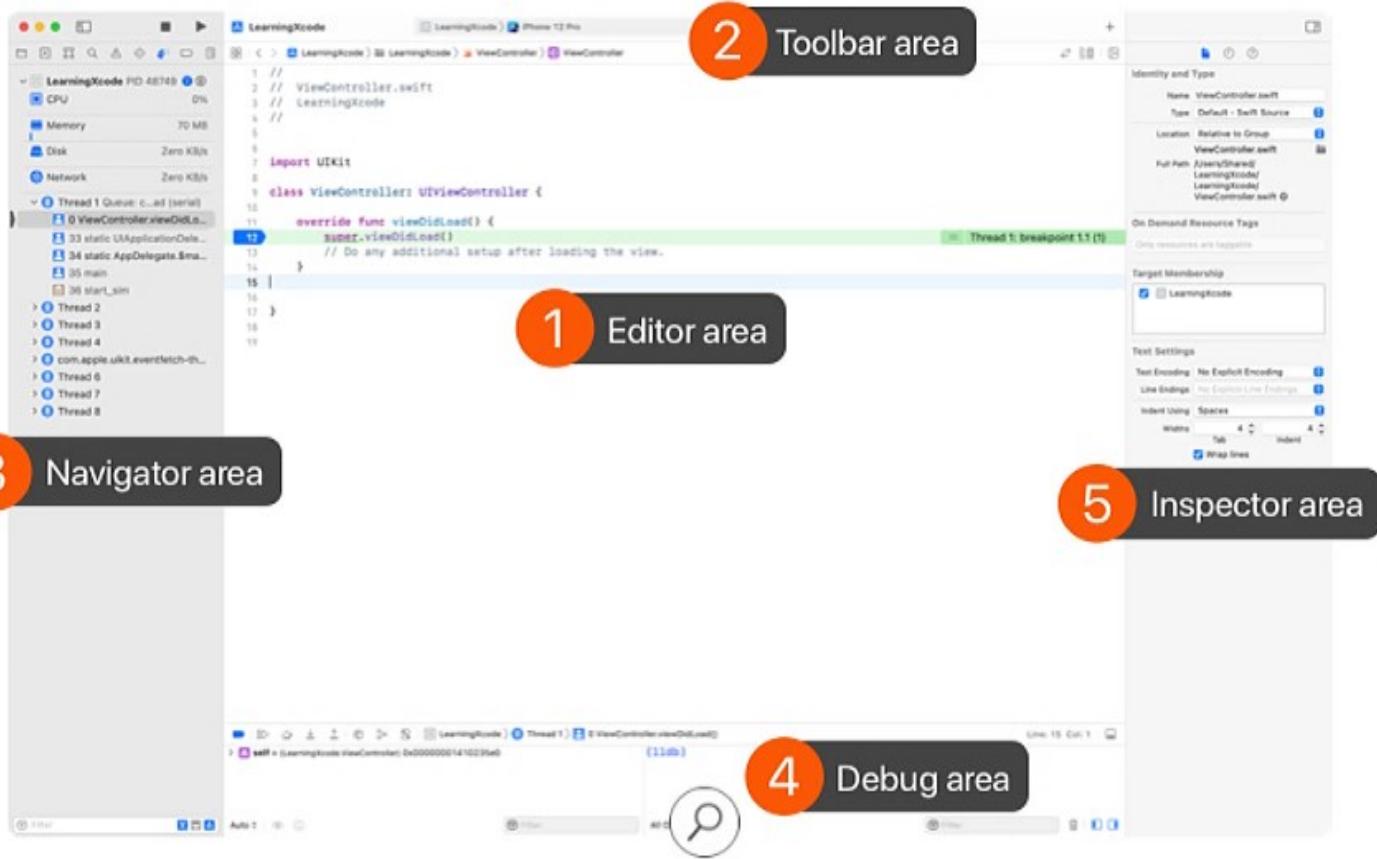
Related Resources

- [Xcode Help](#)
- [Xcode Overview](#)



Xcode Interface

The Xcode workspace is divided into five main sections. Click each call out to see more information.



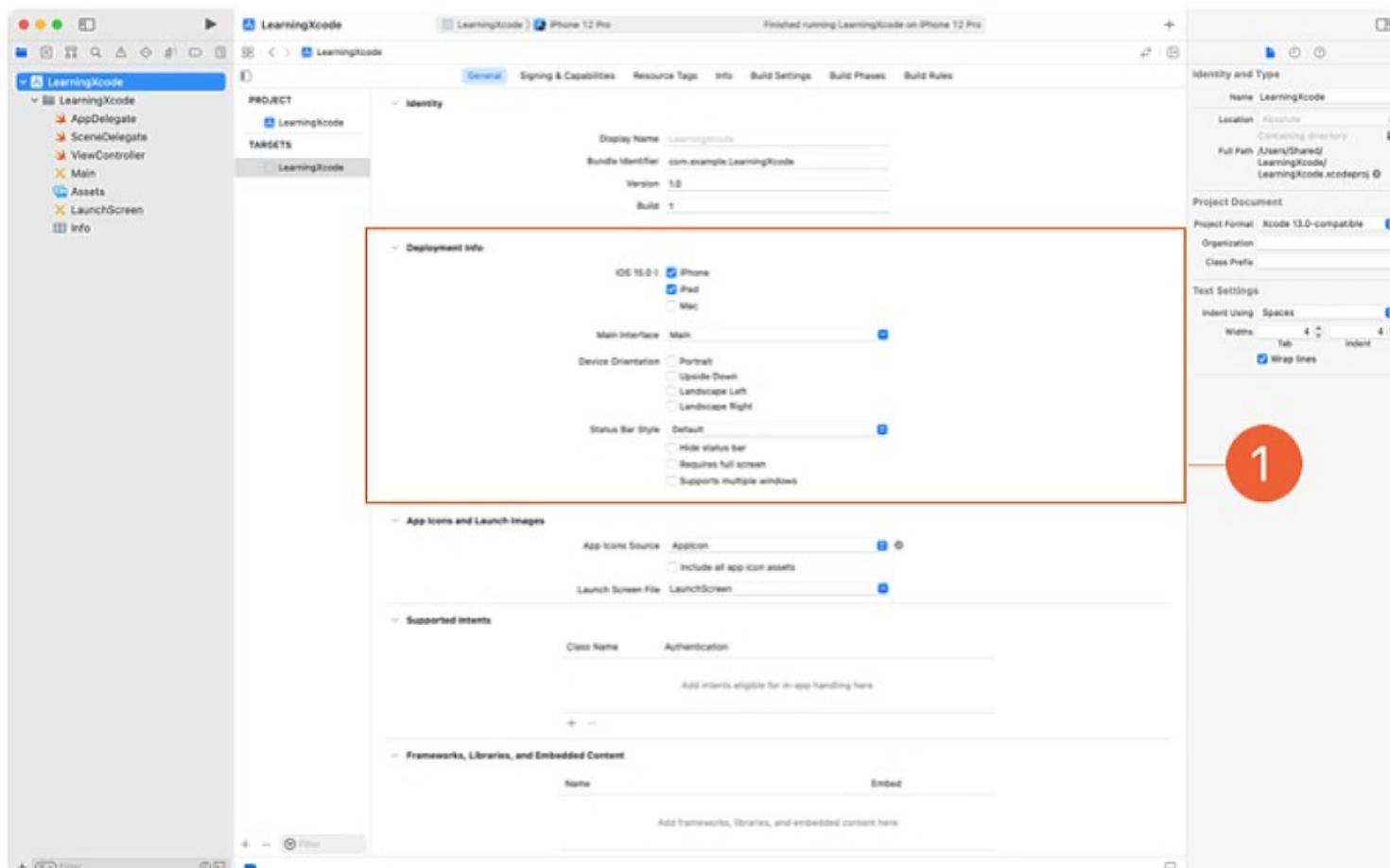
Xcode File Types

Xcode knows how to work with a variety of files that span across multiple programming languages. For now, you'll learn about files related to projects written in the Swift language. The files in your project have filename extensions that you might see in the Finder, such as `.swift` and `.xcodeproj`, which Xcode doesn't show by default. Instead, it uses icons to indicate the file type.



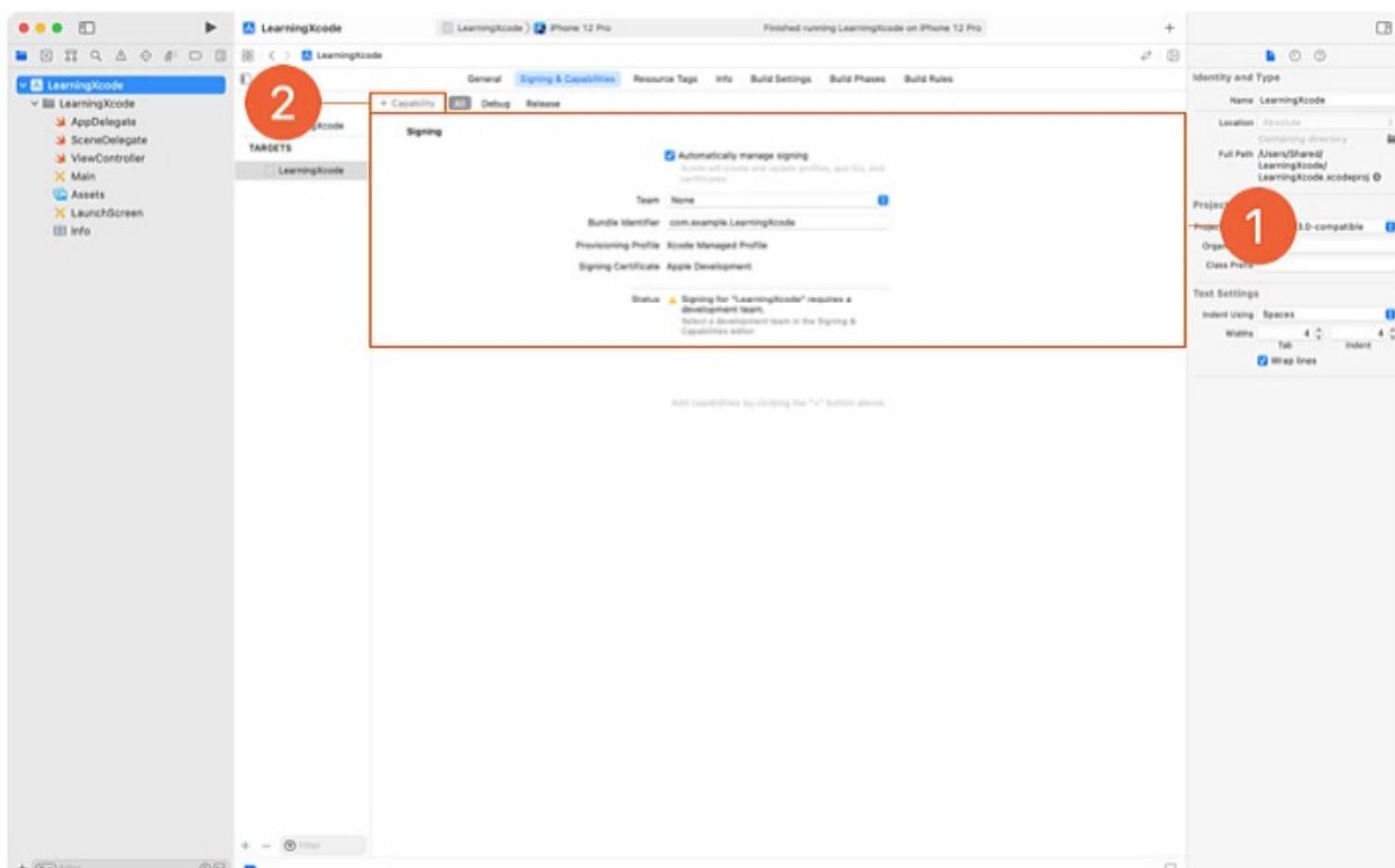
At the very top of the Project navigator, you'll see a file with a tiny blue Xcode icon. Click the file to open it in the editor area. This is the project file, which includes all the settings for your project and its targets. Each target is a product that Xcode can build from the project. For now, the targets you'll build will be executable apps. Later, you may use targets to build frameworks, different versions of a particular app, or versions for different platforms like watchOS or tvOS.

The particular project you're working with has only one target: an iOS app. Within the project file, you can change all the details of a particular target. For example, under Deployment Info, you can specify which version of iOS your code must support, change which devices your app is intended to run on, or show/hide the status bar. ①



By selecting Signing & Capabilities at the top of the pane, you can configure code signing, which is a requirement for deploying to devices or the App Store. ① From this screen, you can also enable different features within the selected target. For example, if your application needs to accept push notifications, you can add the Push Notifications capability and Xcode configures everything necessary for your app to receive notifications from the Apple Push Notifications service. You can add capability configurations by clicking the + Capability button. ②

Some capabilities have configuration options that you can disclose by clicking the triangle ▾ next to them.





Files with this icon contain Swift code, as you'd expect. Whenever you build your app, Xcode gathers up all included Swift files and runs them through the Swift compiler, which converts the code into a format your selected device understands.



This icon represents a storyboard. All storyboard files are unique to Interface Builder. They contain information about the design of each scene within your application, as well as how one screen transitions to another. You'll learn more about storyboard files in an upcoming lesson.



This icon represents an asset catalog. In an asset catalog, you can manage many different kinds of assets. This includes your app's icon, images, color definitions, and other forms of data to be bundled with your app. An asset catalog also allows you to specify variants of your assets based on device settings and capabilities such as light and dark appearance, accessibility settings for high and low contrast, and hardware differences from screen resolutions to memory capacity and graphics chip support.



This icon represents a file containing a list of properties and settings for your app. Xcode provides a special interface for editing this file so that you rarely need to interact with it directly. These settings are organized on various screens that you can find when you select the project file you learned about earlier.

Keyboard Shortcuts

As you become more proficient with Xcode, you'll discover that it's much faster to execute tasks using keyboard shortcuts. Make sure to learn the most common shortcuts right away:

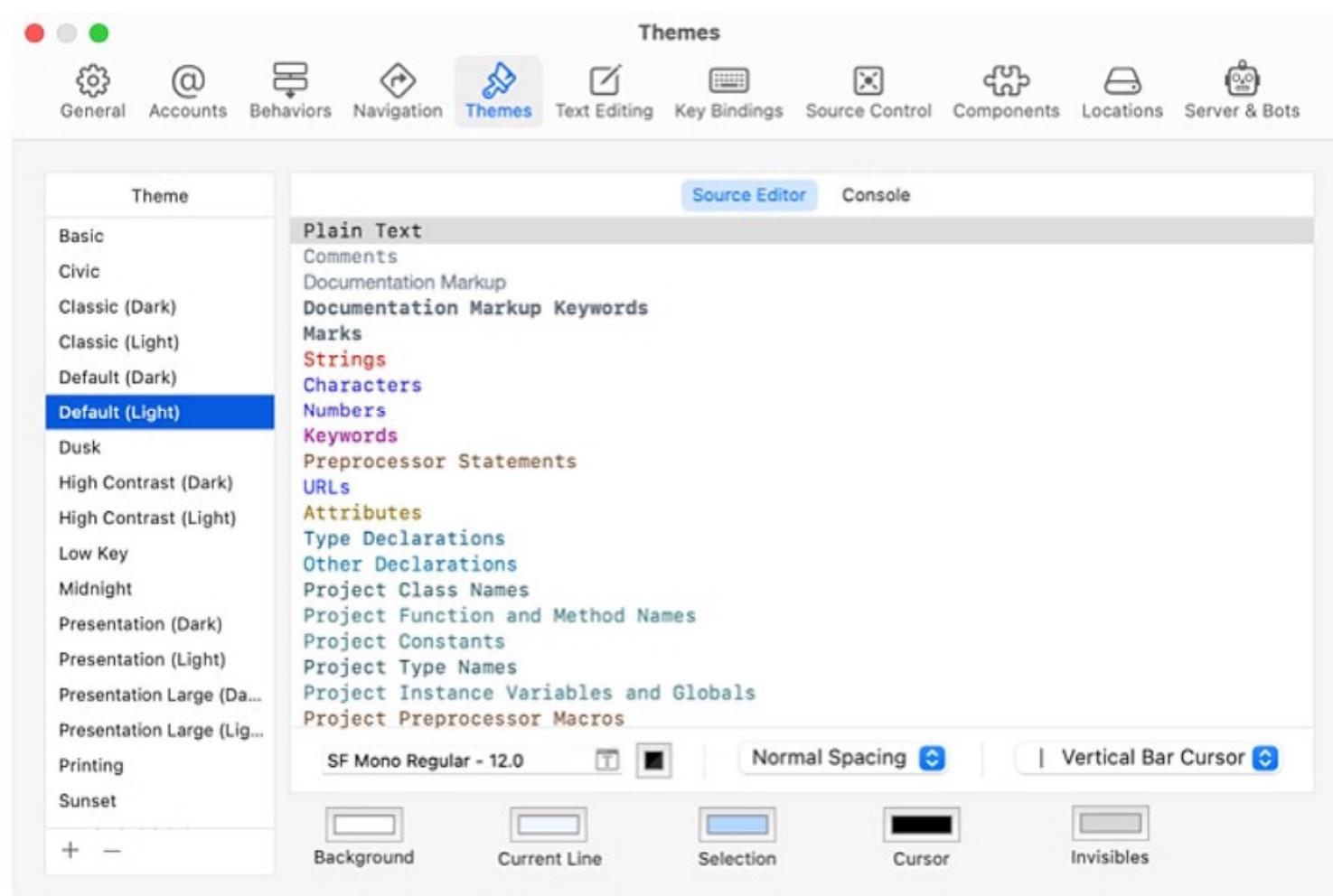
- **Command-B**—Build the project
- **Command-R**—Build and run the project
- **Command-.**—Stop building or running
- **Command-/**—Toggle comments on selected rows of code
- **Command-[**—Shift the selected code left
- **Command-]**—Shift the selected code right
- **Control-I**—Reindent the selected code
- **Command-0**—Show and hide the Navigator area
- **Option-Command-0**—Show and hide the Inspector area

To learn more keyboard shortcuts, look on the right side of any menu, or choose **Xcode > Preferences** from the menu bar and select Key Bindings at the top of the window.

Xcode Preferences

Xcode is a powerful tool with many options. As you work with Xcode you'll learn more about what it can do and how you can customize it to work how you want it to. Check out the Xcode Preferences by choosing **Xcode > Preferences** from the menu bar. You can use the Preferences to add developer accounts, customize navigation or fonts, or choose certain behaviors when text editing and more.

Open Xcode Preferences and navigate to the Fonts & Colors pane.



You can choose a new theme, or set of colors and fonts, by selecting different options in the list on the left. Some themes have a suffix of "(Light)" or "(Dark)." These variants are used respectively based on the system's current light or dark appearance. Xcode also allows you to override its appearance behavior independent of the system setting from the General pane.

To read more about Xcode and its various tools, read the [Xcode Help](#) documentation by choosing **Help > Xcode Help** from the menu bar.

Review Questions

Question 1 of 4

Where is the “Build and run” button located in Xcode?

- A. Debug area
- B. Interface Builder
- C. Toolbar
- D. Project navigator

[Check Answer](#)



Lesson 1.7

Building, Running, and Debugging an App

While Xcode playgrounds are great tools for learning and experimenting with the Swift programming language, Xcode projects are geared toward creating applications. As you write code, you'll want to see what effect it has on your app. For this, Xcode includes Simulator, an application that allows you to test your apps on a variety of devices and screen sizes—including older devices that may lack recent hardware advances, such as Touch or Face ID, or include the Home button that was removed on iPhone X.

In this lesson, you'll walk through the necessary steps to configure your Mac for building and debugging apps, and to use Simulator to test your apps from Xcode. You'll also learn how to run your app on a physical device and how to use breakpoints to locate bugs in your code.

What You'll Learn

- How to use Simulator to run apps within the Xcode environment
 - How to run an app on a physical device
 - How to perform basic debugging using breakpoints
-

Vocabulary

- | | |
|---|--|
| <ul style="list-style-type: none">• breakpoint• bug• deprecated code• compiler error, or error• exception• landscape | <ul style="list-style-type: none">• portrait• step control buttons• warning |
|---|--|
-

Related Resources

- [Xcode Help: Run an app on a simulated device](#)
- [Xcode Help: Debug code](#)

Building And Running

Your first step is to create a new project in Xcode. You'll use the same iOS App template as in the previous lesson. When creating the project, make sure the interface option is set to Storyboard. Name the project "GettingStarted" and choose a location for saving it.

Next, you'll be presented with the Xcode workspace. Locate the Scheme menu on the Xcode toolbar, then choose the device you want to simulate from the list.



Click the Run button ▶, or use the keyboard shortcut (**Command-R**) to begin launching the app in Simulator.

After Simulator has launched, you should see a device image with a white background. To rotate the image from portrait to landscape orientation, use the keyboard shortcuts **Command-Left Arrow** and **Command-Right Arrow**.

The simulator image may appear quite large, depending on the screen resolution of your Mac and the device you chose to simulate. From inside the Simulator application, you can use keyboard shortcuts, from **Command-1** to **Command-3**, to scale the device image up or down. If you're using an older Mac, you may want to select an older lower-resolution device, such as an iPhone SE, to reduce the total impact of the Simulator on your system.

To quit Simulator, use the keyboard shortcut **Command-Q**. Or just go back to Xcode.

Simulator doesn't work for all portions of an app. Certain interactions depend on the actual physical device to run. For example, if you try to use Simulator to test an interaction with the Camera app, the program will crash. If your code depends on other hardware components that don't exist on a Mac—maybe an accelerometer, a gyroscope, or a proximity sensor—you'll want to test with an actual device.

Simulator also has some software limitations. For example, push notifications can be delivered only to physical devices, and the `MessageUI` framework—which helps compose email and text messages—is incompatible with Simulator. If you're running into a lot of issues in Simulator, try testing the code on your iPhone or iPad. Your issues may be resolved.

Simulator runs on a Mac, many of which are built on the Intel "x86-64" architecture, but iOS devices are built on the ARM (Advanced RISC Machine) architecture. This means there can be underlying (and sometimes subtle) differences in how your app works.

Overall, Simulator works well when you're developing and debugging apps, but you should always test your code on actual hardware.

Using A Personal Device

The beauty of mobile apps is that you can take them anywhere. You may find that you want to share your programming progress with friends and coworkers. But before you can begin running your code on a physical device, you must have an account on the Apple Developer website. Accounts are free, so don't worry.

Using a web browser, go to developer.apple.com and click Account. You can sign up using your existing Apple ID. If you don't have an Apple ID, go ahead and create one—it's also free. Once you've entered your Apple ID, your developer account is good and you can head on back to Xcode.

(With a free account, you can run your iOS apps on only one physical device. To distribute your apps to multiple devices or publish them to the App Store, you need to enroll in the [Apple Developer Program](#).)

Now, you need to tell Xcode about your new developer account. Open Xcode Preferences using the keyboard shortcut **Command-Comma** (or choose **Xcode > Preferences** from the Mac menu bar), and then select the Accounts button near the upper-left corner. Click the "+" button in the lower-left corner and choose Add Apple ID from the menu. After entering your credentials, you're ready to run and debug apps on a physical iOS device.

Connect your iOS device to your Mac using the appropriate USB cable. Xcode will automatically download the necessary information from the device and will display its name in the Scheme menu. Choose the name of your physical device—which will typically be at the top of the list, before the device simulators.

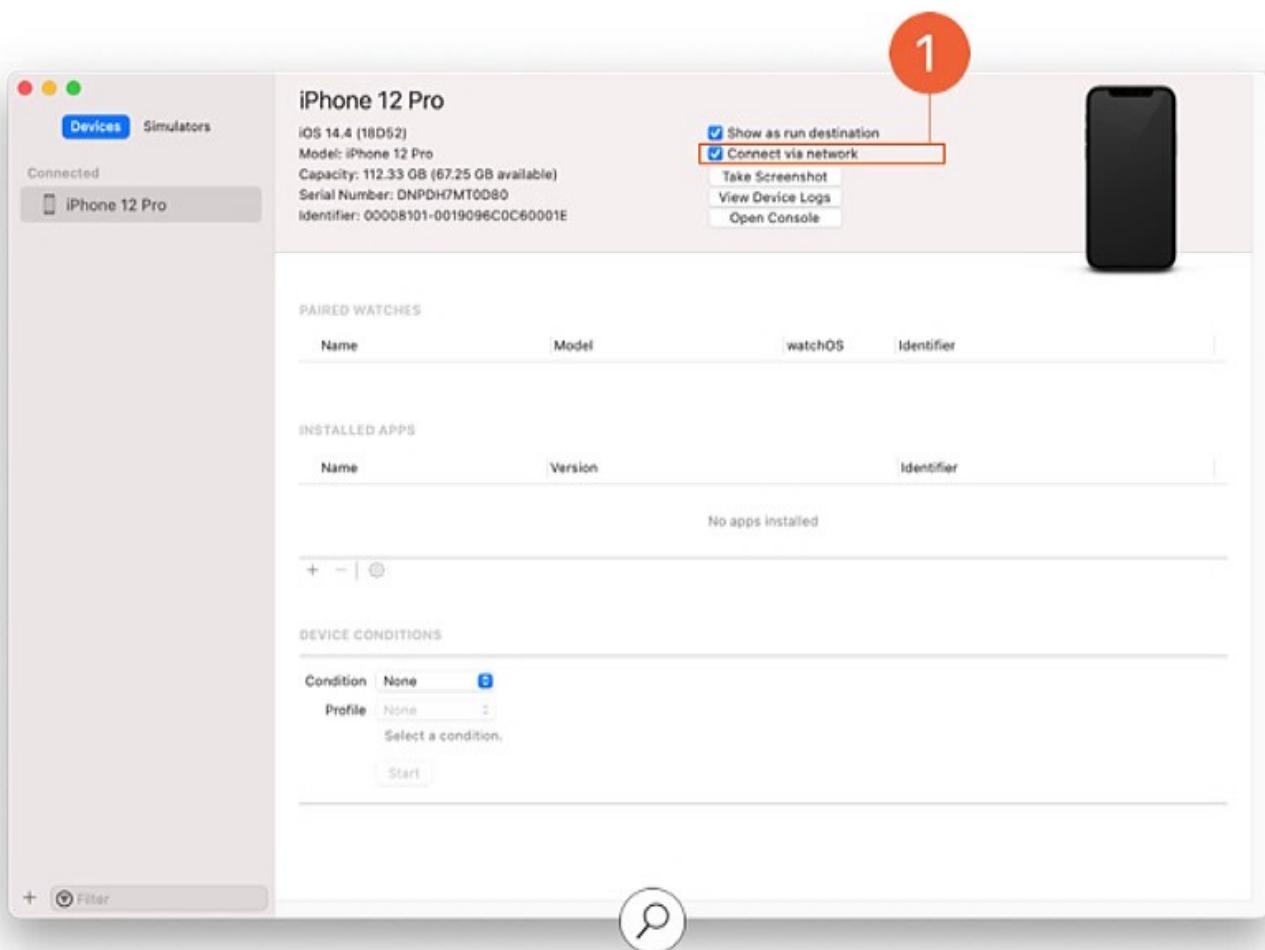
Build and run the app again. You may receive a prompt asking you to trust the developer certificate. Follow the instructions within the alert to allow the device to run your apps.

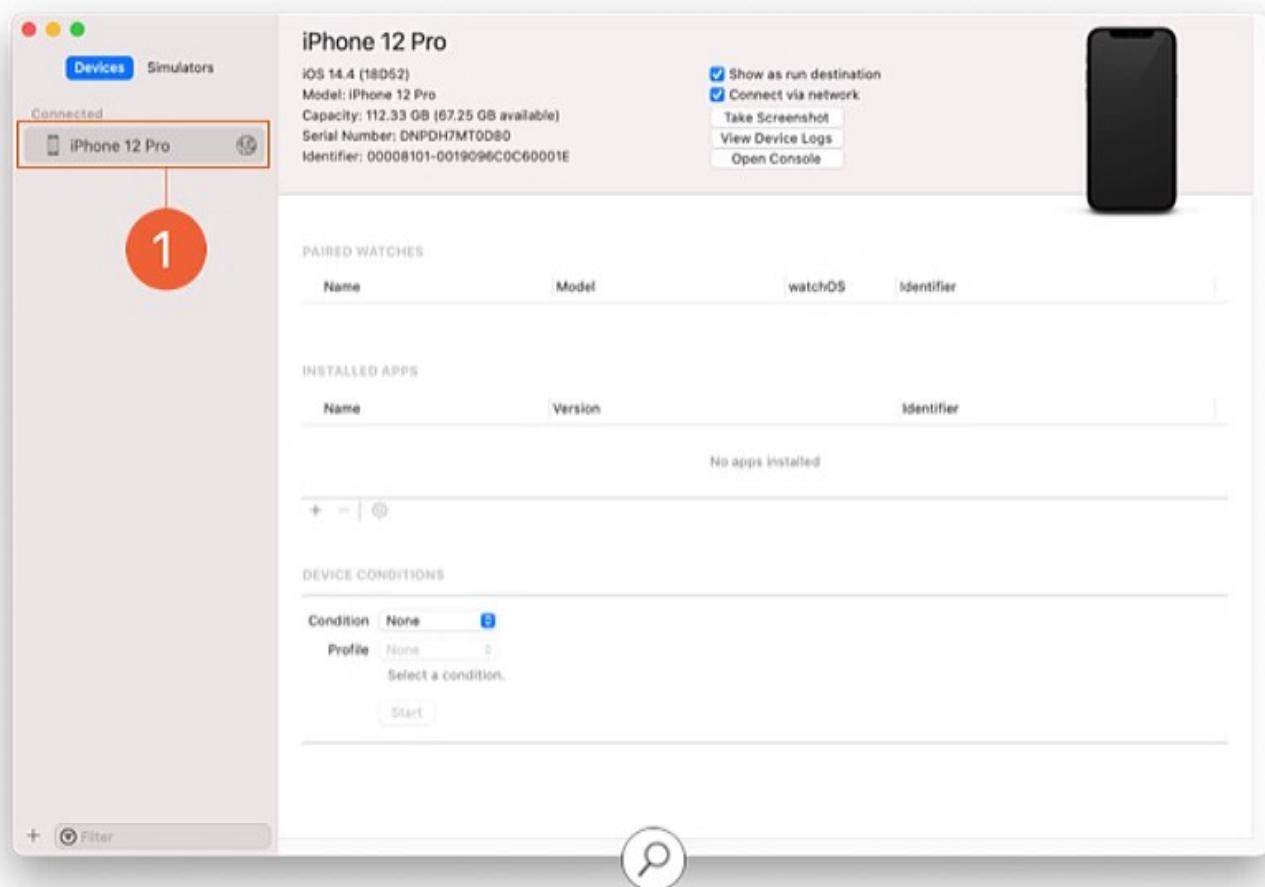
Build and run once more, and you should see the same simple white screen on your iOS device. To stop the app from running, click the Stop button near the left end of the Xcode toolbar (or use the **Command-Period** keyboard shortcut).

Building and Running Wirelessly

Xcode also gives you the option of deploying an app to your device over your network. To do this, connect your iOS device to your Mac using the appropriate USB cable, and open the Devices and Simulators window by selecting Devices and Simulators from the Window dropdown.

Ensure that your device is selected in the list farthest to the left in the Devices and Simulators window. Check the Connect via network box. ①





If your device is on the same network as your Mac, you'll see a globe appear next to your device's name within a few moments. This indicates that your device is wirelessly connected. ^①

You can now disconnect the USB cable connecting your device to your Mac, and build and run your app wirelessly.

In most cases, the above steps are sufficient for wireless pairing. However, if this doesn't work for you, you might be on a corporate or institutional network where the system administrator has put in place certain network restrictions. In this case, open the Devices and Simulators window, hold Control and click your device, then click Connect via IP Address in the menu presented. You then need to find your device's IP Address from your device's Settings, enter it in the prompt, and click Connect. This should successfully pair your device. If you run into problems with wireless debugging, you can always pair over a USB cable.

Debugging An Application

Even the best developers struggle to write perfect code. Debugging is the process of identifying and removing problems that may arise in your app. Xcode provides tools to help you through this process.

When you run an app as described above, either on Simulator or on your device, Xcode will connect the app to its debugger. This allows you to watch the execution of your code in real time, stop code execution using breakpoints, print information from your code to the console, and much more.

As you proceed through this course, you'll encounter three types of issues: warnings, compiler errors, and bugs.

Warnings

Warnings are the simplest kinds of issues to fix. They're generated whenever your code is built, but they don't prevent your program from successfully compiling and running. Some of the conditions that can throw a warning include:

- Writing code that never gets executed
- Creating a variable that never changes
- Using code that's out of date (also known as deprecated code)

Take a look at a warning. Back in Xcode, select **ViewController** in the Project navigator and, in the editor area, add the following line of code just below `super.viewDidLoad()`:

```
let x = 4
```

This code assigned a value of 4 to a constant named `x`. Build your application using **Command-B**. You should see a yellow caution sign with an explanation on the line you just added.

 Initialization of immutable value 'x' was never used; consider replacing with assignment to '_' or removing it

Xcode will do its best to explain the warning in a straightforward way:

Initialization of immutable value 'x' was never used; consider replacing with assignment to '_' or removing it.

The compiler is telling you that it created `x` but, because it's not used in a meaningful way, you can remove it. Delete the line and rebuild to remove the warning.

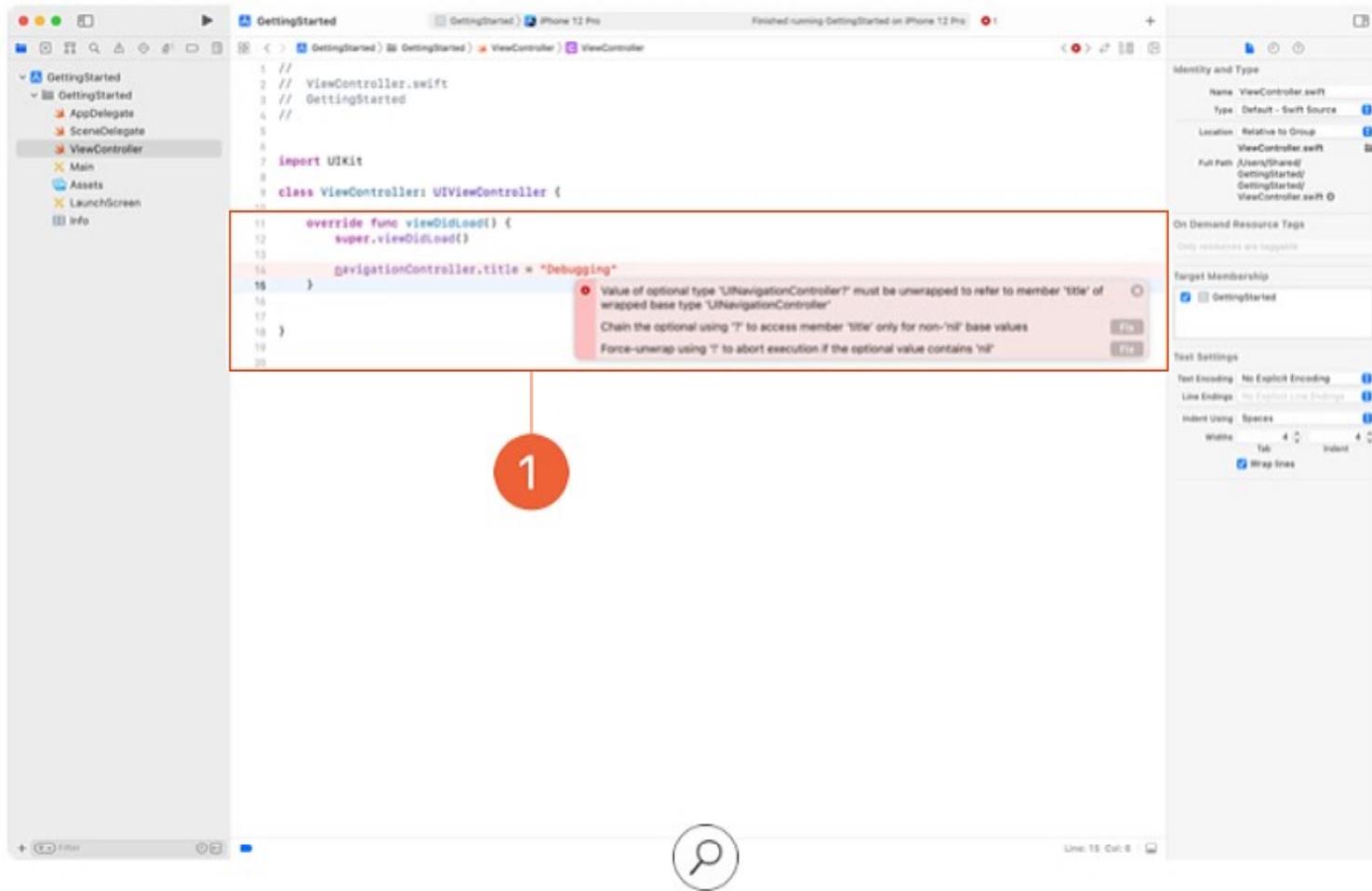
Compiler Errors

An error is indicative of a more serious issue—for example, invalid code (such as a typo), improperly declaring a variable, or improperly calling a function. Unlike a warning, an error prevents the code from ever being executed. Simulator won't even launch if your code has an error.

Here's a look at two different errors in the making. In `ViewController`, remove the open and closing parenthesis on the end of `super.viewDidLoad()`, leaving `super.viewDidLoad`. Beneath this line of code, write `navigationController.title = "Debugging"`.

After you build the app, you'll see two red error symbols with explanations on the lines you just updated.

```
super.viewDidLoad  
// Do any additional setup after loading the view.  
  
navigationController.title = "Debugging" 2 ⓘ Value of optional type 'UINavigationController?' must be unwrapped to refer to member 'title...  
Function is unused
```



The first error has an icon with an "X" in the center . Xcode provides a message that can help you identify the issue. The compiler expected to see an open and closing parenthesis as the proper syntax for calling a function. Add both characters at the end of `super.viewDidLoad` to remove the error.

The other error of the errors has a dot in the center . Click this error and Xcode displays a suggestion to fix the code. Click the Fix button to have Xcode implement this suggestion. In this instance, Xcode provides a valid fix. However, you can't rely on the compiler to properly fix all errors—you need to be able to address errors on your own.

Bugs

The third type of issue is known as a bug—and it's the hardest issue to track down. A bug is an error that occurs while running the program, resulting in a crash or incorrect output. Finding bugs can involve some time and some real detective work.

In `ViewController`, add the following lines of code below `super.viewDidLoad()`:

```
var names = ["Tammy", "Cole"]
names.removeFirst()
names.removeFirst()
names.removeFirst()
```

You may not be familiar with Swift at this point, and that's OK. These lines are simple to understand. `names` is a list containing two pieces of text, "Tammy" and "Cole." Each subsequent line removes the first item from the list. Since there are three calls to remove the first item, but only two items in the list, what do you expect will happen?

Build your application. Since the syntax is valid, you'll receive the "Build Succeeded" message. Now try running the app. After a few moments, the program will crash, and the following message will appear in red on the last line of code above:

```
Thread 1: Fatal error: Can't remove first element from an empty collection
```

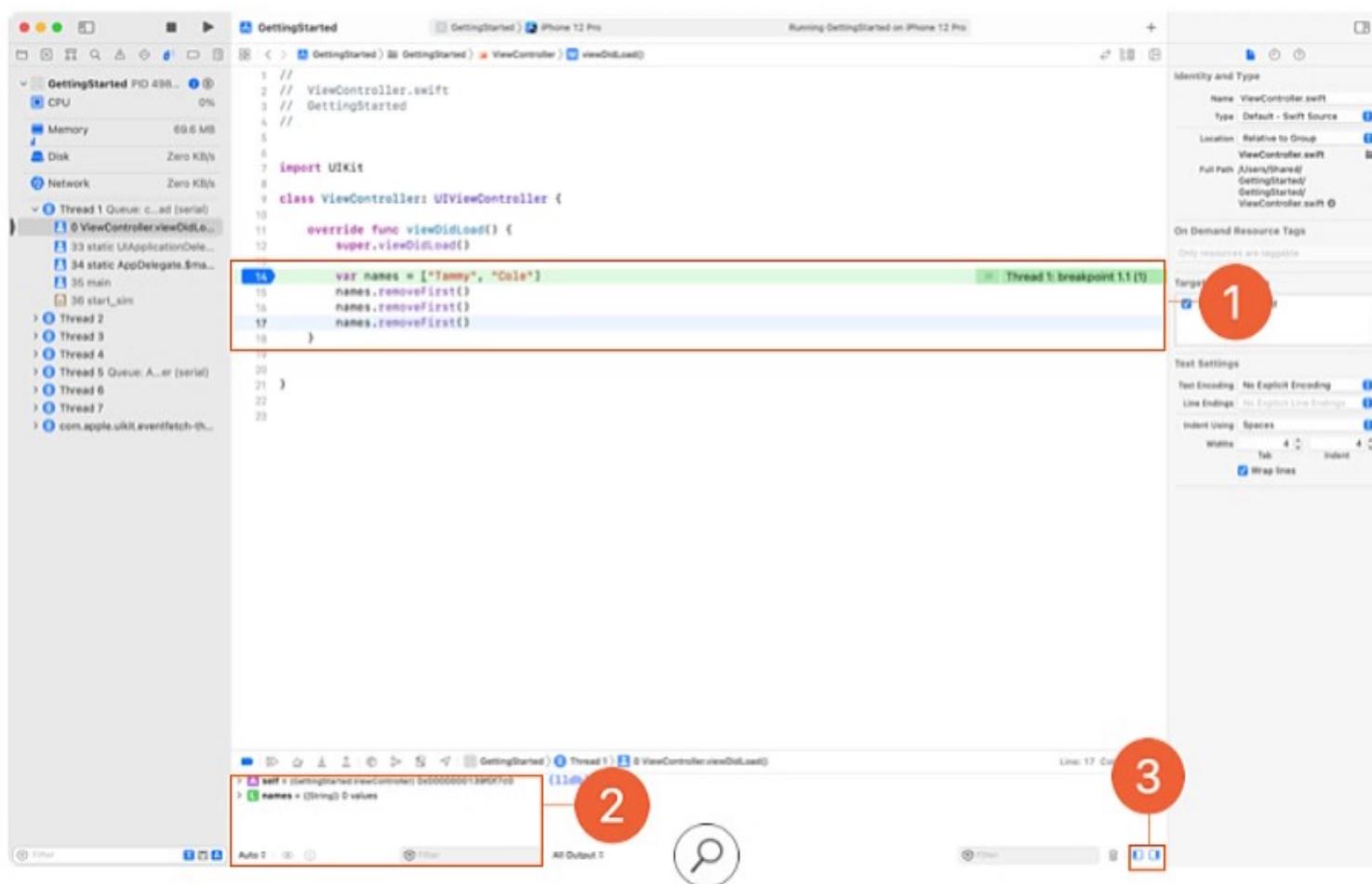
The program crashed when it tried to remove the remaining first element and couldn't find one. You've already guessed that. But imagine you're unsure how to fix the problem. Xcode can help you to step through the program one line at a time.

Before you start looking for the bug, you'll need to add a breakpoint to your code. A breakpoint pauses the execution of a program at a specified point. Create a breakpoint by clicking in the gutter area to the left of the line where you want execution to pause. In this case, add the breakpoint to the `var names = ["Tammy", "Cole"]` line.¹

The screenshot shows the Xcode interface with the 'ViewController.swift' file open. The code editor displays the following Swift code:

```
1 //  
2 // ViewController.swift  
3 // GettingStarted  
4 //  
5  
6  
7 import UIKit  
8  
9 class ViewController: UIViewController {  
10  
11     override func viewDidLoad() {  
12         super.viewDidLoad()  
13  
14         var names = ["Tammy", "Cole"]  
15         names.removeFirst()  
16         names.removeFirst()  
17         names.removeFirst()  
18     }  
19  
20 }  
21 }  
22  
23 }
```

A red circle with the number '1' is drawn around the line `var names = ["Tammy", "Cole"]`, indicating where a breakpoint should be set. The Xcode interface includes the Project Navigator, the Utilities panel, and the Debug bar at the bottom.



Build and run your app. You'll see the program pause at the breakpoint.^① That's good. In the debug area, show the variables view to inspect the current values^② (the button is on the lower right^③). Because the breakpointed line hasn't yet been executed, `names` contains no values.

From here, you can use the step control buttons at the top of the debug area to slowly continue code execution:

- Continue ⏴ — Resumes code execution until the next breakpoint is reached. If you select this button now, the code will crash, since there are no other breakpoints before the third `names.removeFirst()`.
- Step over ⏵ — Executes the selected line and pauses execution on the next line.
- Step into ⏷ — If clicked on a line with a function call, advances to the first line of the function, then pauses execution again.
- Step out ⏸ — Executes all remaining lines in the function call and pauses execution on the line after the function.

Click the "Step over" button to advance execution by one line. In the variables view, you can see that `names` has been assigned the proper values. So far, so good.



```
GettingStarted > Thread 1 > ViewController.viewDidLoad()
> A self = (GettingStarted.ViewController) 0x000000013bd2af20
> L names = ([String]) 2 values
  > [0] = (String) "Tammy"
  > [1] = (String) "Cole"

Auto ⌂ ⌂ ⓘ Filter All Output ⌂ ⌂ ⓘ Filter
```

Click the "Step over" button to execute the first call to `names.removeFirst()`. `names` no longer includes "Tammy" in its list, so that worked fine. Click "Step over" again to execute the second call to `names.removeFirst()`, leaving `names` an empty list with zero values. Still OK. By now, it should be clear that the third call to `names.removeFirst()` is responsible for the bug.

Remove the third call to `names.removeFirst()` and run the program again to verify that the error has been fixed.

Debugging is a crucial skill for developers to build. When debugging, take the following approach:

1. Try to understand the problem
2. Brainstorm a potential solution
3. Try the solution
4. Verify it worked, repeat as necessary

Take each bug one step at a time. It can be frustrating to run into bugs when building apps, but it feels great when you're able to fix them.

Lab—Debug Your First App

The objective of this lab is to find and resolve compiler errors, runtime errors, and compiler warnings.

Step 1

Find And Fix Compiler Errors

- Open the Xcode project, “FirstTimeDebugging.”
 - Try to run the app. Note that it won’t run due to a few compiler errors. As you’ve learned in this lesson, compiler errors are indicated by red symbols in line with the mistake—or where the compiler guesses the mistake might be. All compiler errors are also listed in the Issue navigator.
 - Fix the compiler errors so that you can run the app. Here are two of the more common mistakes that may have caused this app’s compiler errors:
 - Missing or extra parentheses or braces (whether opening or closing).
 - Referencing a function or property but with incorrect spelling (The compiler is very literal and expects you to reference a function or property exactly by the name you gave it.)
- Hint:** Sometimes a single mistake can cause the compiler to flag multiple errors. Fix one mistake and you might eliminate all the error symbols.

Step 2

Find And Fix Runtime Errors

- Were you able to remove all the red error symbols from the project? If so, try to run the app again.
- This time, notice that the app stops execution right after opening in Simulator and that there’s a red line across one of the lines of code on the screen. The fact that the line is red indicates something went wrong. Look at the text in the console area to learn what the issue might be. Try to solve this runtime error. If it’s helpful, you might want to add breakpoints at and before the affected code, then run the app again.

Step 3

Find And Fix Compiler Warnings

- Now that the app runs, focus your attention on a few more problems. Open the project’s Issue navigator. You’ll note several warnings, indicated by yellow triangles. Address all of these warnings.

Congratulations! You should now have a project free of compiler errors, runtime bugs, and warnings.

Review Questions

Question 1 of 3

What do you use to switch to running code on a physical device instead of using Simulator?

- A. Run button
- B. Scheme menu
- C. Debug area

Check Answer



Lesson 1.8

Documentation

Whether you're stuck on a difficult bug or getting familiar with some new code, you'll have access to a rich set of Xcode documentation that will move your development forward.

In this lesson, you'll learn how to view multiple forms of documentation, including a library of sample code written by Apple experts.

What You'll Learn

- How to use the documentation browser
 - How to find sample code and framework guides
-

Vocabulary

- [documentation browser](#)
 - [Quick Help](#)
 - [symbol](#)
-

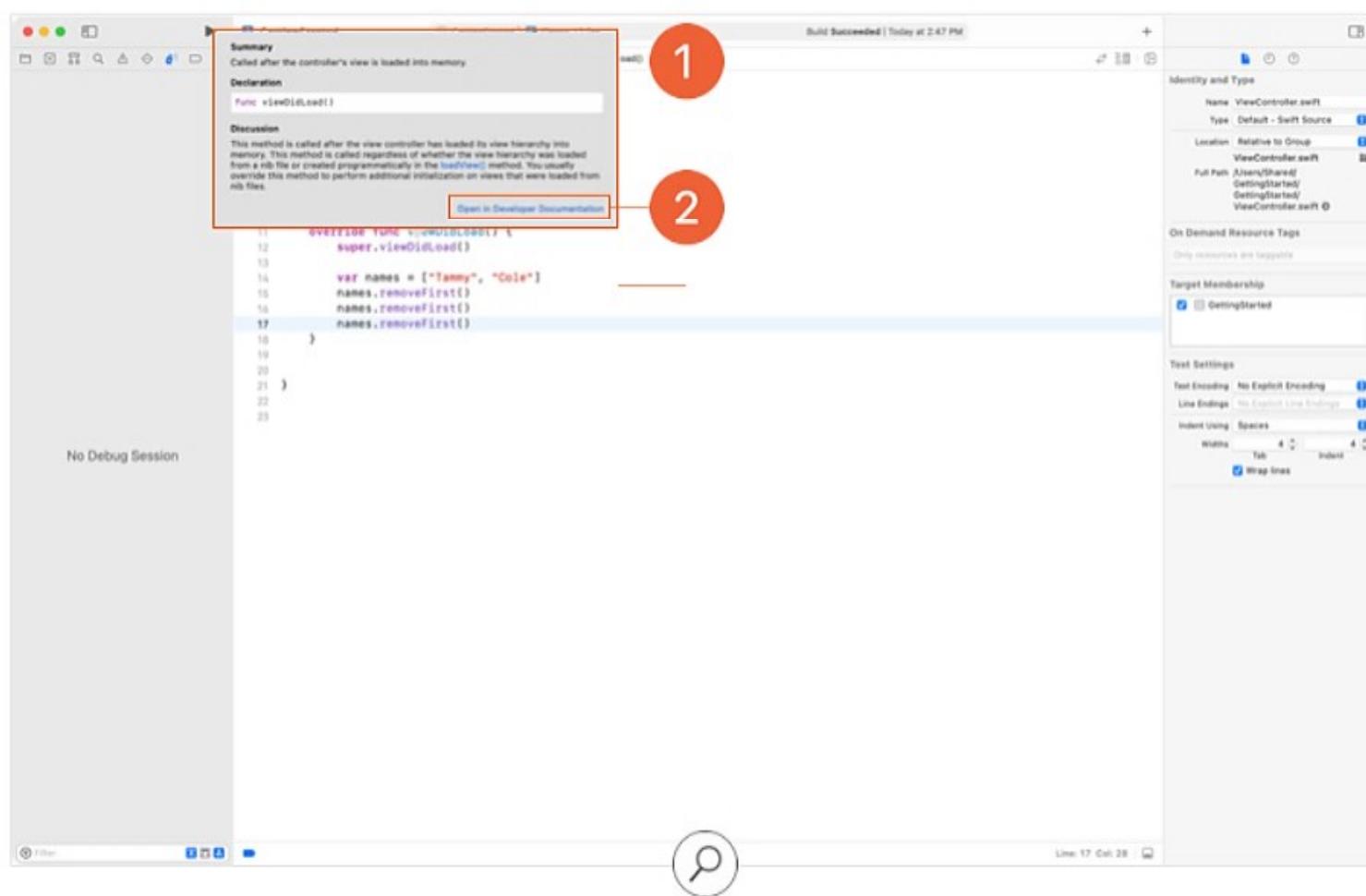
Related Resources

- [API Reference](#)
- [Xcode Help: Search for developer documentation](#)

Every developer has a preferred method of accessing documentation. Some prefer to view it in a web browser, while others like to use the documentation browser provided by Xcode. By learning multiple ways to interact with documentation, you can decide which method works best for you. The faster you can find answers to your questions, the faster you can get back to writing code.

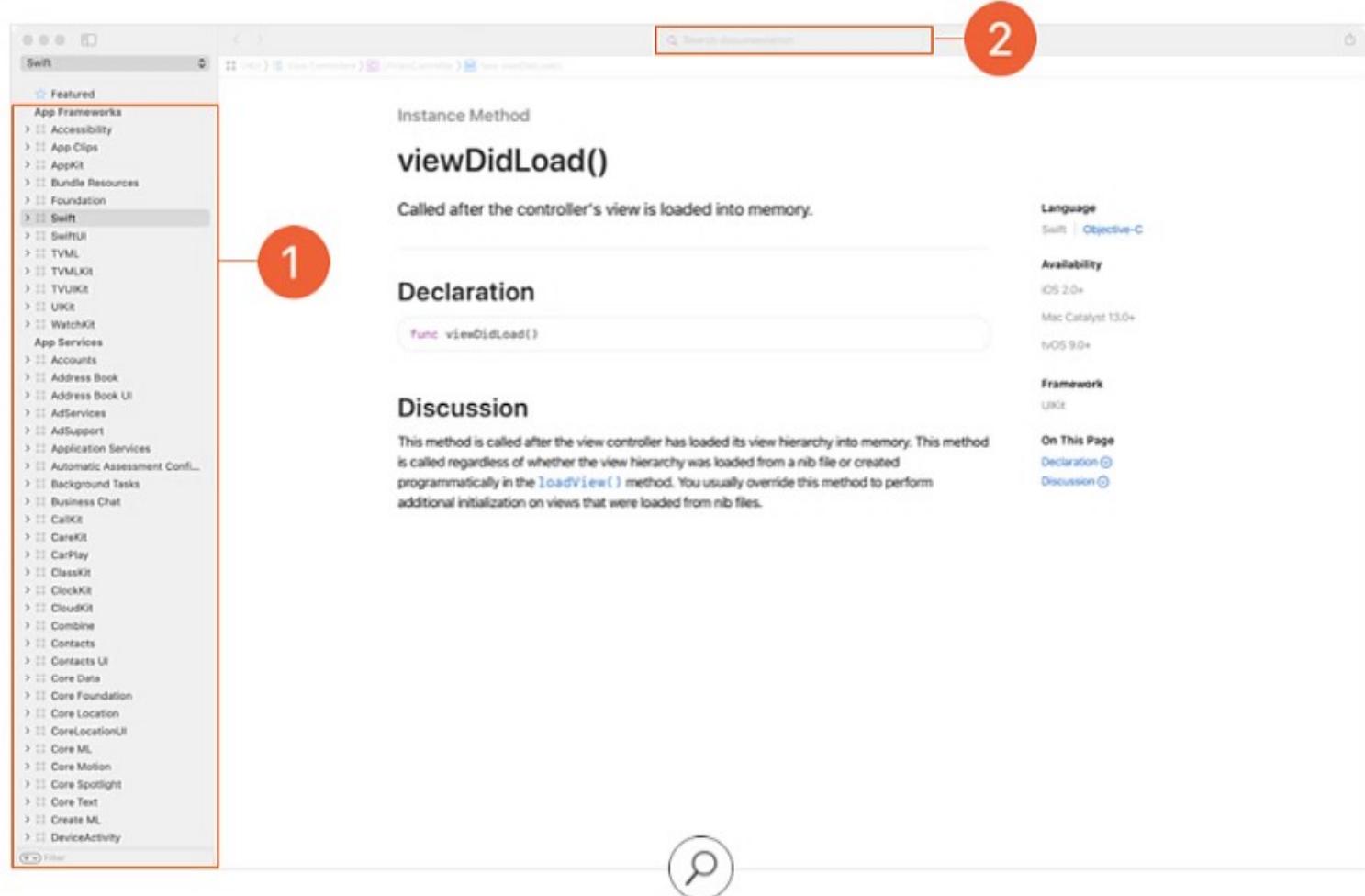
Documentation Browser

In the previous lesson, you added lines of code inside the `viewDidLoad()` function. Do you have any idea what that function does or when the function is called in your program? Xcode provides a fast answer using the Quick Help feature. **Option-click** the `viewDidLoad()` method name, and Xcode displays a popover with a brief description of the function. ①



Within the Quick Help popover, click Open in Developer Documentation to conveniently access the Xcode Developer Documentation for the function. ② The documentation includes a more thorough explanation of the function, the OS versions that support it, the framework it belongs to (in this case `UIKit`), and references to related functions.

In the navigation on the left, you can jump quickly to different sections of the Developer Documentation.^① (You can also access Developer Documentation from the Window menu or by using the shortcut, Command-Shift-0.)



At the top of the documentation window, you'll see a search field, which allows you to look up documentation about any function, class, or framework.^② This feature makes it quick to search through thousands of pages of documentation, so answers are readily available—which is why many developers choose to leave the documentation window open while they're coding.

Try navigating through the Xcode documentation. Start typing `UIViewController`. As you type, Xcode will suggest matching options with autocomplete. As soon as you see `UIViewController` in the menu, select it to display the documentation for `UIViewController`.

In the right-hand column, under On This Page, click the Topics link to jump down the page. The Topics section displays a list of symbols grouped by topic. A symbol is a function or variable associated with a particular class. In the documentation, these symbols are sorted into logical categories rather than in alphabetical order, so you can more easily locate items that interest you.

The screenshot shows the Xcode documentation interface. On the left is a sidebar with a tree view of Swift frameworks and modules. The 'UIKit' module is selected. The main content area has a search bar at the top with the text 'Q: uiviewcontr'. Below the search bar, the title 'Topics' is displayed. Under 'Topics', there are three sections: 'Creating a View Controller', 'Getting the Storyboard and Nib Information', and 'Managing the View'. Each section contains one or more symbols, each with a code snippet and a brief description. A magnifying glass icon is located at the bottom center of the main content area.

Topics

Creating a View Controller

`init(nibName: String?, bundle: Bundle?)`
Creates a view controller with the nib file in the specified bundle.

`init?(coder: NSCoder)`
Creates a view controller with data in an unarchiver.

Getting the Storyboard and Nib Information

`var storyboard: UIStoryboard?`
The storyboard from which the view controller originated.

`var nibName: String?`
The name of the view controller's nib file, if one was specified.

`var nibBundleOrNil: Bundle?`
The view controller's nib bundle if it exists.

Managing the View

`var view: UIView!`
The view that the controller manages.

`var viewIfLoaded: UIView?`
The view controller's view, or nil if the view is not yet loaded.

`var isViewLoaded: Bool`
A Boolean value indicating whether the view is currently loaded into memory.

`func loadView()`
Creates the view that the controller manages.

When you're building a new feature, these logical categories are a great way to find out if the object can do what you need.

Look through the topics for `UIViewController` and try to find the symbol for the `view` property. Click this symbol.

The screenshot shows the Apple Developer Documentation interface. The search bar at the top contains the text "view". The left sidebar is titled "Swift" and lists various Swift framework categories such as Accessibility, App Clips, AppKit, Bundle Resources, Foundation, and many others under "App Support" like Task Management, Undo, Progress, and Activity Sharing. The main content area displays the documentation for the `view` property of `UIViewController`. The title "Instance Property" is followed by the name "view". The description states: "The view that the controller manages." To the right, there are sections for "Language" (Swift | Objective-C), "Availability" (iOS 2.0+, Mac Catalyst 13.0+, tvOS 9.0+), and "Framework" (UIKit). Below the description, there is a code snippet: `var view: UIView! { get set }`. The "Declaration" section includes a note about the property being the root view of the view controller's hierarchy and its default value being `nil`. It also mentions that accessing it when `nil` triggers a `loadView()` call. The "Discussion" section provides information on view controllers being sole owners of their views and the exception of container view controllers adding subviews to their own hierarchies. It also notes that using `isViewLoaded` instead of `view` prevents automatic loading. A "See Also" section links to `View Controller Programming Guide for iOS`.

Here you'll see information for your selected property. Most instance property documentation will follow a similar pattern:

- **A short description** — A quick summary of what the property is
- **Declaration** — The name used to access the property and the property's associated type
- **Discussion** — An in-depth description, discussing the finer (albeit, important) elements to consider when using the property
- **See Also** — Other symbols that are related to the property that may be of interest

In the upper-left corner, you'll see two buttons with chevrons . Use these to navigation backward and forward through your viewing history. Click the back button to return to the documentation for `UIViewController`. Find the symbol for the function `viewWillAppear(_:)`.

The screenshot shows the Xcode documentation interface for the `viewWillAppear(_:)` method of `UIViewController`. The left sidebar shows the Swift module hierarchy. The main content area displays the following details:

- Instance Method**
- viewWillAppear(_:)**
- Description**: Notifies the view controller that its view is about to be added to a view hierarchy.
- Declaration**: `func viewWillAppear(_ animated: Bool)`
- Parameters**:
 - animated**: If `true`, the view is being added to the window using an animation.
- Discussion**: This method is called before the view controller's view is about to be added to a view hierarchy and before any animations are configured for showing the view. You can override this method to perform custom tasks associated with displaying the view. For example, you might use this method to change the orientation or style of the status bar to coordinate with the orientation or style of the view being presented. If you override this method, you must call `super` at some point in your implementation.
- Note**: If a view controller is presented by a view controller inside of a popover, this method is not invoked on the presenting view controller after the presented controller is dismissed.

Most method documentation will follow a very similar pattern to instance property documentation with one possible addition:

- **Parameters** — If the function has parameters (inputs), they are listed with the name used to access the parameter and a short description of the parameter

You'll work more with the documentation browser as you work through this course.

Sample Code And Topics

There's more to Developer Documentation than just descriptions of types and methods. As you work your way through coding concepts, it can be useful to read more about a certain topic or try out some sample code. The [top level of documentation for the UIKit framework](#), for example, includes numerous overview topics and articles to help you get started. And the [views and controls](#) subtopic includes a [UIKit Catalog](#) sample code project.

You'll need to learn how to read and understand documentation to be a successful developer. Because documentation is very technical, it can be hard to digest when reading it the first time. It takes practice.

Throughout this course you will be given links to reference guides and API documentation in the "Related Resources" section. Read through them. You may not always understand every sentence or concept from reading, but you will be building a very useful skill that will help you be a successful developer.

Lab—Use Documentation

The objective of this lab is to use Xcode documentation to learn about iOS frameworks. You'll navigate through documentation and Quick Help to answer questions about the `UIView` class and about a method provided in an Xcode project.

Step 1

Create A Pages Document For Your Answers

- Create a new Pages document named “Lab – Use Documentation” and save it to your project folder. Use this document to write your answers to the questions in the rest of this lab. (If you don't have Pages available, you can use any text editor.)

Step 2

Use Documentation To Learn About The `UIView` Class

- What are three of the primary responsibilities of a `UIView` object?
- What does documentation call a view that's embedded in another view?
- What does documentation call the parent view that's embedding the other view?
- What is a view's frame?
- How is a view's bounds different from its frame?

Be sure to save your Pages document to your project folder.

Lesson 1.9

Interface Builder Basics

Xcode has a built-in tool called Interface Builder that makes it easy to create interfaces visually. In this lesson, you'll learn how to navigate through Interface Builder, add elements onto the canvas, and interact with those elements in code.

What You'll Learn

- How to use Interface Builder to build user interfaces
 - How to preview user interfaces without compiling the app
-

Vocabulary

- [action](#)
 - [canvas](#)
 - [Document Outline](#)
 - [view controller](#)
 - [initial view controller](#)
 - [outlet](#)
 - [scene](#)
 - [XIB](#)
-

Related Resources

- [Xcode Help: Interface Builder workflow](#)
- [Build a Basic UI](#)

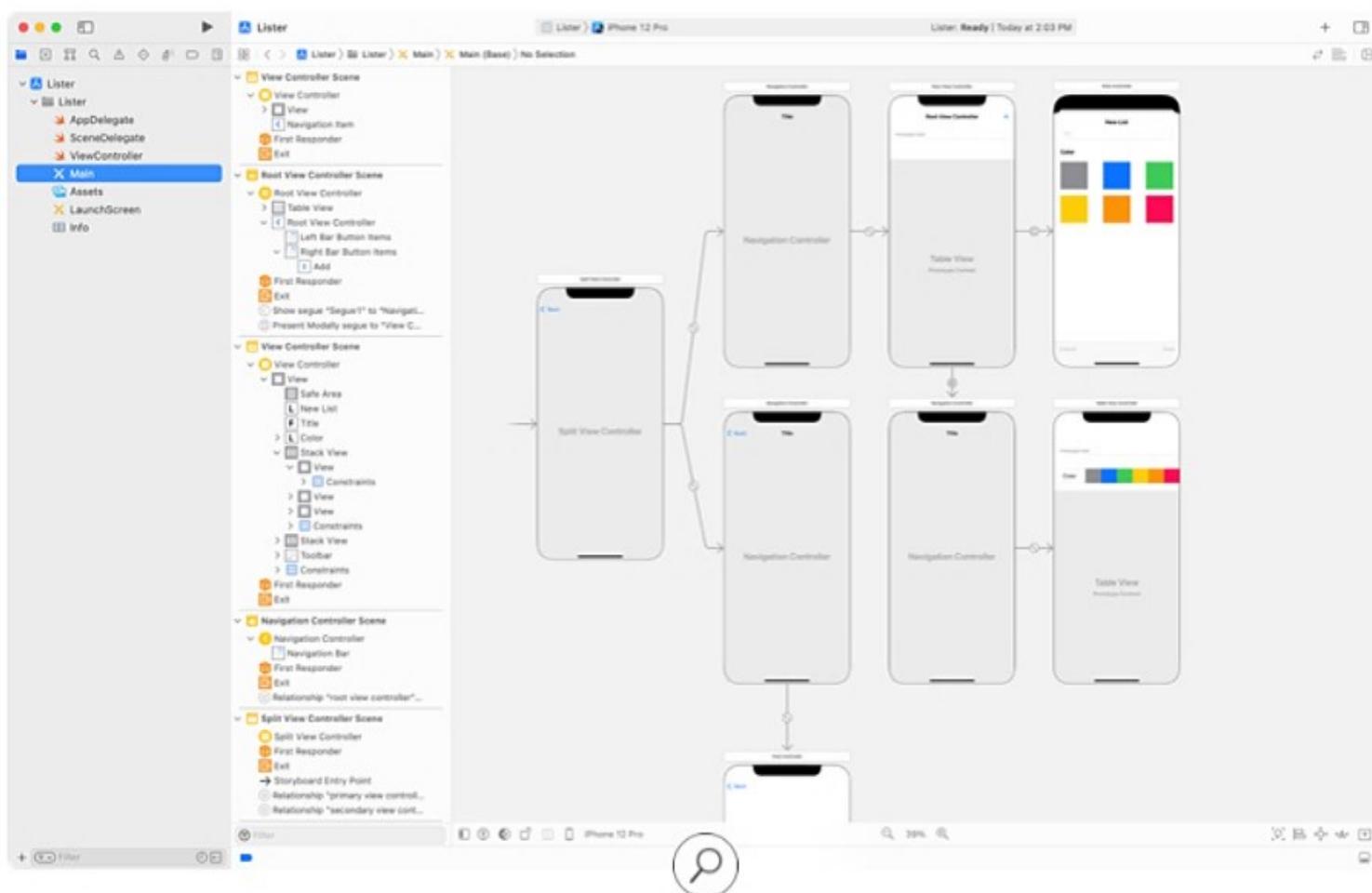
The best way to learn the basics of Interface Builder is to dive into Xcode and explore some of its features. Start by creating a new Xcode project using the iOS App template. When creating the project, make sure the interface option is set to Storyboard. Name the project "IBBasics."

Storyboards

Interface Builder opens whenever you select an XIB file (**.xib**) or a storyboard file (**.storyboard**) from the Project navigator.

An XIB file contains the user interface for a single visual element, such as a full-screen view, a table view cell, or a custom UI control. XIBs were used more heavily before the introduction of storyboards and you may hear seasoned macOS or iOS developers refer to XIB files as “Nib” files. They’re still a useful format in certain situations, but this lesson focuses on storyboards.

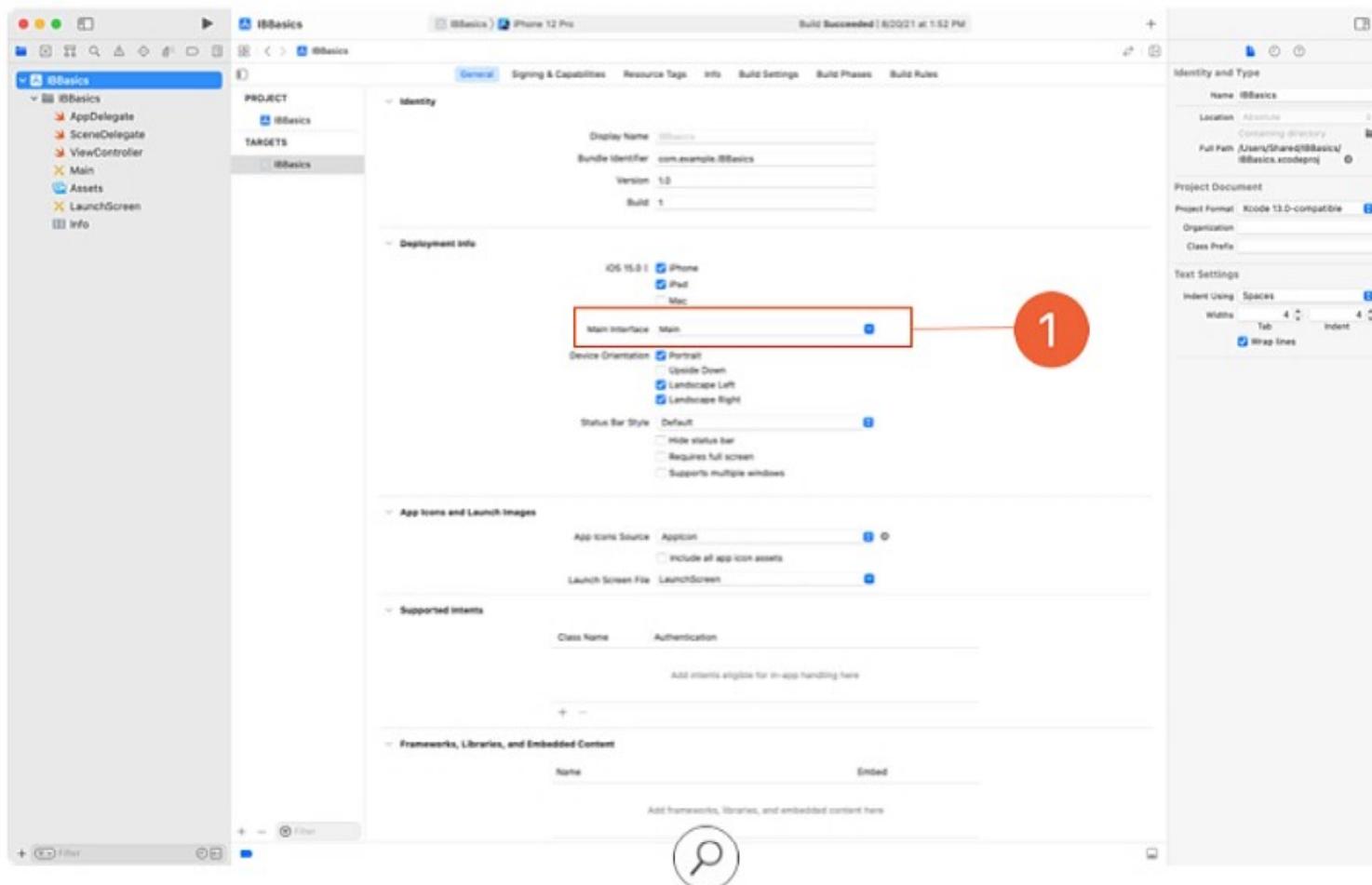
In contrast with an XIB, a storyboard file includes many pieces of the interface, defining the layout of one or many screens as well as the progression from one screen to another. As a developer, you’ll find that the ability to see multiple screens at once will help you understand the flow within your app.



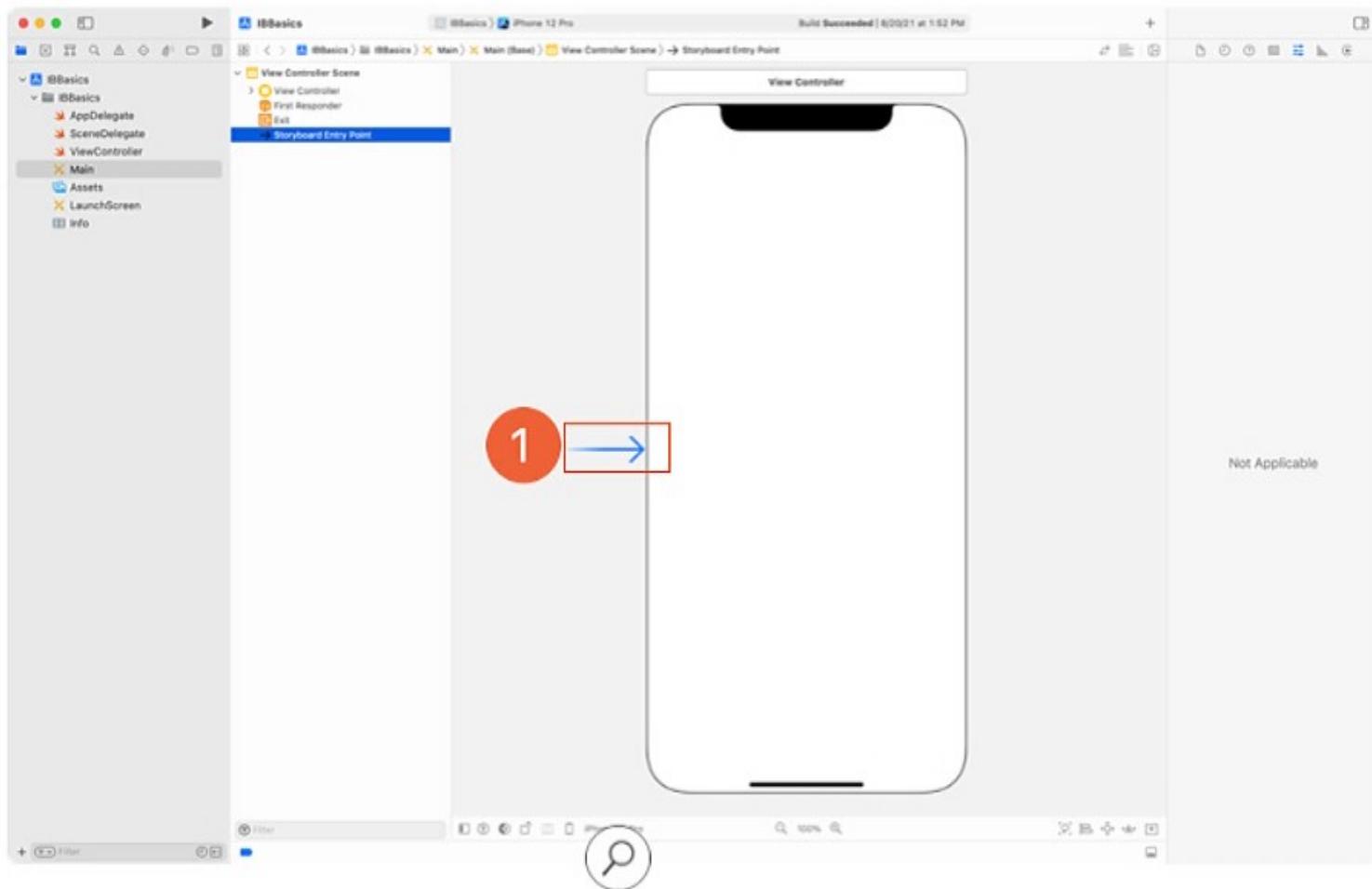
By default, a new iOS app project comes with one storyboard named **Main.storyboard**. Because Xcode doesn't show file extensions by default, it appears as **Main** in the Project navigator. Click it now to open the storyboard in Interface Builder. In the center of the screen, you'll see a single scene with a plain white view on an otherwise blank canvas. This scene is called the *initial view controller*, and it is the first screen that will appear when you open the app. As you add more scenes to the storyboard, you can drag them anywhere on the canvas. To see more view controllers at once, use two fingers on a Multi-Touch trackpad to pinch and zoom the canvas out or to zoom in on a particular view. If you don't have a trackpad, you can use the zoom buttons near the bottom of the canvas to achieve the same result.



Build and run the project. Simulator displays the same white screen you saw in the storyboard. How did the app know to display this screen? To investigate, select the top-most file (**IBBasics**) in the Project navigator, and find the Deployment Info section under the General heading. The entry in Main Interface defines which storyboard file is loaded first when the app launches. ① Because you created the project using the iOS App template, this entry was preconfigured to use the **Main** storyboard.

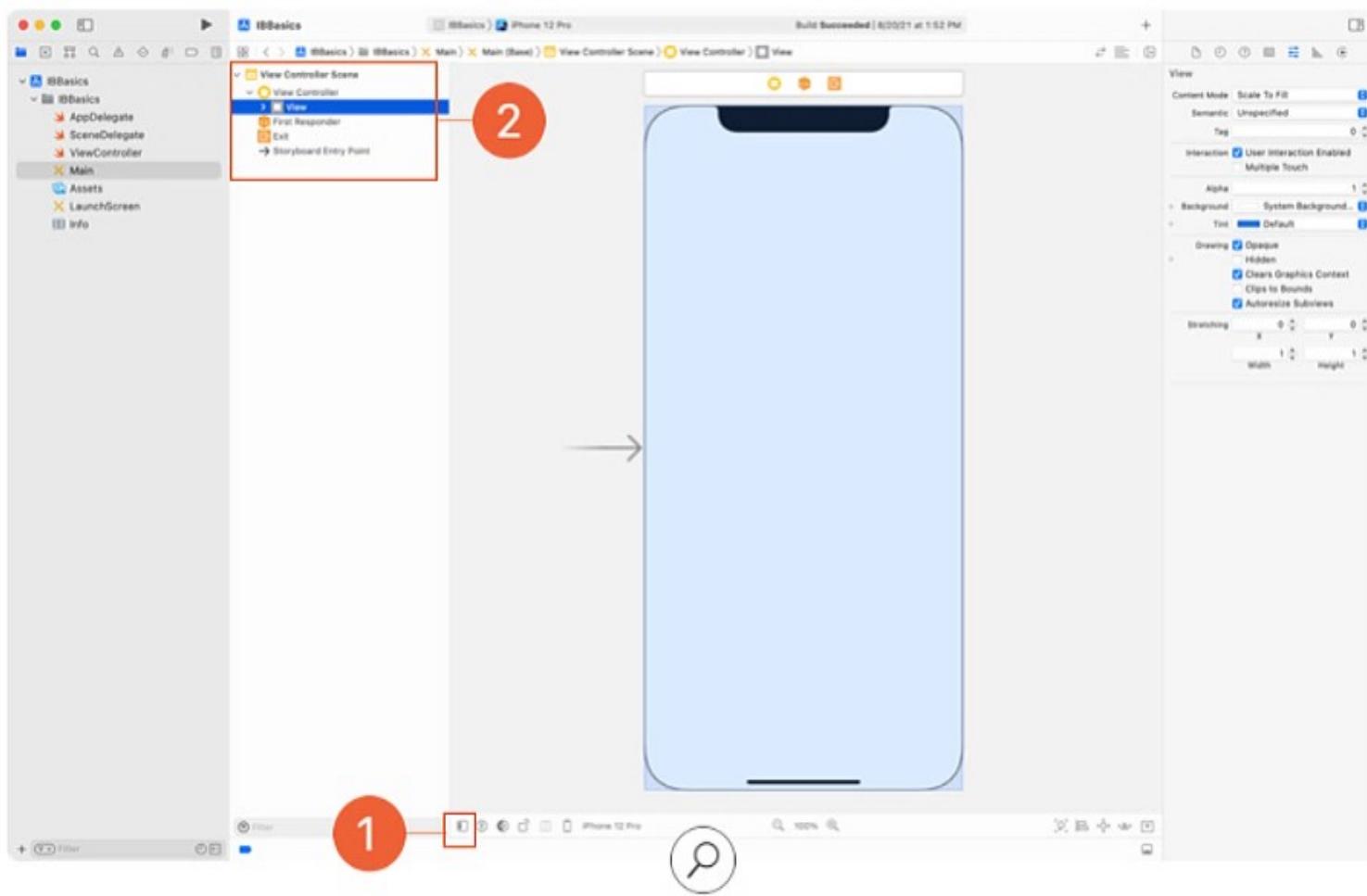


You can change the initial view controller by moving the entry point (the right-facing arrow) to the left of the desired view controller. ① Currently in this project, the Main storyboard has only one view controller, so you won't be able to change it.



Interface Builder Layout

To the left of the main canvas is the Document Outline view. To reveal it, click the Show Document Outline button  in the bottom left corner of the canvas.  The Document Outline displays a list of each view controller in the scene, along with a hierarchical list of the elements within each view controller. Click the gray triangles to the left of each item to inspect the contents. As you might notice, clicking the view in the outline will highlight the corresponding element on the canvas. 



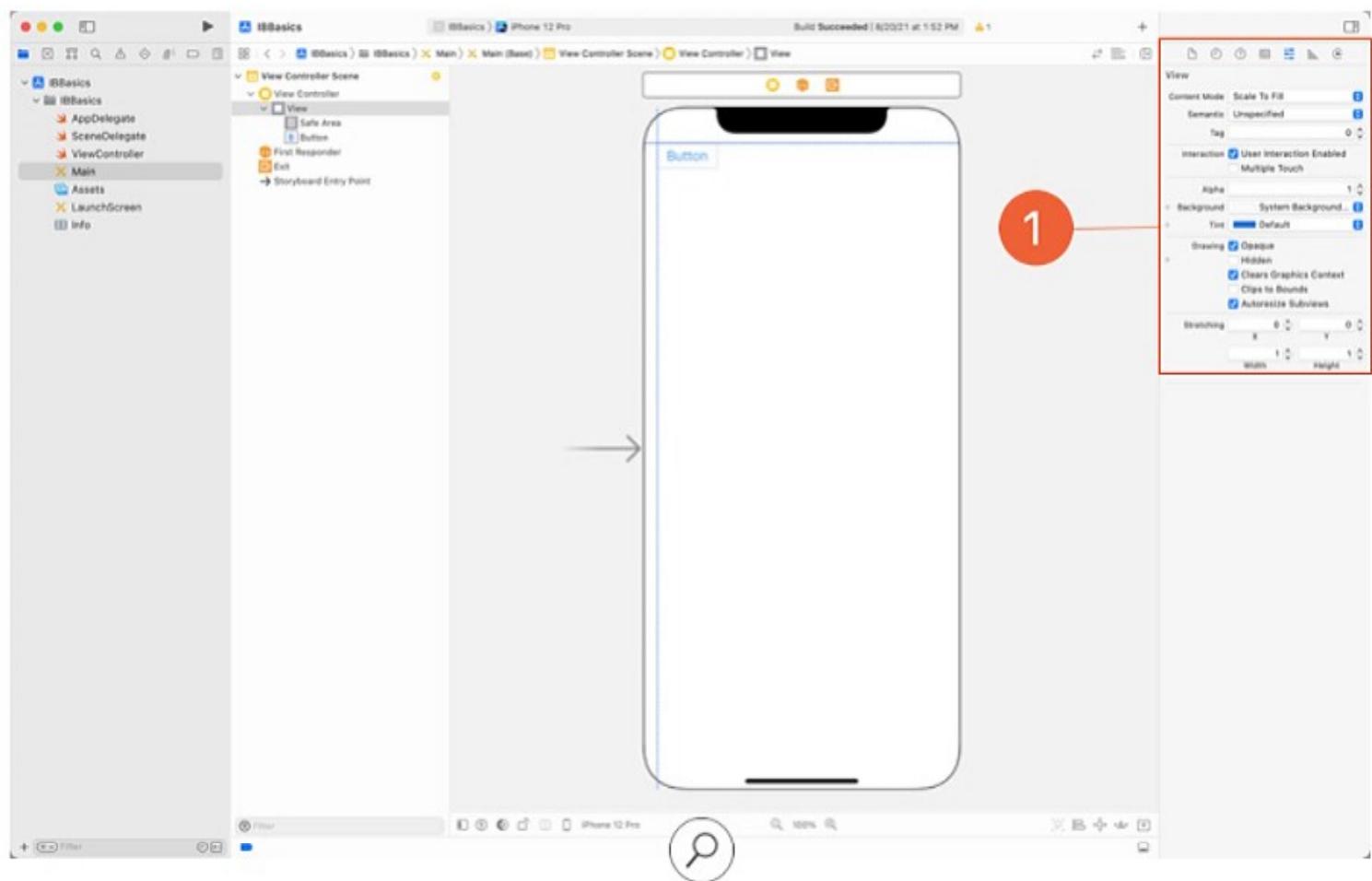
Select the Library button  in the Xcode toolbar to display the Object library. The Object library contains buttons, text fields, labels, and other view controllers that you can add to your scene.

The screenshot shows the Xcode Interface Builder Objects library. On the left, there is a sidebar titled "Objects" with a circular icon. At the top right of the library window are three icons: a blue square with a white circle, a camera-like icon, and a gear-like icon. The main area contains several categories of objects:

- Label**: A blue button labeled "Label". To its right, under the heading "Label", is the description "UILabel: Presents read-only text". Below this is a detailed description: "A label can contain an arbitrary amount of text, but UILabel may shrink, wrap, or truncate the text, depending on the size of the bounding rectangle and properties you set. You can control the font, text color, alignment, highlighting, and shadowing of the text in the label."
- Button**: A standard button labeled "Button".
- Gray Button**: A gray button labeled "Gray Button".
- Tinted Button**: A button with a tinted background labeled "Tinted Button".
- Filled Button**: A button with a solid blue background labeled "Filled Button".
- Pull Down Button**: A button with a dropdown menu icon labeled "Pull Down Button".
- Pop Up Button**: A button with a small arrow icon labeled "Pop Up Button".
- Segmented Control**: A segmented control with two segments labeled "1" and "2" labeled "Segmented Control".

Scroll through the Object library to find and select “Button,” then drag the item from the library onto the view. As you drag the button toward the upper-left corner of the view, try to use the layout guides. Layout guides appear as a dotted blue line when you are placing or resizing a view object in Interface Builder. Layout guides help you center your content, use appropriate margins and spacing between objects, and avoid putting content in the status bar at the top of the screen.

Go ahead and release the mouse or trackpad to insert the button into the view. Notice that the button object now shows up in the Document Outline as well.



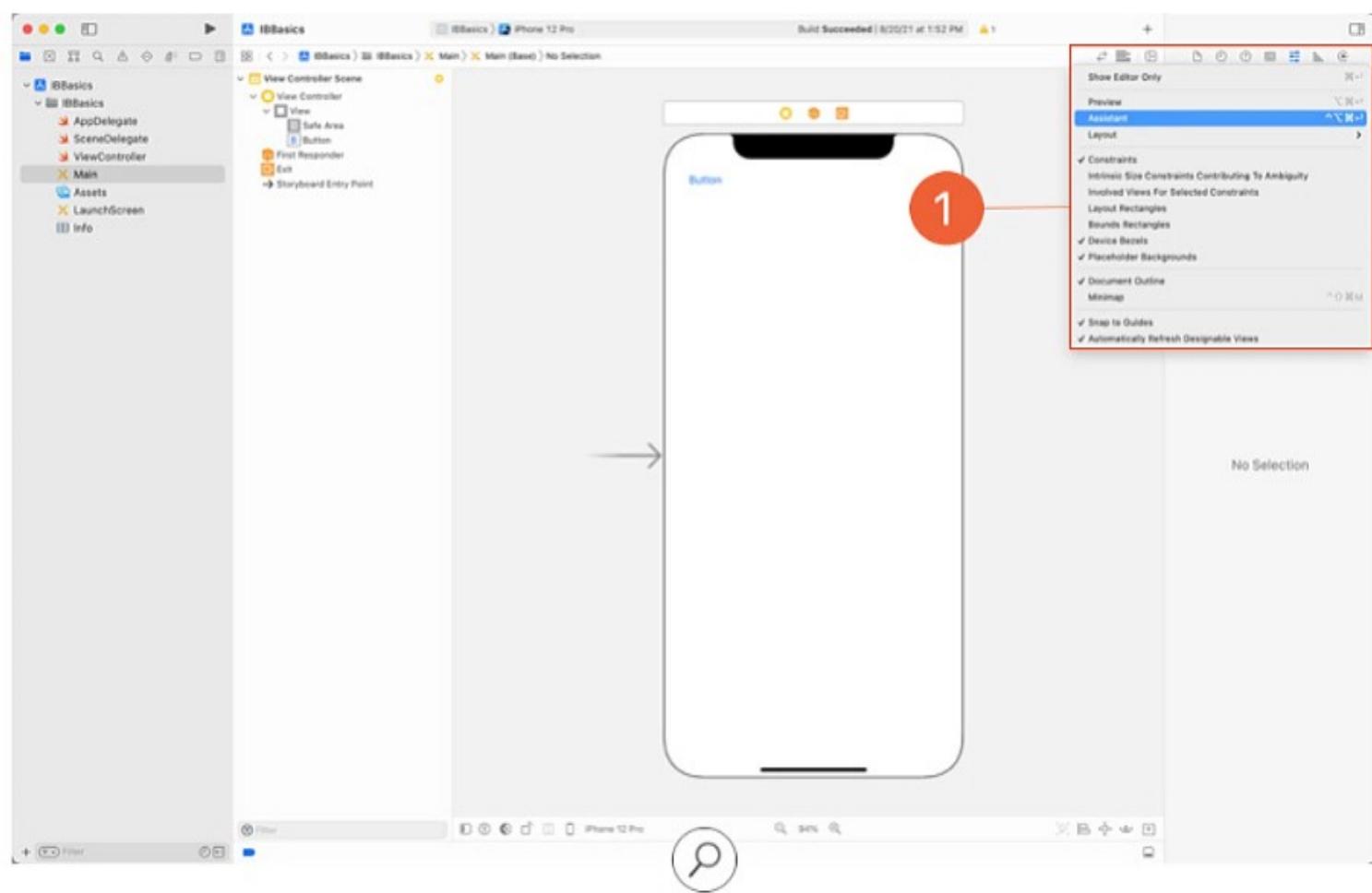
On the right side of the screen, you see the Inspector area. If you don't see it, click the "Hide or Show the Inspectors" button at the top left of the toolbar or use the keyboard shortcut, **Option-Command-0**.

In addition to the File, History, and Quick Help inspectors (which are always available), the top of the Inspector area displays four context-sensitive inspectors when you're in Interface Builder.^① To explore these different inspectors and how they can help you customize the objects in your view, select the button you just added—either in the Document Outline or in the scene itself.

Outlets And Actions

You'll often need a way to reference your visual elements in code so that they can be adjusted at runtime, or when the app is already running. This reference from Interface Builder to code is called an outlet. When you have an object that you want the user to interact with, you create an action—a reference to a piece of code that will execute when the interaction takes place.

Select the view controller in the outline view, then select the Identity inspector. The template you chose when creating the project has set the Custom Class to `ViewController`. Open the assistant editor by clicking the Adjust Editor Options button  and choosing Assistant Editor.

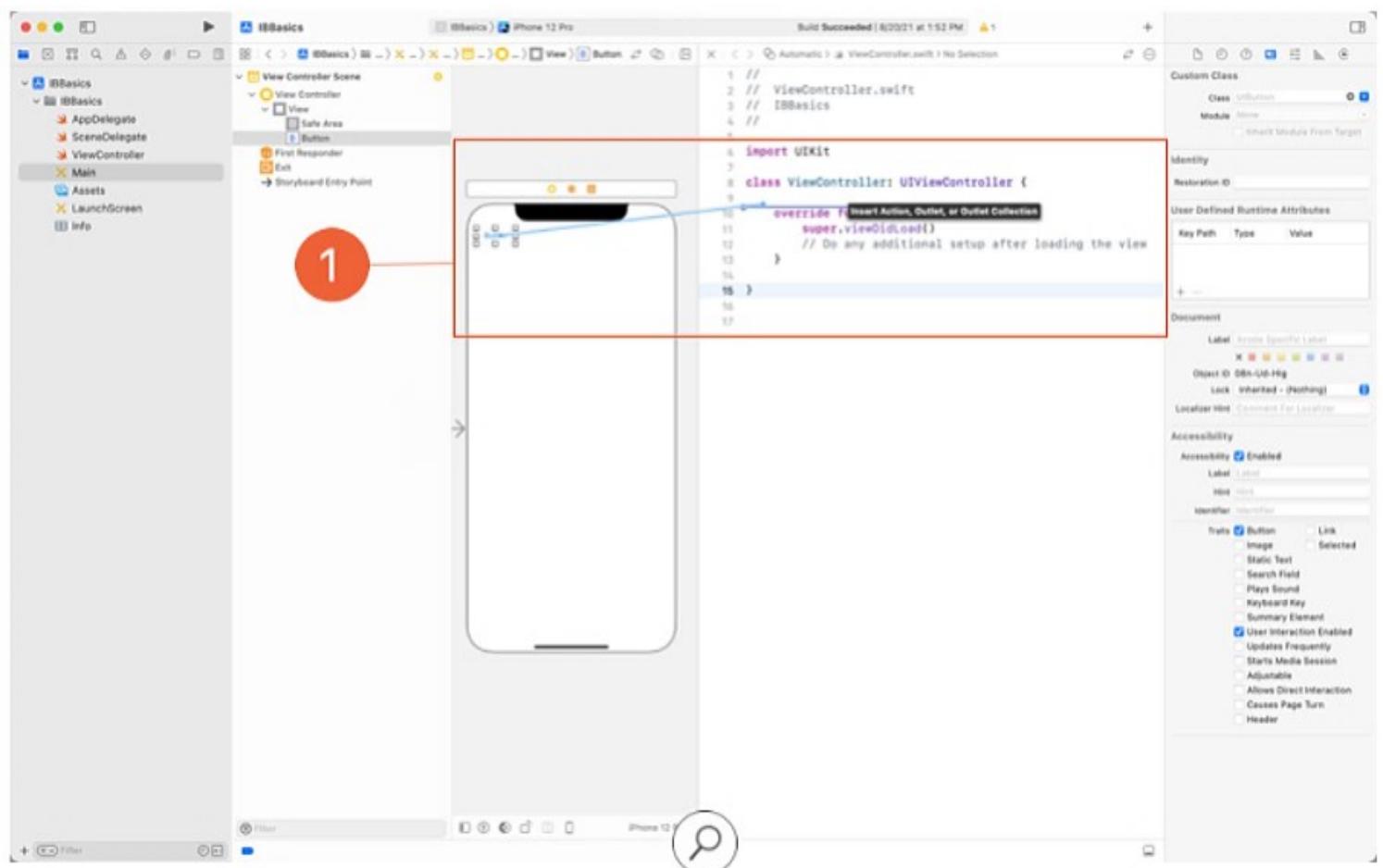


The source code of `ViewController` is displayed alongside the storyboard (because it corresponds to the Custom Class field in the Identity inspector).

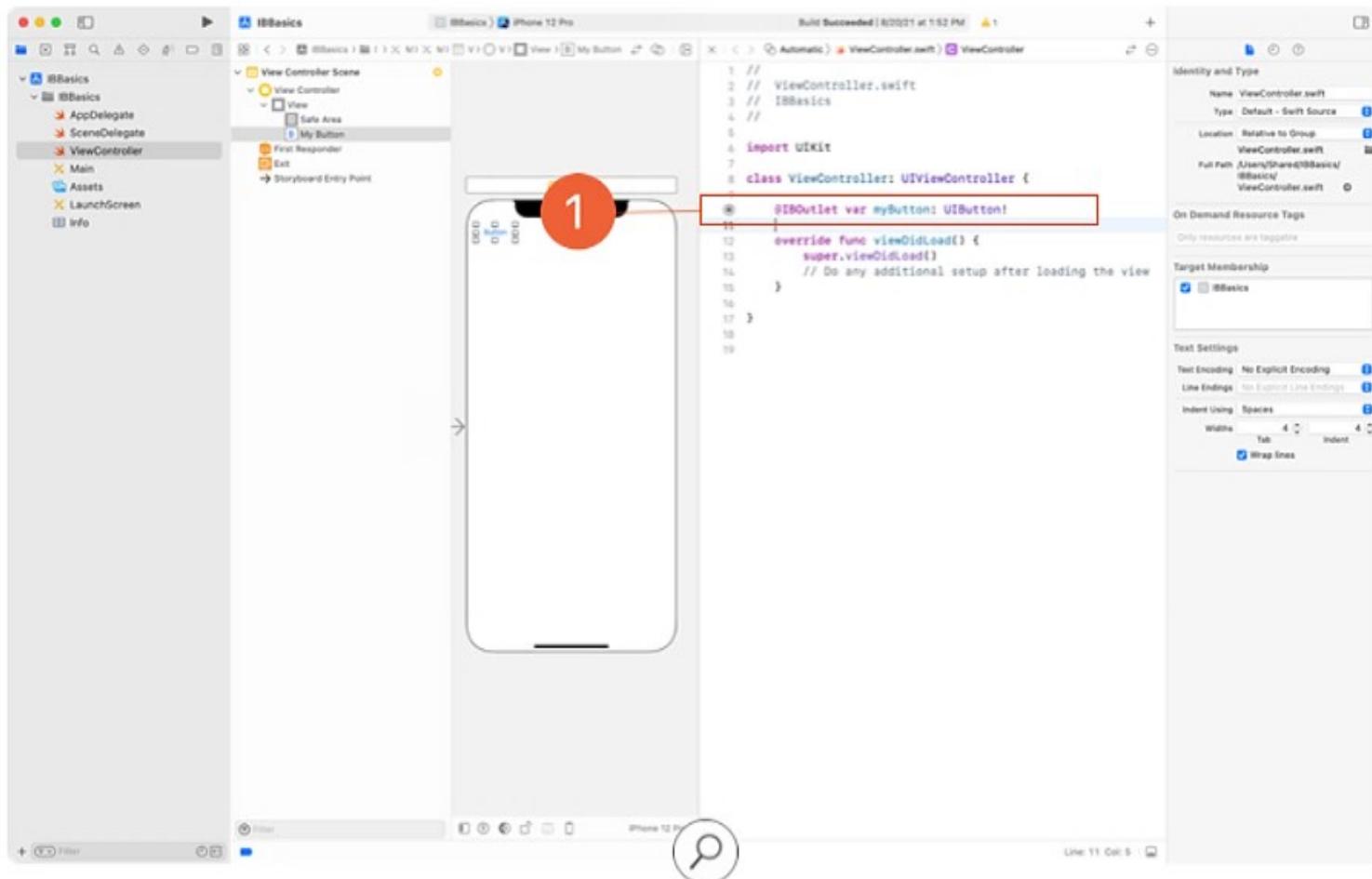
But the `ViewController` class still doesn't have access to the button you added. To make the object accessible in code, you'll need to create an outlet.

Creating an Outlet

Control-click (or right-click) the button in the storyboard, and start dragging toward the assistant editor pane that contains the `ViewController` class definition. As you drag the pointer into the code, you see a blue line. ①



When you release the pointer, the "Outlets and Actions" dialog appears. Make sure that Connection is set to Outlet and Storage is set to Strong. In the Name field, specify a variable name for the button: "myButton." Click the Connect button to finalize the creation of the outlet, generating a line of code that defines the outlet. ①



Now that you have access to the button in code, add the following line inside the `viewDidLoad()` function:

```
myButton.tintColor = .red
```

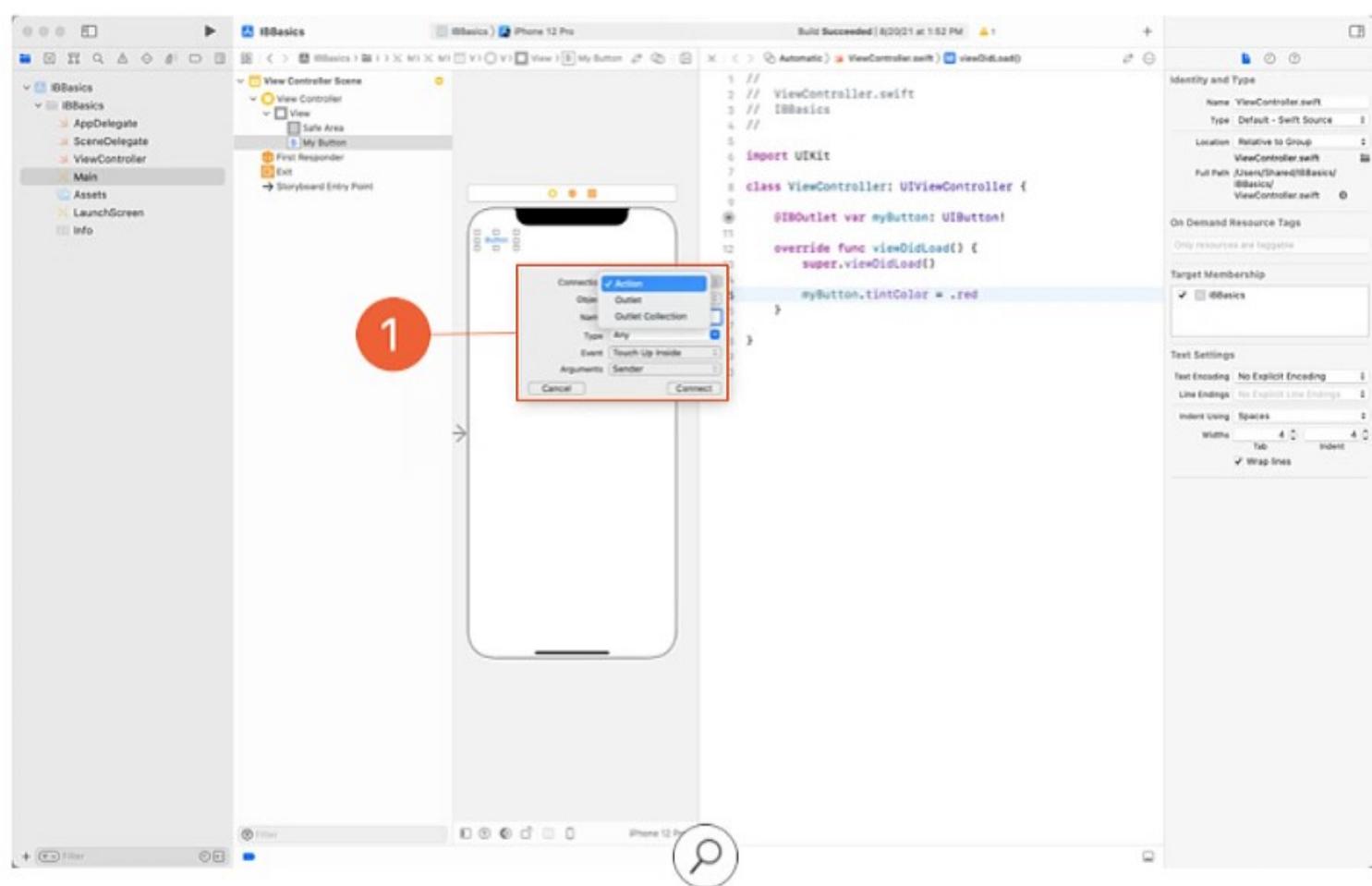
This line changes the color of the button's title from blue to red. Build and run your application to see the change take effect.

That's great, but you'll probably notice that nothing happens when you try clicking the button. To add functionality, you'll need to create an action that's tied to the button.

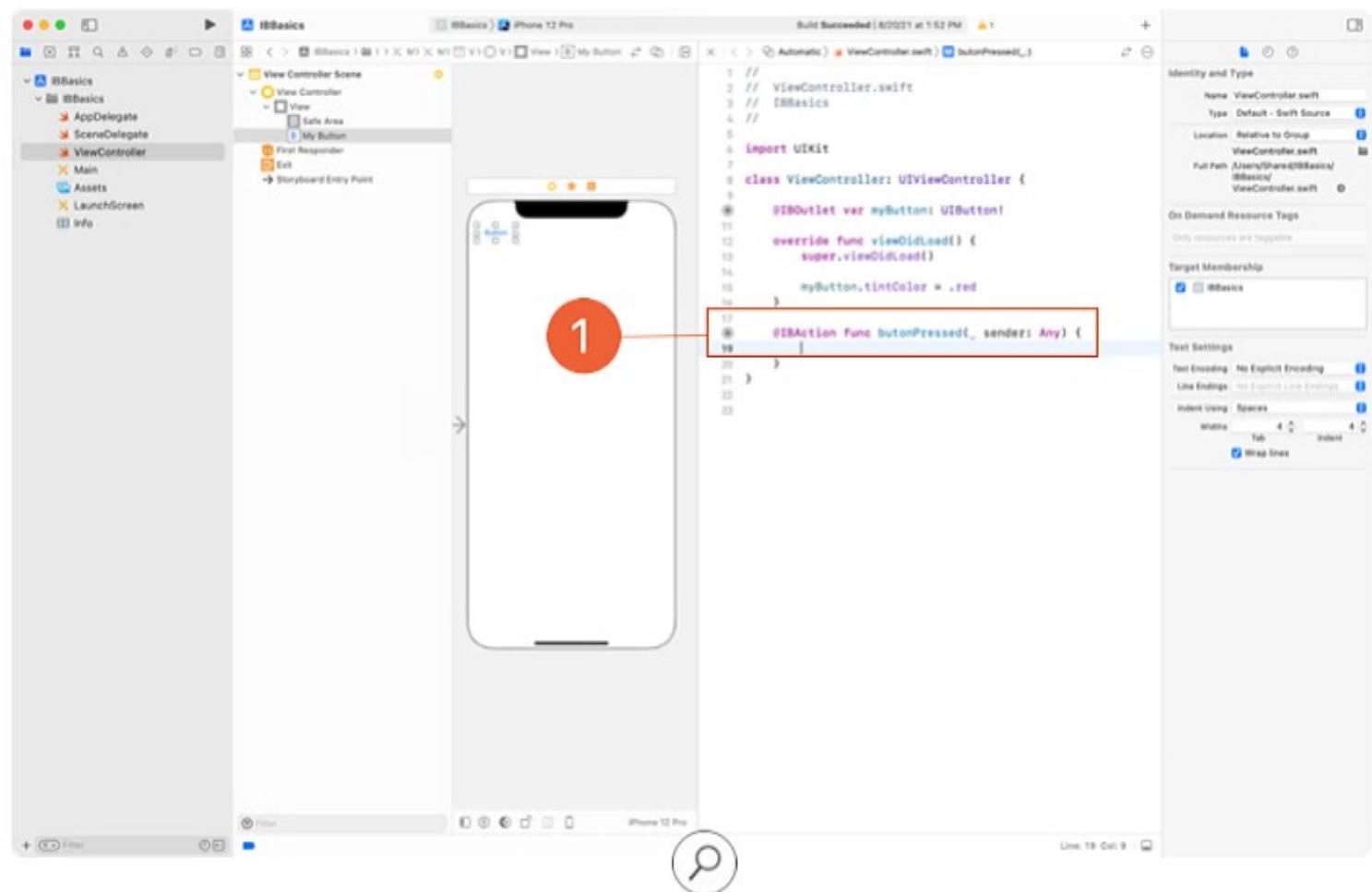
Creating an Action

Once again, **Control-click** (or right-click) the button in the storyboard, and drag the cursor into the `ViewController` class definition.

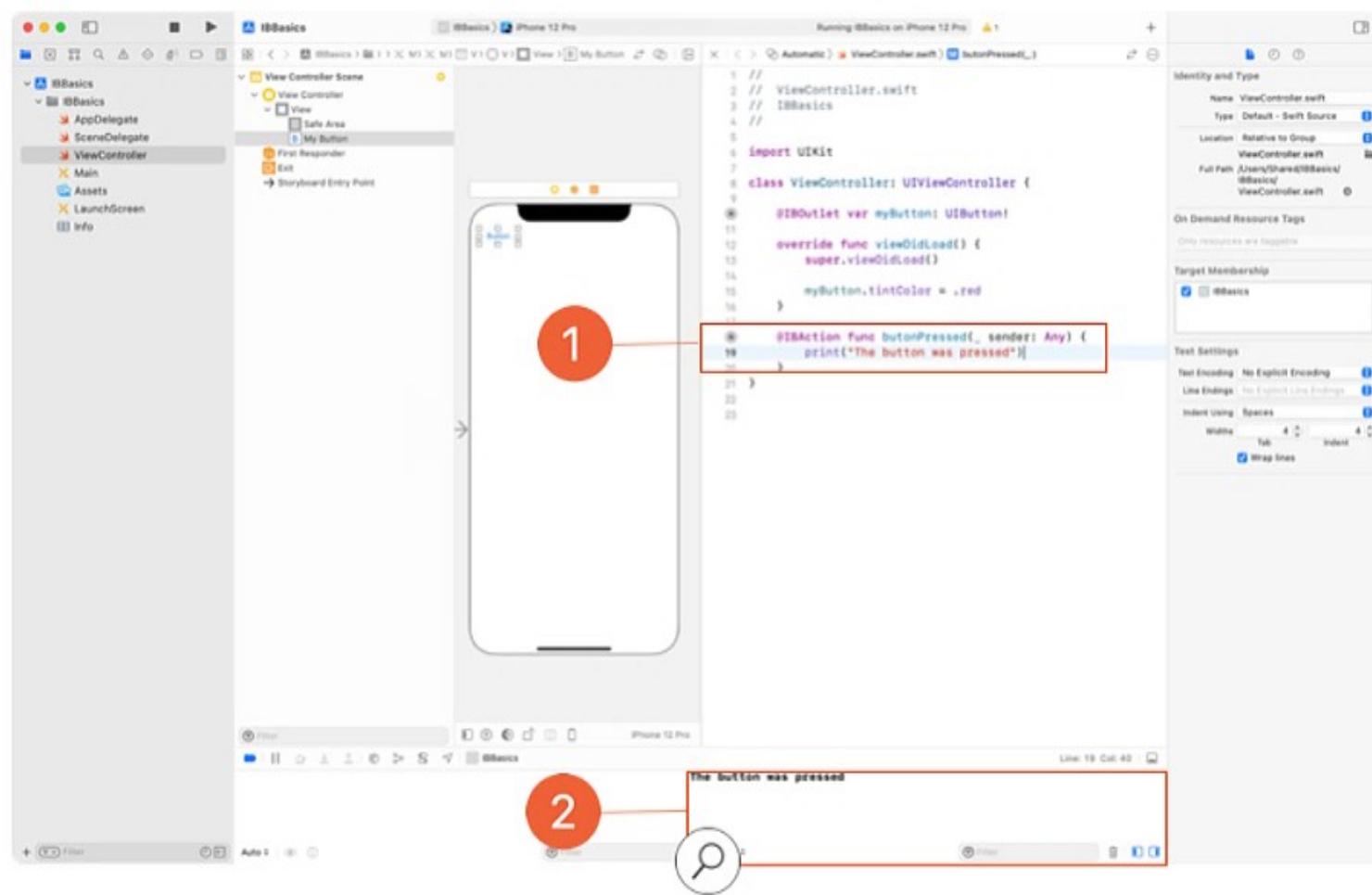
When the “Outlets and Actions” dialog appears, set Connection to Action.^① In this situation, your entry in the Name field doesn’t define a variable; it defines the action the button tap is tied to. Name the action “buttonPressed.” Click the Connect button to finalize the creation of the action.



Check out the new line of code, from left to right: ①



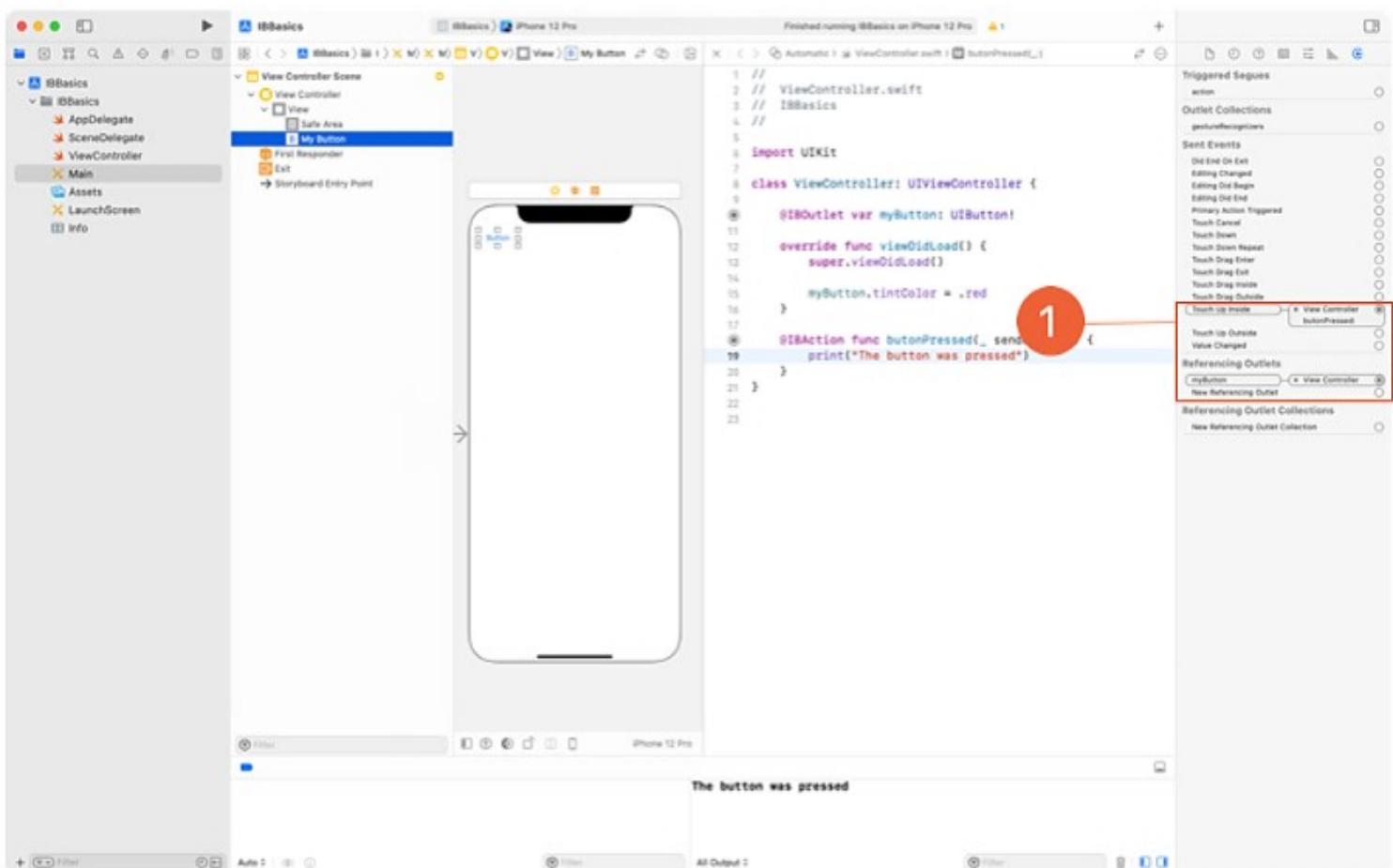
You now have a function to execute whenever the button is tapped. To test it, add the following line inside the `buttonPressed` function: ①



```
print("The button was pressed")
```

This code prints a message to the Xcode console whenever the function is executed. Build and run your app, then click the button to see the message print in the console at the bottom right of the screen. ②

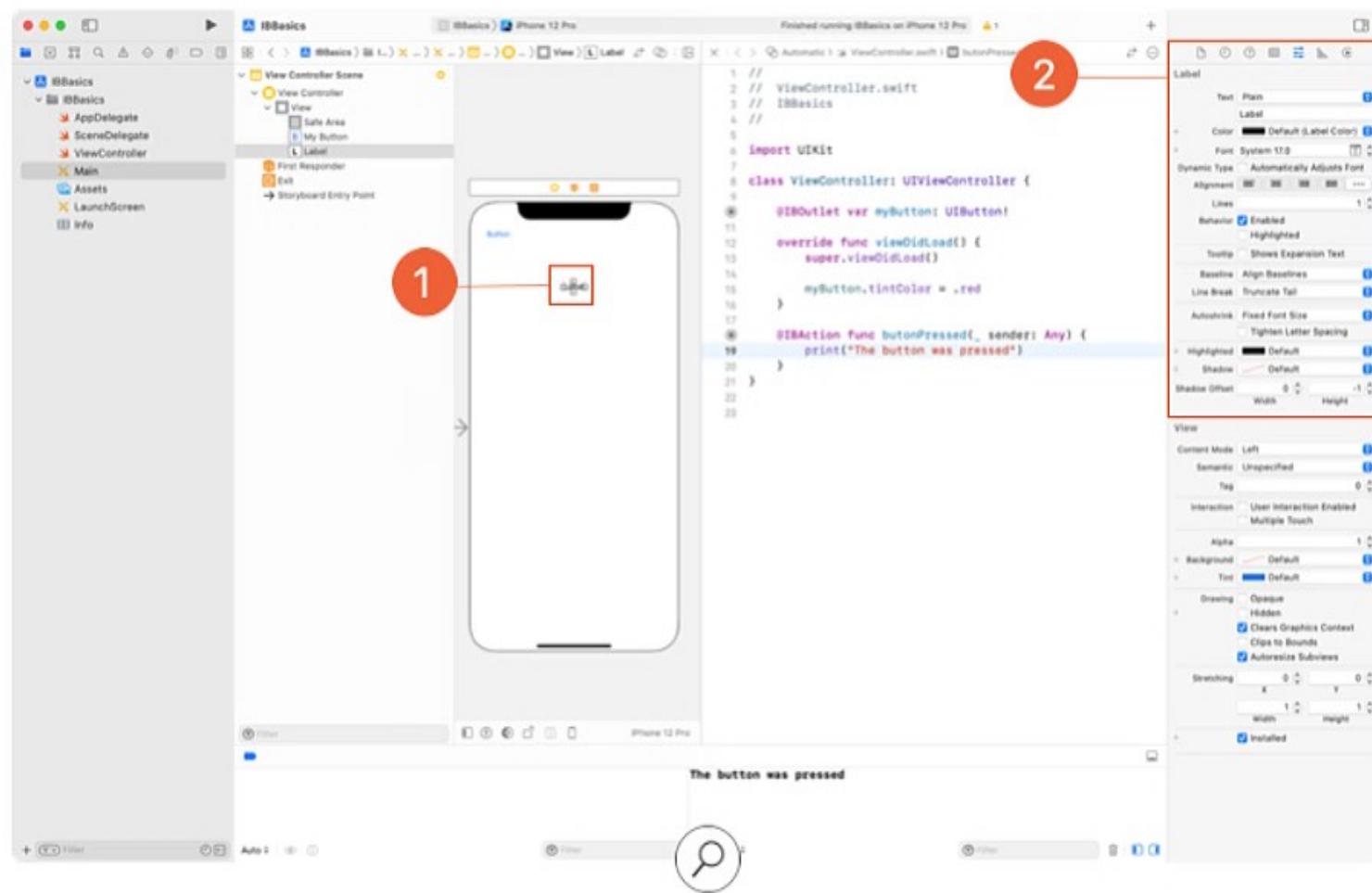
Back in the outline view, select the button again, then select the Connections inspector. Now that you've wired up an outlet and an action to the button, you'll see them both in the Connections pane. ①



A Note About Interface Builder

Every attribute in Interface Builder represents a property that can also be configured programmatically, or in code. Interface Builder is simply a graphical interface for configuring and setting properties on UIKit classes that are displayed in your app.

Add a label to your scene, ① then look at the Attributes inspector for the label. ②



Now look up the symbols, or properties and functions, for `UILabel` in the Documentation Browser. You'll find that each setting in Interface Builder has a companion property.

Interface Builder Attribute	Property Name
Text	<code>text</code>
Color	<code>textColor</code>
Dynamic Type	<code>adjustsFontForContentSizeCategory</code>
Font	<code>font</code>
Alignment	<code>textAlignment</code>
Lines	<code>numberOfLines</code>
Enabled	<code>enabled</code>
Highlighted	<code>isHighlighted</code>
Baseline	<code>baselineAdjustment</code>
Line Break	<code>lineBreakMode</code>
Autoshrink	<code>adjustsFontSizeToFitWidth</code>
Tighten Letter Spacing	<code>allowsDefaultTighteningForTruncation</code>
Highlighted	<code>highlightedTextColor</code>
Shadow	<code>shadowColor</code>
Shadow Offset	<code>shadowOffset</code>

Many objects that you can configure in Interface Builder have properties that can only be set programmatically. For example, `UIScrollView` has a `contentSize` property that does not have a matching option in the Attributes inspector.

When you need to adjust one of these settings, you can do so programmatically by setting up an `@IBOutlet` for the scroll view and updating the properties using dot-notation.

```
scrollView.contentSize = CGSize(width: 100, height: 100)
```

In fact, everything that you can do in Interface Builder can also be done programmatically, including setting up all child views and adding them to the screen.

```
let label = UILabel(frame: CGRect(x: 16, y: 16, width: 200,  
height: 44))  
view.addSubview(label) // Adds label as a child view to 'view'
```

This type of setup is most commonly done in the `viewDidLoad()` method of a view controller, which gives you access to the view controller's main `view` property before it's displayed on the screen.

While you *can* do everything programmatically, you can see that Interface Builder can save you a lot of time when setting up complex views. As your projects grow, storyboards help you more easily maintain your interface setup. Additionally, Interface Builder has support for building complex views that support multiple device sizes, all in one place.

You will learn much more about Interface Builder and storyboards as you work through the rest of the course.

Lab—Use Interface Builder

The objective of this lab is to use Interface Builder and the assistant editor to create a basic view.

Step 1

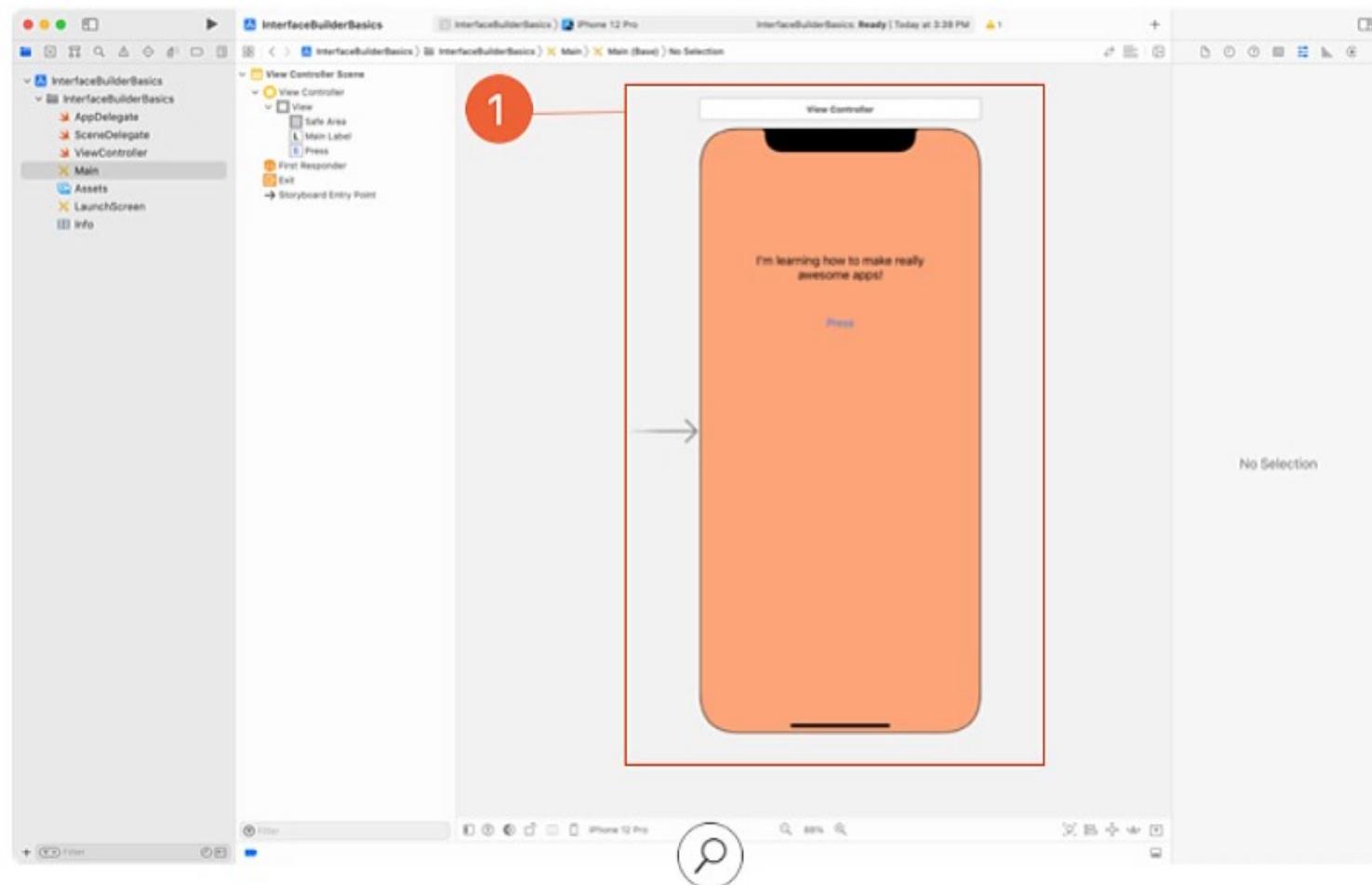
Create An Xcode Project

- Create a new Xcode Project called “InterfaceBuilderBasics” and save it in your project folder.

Step 2

Create A Simple View With Interface Builder

- Open Interface Builder by clicking the **Main** storyboard in the Project navigator.
- Click the Devices icon at the bottom of the window (the current selection is listed next to the icon) and select **iPhone 13 Pro** from the popup that appears. This allows you to see your interface layout as it will appear on the iPhone 12 Pro and iPhone 13 Pro simulators.
- Use what you learned in the lesson to recreate the following image. You’ll need to use a label and a button. ①



Step 3

Use The Assistant Editor To Connect Your View

- Now create an assistant editor and make sure that it's displaying the file named `ViewController`. Create an outlet from your label and name it `mainLabel`.
- Create an action from your button and call it `changeTitle`.
- Inside `changeTitle`, write `mainLabel.text = "This app rocks!"`. Run the app and tap the button. What happened to the text?

Congratulations! You should now have a simple view with text that you can swap out in code. Be sure to save your project to your project folder.

Review Questions

Question 1 of 10

How is an XIB different from a storyboard?

- A. A XIB represents a single view object, while a storyboard represents multiple views, scenes, and how to navigate between them
- B. A XIB represents multiple views, scenes, and how to navigate between them, while a storyboard represents a single view object

[Check Answer](#)



Guided Project: Light

So far in this lesson, you've learned the basics of Xcode and Interface Builder. You'll now apply your knowledge of these tools by building a project.

By the end of this Guided Project, you'll have created an app, called Light, that changes the screen from black to white, and back again, whenever the user taps a button. To successfully build the light, you'll need to use Xcode documentation, set breakpoints, and create outlets and actions.

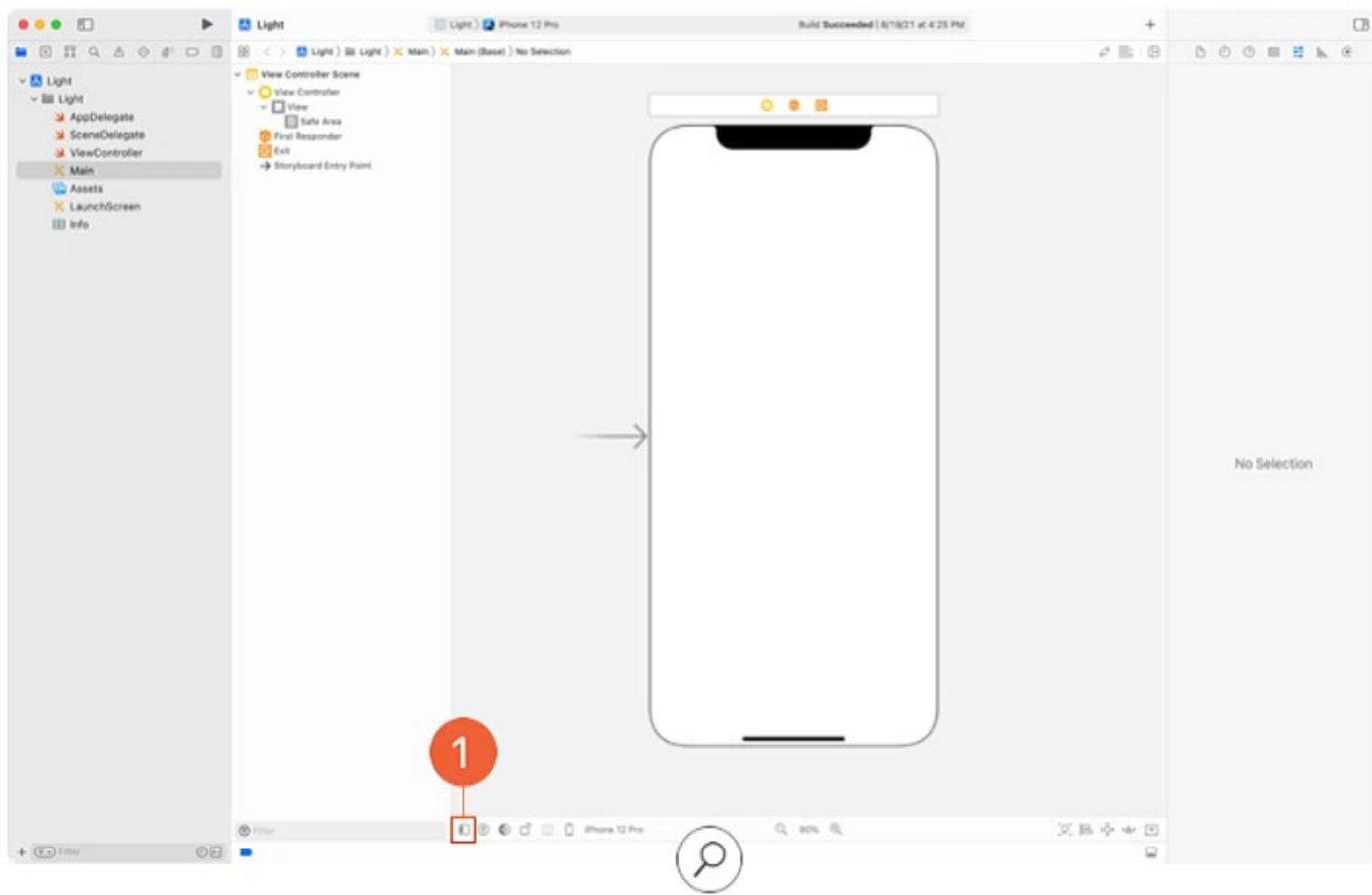
This project will involve modifying some code—even though you're relatively new to the Swift language. Don't be discouraged if you struggle with the code-specific pieces of the project. Just keep at it!

Part One

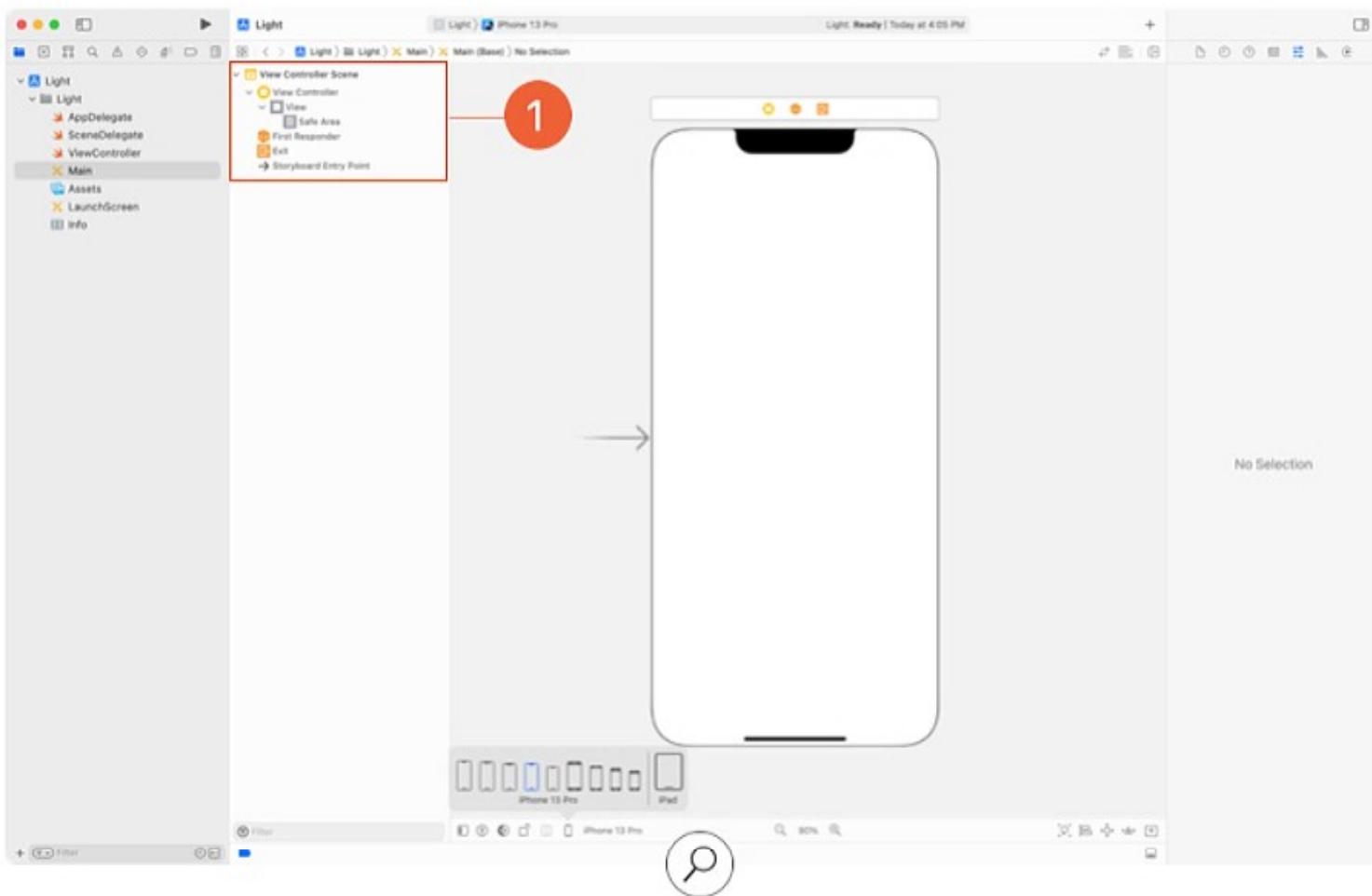
Create A Button And An Action

Create a new Xcode project using the iOS App template that you've used in previous lessons. When creating the project, make sure the interface option is set to Storyboard. Name the project "Light." When you build and run the project, you'll notice there's nothing for the user to interact with. You'll be changing that soon.

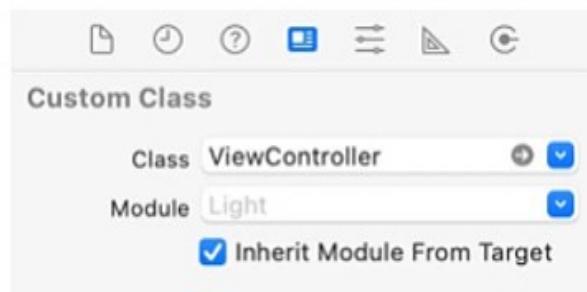
Select the **Main** storyboard in the Project navigator to open your storyboard in Interface Builder.



The initial interface created by Xcode includes an instance of `ViewController`. `ViewController` is a subclass of `UIViewController` that comes predefined as part of the iOS App template and is listed in the Project navigator. You can click the Show Document Outline button to reveal all the view controllers defined in the `Main` storyboard.



Select View Controller in the Document Outline 1, then click the Identity inspector button at the top of the Inspector area to confirm that this particular view controller is of type ViewController.



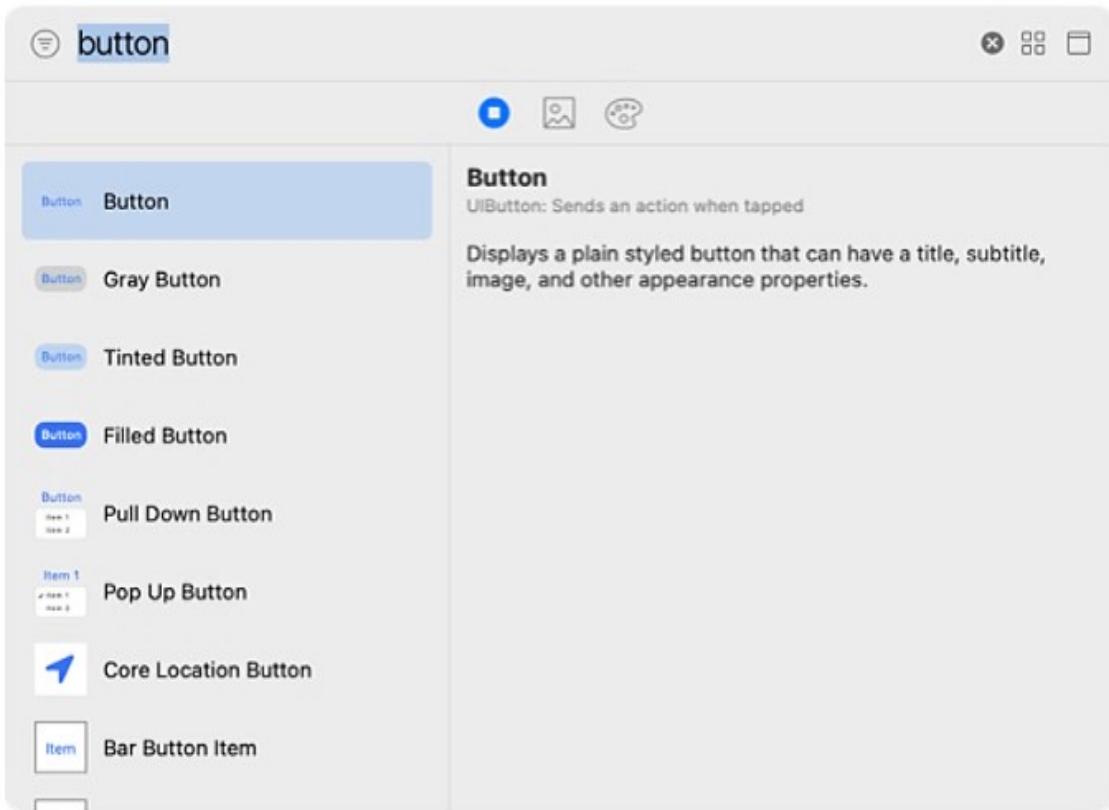
If you're using Simulator, use the Devices button at the bottom of the canvas (the current selection is shown next to the button) to adjust the size of the canvas. Set the canvas size to the iPhone 13 Pro configuration, and select the iPhone 12 Pro or iPhone 13 Pro simulator from the menu toward the left end of the Xcode toolbar.

If you're using your own device for this project, the size of the view controller may not be identical to the size of your screen. Use the "View as" button at the bottom of the canvas to adjust the size of the canvas to match your device, but leave it in portrait orientation.

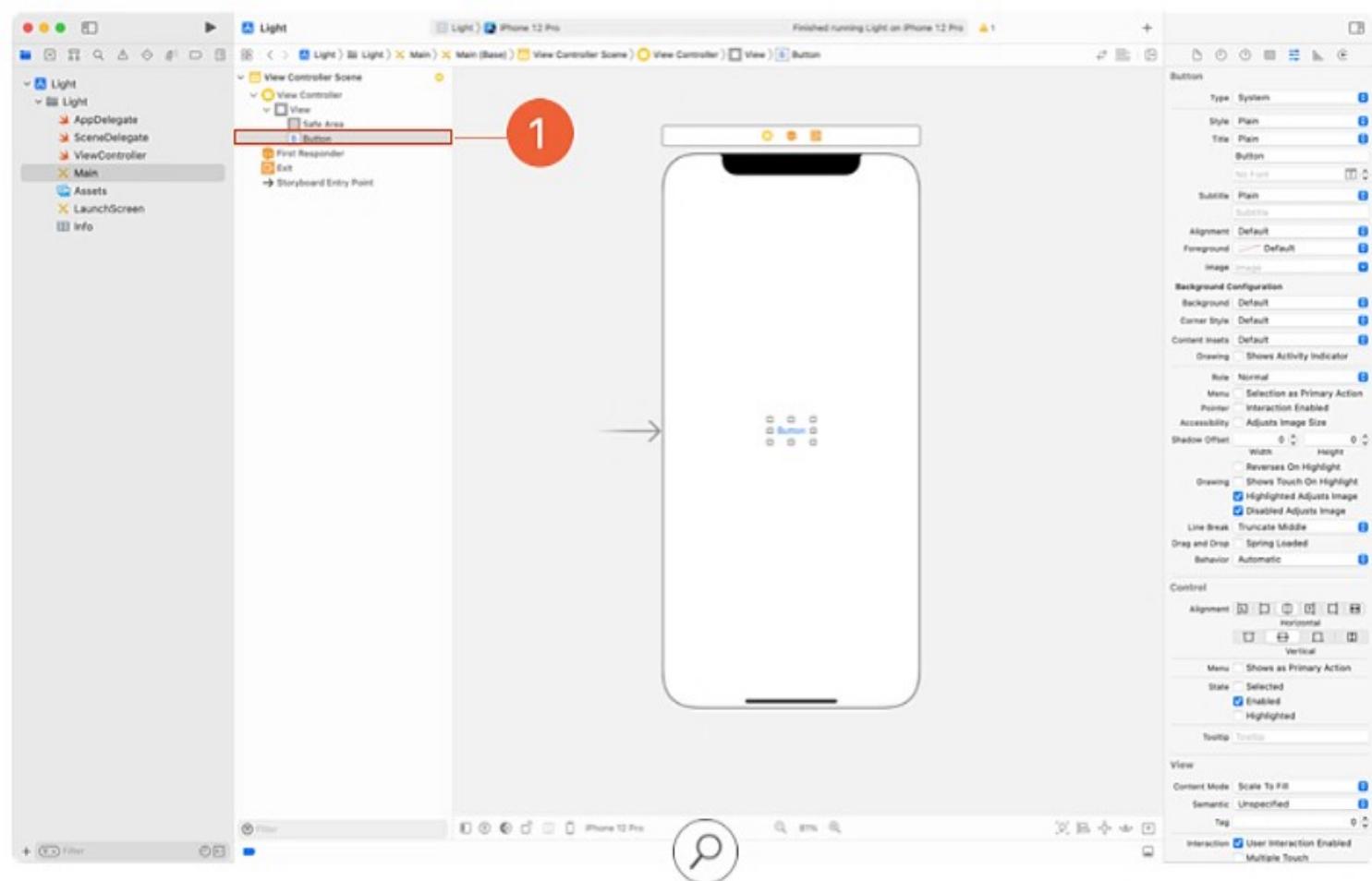


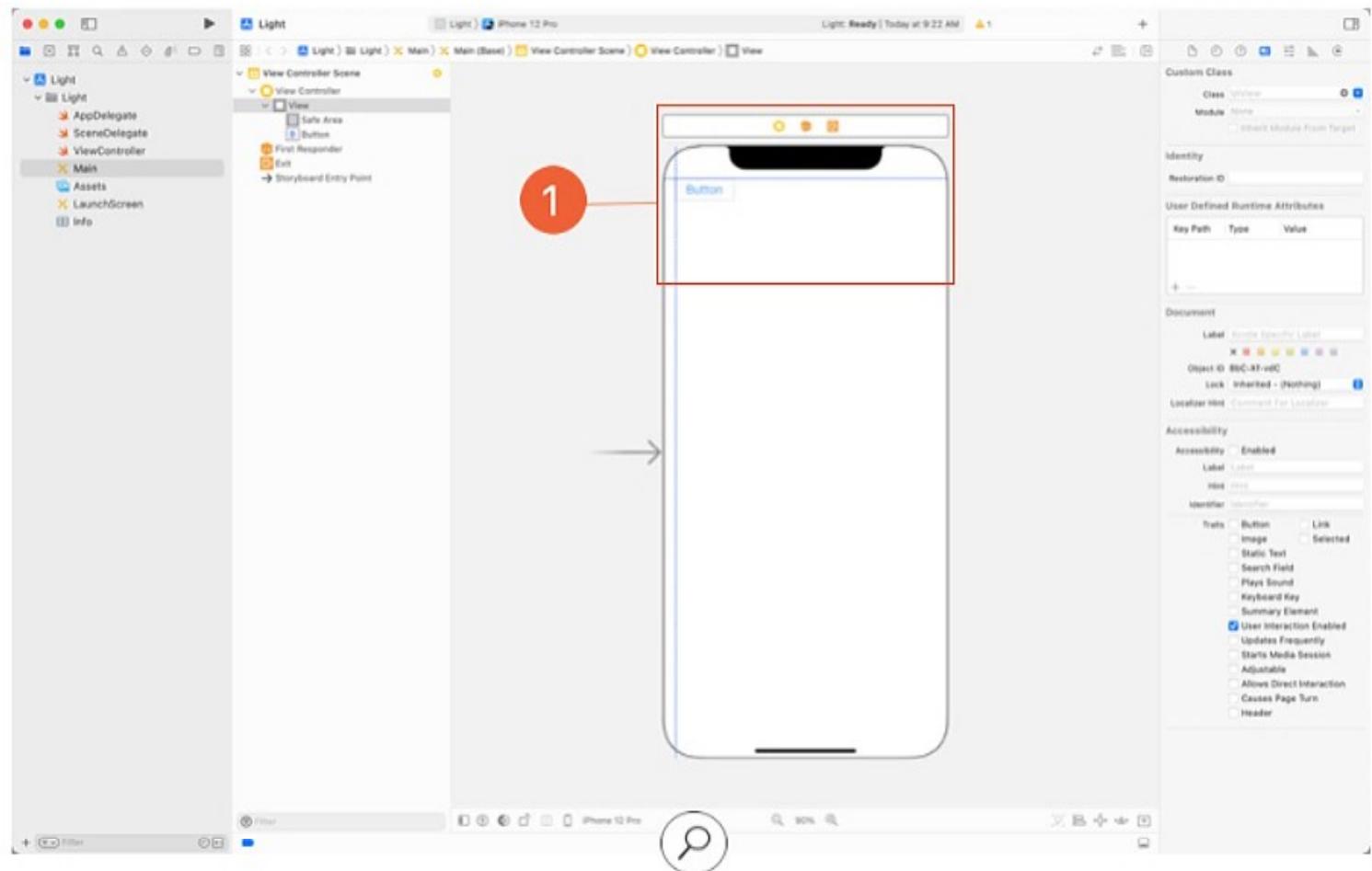
Select the view controller's view, then select the Attributes inspector , which will allow you to customize the attributes of any interface element. The background color for this view has already been set to white. That's the desired state of your app on launch, so you don't need to change anything right now.

Next, add a button that'll be used to change the view's background color, simulating a light turning on and off. Press the + button at the top right of the toolbar to open the Library. As you learned in an earlier lesson, this library includes a list of common objects you can add to a storyboard. Scroll through the list, or use the search filter at the top of the library area.



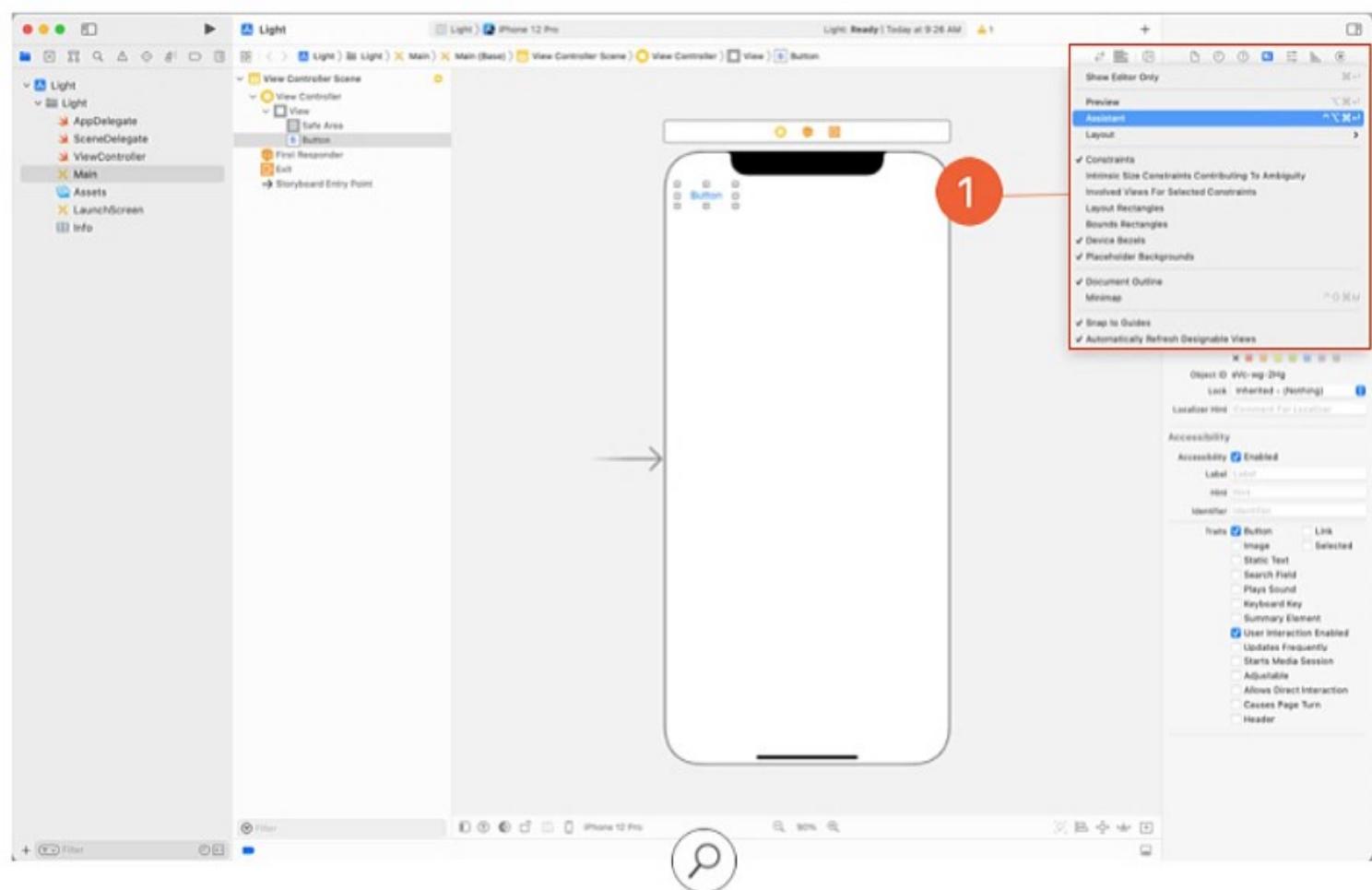
Find the “Button” object in the library and drag it over on top of the view. When you release the cursor, the Document Outline should indicate that the button has been added as a subview. ①

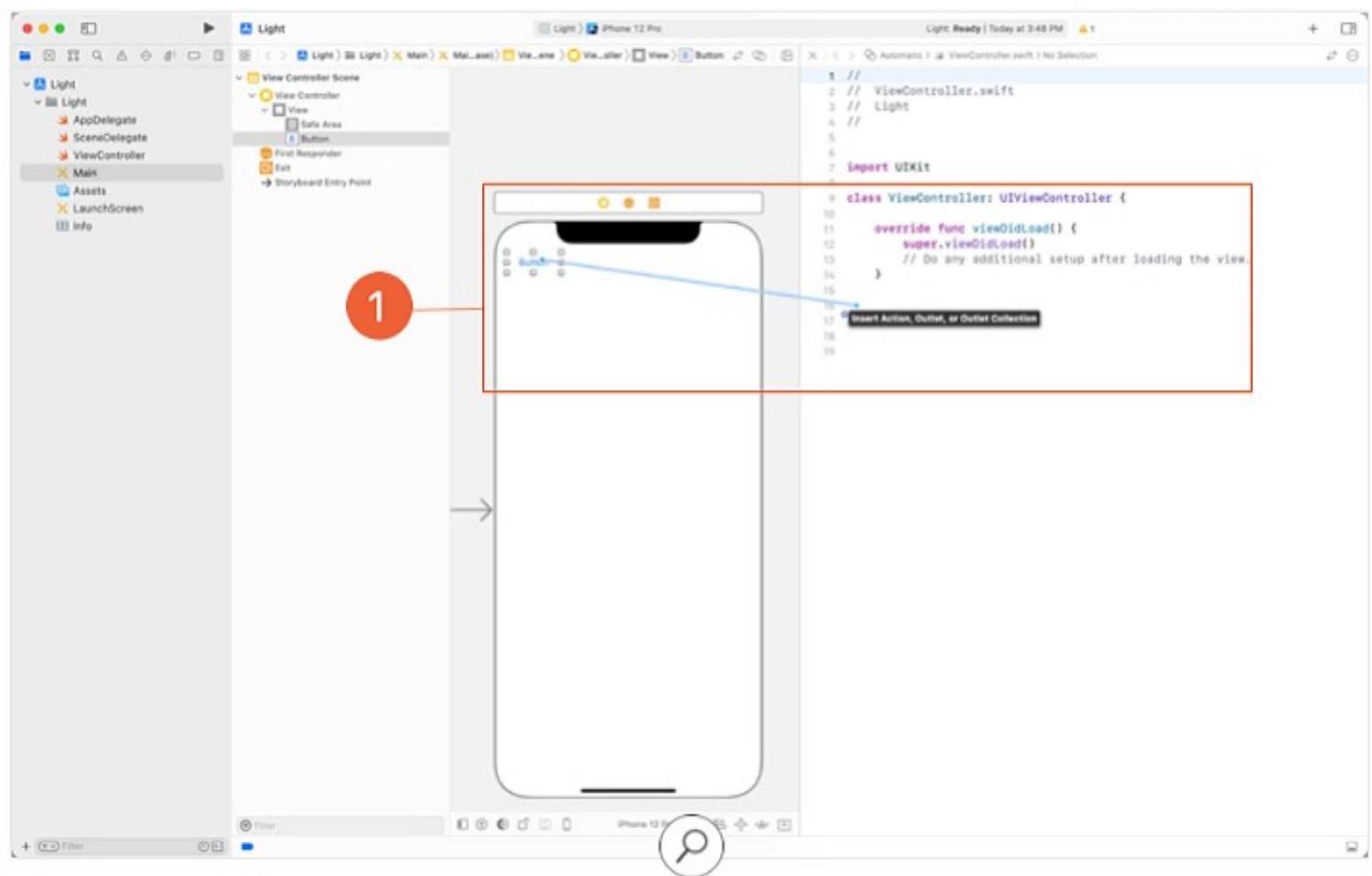




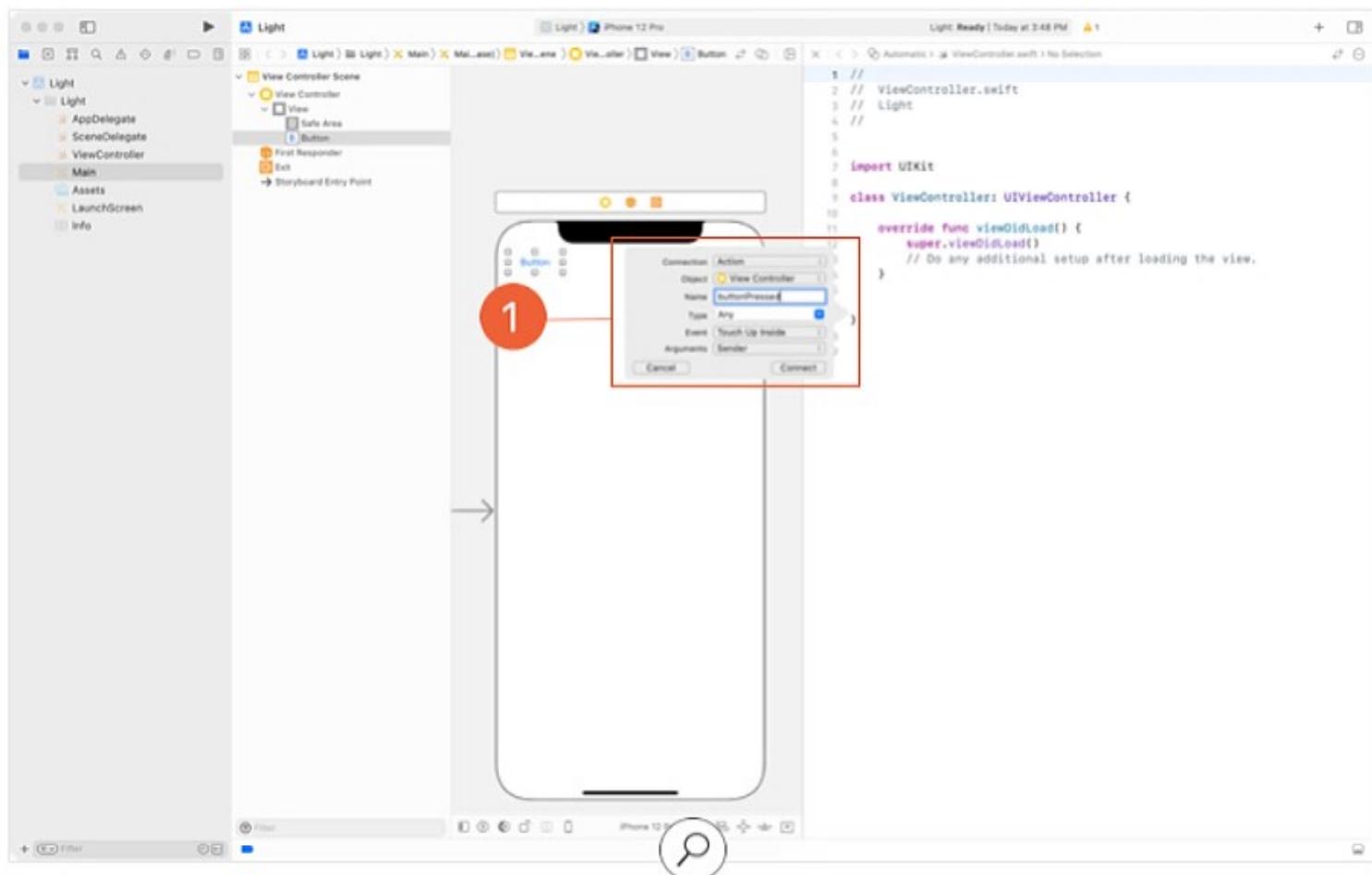
Move the button to the upper-left corner of the view. Margin-alignment guides will help snap the object into the correct position. ①

Give your button an action to perform when it's clicked or tapped. Add an assistant editor using the Adjust Editor Options button and choose Assistant Editor to split the Xcode workspace into two parts: Interface Builder and its corresponding code.



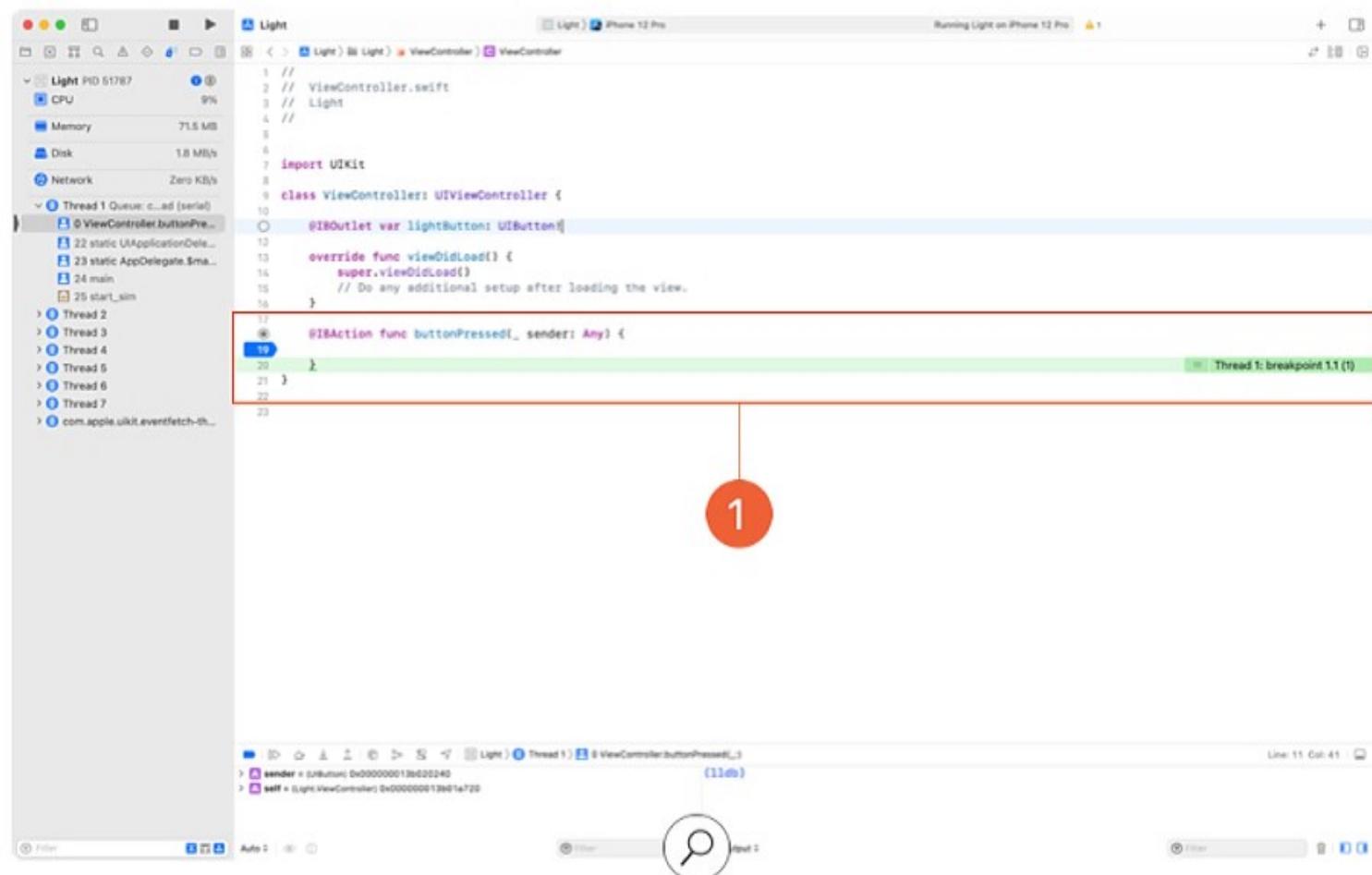


Now **Control-drag** (or right-click) the button into an available area within the `ViewController` class definition. As you drag, a blue line extends from the button and a blue horizontal bar will display below your cursor, indicating a valid place to create a connection. ①



When you release the mouse cursor, you will be prompted to create an outlet or define an action. Change Connection to Action, then set Name to “buttonPressed.” ^① When you click the Connect button, Xcode will create a new method that will be called whenever the button is tapped. You may notice the `@IBAction` keyword right before the `buttonPressed(_:)` method. `@IBAction` signals to Xcode that a relationship can be created between a visual element in a storyboard and the function.

Build and run the app. When you press the button, nothing happens, because there's no code within the `buttonPressed(_ :)` method to execute. To verify that the method is being called, you can place a breakpoint within the method definition. Now press the button and the breakpoint will be triggered. ①



Now that you've verified that the action is working correctly, you can add some code to change the background color. Remove the breakpoint before proceeding.

Part Two

Change The Background

If the light is on (that is, if the background color is white) then the light should be switched off. Otherwise, if the light is off (the background is black), then the light should be switched on. The statement "the light is on" is either a true or false statement, so it would make sense to use a Boolean to determine how to change the background color.

Near the top of the `ViewController` class definition, create a variable called `lightOn` and set the initial value to `true`, since the screen starts off with a white background.

```
var lightOn = true
```

Each time the button is tapped, the value should change from `true` to `false`, or from `false` to `true`. Swift booleans have a method, `toggle()`, that does precisely this. Since the `buttonPressed(_:)` method is called whenever the tap is executed, make the change there.

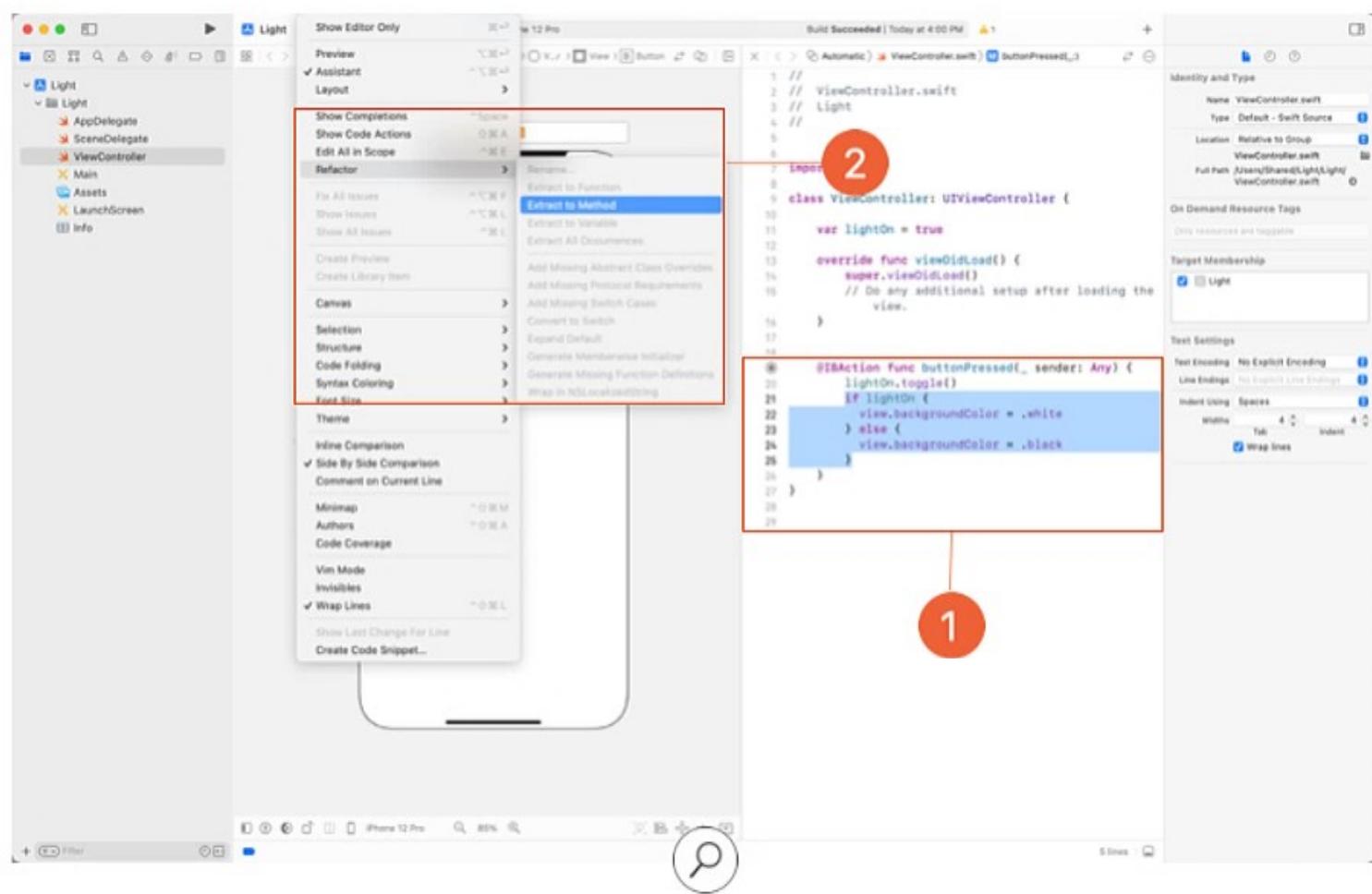
```
@IBAction func buttonPressed(_ sender: Any) {  
    lightOn.toggle()  
}
```

After the value has been changed, you can use the new value to determine how to change the background color. If the light is supposed to be off, change it to black. Or if it's supposed to be on, change it to white. You can write this logic using a simple if-statement.

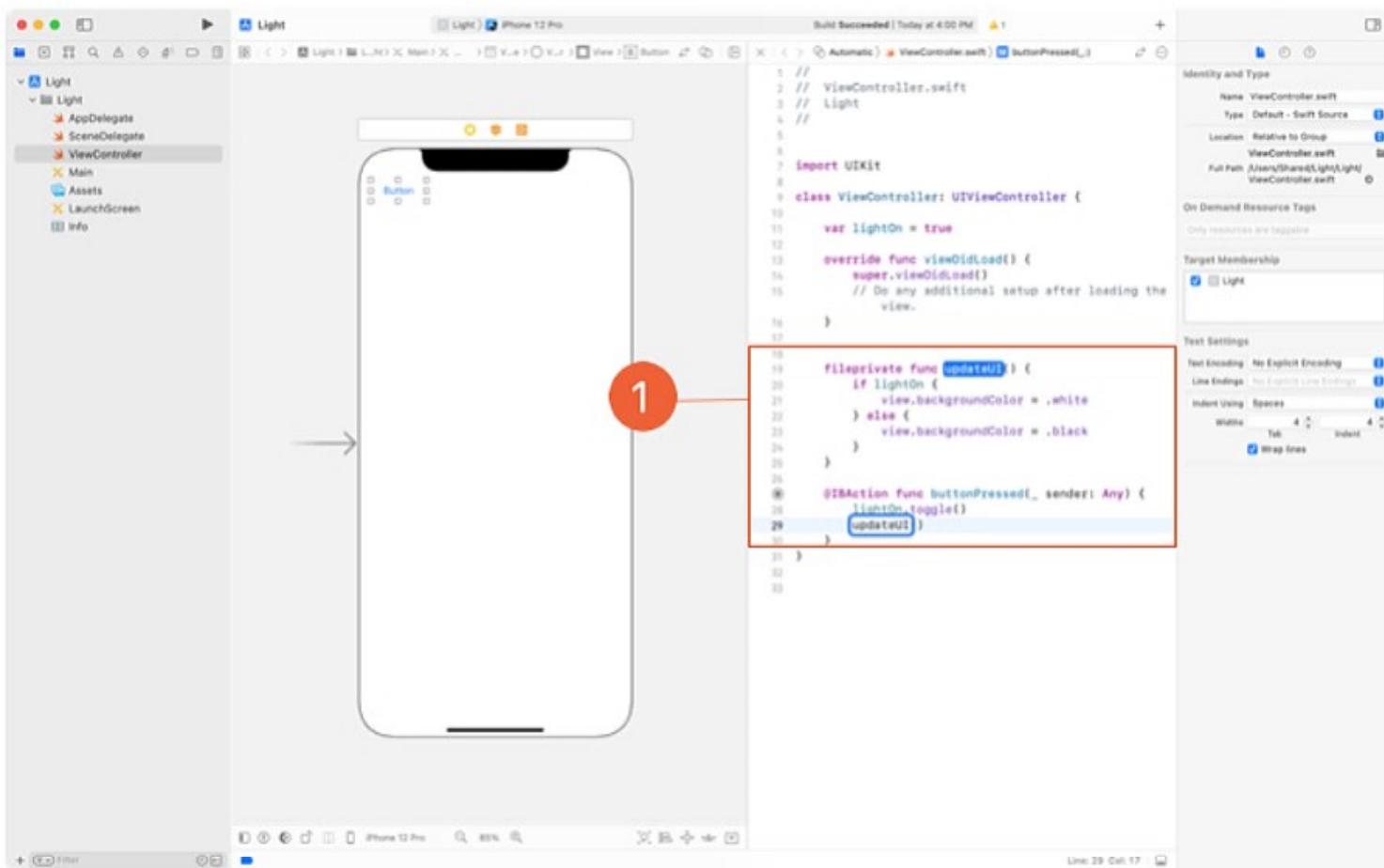
```
@IBAction func buttonPressed(_ sender: Any) {  
    lightOn.toggle()  
    if lightOn {  
        view.backgroundColor = .white  
    } else {  
        view.backgroundColor = .black  
    }  
}
```

Build and run your application, and the background should successfully change on each press. Great job!

As your application's size continues to grow, make sure that your code stays organized. Rather than put the if-statement directly inside the `buttonPressed(_:)` method, you can move it to a new method that handles updating the entire user interface.



Select the entire if-statement, ① and then choose **Editor > Refactor > Extract to Method**. ②



Xcode moves the code out of place into a new method and enters a special editing mode that allows you to set the new method's name and update where it's called at the same time. Type `updateUI` and press the Return key to set the name. ①

Xcode moved the selected code into a new method named `updateUI()`, and then added a call to `updateUI()` where the code once lived.

The result should look like the following:

```
fileprivate func updateUI() {
    if lightOn {
        view.backgroundColor = .white
    } else {
        view.backgroundColor = .black
    }
}

@IBAction func buttonPressed(_ sender: Any) {
    lightOn.toggle()
    updateUI()
}
```

Notice the `fileprivate` keyword that Xcode used in front of the method name. Swift uses this special access modifier to specify where the method can be called from. If you use the Xcode refactor feature and need to call your method outside of the file it's defined in, remove the `fileprivate` keyword in front of it.

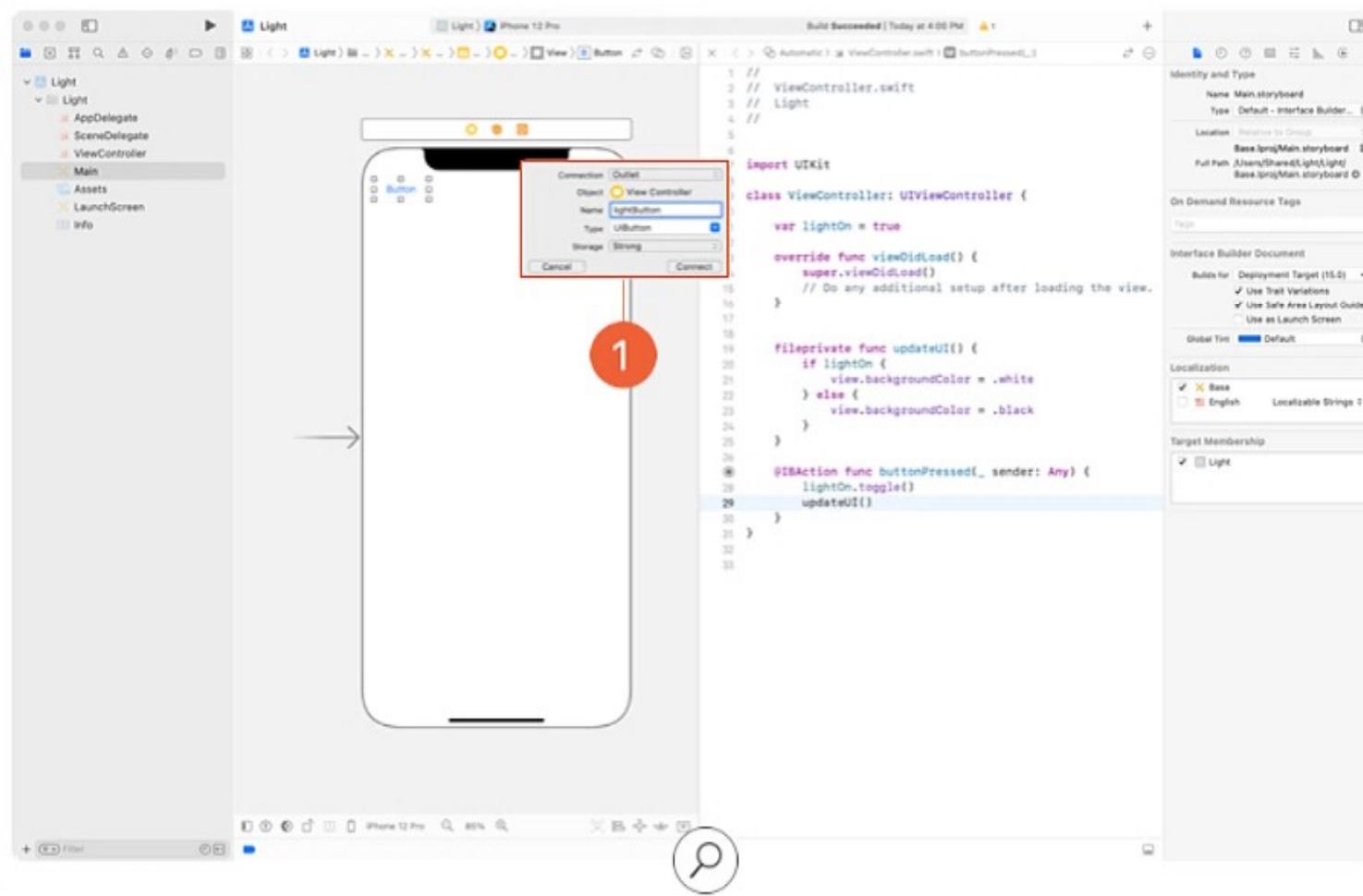
This is a small change for a small app. You will see this pattern persist throughout this book, and you will find it much easier to debug interface issues if the code that updates the view is all contained in a single method.

Part Three

Update The Button Text

You've now successfully set the button to change the background color of the view, but the title of the button text remains the same no matter the state of the light. At the moment, there's no way to reference your newly created button in code. You must first create an outlet to the button. **Control-drag** the button in Interface Builder to an empty space at the top of the view controller's definition in the editor area. In the popover, verify that **Outlet** is selected under Connection, and enter "lightButton" in the Name field.

The popover menu should look like this:

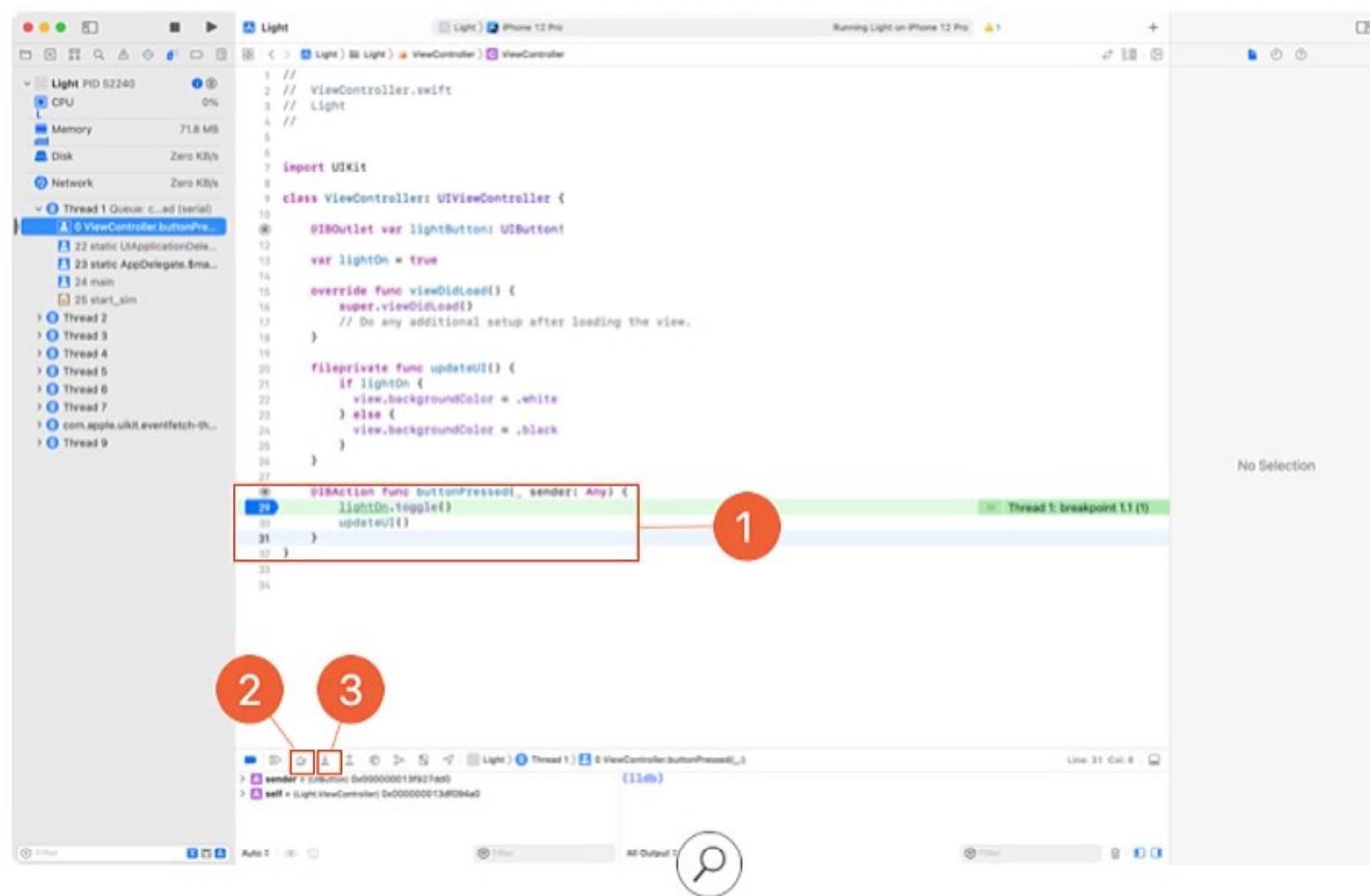


Click Connect.

As you learned in an earlier lesson, you’re able to see these details of the outlet, from left to right:

- Circle — The filled circle indicates that the outlet is connected. The circle would be empty if the property isn’t connected to anything.
- @IBOutlet — This keyword signals to Xcode that the property on this line is an outlet.
- var lightButton — This declares a property called “lightButton.”
- UIButton! — The type of the property is UIButton!. The exclamation point warns you that the program will crash if you try to access this property and the outlet isn’t connected. You’ll learn more about the exclamation point when you learn about optionals in a later unit.

Having created the outlet, you can now make changes to the button programmatically. Start by finding the right place for updating the button’s title and the right action for it to perform.



Add a breakpoint to the first line of the `buttonPressed(_ :)` method, then build and run your app. As you might expect, when you press the button to change the color, the corresponding action, `buttonPressed(_ :)`, will be executed. But since you added a breakpoint, the code will pause before executing the line with the breakpoint. ①

Click the “Step over” button ② to run the first line. Now your app is paused just before running the `updateUI()` method. Click the “Step into” button ③. Now the app is paused just before executing the first line of `updateUI`.

```

1 // ViewController.swift
2 // Light
3 // Light
4 //
5
6 import UIKit
7
8 class ViewController: UIViewController {
9
10    @IBOutlet var lightButton: UIButton!
11
12    var lightOn = true
13
14    override func viewDidLoad() {
15        super.viewDidLoad()
16        // Do any additional setup after loading the view.
17    }
18
19    fileprivate func updateUI() {
20        if lightOn {
21            view.backgroundColor = .white
22        } else {
23            view.backgroundColor = .black
24        }
25    }
26
27    @IBAction func buttonPressed(_ sender: Any) {
28        lightOn.toggle()
29        updateUI()
30    }
31 }
32
33
34

```

Take a moment to try and read the body of the `updateUI()` method. ① Since Swift is a very readable programming language, you can probably interpret the lines to say: “If the light is on, the view’s background color should be white, or else the view’s background color should be black.”

Now consider how the button’s title should be decided. If the light is on, the button’s title should read “Off,” or else the button’s title should read “On.” It makes sense that the button’s title should be updated in a similar way to the view’s background color.

You’ve now found a reasonable place to update the text—in the `updateUI()` method. But what’s the actual function for doing so? As you learned earlier, Xcode documentation is always available to support you. So whenever you’re unsure of something, it’s a good habit to look to the documentation for answers.

The screenshot shows the Xcode documentation viewer. On the left is a sidebar with a tree view of Swift frameworks, with 'UIKit' selected. The main content area has a header 'Framework' and 'UIKit'. Below the header is a brief description: 'Construct and manage a graphical, event-driven user interface for your iOS or tvOS app.' To the right are sections for 'Language' (Swift, Objective-C), 'Availability' (iOS 2.0+, Mac Catalyst 13.0+, tvOS 9.0+, watchOS 2.0+), and 'On This Page' (Overview, Topics). A central box contains the 'Overview' section, which describes the UIKit framework's purpose and features. Below this is a callout box labeled 'Important' with the note: 'Use UIKit classes only from your app's main thread or main dispatch queue, unless otherwise indicated. This restriction particularly applies to classes derived from UIResponder or that involve manipulating your app's user interface in any way.' At the bottom of the main content area is a 'Topics' section with a 'Essentials' heading and two items: 'About App Development with UIKit' and 'Protecting the User's Privacy'.

To learn how to set the title of `lightButton`, start by reading the documentation on its type, `UIButton`. You can access the documentation directly using **Window > Developer Documentation** from the Xcode menu. This opens a separate window with the documentation viewer.

Start entering `UIButton` in the documentation search field. Autocompletion quickly suggests options. As soon as you see `UIButton` in the menu, go ahead and select it. ①

The screenshot shows the Xcode documentation interface. On the left, there's a sidebar with a tree view of frameworks and topics. The 'UIKit' framework is selected. In the main content area, the 'Overview' page for UIKit is displayed. At the top right of this page, there's a search bar with the text 'UIButton'. Below the search bar, a list of suggested results is shown, with 'UIButton' being the first item and highlighted with a purple background. To the right of the search results, there are sections for 'Language' (Swift, Objective-C), 'Availability' (iOS 2.0+, Mac Catalyst 13.0+, tvOS 9.0+, watchOS 2.0+), and 'On This Page' (Overview, Topics). A callout box with the number '1' points to the 'UIButton' entry in the search results.

Scroll down to the Symbols section to find a list called “Configuring the Button.” There you’ll find a function to set the title: `func setTitle(String?, for: UIControl.State)`.

Click the function name to see its declaration and a list of its parameters. ¹ The first parameter is the desired text, and the second parameter is a `UIControl.State`. `UIControl.State` represents the different potential states of a button, for example: if it is sitting idle, if the user has begun tapping it, or if the button is disabled. Click `UIControl.State` in the Declaration section.

The screenshot shows the Xcode documentation for the `setTitle(_:for:)` method of `UIButton`. The sidebar on the left is titled "Swift" and lists various Swift-related topics. The main content area shows the following:

- Instance Method**: `setTitle(_:for:)`
- Description**: Sets the title to use for the specified state.
- Declaration** (highlighted by a red box):


```
func setTitle(_ title: String?,
               for state: UIControl.State)
```
- Parameters** (highlighted by a red box):
 - title**: The title to use for the specified state.
 - state**: The state that uses the specified title. `UIControl.State` describes the possible values.
- Discussion** (highlighted by a red box):

Use this method to set the title for the button. ¹ If you specify derives its formatting from the button's associated label object. If you set both an attributed title for the button, the button prefers the use of the attributed title over this one.

At a minimum, set the value for the `normal` state. If you don't specify a title for the other states, the button uses the title associated with the `normal` state. If you don't set the value for `normal`, then the property defaults to a system value.
- Important** (highlighted by a red box):

When the user interface idiom is `UIUserInterfaceIdiom.mac` and `behavioralStyle` is `UIBehavioralStyle.mac`, your app thrombs the `UIControl.State` option if you use this method to set the title.

In this new list of symbols, the Constants section contains a `normal` constant¹, which corresponds to the state of the button when it's enabled and sitting idle on the screen. This is the correct state for setting the button's title.

The screenshot shows the Xcode documentation for the `UIButton` class. The sidebar on the left lists various Swift framework categories. The main content area shows the `UIButton` class with its `UIControl.State` enum. A red circle with the number "1" is overlaid on the first item in the `Constants` section, which is the `normal` constant. The `normal` constant is described as the default state of a control. Other constants listed include `highlighted`, `disabled`, `selected`, `focused`, `application`, and `served`. The `application` constant is noted as being available for application use. The `focused` constant is described as the focused state of a control. The `disabled` constant is described as the disabled state of a control. The `selected` constant is described as the selected state of a control. The `highlighted` constant is described as the highlighted state of a control. The `normal` constant is described as the normal or default state of a control. The `served` constant is described as a reserved state for internal framework use.

With your new knowledge of the `updateUI()` method, you can remove the breakpoints you added earlier and adjust the method to look like the following:

```
func updateUI() {  
    if lightOn {  
        view.backgroundColor = .white  
        lightButton.setTitle("Off", for: .normal)  
    } else {  
        view.backgroundColor = .black  
        lightButton.setTitle("On", for: .normal)  
    }  
}
```

Build and run your app. Clicking or tapping the button should now change both the background color and the button text. But there's a bug: Before the button is clicked, the text still reads "Button," not "On" or "Off." How can you update the code to resolve this issue?

Your project already has code that will make sure the button's text matches the on or off state of the light: `updateUI()`. You just need to call it when the view first loads—so that the button is updated when the view appears, instead of when the user first taps the button. You can run setup code in the `viewDidLoad()` function, which is already defined in your view controller subclass and is called when the view controller is ready to appear on the screen.

Call `updateUI()` within this method, as shown in the following example:

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

Build and run your app again. On launch, the text of the button should now read "On."

Part Four

Improve The User Experience

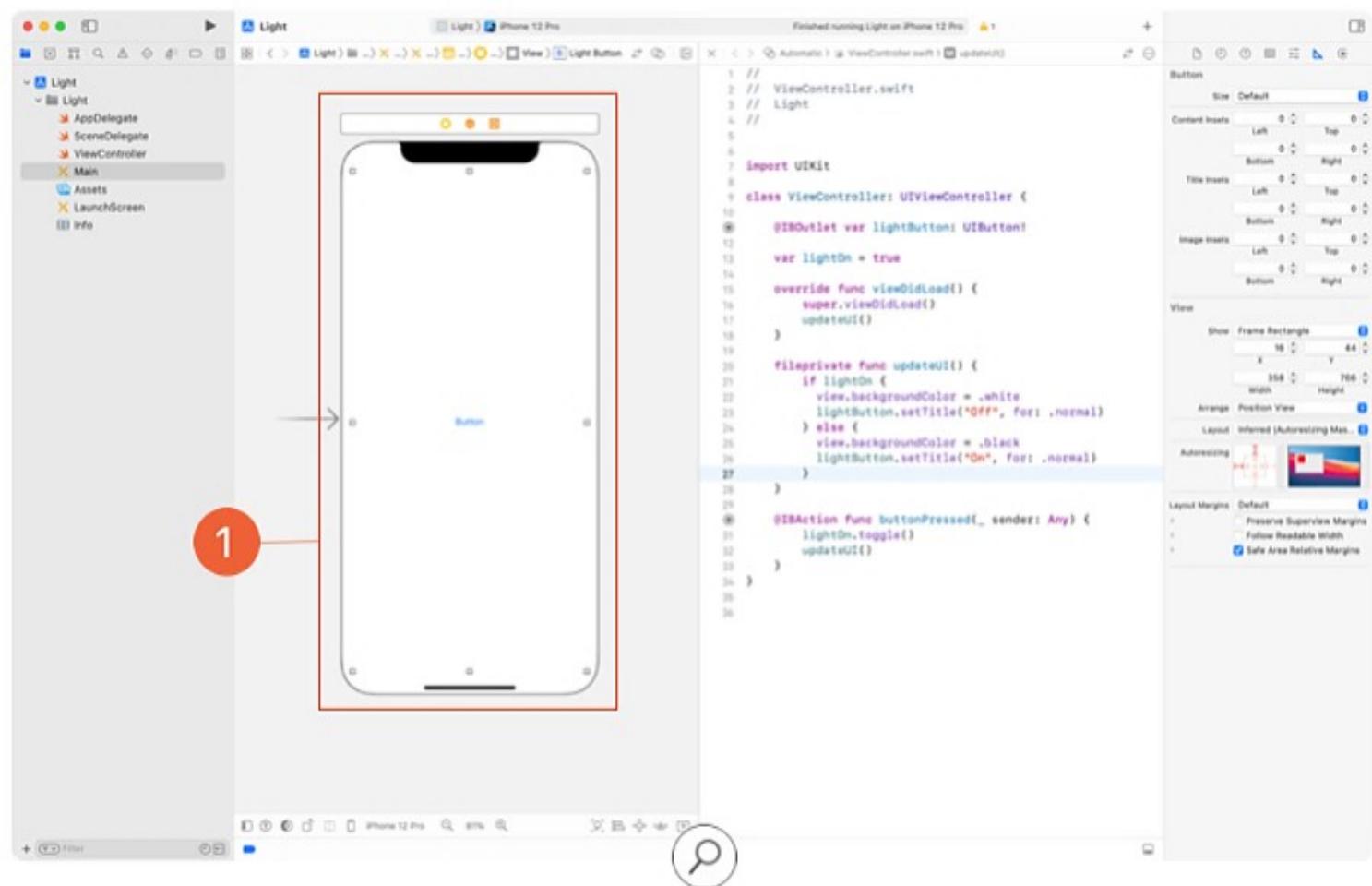
You have an app that works. That's a great start. Now take a moment to think about its design and how the user experiences the app. What could be improved?

Looking at the button, you might realize that the text feels extraneous. It's pretty clear whether the light is currently on or off—the background color indicates that. Does the user need to see the text at all?

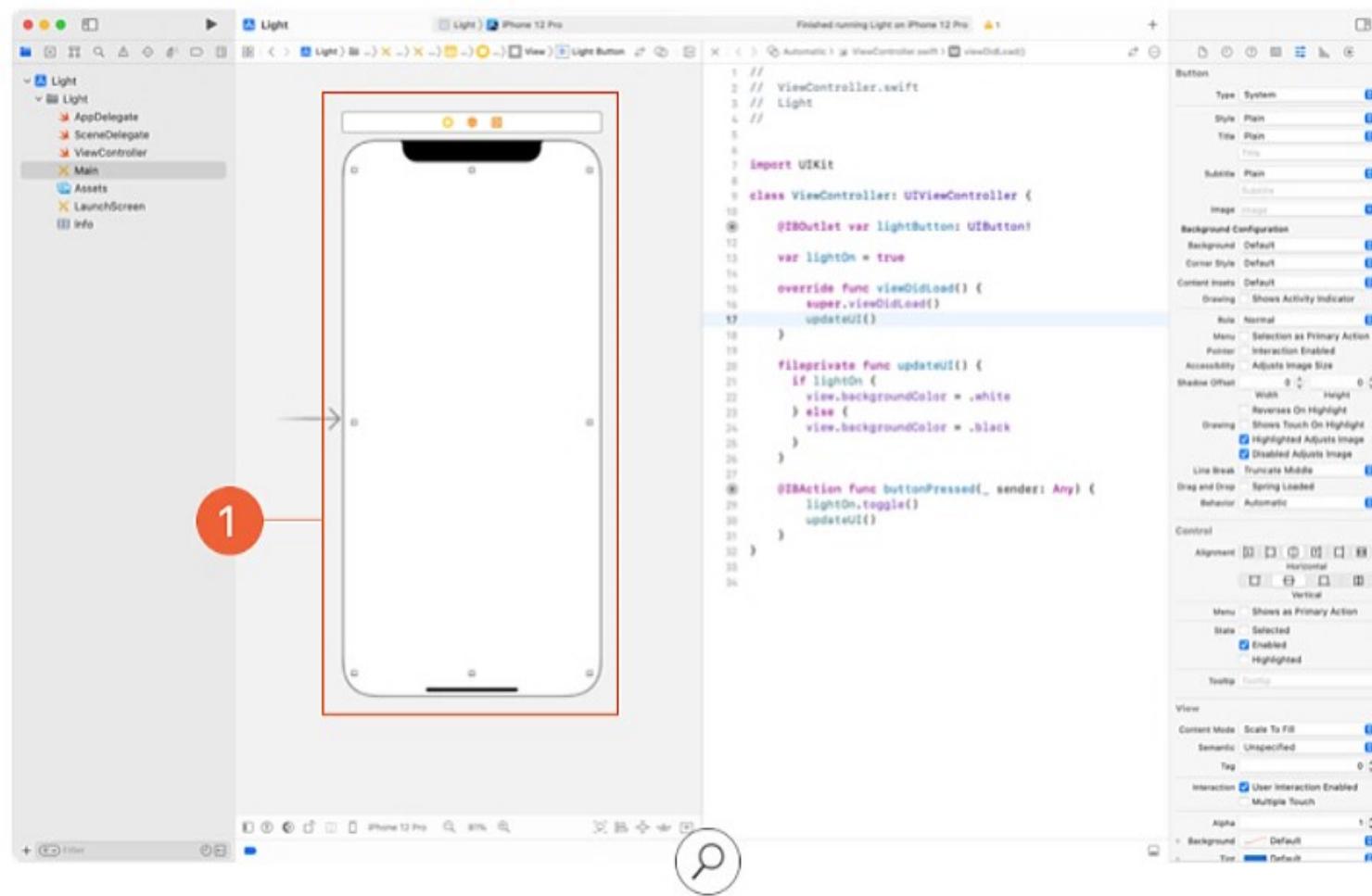
Does it matter where on the screen the button is located? And isn't it a little odd that most of the screen isn't tappable?

It may be better to remove the text from the button and to make the button fill the entire screen. That way, the user can tap anywhere on the screen to turn the light on and off. An important part of building any app is considering the little details and trying to build an intuitive, simple, powerful user interface. In this case, it makes sense for the user to be able to tap anywhere on the screen to toggle the background color.

To begin, you'll need to resize the button to fill the screen. Select the button in Interface Builder, then drag its top-left and bottom-right corners, extending the button to cover the entire white view. ①



Remove the title text for the button. The tappable area of the button will still perform as expected, but it will now appear as though only the white view exists on the canvas.



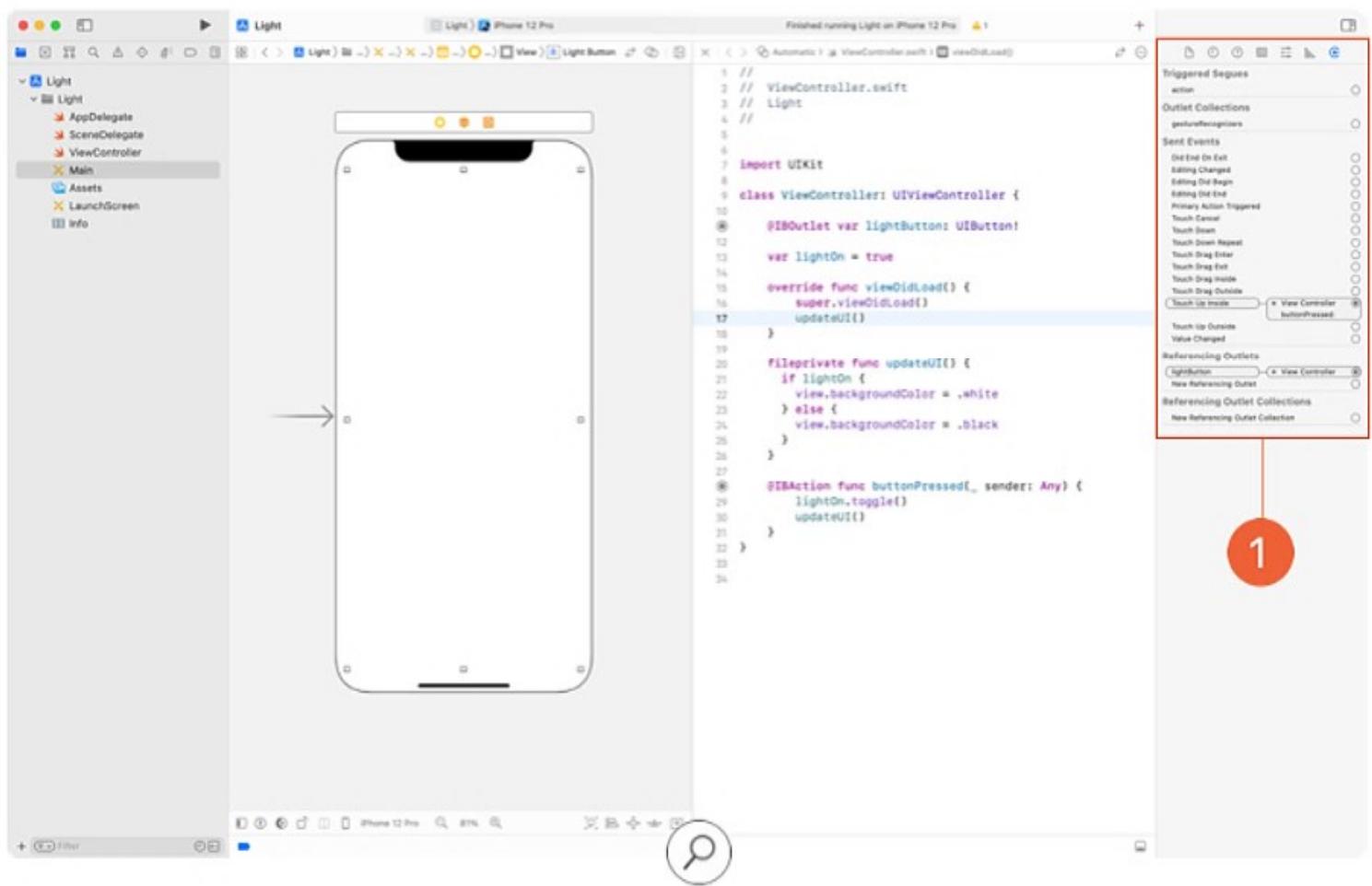
With the text gone, you can remove the lines of code in `updateUI()` that make changes to the button's title.

```
func updateUI() {  
    if lightOn {  
        view.backgroundColor = .white  
    } else {  
        view.backgroundColor = .black  
    }  
}
```

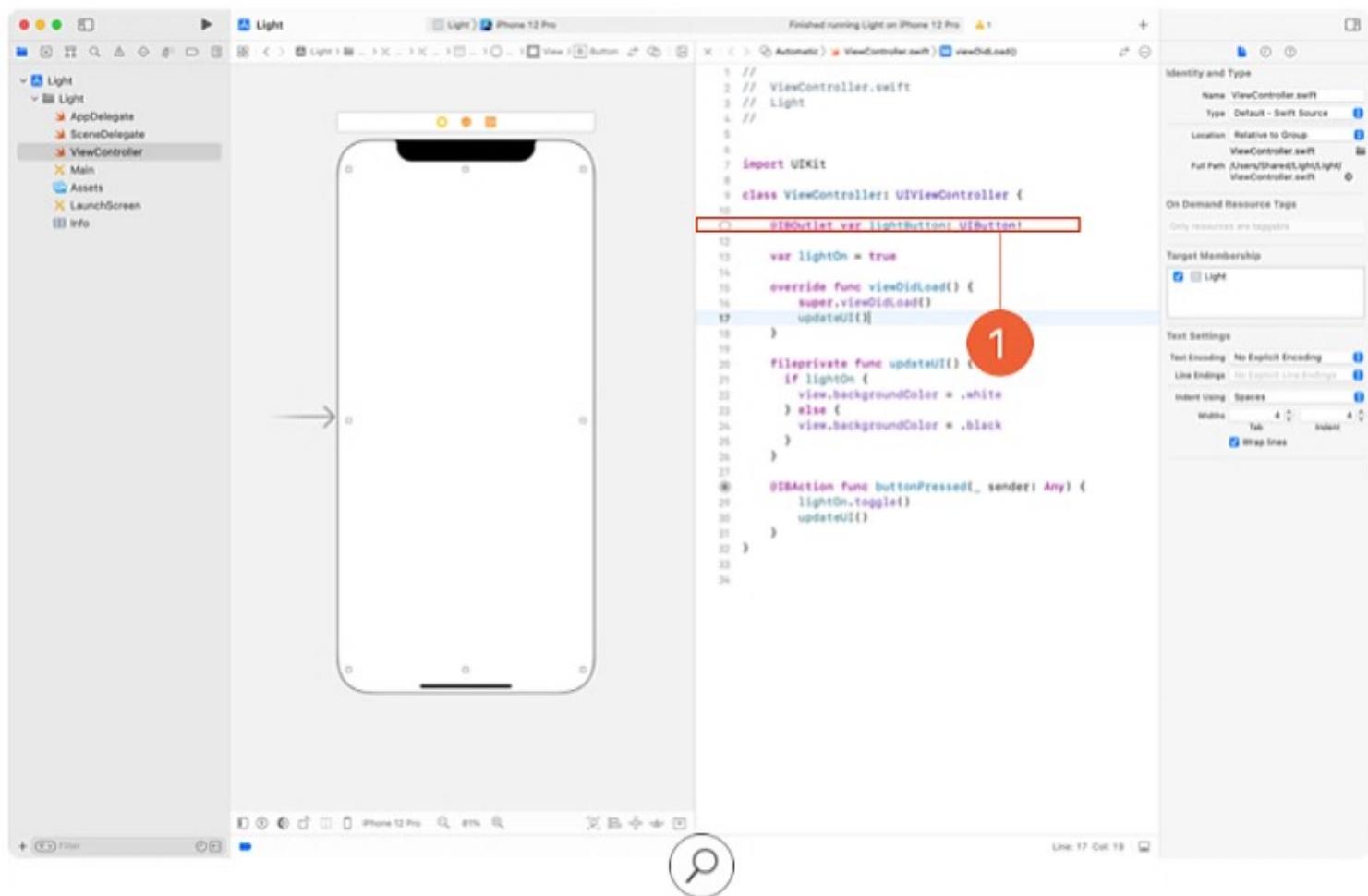
In this updated design, the button's properties don't need to change while someone is using the app. That means the outlet—which you created at the beginning of the project—is no longer needed. As a developer, you want to keep your code as clean as possible, so it's a good idea to clear out the `@IBOutlet`.

First, sever the connection between the button and its outlet. To do so, select the button in Interface Builder, then show the Connections inspector (1). Here you can see all the events and outlets connected to a particular object. In this case, you can see that the button is tied to the `buttonPressed:` action and to the `lightButton` outlet.

Click the X next to the outlet to remove it. (1)



Since there are no longer any objects connected to `lightButton`, the little circle next to `@IBOutlet` is no longer filled in. ①



It's now safe to delete the declaration of `lightButton`. Build and run your app one last time to test your new design improvements. You should now be able to tap or click anywhere on the screen to change its background color.

Because both lines in the if-else statement do the same thing (set the `backgroundColor`), the `updateUI()` method might look cleaner if you use a ternary operator instead of an if-else statement.

```

func updateUI() {
    view.backgroundColor = lightOn ? .white : .black
}

```

Build and run your application once more to verify that the ternary operator works as intended.

Wrap-Up

You've successfully created an app that changes the screen from black to white, and back, at the tap of a button—something like a flashlight. You've simplified the design of the app as well, making conscious decisions to improve the user experience by removing the button title and filling the screen with a tappable area. Even though you have limited knowledge of Swift, you were able to step through lines of code and refer to the documentation to complete this project.

Did you find it difficult to navigate Xcode or use Interface Builder? If so, revisit the previous lessons that discussed them in more detail. You'll find that the more you build, the more familiar it becomes.

As you move onto the next unit, keep the developer documentation handy. Developers are only as good as the tools and resources they use.

Lesson 1.10

Finish Defining Your App

Now that you have created the Light app and thought about the user experience, it's time to return to your own app idea! At the beginning of this unit, you defined the problem and audience for your app. Throughout the unit you added notes about how the things you were learning could help you bring your app to life. Now it's time to make a concrete plan for what your app will actually do.

What You'll Learn

- How to create a plan for building your app
-

Related Resources

- [WWDC 2018 The Qualities of Great Design](#)
- [WWDC 2021 The Process of Inclusive Design](#)

Guide

Plan

The planning stage is where you map out some of your app's more granular details and figure out how it can achieve its goal. What will a user do with your app? What sets it apart from other apps trying to solve this same problem? Now is the time to focus in on the features and functionality your app will have.

Work through the Plan section of the App Design Workbook. By the end of these activities, you'll have a concise, well-defined plan that you can begin building into a prototype. You'll build your plan by identifying key differentiators, setting goals, and then narrowing down your feature set to exactly those you'll need to test whether your app will impact real users.

Summary

Excellent work! Now that you've finished this introductory unit, you've learned about Xcode, Interface Builder, and the environment you'll use to build apps, as well as some basic Swift concepts. You've also built your first app!

In the next unit, you'll learn about views and controls in [UIKit](#), which provides the crucial infrastructure needed to build iOS apps, and more Swift concepts that will lay the foundation for building more complex apps throughout the course.

Unit 2

Introduction To UIKit

In the last unit, you got started with the basics of the Swift programming language and the basics of the Xcode development environment. You learned about constants, variables, types, operators, and control flow—and you had a chance to experiment with some Xcode features.

This unit will take you many steps further. You'll learn about structures, collections, loops, and different ways to work with the information that makes up an app. You'll also build a Keynote prototype of your own app idea and think about how the skills you're learning can be used to eventually build a functioning app. At the end of the unit, you'll build an ambitious project. Don't worry if it feels hard—because it is! Every coder in the world went through the same challenges. Keep at it, and you'll start to enjoy solving problems with code.



Swift Lessons

- Strings
- Functions
- Structures
- Classes and Inheritance
- Collections
- Loops



SDK Lessons

- Introduction to UIKit
- Displaying Data
- Controls in Action
- Auto layout and Stack Views

What You'll Design

The App Design Workbook will guide you through designing a working prototype of your app in Keynote.

What You'll Build

Apple Pie is a simple word-guessing game where the user must guess a word, letter by letter, before all the apples fall off the apple tree. If there are apples remaining, the user wins—and can eat delicious Apple Pie.

Lesson 2.1

Start Your App Prototype

In the last unit, you learned about how you can control how your app functions for a user with control flow and Interface Builder. You might even be able to program a small bit of your app right now! But before diving into coding, most app developers save time and energy by first creating a working prototype to iterate on their ideas and get feedback before finalizing the structure of their app. Developers build prototypes in all sorts of ways, from simple notebook sketches to partially completed apps in Xcode.

In this unit, you'll build a prototype in Keynote—a happy medium. It'll be easier to create than using Xcode, but have more fidelity to the look and feel of an app than paper. In this lesson, you'll start by mapping screens to form an app architecture.

What You'll Learn

- How to create a Keynote prototype showing your app's architecture
-

Related Resources

- [WWDC 2017 60 Second Prototyping](#)
- [WWDC 2018 I Have This Idea for an App...](#)
- [WWDC 2017 Essential Design Principles](#)

Guide Prototyping

A prototype is a fake version of your app that you can show other people to get feedback. It should give users a sense of how your app works, even though it may not do much. If you start from your app's goals and features, creating a prototype can be fun, and it can delight your users even if you don't write a single line of code.

Map

In the last unit, you outlined a concrete plan for what features and functionality your app should have, including choosing an MVP feature that can help a user solve the primary problem. Starting with this feature set, you are ready to create a prototype to show potential users. The first step in this process is to turn your list of features into workflows showing how a user will navigate within the app. This is where your app starts to take shape.

Complete the Map section of the App Design Workbook. In these activities, you'll create an outline of the information and functions on each screen—and how they relate to each other. You'll derive your app's architecture from its key functions, making decisions based on how you expect users to work with it. By the end of this section you'll have a Keynote file with linked screen outlines, which is the beginning of your prototype.

Lesson 2.2

Strings

 Text is everywhere. You'll find it on billboards, on social media, on bills, and on cereal boxes. So it should come as no surprise that text is just as vital in programming as it is everywhere else in our world.

In this lesson, you'll learn how to create and store text using the `String` type. You'll learn a variety of `String` methods that allow you to compare two strings, access particular characters within a string, and insert and remove values.

What You'll Learn

- How to declare a `String`
 - How to compare two strings
 - How to access particular characters within a string
-

Vocabulary

- [case sensitivity](#)
 - [concatenation](#)
 - [equality](#)
 - [escape character](#)
 - [index](#)
 - [literal](#)
 - [range](#)
 - [string interpolation](#)
 - [substring](#)
 - [Unicode](#)
-

Related Resources

- [Swift Programming Language Guide: Strings and Characters](#)

In a previous lesson, you printed the iconic “Hello, world!” string. You created “Hello, world!” by stringing together a collection of characters. Swift’s strings are represented by the `String` type, and the most common way to define one is by using a string literal.

A *string literal* is a raw representation of a `String` value. You write a string literal by surrounding a set of characters with double quotation marks “”.

String literals are commonly used to set an initial value for a constant or variable:

```
let greeting = "Hello"  
var otherGreeting = "Salutations"
```

If you assign a string to a constant (using `let`), the string is *immutable* and can’t be modified. If the string is assigned to a variable (using `var`), the string is *mutable* and can change.

If your string literal needs to be multiple lines, simply surround your set of characters with three double quotation marks “””. In its multiline form, the string literal includes all of the lines between its opening and closing quotes. The string begins on the first line after the opening quotes and ends on the line before the closing quotes.

```
let joke = """  
Q: Why did the chicken cross the road?  
A: To get to the other side!  
"""  
  
print(joke)
```

Console Output:

Q: Why did the chicken cross the road?
A: To get to the other side!

Any space preceding each line in your multiline string literal will be ignored if it aligns with the closing """ quotation marks. Multiline string literals may contain double quotes ("").

```
let greeting = """
    It is traditional in programming to print "Hello, world!"
"""

```

To include double quotes in single-line string literals, you'll need to use the backslash (\), known in Swift as the *escape character*. That's because you're “escaping” the normal interpretation of characters in the string.

```
let greeting = "It is traditional in programming to print
\"Hello, world!\""

```

You can use the escape character with other letters and symbols to produce specific results:

- Double quote: \"
- Single quote: \'
- Backslash: \\
- Tab: \t
- Newline (go to the next line—like pressing Return): \n

You'll often find that you want to start with an empty string, then add to it over time. This means you'll need to make it a variable. Use the following syntax to initialize the string without any text:

```
var myString = """

```

If you need to check if a Swift `String` is empty, you can use the Boolean property, `isEmpty`:

```
var myString = ""

if myString.isEmpty {
    print("The string is empty")
}
```

As you might expect, individual characters are of type `Character`. But since strings are much more common in programming than individual characters, Swift will always infer the type of a collection of characters — or even a single character — as `String`, unless you specify otherwise with a type annotation:

```
let a = "a" // 'a' is a String  
let b: Character = "b" // 'b' is a Character
```

Concatenation And Interpolation

Sometimes you need to combine strings. The `+` operator isn't just for numbers; it can add strings together too. You can use `+` to create a new `String` value from multiple `String` values. This is called *concatenation*.

```
let string1 = "Hello"  
let string2 = ", world!"  
let myString = string1 + string2 // "Hello, world!"
```

If the existing `String` is a variable, you can use the `+=` operator to add to it or modify it.

```
var myString = "Hello"  
myString = myString + ", world!" // "Hello, world!"  
myString += " Hello!" // "Hello, world! Hello!"
```

As strings grow in complexity, the use of the `+` operator can make your code tricky to handle. In the code above, for example, you might forget to add a space before "Hello!"

Swift provides a syntax, known as *string interpolation*, that makes the inclusion of constants, variables, literals, and expressions easier. String interpolation allows you to easily combine many values into a single `String` constant or variable.

You can insert the raw value of a constant or variable into a `String` by preceding the name with a backslash `\` and wrapping the name in parentheses `()`. In the example below, the printed `String` will contain the raw values of the `name` and `age` constants.

```
let name = "Rick"  
let age = 30  
print("\((name) is \(age) years old") // Rick is 30 years old
```

You can place entire expressions within the parentheses. These expressions will always be evaluated first, before being printed or stored.

```
let a = 4  
let b = 5  
print("If a is \(a) and b is \(b), then a + b equals \(a+b)")
```

Console Output:

If a is 4 and b is 5, then a + b equals 9

Using string interpolation, the raw values of the `a` and `b` constants are inserted into the printed `String` value.

It was hinted at above, but since the result of string interpolation is a `String`, you can also use it anywhere you'd use a `String` or string literal:

```
let listName = "Shopping"  
var items = 14  
myLabel.text = "There are \(items) items on your \(listName) list"  
// The label displays "There are 14 items on your Shopping list"  
  
func setLabel(_ label: UILabel, to text: String) {  
    label.text = text  
}  
  
setLabel(myLabel, to: "There are \(items) items on your  
\(listName) list")
```

String Equality And Comparison

Developers often need to compare `String` values to see if they're equal to each other. Just as you do with numbers, you can check for equality between two strings using the `==` operator. As you might expect, `==` checks for identical characters in the same order. Since uppercase characters aren't identical to their lowercase counterparts, the strings have the same value if the case of each character also matches. This is known as *case sensitivity*.

```
let month = "January"
let otherMonth = "January"
let lowercaseMonth = "january"
if month == otherMonth {
    print("They are the same")
}

if month != lowercaseMonth {
    print("They are not the same.")
}
```

Console Output:

They are the same.
They are not the same.

But maybe you want to ignore the capitalization of a string when checking for string equality. You can use the `lowercased()` method to normalize the two, comparing an all-lowercase version of the string with an all-lowercase version of the calling string.

```
let name = "Johnny Appleseed"  
if name.lowercased() == "joHnnY aPPleseeD".lowercased() {  
    print("The two names are equal.")  
}
```

Console Output:

The two names are equal.

You could have also used `lowercased()`'s counterpart, `uppercased()`, which creates an all-uppercase version of the string.

If you want to match the beginning or the end of the string, you can use the `hasPrefix(_:_)` or the `hasSuffix(_:_)` method. Just like `==`, these matches are case-sensitive.

```
let greeting = "Hello, world!"  
print(greeting.hasPrefix("Hello"))  
print(greeting.hasSuffix("world!"))  
print(greeting.hasSuffix("World!"))
```

Console Output:

true
true
false

Maybe you want to check if one string is somewhere within another string. You can use the `contains(_ :)` method to return a Boolean value that indicates whether or not the substring was found.

```
let greeting = "Hi Rick, my name is Amy."  
if greeting.contains("my name is") {  
    print("Making an introduction")  
}
```

Since a string is a collection of characters, its length is equal to the total number of characters. The size of any collection can be determined using its `count` property. You can use this property to compare strings or to evaluate whether strings meet a certain requirement.

```
let name = "Ryan Mears"  
let count = name.count // 10  
  
let newPassword = "1234"  
  
if newPassword.count < 8 {  
    print("This password is too short. Passwords should have at  
    least eight characters.")  
}
```

Console Output:

This password is too short. Passwords should have at least eight characters.

In the last unit, you learned that you can use `switch` statements to perform specific blocks of code based on a particular case. You can also use the `switch` statement to pattern-match multiple values of strings or characters and to respond accordingly.

```
let someCharacter: Character = "e"
switch someCharacter {
    case "a", "e", "i", "o", "u":
        print("\(someCharacter) is a vowel.")
    default:
        print("\(someCharacter) is not a vowel.")
}
```

Console Output:

e is a vowel.

More Advanced String Topics

Similar to the `lowercased()`, `hasPrefix(_:_)`, `hasSuffix(_:_)`, and `contains(_:_)` methods mentioned previously, strings come with a variety of properties and methods that can be useful for tracking locations of characters within strings, creating new strings from “substrings,” inserting characters or strings into existing strings, and much more. Much of this functionality is shared among all Swift collections. For now you don’t need to worry about these more advanced string topics, but keep in mind that you can always turn to documentation to learn more should you wish to do more advanced string manipulation with Swift. For reference, here are a few useful `String` properties and methods that you could look up:

- `startIndex`
- `endIndex`
- `index(before:)`
- `index(after:)`
- `index(_:_offsetBy:)`
- `insert(_:_at:)`
- `insert(contentsOf:at:)`
- `remove(at:)`
- `removeSubrange(_:_)`
- `replaceSubrange(_:_with:)`

For a full list of methods and more information you can review the [String documentation](#).

Unicode

Every Swift `String` adheres to an international computing standard called Unicode. Unicode compliance allows Swift to go beyond the short list of letters and symbols in the English language. Instead, Unicode encompasses over 128,000 different characters used across multiple languages. This includes accents on characters (é), emoji (🐮), symbols (∞), Kanji (七), and other specialized characters. In addition, Unicode supports text that reads right to left, as well as left to right.

Swift ensures that you can work with Unicode characters conveniently, so that "e," "é," and "🐮" are treated as single characters with a length of 1.

```
let cow = "🐮"  
let credentials = "résumé"  
print("∞".count) // 1
```

In reality, a character is a Unicode “extended grapheme cluster” which is a fancy way of saying that some characters are actually comprised of multiple (often invisible) characters although they appear to the user as a single character. For example, “ü” comprises two characters and “👨” comprises *four* characters! Swift makes working with these much easier.

Lab

Open and complete the exercises in [Lab—Strings.playground](#).

Connect To Design

In your App Design Workbook, reflect on where text might appear in your app. What kinds of messages might be displayed to a user? Will there be instructions or other text needed in the interface? Make comments in the Map section or add a blank slide at the end of the document.

In the workbook's Go Green app example, a user might log the types of trash and recycling they have throughout the day. Each of these types of materials, like paper or coffee grounds, would be strings stored by the app. The title a user enters for a recycled item is also a string.

Review Questions

Question 1 of 5

When you declare a `String`, how do you set the initial value?

- A. With Unicode
- B. With a string literal
- C. With the operator `+=`
- D. With an index

[Check Answer](#)



Lesson 2.3

Functions

In previous lessons, you learned to use existing functions, such as `print`, to perform a task. In those cases, you didn't have to worry about the details of *how* the function was implemented; it just worked. But if you want to write maintainable, reusable code, you'll need to be able to create your own functions.

In this lesson, you'll learn how to declare functions with different parameters and return types, while gaining a better understanding of the importance of abstraction.

What You'll Learn

- How to write your own functions, with or without a return type
 - How to specify input parameters for your functions
 - How to return multiple values when necessary
 - How to customize the way your functions are called
 - How to provide default parameter values to a function
-

Vocabulary

- argument label
 - parameter
 - return type
 - return value
-

Related Resources

- [Swift Programming Language Guide: Functions](#)

When someone gives you the task “get dressed,” it seems like such a simple instruction. But the details of what you’ll wear, how you’ll put the clothes on, and where you’ll get the clothes from are all wrapped up within the “get dressed” phrase. The idea of taking something that is complex and defining a simpler way to refer to it is an abstraction, and a function is one of the fundamental ways to create an abstraction in code.

A function is made up of a name, a set of inputs, and a set of outputs. Both inputs and outputs are optional. You call or invoke a function to cause your program to *do* something such as print text to the console.

Imagine you want to send a text message to your friend. “Send text message” is what you want to do. The parameters are your friend’s contact information and the message you want to send them. The return value is whether they’ve received the message.

In Swift, a function can take zero, one, or many parameters, and likewise return zero, one, or many values. When a function *does* return values, you can choose to ignore them.

Similar to the declaration of a constant with `let` or a variable with `var`, the `func` keyword tells the Swift compiler that you’re declaring a function. Immediately following `func`, you add the name of the function followed by parentheses `()`, which may or may not include a list of parameters within them. If the function has a return value, you’ll write an arrow `(->)`, followed by the type of data the function will return, such as `Int`, `String`, `Person`, and so on.

Defining A Function

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

Here's an example of a function that will display the first ten digits of pi (π). Since printing pi requires no additional input, there are no parameters. And since printing pi doesn't need to return anything relevant to the function caller, no return value is specified.

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

Once a function has been properly declared, you can call it, or execute it, from anywhere just by writing its name.

```
displayPi()
```

Console Output:

3.1415926535

Parameters

To specify a function with a parameter, insert a name for the value, a colon (:), and the value's type—all inside the parentheses. For example, say you wanted to write a function called `triple` that takes in an `Int`, triples the value, and then prints it.

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

In the case above, the parameter `value` is a constant you can use within the function. When you call a function, you pass values for its parameters as *arguments*. In the code below, the argument for `value` in the call to the function `triple(value:)` is 10.

```
triple(value: 10)
```

Console Output:

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

To give a function multiple parameters, separate each parameter with a comma (,). Here's an example of a function that takes in two `Int` parameters, multiplies them together, and then prints the result:

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

The function is then called with one argument per parameter:

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

Console Output:

The result is 50.

Argument Labels

So far, the labels for the arguments passed to a function are the same as the names for the parameters it uses internally. In the next example, `firstName` is used both within the function and when the function is called:

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

But imagine that you want your function to read a little more cleanly:

```
sayHello(to: "Miles", and: "Riley")
```

The argument labels `to` and `and` make the function read very clearly: "Say hello to Miles and Riley." However, check out the implementation of this function:

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

Within the body of the function, `to` and `and` are poor names for parameters. To make the name of the parameter within the function different from the label used to call the function, specify a separate *argument label* before the parameter name. In the code below, `to` is the argument label for the parameter `person`, while `and` is the argument label for the parameter `anotherPerson`:

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

If the function is clearer without an argument label, you can go ahead and omit it. For example, take the `print` function:

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

The name of the function already makes it very clear what should be passed in as a parameter. It would feel extraneous if you had to call the function in the following way:

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

To omit the argument label, use `_`. In the next example, the label for the `firstNumber` parameter is omitted, while the `to` label adds meaning and makes the function call more readable:

```
func add(_ firstNumber: Int, to secondNumber: Int) -> Int {  
    return firstNumber + secondNumber  
}  
  
let total = add(14, to: 6)
```

Default Parameter Values

Parameters allow a function to remain flexible, but often there's a value you want to use most of the time. For example, you might drink water with nearly all of your meals, but every now and then you have a glass of orange juice instead.

As part of the function definition, you can provide a default value for any parameter. This way, you can call the function with or without the parameter. If the parameter is unspecified, the function simply uses the default value.

```
func display(teamName: String, score: Int = 0) {  
    print("\(teamName): \(score)")  
}  
  
display(teamName: "Wombats", score: 100) // "Wombats: 100"  
display(teamName: "Wombats") // "Wombats: 0"
```

A certain level of thought needs to go into defining a function with default parameter values and argument labels. The compiler can do only so much when inferring what you're calling. If you mix default values and parameters without argument labels, you need to validate that every variation of the function call can be inferred.

For example, the compiler can't infer which variant of your function to call when defined like the following:

```
func displayTeam(_ teamName: String, _ teamCaptain: String =  
    "TBA", _ hometown: String, score: Int = 0) {  
    // ...  
}  
  
displayTeam("Dodgers", "LA") // ERROR: Missing argument for  
parameter #3 in call
```

You may find it obvious that you want the function to use the default value of “TBA” for `teamCaptain` and assign “Dodgers” to `teamName` and “LA” to `hometown`. But the compiler recognizes only three `String` parameters without argument labels, and only two values to assign. It can’t assume that you intended to use the default value for `teamCaptain`.

It’s best to leave arguments with default values at the end of the function signature and always provide an argument label. In the above example, the compiler would be satisfied if `teamCaptain` had an argument label.

```
func displayTeam(_ teamName: String, _ hometown: String,  
teamCaptain: String = "TBA", score: Int = 0)
```

Return Values

It’s unlikely that you’ll always want to print “The result is” before the result from `multiply`. Instead, it might make more sense if the function simply returned the new value. To do so, you’ll need to adjust the function declaration to have a return value and to specify the value’s type. You’ve learned that multiplying two `Int` types will always result in an `Int`, so that’s the return type you’ll use.

```
func multiply(firstNumber: Int, secondNumber: Int) -> Int
```

Within the function’s body, you’ll use the `return` keyword to specify what the return function will return. In this example, you’ll want to return `result`:

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

Rather than multiplying the values, storing the new value in `result`, and immediately returning it, the function could also be written without the constant:

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

Starting with Swift 5.1, you can even omit the `return` keyword for functions that have a single line implementation. Because `multiply(firstNumber:secondNumber:)` has a return type and only one line of code in its body, the result of that expression is used as its return value.

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

To call this function and use the return value, you can assign the return value to a constant:

```
let myResult = multiply(firstNumber: 10, secondNumber: 5)  
//myResult = 50  
print("10 * 5 is \(myResult)")
```

If you don't need to use `myResult` ever again in the future, you could use the function inside the `print` statement and skip assigning the value to a constant:

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

Lab

Open and complete the exercises in [Lab—Functions.playground](#).

Connect To Design

Open up your App Design Workbook and review the Map section, where you defined functions for your app. How might these correspond to functions in your app code? Reflect on the kinds of information your app might need to use, and think about how you might call a function multiple times with different information. Add comments to the Map section or in a new blank slide at the end of the document. Can you outline or write pseudocode for the functions?

In the workbook's Go Green app example, each time a user logs an item they categorize it as trash or recycling, and the app updates a graph. This would be a great place to use a function, since the task of logging an item is something that will be done over and over.

Review Questions

Question 1 of 3

Which of the following statements are true about the function below?

```
func greet(name: String) {  
    print("Hello, \(name)")  
}
```

- A. It can be called using `greet("Jason")`.
- B. The first parameter requires an argument label.
- C. The function returns a `String`.
- D. The function has no return value.

[Check Answer](#)



Lesson 2.4

Structures

Swift comes with many useful types for representing data like numbers, text, collections, and true or false values. But as you get into building apps, you'll find you want to create your own data types, with properties and functions of your own design.

You create a custom data type by declaring a structure. A structure combines one or more variables into a single type. You can define functionality by adding type and instance methods to a structure.

In this lesson, you'll learn how to create structures and you'll discover how structures form the building blocks of your code.

What You'll Learn

- How to create a custom structure
 - How to define properties on a structure
 - How to add methods, or functions, to a structure
-

Vocabulary

- [computed property](#)
 - [function](#)
 - [initializer](#)
 - [initialization](#)
 - [instance method](#)
 - [memberwise initializer](#)
 - [method](#)
 - [property](#)
 - [self](#)
 - [structure](#)
 - [type](#)
-

Related Resources

- [Swift Programming Language Guide: Classes and Structures](#)

You may not have realized it, but you've been working with structures since the beginning of this course. Swift comes with predefined structures for representing common types of data. You can define your own structures when you want a type that fits the specific needs of your program or app.

In its simplest form, a structure is a named group of one or more properties that make up a type. Properties represent the information about an instance of the structure.

You define a structure using the `struct` keyword along with a unique name. You can then define properties as part of the `struct` by listing the constant or variable declarations with the appropriate type annotation. The convention is to capitalize the names of types and to use camel case for the names of properties.

Consider the simple declaration of a `Person` structure with a `name` property:

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

The above declaration simply defines the properties of a `Person` type. It doesn't have a value in itself. For that, you have to create an instance of the `Person` type. Then you can access the data stored in its properties, such as the person's `name`, using dot syntax.

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

Console Output:

Jasmine

As the name in the example implies, more than one `Person` might get created throughout the lifetime of your program. The next instance you create could be called `secondPerson`, or you could create a collection called `people` that holds multiple `Person` objects. You'll learn about collections in an upcoming lesson.

You can add functionality to a structure by adding a method. A method is a function that's assigned to a specific type. In this example, our `Person` instances can now `sayHello`.

```
struct Person {  
    var name: String  
    func sayHello() {  
        print("Hello, there! My name is \(name)!")  
    }  
}
```

You can now call the instance method directly using dot syntax.

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

Console Output:

Hello, there! My name is Jasmine!

In the following sections, you'll learn more about instances and instance methods.

Instances

You've just learned that a structure defines a new type. To use that type, you must create an instance of it, a process called initialization. After initialization, each instance inherits all the properties and features of the structure.

In the following example, both `myShirt` and `yourShirt` are `Shirt` objects with `size` and `color` properties. But each one is a separate shirt with distinct values for each property, and therefore a separate instance of the `Shirt` type. The `Size` and `Color` types define a group of available options, called an enumeration, which you'll learn about in a future lesson. For now, since you haven't defined `Size` and `Color`, this example won't compile if you copy it into a playground. Instead, just look at the code and try to understand conceptually what is happening.

```
struct Shirt {  
    var size: Size  
    var color: Color  
}  
  
// Defines the attributes of a shirt.  
  
let myShirt = Shirt(size: .xl, color: .blue)  
// Creates an instance of an individual shirt.  
let yourShirt = Shirt(size: .m, color: .red)  
// Creates a separate instance of an individual shirt.
```

Here's another example of a structure definition, this time with functionality added. The structure defines the attributes and functionality of a `Car` object and describes `firstCar` and `secondCar` as two instances.

```
struct Car {  
    var make: String  
    var model: String  
    var year: Int  
    var topSpeed: Int  
  
    func startEngine() {  
        print("The \(year) \(make) \(model)'s engine has started.")  
    }  
  
    func drive() {  
        print("The \(year) \(make) \(model) is moving.")  
    }  
  
    func park() {  
        print("The \(year) \(make) \(model) is parked.")  
    }  
}  
  
let firstCar = Car(make: "Honda", model: "Civic", year: 2010,  
topSpeed: 120)  
let secondCar = Car(make: "Ford", model: "Fusion", year: 2013,  
topSpeed: 125)  
  
firstCar.startEngine()  
firstCar.drive()
```

Console Output:

The 2010 Honda Civic's engine has started.
The 2010 Honda Civic is moving.

What did the cars in this code do? If you imagine `firstCar` and `secondCar` facing out of the same driveway, only `firstCar` has moved after executing this code, while `secondCar` hasn't even started the engine.

Initializers

All structures come with at least one initializer. An initializer is similar to a function that returns a new instance of the type. Many common types have a default initializer with no arguments, `init()`.

Instances created from this initializer have a default value. The default `String` is `""`, the default `Int` is `0`, and the default `Bool` is `false`:

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

But there's a shorthand syntax for initializers that's much more common. The code in the following snippet is more concise, but works the same as the example above:

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

Whenever you define a new type, you must consider how you'll create new instances. This lesson covers different approaches to initializing property values.

Default Values

During initialization of new instances, Swift requires you to set values for all instance properties.

One approach is to provide default property values in your type definition. Each instance is initialized with those values. This is useful when defining objects that have a consistent default state, such as a zero reading on an odometer.

If you provide default values for all instance properties of a structure, the Swift compiler will generate a default initializer for you.

In the previous section you saw default values for `String`, `Int`, and `Bool`. Using default property values, you can create a default state for each new instance of your custom types.

```
struct Odometer {  
    var count: Int = 0  
}  
  
let odometer = Odometer()  
print(odometer.count)
```

Console Output:

0

Note that `count` is set to a default value of `0` when declaring the property. All new instances of `Odometer` will be created with that default value.

Memberwise Initializers

When you define a new structure and don't declare your own initializers, Swift creates special initializers, called memberwise initializers, that allow you to set initial values for each property of the new instance.

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

Console Output:

27000

Memberwise initializers are the correct approach when there's not a default state for new instances of your type.

Consider a `Person` structure with a `name` property. What would you assign as the default value for `name`?

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

At first glance, you may say that the default value for `name` could be `""`. But an empty `String` is not a `name`, and you don't want to accidentally initialize a `Person` with an empty name.

To call a memberwise initializer, use the type name followed by parentheses containing parameters that match each property. In fact, you've seen memberwise initializers in the various code snippets throughout this lesson:

```
struct Person {  
    var name: String  
  
    func sayHello() {  
        print("Hello, there!")  
    }  
}  
  
let person = Person(name: "Jasmine") // Memberwise initializer  
  
struct Shirt {  
    var size: Size  
    var color: Color  
}  
  
let myShirt = Shirt(size: .xl, color: .blue) // Memberwise  
Initializer  
  
struct Car {  
    var make: String  
    var model: String  
    var year: Int  
    var topSpeed: Int  
}  
  
let firstCar = Car(make: "Honda", model: "Civic", year: 2010,  
topSpeed: 120) // Memberwise initializer
```

You might define a structure for which some properties have reasonable default values, but others don't. For example, a bank account might start with a default zero balance, but require a unique account number.

```
struct BankAccount {  
    var accountNumber: Int  
    var balance: Double = 0  
}
```

In this case, you'll get two memberwise initializers: One that provides parameters for each property, and another that provides parameters only for the properties without default values.

```
var newAccount = BankAccount(accountNumber: 123)  
var transferredAccount = BankAccount(accountNumber: 456,  
    balance: 1200)
```

Memberwise initializers are the most common way to create new instances of your custom structures. But there may be times when you want to define an initializer that completes some custom logic before assigning all of the properties. In those cases, you can define a custom initializer.

Custom Initializers

You can customize the initialization process by defining your own initializer. Custom initializers have the same requirement as default and memberwise initializers: All properties must be set to initial values before completing initialization.

Consider a `Temperature` struct with a `celsius` property. If you have access to a temperature in Celsius, you could initialize it using a memberwise initializer.

```
struct Temperature {  
    var celsius: Double  
}  
  
let temperature = Temperature(celsius: 30.0)
```

But if you have access to a temperature in Fahrenheit, you would need to convert that value to Celsius before using the memberwise initializer.

```
let fahrenheitValue = 98.6  
let celsiusValue = (fahrenheitValue - 32) / 1.8  
  
let temperature = Temperature(celsius: celsiusValue)
```

But the memberwise initializer required you to calculate the Celsius value before initializing a new `Temperature` object. Instead, you could create a custom initializer that takes a Fahrenheit value as a parameter, performs the calculation, and assigns the value to the `celsius` property.

```
struct Temperature {  
    var celsius: Double  
  
    init(celsius: Double) {  
        self.celsius = celsius  
    }  
  
    init(fahrenheit: Double) {  
        celsius = (fahrenheit - 32) / 1.8  
    }  
}  
  
let currentTemperature = Temperature(celsius: 18.5)  
let boiling = Temperature(fahrenheit: 212.0)  
  
print(currentTemperature.celsius)  
print(boiling.celsius)
```

Console Output:

18.5
100.0

You might notice that there's a memberwise initializer in the example above. When you add a custom initializer to a type definition, you must define your own memberwise initializers and default initializers; Swift no longer provides them for you.

You can add multiple custom initializers. The code below redefines `Temperature` to add a Kelvin initializer and a default initializer.

```
struct Temperature {
    var celsius: Double

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

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

    init(kelvin: Double) {
        celsius = kelvin - 273.15
    }

    init() {
        celsius = 0
    }
}

let currentTemperature = Temperature(celsius: 18.5)
let boiling = Temperature(fahrenheit: 212.0)
let absoluteZero = Temperature(kelvin: 0.0)
let freezing = Temperature()

print(currentTemperature.celsius)
print(boiling.celsius)
print(absoluteZero.celsius)
print(freezing.celsius)
```

Console Output:

18.5
100.0
0

Each instance of `Temperature` is created using a different initializer and a different value, but each ends as a `Temperature` object with the required `celsius` property.

Instance Methods

Instance methods are functions that can be called on specific instances of a type. They provide ways to access and modify properties of the structure, and they add functionality that relates to the instance's purpose.

As you learned earlier, you add an instance method by adding a function within a type definition. You can then call that function on instances of the type. You may have noticed this syntax in the `Person` or `Car` structures earlier.

Consider a `Size` struct with an instance method `area()` that calculates the area of a specific instance by multiplying its width and height:

```
struct Size {  
    var width: Double  
    var height: Double  
  
    func area() -> Double {  
        width * height  
    }  
}  
  
let someSize = Size(width: 10.0, height: 5.5)  
let area = someSize.area() // Area is assigned a value of 55.0
```

The `someSize` instance is of the `Size` type, and `width` and `height` are its properties. The `area()` is an instance method that can be called on all instances of the `Size` type.

Mutating Methods

Occasionally you'll want to update the property values of a structure within an instance method. To do so you'll need to add the `mutating` keyword before the function.

In the following example, a simple structure stores mileage data about a specific `Car` object. Before looking at the code, consider what data the mileage counter needs to store and what actions it needs to perform.

- Store the mileage count to be displayed on an odometer
- Increment the mileage count to update the mileage when the car drives
- Potentially reset the mileage count if the car drives beyond the number of miles that can be displayed on the odometer.

```
struct Odometer {  
    var count: Int = 0 // Assigns a default value to the `count`  
    property.  
  
    mutating func increment() {  
        count += 1  
    }  
  
    mutating func increment(by amount: Int) {  
        count += amount  
    }  
  
    mutating func reset() {  
        count = 0  
    }  
}  
  
var odometer = Odometer() // odometer.count defaults to 0  
odometer.increment() // odometer.count is incremented to 1  
odometer.increment(by: 15) // odometer.count is incremented to 16  
odometer.reset() // odometer.count is reset to 0
```

The odometer instance is of the `Odometer` type, and `increment()` and `increment(by:)` are instance methods that add miles to the instance. The `reset()` instance method resets the mileage count to zero.

Computed Properties

Swift has a feature that allows a property to perform logic that returns a calculated value.

Consider the `Temperature` example. While most of the world uses the Celsius scale of measurement for temperature, some places (such as the U.S.) use Fahrenheit, and certain professions use Kelvin. So it might be useful for the `Temperature` structure to support all three measurement systems.

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

Imagine that you'd used the memberwise initializer for this structure:

```
let temperature = Temperature(celsius: 0, fahrenheit: 32.0,  
                             kelvin: 273.15)
```

Any time you would write code to initialize a `Temperature` object, you would need to calculate *each* temperature and pass all those values as parameters.

An alternative would be to add multiple initializers that handle the calculations.

```
struct Temperature {  
    var celsius: Double  
    var fahrenheit: Double  
    var kelvin: Double  
  
    init(celsius: Double) {  
        self.celsius = celsius  
        fahrenheit = celsius * 1.8 + 32  
        kelvin = celsius + 273.15  
    }  
  
    init(fahrenheit: Double) {  
        self.fahrenheit = fahrenheit  
        celsius = (fahrenheit - 32) / 1.8  
        kelvin = celsius + 273.15  
    }  
  
    init(kelvin: Double) {  
        self.kelvin = kelvin  
        celsius = kelvin - 273.15  
        fahrenheit = celsius * 1.8 + 32  
    }  
}  
  
let currentTemperature = Temperature(celsius: 18.5)  
let boiling = Temperature(fahrenheit: 212.0)  
let freezing = Temperature(kelvin: 273.15)
```

The approach above, using multiple initializers, involves managing a lot of state, or information. Any time the temperature changes, you would be required to update all three properties. This approach is error-prone, and would be frustrating for any programmer.

Swift provides a safer approach. With computed properties, you can create properties that can compute their value based on other instance properties or logic.

```
struct Temperature {  
    var celsius: Double  
  
    var fahrenheit: Double {  
        celsius * 1.8 + 32  
    }  
  
    var kelvin: Double {  
        celsius + 273.15  
    }  
}
```

To add a computed property, you declare the property as a variable (because its value can change). You must also explicitly declare the type. Then you use an open curly brace ({}) and closing curly brace (}) to define the logic that calculates the value to return.

You can access computed properties using dot syntax, just as you would with any other property.

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

Console Output:

32.0
273.15

The logic contained in a computed property will be executed each time the property is accessed, so the returned value will always be up to date.

Property Observers

Swift allows you to observe any property and respond to the changes in the property's value. These property observers are called every time a property's value is set, even if the new value is the same as the property's current value. There are two observer closures, or blocks of code, that you can define on any given property: `willSet`, and `didSet`.

In the following example, a `StepCounter` has been defined with a `totalSteps` property. Both the `willSet` and `didSet` observers have been defined. Whenever `totalSteps` is modified, `willSet` will be called first, and you'll have access to the new value that will be set to the property value in a constant named `newValue`. After the property's value has been updated, `didSet` will be called, and you can access the previous property value using `oldValue`.

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

Here's the output when modifying the `totalSteps` property of a new `StepCounter`:

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

Console Output:

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

Type Properties And Methods

You learned that instance properties are data about the individual instance of a type, and instance methods are functions that can be called on individual instances of a type.

Swift also supports adding type properties and methods, which can be accessed or called on the type itself. Use the `static` keyword to add a property or method to a type.

Type properties are useful when a property is *related* to the type, but not a characteristic of an instance itself.

The following sample defines a `Temperature` structure that has a static property named `boilingPoint`, which is a constant value for all `Temperature` instances.

```
struct Temperature {  
    static var boilingPoint = 100  
}
```

You access type properties using dot syntax on the type name.

```
let boilingPoint = Temperature.boilingPoint
```

Type methods are similar to type properties. Use a type method when the action is related to the type, but not something that a specific instance of the type should perform.

The `Double` structure, defined in the Swift Standard Library, contains a static method named `minimum` that returns the lesser of its two parameters..

```
let smallerNumber = Double.minimum(100.0, -1000.0)
```

Copying

If you assign a structure to a variable or pass an instance as a parameter into a function, the values are copied. Separate variables are therefore separate instances of the value, which means that changing one value doesn't change the other.

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

someSize.width = 500

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

Console Output:

```
500
250
```

Note that the `width` property of `someSize` changed to have a value of 500, but the `width` property of `anotherSize` did not, because although we set `anotherSize` equal to `someSize`, this created a copy of `someSize`, and the copy's `width` did not change when the original `width` was changed.

Self

You may have noticed the word `self` in this section. In Swift, `self` refers to the current instance of the type. It can be used within an instance method or computed property to refer to its own instance.

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

Many languages require the use of `self` to refer to instance properties or methods within the type definition. The Swift compiler recognizes when property or method names exist on the current object, and makes using `self` optional.

This example is functionally the same as the previous example.

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

The use of `self` is required within initializers that have parameter names that match property names.

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

This concept is called shadowing, and you will learn more about it in a future lesson.

Variable Properties

Did you notice that all of the structure examples in this lesson used variable properties instead of constants? Consider the `Car` definition used at the beginning of the lesson.

```
struct Car {  
    var make: String  
    var year: Int  
    var color: Color  
    var topSpeed: Int  
}
```

A car's color could easily be changed with a paint job, and the top speed might get updated if the owner upgrades the engine. But the make and year of a car will never change, so why not define the properties using `let`?

Variable properties provide a convenient way to create new data from old data. In the following code, making a blue, 2010 Ford is as simple as making a copy of the 2010 Honda and modifying the `make` property. If the `make` property were constant, the second car would need to be created using the memberwise initializer.

```
var firstCar = Car(make: "Honda", year: 2010, color: .blue,  
    topSpeed: 120)  
var secondCar = firstCar  
secondCar.make = "Ford"
```

Earlier in this lesson, you learned that whenever an instance of a struct is assigned to a new variable, the values are copied. In the previous example, `firstCar` is still a Honda. If you want to explicitly prevent `firstCar`'s properties from ever changing, simply declare the instance using a `let`.

```
let firstCar = Car(make: "Honda", year: 2010, color: .blue,  
topSpeed: 120)  
firstCar.color = .red // Compiler error!
```

Even if the properties are declared using `var`, the values of a constant cannot be changed. As a general rule, you should use `let` whenever possible to define an instance of a structure, use `var` if the instance needs to be mutated, and use `var` when defining the properties of a structure.

Lab

Open and complete the exercises in [Lab—Structures.playground](#).

Connect To Design

In your App Design Workbook, reflect on the kinds of data that might be needed in your app and how you could model that data. Will you only need predefined types, such as numbers and strings, or would it be useful to define your own type of data with a structure? For structures, outline the properties and methods they might have. Make comments in the Map section or in a new blank slide at the end of the document.

In the workbook's Go Green app example, there could be a structure called `ItemEntry` for each trash or recycling log. The struct could have the properties `itemType`, `date`, `weight`, and `name`. It could also have a method called `environmentalImpact()` that would calculate the amount of CO₂ or other pollution of an item based on its name (such as "paper") and `weight`.

Review Questions

Question 1 of 3

When is it useful to define a custom structure?

- A. When you want to represent simple data already handled by a Swift type
- B. When you want to represent a new type of data with a collection of properties and functions

Check Answer



Lesson 2.5

Classes and Inheritance

You've now learned about structures as a way to compile data and functionality into a type. Many programming languages also support another feature, called classes, that perform similar functionality. In special cases, classes can (and should) be used instead of structures.

Classes and structures are very similar, and either can be used as the building blocks for your program or app. In this lesson, you'll learn what makes classes different than structures and when to use classes instead of structures. You'll also learn about inheritance, superclasses, and subclasses.

What You'll Learn

- The difference between a structure and a class
 - How to define a class
 - The concept and importance of inheritance
 - How to write a class that inherits from another class
 - How to use a class to manage complex states in an application
-

Vocabulary

- **base class**
 - **class**
 - **inheritance**
 - **state**
 - **subclass**
 - **superclass**
-

Related Resources

- [Swift Programming Language Guide: Classes and Structures](#)
- [Swift Programming Language Guide: Inheritance](#)

Classes are very similar to structures. Among other similarities, both can define properties to store values, define methods to provide functionality, and define initializers to set up the initial state. The syntax for doing these things is almost always identical.

You define a class using the `class` keyword along with a unique name. You then define properties as part of the `class` by listing the constant or variable declarations with the appropriate type annotation. As with structures, it's best practice to capitalize the names of types and to use lowercase for the names of properties:

```
class Person {  
    let name: String  
  
    init(name: String) {  
        self.name = name  
    }  
}
```

You can add functionality to a class by adding functions to the class definition:

```
class Person {  
    let name: String  
  
    init(name: String) {  
        self.name = name  
    }  
  
    func sayHello() {  
        print("Hello, there!")  
    }  
}  
  
let person = Person(name: "Jasmine")  
print(person.name)  
person.sayHello()
```

Console Output:

Jasmine
Hello, there!

You've seen an almost-identical example in the previous lesson on structures. But how do classes differ from structures?

Inheritance

The biggest difference between structures and classes is that classes can have hierarchical relationships. Any class can have parent and child classes. A parent class is called a superclass, and a child class is called a subclass.

Just as in family relationships, subclasses inherit properties and methods from superclasses. However, subclasses can augment or replace the implementation of superclass properties and methods.

In the sections below, you'll read how to define a base class, which is a class that has no parent classes, how to define subclasses, and how to override properties or methods from the base class.

Defining a Base Class

A class that doesn't inherit from a superclass is known as a base class. All the classes that you've seen so far are base classes.

Here's an example of a base class for a `Vehicle` object:

```
class Vehicle {  
    var currentSpeed = 0.0  
  
    var description: String {  
        "traveling at \(currentSpeed) miles per hour"  
    }  
  
    func makeNoise() {  
        // do nothing – an arbitrary vehicle doesn't necessarily  
        // make a noise  
    }  
}  
  
let someVehicle = Vehicle()  
print("Vehicle: \(someVehicle.description)")
```

Console Output:

Vehicle: traveling at 0.0 miles per hour

The `Vehicle` class has a property called `currentSpeed` with a default value of `0.0`. The `currentSpeed` property is used in the computed `description` variable that returns a human readable description of the vehicle. The class also has a method called `makeNoise()`. This method does not actually do anything for a base `Vehicle` instance, but will be customized by subclasses later.

The `Vehicle` class defines common characteristics, like properties and methods, for any type of vehicle—like cars, bicycles, or airplanes. It is not very useful on its own, but it will make it easier to define other more specific types.

Create a Subclass

Subclassing is the act of basing a new class on an existing class. The subclass inherits properties and methods from the superclass, which you can then refine and make more specific. You can also add new properties or methods to the subclass.

To define a new type that inherits from a superclass, write the type name before the superclass name, separated by a colon:

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

The following example defines a subclass called `Bicycle`, with a superclass of `Vehicle`:

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

The new `Bicycle` class automatically inherits all of the properties and methods of `Vehicle`, such as its `currentSpeed` and `description` properties and its `makeNoise()` method.

In addition to the inherited properties and methods, the `Bicycle` class defines a new boolean property `hasBasket`. By default, new instance of `Bicycle` will not have a basket. You can set the `hasBasket` property to `true` for a particular `Bicycle` instance after it has been created, or within a custom initializer for the type.

```
let bicycle = Bicycle()  
bicycle.hasBasket = true
```

As you'd expect, you can also update the `currentSpeed` property, which will impact the description of the instance.

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

Console Output:

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

Subclasses can themselves be subclassed. For example, you can create a class that represents a two-seater `TandemBike`.

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

`Tandem` inherits all of the properties and methods from `Bicycle`, which in turn inherits all of the properties and methods from `Vehicle`. The `Tandem` subclass also adds a new property called `currentNumberOfPassengers`, with a default value of `0`.

If you create a new instance of `Tandem`, you'll have access to all of the inherited properties and methods.

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

Console Output:

Tandem: traveling at 22.0 miles per hour

Override Methods and Properties

One advantage of subclassing is that each subclass can provide its own custom implementation of a property or method. To override a characteristic that would otherwise be inherited, like writing a new implementation for a function, you prefix your new definition with the `override` keyword.

The following example defines a new `Train` class, which overrides the `makeNoise()` method that `Train` inherits from `Vehicle`:

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

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

Console Output:

Choo Choo!

You can also override properties by providing a getter, or a block of code that returns the value, like a computed property. The following example defines a new class called `Car`, which is a subclass of `Vehicle`. The new class has a new `gear` property, with a default value of `1`. The `Car` class also overrides the `description` property to provide a custom description that includes the current gear:

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

Notice that the new implementation accesses the `description` from the superclass by calling `super.description`. The new implementation then adds some extra text onto the end of the description to provide information about the gear.

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

Console Output:

Car: traveling at 25.0 miles per hour in gear 3

Override Initializer

Suppose you have a `Person` class with a `name` property. The initializer sets the `name` to the parameter specified.

```
class Person {  
    let name: String  
  
    init(name: String) {  
        self.name = name  
    }  
}
```

Now imagine you want to create a `Student`, which is a subclass of `Person`. Each `Student` includes an additional property, `favoriteSubject`.

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

If you try to compile this code, `Student` will fail, because you haven't provided an initializer that sets the `favoriteSubject` property to an initial value. Since the `Person` superclass already does the work of initializing the `name` property, the `Student` initializer can call the superclass's initializer using `super.init()`. You should call this initializer after you've provided values for any properties added within your subclass.

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

By calling the superclass's initializer, the `Student` class doesn't have to duplicate the work that was already written in the `Person` initializer.

References

A special feature of classes is their ability to reference values assigned to a constant or variable. When you create an instance of a class, Swift picks out a region in the device's memory to store that instance. That region in memory has an address. Constants or variables that are assigned the instance store that address to refer to the instance.

So the constant or variable does not contain the value itself, it points to the value in memory.

When you assign a class to multiple variables, each variable will reference, or point to, the same address in memory. So if you update one of the variables, both variables will be updated.

```
class Person {  
    let name: String  
    var age: Int  
  
    init(name: String, age: Int) {  
        self.name = name  
        self.age = age  
    }  
}  
  
var jack = Person(name: "Jack", age: 24)  
var myFriend = jack  
  
jack.age += 1  
  
print(jack.age)  
print(myFriend.age)
```

Console Output:

25

25

In contrast, when you create an instance of a structure, you're assigning a literal value to that variable. If you create a variable that's equal to another structure variable, the value is copied. Any operations you perform on either variable will be reflected only on the *modified* variable.

```
struct Person {  
    var name: String  
    var age: Int  
}  
  
var jack = Person(name: "Jack", age: 24)  
var myFriend = jack  
  
jack.age += 1  
  
print(jack.age)  
print(myFriend.age)
```

Console Output:

25

24

Memberwise Initializers

You've learned that all property values must be set when you create an instance of a class. Unlike for structures, Swift does *not* create a memberwise initializer for classes. However, it's common practice for developers to create their own memberwise initializer for classes they define, ensuring that all initial values are set for each instance of any object.

Class or Structure?

Because classes and structures are so similar, it can be hard to know when to use classes and when to use structures for the building blocks of your program.

As a basic rule, you should start new types as structures until you need one of the features that classes provide.

Start with a class when you're working with a framework that uses classes or when you want to refer to the same instance of a type in multiple places.

Working with Frameworks That Use Classes

You'll often work with resources that you didn't write, such as frameworks like Foundation or UIKit. In these situations, you may start with a base class, then create a subclass to add your own specific functionality. For example, when you're working with UIKit and want to create a custom view, you create a subclass of UIView.

When working with frameworks, it's often an expectation that you'll pass around class instances. Many frameworks have method calls that expect certain things to be classes. So in these cases, you'll always choose to use a class over a structure.

Stable Identity

There are times when you want to use a single instance of an object in multiple places, but it doesn't make sense to copy the instance. Because each class constant or variable is an address that points to the same data, the data has a *stable identity*.

Consider a `UITableViewCell` subclass `MessageCell` that represents a row in a table view. The cell is designed to display information about an email message. Each instance of a `MessageCell` will be accessed in many places in code, as you will see.

```
class MessageCell: UITableViewCell {  
  
    func update(message: Message) {  
        // Update `UITableViewCell` properties with information  
        // about the message  
       .textLabel.text = message.subject  
        detailTextLabel.text = message.previewText  
    }  
}
```

When a table view is preparing to display cells, it calls a function `cellForRow(at: IndexPath)` to request the cell it should display at each position in the list. All cells for a table view are initialized (and put into memory) during this function call.

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {  
  
    let cell = MessageCell(style: .default, reuseIdentifier:  
    "MessageCell")  
    cell.update(message: message)  
  
    // Returns the cell to the table view that called this method  
    // to request it  
    return cell  
}
```

When the user scrolls through the list of messages in the table view, a method is called that allows your program to set up, update, or modify the cell as it is about to be displayed. The function passes a reference to the cell as a parameter.

```
override func tableView(_ tableView: UITableView, willDisplay  
cell: UITableViewCell, forRowAt indexPath: IndexPath) {  
    // Perform any operations you want to take on the cell here  
}
```

There's some unfamiliar syntax here, which you'll learn more about later. For now, focus on the fact that the function is called for each cell that's about to appear on the table view. Also notice that the function passes three parameters: a reference to the `tableView`, which already exists; the `cell` that already exists but is about to be displayed; and something called `indexPath`.

Because `tableView` and `cell` are both classes, the parameters are references to the `tableView` and `cell` in memory. If you update the `cell` parameter, you're updating the same cell that was initialized in the `cellForRow` function, the same cell that's about to be displayed.

Consider how a class works differently from a structure. If `cell` were a structure, or a value type, the parameter would be a *new copy* of the cell that's about to be displayed. So if you updated `cell`, you wouldn't see your changes, because you updated the copy and not the cell that's about to be displayed.

Having a hard time understanding references in the context of stable identity? That's OK. You'll learn more in future lessons, when you'll need to use these concepts to complete a project.

Lab

Open and complete the exercises in [Lab—Classes.playground](#).

Review Questions

Question 1 of 3

Which of the following statements are true about these classes?

```
class Scientist {}  
class Geologist: Scientist {}  
class Physicist: Scientist {}  
class Astrophysicist: Physicist {}
```

- A. Scientist is a base class.
- B. There is only one subclass of Scientist.
- C. Geologist, Physicist, and Astrophysicist are all descendants of Scientist.
- D. Astrophysicist does not inherit from Scientist.

[Check Answer](#)



Lesson 2.6

Collections

So far in this course, you've learned to work with single items, whether constants or variables. But sometimes you'll find that it makes more sense to work with a group of objects as a whole. A collection allows you to reference multiple objects at once. For example, you could handle a dozen eggs using one constant instead of managing twelve separate constants. In this lesson, you'll learn about the various collection types available in Swift and how to choose the appropriate one for your program.

What You'll Learn

- How to declare constant and variable collections
 - How to add and remove values from arrays and dictionaries
 - How to choose the appropriate collection type
-

Vocabulary

- [array](#)
 - [dictionary](#)
-

Related Resources

- [Swift Programming Language Guide: Collection Types](#)

Swift defines two collection types you will frequently work with: arrays and dictionaries. Each type provides a unique method for interacting with multiple objects. As you progress through learning the different types, you'll notice that they share certain functionality: adding/removing items, accessing individual items, and providing type information about the data within the collection.

Arrays

The most common collection type in Swift is an array, which stores an ordered list of same-typed values. When you declare an array, you can specify what type of values will be held in the collection, or you can let the type inference system discover the type.

An array is often initialized using a literal, similar to an `Int` or `String` literal. To declare an array literal, surround the collection of values with brackets, with commas separating the values:

```
[value1, value2, value3]
```

You'll use similar syntax to declare an array that stores strings. Notice in the following example that the variable's type, `String`, is surrounded by brackets:

```
var names: [String] = ["Anne", "Gary", "Keith"]
```

Since type inference can determine that "Anne", "Gary", and "Keith" are strings, you could actually skip a step and declare the array without specifying the array's type:

```
var names = ["Anne", "Gary", "Keith"]
```

But there might be situations when you want to specify the array's type even though type inference can discover it. Imagine, for example, you want a collection of 8-bit integers (numbers between -127 and 128). You begin by assigning a variable to the following collection of numbers:

```
var numbers = [1, -3, 50, 72, -95, 115]
```

Swift will infer all the values to be of type `Int` and set `numbers` to be an array of integers, or `[Int]`. While this inference is certainly correct, there's a problem: An `Int` can hold positive and negative numbers that exceed beyond the range from -127 to 128. To help Swift understand that you want to restrict the array to smaller integers, you can specify the type as `[Int8]`, an array of 8-bit integers whose values are restricted to the aforementioned range:

```
var numbers: [Int8] = [1, -3, 50, 72, -95, 115]
```

What if you tried to include a larger number, such as 300, in an array literal of type `[Int8]`? Swift will return an error because 300 is greater than the maximum number for an `Int8`. You should specify the type if the inferred type is not specific enough, and it will help to restrict the values that you don't want to allow.

It's often useful to check if a certain value exists in an array. To do so, you can use the `contains(_:_)` method, passing in the desired value. If the array contains the value, the expression is true; otherwise, it's false:

```
let numbers = [4, 5, 6]
if numbers.contains(5) {
    print("There is a 5")
}
```

Early in this course, you learned that constants can't be modified once they're declared. It's the same when you assign a collection to a constant using `let`: You can't add, remove, or modify any items within the collection. However, if you store the collection within a variable using `var`, you'll be able to add to, remove from, or modify items in the collection. In addition, you'll be able to empty the collection or set the variable to a different collection entirely.

Array Types

An array is like a basket: It can start out empty and you can fill it with values at a later time. But if an array literal doesn't contain any values, how can its type be inferred?

You can declare the type of an array using type annotation, collection type annotation, or an array initializer.

This example defines an array with traditional type annotation.

```
var myArray: [Int] = []
```

This example defines an array using a special collection type annotation. This is a less common practice you should be familiar with in case you run across it in code you're working with.

```
var myArray: Array<Int> = []
```

Just like all objects can be initialized by adding a () after the type name, you can add a () after [Int] to initialize an empty array of Int objects. You should also be familiar with this code in case you run across it in the future.

```
var myArray = [Int]()
```

Working With Arrays

You may find some situations when it's tedious or error-prone to use array literals. For example, say you want an array filled with 100 zeros. In array literal form, you'd have to enter all 100 zeros, and it would be easy to miscount them.

That's OK. Swift has a solution. Instead of counting out the 100 zeros, you can use the following initializer to create an array of count `100` with repeating default values:

```
var myArray = [Int](repeating: 0, count: 100)
```

To find out the number of items within an array, you can use its `count` property. What if you wanted to check if the array is empty? Rather than comparing `count` to 0, you can check the array's `isEmpty` property, which returns a `Bool`:

```
let count = myArray.count
if myArray.isEmpty { }
```

Once you've defined an array, you can use various methods and properties to access or modify it. You can also use an array's subscript syntax. Subscript syntax uses square brackets after a collection to refer to a specific element or range inside the collection. To retrieve a particular value from a collection, specify inside square brackets the index of the value you wish to access. An array's values are always zero-indexed, meaning that the first element of an array has an index of zero. The following example will retrieve the first value in a collection of first names:

```
let firstName = names[0]
```

You can also use subscript syntax to change a value at a given index. By putting the subscript on the left side of the `=` sign, you can set a particular value rather than access the current one. In the example below, you're changing the second name in the collection to "Paul".

```
names[1] = "Paul"
```

When subscripting an array, you need to be sure that the index you specify exists in the array. For example, if you specify 3 as the index, the array must have at least four elements. If it doesn't, your program will crash when that line of code is executed.

After defining an array and setting some values, you'll often want to `append` new values. You can use a variable array's `append` function to add a new element at the end of the array. To append multiple elements at once, use the `+=` operator.

```
var names = ["Amy"]
names.append("Joe")
names += ["Keith", "Jane"]
print(names) // ["Amy", "Joe", "Keith", "Jane"]
```

What if you didn't want the new element to show up at the end of the array? A variable array also has an `insert(_:_:)` function that allows you to specify the value and the index. In this scenario, as with subscripting, you'll need to be sure that you supply a valid index.

Because an array is zero-indexed, the first element always has an index of 0, and the last element's index is equal to `count - 1`. In the following example, "Bob" is inserted at the beginning of the array:

```
names.insert("Bob", at: 0)
```

Similarly, the `remove(at:)` function works to remove an item at a specified index. Unlike `insert(_:_:)`, you don't need to enter the value; the function returns the removed item by default, as in the example below. Another method is `removeLast()`, which removes the last item from the array, eliminating the need to calculate the index. Finally, the `removeAll()` method will remove all elements from the array.

```
var names = ["Amy", "Brad", "Chelsea", "Dan"]
let chelsea = names.remove(at: 2)
let dan = names.removeLast()
names.removeAll()
```

Swift also allows you to create a new array by adding together two existing arrays of the same type. You can specify which array comes first within the resulting array:

```
var myNewArray = firstArray + secondArray
```

What about arrays within an array? Use the array literal syntax to place two arrays inside another containing array. You can then use subscript syntax on the container array to access one of the nested arrays. To access a particular element within one of the nested arrays, you can use two sets of subscripts.

```
let array1 = [1, 2, 3]
let array2 = [4, 5, 6]
let containerArray = [array1, array2] // [ [1,2,3], [4,5,6] ]
let firstArray = containerArray[0] // [1,2,3]
let firstElement = containerArray[0][0] // 1
```

Dictionaries

The second type of collection in Swift is a dictionary. Like a real-world dictionary that contains a list of words and their definitions, a Swift dictionary is a list of keys, each with an associated value. Each key must be unique, just like each word in the dictionary is unique. And just as an English dictionary is in alphabetical order to make the words easy to look up, a Swift dictionary is optimized to make key lookups very fast.

You can set up a dictionary using a dictionary literal: a list of comma-separated key/value pairs surrounded by square brackets. A colon separates each key and its resulting value:

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

Just as with an array, the dictionary's type can be inferred based on the types used in the dictionary literal. Say you want to store a list of high scores in a game. You'll use the player's name as the key and the player's score as the corresponding value. The dictionary's type will be inferred as a dictionary where the keys are of type `String` and the values are of type `Int`. This can be represented as either `[String: Int]` or `Dictionary<String, Int>`:

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

As with other collection types, a dictionary has a `count` property to determine the number of key-value pairs and an `isEmpty` property to determine whether the dictionary has no key-value pairs. The syntax for creating an empty dictionary is probably familiar by now:

```
var myDictionary = [String: Int]()

var myDictionary = Dictionary<String, Int>()

var myDictionary: [String: Int] = [:]
```

All of these examples result in the same type of empty dictionary. You'll likely have a preferred method you use regularly, but you should be familiar with all of them. Take note of the syntax for an empty dictionary literal in the last example.

Add/Remove/Modify A Dictionary

Subscript syntax is particularly handy with a dictionary. Since the order in a Swift dictionary doesn't matter, there's no index and there's no risk of subscripting errors associated with indices.

The following example adds a new score to the dictionary of high scores. If the key "Oli" already exists in the dictionary, this code will replace the old value with the new one:

```
scores["Oli"] = 399
```

But what if you want to know if there's an old value in the dictionary *before* replacing it? You can use `updateValue(_:, forKey:)` to update the dictionary, and the value returned from the method will be equal to the old value, if one existed. In the following example, `oldValue` will be equal to Richard's old value before the update. If there was no value, `oldValue` will be `nil`. You'll learn more about the `nil` keyword later. It's a special way of representing the *absence* of a value.

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

Swift uses if-let syntax to let you run code *only if* a value is returned from the method. If there wasn't an existing value, the code within the braces wouldn't be executed:

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

To remove an item from a dictionary, you can use subscript syntax, setting the value to `nil`. Similar to updating a value, dictionaries have a `removeValue(forKey:)` method if you need the old value returned before removing it:

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

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

Accessing A Dictionary

Swift dictionaries provide two properties not included in other collection types. You can use `keys` to return a list of all the keys within a dictionary and `values` to return a list of all the values. If you want to use these collections subsequently, you'll need to convert them to arrays:

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

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

To look up a particular value within a dictionary, use if-let syntax. If the key you specify is in the dictionary, the result will be the key's corresponding value. However, if the key isn't in the dictionary, the code within the brackets won't be executed.

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

if let henrysScore = scores["Henry"] {
    print(henrysScore) // not executed; "Henry" is not a key in
                      // the dictionary
}
```

Console Output:

400

Lab

Open and complete the exercises in [Lab—Collections.playground](#).

Connect To Design

Open your App Design Workbook and review the Map section for your app. Reflect on the kinds of information your app might need to use. Is there information in your app that you will need to use a collection for—that is, where you will need to reference multiple objects at once? Add comments to the Map section or in a new blank slide at the end of the document. Try to identify the information you will need an array or dictionary for in your app.

In the workbook's Go Green app example, each time a user logs an item they could create a new instance of a structure called 'ItemEntry'. The new instance could be added to an array of items that keeps track of all the log entries.

Review Questions

Question 1 of 2

Given the following dictionary, what is the result of executing
numberOfLegs["snake"] = 0?

```
var numberOfLegs = ["spider": 8, "human": 2, "dog": 4,  
"cat": 4]
```

- A. Nothing; `numberOfLegs` cannot be modified.
- B. "snake" is added as a key, with a value of 0.
- C. "snake" is updated to have a value of 0 instead of 4.
- D. An error is thrown because two keys in the dictionary have the same value.

Check Answer



Lesson 2.7

Loops

 You can probably think of many everyday tasks that you continue performing until a condition is met. You might continue filling a water bottle until it's full or continue doing homework until the assignment is complete.

Scenarios that require completion and repetition of a task can be done in code using loops. In this lesson, you'll learn how to create loops in Swift, control the conditions for looping, and specify when to stop.

What You'll Learn

- How to step through each value of a collection using a `for` loop
 - How to iterate through a range of values
 - How to write a loop that continues until a condition is no longer true
-

Vocabulary

- `closed range operator`
 - `for-in loop`
 - `half-open range operator`
 - `iteration`
 - `repeat-while loop`
 - `while loop`
-

Related Resources

- [Swift Programming Language Guide: For-in Loops](#)
- [Swift Programming Language Guide: Range Operators](#)
- [Swift Programming Language Guide: While Loops](#)

Computer programs are great at repeating things. Developers often write code to perform the same work across multiple objects, perform a task many times, or continue to perform work until specific conditions have been met. Swift provides three different ways to loop through, or repeat, blocks of code.

For Loops

The first loop you'll learn is the `for` loop, also known more specifically as a `for-in` loop. A `for` loop is useful for repeating something a set number of times or for performing work across a collection of values.

A `for-in` loop executes a set of statements for each item within a range, sequence, or collection. Suppose you have a range of numbers between 1 and 5, and you want to print each value within the range. Rather than writing out five `print` statements, you can use `for-in` over the range and write one `print` statement. The syntax looks like this:

```
for index in 1...5 {  
    print("This is number \(index)")  
}
```

Console Output:

This is number 1
This is number 2
This is number 3
This is number 4
This is number 5

The `...` is known as the *closed range operator*, which is how you define a range of values that runs from `x` up to `y` and includes both `x` and `y` in the range. There's a companion operator `(..y)` known as the *half-open range operator*. These ranges run from `x` up to `y`, but don't include `y`.

In the code above, `index` is a constant that's available to customize the work performed within the braces (the `for-in` loop). The first time the statement within the braces is executed, `index` has a value of 1, the first value in the range. When execution is complete, the value of `index` is updated to 2, the next value in the range. As `index` is updated to each of the values, the `print` statement is executed five times. After the entire range has been exhausted, the loop is complete, and the code moves on to statements after the loop.

But let's say your result doesn't need to use the values in the range. If you just need a way to perform a series of steps a certain number of times, you can skip assigning a value to a constant and replace its name with a `_`:

```
for _ in 1...3 {  
    print("Hello!")  
}
```

Console Output:

Hello!
Hello!
Hello!

You can use the same `for-in` syntax to iterate over each item in an array.

```
let names = ["Joseph", "Cathy", "Winston"]  
for name in names {  
    print("Hello \(name)")  
}
```

Console Output:

Hello Joseph
Hello Cathy
Hello Winston

Because `String` is a collection type, it has similar functionality to an array. You can use a `for-in` loop to iterate over each character in a `String`:

```
for letter in "ABCD" {  
    print("The letter is \(letter)")  
}
```

Console Output:

The letter is A
The letter is B
The letter is C
The letter is D

What if you need the index of each element in addition to its value? You can use the `enumerated()` method of an array or string to return a tuple—a special type that can hold an ordered list of values wrapped in parentheses—containing both the index and the value of each item:

```
for (index, letter) in "ABCD".enumerated() {  
    print("\(index): \(letter)")  
}
```

Console Output:

0: A
1: B
2: C
3: D

Alternatively, you could use the half-open range operator to do the same thing:

```
let animals = ["Lion", "Tiger", "Bear"]
for index in 0..
```

Console Output:

0: Lion
1: Tiger
2: Bear

If you use a `for-in` loop with a dictionary, the loop generates a tuple that holds the key and value of each entry. Because a dictionary is typically accessed by specifying a key, the loop doesn't guarantee any particular order of the items as it works through the dictionary:

```
let vehicles = ["unicycle": 1, "bicycle": 2, "tricycle": 3,
"quad bike": 4]
for (vehicleName, wheelCount) in vehicles {
    print("A \(vehicleName) has \(wheelCount) wheels")
}
```

Console Output:

A unicycle has 1 wheels
A bicycle has 2 wheels
A tricycle has 3 wheels
A quad bike has 4 wheels

While Loops

A while loop will continue to loop until its specified condition is no longer true. Imagine you want to keep playing a game until you run out of “lives.” The condition under which you continue looping is that the number of lives is greater than 0:

```
var numberOfLives = 3

while numberOfLives > 0 {
    playMove()
    updateLivesCount()
}
```

Swift checks the condition before each loop is executed, which means it’s possible to skip the loop entirely if the condition is never satisfied. If `numberOfLives` had been initialized to 0 in the above example, the while loop would determine that `0 > 0` is false and would never proceed to the body of the loop.

For this reason, the body of the while loop should perform work that will eventually change the condition. In the example below, nothing is updating the value of `numberOfLives`, so the condition always resolves to `true` and continues forever.

```
var numberOfLives = 3

while numberOfLives > 0 {
    print("I still have \(numberOfLives) lives.")
}
```

Instead, the body should execute statements that will, at some point, result in a `false` condition:

```
var numberOfLives = 3
var stillAlive = true
while stillAlive {
    numberOfLives -= 1
    if numberOfLives == 0 {
        stillAlive = false
    }
}
```

Repeat-While Loops

A repeat-while loop is similar to the `while` loop, but this syntax executes the block once before checking the condition.

```
var steps = 0
let wall = 2 // there's a wall after two steps

repeat {
    print("Step")

    steps += 1

    if steps == wall {
        print("You've hit a wall!")
        break
    }
} while steps < 10 // maximum in this direction
```

Console Output:

```
Step
Step
You've hit a wall!
```

Control Transfer Statements

You may have situations when you want to stop execution of a loop from within the loop's body. The Swift keyword `break` will break the code execution within the loop and start executing any code defined after the loop (you may recognize this from the `switch` statement).

In the code below, the loop breaks if the counter reaches 0:

```
// Prints -3 through 0
for counter in -3...3 {
    print(counter)
    if counter == 0 {
        break
    }
}
```

Console Output:

```
-3
-2
-1
0
```

There may also be situations in which you want to skip to the next iteration in a loop. While the `break` keyword will end the loop entirely, `continue` will move onto the next iteration. For example, you may have an array where each element is of type `Person`, and you want to loop through the array and send an email to everyone 18 years of age and older:

```
for person in people {  
    if person.age < 18 {  
        continue  
    }  
  
    sendEmail(to: person)  
}
```

Computers are incredibly good at performing tasks as many times as requested. Whether you need to iterate through a collection or repeat a series of steps until a condition is met, loops are an important concept to understand well. You'll use them frequently throughout your programming career.

Lab

Open and complete the exercises in [Lab—Loops.playground](#).

Connect To Design

In your App Design Workbook, reflect on the kinds of actions that will need to be repeated in your app. This will require you to dig into the structure of your app's code a bit more. Rather than thinking about what the user will do, think about how you could use loops to control the conditions for when and how to repeat certain blocks of code. Make comments in the Map section or in a new blank slide at the end of the document.

In the workbook's Go Green app example, a loop could be used to calculate the average weight of a recycled item by stepping through an array of all the logged entries of recycled items, adding up their weights, and dividing by the number of items in the array.

Review Questions

Question 1 of 2

Which of the following are used to loop over a section of code?

- A. for, if, while
- B. while, Range, if
- C. for, while, do
- D. for, while

Check Answer



Lesson 2.8

Introduction to UIKit

As a user of computers, appliances, and all sorts of devices, you already know that user interface is important. Now that you're creating your own apps on iOS, you'll rely heavily on `UIKit`, a foundational framework for building and managing user interfaces, or UIs. `UIKit` defines how you display information to the user and how you respond to user interactions and system events. It also allows you to work with animations, text, and images. Aside from games, if you can see it on an iOS device, it was likely built with `UIKit`.

In this lesson, you'll learn about some of the most commonly used interface elements in `UIKit` and where you can go to find out more.

What You'll Learn

- Why `UIKit` is such an important part of app development
 - The name and appearance of five common views in apps
 - The name and functionality of five controls in apps
 - Where to find out more
-

Vocabulary

- | | | |
|---------------------------------|-------------------------------------|------------------------------|
| • button | • label | • tab bar |
| • control | • navigation bar | • table view |
| • control event | • scroll view | • text field |
| • date picker | • segmented control | • toolbar |
| • image view | • slider | |
| • UIKit | • switch control | |

Related Resources

- [Human Interface Guidelines](#)

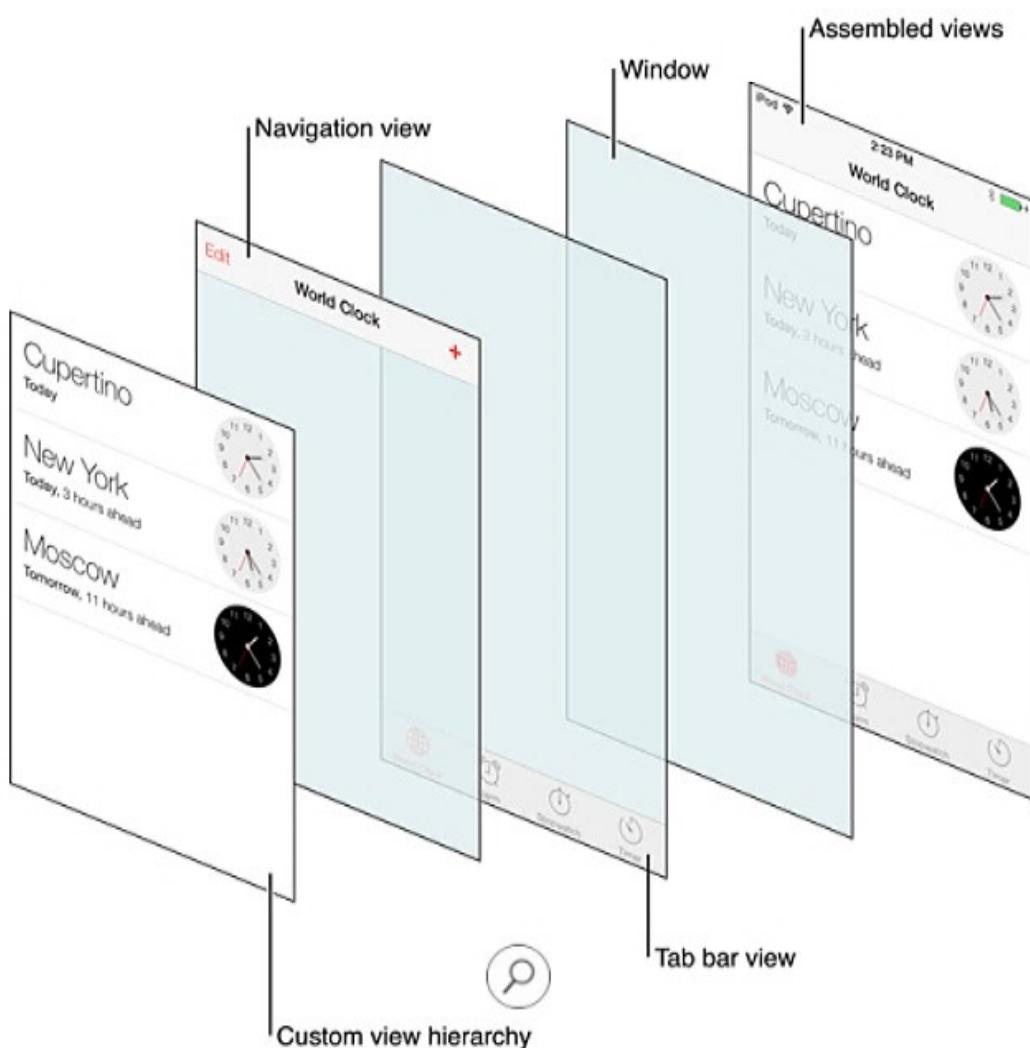
The UIKit framework provides the crucial pieces you need to build and manage iOS apps. It includes definitions for all user interface objects, the event-handling system that responds to user input, and the entire model that allows apps to run in iOS.

Common System Views

The foundational class for all visual elements defined in UIKit is the `UIView`, or view. A view defines a rectangular shape that can be customized to display anything on the screen. Text, images, lines, and graphics all depend on `UIView` as the base.

UIKit defines dozens of special `UIView` subclasses that perform specific tasks. For example, `UILabel` displays text, `UIImageView` displays an image, and `UIScrollView` allows you to put scrollable content onto the screen.

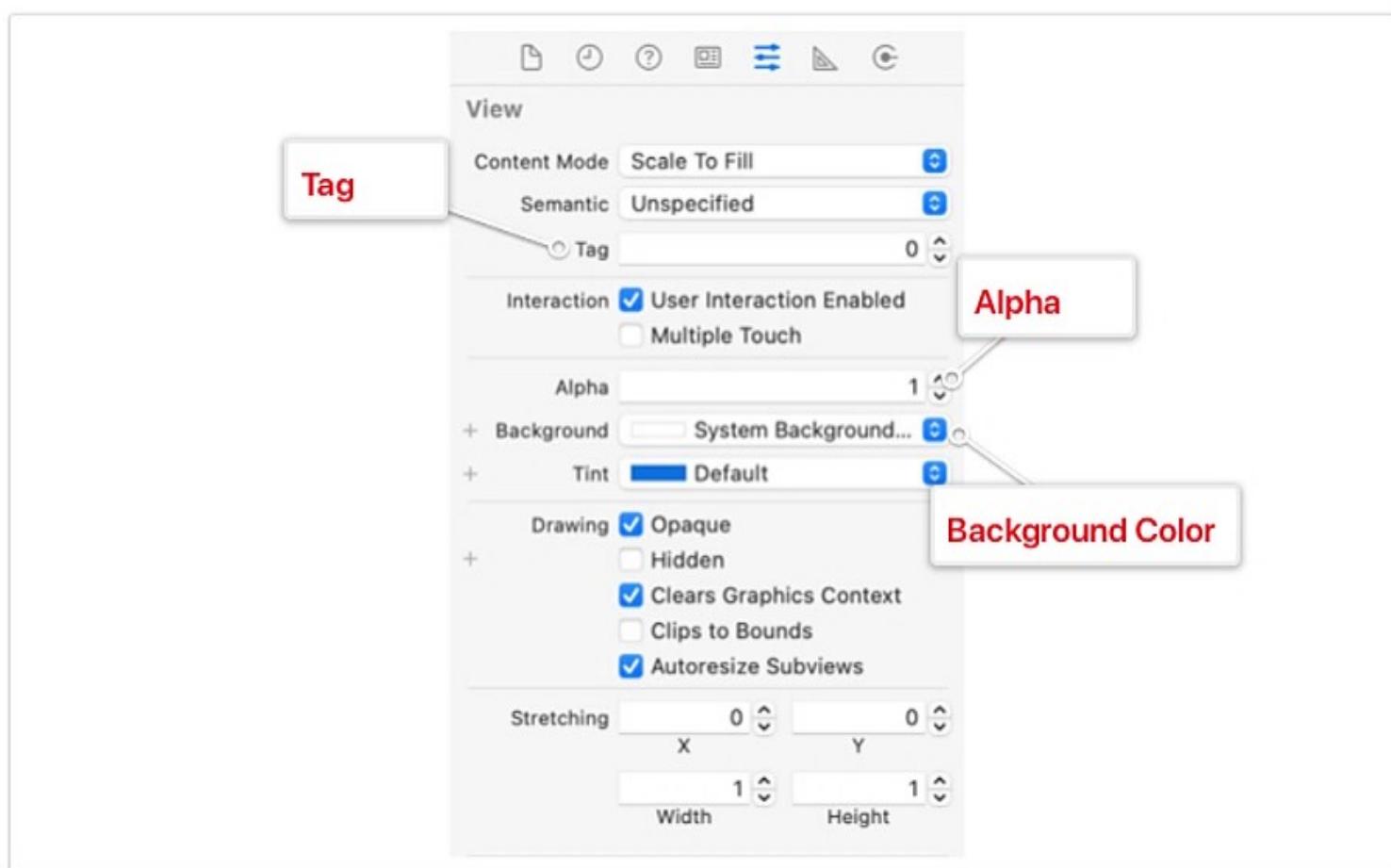
Almost all screens in an app consist of many views, which together make up a view hierarchy.



Views are often nested, as in the previous view hierarchy diagram. A view that's contained in another view is called a child view. A view that contains one or more views is called a parent view. In this example, each cell in the list is a child view of a table view, and each cell is a parent to three views: a label that displays a city name, another label that displays how the time zones are related, and an image view that displays an image of a clock.

To display a view onscreen, you need to give it a frame—which consists of a size and a position—and add it to the view hierarchy. The area within the view is its bounds. When adding a view in Interface Builder, its background color is white by default. When creating a view in code, the background is transparent by default. You can set a new background color in either scenario.

Here are some of the attributes you can change when working with views:



Next, take a look at some of the most common views defined in UIKit.

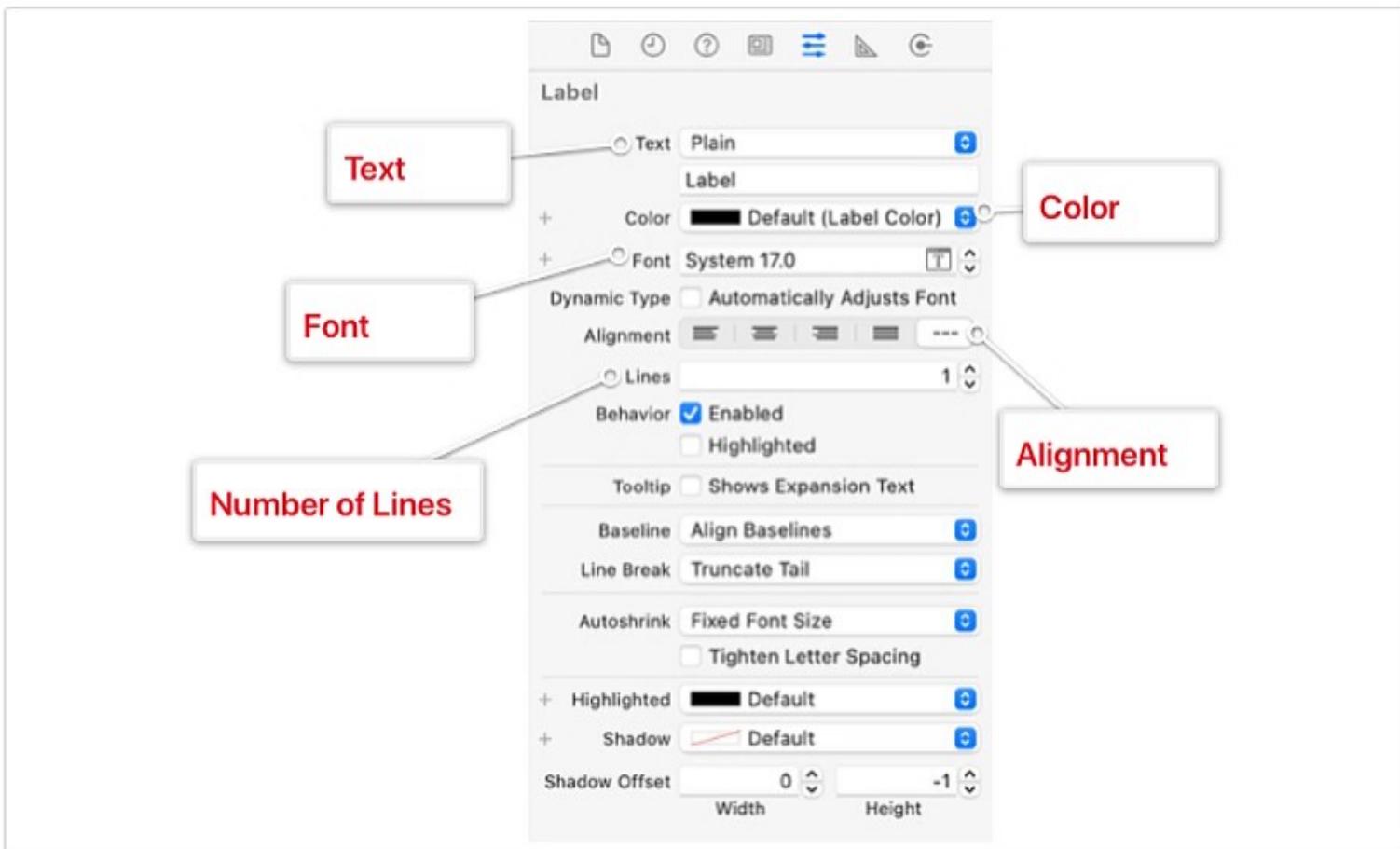
This section covers some common views you'll work with using `UIView`. As you learn about the different views, think about apps you use and which views might be used to create their interfaces.

Label (UILabel)

Labels use static text to relay information to the user.

The volume of the ringer and alerts can be adjusted using the volume buttons.

Configuration



[UILabel Developer Documentation](#)

Image View (UIImageView)

An image view displays an image or an animated sequence of images. Some people confuse the image view with the image itself. Think of it this way: Just like a label displays text, an image view displays an image.



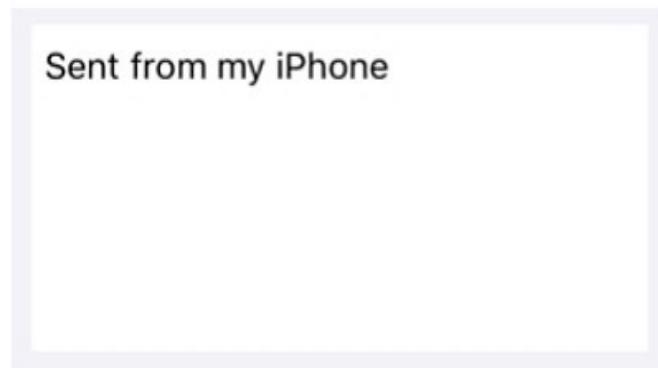
Configuration

The screenshot shows the Xcode Interface Builder Attributes Inspector for an UIImageView. At the top, there are two buttons: "Highlighted Image" (highlighted in red) and "Image". Below them is a toolbar with icons for file, undo, redo, help, and other utilities. The main area is titled "Image View" and contains the following settings:

- Image:** A dropdown menu set to "Image".
- Highlighted:** A dropdown menu set to "Highlighted Image".
- State:** A checkbox labeled "Highlighted" is checked.
- Accessibility:** A checkbox labeled "Adjusts Image Size" is checked.

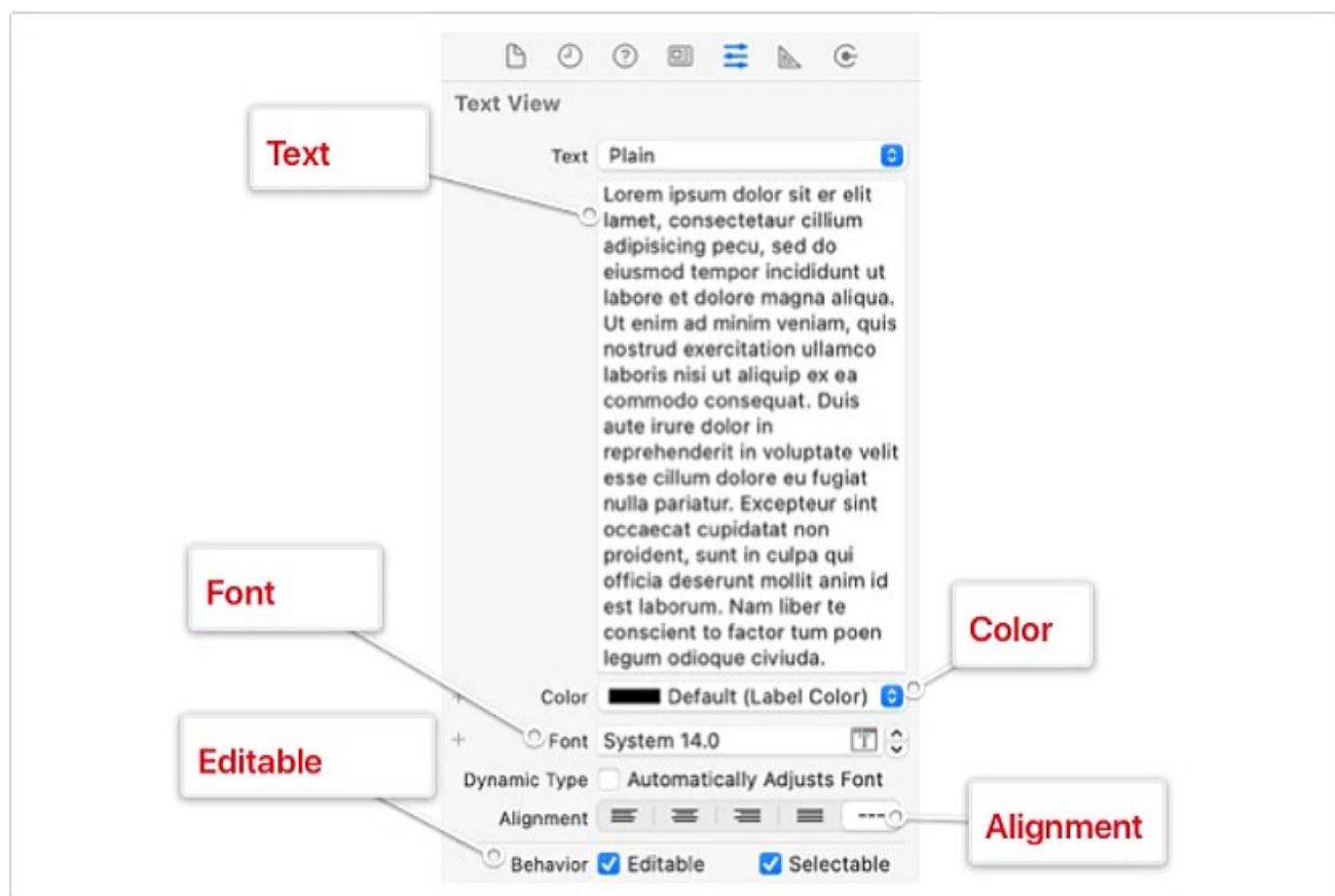
[UIImageView Developer Documentation](#)

Text View (UITextView)



A text view allows the user to input text in your app. Text views accept and display multiple lines of text, with support for scrolling and editing. You'll typically use a text view to display a large amount of text, such as the body of an email message.

Configuration



[UITextView Developer Documentation](#)

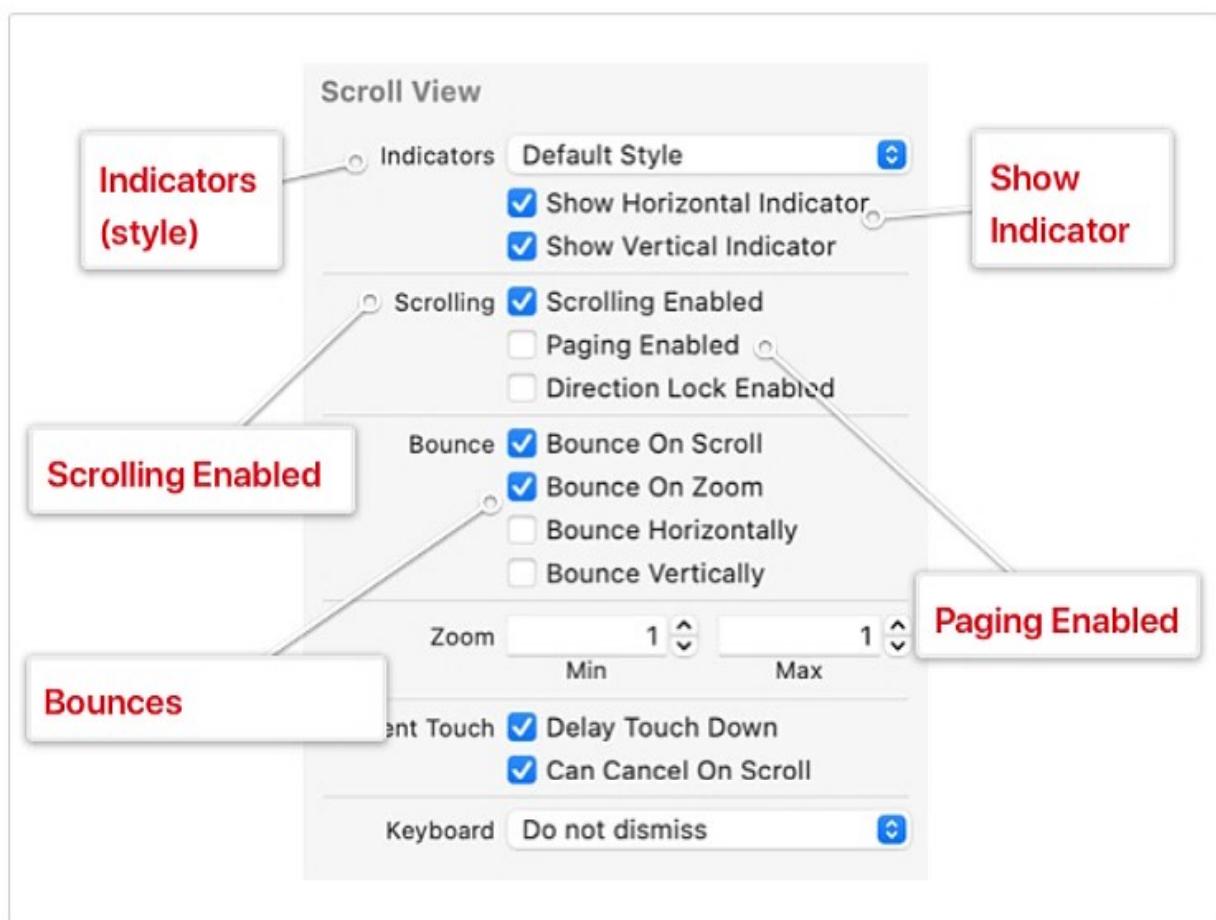
ScrollView (UIScrollView)



A scroll view allows the user to see content that runs beyond the boundaries of the view. You'll typically use a scroll view when the information you want to display is larger than the device's screen.

When the user interacts with a scroll view, a vertical or horizontal scroll indicator briefly appears to indicate there's more content to view.

Configuration



[UIScrollView Developer Documentation](#)

Table View (UITableView)

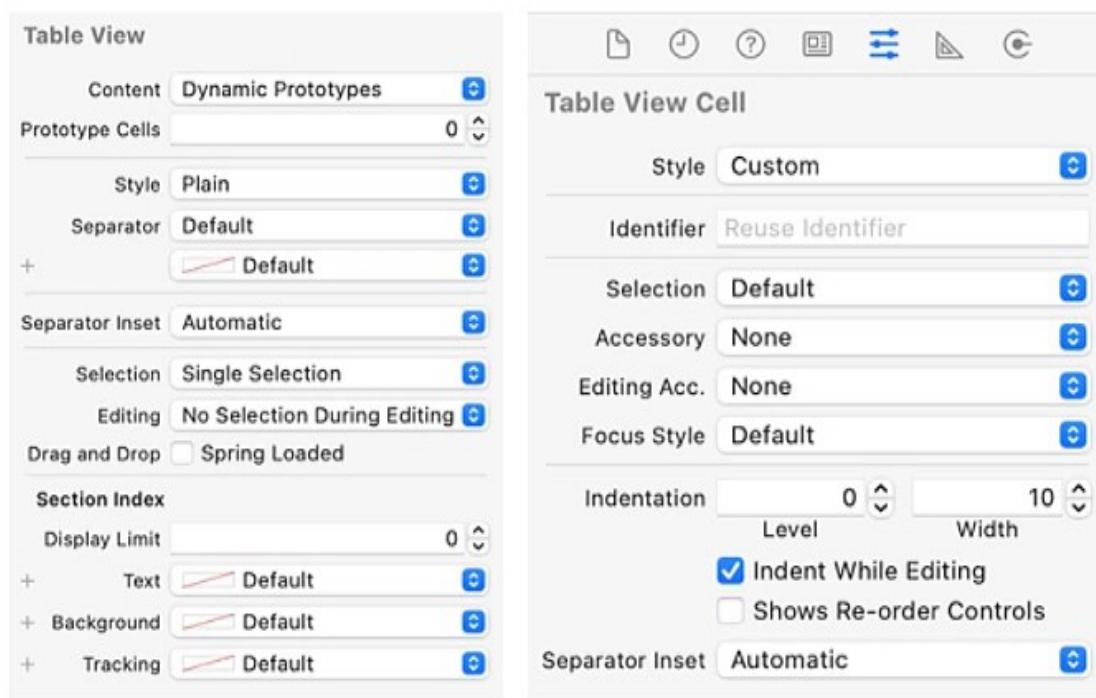


A table view presents data in a single scrollable column of rows and sections, allowing users to navigate easily through groups of information. Table views are an excellent format for displaying and editing hierarchical lists of information.

For example, the Mail app uses a table view to display email messages in the user's inbox, and the Messages app uses a table view to display message threads organized by contacts.

Configuration

Table views are interesting because you can customize the look and feel of the table view itself, as well as the look and feel of the rows (or cells) that it displays.



You'll learn more about table views and how to customize them in future lessons.

[UITableView Developer Documentation](#)

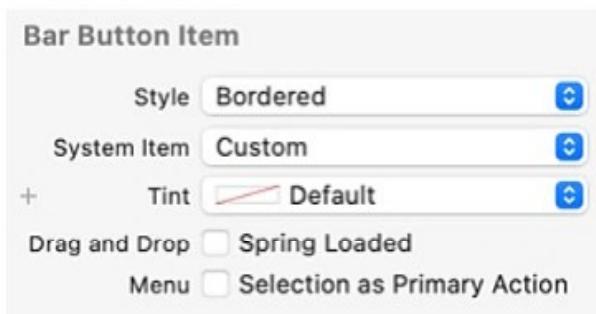
Toolbars (UIToolbar)

A toolbar usually appears at the bottom of a screen and displays one or more buttons, called bar button items. Users can select a button, or tool, to perform an action within a given view.



Configuration

You can configure the toolbar itself, or the bar items it displays. Each bar item consists of a title and an image, and can be enabled or disabled programmatically.



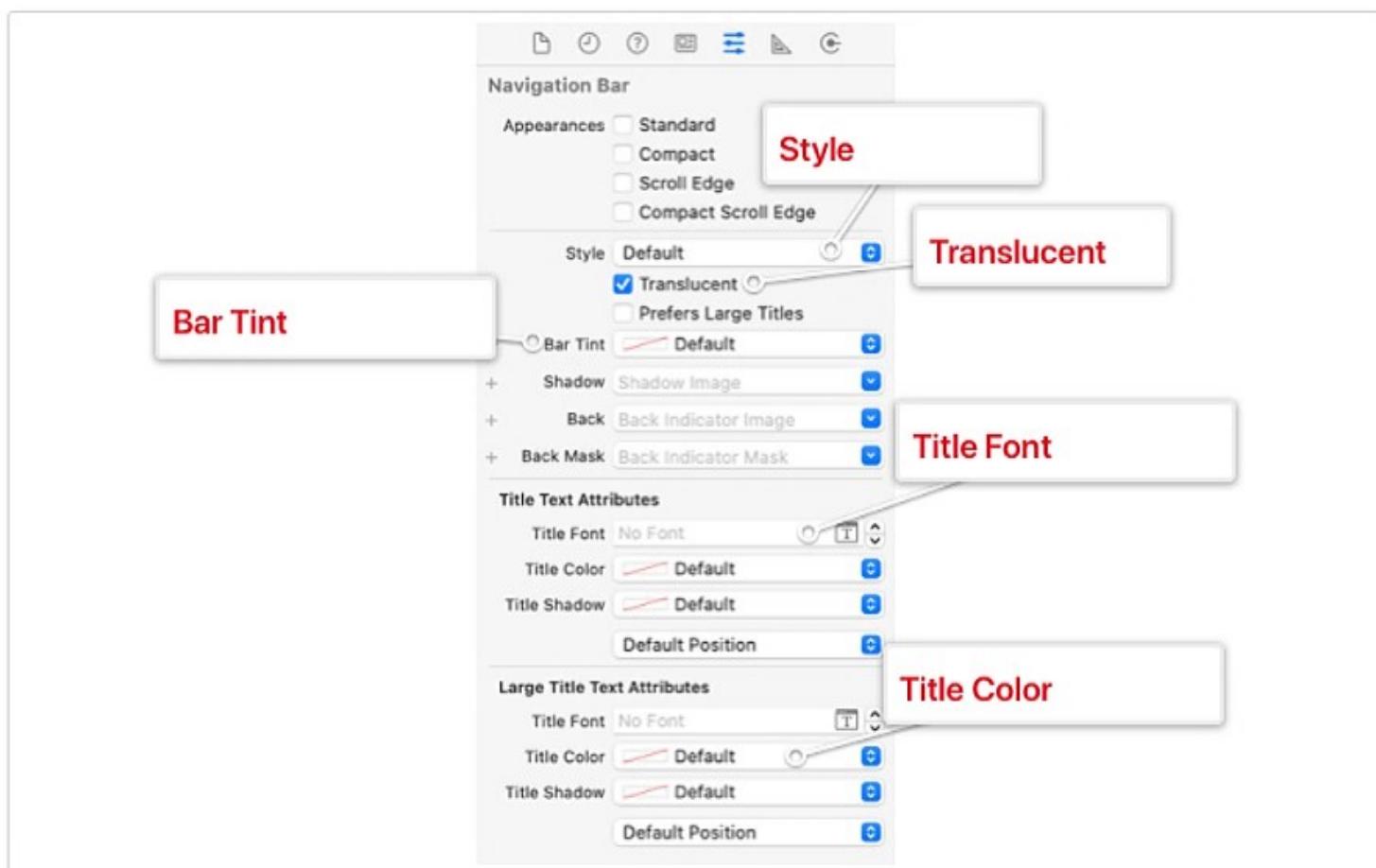
[UIToolBar Developer Documentation](#)

Navigation Bars (UINavigationBar)

You'll use the navigation bar to present your app's primary content in an organized way, one that you hope will be intuitive to the user. Navigation bars are typically displayed at the top of the screen, with bar buttons for navigating through a hierarchy of screens. They'll most likely include a title and a back button.

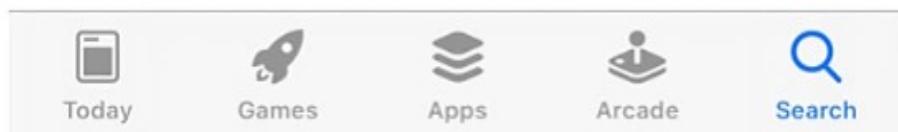
◀ Settings Sounds & Haptics

Configuration



[UINavigationBar Developer Documentation](#)

Tab Bars (UITabBar)



A tab bar provides easy access to different views in an app. It displays multiple tab bar items, each made up of an icon image and text. You'll use tab bars in your app to organize information by a specific feature or task. The most common way to use a tab bar is with a tab bar controller, which holds a property of each view controller that represents each scene you want presented in the tab bar. When the user taps an item, a new view related to the new task is displayed.

Tab bars are frequently used in apps that present multiple workflows or courses of action. For example, the App Store app has separate tabs to browse featured apps, search for a specific app, or check for available updates.

Configuration

Because you'll most often use the tab bar in conjunction with a tab bar controller, you'll add scenes to be displayed to the controller. The view controller for each scene has a `UITabBarItem` property that defines the text and optional image that will be displayed by the tab bar. To add a scene to a tab bar controller, and the tab bar view, you link it to the tab bar controller's `viewControllers` property in a storyboard.

[UITabBar Developer Documentation](#)

Controls

In the previous section, you learned about common views that display information to the user. What about responding to user input? You'll use tools in `UIKit`, known as controls, to tell the app what to do.

Think of a control as a communication tool between the user and the app. When the user interacts with a control, the control triggers a control event. Different controls trigger different control events.

After setting up a control in Interface Builder, you set up an `@IBAction` that responds to a specific control event and allows you to execute a block of code. Most often you will use the Primary Action Triggered (`UIControl.Event.primaryActionTriggered`) control event. This control event is triggered when a button is tapped or when the value of a control changes.

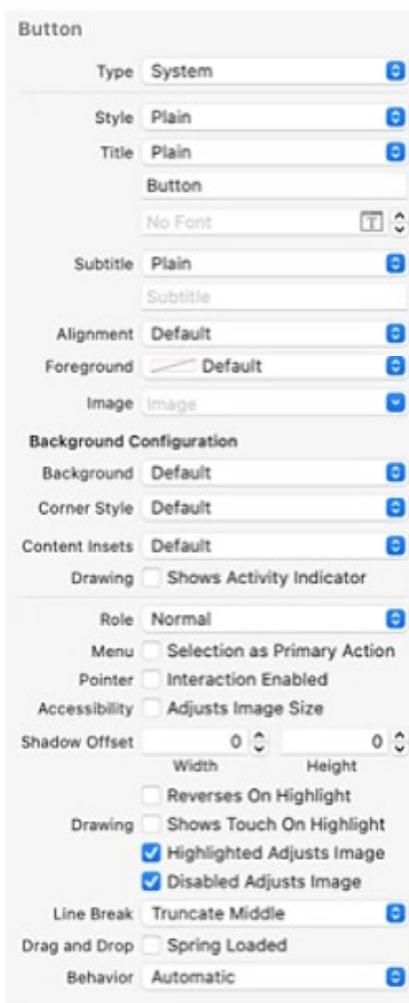
Controls are simple, straightforward, and familiar to users because they appear throughout most iOS apps. As you learn about some of the different subclasses of `UIControl`, on the following pages, think about your favorite apps and how you use controls to interact with them.

Buttons (UIButton)

Button

Users of iOS devices can initiate a control event with the tap of a button. When you set up a button, you give it a title or an image that conveys what the button will do when tapped. The appearance of the button changes with different states of being tapped: tapping down and lifting up.

Configuration



The primary control event is triggered when the user releases a button after tapping it. Buttons can also execute code at different stages of a tap, such as when the user first touches the button, holds down the button, or cancels the tap by dragging their finger outside of the frame of the button before lifting their finger.

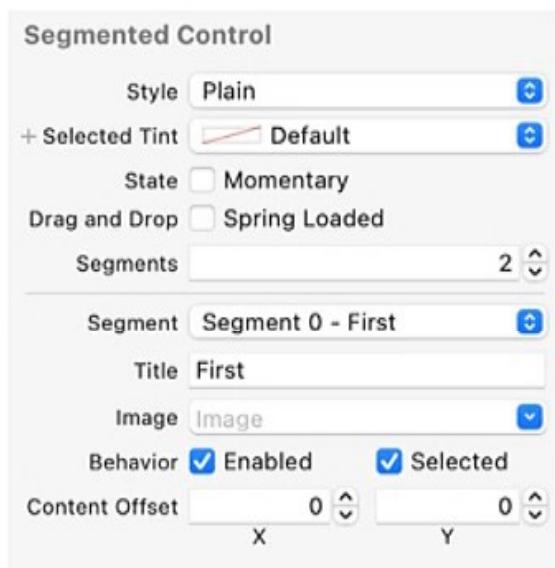
[UIButton Developer Documentation](#)

Segmented Controls (UISegmentedControl)



A segmented control is a horizontal set of multiple segments. Each segment functions as a discrete button, allowing the user to choose from a limited, compact set of options. This example, from Maps, allows the user to change the display mode, with three choices: Map, Transit, or Satellite.

Configuration



Segmented controls execute code when the control's value changes. The value represents which segment of the control is selected.

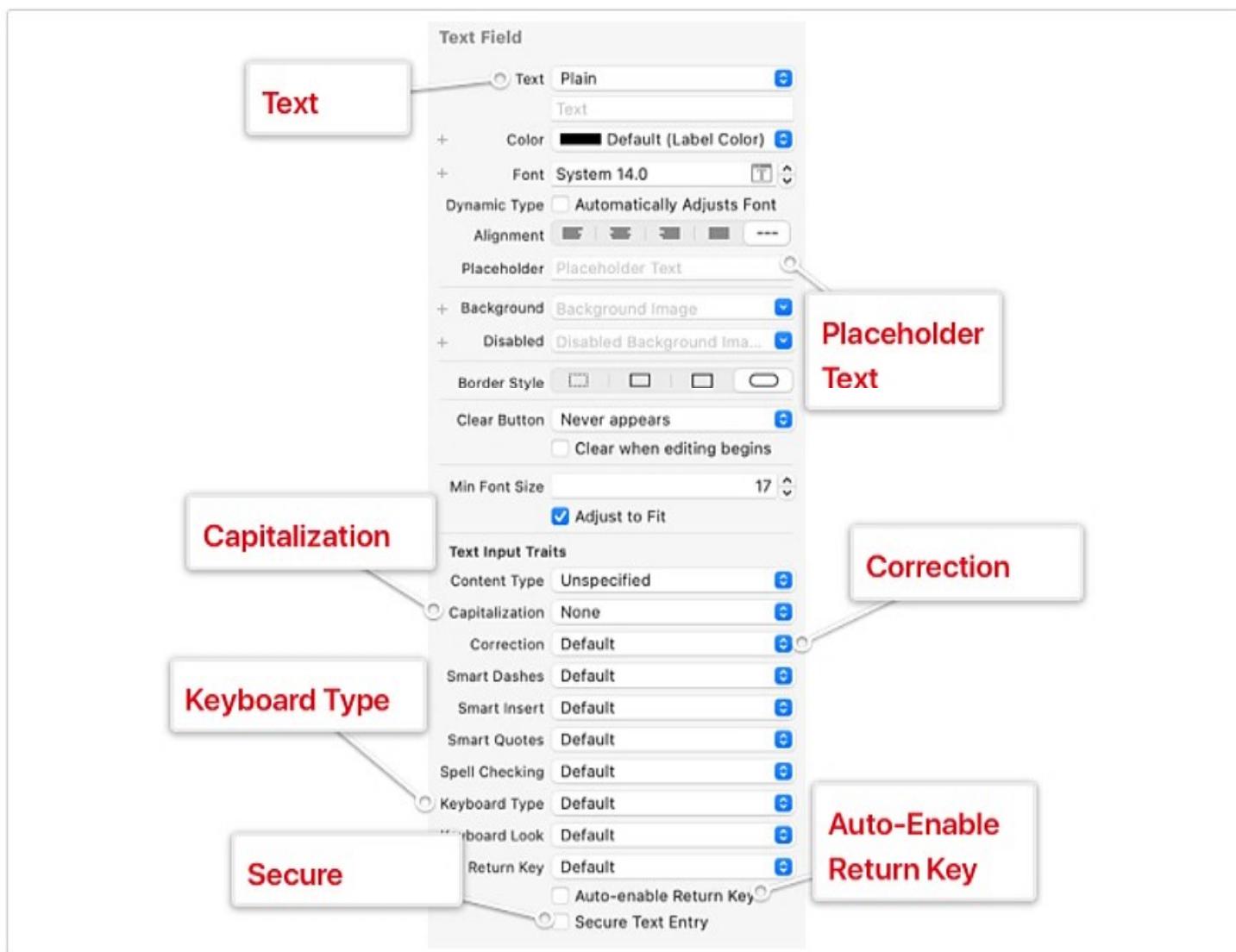
[UISegmentedControl Developer Documentation](#)

Text Fields (UITextField)

Text fields allow the user to input a single line of text into an app. You'll use them to gather a small amount of text and to perform some action based on that text.

Text Field

Configuration



Text fields execute code when the user presses the 'Return' or 'Done' key on the keyboard or when the user edits the text.

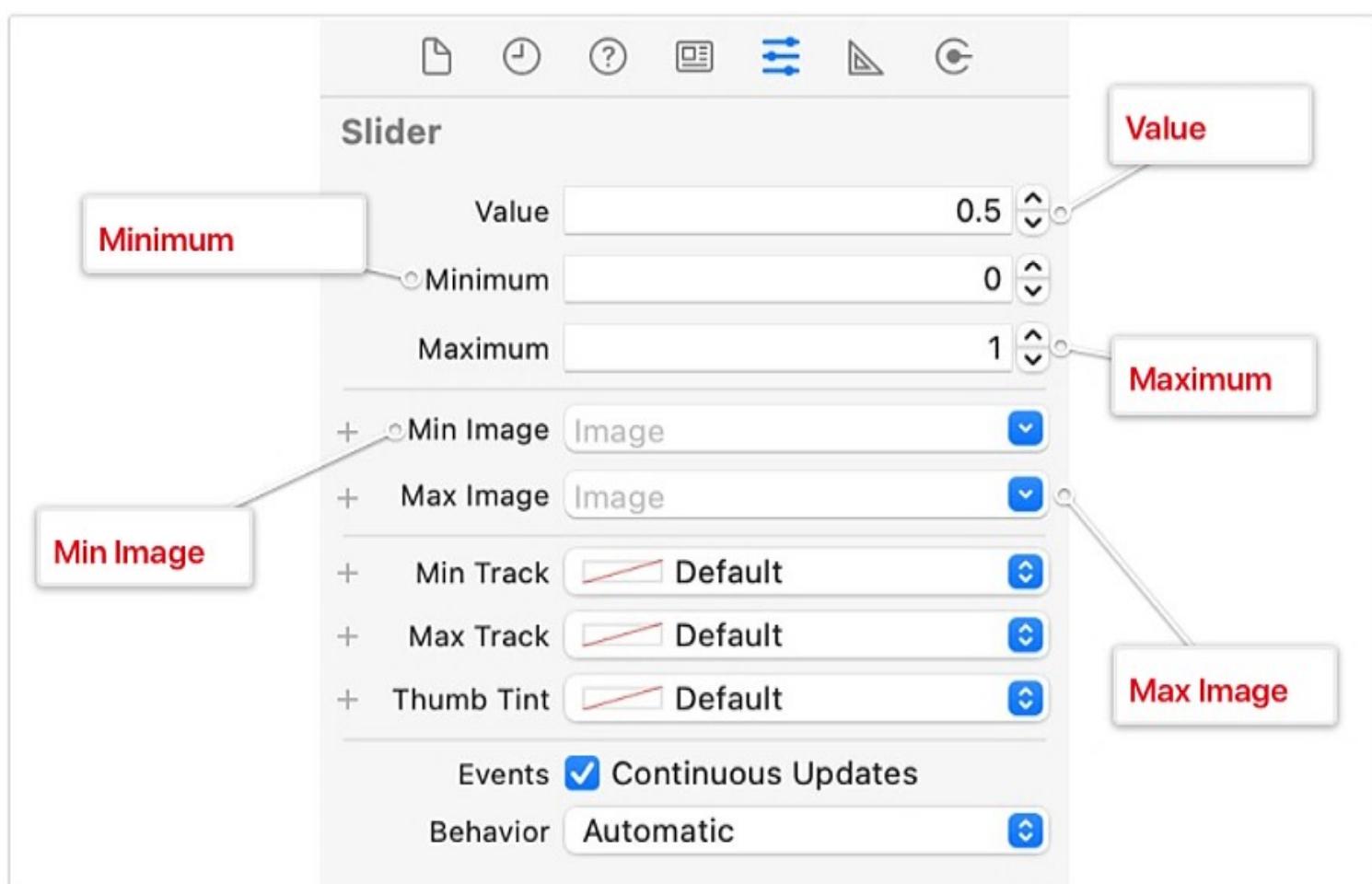
[UITextField Developer Documentation](#)

Sliders (UISlider)



Sliders allow users to make smooth and gradual adjustments to a value—useful for controls that modify things like speaker volume, screen brightness, or color values. The user controls a slider by moving from a starting value along a continuous range between minimum and maximum values.

Configuration



Sliders execute code as the user changes the value of the slider.

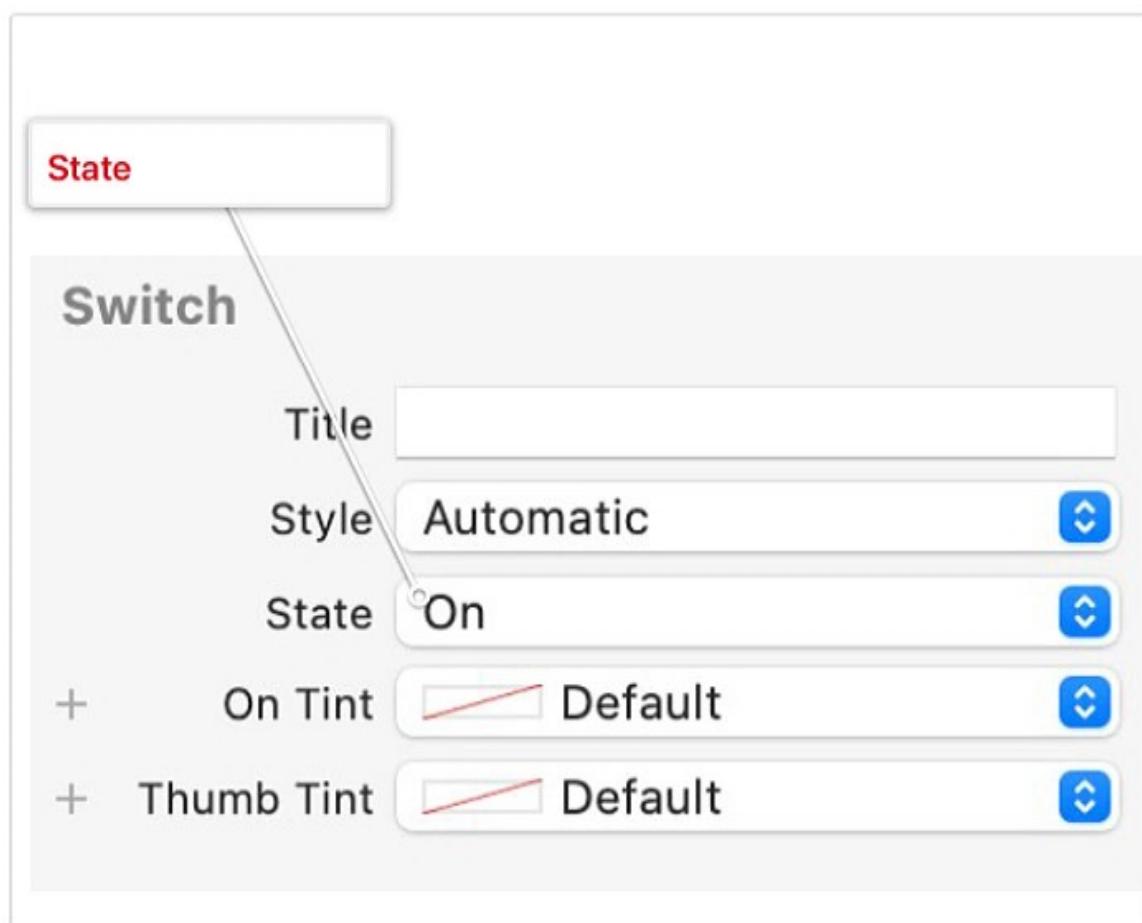
[UISlider Developer Documentation](#)

Switches (UISwitch)



A switch lets the user turn an option on or off. You've probably noticed—and used—switches throughout the Settings app.

Configuration



Switches execute code as the user toggles the control.

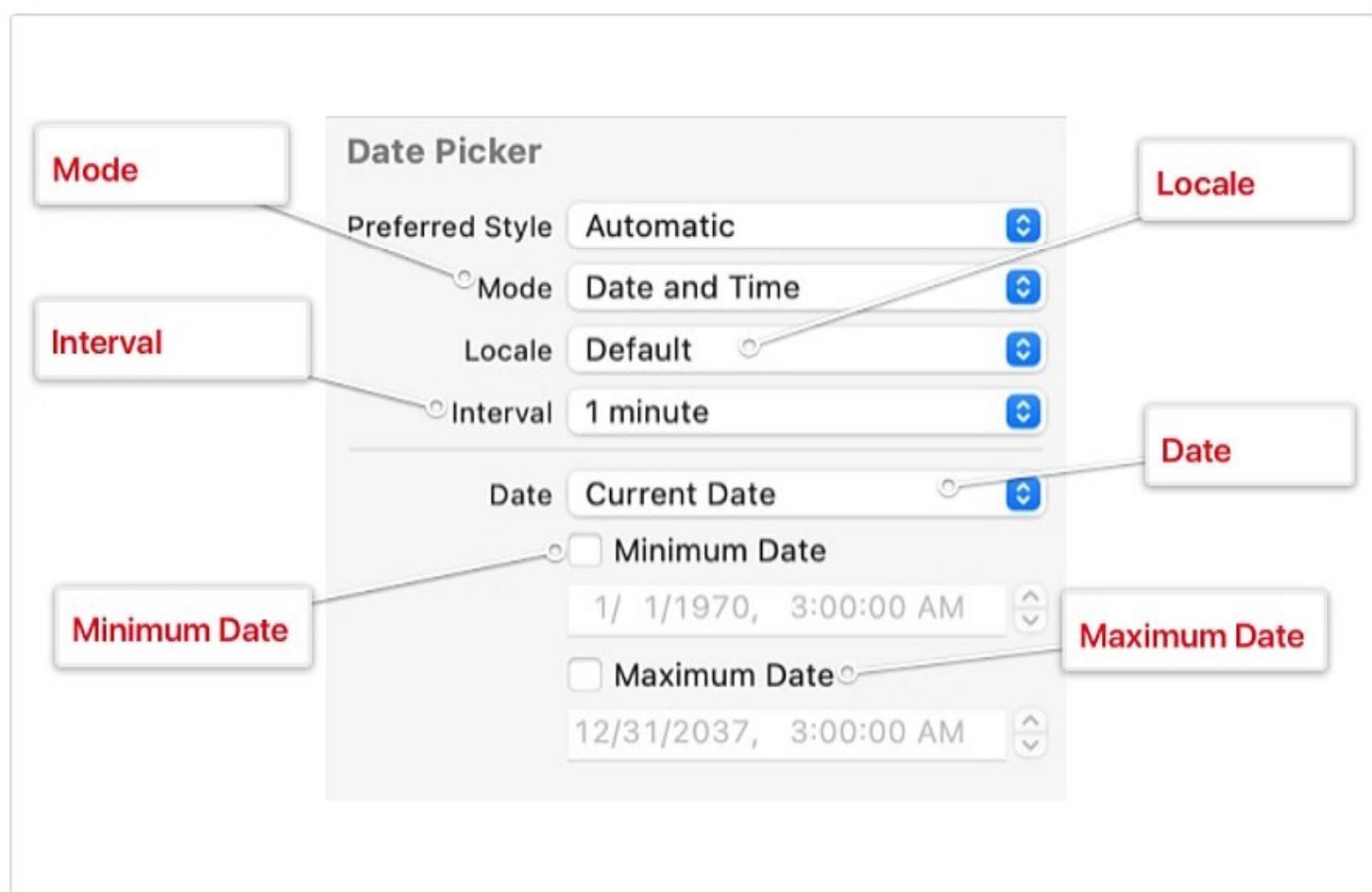
[UISwitch Developer Documentation](#)

Date Pickers (UIDatePicker)

Sun Nov 24	11	28	
Mon Nov 25	12	29	AM
Today	1	30	PM
Wed Nov 27	2	31	
Thu Nov 28	3	32	

Date pickers provide a straightforward interface for managing date and time selection, allowing users to specify a particular date quickly and efficiently.

Configuration



Date pickers execute code whenever the value of the selected date or time is changed.

[UIDatePicker Developer Documentation](#)

View Controllers

UIKit defines a special class that controls a view, sets up child views, controls what they display, and responds to user interaction. This class is called the `UIViewController`.

Most commonly, each screen in an app is represented by a scene in a storyboard, and each scene in a storyboard is associated with a subclass of `UIViewController`. The associated `UIViewController` subclass is defined in a `.swift` file that holds all of the logic that controls the scene. Each `UIViewController` class has a `view` property that represents the parent view of the scene.

Think back to the Light project you built in the first unit. The project has a single storyboard with a single scene. That scene was linked to a `UIViewController` subclass called `ViewController`. The light was toggled by adjusting the `view` property of `ViewController`. That `view` property is the same instance of `UIView` as the parent view of the scene in the storyboard.

When you added a button to the screen, the button became a child view of the scene's main view. When you wired up actions and outlets, you linked them to the `ViewController` file, which defined the view controller for that scene.

You'll learn how to make complex view controllers in later lessons. But for now, it's enough to understand that each different type of screen you see in an app is managed by a different type of view controller.

Where To Learn More

You've just learned about the most common views and controls in UIKit. But where would you go to find out more about these and other UIKit tools? As you learned earlier, Apple has large teams dedicated to writing documentation to help developers like you learn more about the system tools for building apps.

Most of the information for this section is from a set of guidelines called the [Human Interface Guidelines](#), which was written and maintained by one of those teams. When you want to learn more about UIKit, you can go to developer.apple.com and search for more information about the topics in this lesson.

Lab—Uikit Survey

The objective of this lab is to identify different views and controls in some of the most common system apps on iOS. You'll use Simulator to look through Settings, Contacts, News, and Calendar, and then use Pages to complete a survey of your findings.

Step 1

Create A Pages File For Your Survey

- Create a new Pages document titled "2.7 UIKit Survey," and save it to your project folder.
- Add section headers for six views: label, image view, text view, table view, navigation bar, and tab bar.
- Add section headers for five controls: button, segmented control, text field, slider, and switch.

As you go through the following steps, one app at a time, you'll add screenshots to the sections you just created. For each app, you don't need find *all* the views and controls, but make sure that each section in your survey includes at least one screenshot.

Step 2

Survey The Settings App

- Open the Simulator application and click Settings.
- Navigate through the Settings app to find examples of at least five views and at least three controls.
- For each example, take a screenshot using the **Command-S** keyboard shortcut, and add the screenshot to your document below the correct heading. Screenshots are saved to your Desktop.
- If you come across an unfamiliar view or control, use the [Human Interface Guidelines](#) to identify it and add it to your document under a new section heading.

Step 3

Repeat For Contacts, News, And Calendar

- Repeat the search for Contacts, News, and Calendar.

Step 4

Review The Examples

- Review the document with a partner or with someone who isn't familiar with iOS development.

Connect To Design

Open your App Design Workbook, and review the Map section for your app. Although you are only just beginning to map out your prototype screens, you can start to think about what the different views will look like in your app and how you might control them using UIKit. Add comments to the Map section or in a new blank slide at the end of the document. What common views, such as a tab or navigation bar, might your app use to display information to the user? Are there any common controls, like buttons or text fields, that could control how your app responds to user input?

In the workbook's Go Green app example, a table view could show a scrolling view of the date, the amount of trash and recycling, and the log of items for that day. The user might expect each date to function as a button to control which information they are viewing.

Review Questions

Question 1 of 5

What is the main distinction between a view and a control?

- A. Views respond to user interaction; controls display information.
- B. Views display information; controls respond to user interaction.

[Check Answer](#)



Lesson 2.9

Displaying Data



In the last lesson, you learned about common views and controls. Now you can put that knowledge to use.

In this lesson, you'll use Interface Builder to create the beginnings of an app, called Hello, that you can use to introduce yourself. You'll plan out the content for the app, and begin to build it by adding labels and a profile image.

What You'll Learn

- How to configure views using Interface Builder
 - How to customize a label
 - How to customize an image view
-

Vocabulary

- aspect ratio
 - clipping
 - content mode
 - dynamic data
 - frame
 - static data
-

Related Resources

- API Reference: [UILabel](#)
- API Reference: [UIImageView](#)

In this lesson, you'll plan out and build a simple app called Hello, an app that you could use to introduce yourself. As you work, you'll apply what you've learned about some of the common views in UIKit.

Planning The App

Most apps start with a simple idea, which can usually be summed up in one short sentence. Think of apps that you use every day. They may have started with the following straightforward ideas:

- “Send and receive messages.”
- “Share photos with my friends and family.”
- “Post short messages, and see the short messages that others have written.”

As an app developer, you'll learn how to take a one-sentence overview of an app or a feature and plan out how to make it a reality.

In this project, your task is to build an app for introducing yourself. This guide will give you a framework for what to include, but the app will introduce *you*. Bring in your own ideas, and make it yours.

Think about how you would introduce yourself to someone you haven't met before. What information would you want to make sure someone knew about you? Start by writing down your name, then take a few minutes to write down what you might share about yourself. Use the following prompts to help you come up with enough information for your app.

- Who are you? What is your life all about?
- What does a day in your life look like?
- If you had all the time and money you needed, what would you do with your life?
- What are some of your favorite things or activities?

Include any other information you think other people would want to know.

Sometimes the information you want to display in an app is consistent and never changes. In programming, this information is called static data. As you plan your app, you're brainstorming static data about yourself. In future apps, you'll work with data that changes, which is called dynamic data.

Example

About Me

My name is Cynthia Yao. I'm a dedicated student, I play lacrosse, I love animals, and I'm looking for an internship in Marine Biology.

Location: Half Moon Bay, California

Website: www.example.com

Day in the Life

Eat breakfast

Go to school

Go to lacrosse practice

Eat dinner

Hang out with friends

Watch TV

When I Grow Up

I want to sail around the world, spend time in new countries, learn new languages, and meet awesome people everywhere I go.

Favorites

Food: Ice cream

Sport: Lacrosse

Book: A Tale of Two Cities

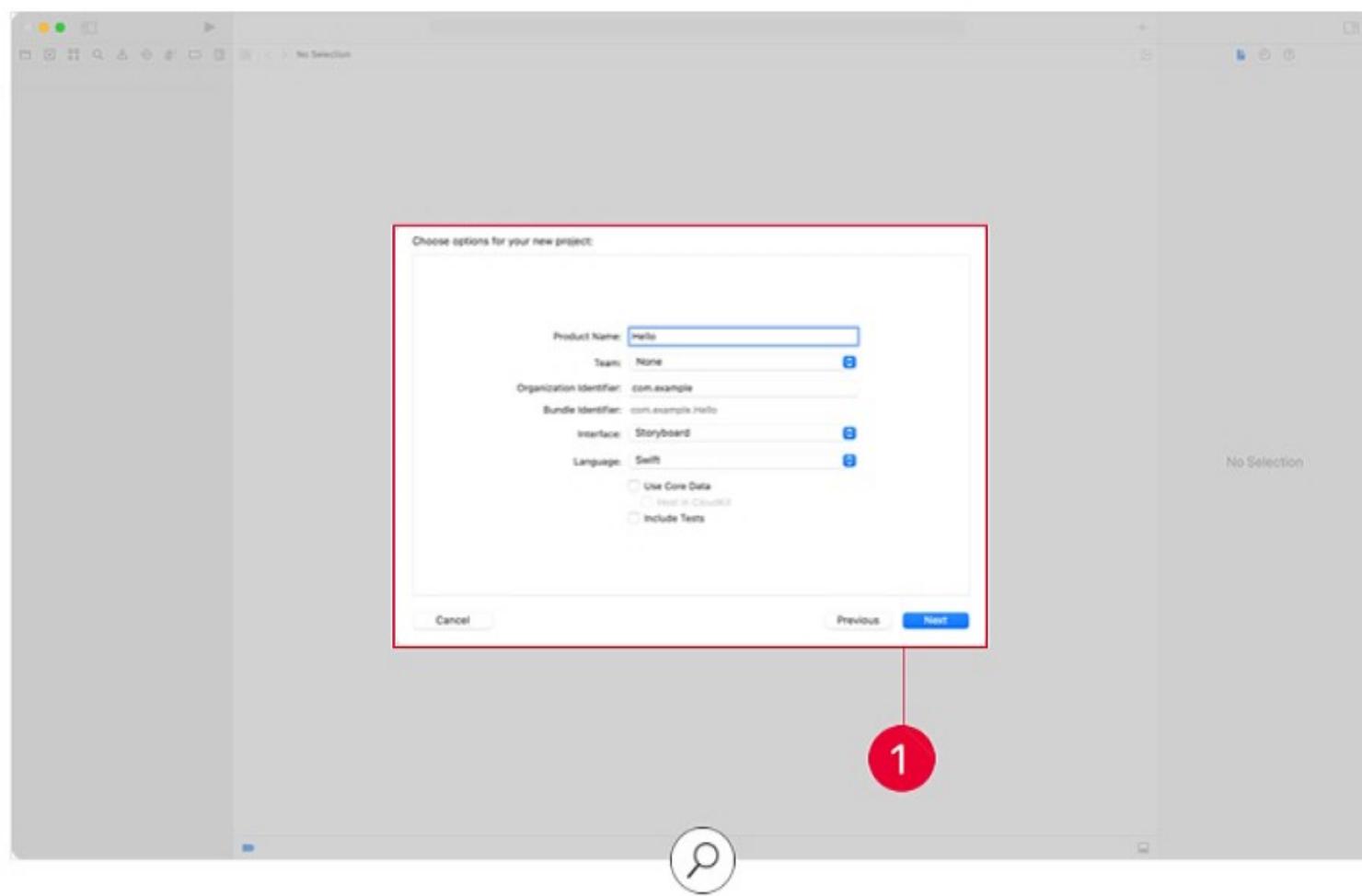
As you come up with information that's unique to you, you can probably imagine many ways to display it and share it with others. For this project, try to think about how you might display this information on a single screen.

Since you're still in the planning phase, you might find it helpful to open Pages or Keynote and create a rough outline of the data you want your app to display.

You'll probably discover that all the information you want to include won't fit on a single iOS device screen—which means you'll need to enable scrolling. In this lesson, you'll create part one of the project, adding your name, a profile image, and a few sentences about yourself. In a future lesson, you'll complete part two, using more advanced layout features in Interface Builder to add images and labels in a scroll view.

Create The Project

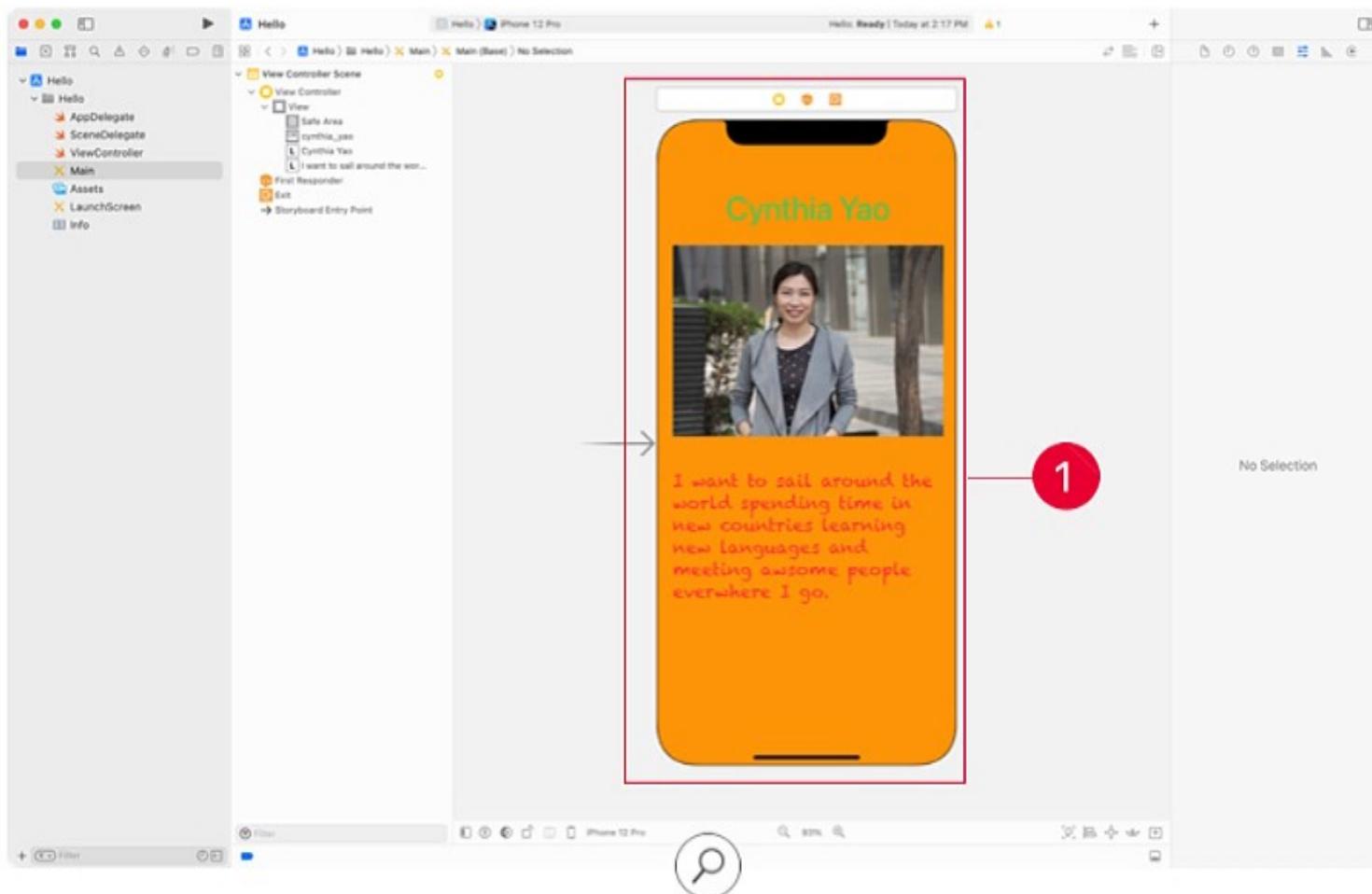
Create a new Xcode project using the iOS App template. When creating the project, make sure the interface option is set to Storyboard. Name the project "Hello" and save it to your project folder. ①



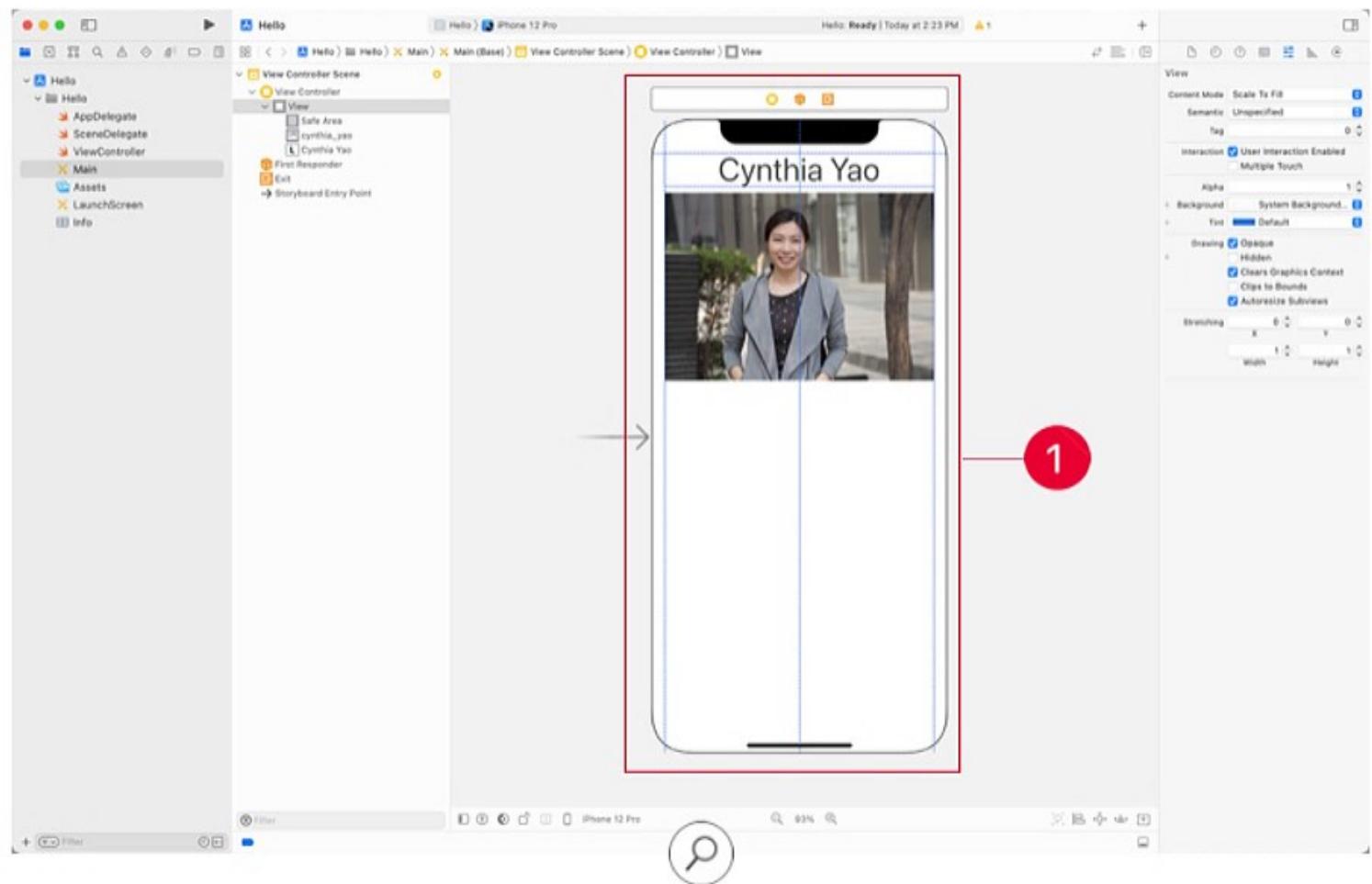
Enter Your Information

Open the **Main** storyboard. You'll use the initial scene to add your information. Because your app will always display the same data, you can build your static data directly into your screens using labels or images in Interface Builder. In future lessons, you'll learn to write code for displaying dynamic data.

As you choose fonts, colors, and other display settings, do your best to keep a clean-looking presentation that fits on an iOS device. Avoid unreadable fonts, conflicting color schemes, and confusing text layouts. ①



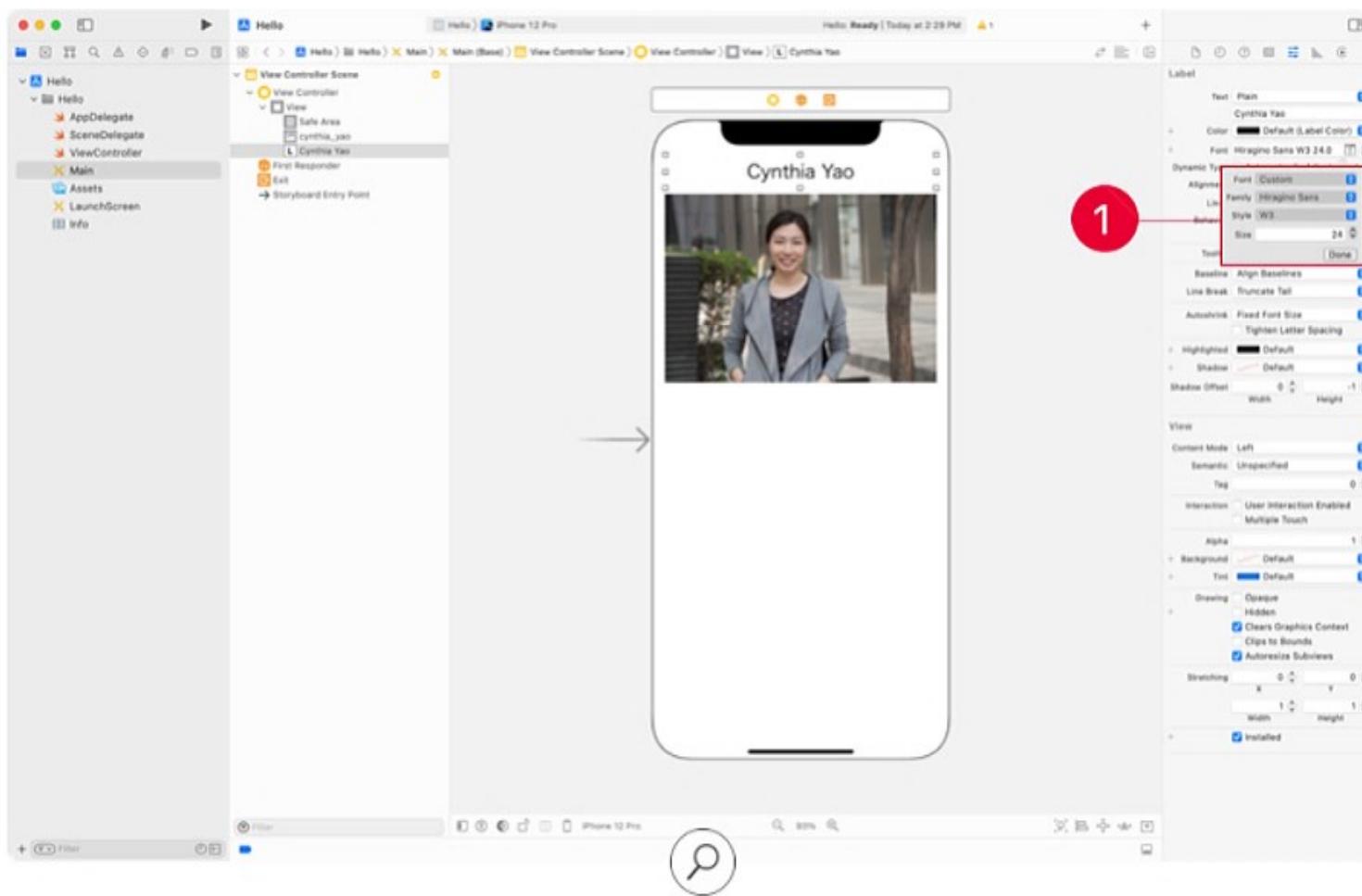
As you place views, take advantage of the layout guides to center your content and be careful to observe margins and allow sufficient spacing between objects. You'll also want to avoid putting content in the status bar at the top of the screen. ①



Add Labels for Displaying Text

For each of the text items you want to include, drag a Label object from the Library to the scene. At a minimum, add labels for your name and one or two sentences about yourself. You can add the rest when you finish this project in a future lesson.

Selecting one label at a time in the storyboard, use the Attributes inspector to set the label's text and choose an appropriate font. If you want a label to display as a header, make the font bigger or bolder than the default text. Do you want to use a custom font? Open the Font pop-up menu and choose Custom to select from all the fonts installed on your Mac. ①



Experiment with different attributes in the inspector to see how each of them modifies the appearance of your labels. For example, you can change the alignment of the text or the number of lines you want the label to display. By default, labels display only one line of text, which may cut off any remaining text at the end of the line.

Cynthia is a dedicated student looking for...

When working with static information, you can keep increasing the number of lines until you see that the label can display all your text. But there's a better way that allows the text to flow to as many lines as it needs to display the entire text you provide.

Look at the documentation for `UILabel` and find the `numberOfLines` symbol. In the Description section, you'll see the following text:

This property controls the maximum number of lines to use in order to fit the label's text into its bounding rectangle. The default value for this property is 1. To remove any maximum limit, and use as many lines as needed, set the value of this property to 0.

The `numberOfLines` property is useful when you want a label to display text that flows beyond a single line. Feel free to explore the documentation for `UILabel` to discover other strategies for fitting text into a label.

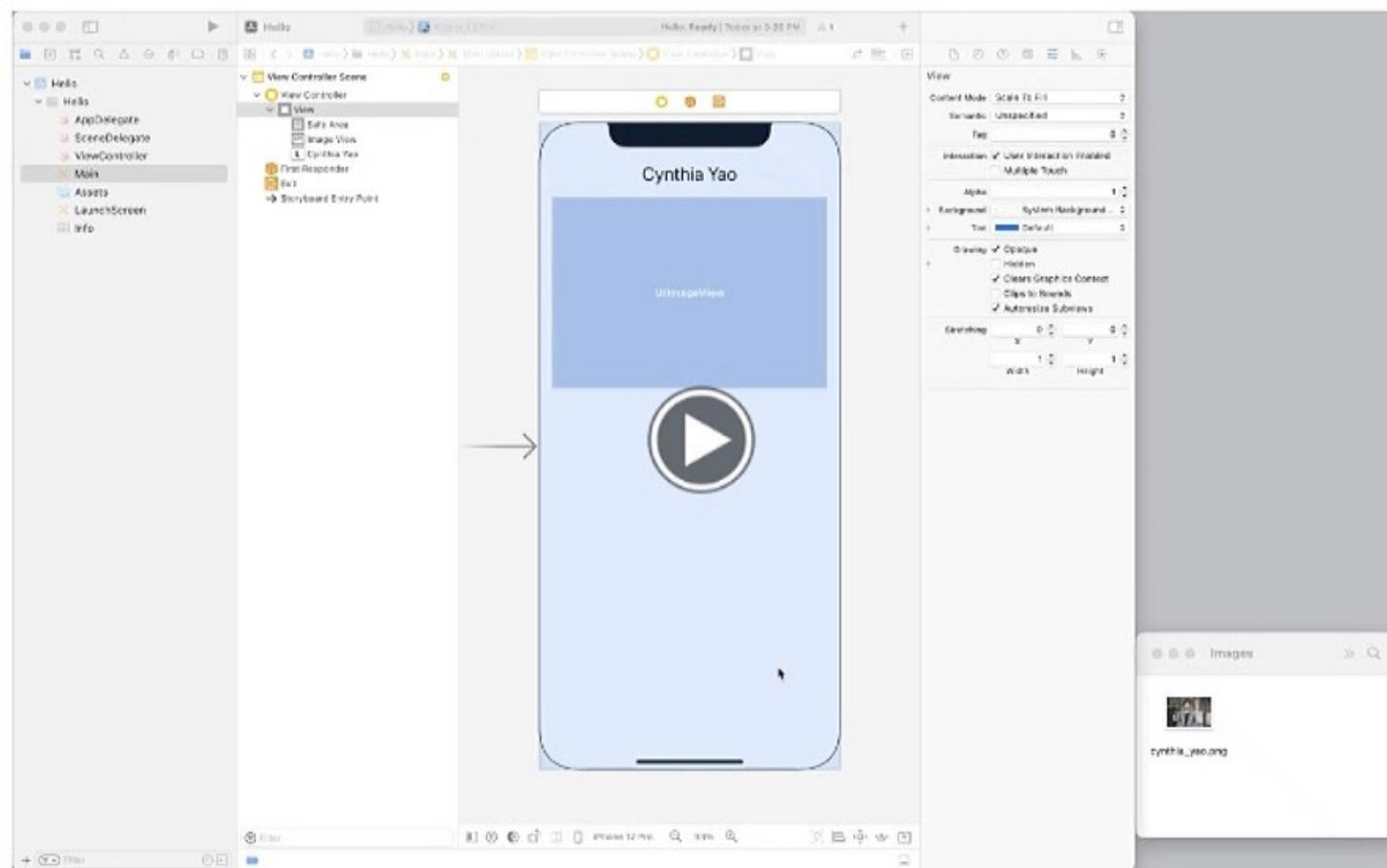
Finish up adding text to all the labels you want to include in your Hello app.

Add an Image

Your introduction wouldn't be complete without a photograph of yourself. If you don't already have an image you like, you can use the Photo Booth app to take a selfie. Make sure you have the image file handy on the desktop or in the Finder. If you have more images, you'll have a chance to add them to the app in a future lesson.

Now go ahead and add the image file to your scene so that the app will display it.

Here's how to add an image to your Xcode project:



1. *Open the Navigator area.* Make sure the Navigator area on the left is open and that the Project navigator shows a list of your files. If you're working on a small screen, you may want to close the Inspector area on the right.
2. *Select Assets.* Xcode has automatically added this Asset Catalog to your project. This is the best place to keep the images you'll use in your app.
3. *Drag in the image file.* Add the image to your storyboard and give it a name in the Document Outline. The image is now part of your app.

Now that you've added an image of yourself, you can set up an image view in Interface Builder. Go back to the **Main** storyboard. From the Library, drag an Image View object to the scene.

Using the Attributes inspector, choose the name of your image from the Image pop-up menu. This will assign that image to the image view you just created.

In the previous lesson, you learned that an `image view` is like a frame around the *image* it will display. The size of the `image view` determines the total potential display size of the image. By setting the content mode of the `image view`, you can control if the image will display within the frame, potentially leaving empty white space, if it will stretch to fit within the frame, potentially distorting the image, or if it will stretch to fill the entire frame, potentially cropping some of the image.

In the table below, you can see the most common `contentMode` options for a square `UIImageView` displaying a non-square image.

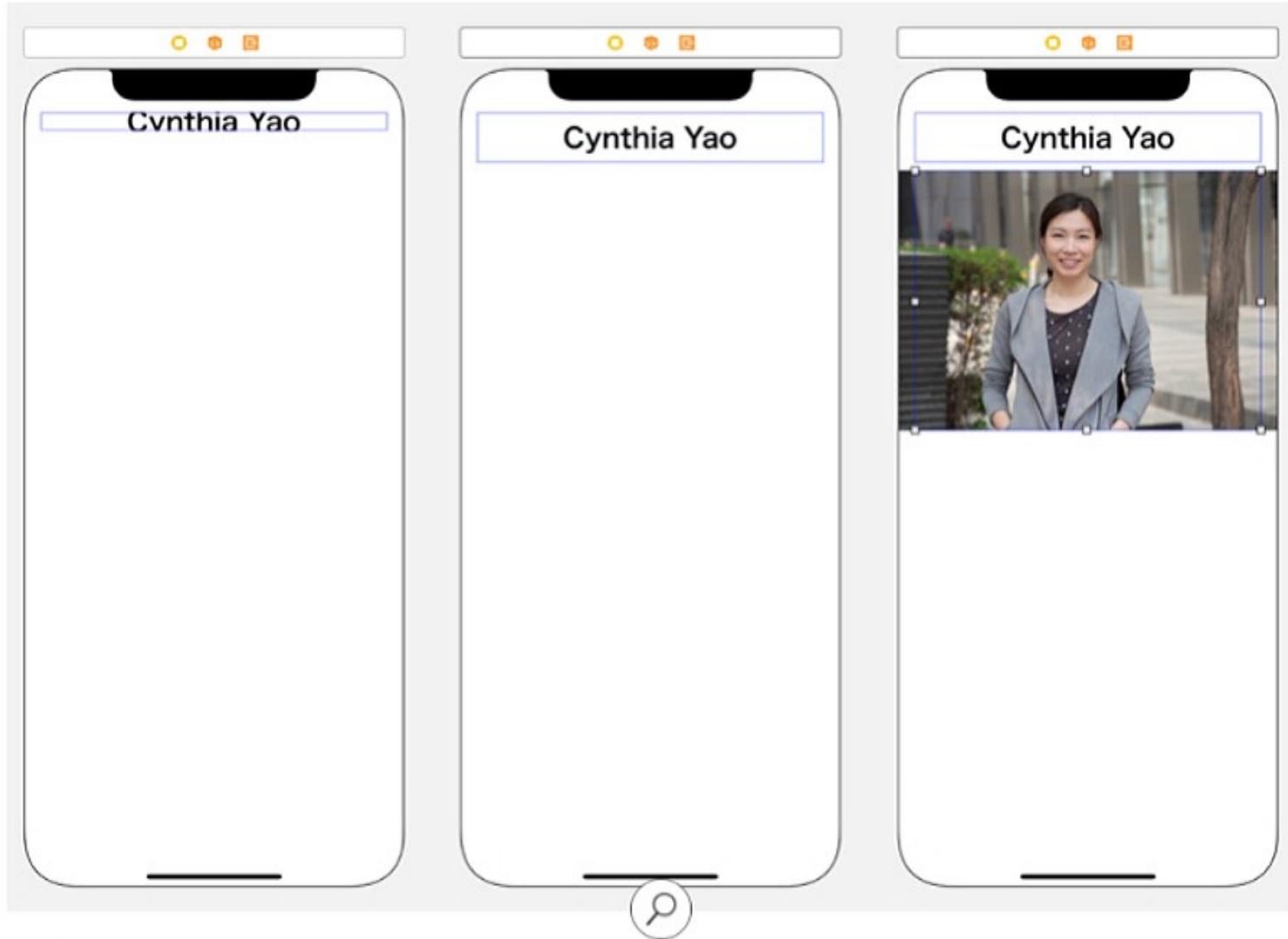
Scale to Fill	Aspect Fit	Aspect Fill
Resized to image view bounds	Maintains aspect ratio, resized to fit whole image into the image view bounds	Maintains aspect ratio, resized to fill the image view bounds, overflows the image view frame
		

Choose the correct content mode for the image you added to your project.

Address Layout Issues

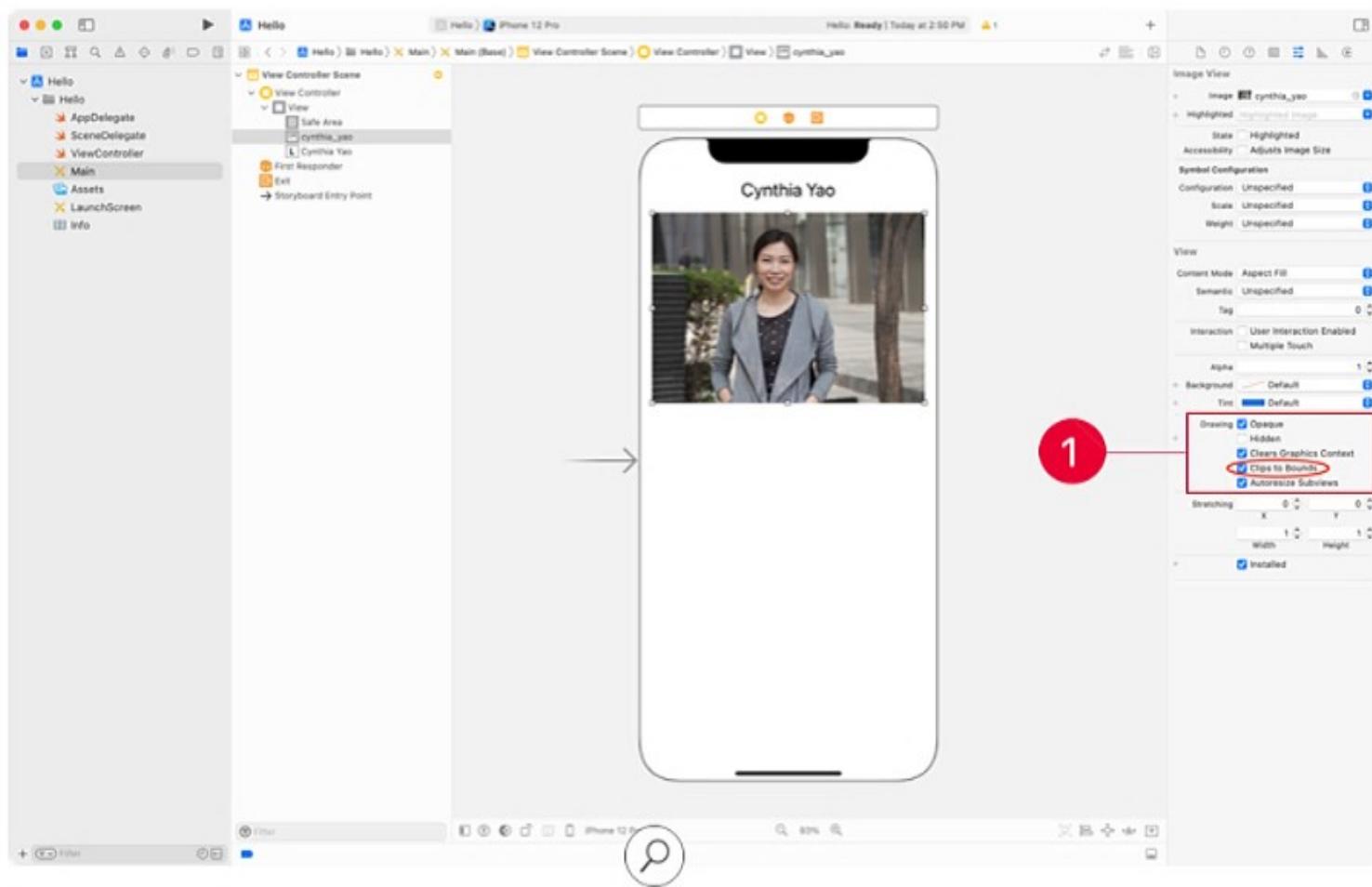
Unexpected Clipping

It's possible you've encountered occasional warnings or bugs with your layout. Don't worry about these right now. In a later lesson, you'll learn how to resolve them and how to use more advanced layout options, including Auto Layout, in Interface Builder.



- However, if you've run into unexpected clipping (a common issue), you can resolve it now. Just as an image view frames the image it's displaying, a label frames the text it displays. If a label's content is larger than the label itself, it results in clipped text.
- To fix clipped text, you have two choices: decrease the font size or increase the label size.
- Images can also have clipping issues. For example, if you have an image view with a different aspect ratio than the image it displays and you choose the Aspect Fill content mode, some of the image will overflow the bounds of the image view.

Interface Builder allows you to modify this behavior, or choose whether or not content outside the bounds of the view will be visible. ①



Placement Errors with Different Screen Sizes

When you learn about Auto Layout, you'll learn how to make your interface scale across various device sizes. But for now, make sure you set up your interface for the same device size that you're using in Simulator. This will ensure that you see a consistent view across your projects.

Showing More Content

You've now built a simple app with a few labels and one image in a single view. In the "Auto Layout and Stack Views" lesson, you'll learn how to use advanced layout tools to position views, and how to use a scroll view to add more content.

Lab—Tutorial Screen

The objective of this lab is to create a tutorial about a hobby you enjoy. You'll use Interface Builder to set up a view with relevant images and text.

Create a new project using the iOS "App" template and name it "Hobby Tutorial."

Step 1

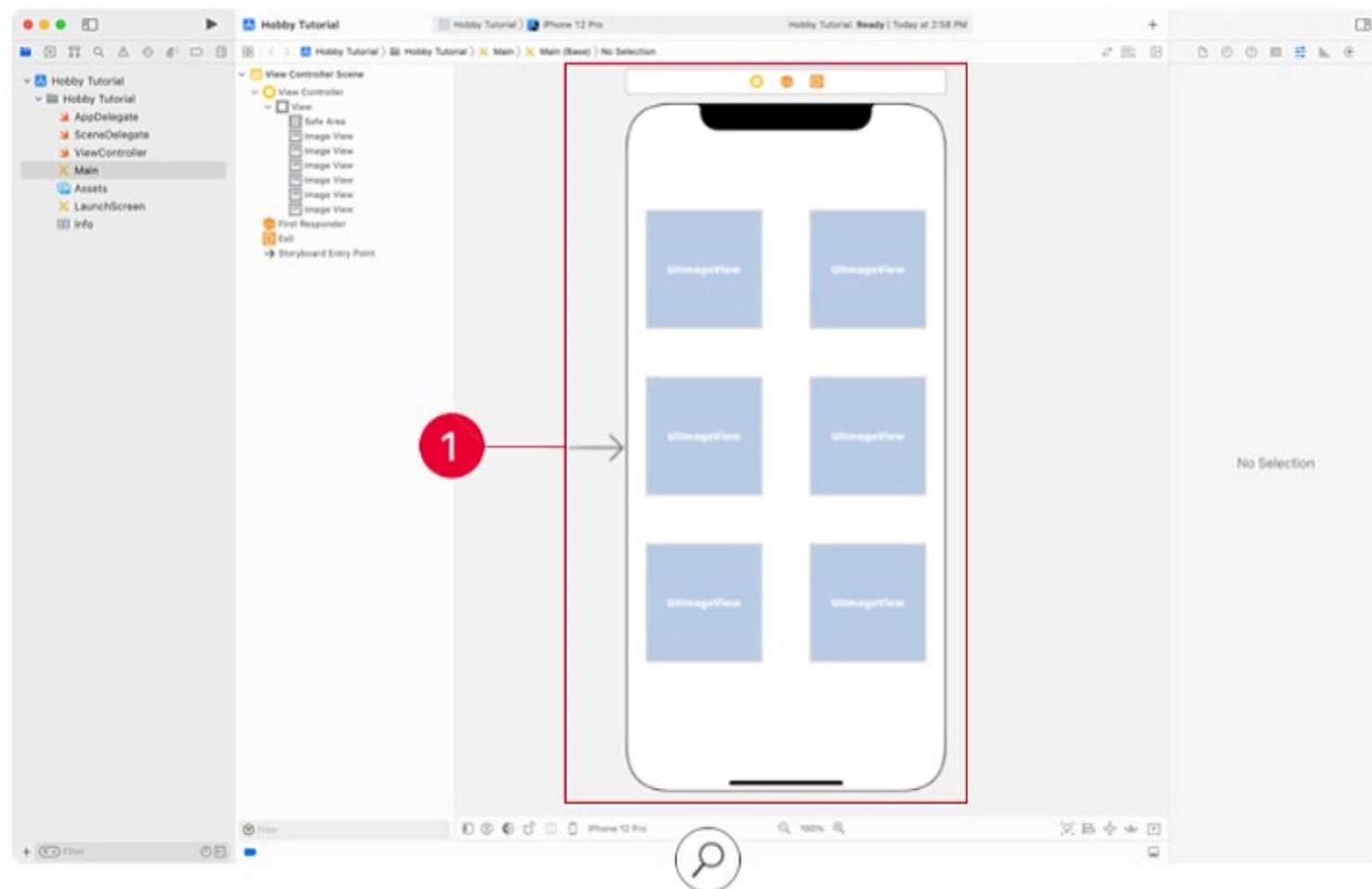
Create A Project And Add Images

- Choose three to six images that explain something about your hobby.
- Add the images to your project by dragging them into **Assets**.
- Assign appropriate names for each image in the Asset Catalog.

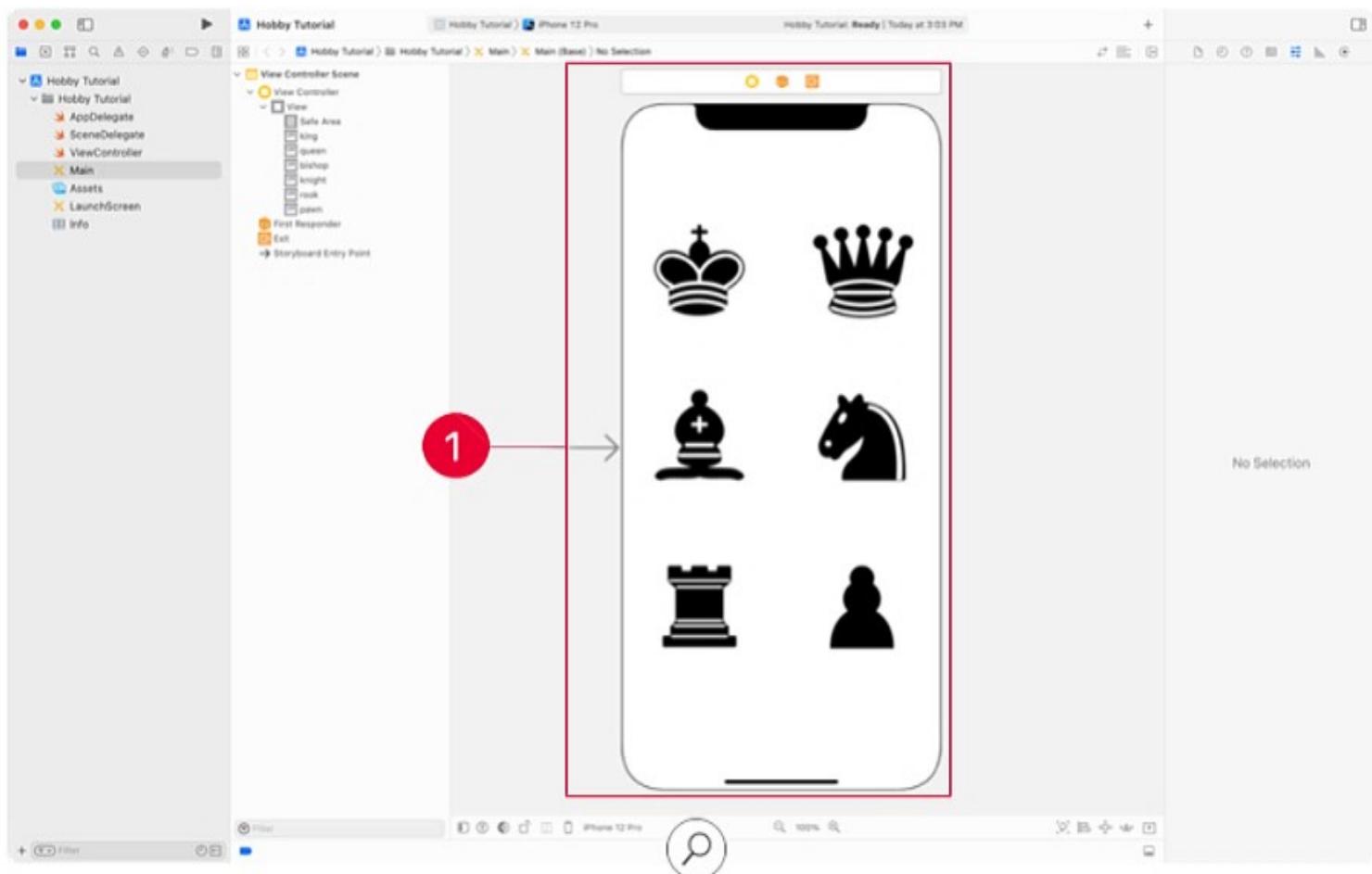
Step 2

Set Up The Image Views

- Add an image view in storyboard for each of the images. Use the layout guides to set up a clean, well-balanced interface. ^①



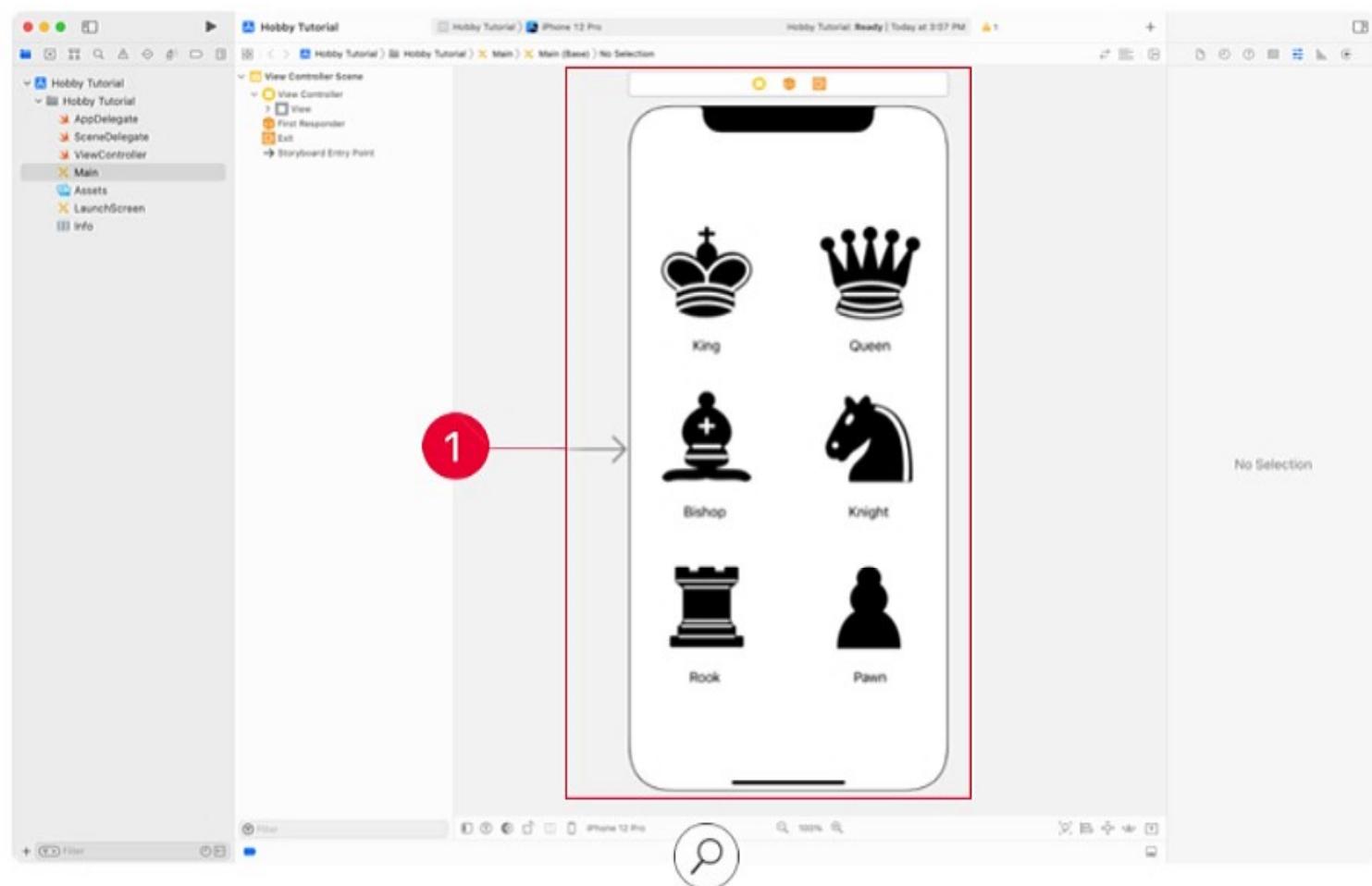
Assign the images to the image views using the Attributes inspector. ①



Step 3

Set Up The Labels

- For each image, add a label and enter a brief description of the image as the text.
Use an appropriate font.



Excellent work! Labels and image views are two of the most common views for displaying data in an app. Practice using labels and images in new ways to display different kinds of information. Be sure to save your project to your project folder.

Connect To Design

In your App Design Workbook, reflect on how you can bring the content of your app to life. You just got hands-on experience using Interface Builder. Will you need to use a label in your app? What about an image view? Make comments in the Map section or in a new blank slide at the end of the document.

In the workbook's Go Green app example, labels will be used on the home screen to provide information next to each icon or to give background information about trash use.

Lesson 2.10

Controls In Action

Two lessons ago, you learned about common views and controls. In the last lesson, you had a chance to practice creating labels and image views. Now you can take another step forward by setting up controls and responses to control events.

In this lesson, you'll use Interface Builder to add buttons, switches, and sliders to a scene. You'll also create actions and outlets, write some basic code, and gain an understanding of how these tools work together.

What You'll Learn

- How to use a button to execute code
 - How to use a switch and access its value
 - How to use a slider and access its value
 - How to use a text field and access its value
 - How to respond to user interactions with gesture recognizers
 - How to connect controls to actions programmatically
-

Vocabulary

- [gesture recognizer](#)
-

Related Resources

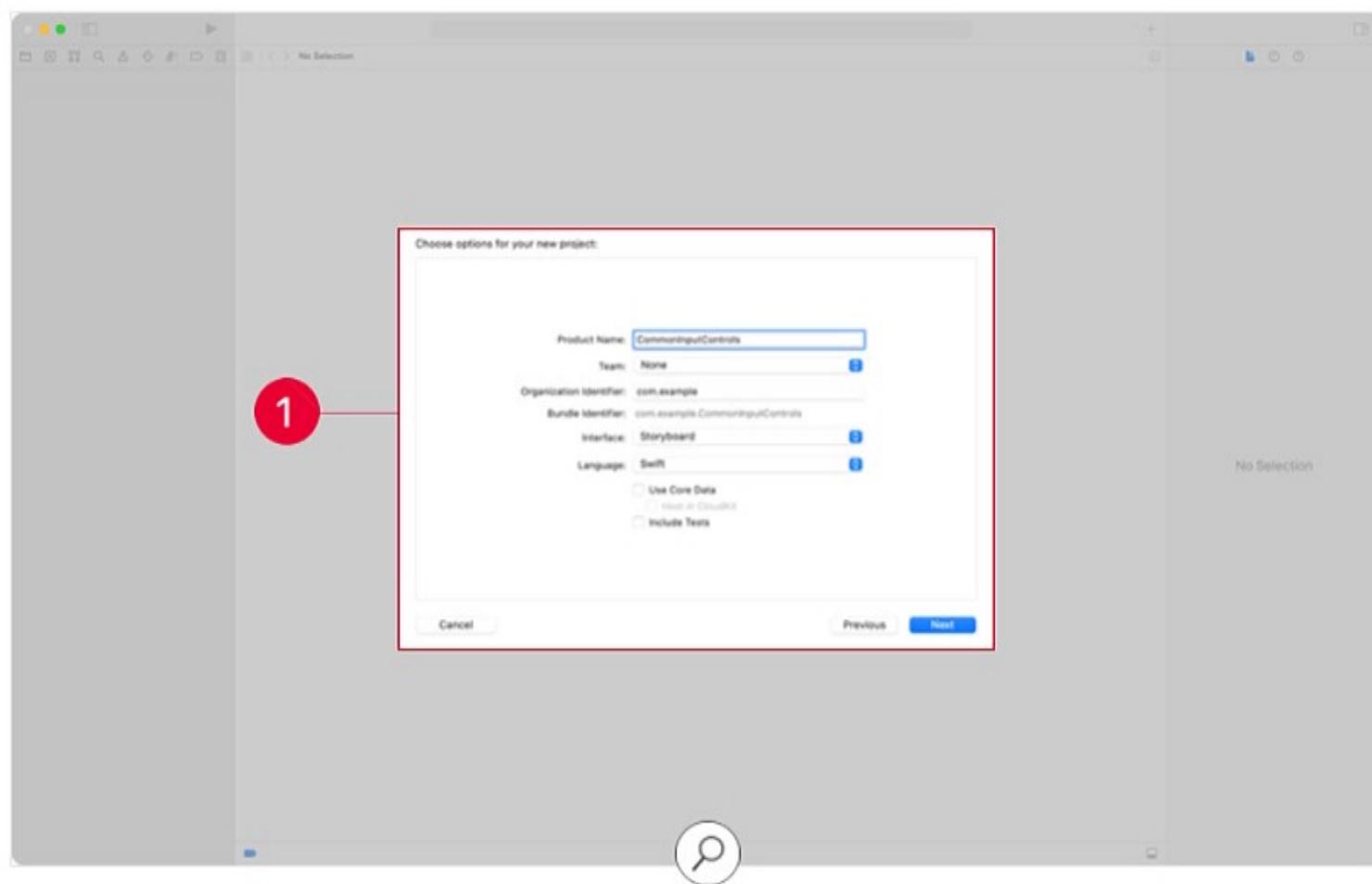
- [API Reference: UIButton](#)
- [API Reference: UISwitch](#)
- [API Reference: UISlider](#)
- [API Reference: UITextField](#)
- [API Reference: Gesture Recognizers](#)

When you first learned about Interface Builder, you added a button to the screen and printed when the button was tapped. In this lesson, you'll review that exercise with your new knowledge of views and controls, and you'll repeat it with switches and sliders.

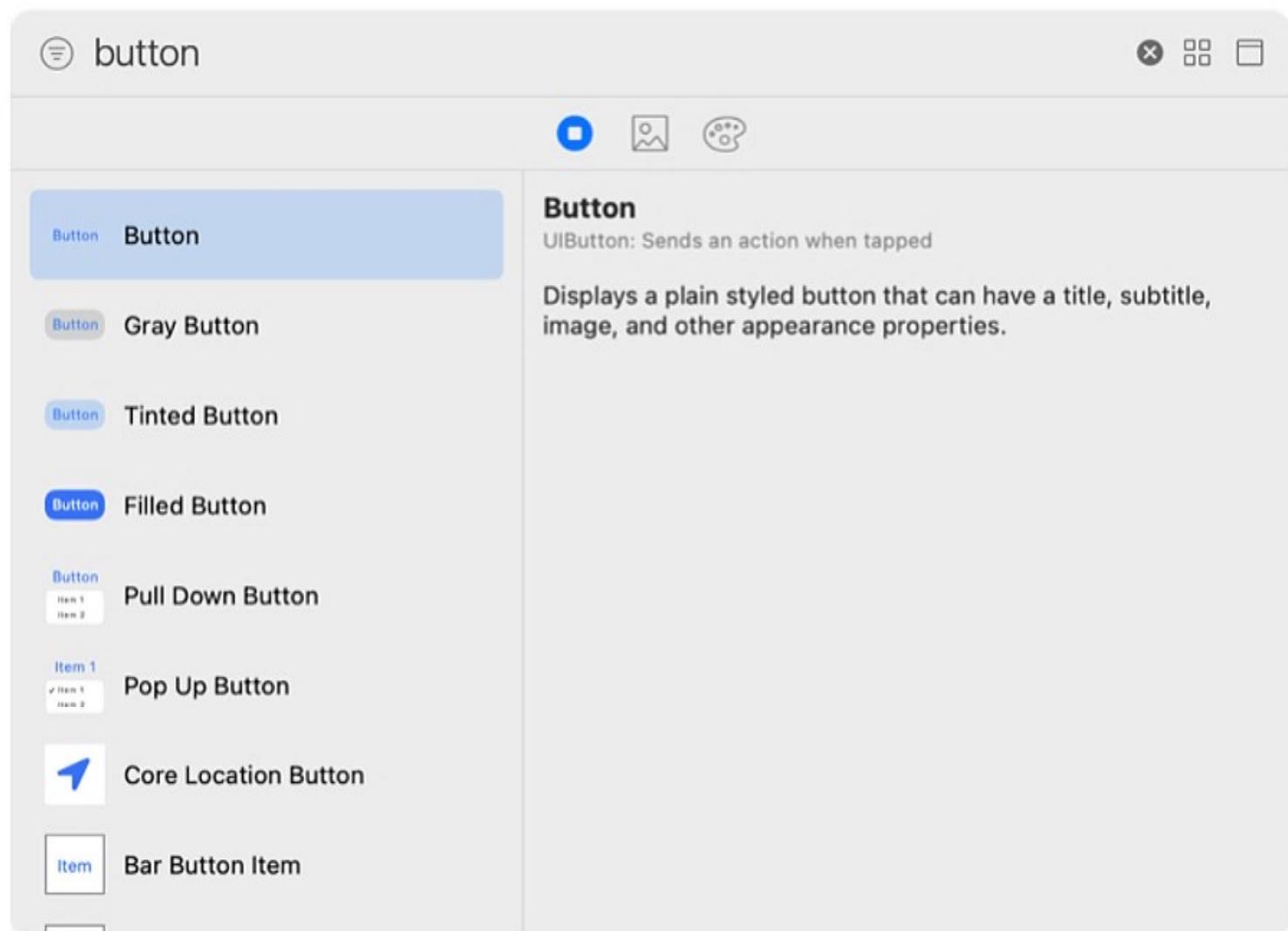
Buttons

As you know from your own direct experience with apps, buttons are the most common type of input control. Add a button to your main scene and have it print a statement to the console when it's tapped.

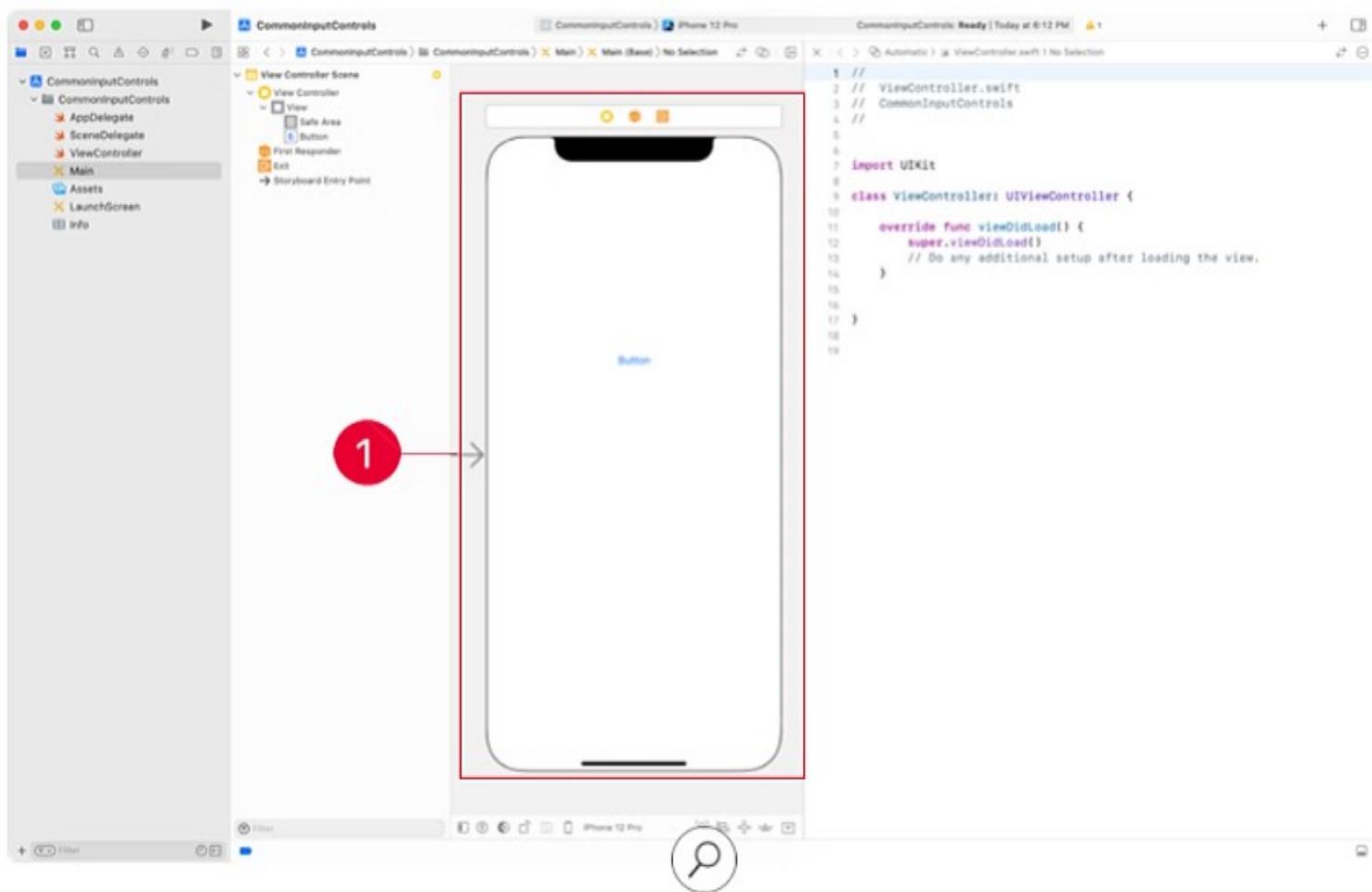
- 1 Create a new project called "CommonInputControls" using the iOS "App" template.



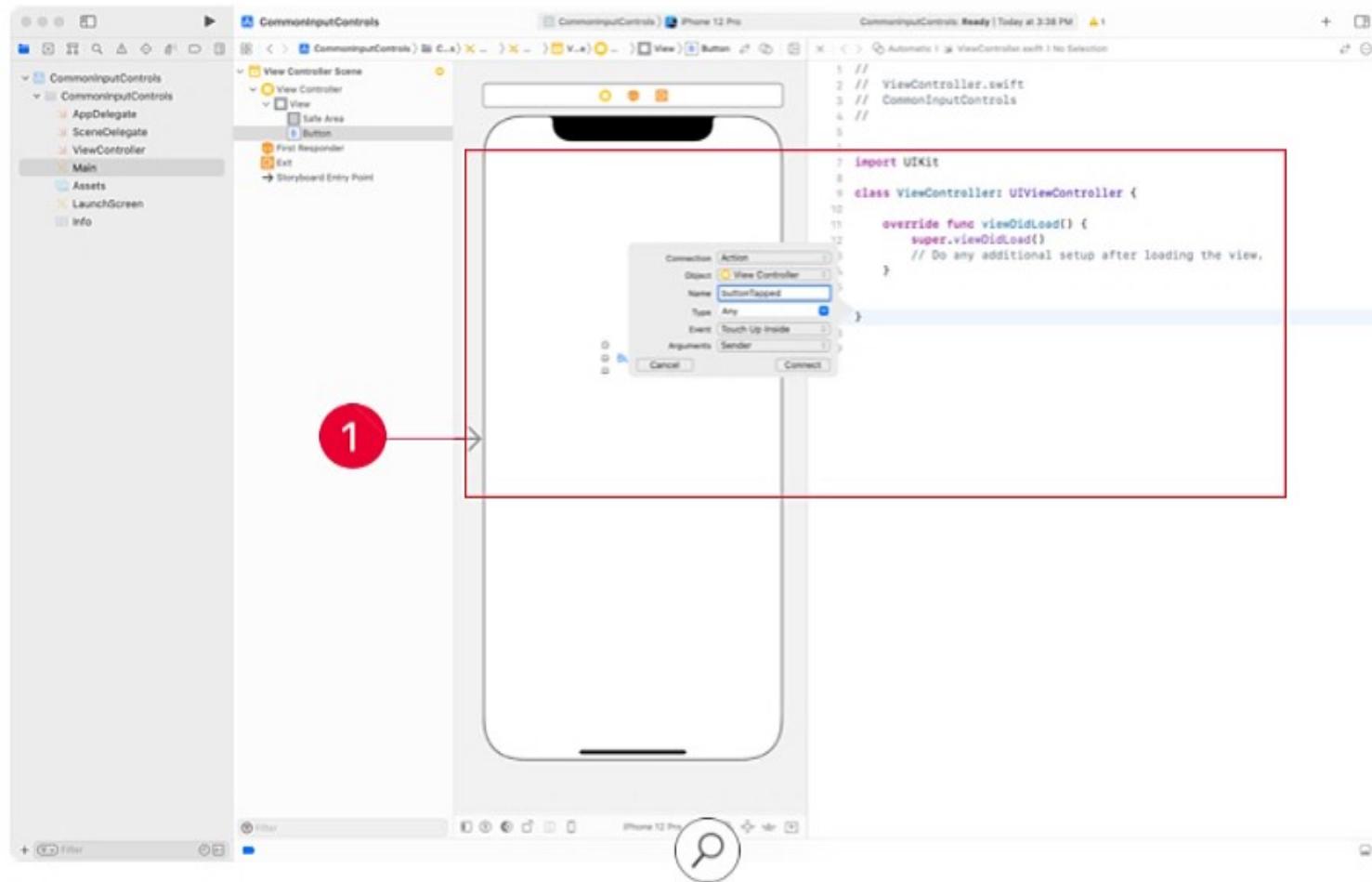
2. Open the **Main** storyboard. Then find a Button object in the Object library.



3. Drag the button to the scene, placing it about a third of the way from the top of the screen. ① Open an assistant editor to see the corresponding **ViewController** file.



4. Using the **Control-drag** shortcut, add an @IBAction from the button to the view controller code. Remember to add the action within the curly braces that define the **ViewController** class. On fresh view controller files, like this one, most developers add new actions below the `viewDidLoad()` block of code. ①



Use a descriptive name for your action, such as `buttonTapped`. Later on, when you look at the code in the **ViewController** file, you'll be glad that the name of the action gives you a clue as to what will trigger it.

The most natural time for a button to execute code is when the user taps it and releases the touch from within the bounds of the button. That's why "Touch Up Inside" is the default control event when you create an `@IBAction` from a button. You can also use the "Primary Action Triggered" control event, which will be triggered at the same time as "Touch Up Inside." If you want code to execute in response to a different control event, you can choose from the options in the pop-up menu.



Remember that when you create an `@IBAction` the function is passed to the `sender` as a parameter. The `sender` refers to the specific control that triggered the action. In this case, the `sender` is the tapped button.

```
@IBAction func buttonTapped(_ sender: Any) {  
}
```

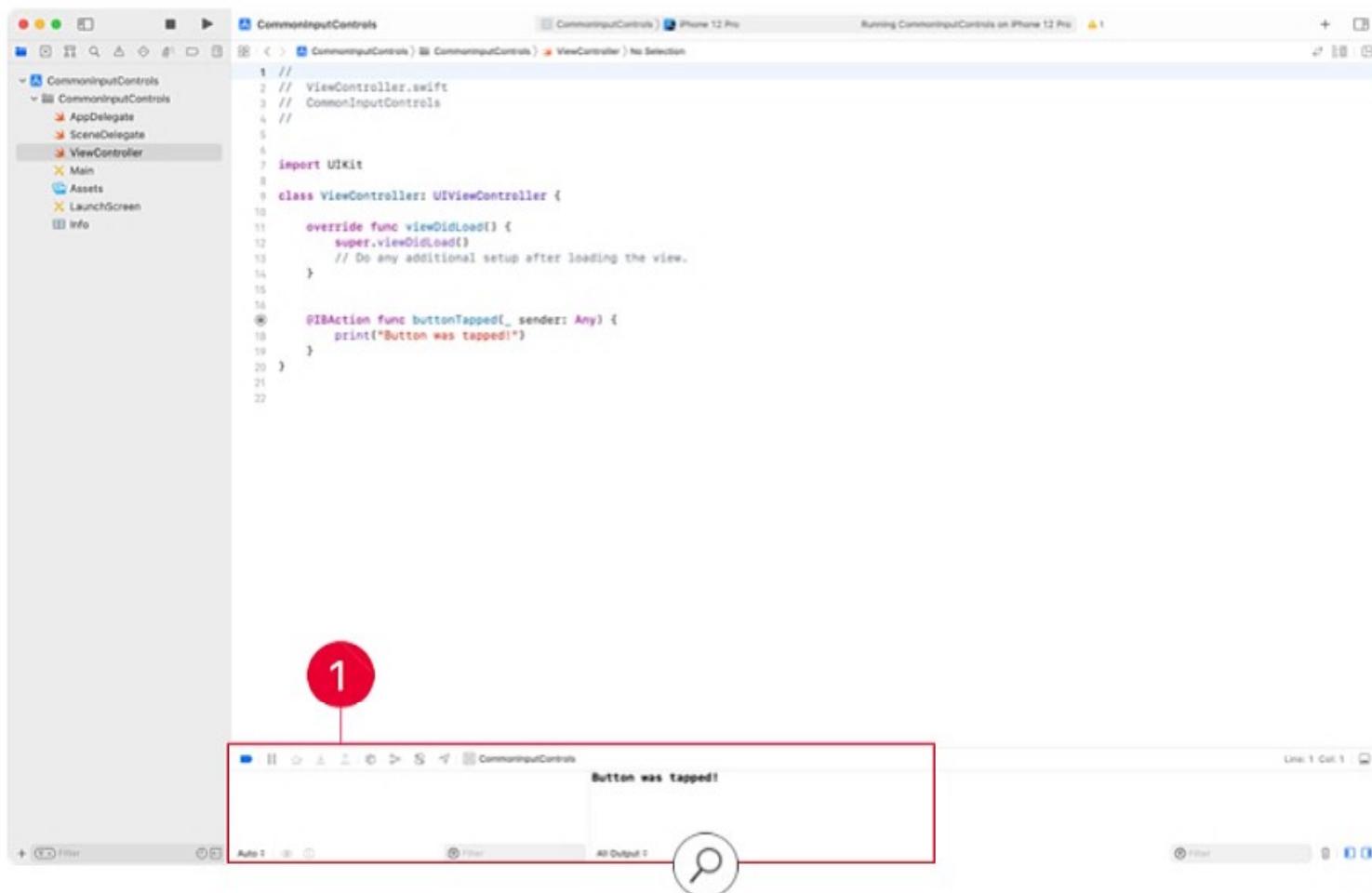
When you know that the `sender` will *always* be the same type of object (like the button in this example), you can change `Any` to the specific control type (`UIButton` in this example). To make this change, you'll access properties on `sender` within the `@IBAction`.

5. Use the `print()` function to print a line of text to the console when the user taps the button.

```
print("Button was tapped!")
```

You can put any code or logic inside the control's action using this same workflow.

6. Run the app in Simulator, and try clicking the button you just created. You should see the printed text in the console area. ①



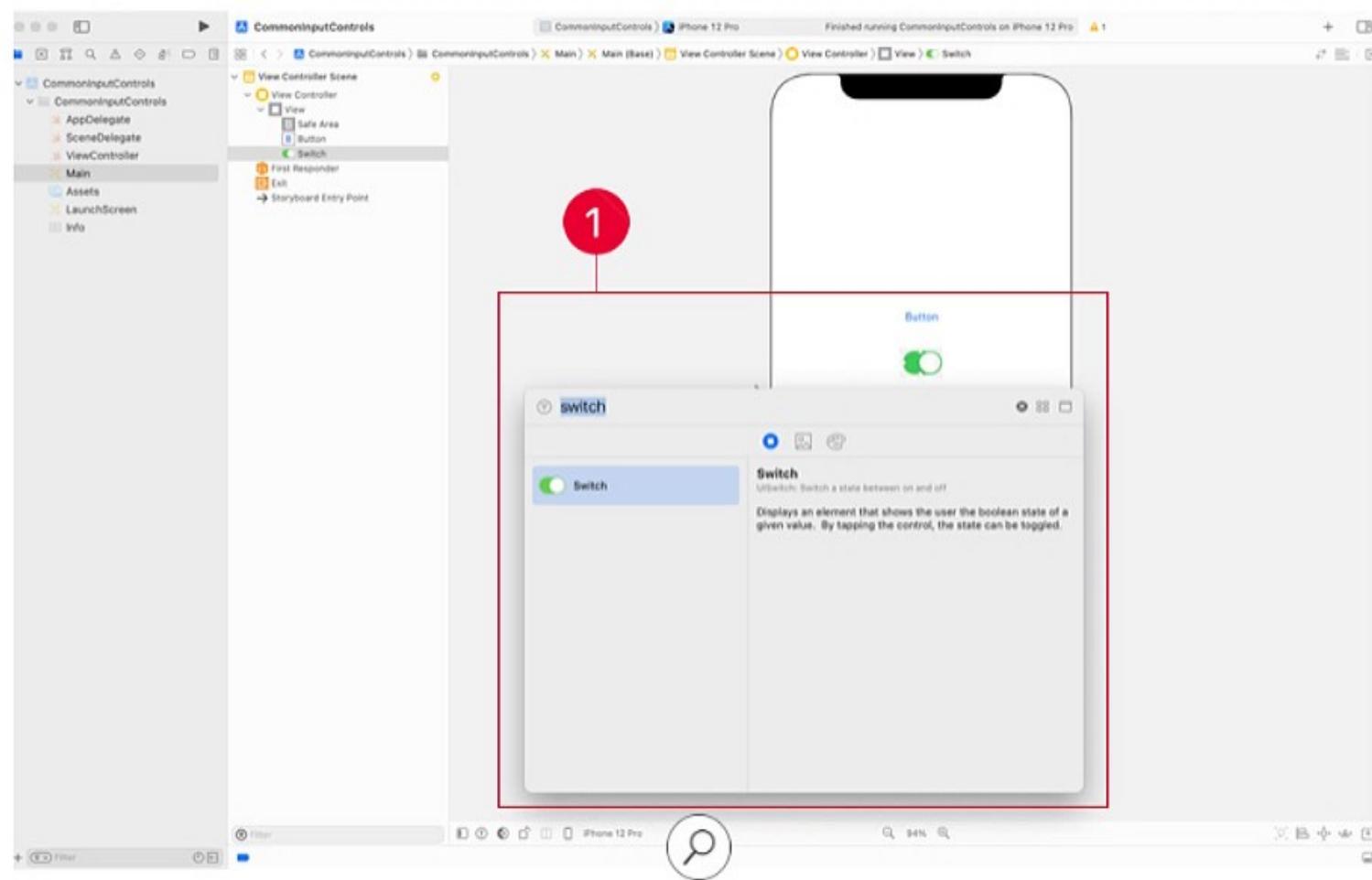
Right now, the text on the button doesn't relate to the action it will trigger. Use Interface Builder to update the button's text to something more descriptive of the code it will execute.

You can also set other button attributes, including its font, alignment, and shadow. To get a feel for the things you can adjust, take a moment to play around with the various options in the Attributes inspector.

Switches

Switches are used to toggle a single option. You can use an `@IBAction` to execute code when a switch is toggled one way or another. Or you can check if the switch is currently toggled to the on or off position by accessing the `isOn` value from the `sender` parameter or from an `@IBOutlet`.

Add a switch to your `CommonInputControls` project, and print whether the switch is on or off when the user toggles the control.

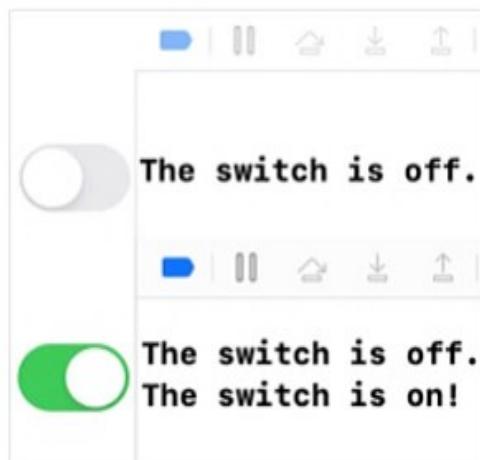


1. Find a Switch object in the Object library, and drag it to your scene. ①

2. Using the assistant editor, add an @IBAction from the switch to the **ViewController** file. As with button actions, use a descriptive name, such as `switchToggled`, and set the sender type to `UISwitch`. Add code that will print to the console whether the switch was turned on or off.

```
@IBAction func switchToggled(_ sender: UISwitch) {  
    if sender.isOn {  
        print("The switch is on!")  
    } else {  
        print("The switch is off.")  
    }  
}
```

3. Run the app in Simulator, and toggle the switch. You should see the printed text in the console area.

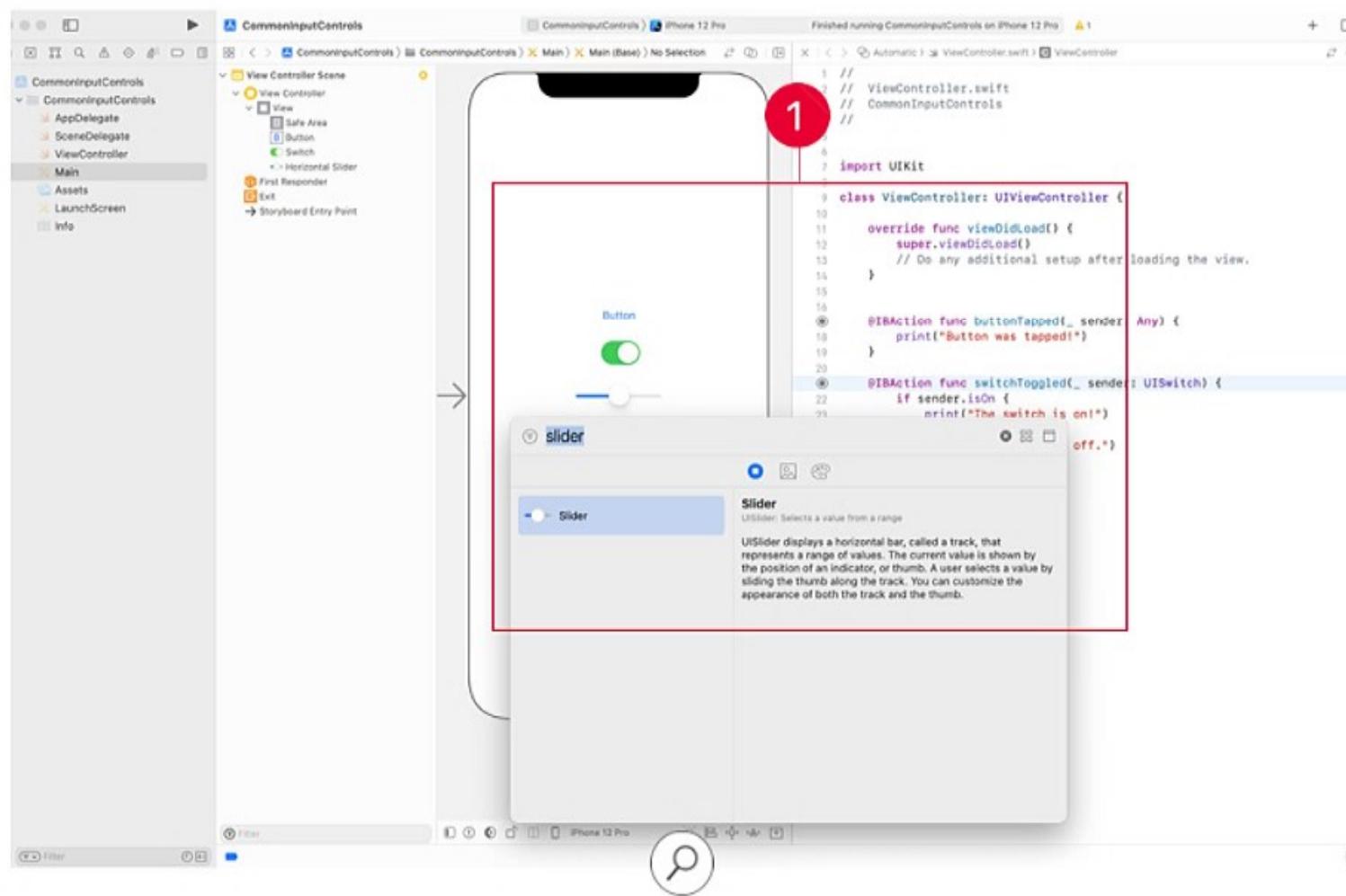


Sliders

Sliders allow the user to have smooth control over a value, such as adjusting volume or setting a numeric value with a large number of potential contiguous values. For example, in Display & Brightness settings, you use a slider to adjust the brightness of your device's display.

Add a slider to your CommonInputControls project, and print the slider's value as it changes.

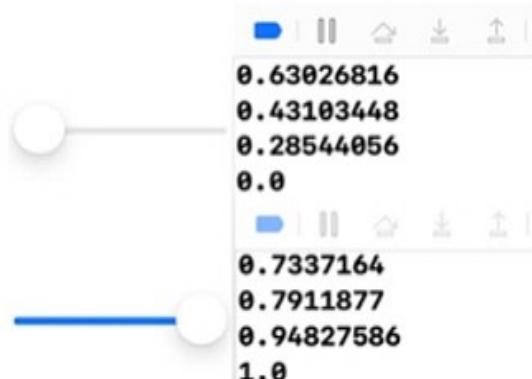
1. Find a Slider object in the Object library, and drag it to your scene.



2. Using the assistant editor, add an @IBAction from the slider to **ViewController**. Use a descriptive name, such as `sliderValueChanged`, and set the sender type to `UISlider`. Add some code that will print the value of the slider to the console.

```
@IBAction func sliderValueChanged(_ sender: UISlider) {  
    print(sender.value)  
}
```

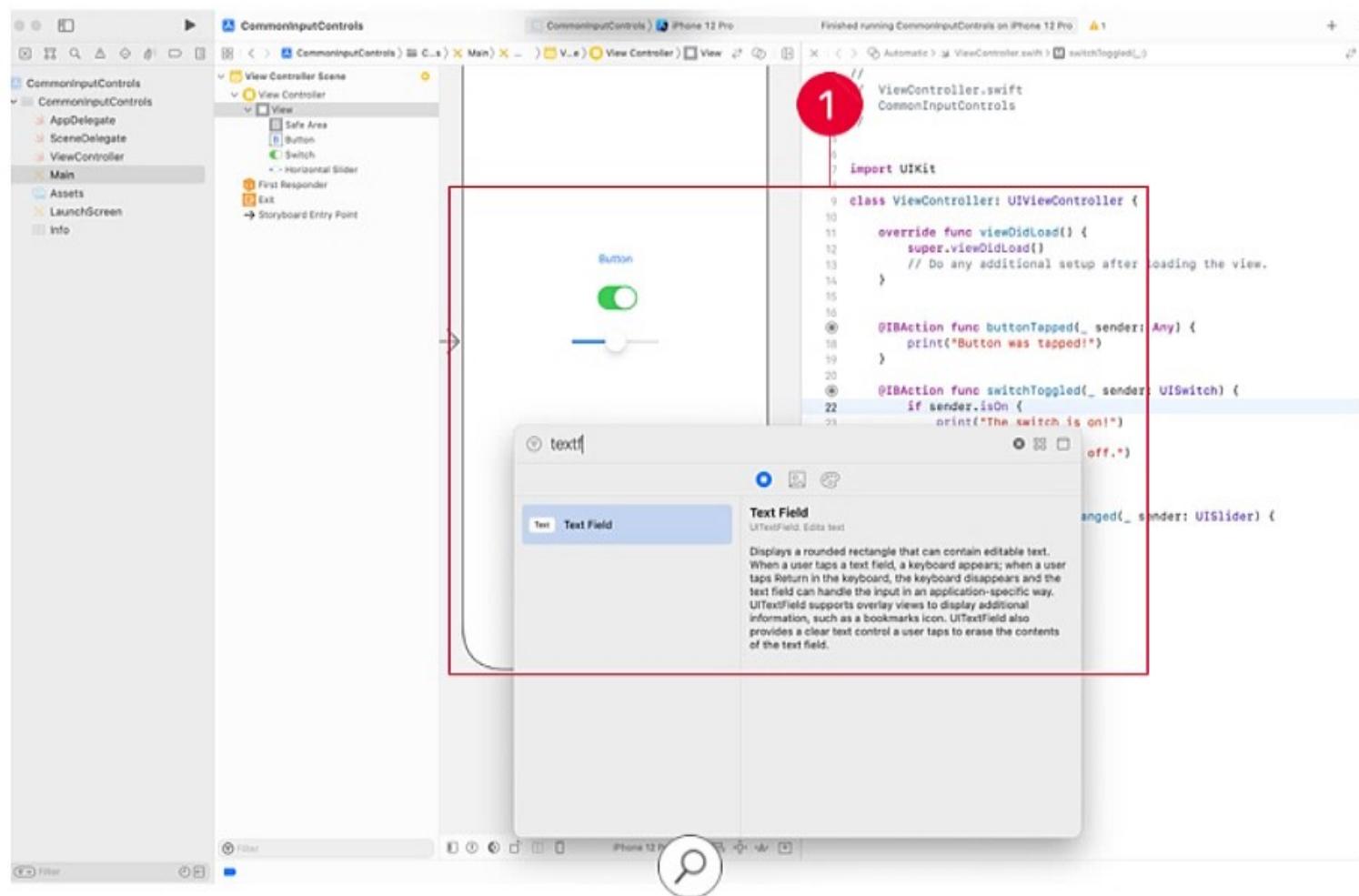
3. Run the app in Simulator, and move the slider. You should see the printed value in the console.



Text Fields

Text fields allow the user to enter a small amount of text, such as entering a username or password. For example, you use a text field when entering a subject on a new email draft.

Add a label and a text field to your "CommonInputControls" project, and set the label's text to the current text in the text field when the user taps the Enter or Done key on the keyboard.



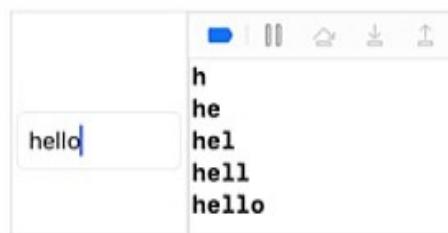
1. Find a Text Field object in the Object library, and drag it to your scene. 1

2. Using the assistant editor, add an @IBAction using the “Primary Action Triggered” event from the text field to **ViewController**. Use a descriptive name, such as keyboardReturnKeyTapped, and set the sender type to UITextField. Use the action to print the current text of the field.

```
@IBAction func keyboardReturnKeyTapped(_ sender: UITextField) {  
    if let text = sender.text {  
        print(text)  
    }  
}
```

3. Remember that text fields can also trigger code when the user edits the contents of the field. Using the assistant editor, add an @IBAction using the “Editing Changed” control event from the text field to **ViewController**. Use a descriptive name, such as textChanged, and set the sender type to UITextField. Use the action to print the current text of the field.

```
@IBAction func textChanged(_ sender: UITextField) {  
  
    if let text = sender.text {  
        print(text)  
    }  
}
```



Text fields are very flexible and should be configured to best fit the situation they are used in. For example, you can set the keyboard to display different keys based on whether the user is entering an email address or a URL. You can also turn autocorrect on or off, or set the text field to a secure input field to hide the characters as the user types them. Take some time to explore the options and use them appropriately when building your app.

Actions And Outlets

You can create outlets that allow actions to access properties on your views and controls—even if the view or control isn’t the sender.

For example, you may have an outlet for a label that gets updated by a button’s action.

In the steps below, you’ll change the `buttonTapped` function triggered by the tapped button to access the current `isOn` state of the switch and the current `value` of the slider.

1. Using the same **Control-drag** technique in the assistant editor, add `@IBOutlet` references for the switch and slider. Developers usually place `@IBOutlet` references above the `viewDidLoad()` function in a view controller file. Just as when naming an `@IBAction`, you should choose a descriptive name for your `@IBOutlet`. You might think to choose `switch` and `slider` for this `@IBOutlet`, but there’s a problem with `switch`. It turns out that “`switch`” is a reserved keyword in Swift, so the best practice is to use `toggle` to name an `@IBOutlet` for a switch.

```
@IBOutlet var toggle: UISwitch!
@IBOutlet var slider: UISlider!
```

2. Update the `buttonTapped` function in the `@IBAction` from the button to print the current `isOn` state and the value of slider.

```
@IBAction func buttonTapped(_ sender: UIButton) {
    print("Button was tapped!")

    if toggle.isOn {
        print("The switch is on!")
    } else {
        print("The switch is off.")
    }

    print("The slider is set to \(slider.value)")
}
```

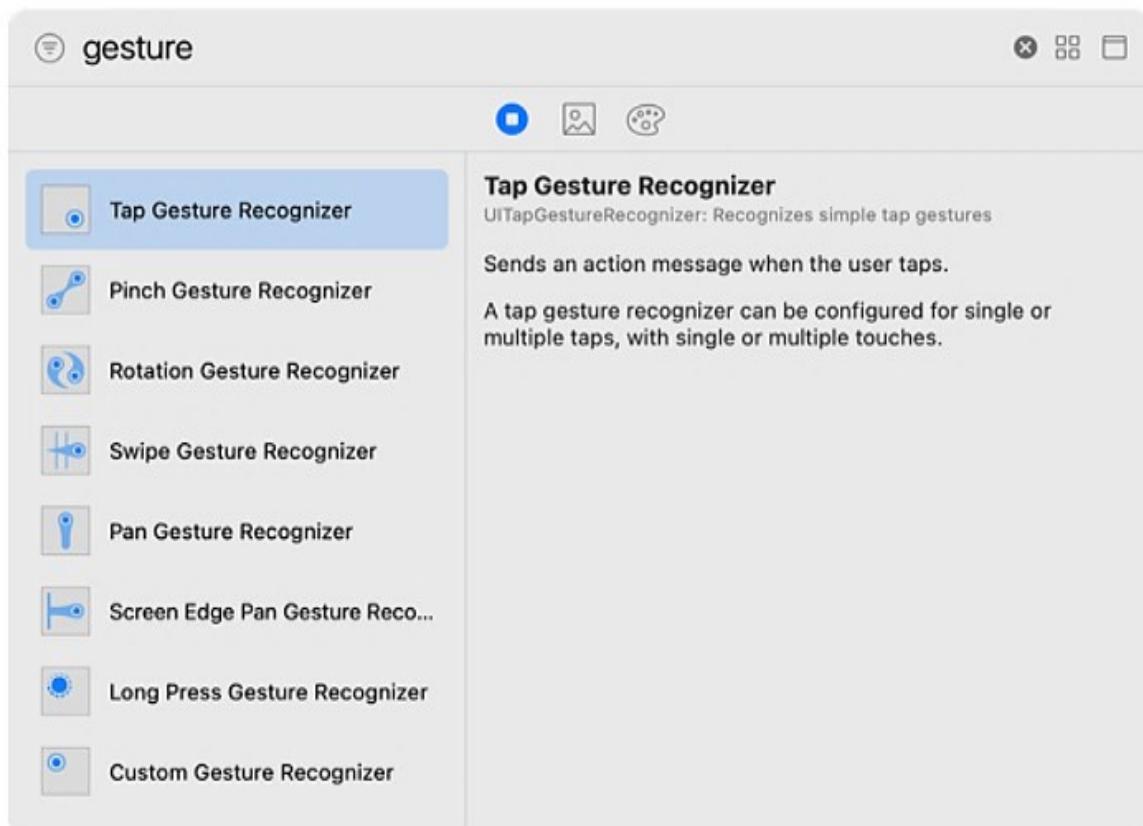
- Run the app in Simulator and click the button. You should see the printed text in the console.



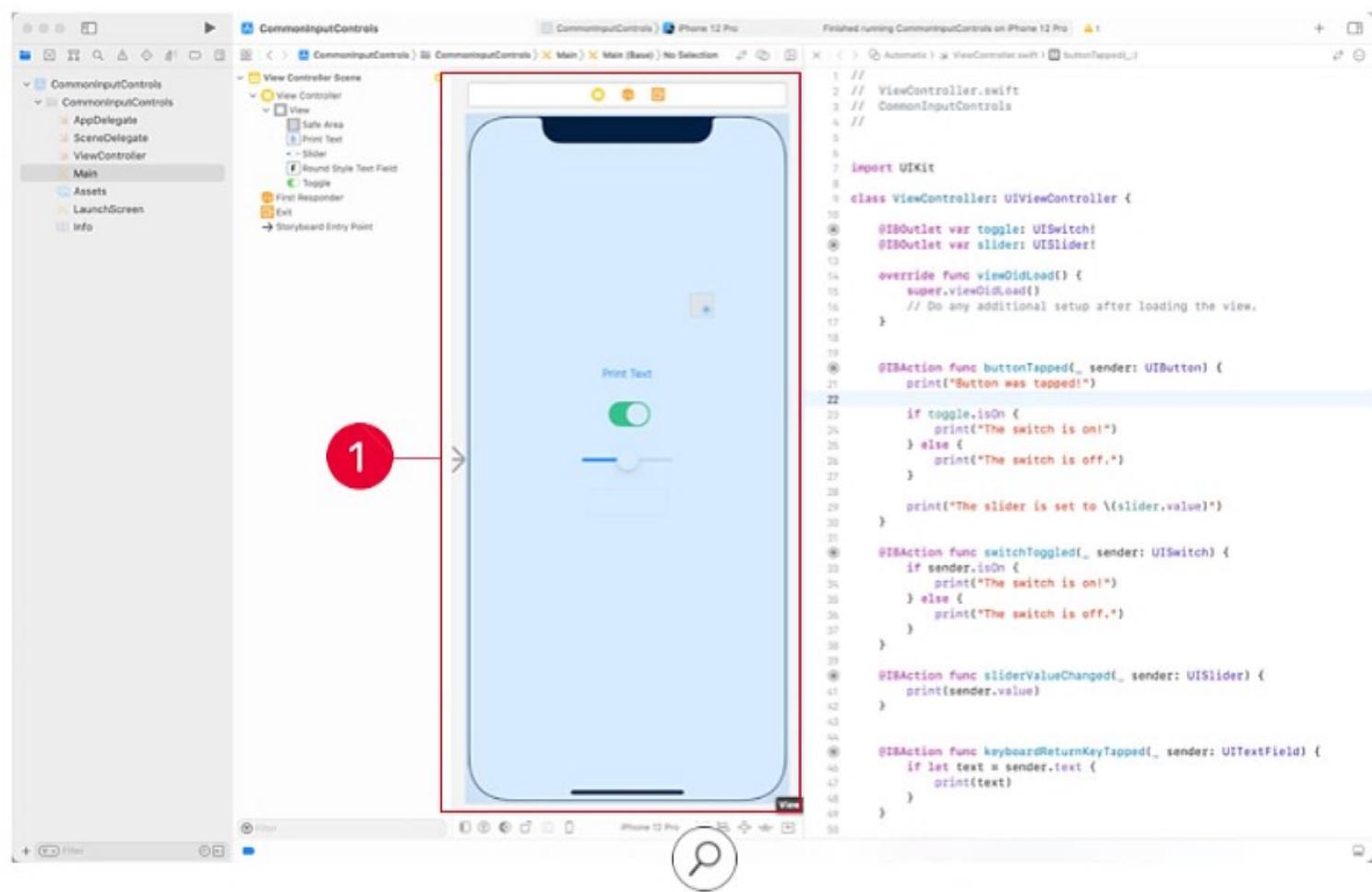
Gesture Recognizers

Each of the controls interact with gestures: buttons and switches are configured to work with taps, sliders work with swipes, etc. A gesture is tied to a single view, but a view can have multiple gestures. For example, a switch responds to a tap, but the user can also swipe a switch left and right to turn it on and off. You can add your own gesture recognizers on top of any view you wish, from simple taps to complex pinches, long-presses, and rotations.

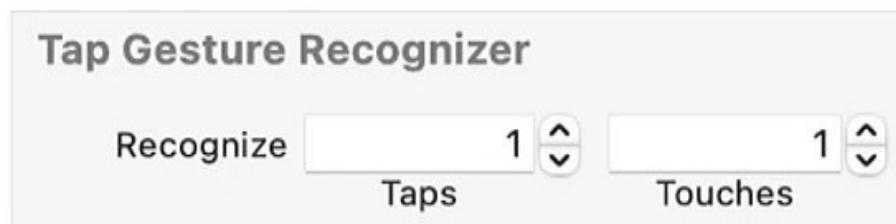
In this example, you'll add a tap gesture recognizer on top of your view controller's view. To begin, search the Object library for "gesture" to see the full list of available gesture recognizers.



Grab the Tap Gesture Recognizer, and drag it on top of the view that should respond to the tap event. In this case, drag it on top of the view controller's view. ①



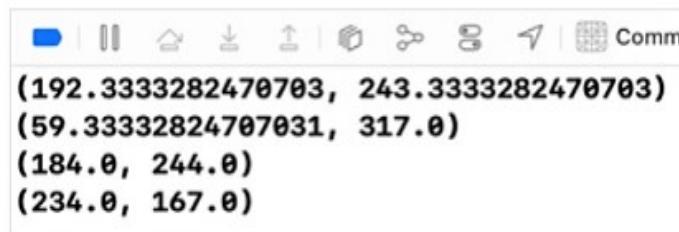
You will see the recognizer added to the Document Outline. Select the recognizer from the Document Outline and open the Attributes inspector. Different gestures will contain a different list of properties. You can adjust the number of taps that the recognizer requires, such as a single or double-tap, as well as the number of touches (or fingers) required to trigger any actions connected to the recognizer. In this example, you want to recognize a single tap using a single finger, so leave both values at 1.



Using the assistant editor, add an `@IBAction` from the recognizer to **ViewController**. Use a descriptive name, such as `respondToTapGesture`, and set the sender type to `UITapGestureRecognizer`. The following implementation of `respondToTapGesture` will determine where the user tapped when the recognizer's action was called, and will print out the x/y value to the console.

```
@IBAction func respondToTapGesture(_ sender:  
UITapGestureRecognizer) {  
    let location = sender.location(in: view)  
    print(location)  
}
```

Build and run your application, then try tapping on the screen. The coordinates of your tap will be printed. Notice how the action does not trigger when you interact with the other controls on-screen, because the gesture is only tied to the view controller's view.



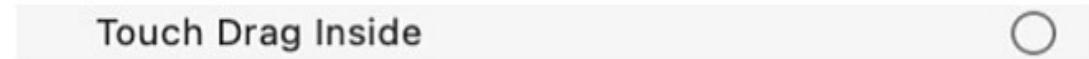
Programmatic Actions

How does Interface Builder connect controls to actions? It is helpful to understand how to connect controls to actions through code. At the very least, it will help you understand the work that Interface Builder is doing on your behalf.

To demonstrate connecting a control to a method programmatically, you will disconnect the existing button from the `buttonTapped(_ :)` action and re-connect the two in code. Highlight the button in your storyboard, then open the Connections inspector. In the list of control events, you can see that the button executes the `buttonTapped(_ :)` method when your finger presses down, then up, inside the control.



Tap the 'x' next to the method name, breaking the connection between the button and the action. Touch Up Inside will no longer be connected to any methods.



In order to connect the button to a method programmatically, you'll need a reference to the button in code. Use Interface Builder to create an `@IBOutlet` for the button.

```
@IBOutlet var button: UIButton!
```

The best place to connect the button to the action is right after the view has finished loading. Add the following code to the bottom of `viewDidLoad()`:

```
button.addTarget(self, action: #selector(buttonTapped(_:)),  
    for: .touchUpInside)
```

The `addTarget(_:action:for:)` method will connect the control to a particular action, and it requires three arguments. The first argument is the the owner of the function you want to execute. The owner of the `buttonTapped(_:)` method is the `ViewController`, or `self`. The second argument is a “selector”: the name used to select a method to execute for an object. Swift uses `#selector` as its syntax to locate a particular method. The last argument is the event that should trigger the action. Just like you saw earlier in the Connections inspector, you should tie your actions to the Touch Up Inside event. (Other controls, such as switches and sliders, should utilize the Value Changed event.)

Build and run your application, and verify that when you tap the button the method still executes. Save your `CommonInputControls` project to your project folder.

Lab—Basic Interactions

The objective of this lab is to create and compile an app with two buttons that, when tapped, change the contents of a label.



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

Step 1

Create Your View In Interface Builder

- In the storyboard, drag a text field from the Object library and place it at the top of the view. Use the layout guides to position the label so it extends from the left margin to the right margin of the view. Set the placeholder text to “Enter text to display in the label below.”
- Drag two buttons from the Object library. Place one just below the text field, and the other just below that. They should be centered in the screen. Change the title of the top button to “Set Text” and change the title of the bottom button to “Clear Text.”
- Drag a label from the Object library and place it under the last button. Use the layout guides to position the label so it extends from the left margin to the right margin of the view.
- In the Attributes inspector, set the label’s text alignment to centered. Double-click the label and delete the existing text. Set new text to say “Placeholder.” New text will be set dynamically when the user taps one of the buttons. You’ll get to that soon.

If you run the app, you should see your text field and two buttons. But if you click them, nothing will happen. That will change soon.

Step 2

Create Outlets And Actions

- With the **Main** storyboard still selected, open the assistant editor. The implementation file for your view controller should appear next to the canvas for the storyboard.
- Create an `@IBOutlet` for the label in the **ViewController** file.
- A popover will appear. Making sure the connection is set to Outlet, name the outlet `label`, and set the type to `UILabel`. Then click Connect.
- Create an `@IBOutlet` for the text field in the **ViewController** file. Name the outlet `textField`.
- Create an `@IBAction` from the Set Text button with the name `setTextButtonTapped`, and from the Clear Text button with the name `clearTextButtonTapped`.

Step 3

Add Code For Your Actions

Your buttons now have actions, but they don't execute any code yet. In this step, you'll finally get your actions to do something.

- Select `ViewController` in the Project navigator. You'll implement the action in this file.
- Implement the `setTextButtonTapped` action to set the `label.text` to the current text in the textfield.

What did you just do? You assigned a new string to the `text` property of your label. The label will now display the string when the Set Text button is tapped or clicked.

- Implement the `clearTextButtonTapped` action to set the `textField.text` and the `label.text` to empty strings.

Now the Clear Text button will clear both the label and the text field.

- Run the app, and test it by adding text in the text field and tapping the Set Text button. The text of the label should reflect the text in the text field. Now tap the Clear Text button. The text in both the text field and the label should be cleared.

Well done! You've now built an app that uses multiple controls to execute code that updates properties on another view. You're ready to move on to the next lesson. Be sure to save your work in your project folder.

Connect To Design

Open up your App Design Workbook, and review the Map section for your app. Start thinking about where you might want to use buttons, switches, sliders, and text fields in each scene. Add comments to the Map section or in a new blank slide at the end of the document. What actions and outlets will you need to create? Will your app need to use gesture recognizers?

In the workbook's Go Green app example, the app will have a challenges section where the user can compete with friends, accept challenges, and earn rewards for recycling. In the challenges section, there will be a button to enter a code sent by a friend. When the button is pressed, a text field will appear where the user can enter the code to unlock a challenge. The button will need to have an action and an outlet.

Review Questions

Question 1 of 6

Which of the following allows you to execute code for a specific control event?

- A. @IBAction
- B. @IBOutlet

[Check Answer](#)



Lesson 2.11

Auto Layout and Stack Views

When you build an app, you want to make sure it looks good on every iOS device. Xcode includes a powerful system called Auto Layout which makes it easy to build intricate interfaces that work on various screen sizes.

Auto Layout relies on constraints, or rules, to dynamically calculate the size and position of all views in a view hierarchy. So your interfaces will look and work the same—no matter what device your users have in their hands or how they’re holding it.

In this lesson, you’ll learn the fundamentals of Auto Layout for building precisely designed user interfaces.

What You'll Learn

- How to use Auto Layout to build precise views
 - How to create constraints
 - How to use stack views to simplify Auto Layout
-

Vocabulary

- [Auto Layout](#)
 - [constraint](#)
 - [sibling](#)
 - [size class](#)
 - [stack view](#)
-

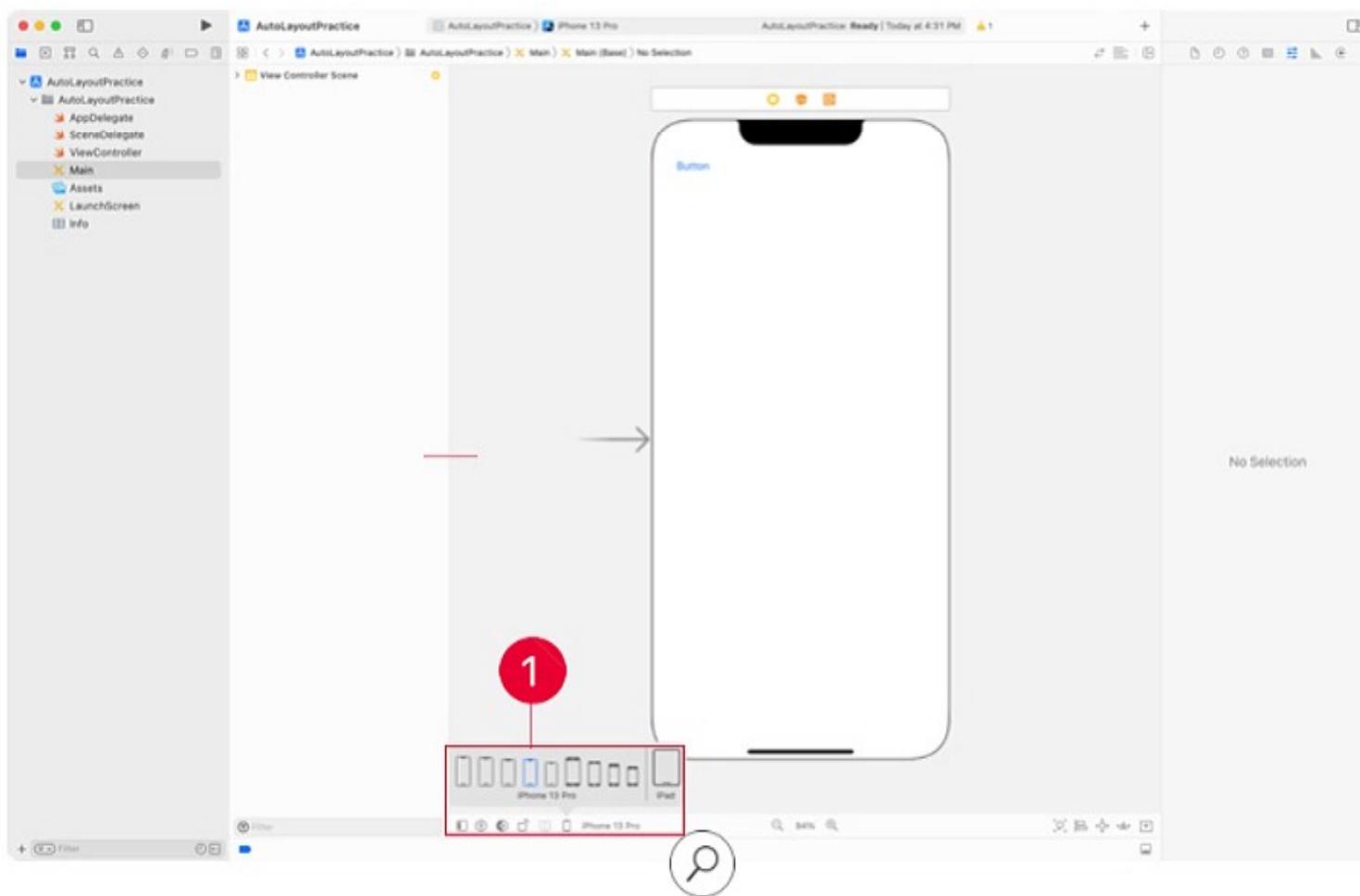
Related Resources

- [API Reference: UIStackView](#)
- [WWDC 2015: Implementing UI Designs in Interface Builder](#)
- [WWDC 2015: Mysteries of Auto Layout, Part 1](#)
- [WWDC 2015: Mysteries of Auto Layout, Part 2](#)
- [API Reference: UITraitCollection](#)
- [WWDC 2017: Auto Layout Techniques in Interface Builder](#)
- [WWDC 2019: Introducing Multiple Windows on iPad](#)

As you begin developing apps for iOS, it's important to think of the different devices that your apps will run on. You've learned that every user interface element has a size and a position on the screen. How might the size and position of those elements change in relation to the size and orientation of the device?

Create a new Xcode project using the iOS App template. When creating the project, make sure the interface option is set to Storyboard. Name the project "AutoLayoutPractice." Open the **Main** storyboard and use the layout guides to add a Button from the Object library to the top-left corner of the screen.

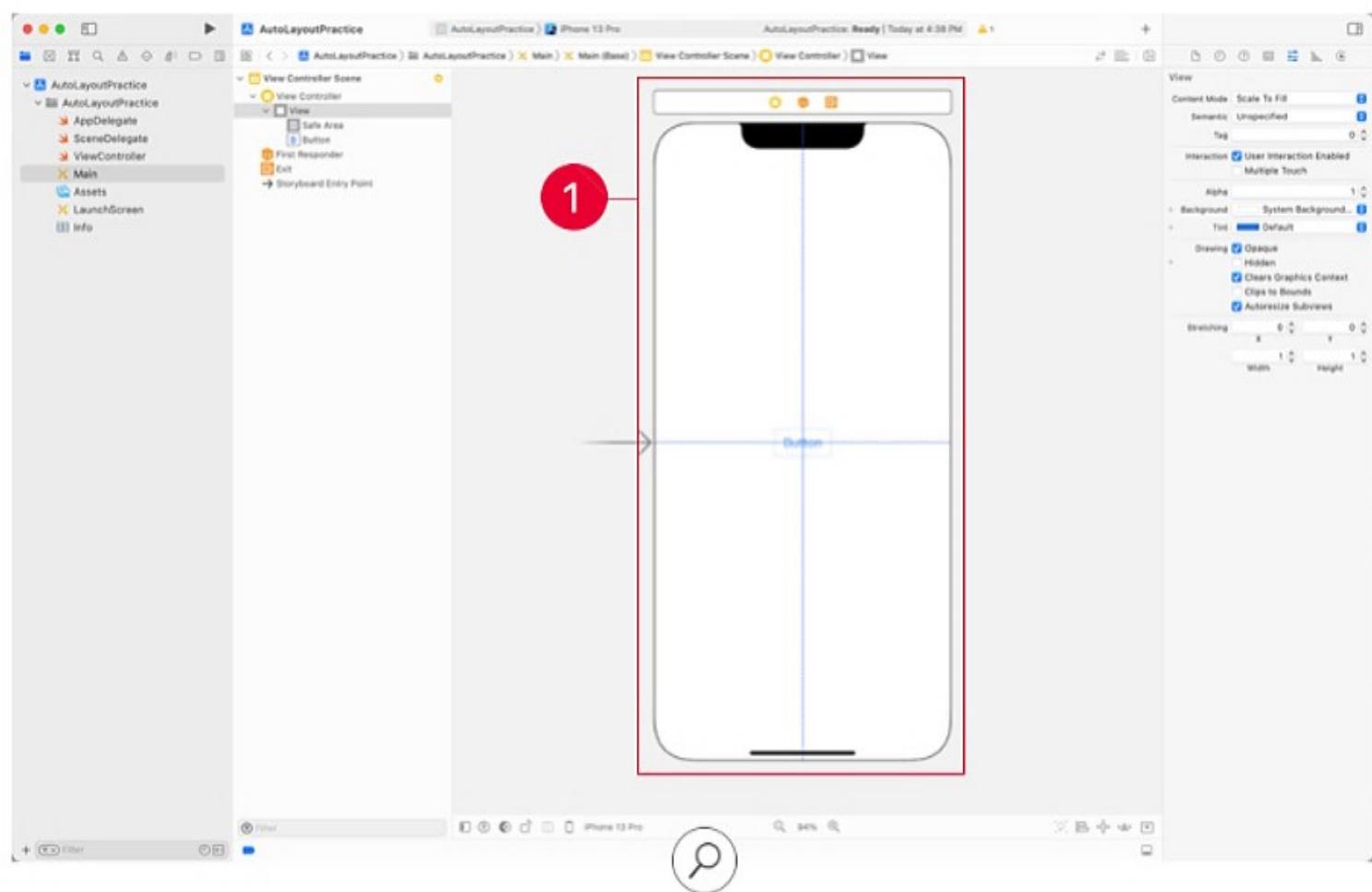
Interface Builder provides a quick way to check what your interface will look like on different screen sizes. Click the "Devices" button at the bottom of the canvas (the current selection is shown next to the button) to reveal a menu that lists multiple iOS devices. To the left of the "Devices" button are buttons for adjusting how the interface style, orientations, appearance and accessibility setting will be viewed. ①



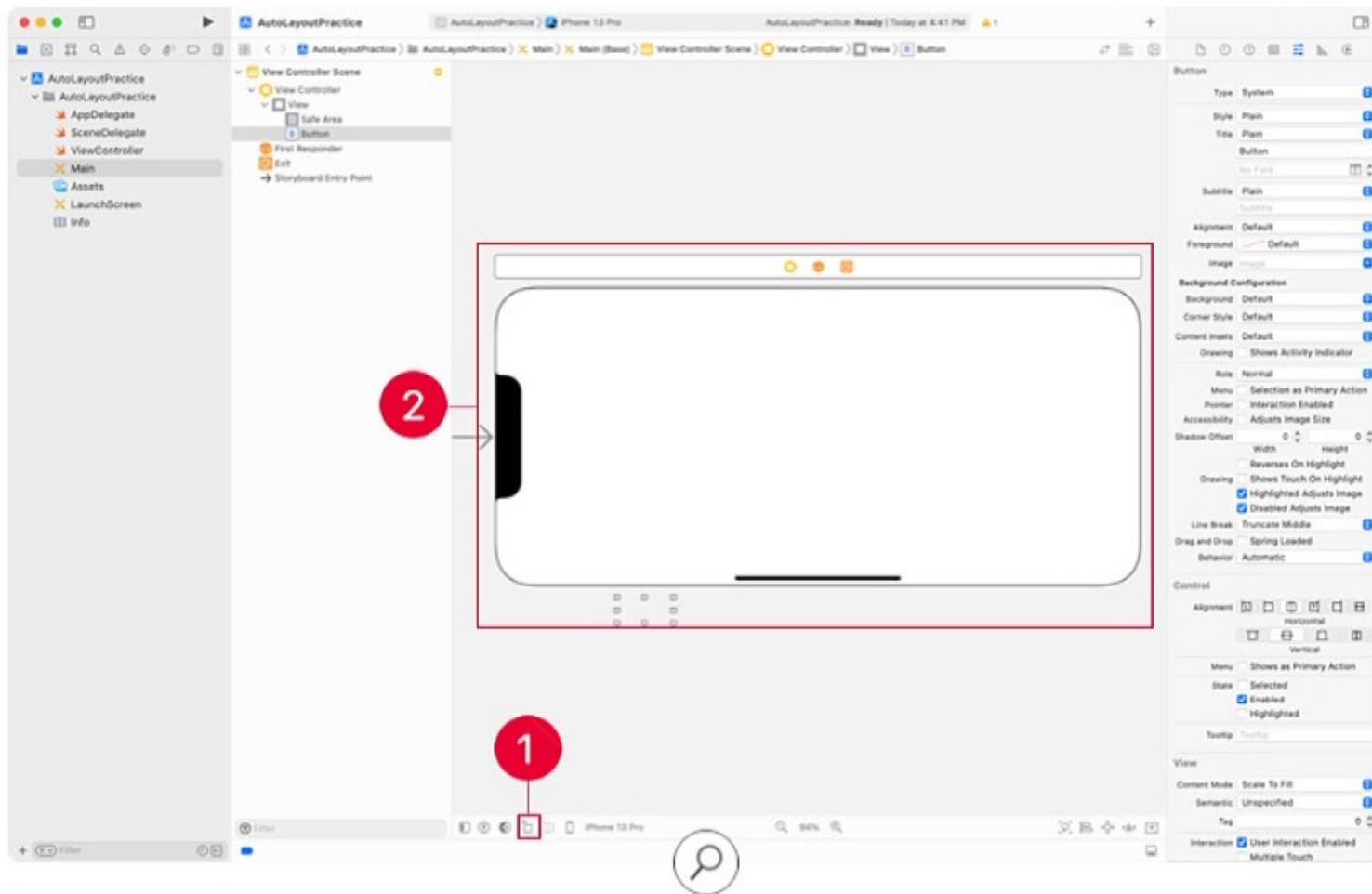
Go ahead and play around with the list. As you choose different screen sizes, styles, and orientations, the canvas redraws your user interface accordingly. If you've positioned a button in the top left of the canvas, it remains in that same position—no matter which device or orientation you've selected.

Why Auto Layout?

Now return the canvas to the iPhone 13 / iPhone 13 Pro screen size in portrait orientation, then move the button to the center of the screen. Use the blue alignment guides to ensure that the button is exactly in the center of the view. ①



Now switch the orientation to landscape mode. ① You'll see that the button is no longer centered. In fact, it isn't even on the screen! ②



What's going on? The button has stayed in the same X/Y position, based on the portrait orientation when you created the button. If you want the button to stay in the exact center, regardless of screen size or orientation, you'll need to create a set of constraints, or rules, that can be used to determine the size and position of your button. This system of using constraints to make adaptive interfaces is called Auto Layout.

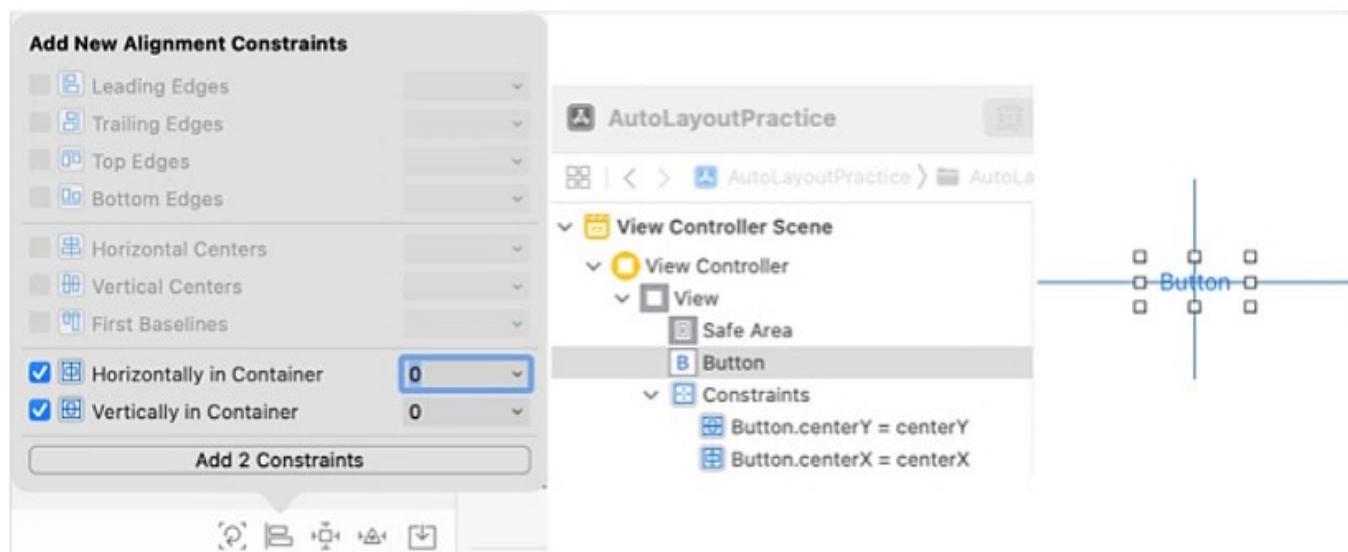
Create Alignment Constraints

At the right of the bottom button bar is a set of tools for creating and managing constraints. To lock the button in the center of the screen, you'll create two constraints that define the position of the button:

- The horizontal center of the button is equal to the horizontal center of the view.
- The vertical center of the button is equal to the vertical center of the view.

Start by returning to the portrait orientation and ensuring that the button is centered in the view. With the button selected, click the Align button  , the second button in the bottom bar of constraint tools.

A popover displays a list of alignment constraints, which define the relationship between the selected object and the parent view. Select the bottom two options, “Horizontally in Container” and “Vertically in Container,” then click “Add 2 Constraints.” Once you’ve created these constraints, you should see crossing blue lines over the top of your button, indicating horizontal and vertical alignment with the parent view.

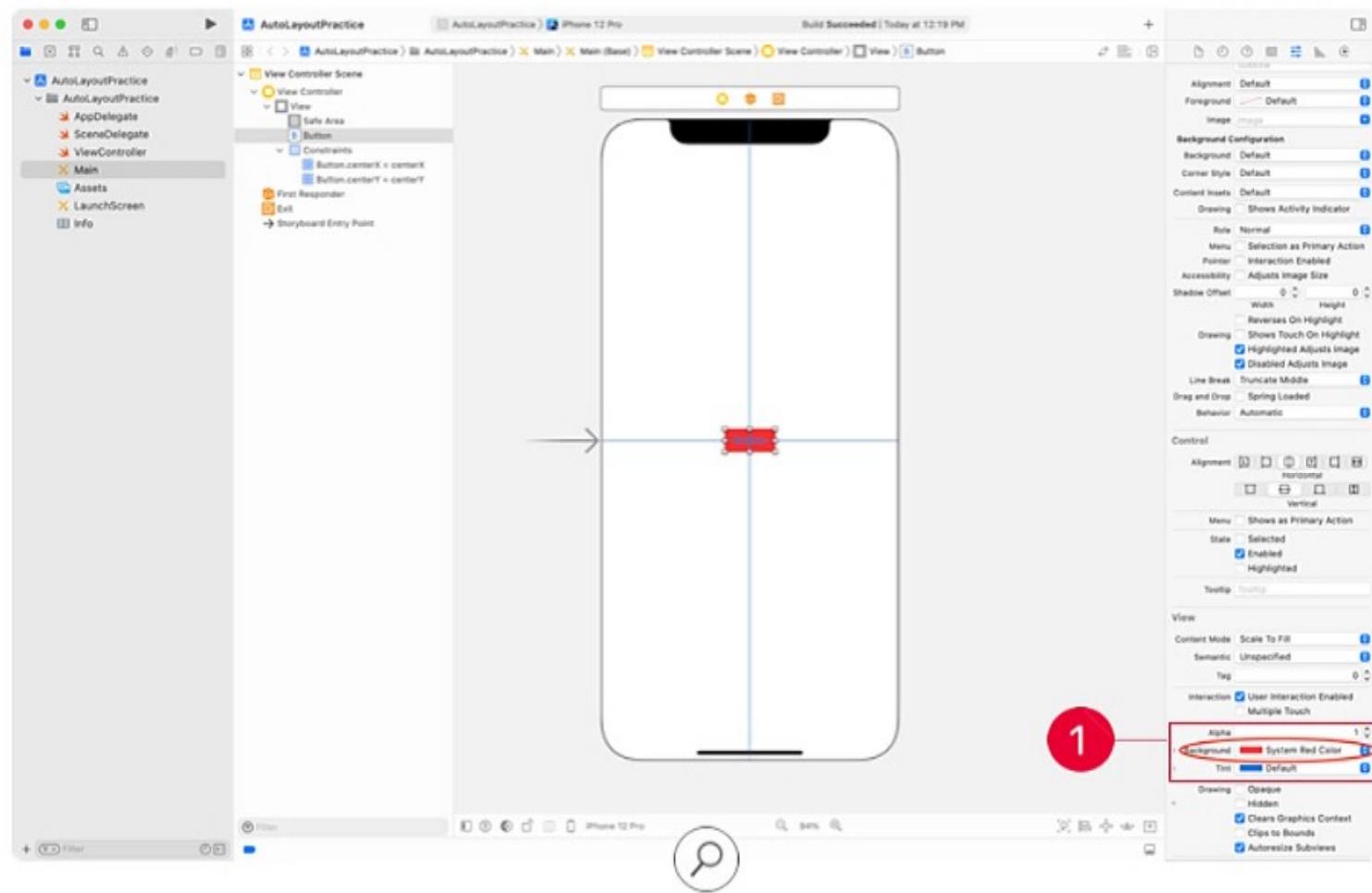


Use the “View as” button to select different devices and orientations. You’ll see that the alignment constraints you added are keeping the button centered in all views.

Create Size Constraints

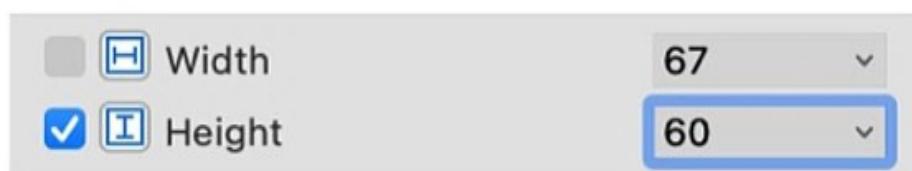
What about the width and height of the button? Since there aren't currently any size constraints on the button, its size is dictated by the button text and font. But Interface Builder provides multiple ways to use constraints to set the width and height of interface elements.

Start by using the Attributes inspector to change the background color of the button to something other than clear. This will make the button's size easier to see. ①



Now assume you want the height of the button to always be 60, no matter the screen size or orientation of the device. Click the Add New Constraints button , the button immediately to the right of the Align tool.

The popover displays a list of text fields and checkboxes to help you add constraints. Select the Height checkbox and adjust the value to 60. Click “Add 1 Constraint” to create the height constraint.



Note again that if you use the “View as” button to select different devices and orientations, you’ll see that the size constraints added are keeping the button size identical in all views.

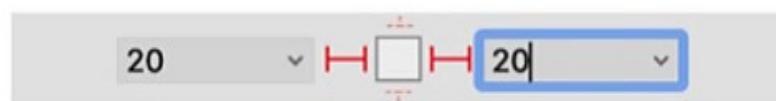
Constraints Relative to the Screen

You’ve fixed the height of the button, but what if you want the button’s width to vary based on the screen size? Assume you want the button’s left and right edges to always be 20 pixels from the left and right edges of the view, respectively.

Click the Add New Constraints button again, and notice the four fields at the top of the popover. These values define the distances from the top, leading, bottom, and trailing edges of the button to the nearest view. In this case, since there are no other views on the screen, they’re dictating the distances from the button edges to the view’s edges.

Leading refers to the left edge of the screen, while trailing refers to the right edge. These are labeled “leading” and “trailing” instead of “left” and “right” since not all languages read in the same direction. For certain users, you’ll want your app flipped. Leading and trailing constraints make this process easier.

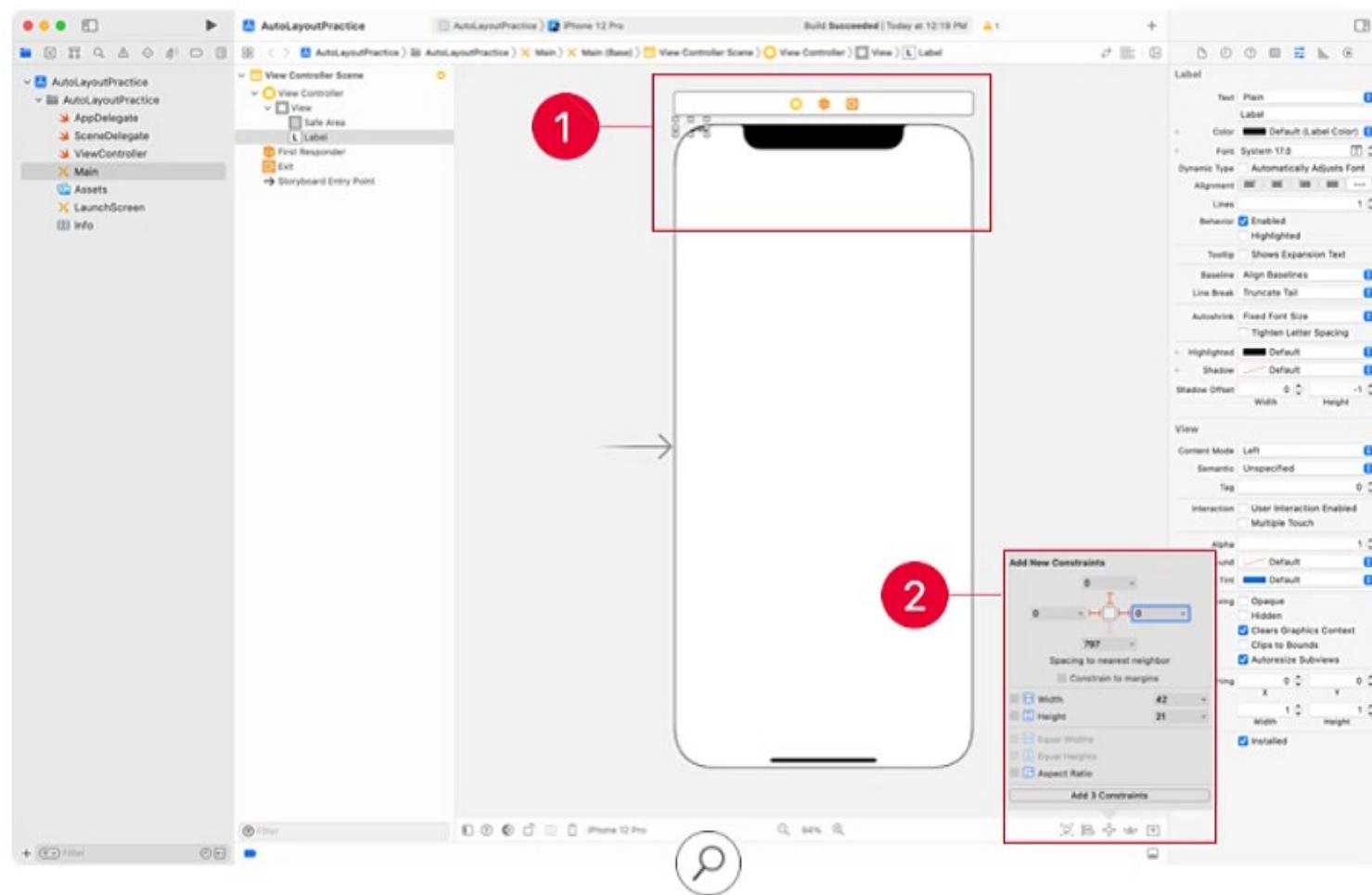
Adjust the left and right-edge values to 20. The red indicators will illuminate to show which edges are being constrained. Click “Add 2 Constraints.”



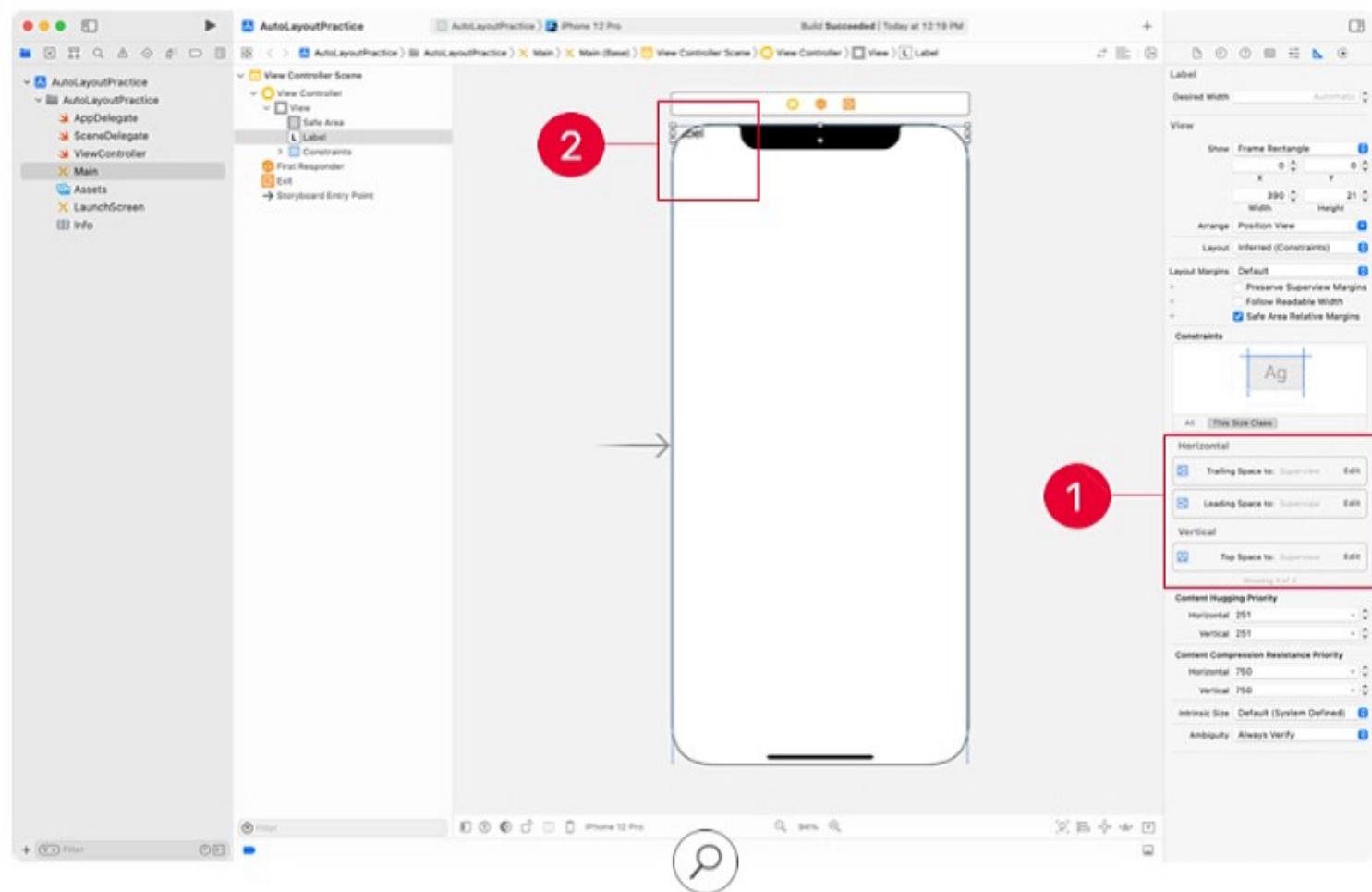
Notice that the button has now expanded to be 20 pixels from each edge of the screen. Use the “View as” button to select different devices and orientations. You’ll see that the width of the button varies such that it always maintains the same distance from the edges of the screen. If for some reason the button didn’t update, update the button’s frame using the Update Frames tool .

Safe Area Layout Guide

Remove the button from the screen by selecting it and pressing the Delete key. Add a label from the Object library to the top-left corner of the screen. ① Using the Add New Constraints tool, create constraints with values of 0 on the top, left, and right of the label to the view. ②

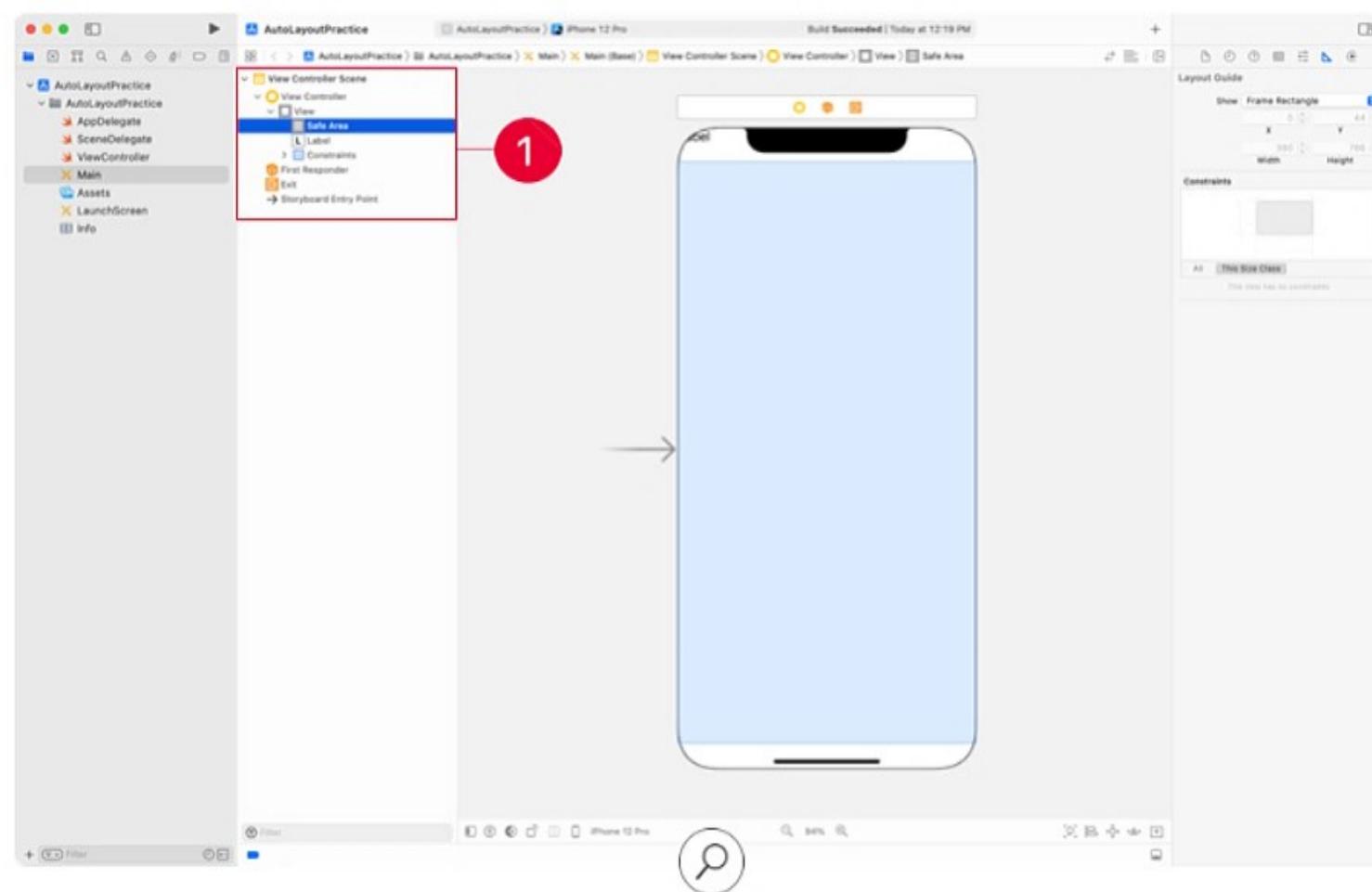


Now select the label and open the Size inspector. You can view the three constraints that you just added. ①



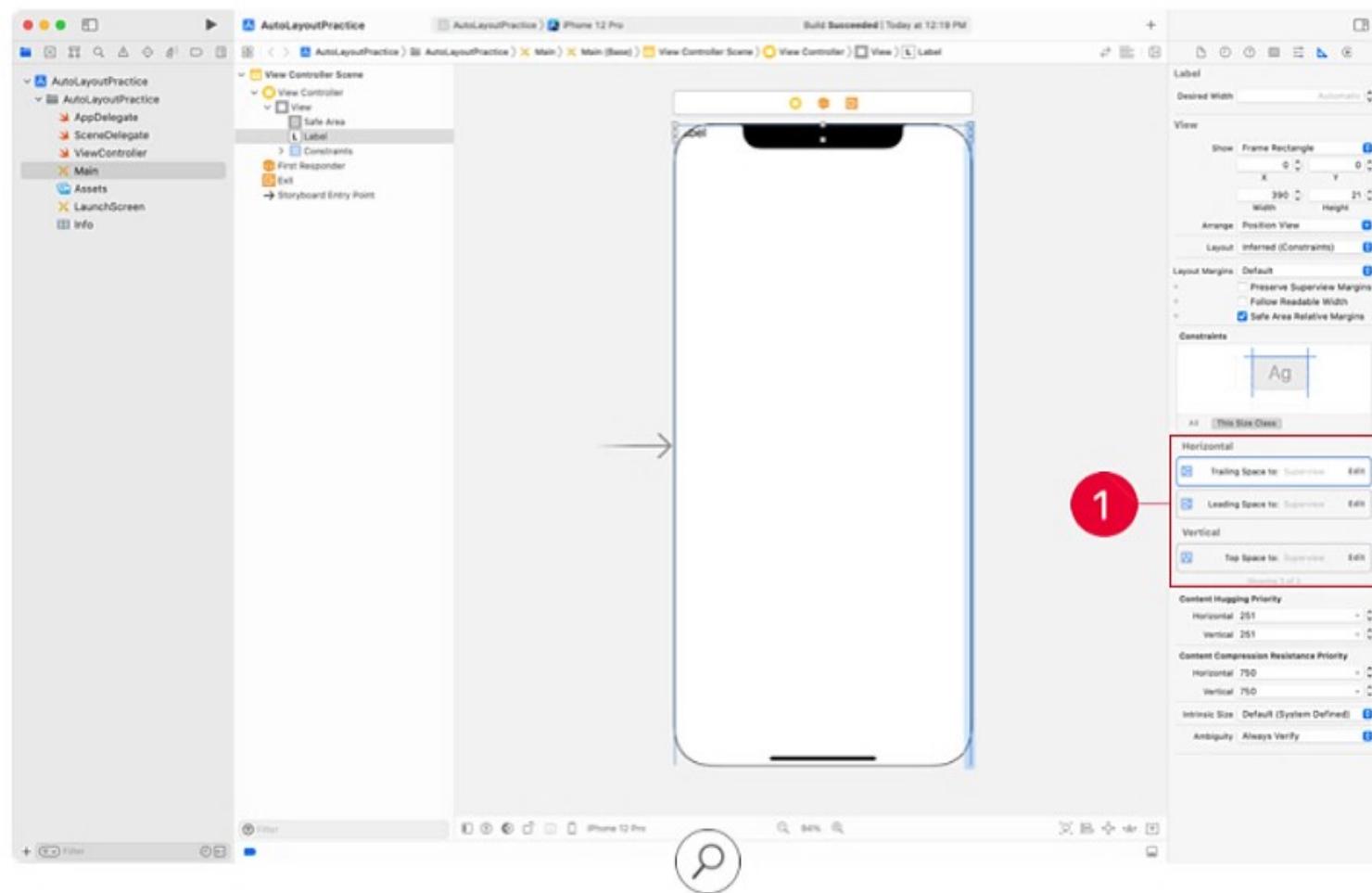
Notice that the constraints are between the label (which you have selected) and the Superview. This means the edges of the label are constrained to the edges of the main view controller's view. You can see in Interface Builder that this places the label under the status bar and obscures some of the label's text. ② You may be tempted to simply change the value of the top constraint so that the distance from the top is equal to the height of the status bar. However, what if later that view controller is embedded in a navigation controller? The label would no longer be covered by the status bar, but it would then be covered by a navigation bar.

To fix this, you'll need to create constraints relative to the Safe Area of your view controllers. The Safe Area is displayed in Interface Builder as a subview of the view controller's primary view.^①

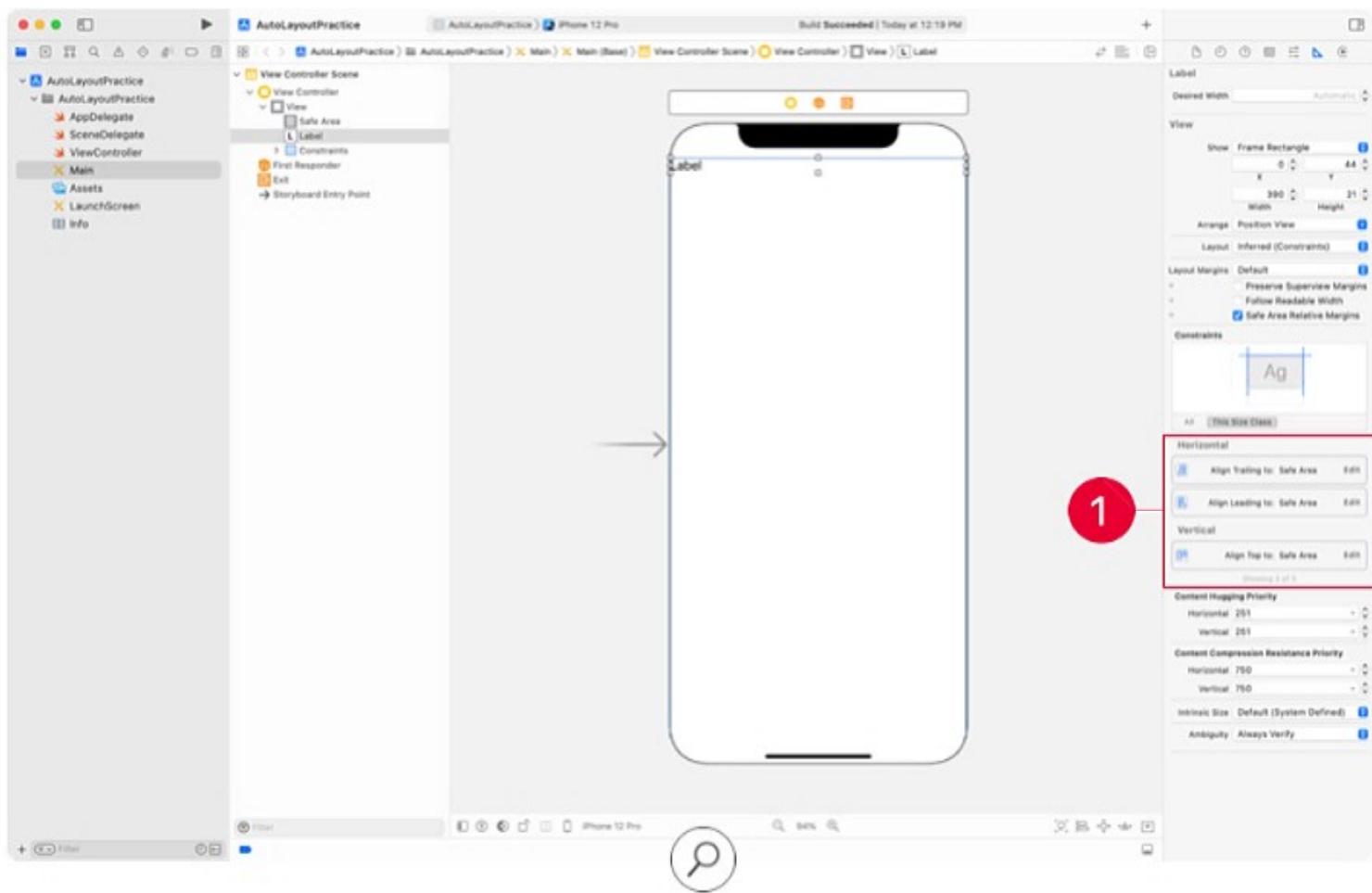


If you select the Safe Area, the entire view controller is selected except for portions of the screen that are taken up by system views like the status bar and home indicator. Interface Builder anticipates the existence of these common system views. However, other system bars can turn up at runtime, and the Safe Area will adjust for those as needed. This allows your content to adapt to system overlays such that it won't be hidden.

Delete your current constraints by selecting them one at a time in the Size inspector and pressing the Delete key. ①



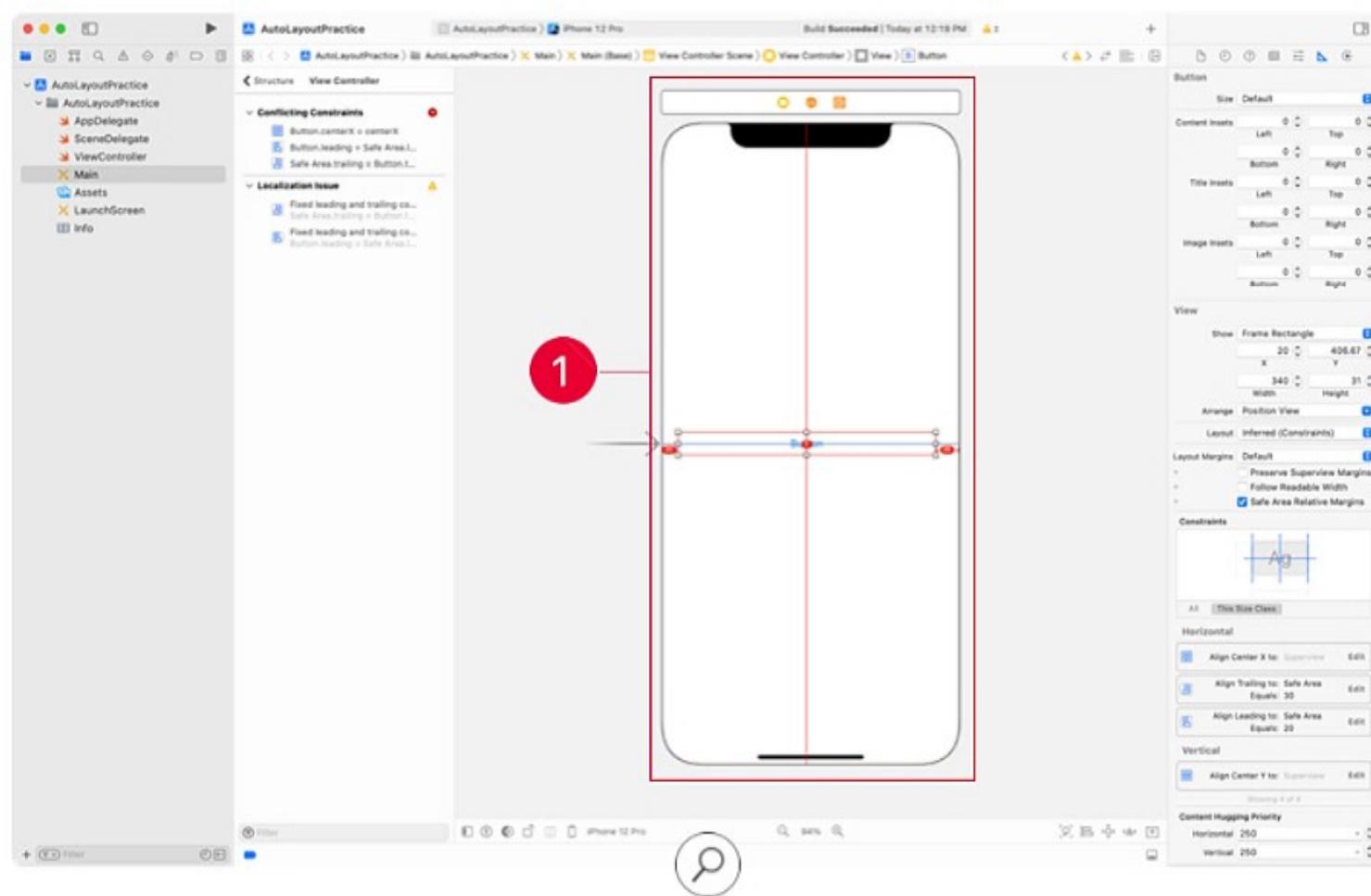
Drag your label a bit lower on the screen so it is below the status bar, and create a constraint between the label's top edge and the top edge of the Safe Area using the Add New Constraints tool. Also add constraints for the leading and trailing edges of the label. Since the Add New Constraints tool constrains your view to the nearest view, creating these constraints with the tool will constrain the top, leading, and trailing edges of the label relative to the Safe Area. ①



Resolve Constraint Issues

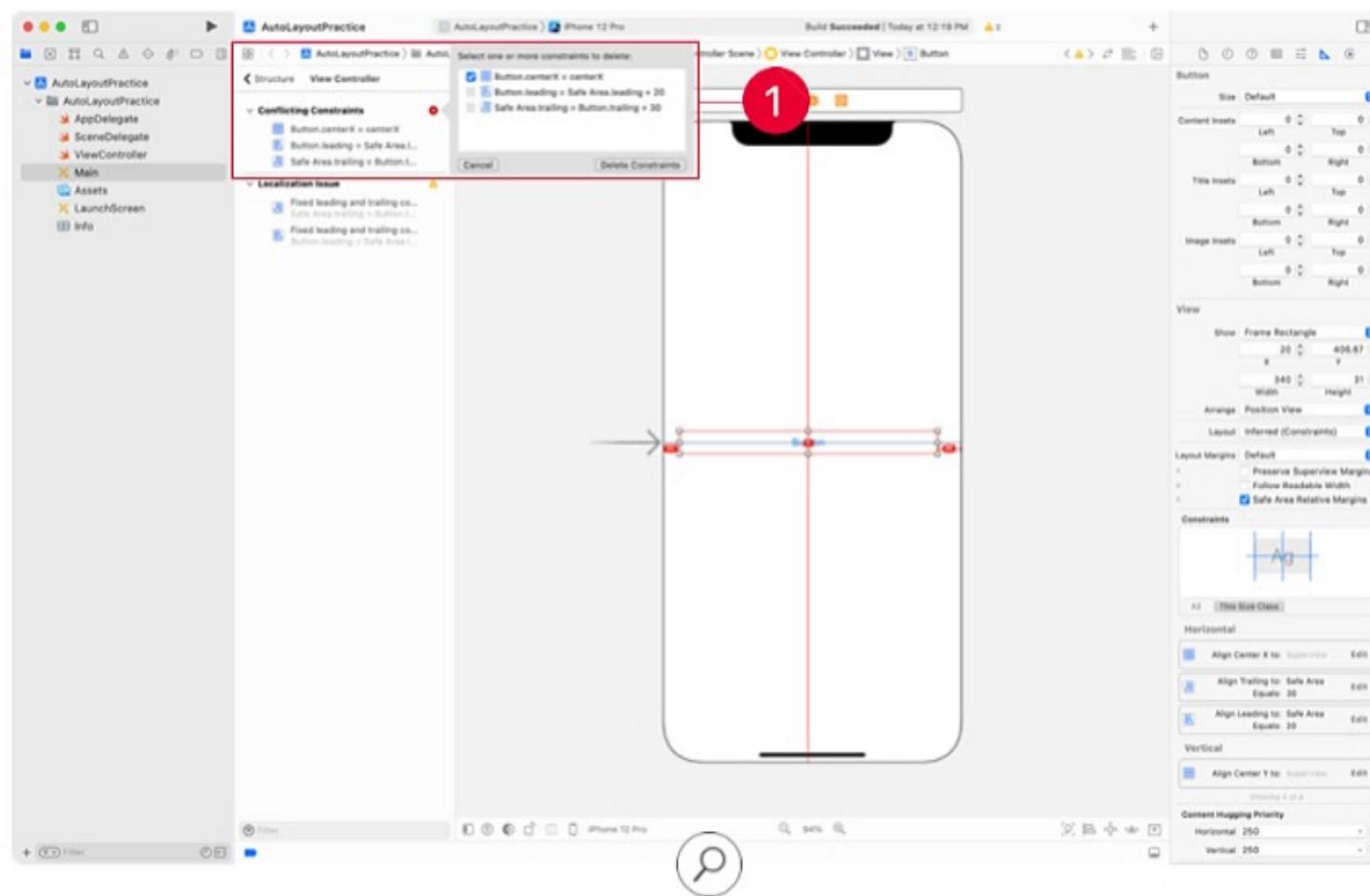
Now delete the label that you added. Add a button to the center of the view controller and give it two alignment constraints – one to center it vertically on the screen, and one to center it horizontally on the screen. Now use the Add New Constraints tool to constrain its left edge 20 pixels from the left edge of the screen, and to constrain its right edge 30 pixels from the right edge of the screen.

The red lines on the canvas indicate that there's a problem. ① You've now created two conflicting constraints, each of which is trying to dictate the X position of the button. To see a list of constraint conflicts, click the red indicator in the top-right corner of the Document Outline.



One of the constraints says, "Set the horizontal center of the button equal to the horizontal center of the view"; and the other says, "The left edge of the button is 20 pixels away from the left edge of the view." Since these two constraints can't coexist, you'll need to remove one.

From the Conflicting Constraints list, click the red error indicator at the top right. Select the constraint that centers the button, and then click Delete Constraints. ①



From time to time, you may run into a constraint issue that's tricky to resolve. Or maybe you're adding interface elements to the scene and you need to reconfigure the view's layout. In either situation, you can click the Resolve Auto Layout Issues button  , next to the Add New Constraints tool, and use the options to try to automatically address constraint issues.

Update Constraint Constants will attempt to update the rules to match what is currently displayed in the storyboard scene.

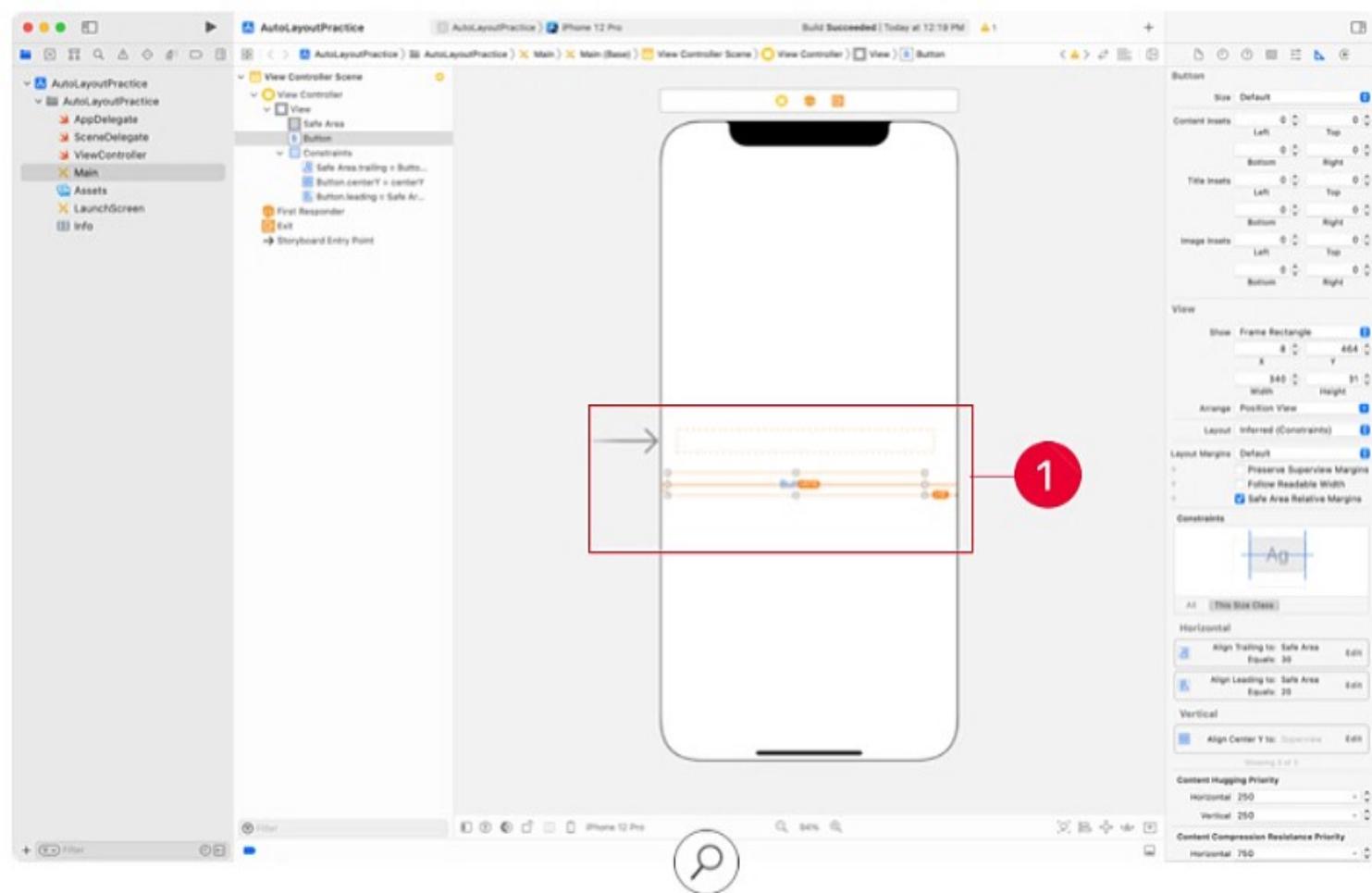
Add Missing Constraints will attempt to add new constraints that match what is currently displayed in the storyboard scene.

Reset to Suggested Constraints will clear all constraints and attempt to accurately assign new constraints that match what is currently displayed in the storyboard scene.

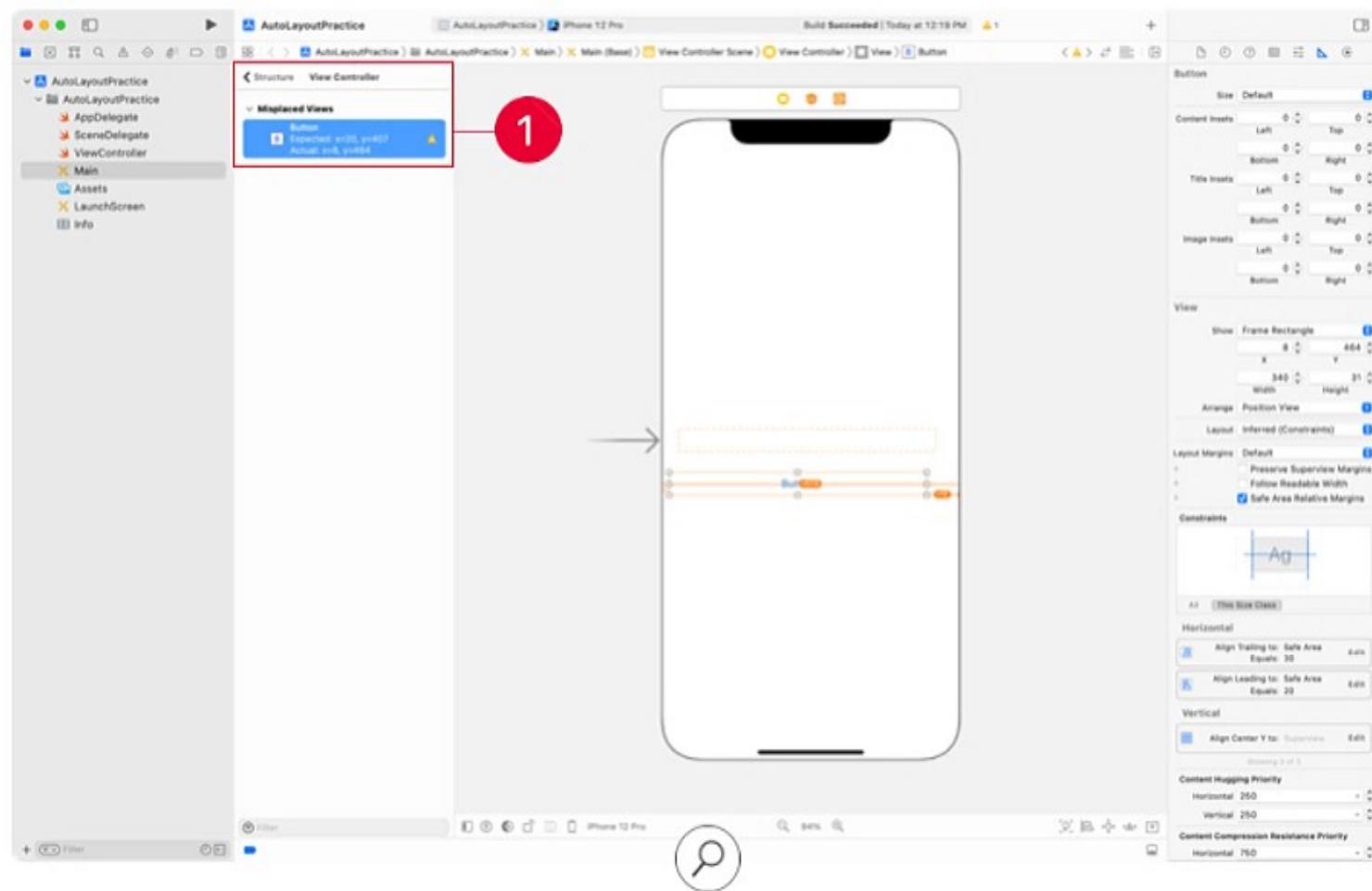
Clear Constraints will remove any rules associated with the position and size of the selected view or all views.

Resolve Constraint Warnings

Move the button you've been working with to a different position in the scene. You should see an Auto Layout warning. 



To understand what's happening, click the yellow indicator in the top-right corner of the Document Outline. The warning informs you that the position of the button is no longer in sync with the position defined by the constraints.^①

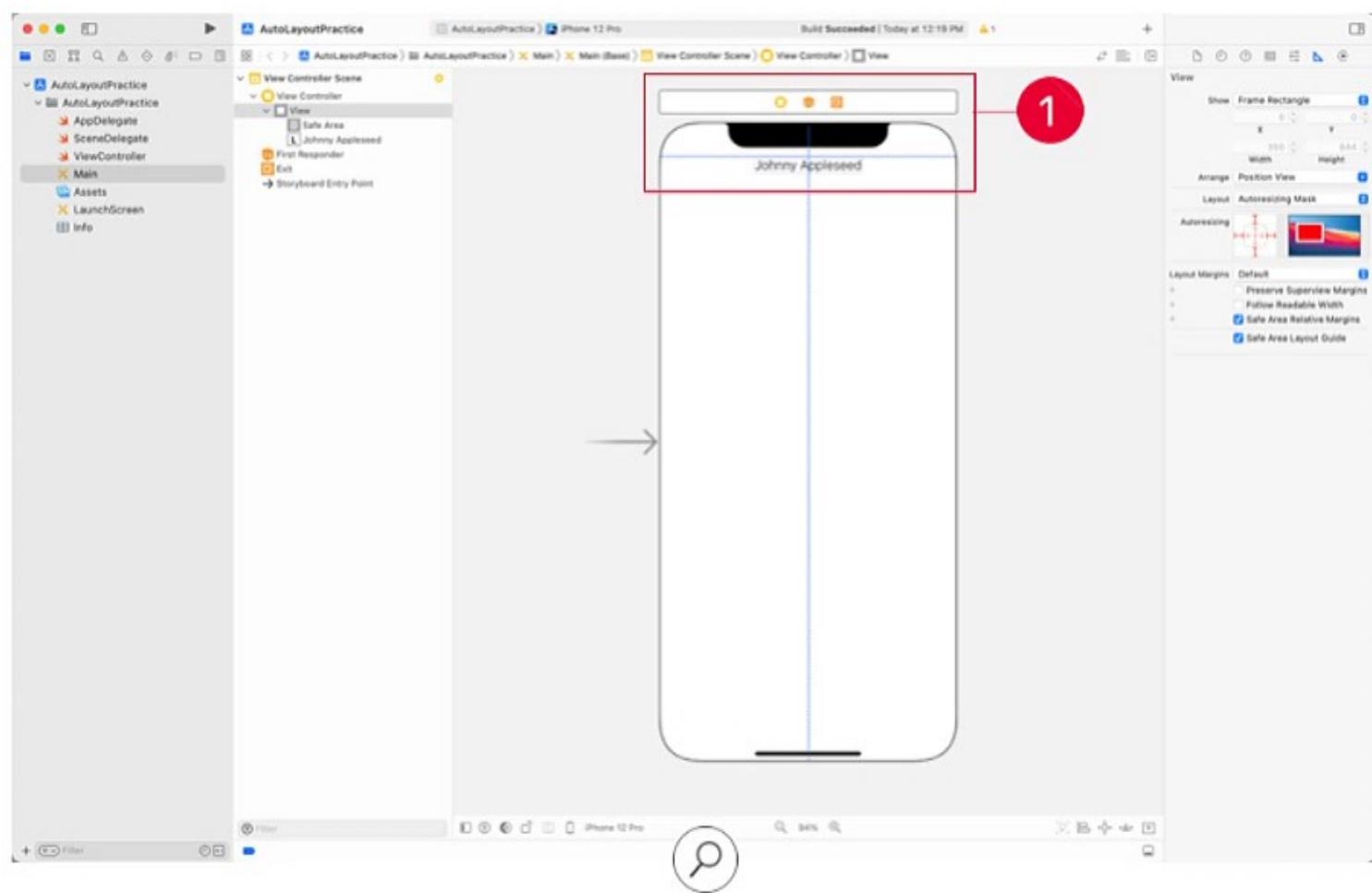


To adjust the button's position to fit the constraints, select the button on the canvas, then click the Update Frames button [?] .

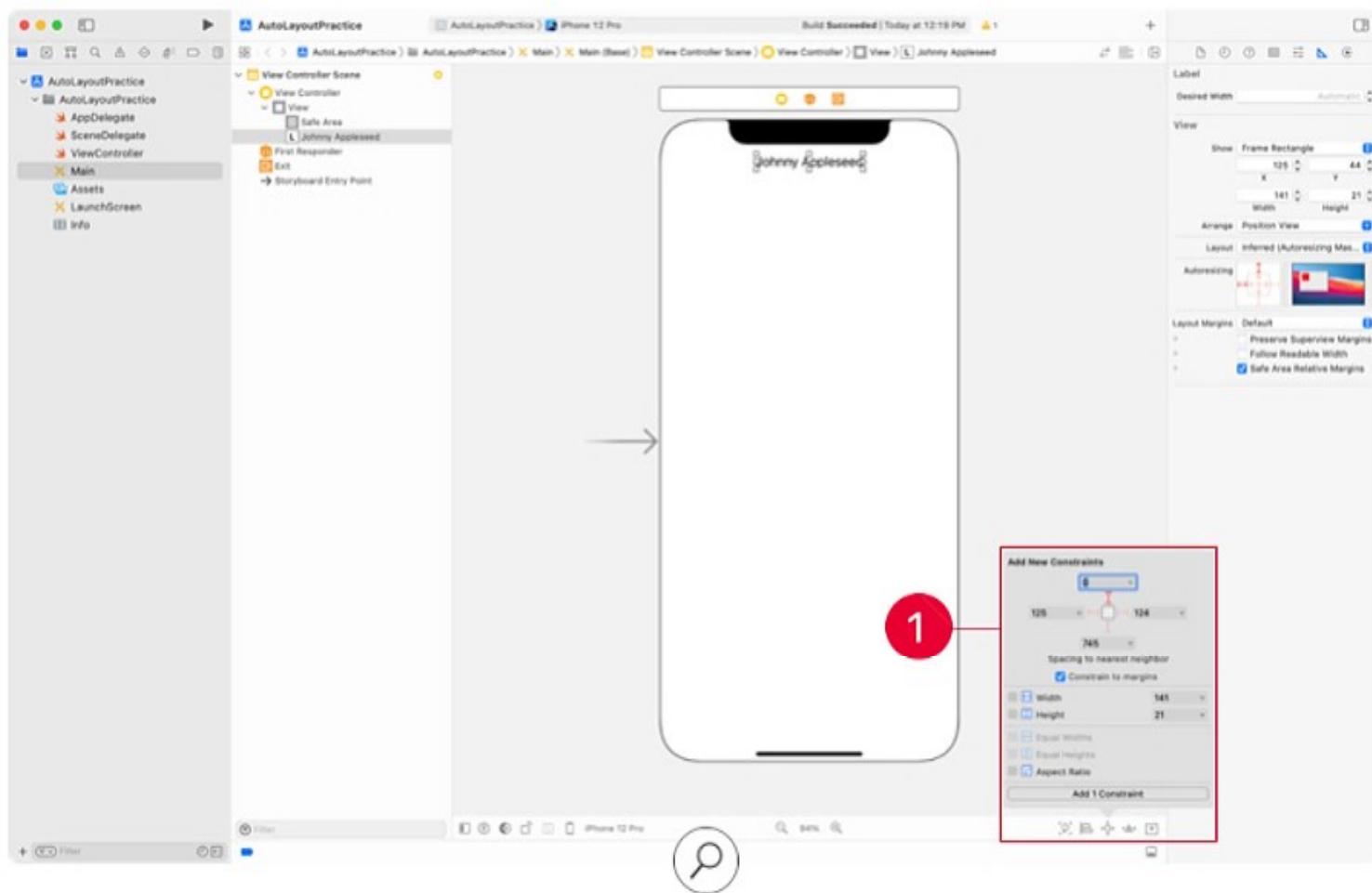
Constraints Between Siblings

So far in this lesson, you've learned how to create constraints that define relationships between a view and its parent view. But there may also be times you want to create constraints between sibling views. In the following exercise, you'll work with multiple labels that display text information about yourself.

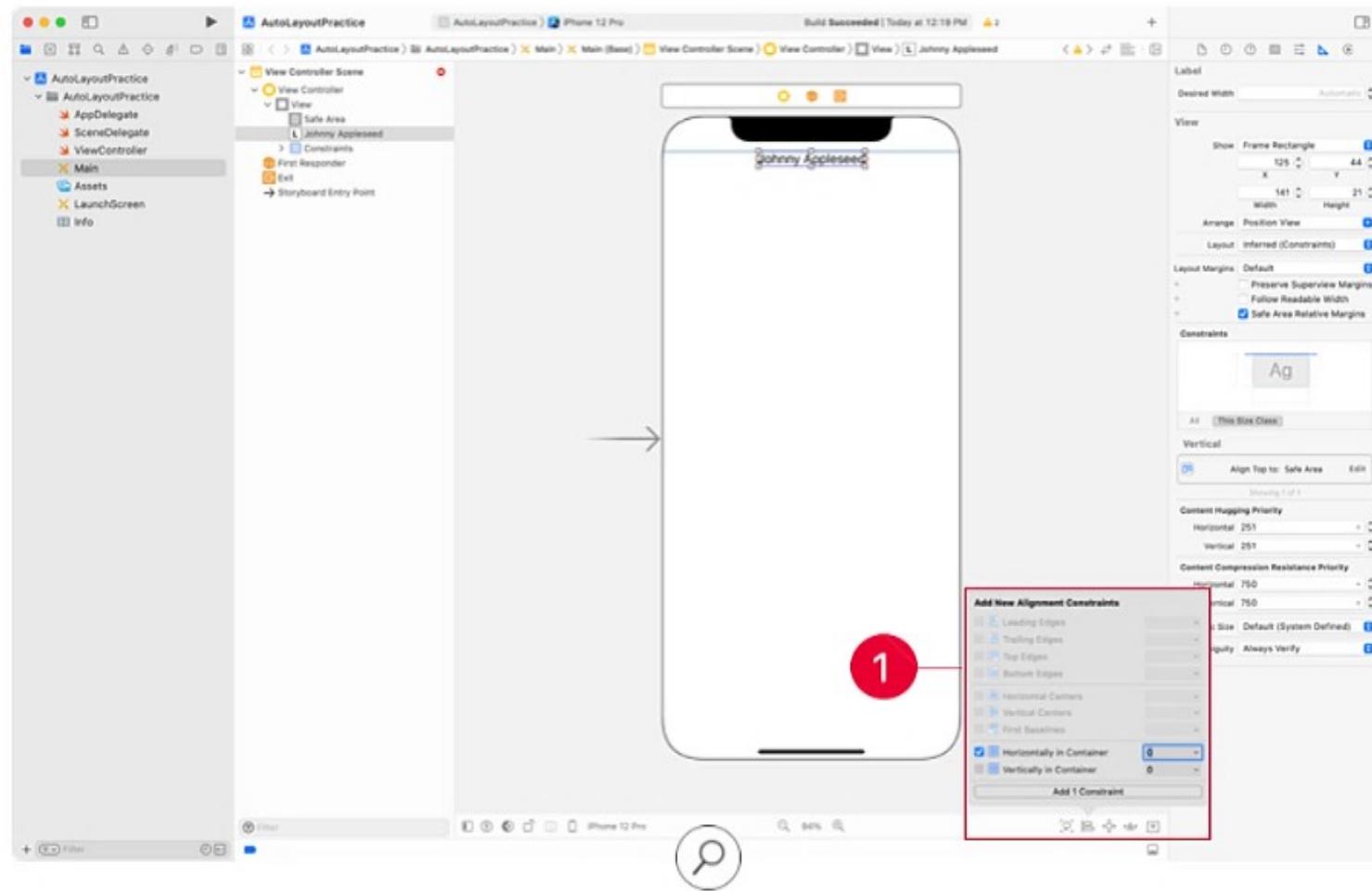
Delete the button you've been working with, and add a label to the scene. Double-click the label and type in your name. Drag the label near the top of the view, using the guides to center it horizontally. ①



With the label selected, use the Add New Constraints tool to constrain it to be 0 pixels from the top of the Safe Area. To enable the constraint, you may need to select the red indicator line below the top field in the popover. ①



Use the Align tool to align the horizontal center of the label with the center of the parent view. ①



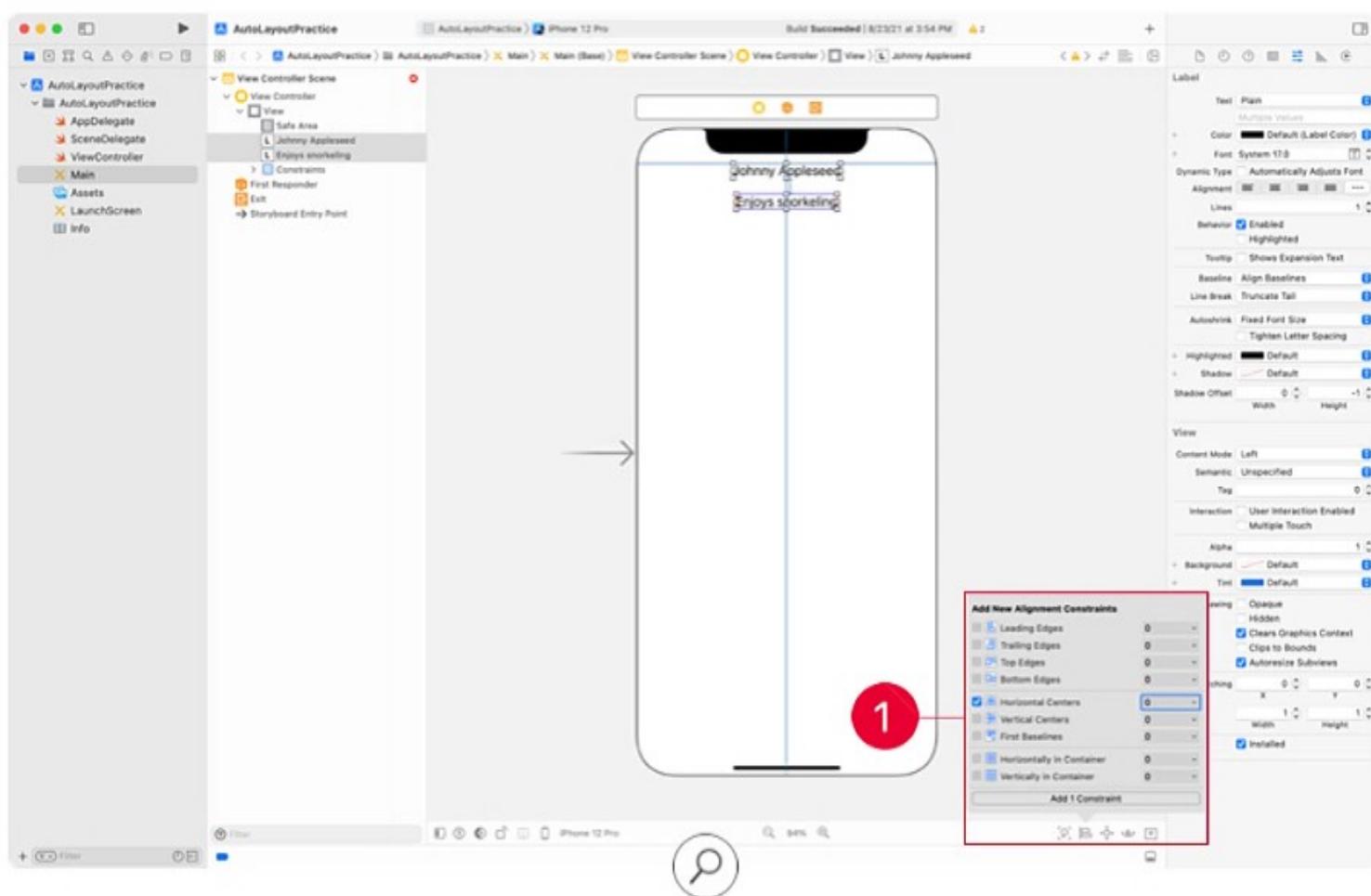
With one label set, you can add a second label and base its position on the position of its sibling. Drag another label from the Object library onto the view, just below the existing label. Enter some information about yourself, perhaps one of your favorite hobbies, into the new label's text.

Now assume you want the new label to have the same horizontal center as the first label and to be positioned 20 pixels below it. With the new label selected, click the Add New Constraints button. In the popover, enter 20 into the top text field, and click "Add 1 Constraint."

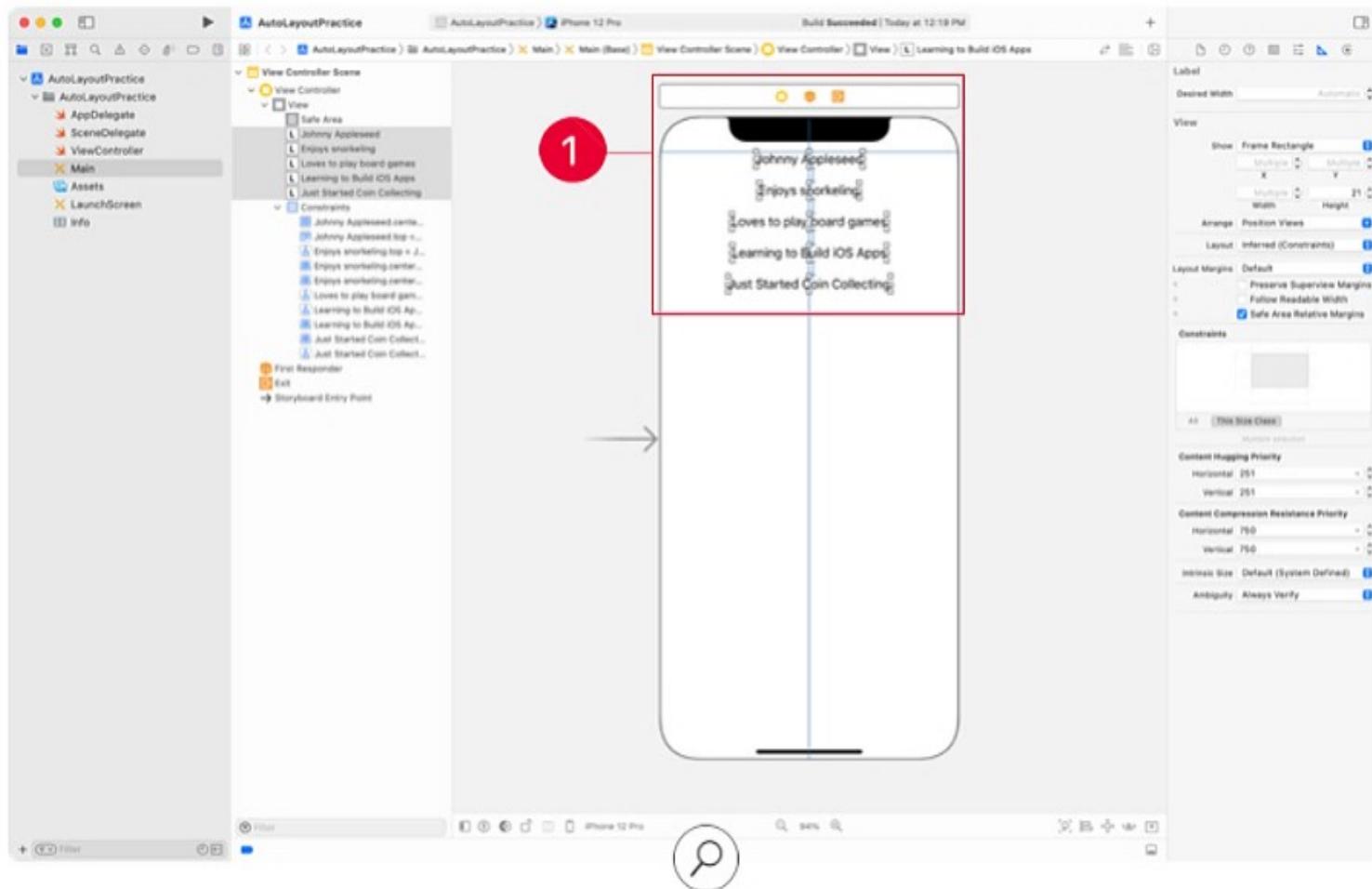
At this point, Interface Builder will display an error, indicating that the new label has a constraint describing its vertical position, but no information about its horizontal position. Interface Builder can help you resolve this error.

Select both labels in the scene by Command-clicking each of them, and click the Align button. In the popover, select the Horizontal Centers checkbox, set its value to 0, and click “Add 1 Constraint.” ^① This tells Interface Builder that you want the selected views to have the same center position, with 0 pixels of offset.

If Interface Builder still displays a warning, click the Update Frames button to update the labels’ frames to match the constraints you just specified.



To practice what you just learned, add three new labels and repeat the above steps three more times to position them on the scene. When you're done, each label should be 20 pixels below the previous one, and they all should be centered horizontally. ①

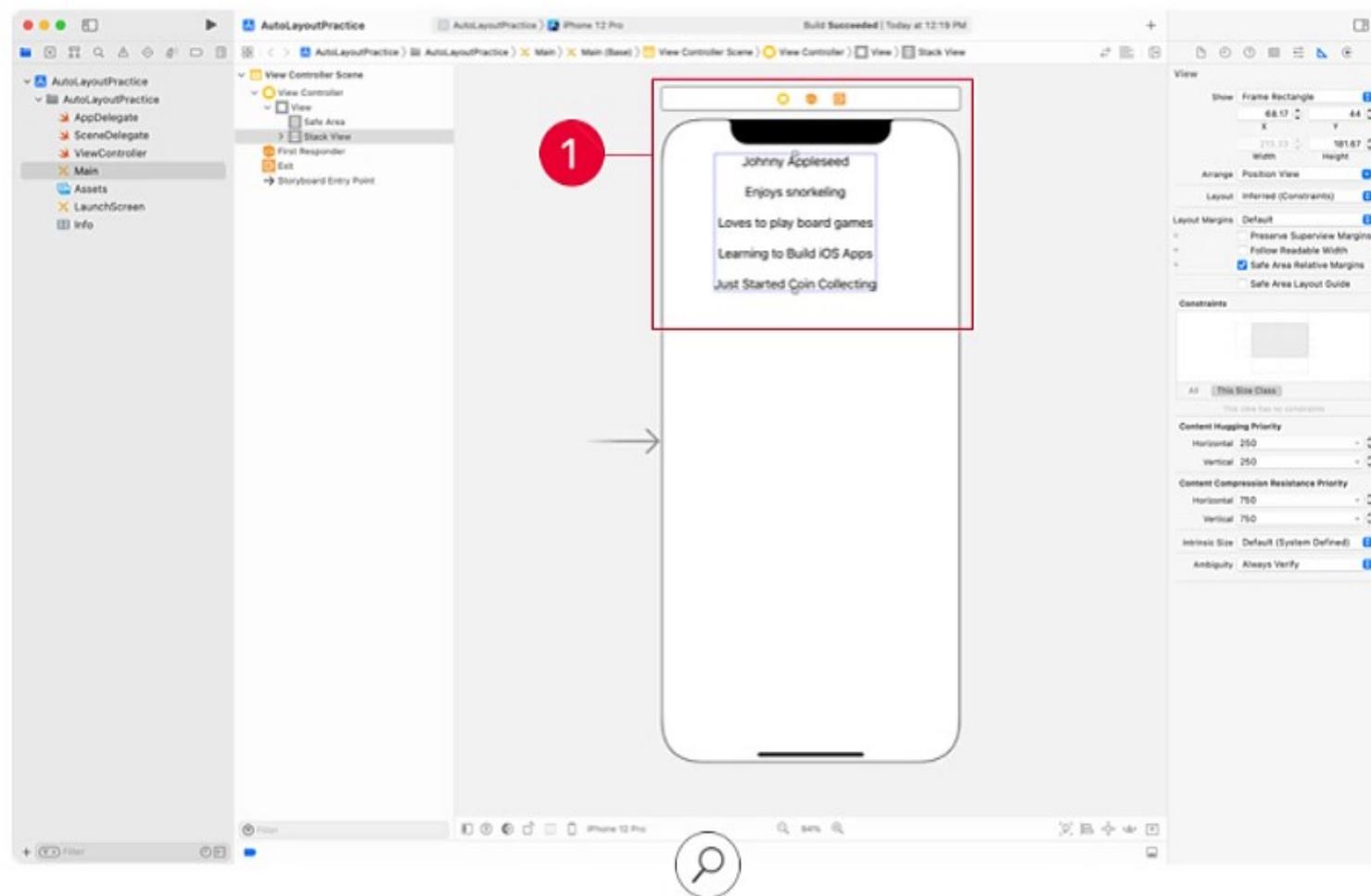


Stack Views

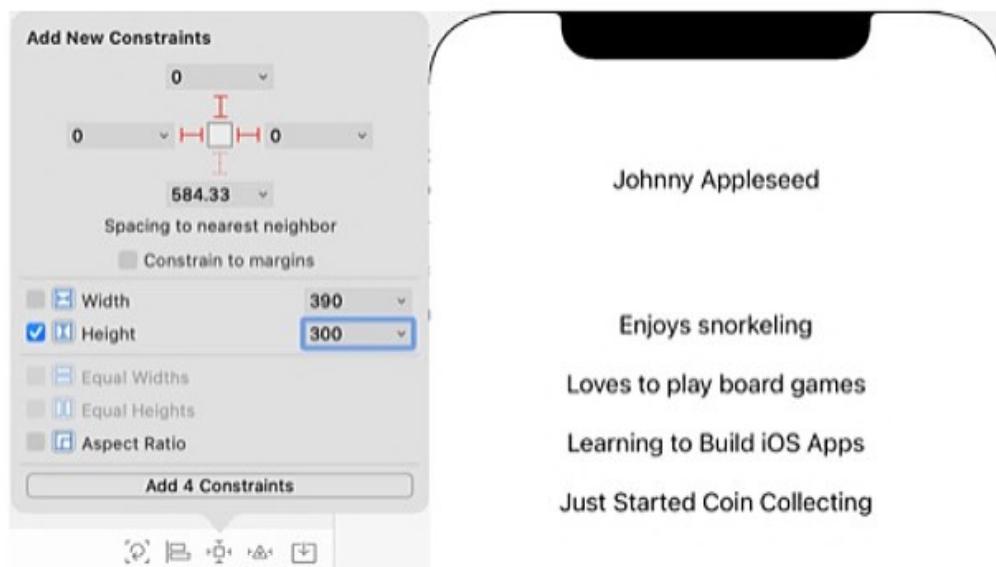
Did you find that last exercise a bit repetitive? Managing the constraints of multiple objects can be tedious, especially when the position of each element is the same distance from the previous one. What if you decide later to have 30 pixels of separation between labels instead of 20? Or what if you want all of the labels to be positioned on the left side of the parent instead of in the center? You would have to spend a lot of time updating each and every constraint.

UIKit provides a smarter approach. Rather than create and update lots of individual constraints, you can use a stack view to automatically manage the constraints of its child views.

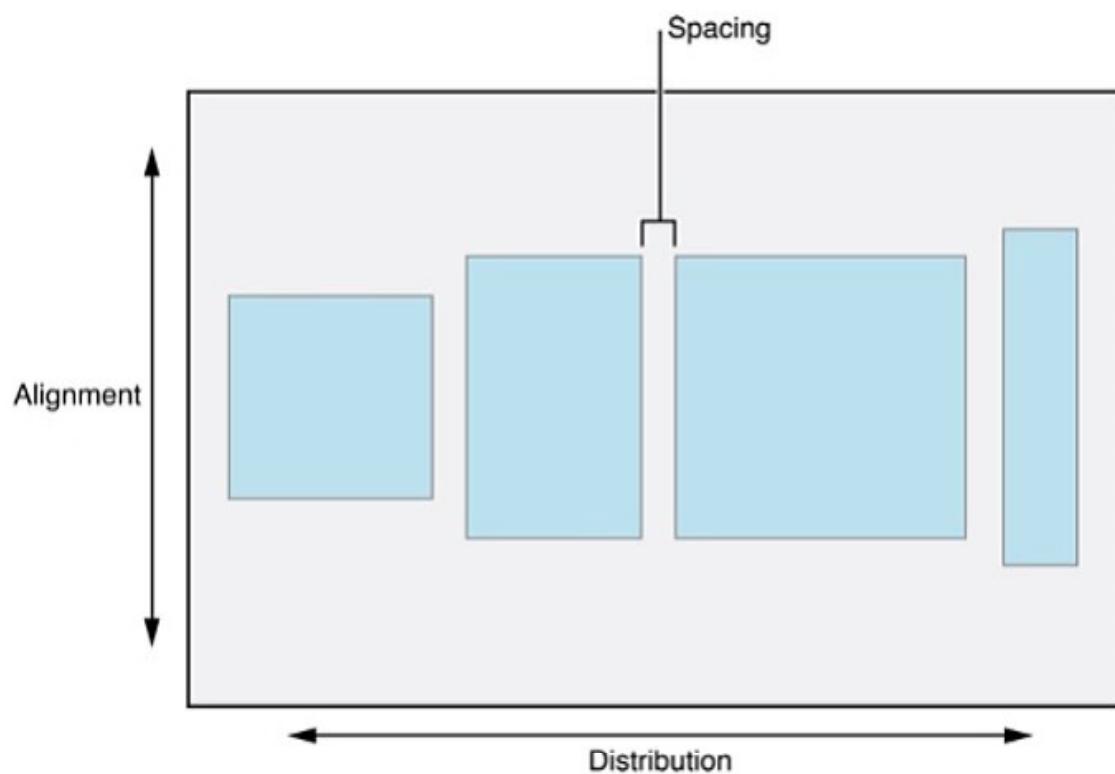
A single stack view manages either a row or a column of interface elements, and arranges those elements based on the properties set on the stack view. Start by selecting all the labels you just created, then click the Embed In Stack button  to the right of the Resolve Auto Layout Issues tool and choose Stack View from the list that appears. This organizes the selected views into a single stack and removes any constraints on the individual labels. ①



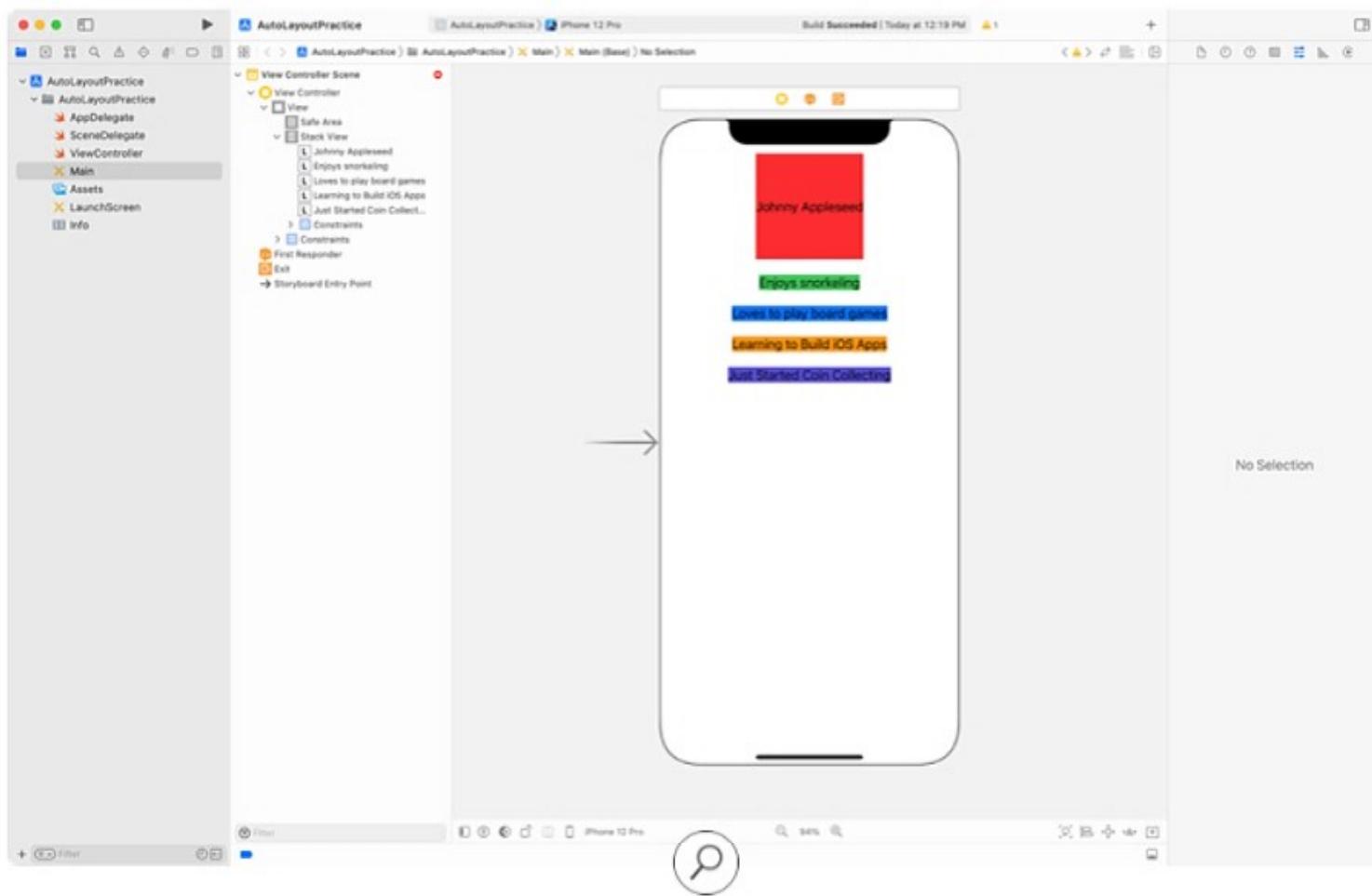
At the moment, your new stack view doesn't have any constraints that define its size or position. To change that, select the stack view, then click the Add New Constraints button. Set the top, leading, and trailing edges of the stack view to 0 pixels from the parent view's edges. (You'll need to select the red indicators associated with each text field to enable the constraints.) Give the stack view a constraint height of 300. Click "Add 4 Constraints," and click the Update Frames button, if necessary.



Stack View Attributes



Now that you have all your labels in a stack, you can learn how to manage the stack view and define the positions of its subviews. But before you begin, set the background of each label to a unique color. This will make it clear how your adjustments are affecting the interface.



Select your stack view from the Document Outline, and open the Attributes inspector to reveal four main properties. Go ahead and explore these attributes, making adjustments along the way.

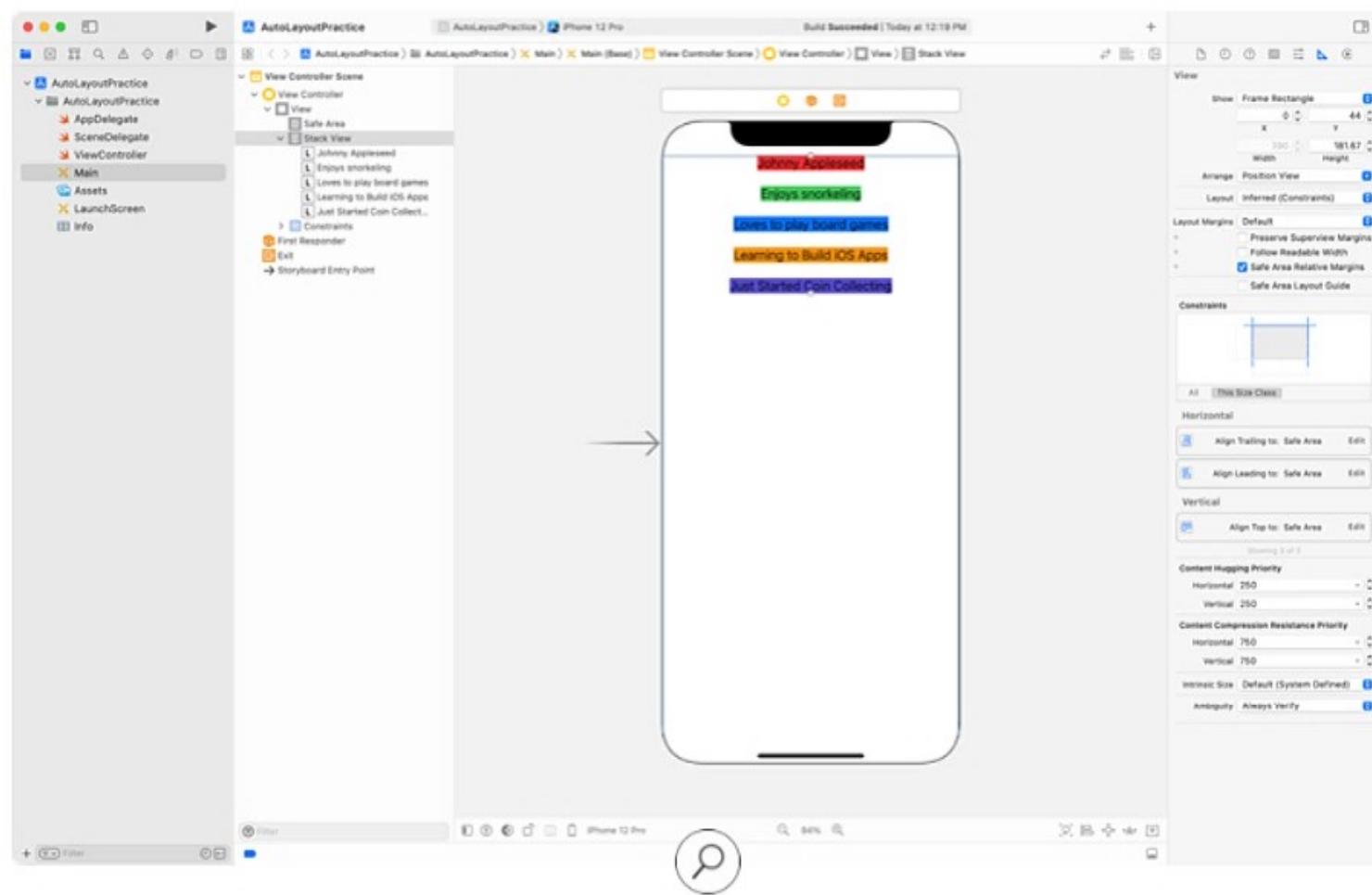
1. Axis determines whether the elements within the view are stacked vertically or horizontally.
2. Alignment describes how the elements within the stack are positioned. You can choose one of the following:
 - Fill—Each of the elements fills the size of the stack. For example, if the stack view has a width of 320 pixels, each element in a vertical stack will also have a width of 320 pixels.
 - Leading—The leading edge of each element is aligned to the leading edge of the stack view.
 - Center—The center of each element is aligned to the center of the stack view.
 - Trailing—The trailing edge of each element is aligned to the trailing edge of the stack view.

3. Distribution defines how the elements are distributed within the stack view. You can choose one of the following:

- Fill—The stack view resizes the arranged subviews so that they fill all the available space along the axis you specified. The stack view sizes the first label as large as possible to completely fill the stack.
- Fill Equally—The stack view resizes the arranged subviews so that they fill all the available space along the axis. The views are resized so that they are all the same size along the stack view's axis.
- Fill Proportionally—The stack view resizes the arranged subviews so that they fill all the available space along the axis. If the size of the stack view changes, the views are resized proportionally to one another. For example, if two labels have heights of 50 and 100, respectively, and you increase the stack view's height by 50 percent, the labels then have heights of 75 and 150.
- Equal Spacing—The stack view doesn't resize the subviews but positions them at an equal distance from one another.
- Equal Centering—The stack view doesn't resize the subviews, but ensures that the center of each subview is an equal distance to the centers of the other subviews.

4. Spacing describes the amount of space between each element in the stack view. For example, you can change the spacing from 0 to 20 to add 20 points of spacing between labels.

Use the Size inspector to remove the height constraint on the stack view. If a stack view doesn't have a specified height, it will resize based on its subviews.



Now that you've defined some attributes, try adding new labels to the stack. Or rearrange the existing labels and adjust the stack's spacing and alignment. Imagine if you hadn't used a stack view and you wanted to adjust the interface. How many constraints would you have needed to add, remove, or adjust?

Whenever possible, it's a good practice to use stack views before trying to manage constraints individually. Stack views allow you to create nice-looking interfaces quickly—and also make it easy to modify or customize them in the future.

Size Classes

With so many different combinations of screen sizes and orientations, it's important to build an interface that works well on all iOS devices. In the case of iPad devices, the split-screen feature adds an additional layer of complexity because an app may run in a smaller amount of space than you originally expected. To help simplify user interface development for many different situations, iOS includes size classes.

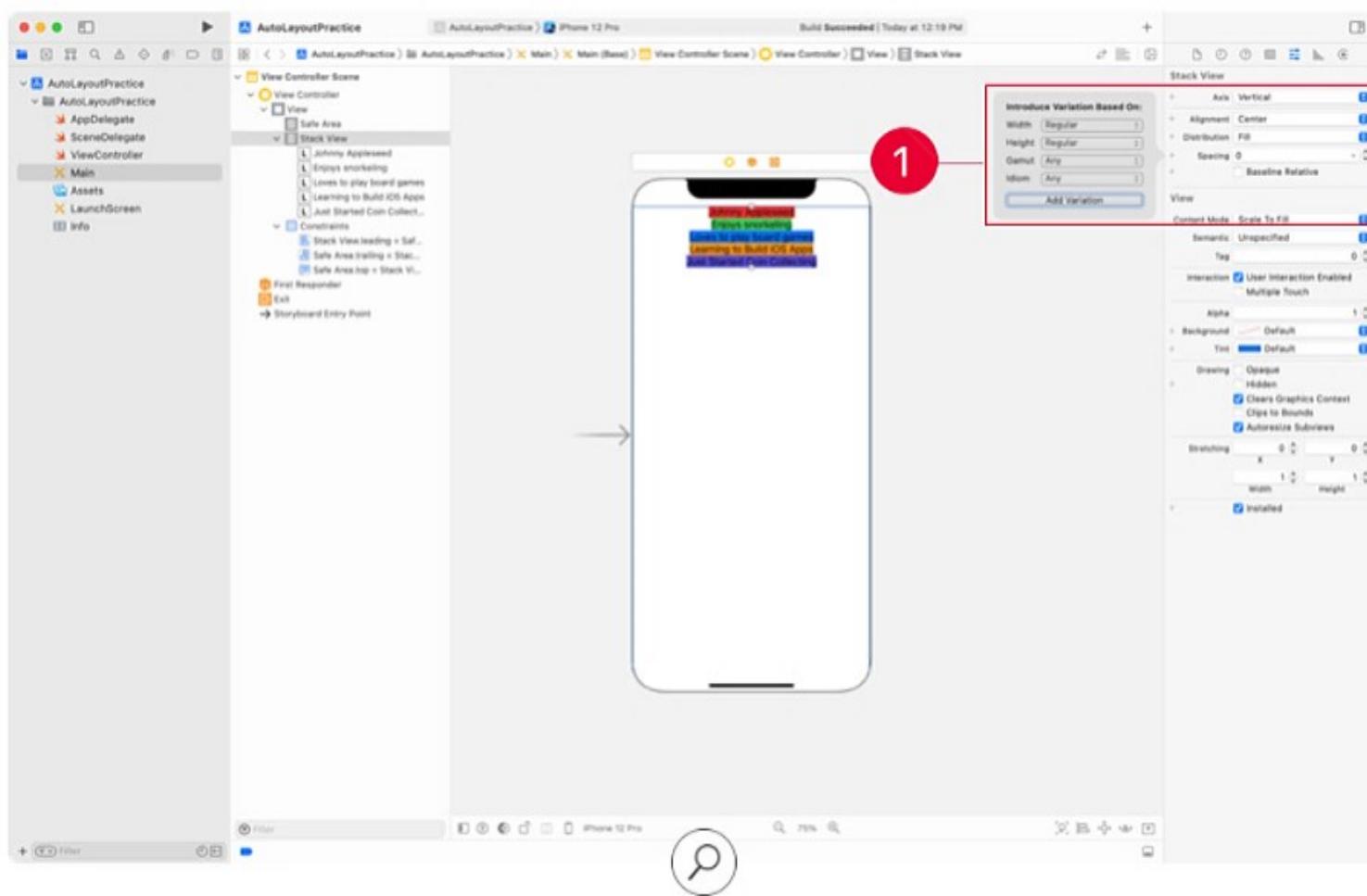
Each iOS device has a default set of size classes that you can use as a guide when designing your interface. Each dimension, the width and the height, is either the Compact or Regular size class, and the two dimensions form a trait collection. The iPhone 13 Pro that you have been using has a trait collection that is Compact Width, Regular Height in portrait orientation but in Landscape orientation is Compact Width, Compact Height.

Though an iPad has a Regular Width, Regular Height size class in both portrait and landscape, an app does not always run full-screen. Using the split-screen feature, an app can consume one third, one half, or two thirds of the screen real estate, and different-sized iPads will have unique trait collections when multi-tasking. It sounds like a lot to handle, but don't think about the wide array of devices you need to support. Instead, focus on supporting the four different width/height trait collections, and your interface will be correct.

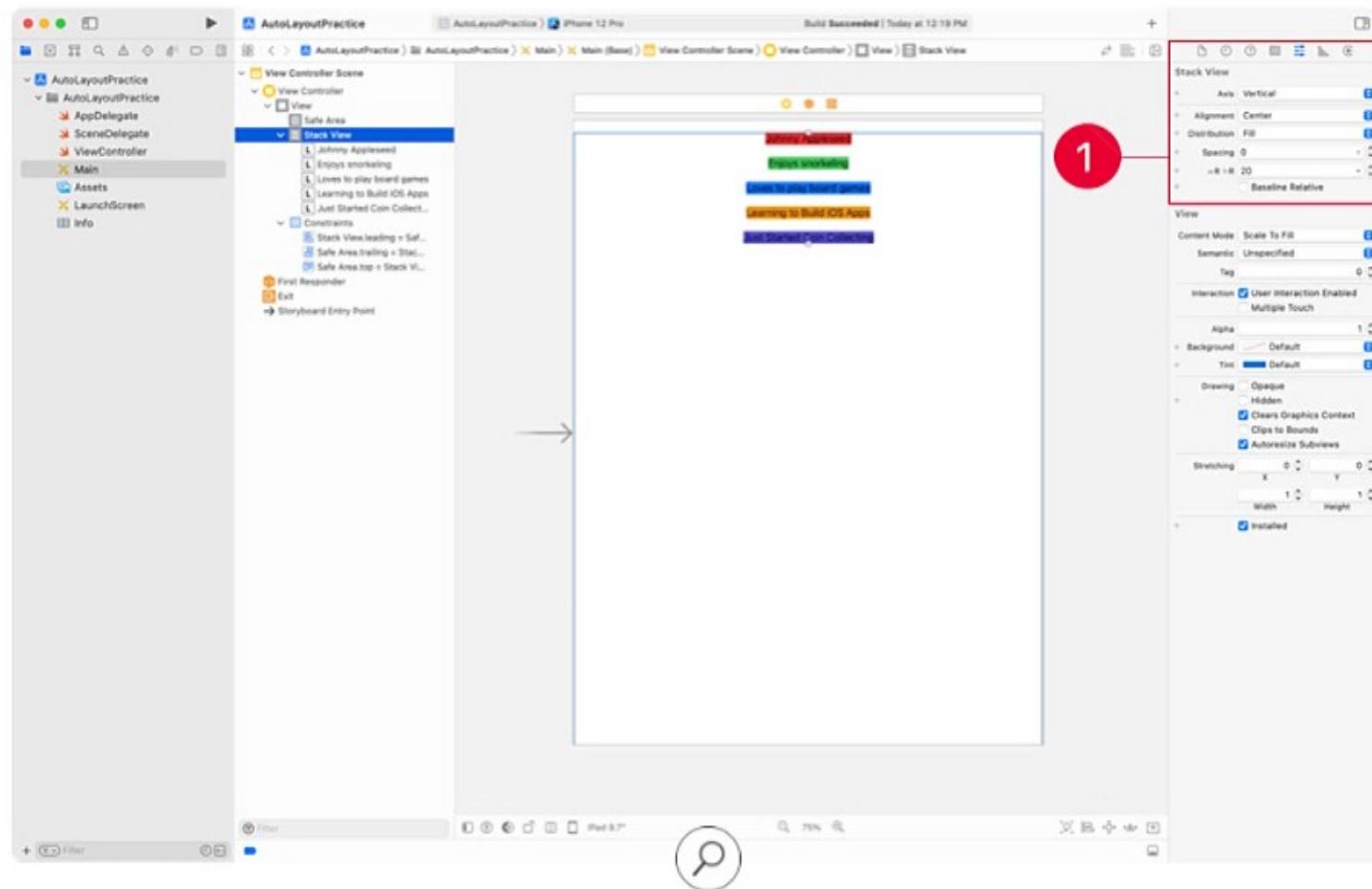
The Layout  button will be enabled for devices that support split-screen operation. The layout options include full screen along with split views at one half, one third and two thirds of the screen to help layout an interface that will work well in all situations.

Vary Traits

Constraints and properties can be set to different values depending on the trait collection. Suppose you want a stack view's Spacing property to be 0 most of the time. However, when run on a Regular Width, Regular Height device, it should be set to 20. With Interface Builder, you can easily add varying traits for a particular property or constraint. Using the stack view that you previously created with colored labels, select the stack and open the Attributes inspector. Set its Spacing to 0, and then click the '+' button, which opens a popover for introducing variations. ①



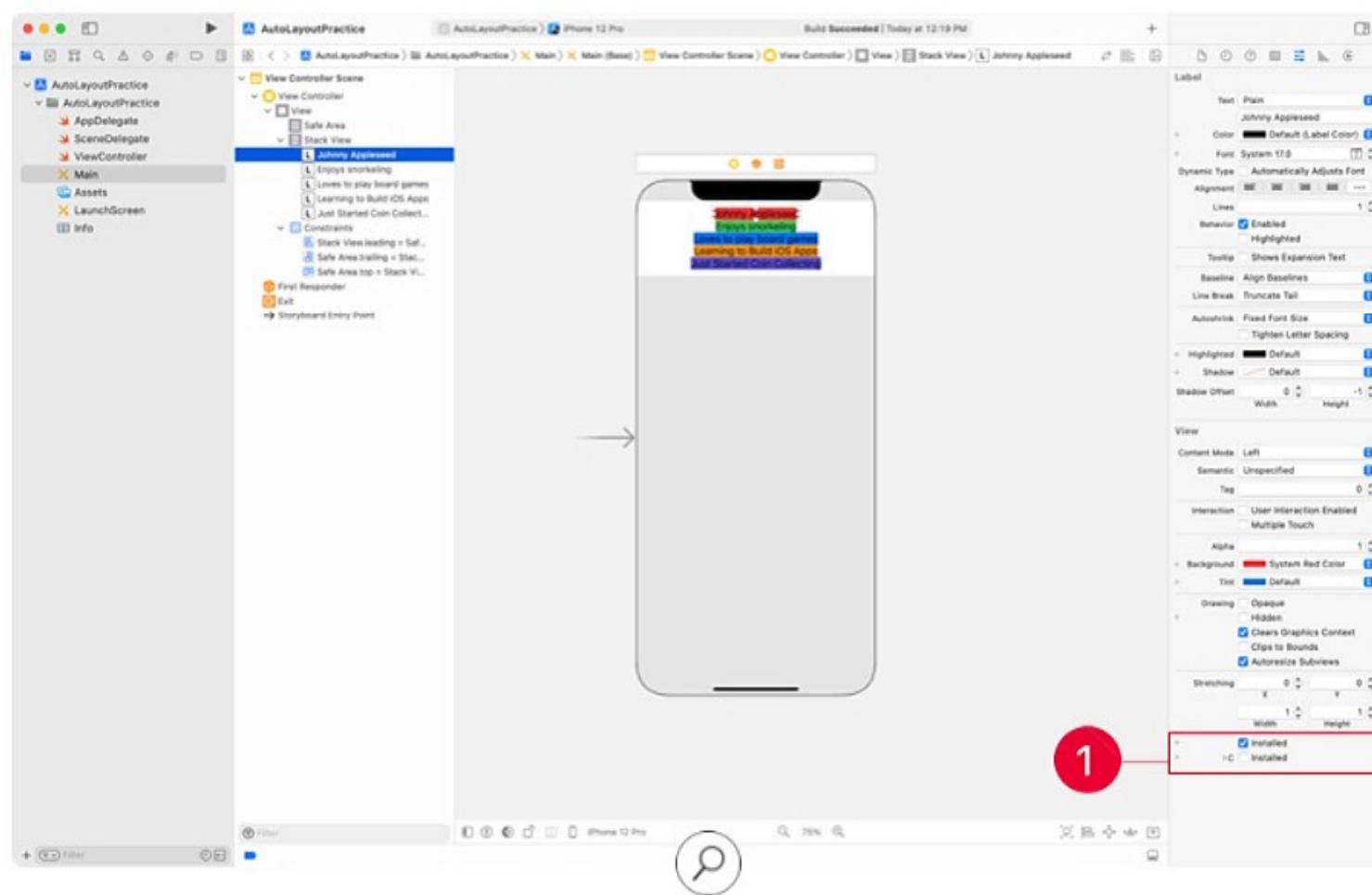
Since you want to add a new variation for the Regular Width, Regular Height trait collection, set the appropriate controls to these values, then select “Add Variation.” This will add a new Spacing property that is unique for this particular trait collection. Change this new Spacing to 20. ①



To verify that this trait variation is working properly, click the "Devices" button and change the device to an iPad (which is Regular Width, Regular Height). You should notice additional spacing between each label. When you return to an iPhone 13 Pro, the label spacing diminishes.

Installed

Sometimes you'll want to disable particular views or constraints depending on the trait collection. For example, maybe in order to save space, your app's interface should not include the red label when the trait collection's height is set to Compact. To begin implementing this design decision, select the red label and open the Attributes inspector. Near the bottom, you'll find an "Installed" checkbox. This determines whether the particular view or constraint exists within the view hierarchy. Click the '+' button next to the checkbox, and you'll be presented with the same popover to introduce a variation. Set the Width to Any and the Height to Compact, then click "Add Variation." This will add a new Installed property that spans any trait collection that includes a Compact Height.¹

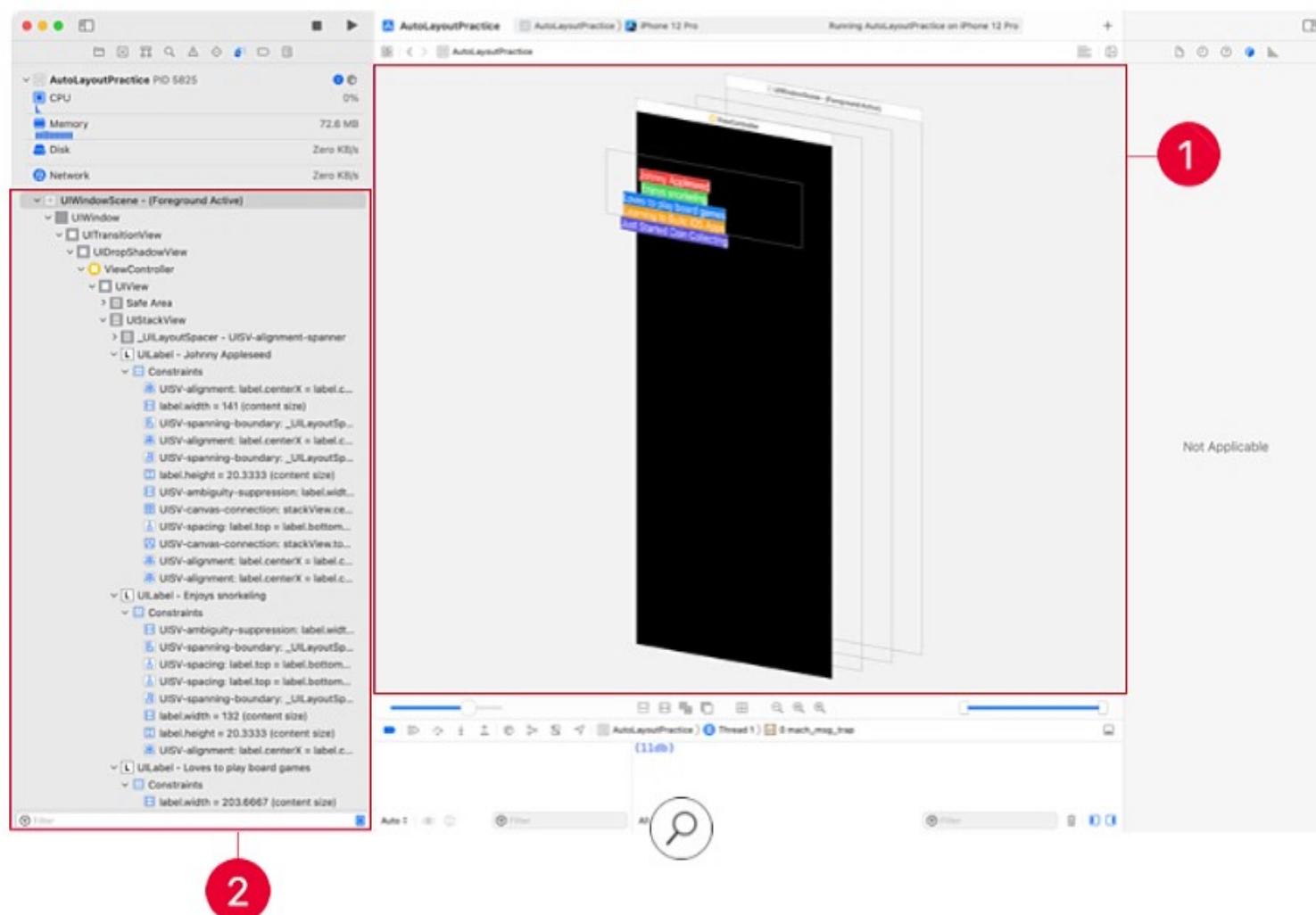


Deselect the checkbox for this new property, and build and run your application using the iPhone 12 Pro or iPhone 13 Pro Simulator. When the device is in Portrait mode (Compact Width, Regular Height), the red label displays within the stack view. When you rotate the device to landscape, the trait collection updates to Regular Width, Compact Height and the red label disappears to save space.

Debug View Hierarchy

The view hierarchy can become *very* complex, with numerous stack views and an assortment of buttons, labels, and images. If you have issues at runtime where one view is covering another, or constraints are not working as you'd expect, Xcode includes a tool that can help you solve the problem.

Build and run the app you just built that includes the colored labels. After the view has loaded, press the Debug View Hierarchy  button at the top of the Debug area that extends from the bottom of Xcode. The view will appear in a 3D environment that you can swivel around to see how the view hierarchy was constructed. ①



Here's a breakdown of what you're seeing in the editor area. As you move back to front, the proceeding view is layered on top of the previous.

- The black view is your application's `UIWindow`
- The white view is your view controller's view
- The small clear rectangle is the `UIStackView` that contains the colored labels
- The red, green, blue, orange, and purple labels are added subsequently

Using the Debug navigator  on the left, you can see a tree view of your hierarchy, along with the values of each of the constraints.^② This is a much quicker way to get the value of each constraint instead of printing them to the console.

The Debug View Hierarchy tool also includes an assortment of buttons at the top of the Debug area that you can use to enable/disable the content of each view, hide/show constraints, and more. If you find yourself struggling to resolve an issue with a view, you should consider using this tool.

Lab—Calculator

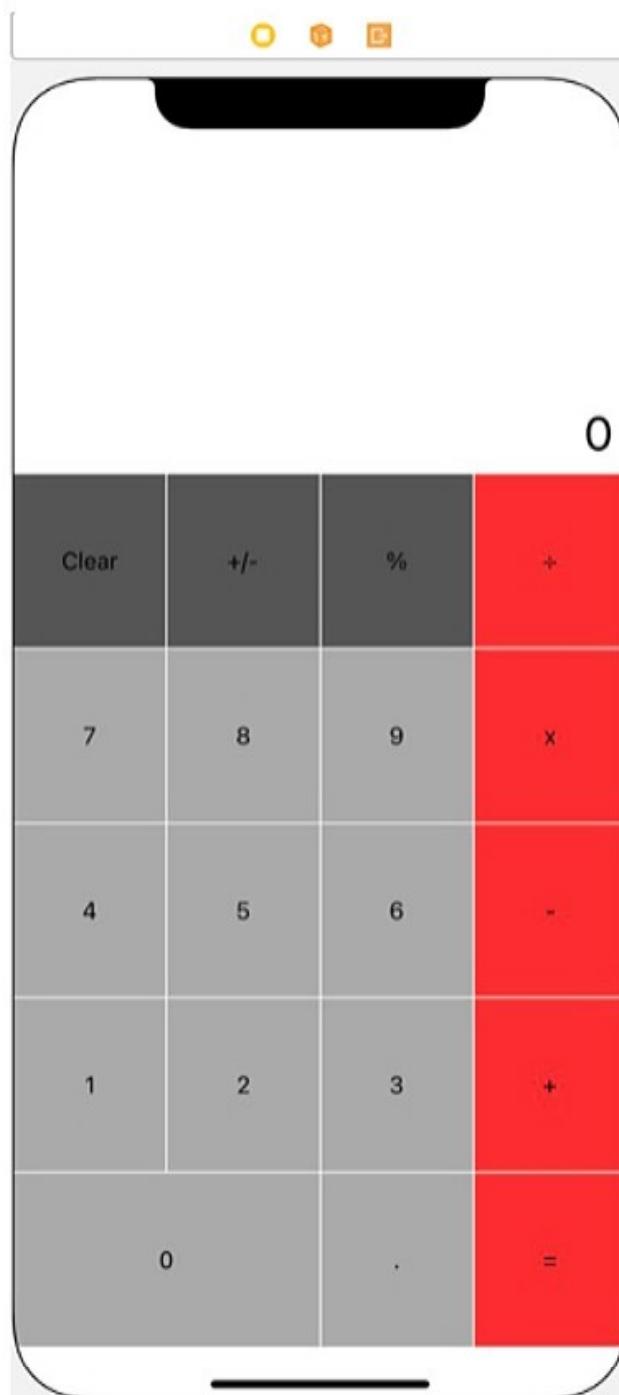
Overview

The objective of this lab is to use Auto Layout to create a view that scales with the size and layout of any screen. You'll use view objects, constraints, and stack views to create a simple calculator that maintains its layout on all device sizes.

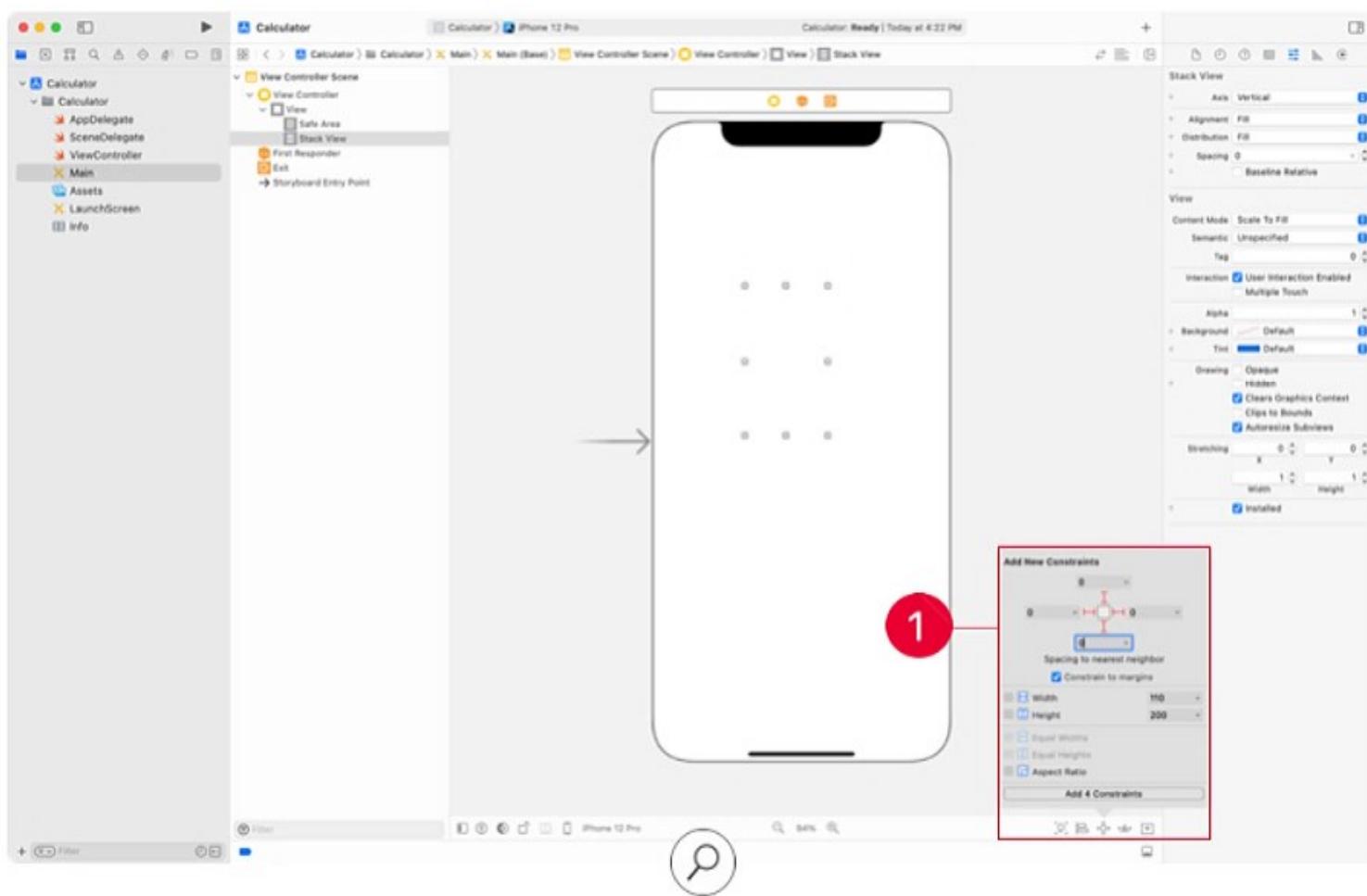
Create a new project called "Calculator" using the iOS App template.

Step 1

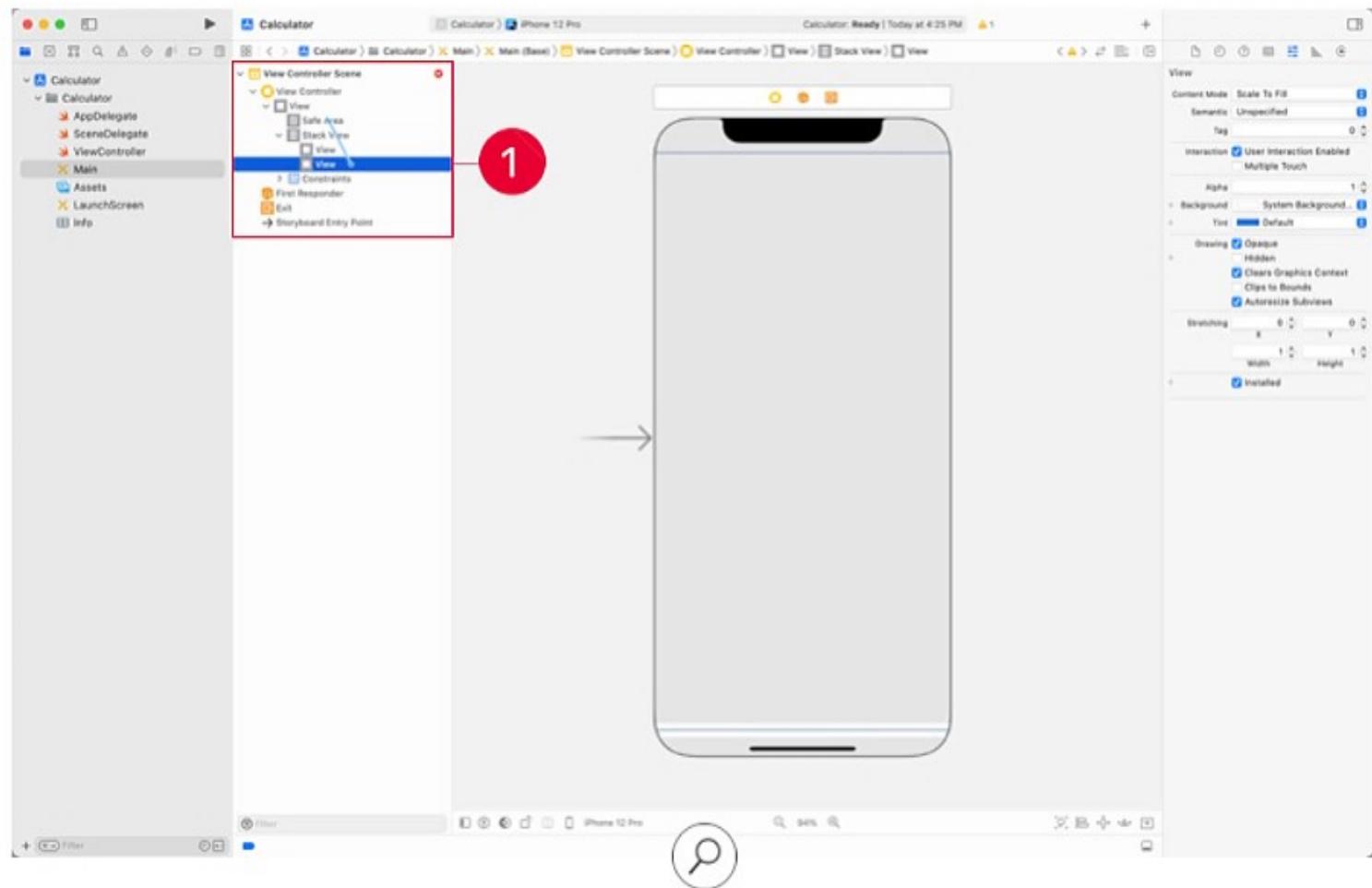
Create The Outer Containers



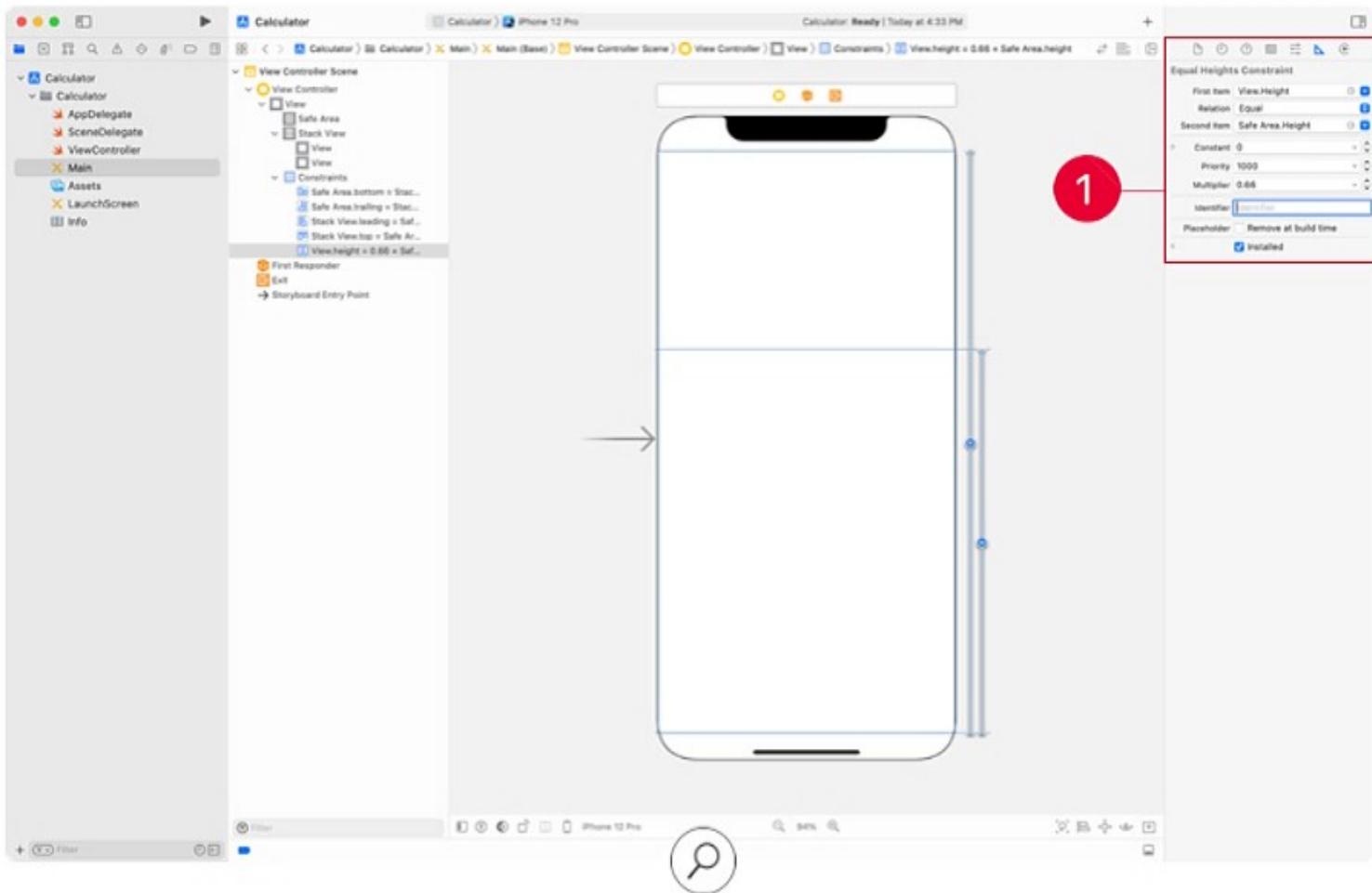
- In Interface Builder, set device to iPhone 13 Pro using the Device button.
- Ignoring the complexity of the buttons for now, it's easiest to break the calculator down into two distinct sections: the top third displays the digits that are entered, and the bottom two-thirds are used for input.
- Drag a vertical stack view from the Object library onto the scene. Use the Add New Constraints tool to add four constraints that align the top, leading, bottom, and trailing edges of the stack view to the the outer view's respective edges with 0 spacing.^① Click the Update Frames button, and the stack view should cover the entire screen.



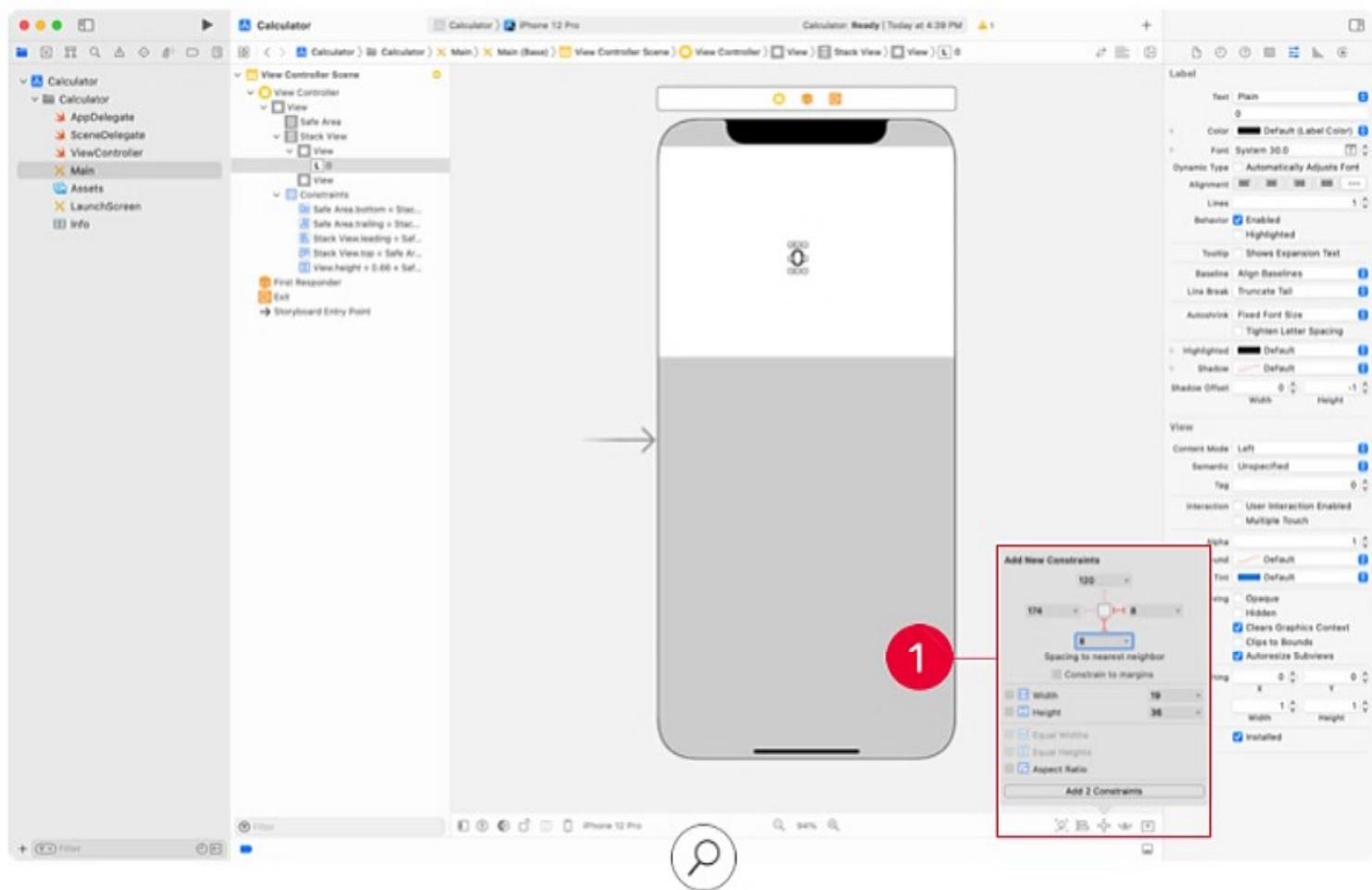
- Drag two `UIView` objects from the Object library into the stack view. Control-drag from the bottom view to the Safe Area Layout Guide, and select Equal Heights in the popup menu. This will set the inner view's height equal to the outer, which you will fix in the next step. ①



- Select the bottom view and open the Size inspector. Locate the height constraint and double-click it. Change the Multiplier value from 1 to 0.66, which will set the inner view's height equal to the outer view's height, multiplied by 0.66.^① The height of the bottom view will always be two-thirds the size of the view controller's view.



Add a `UILabel` from the Object library into the top container view. Set the text to "0" and the font size to 30.0, then open the Add New Constraints tool. Add bottom and trailing constraints, with 8 pixels of spacing between them. ①

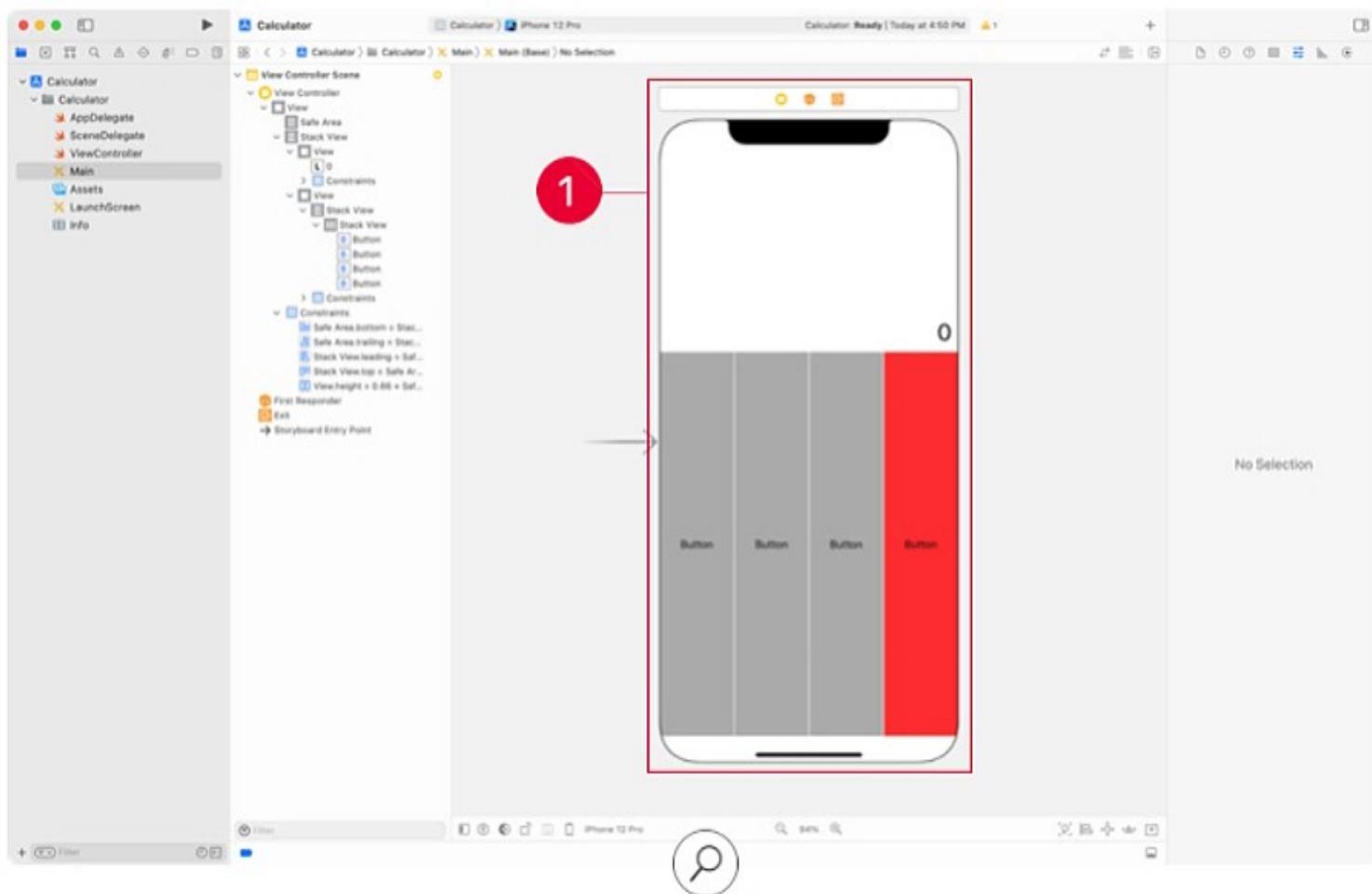


Step 2

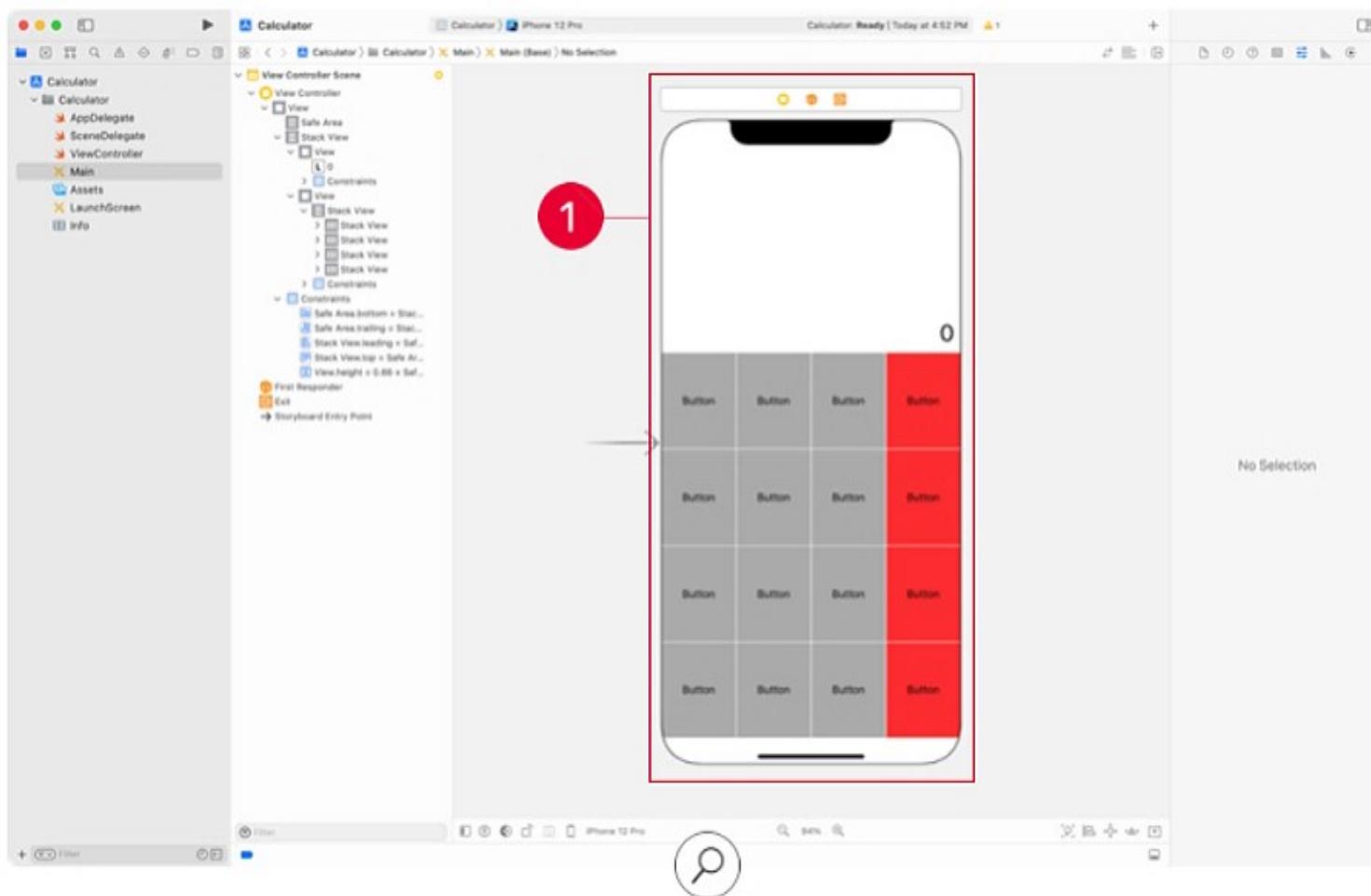
Add And Size Buttons

- The finished calculator has five rows of buttons, where each row has an equal height. You can use another vertical stack view to accomplish this. Drag a vertical stack view into the bottom container view, then use the Add New Constraints tool to constrain the top, leading, bottom, and trailing edges of the stack view to the the bottom container view's respective edges with 0 spacing. Click the Update Frames button, and the stack view should cover the entire bottom container.
- Select the newly-added vertical stack view, and open the Attributes inspector. Set the “Distribution” property to “Fill Equally” so that all subviews within the stack view will have the same height. Set the “Spacing” to 1, which will create the horizontal separation lines between each row.
- You can use horizontal stack views for each row of buttons. Add a horizontal stack view into the vertical stack view, and set its “Distribution” property to “Fill Equally” so that all subviews within the stack view have the same width. Set the “Spacing” to 1, which will create the horizontal separation lines between each subview.
- Add a `UIButton` into the horizontal stack view. Set the tint color to Black Color and the background color to Light Gray.

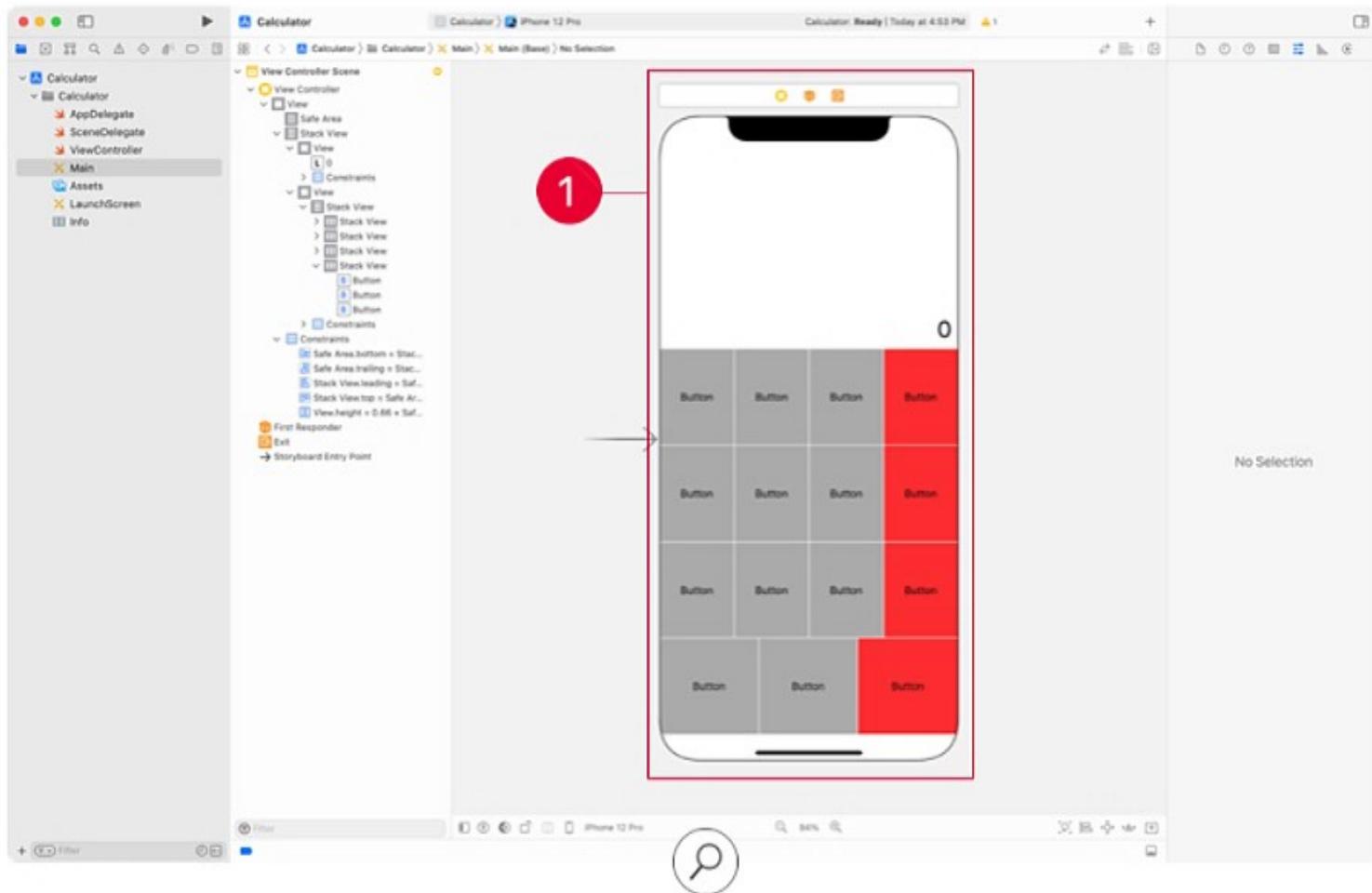
- Copy the button to your clipboard (**Command-C**), then paste (**Command-V**) three additional buttons into the horizontal stack view. Change the background color of the last button to System Red. ^①



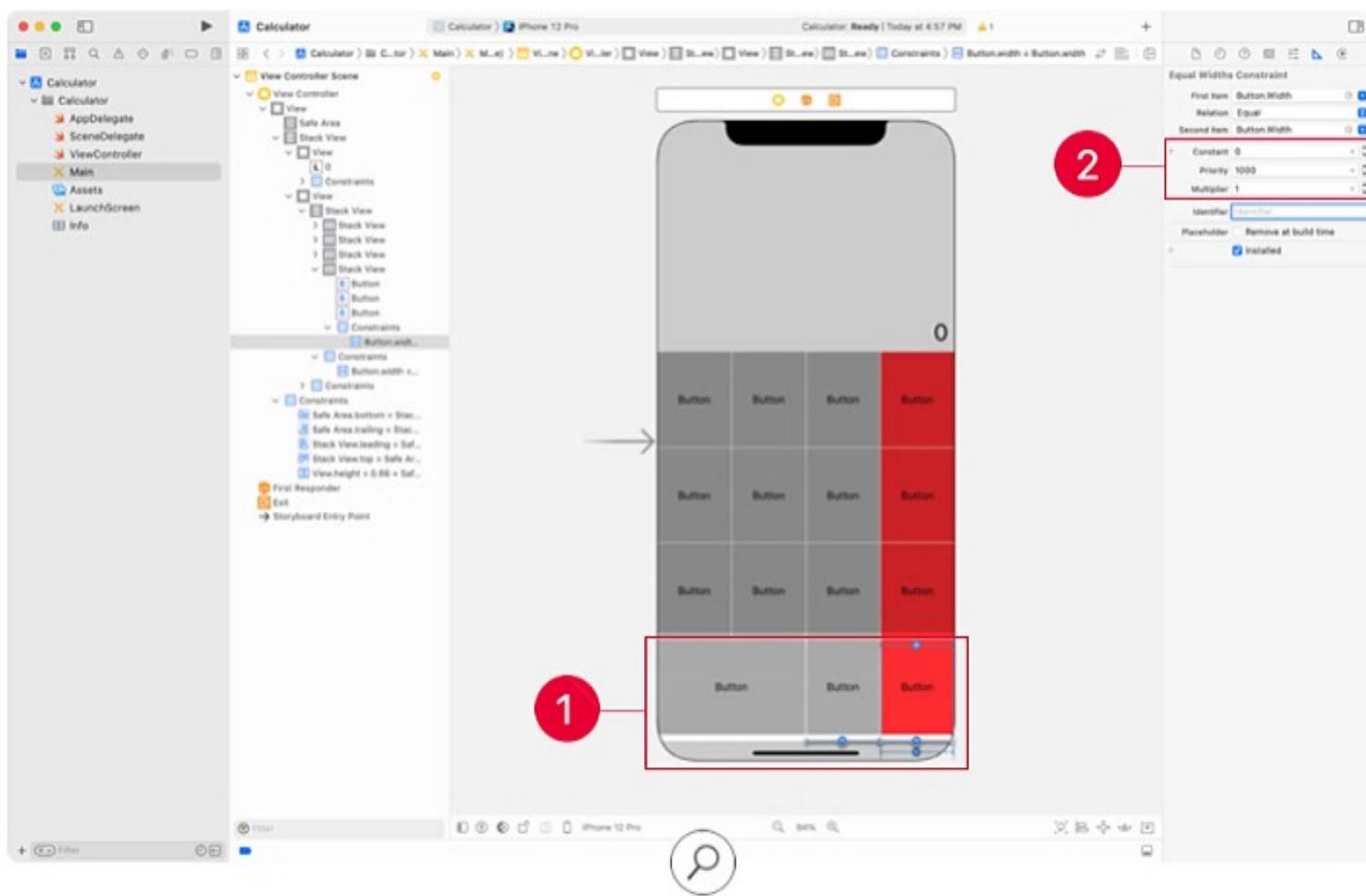
- Select the horizontal stack view in the Document Outline, and copy it to your clipboard (**Command-C**). Then paste (**Command-V**) four additional horizontal stack views into the vertical stack view. **①**



- The bottom horizontal stack view uses only three buttons instead of four. Since some buttons are sized differently than others, you need to update the “Distribution” of this stack view to “Fill Proportionally.” Then, delete one of the light gray buttons from the bottom so the stack view contains the proper number of controls. ①



- Control-drag from the bottom-rightmost red button to the red button above it, and then select "Equal Widths." Now locate this constraint and edit it, changing the multiplier to "1".^① This ensures that these two buttons are always the same size. Repeat the process for the middle light gray button and the red button.^② The leftmost gray button will now fill the remaining space making its total width equal to the two gray buttons above it including the space between them.



- Finally, update the top three buttons to use a dark gray color, and update the button titles to match those on a traditional calculator.

Congratulations! You've used both constraints and stack views to create a simple calculator. Be sure to build and run your application on multiple iOS devices to verify that the constraints you've defined make sense on all screen sizes. Save your work to your project folder.

Connect To Design

In your App Design Workbook, reflect on how you could use stack views to position elements on your screen. With the high-level map you have in your prototype, can you start to identify what information might be in each row or column of interface elements? Make comments in the Map section or add a blank slide at the end of the document.

In the workbook's Go Green app example, a stack view on the daily log page would have the calendar week at the top, then a graphic showing pounds of trash versus recycling, and then a list of today's logged items.

Review Questions

Question 1 of 5

If a button is centered vertically and horizontally within a view, which of the following constraints were most likely defined? (Check all that apply.)

- A. Center X
- B. Center Y
- C. Width
- D. Top

Check Answer

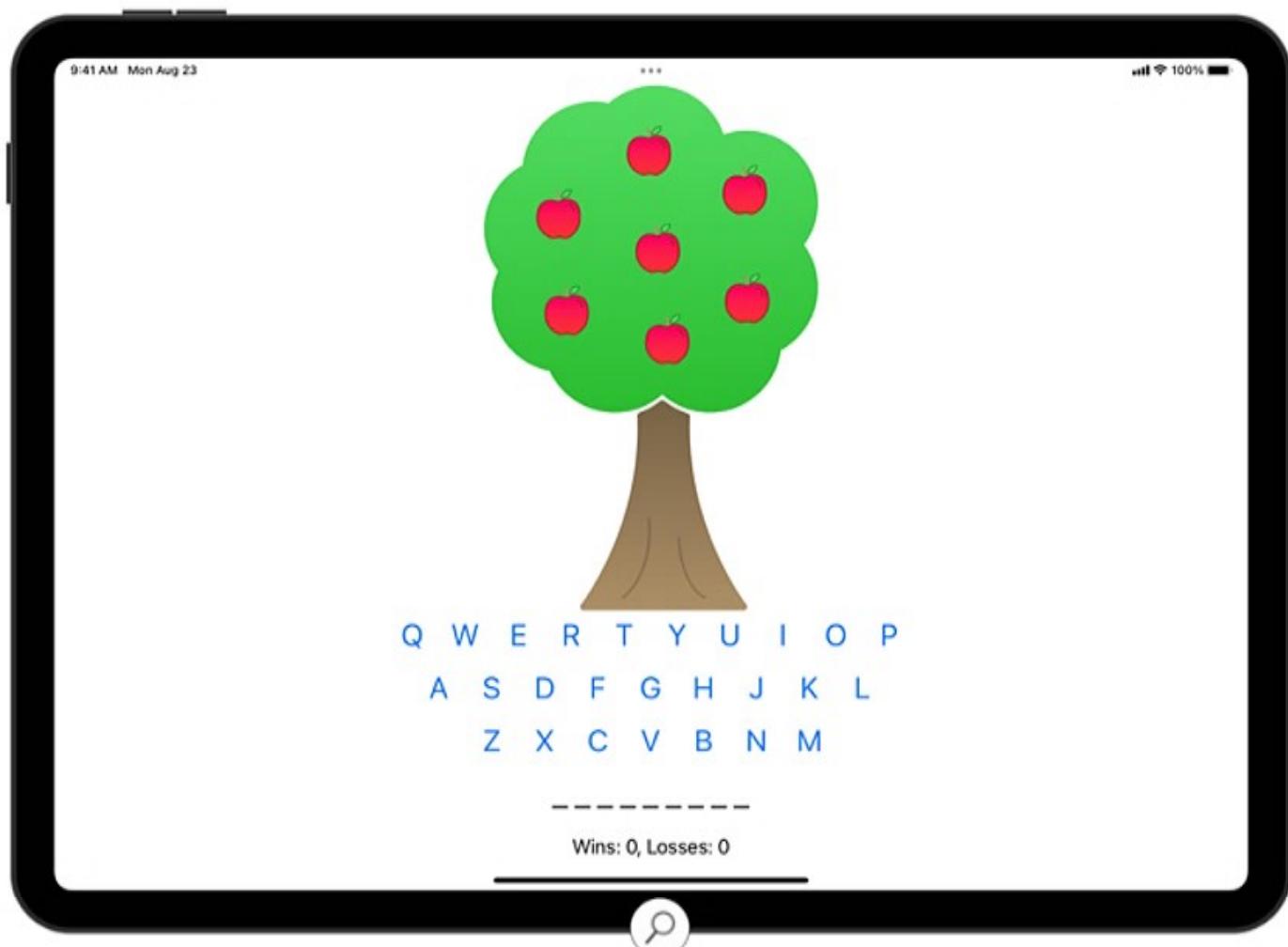


Guided Project: Apple Pie

So far in this unit, you've learned a lot about the fundamentals of Swift. Now it's time to put your knowledge to work.

Your project is to write a game called Apple Pie. In this simple word-guessing game, each player has a limited number of turns to guess the letters in a word. Each incorrect guess results in an apple falling off the tree. The player wins by guessing the word correctly before all the apples are gone.

As a new programmer, you may find this larger project a bit intimidating. Remember, building a working app happens one step at a time. If you get stuck on a particular step, go back and review that lesson. You can do it!

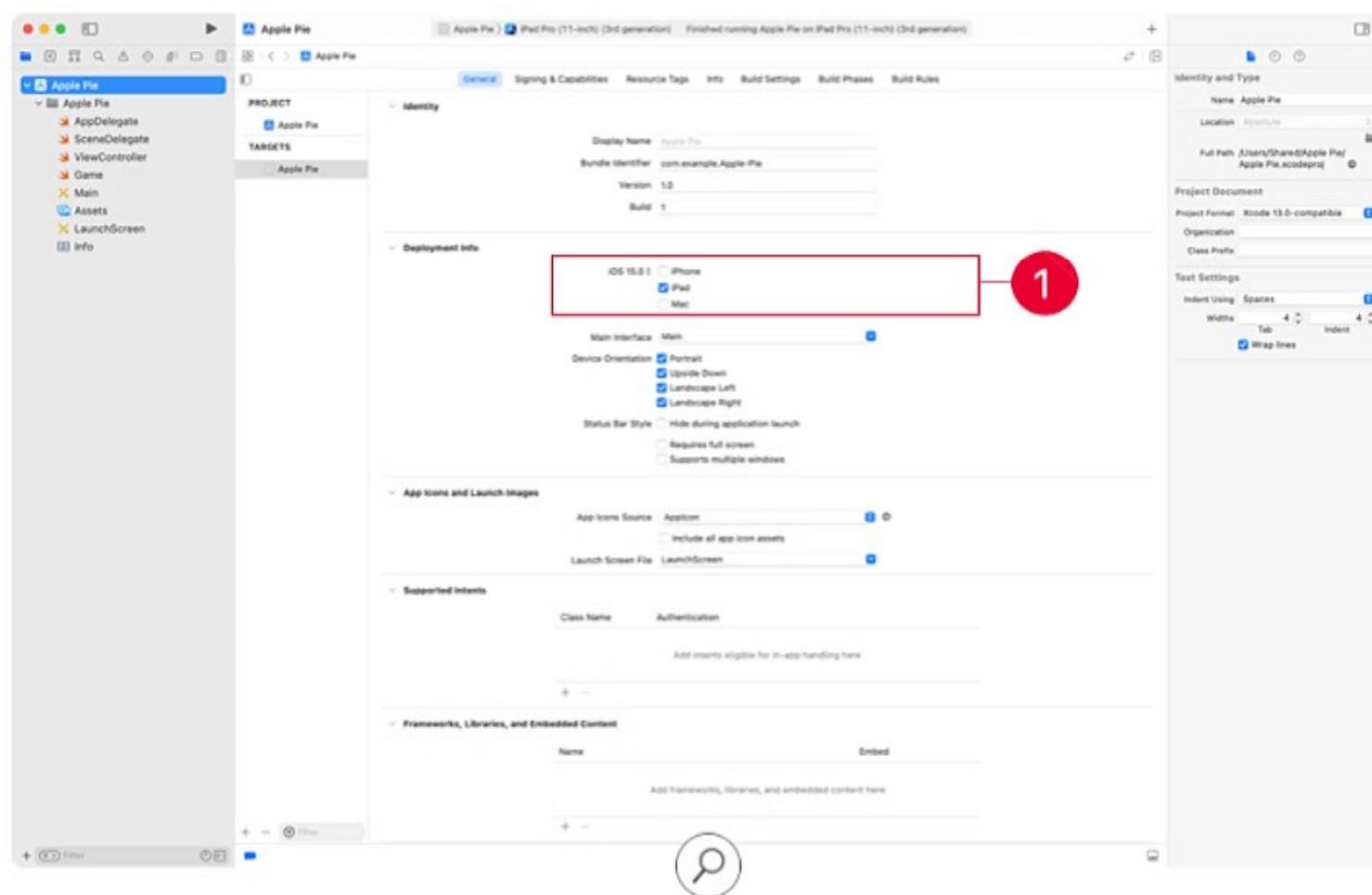


In this particular version of Apple Pie, whenever a round is won or lost, a new game is immediately started. If there are no more words left to guess, all buttons are disabled, and the app needs to be restarted.

Part One

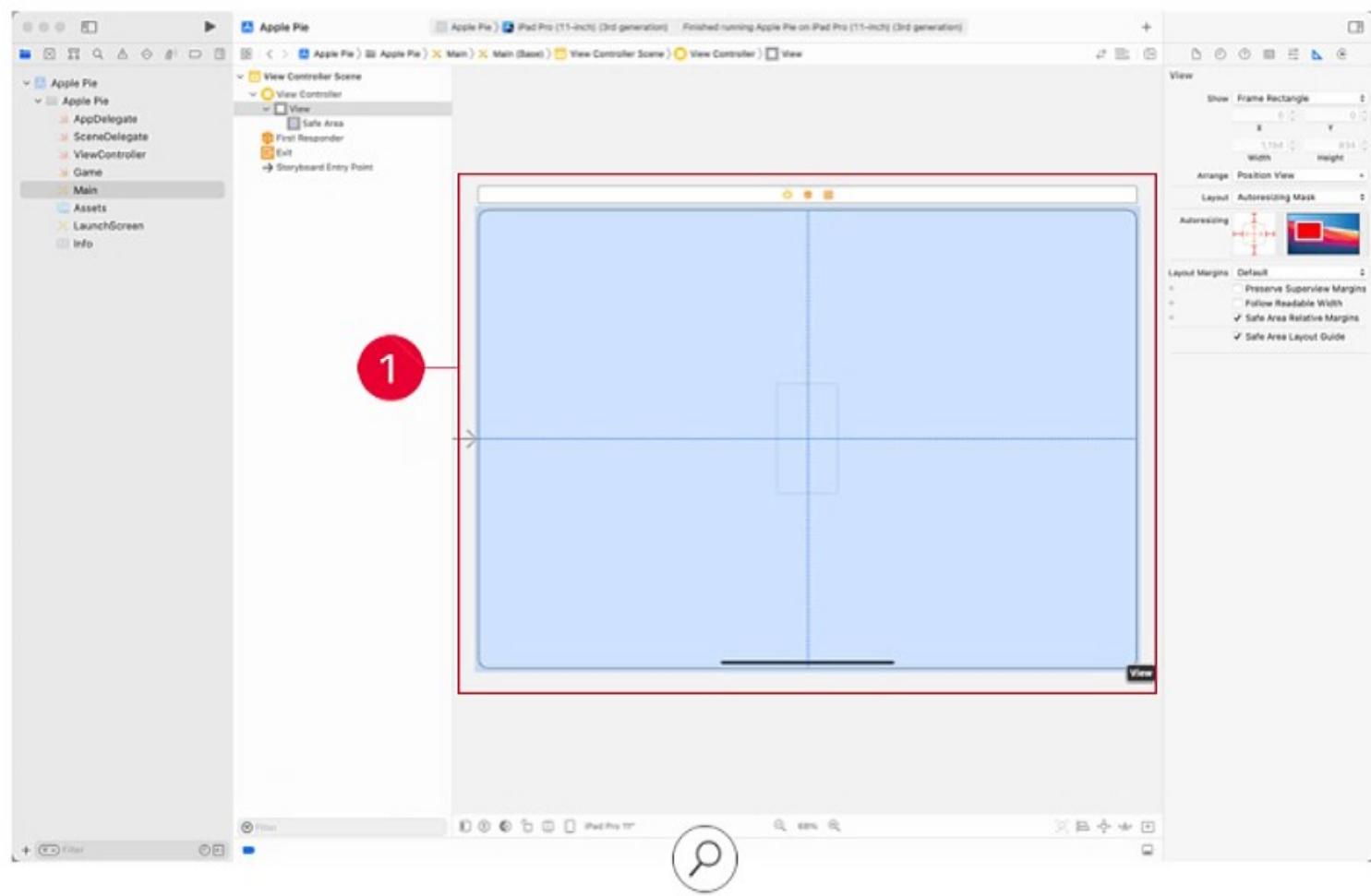
Build The Interface

Create a new project using the iOS App template. Name the project "Apple Pie." This game is meant to be played on the iPad, and the interface that you'll be building doesn't accommodate a small iOS device. To make the app iPad only, select the project file in the Project navigator, then under Deployment Info, uncheck the checkbox under Device for iPhone.¹

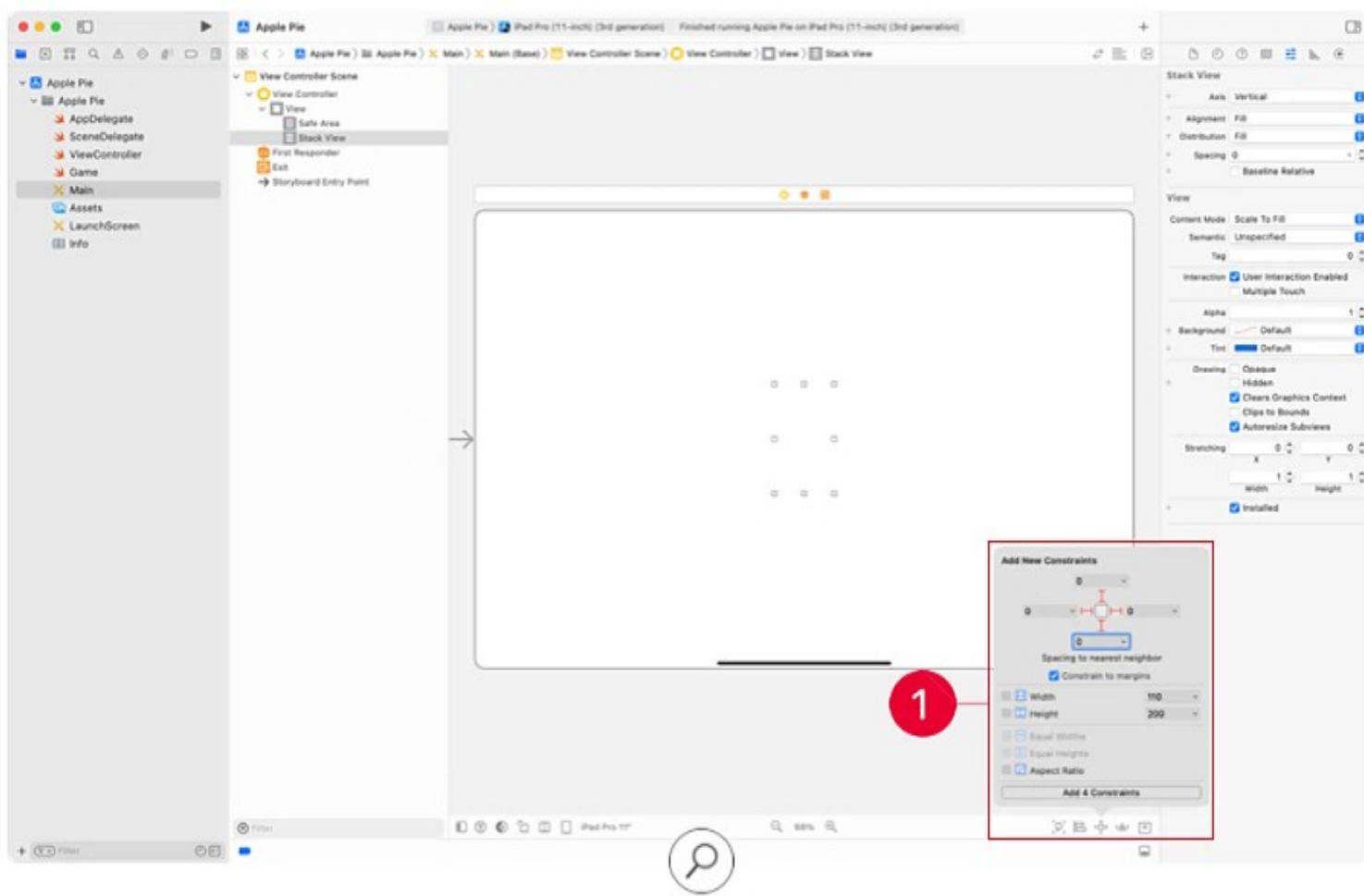


Layout in Storyboard

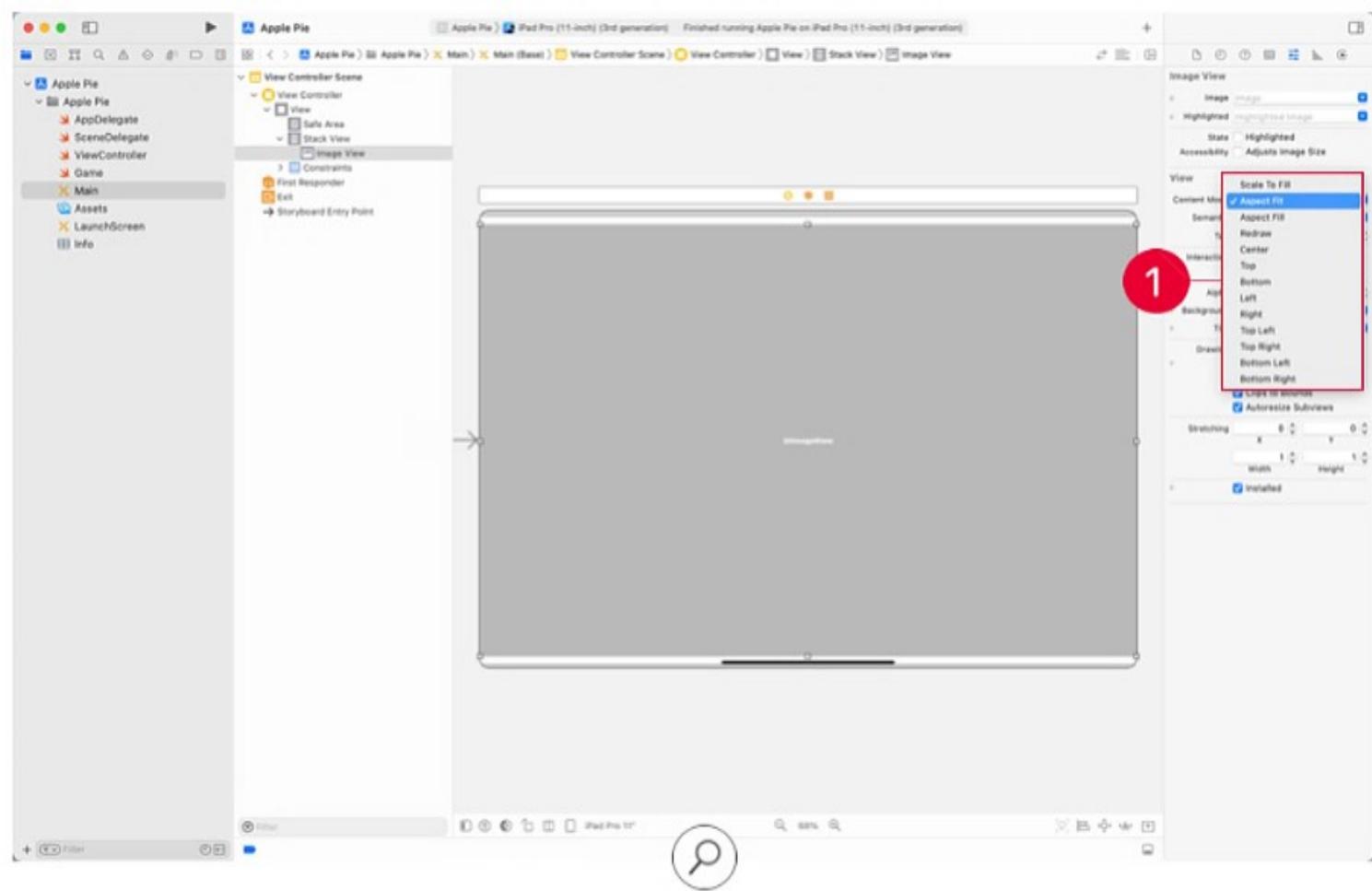
Open the **Main** storyboard and use the "Devices" button to change the device to an iPad and change the orientation to Landscape mode using the "Orientation" button. Since you'll be using Auto Layout to build the interface, it will work in Portrait mode as well. Look back at the image of the finished app. Using the Interface Builder tools that you've learned so far, how might you construct this interface? There is an image at the top, followed by a grid of buttons, and then two rows of labels containing text. Perhaps the simplest way to build this interface is by using a vertical stack view. Find the vertical stack view in the Object library, and drag it onto the iPad canvas. ①



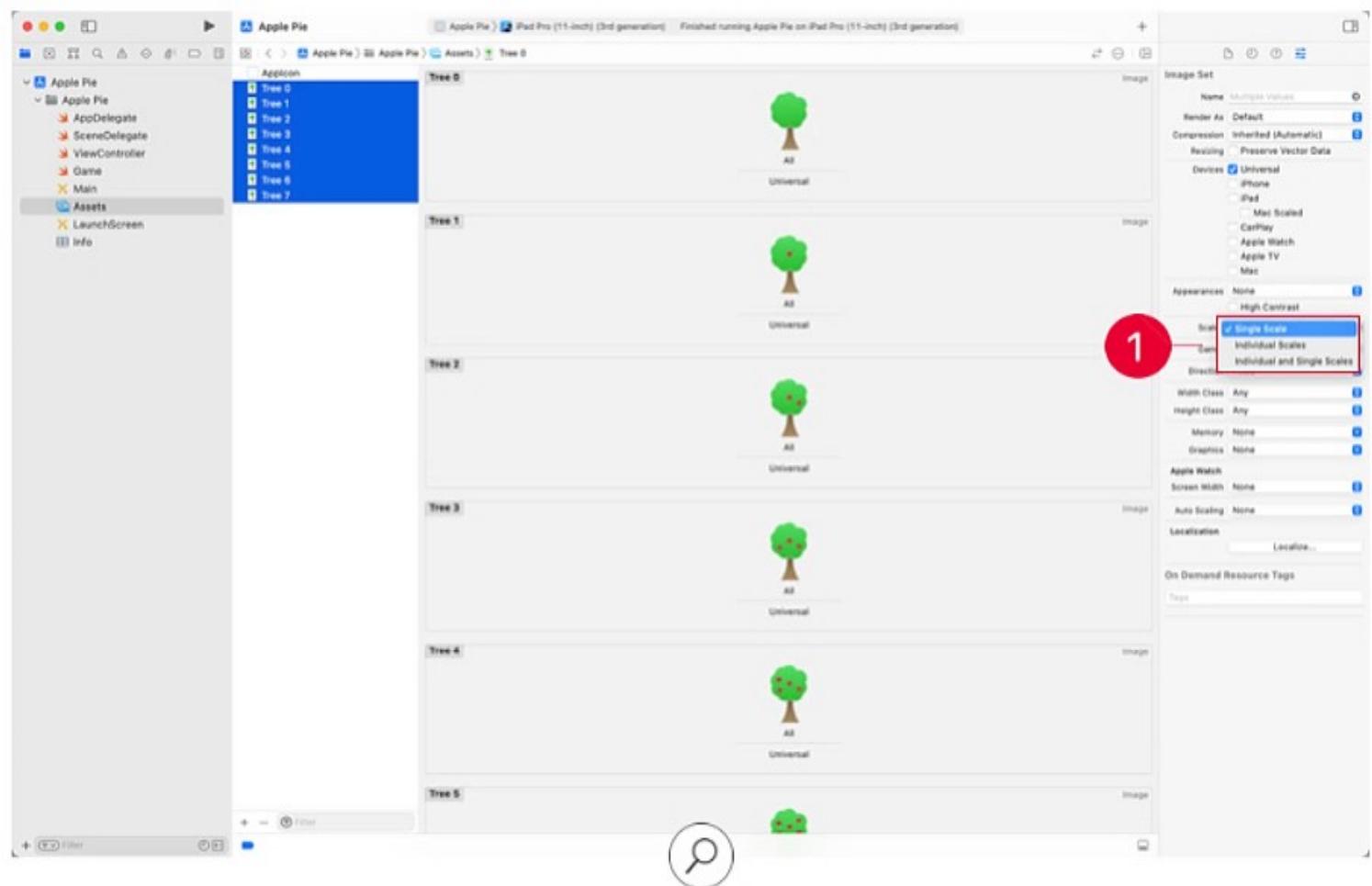
As you've learned in earlier lessons, you can determine the position, width, and height of a stack view by adding constraints between the stack and its superview. Where does the stack start, and where does it end? The image view is near the top, and the last label goes along the bottom of the screen. The grid of buttons begins near the left edge and ends near the right edge. Therefore, the stack view can be constrained to cover the entire screen. Select the stack view, then use the Add New Constraints tool to add four constraints. Set all four fields at the top of the popover to 0. The red indicators illuminate to show the edges that are being constrained. When you're done, click "Add 4 Constraints." ①



Now you're ready to add views and controls to the stack. Search the Object library for an image view, and drag one into the stack view. Since it's the only item within the stack, it will take up the entire space of the stack. Change the "Content Mode" of the image view to "Aspect Fit" in the Attributes inspector. ① This will ensure the width and height of the image is not distorted by the proportions of the image view.

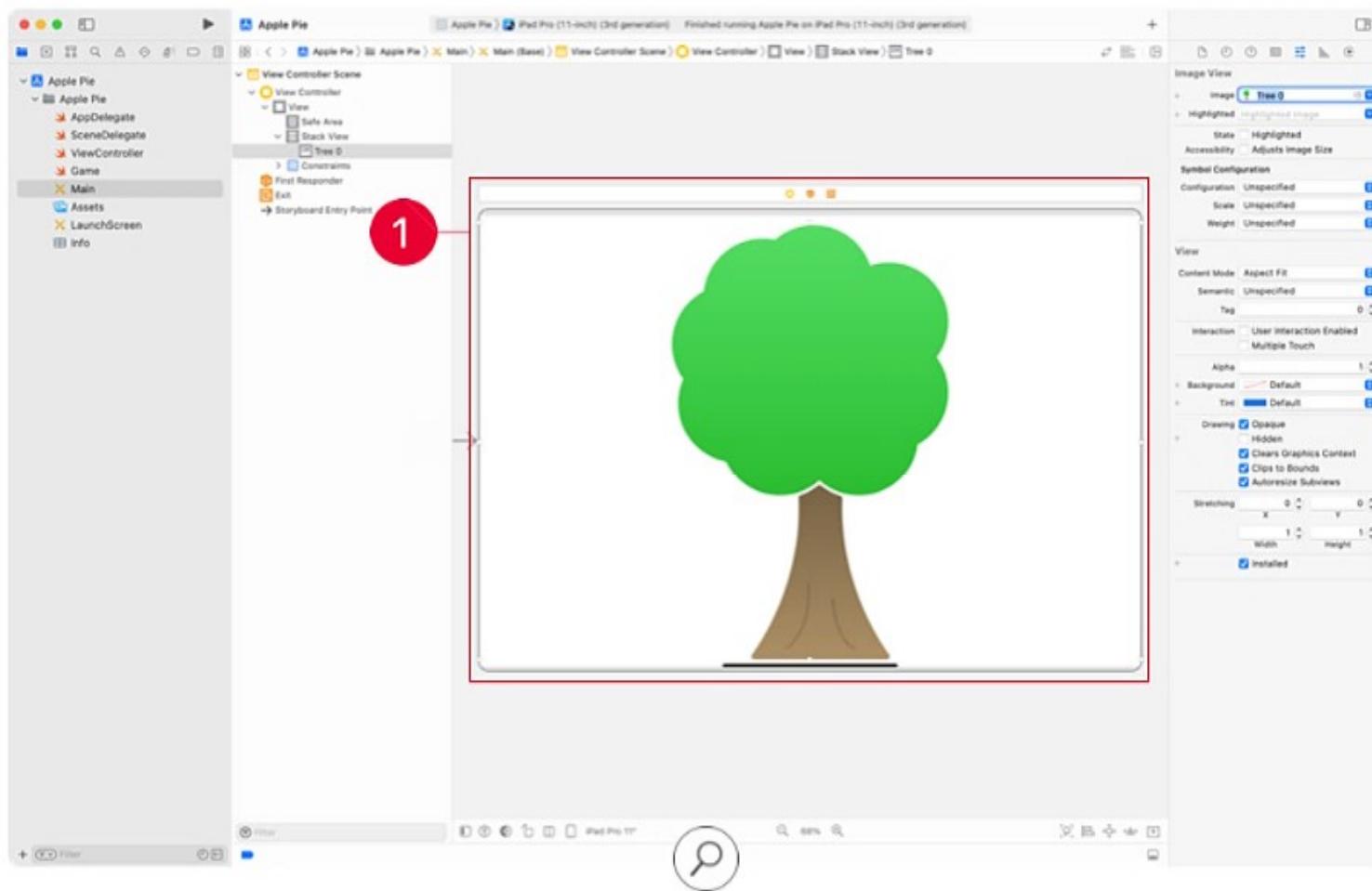


Add the apple tree images (Tree 0.pdf...Tree 7.pdf) from the student resources folder into your **Assets** folder. After they've been added, select all of them and choose "Single Scale" from the Scales menu in the Attributes inspector. ①



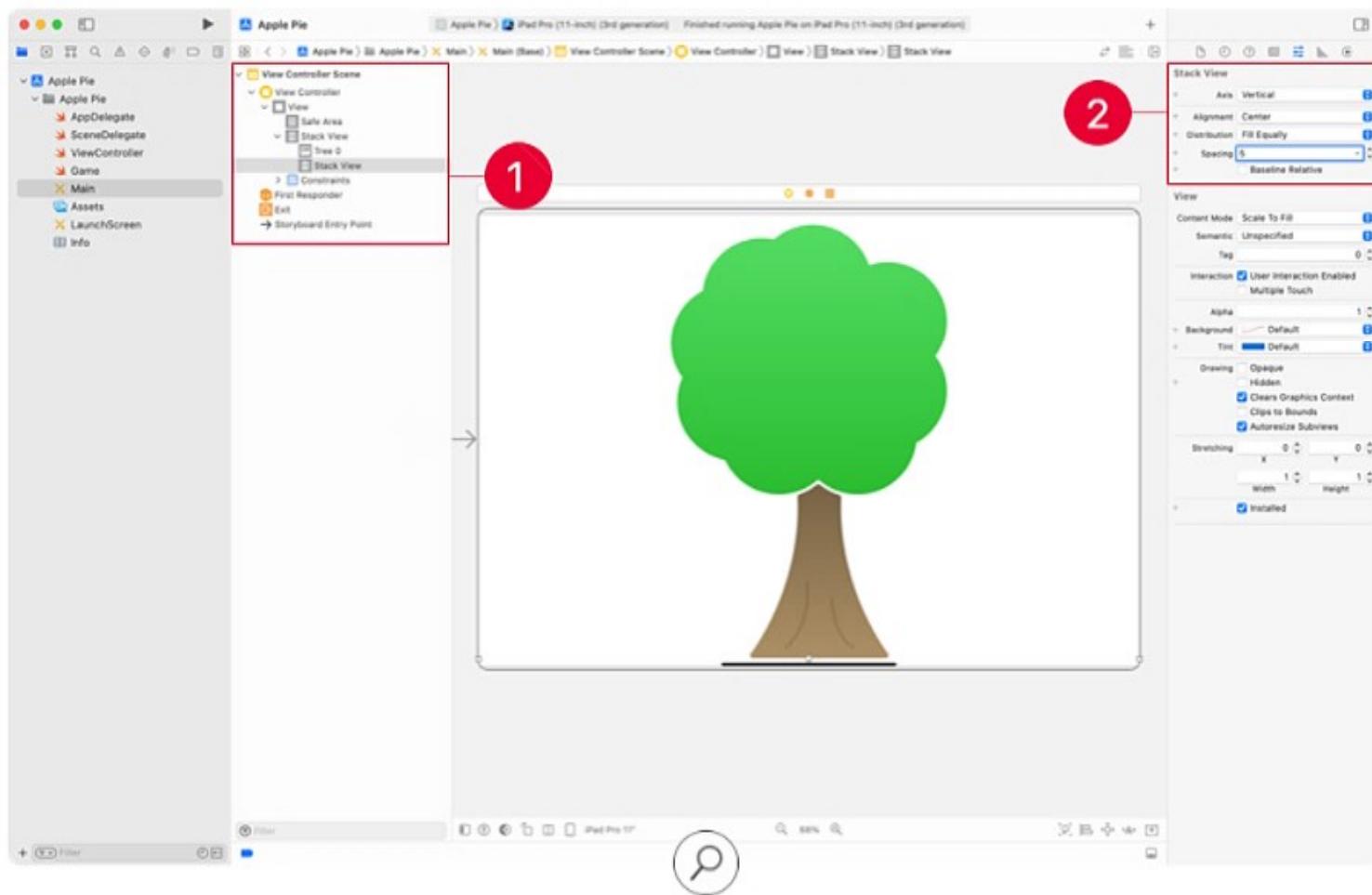
These assets are vector art, which means they can dynamically scale to any size without losing quality. When possible, use PDF files with vector art so that you don't need to supply multiple scales for various devices.

Now, you can go back to the image view in the storyboard and update the image property to use one of the tree images. Even though you're updating the image view's image in code, this can help you visualize your interface from within the storyboard. ①

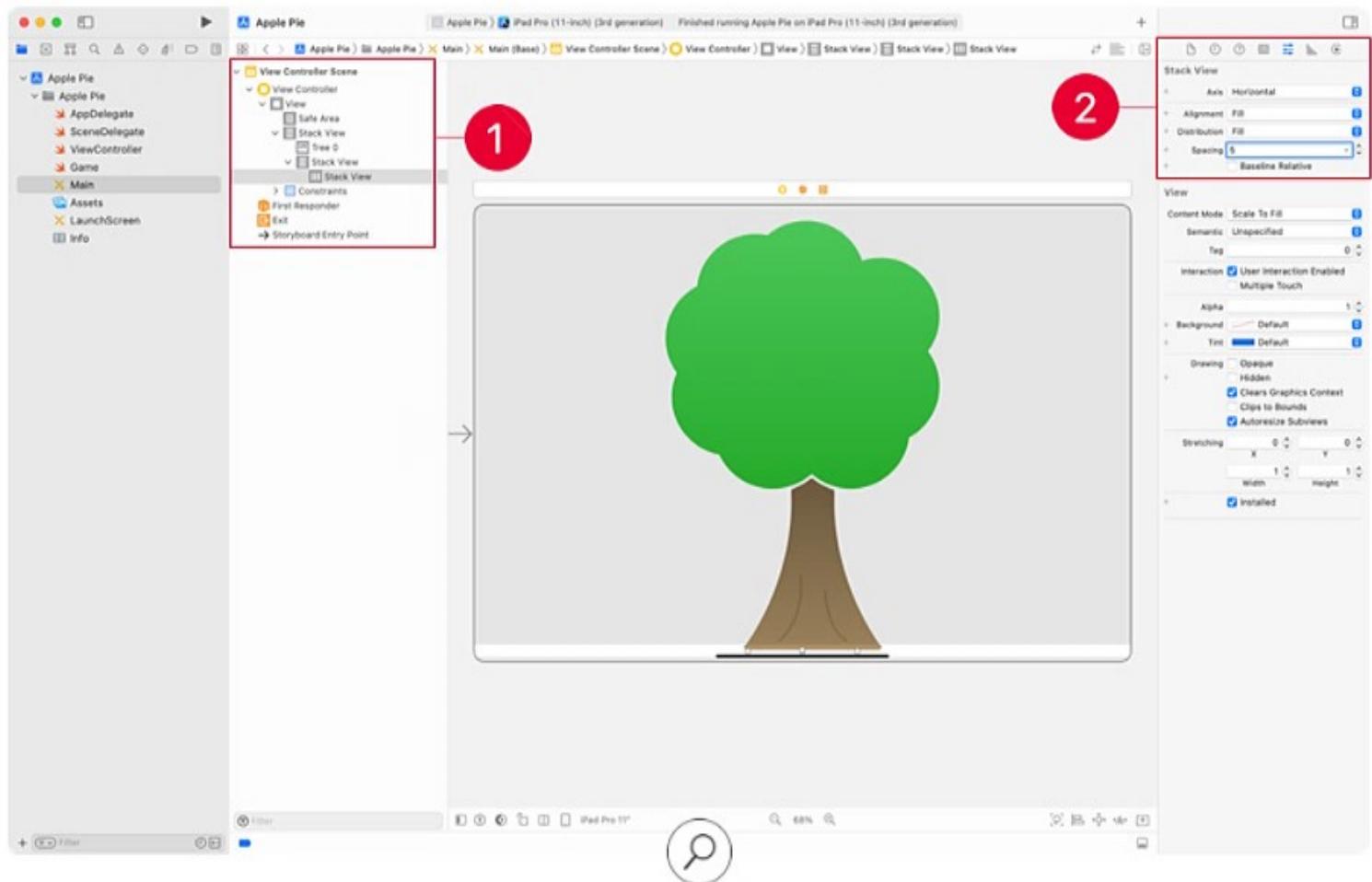


The button grid is more complex. You're emulating the layout of a standard QWERTY keyboard, which has a unique number of keys in each row. You can imagine each row as its own horizontal stack view, and you can contain these rows in a vertical stack view. Use the Object library to add a vertical stack view below the image view.^① You can reorder the items within the stack by dragging them around in the Document Outline.

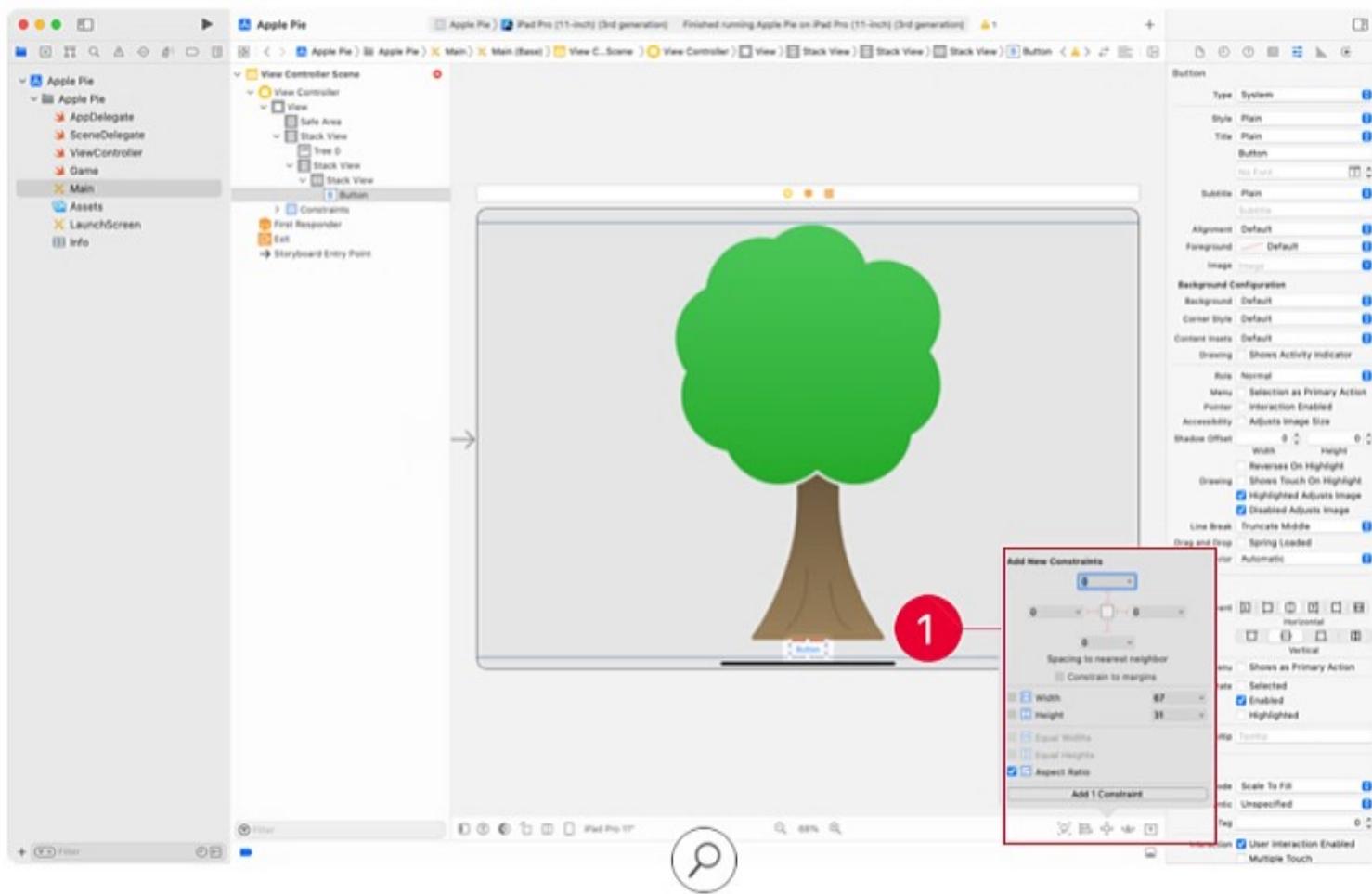
Each of the horizontal stack views (or rows) that you add will be equal in size and centered. Select the newly added vertical stack view, then open the Attributes inspector and change Alignment to Center and Distribution to Fill Equally. Also provide some spacing between rows by setting the Spacing to 5.^②



Now add a horizontal stack view within the vertical stack. ① You want the same spacing between columns as rows, so set the Spacing to 5. Set both the Alignment and Distribution to Fill. ②



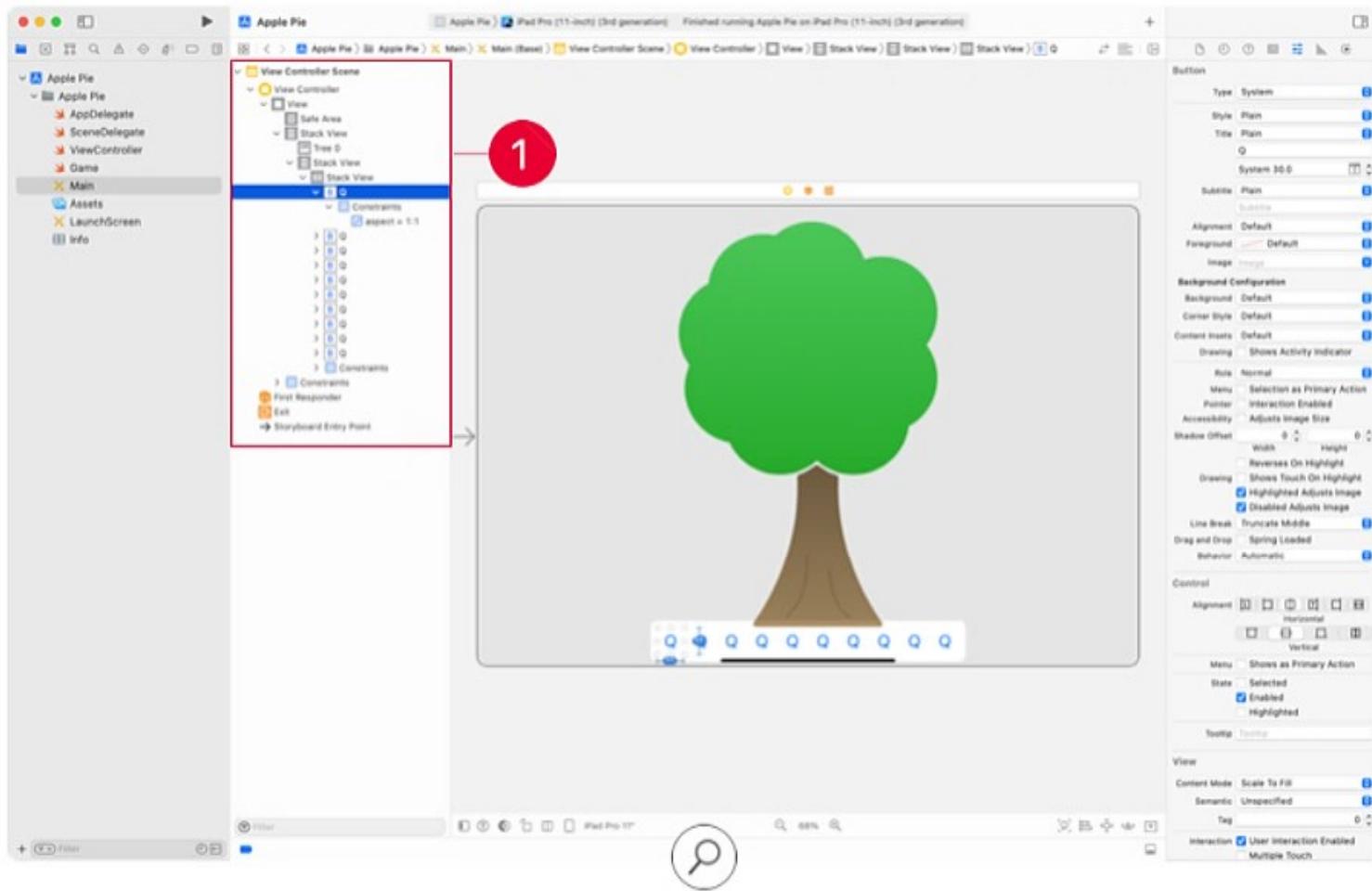
Each row has a unique number of buttons with the first being 10, the second 9, and the third 7. Use the Object library to add a button to the horizontal stack view. The letter keys on a keyboard are typically square. To achieve this appearance, select the button and use the Add New Constraints button to add an Aspect Ratio constraint. ① The constraint will be added but with an unwanted ratio.



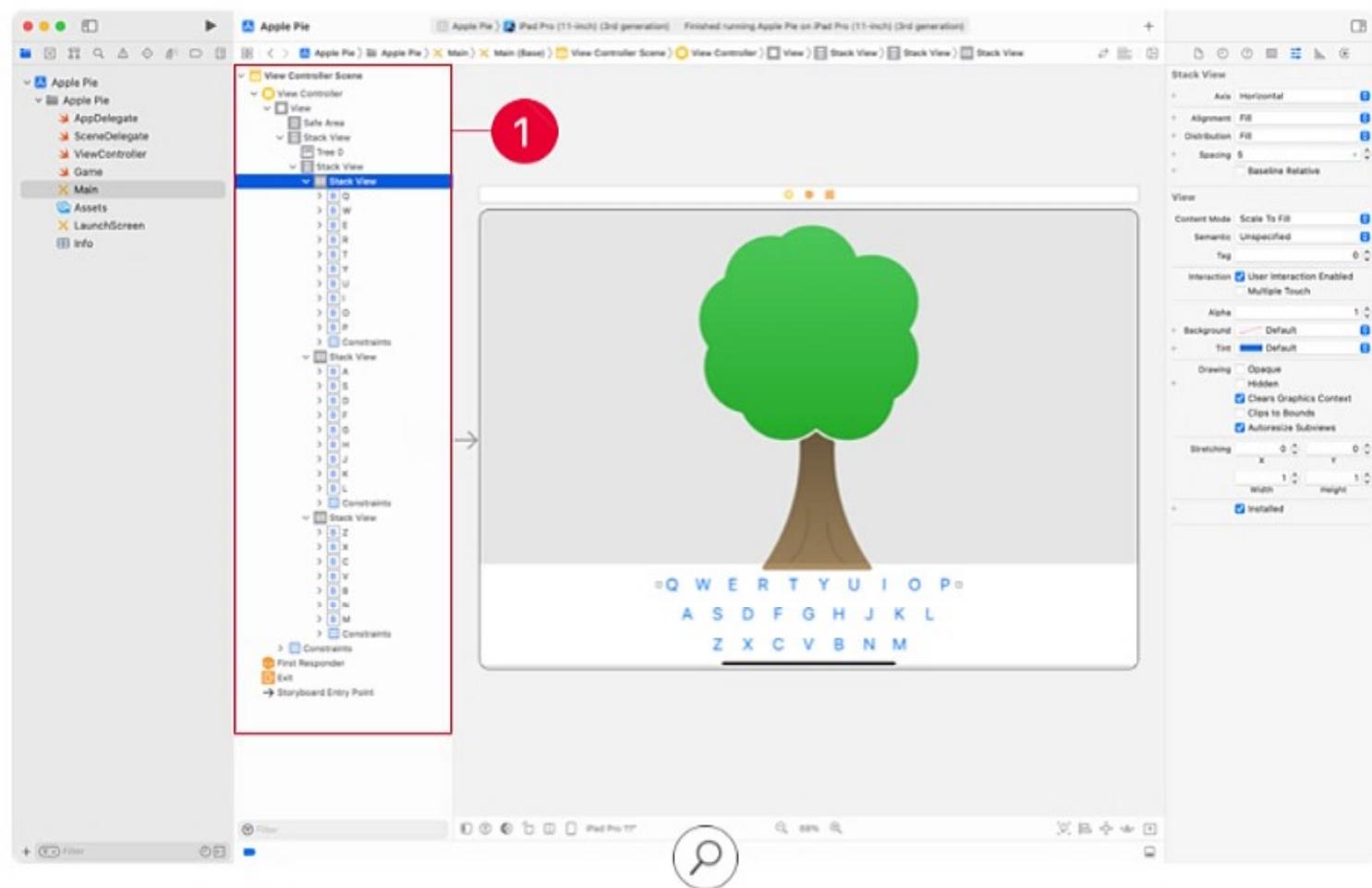
Use the Size inspector for the button to edit the Aspect Ratio constraint, setting the multiplier to 1. This ensures that the button is square.

Use the Attributes Inspector to set the button's font to System and set the font size to 30 to make the button easier to read.

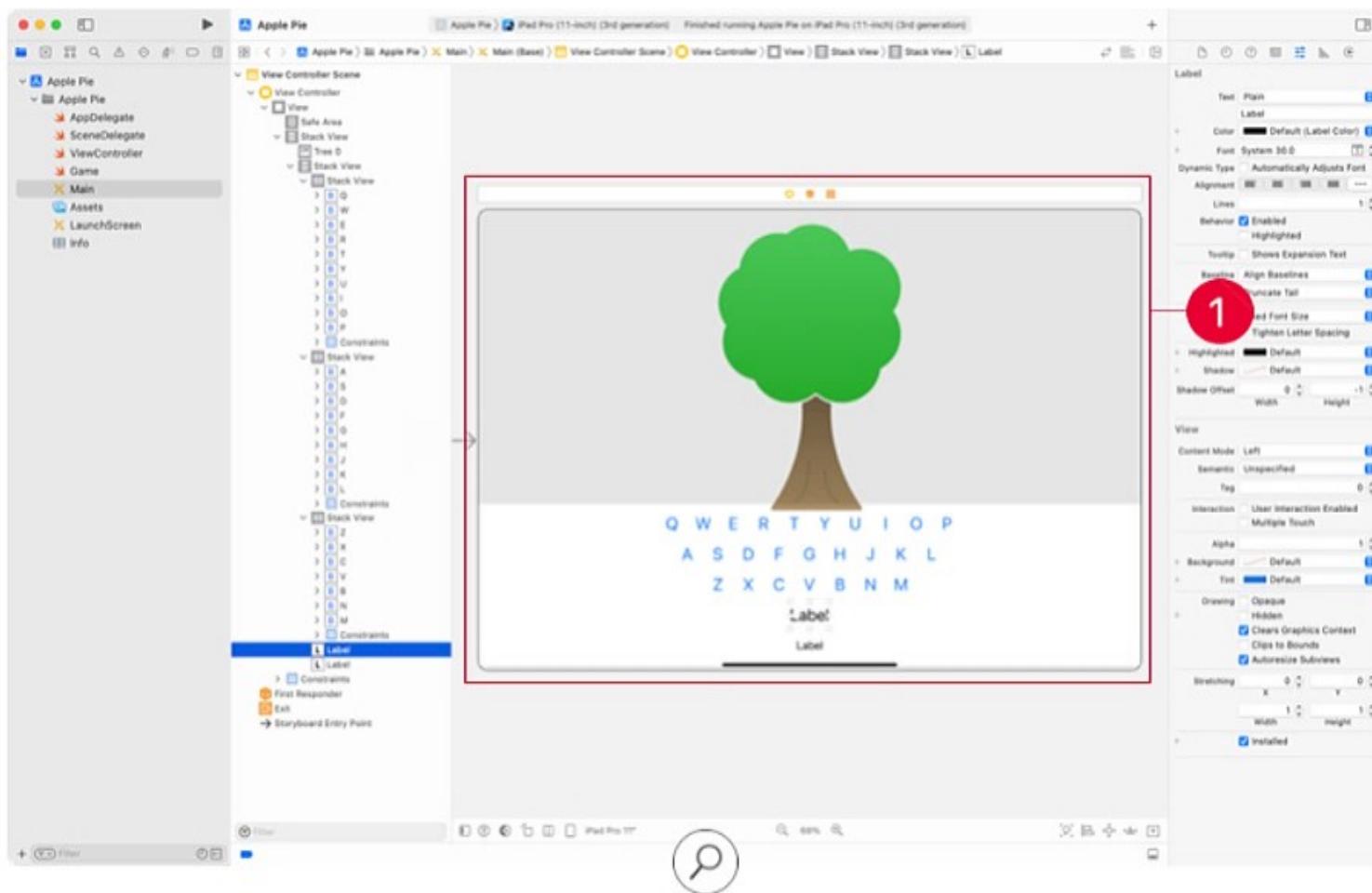
Change the button's text to the letter Q, then select and copy it to your clipboard (**Command-C**). Paste (**Command-V**) a new copy of the button into the stack view, and repeat this step until the stack view contains 10 buttons. Use the Attributes inspector to update the text of each button to a single capital letter using your own keyboard as a reference. ①



Now select the horizontal stack view, and copy it to your clipboard. Paste two new copies of the stack view. Update each button with the appropriate letters once again using your keyboard as reference—deleting any extra button to match. You can quickly edit a button's text by double-clicking it within Interface Builder. ①



Below the grid of buttons within the second vertical stack view, add two labels from the Object library. Use the Attributes inspector to change the font size of the first label to 30.0 and the second to 20.0. The buttons and labels all have what is referred to as intrinsic size, which the text within them determines. Auto Layout uses their intrinsic size along with the stack view attributes to appropriately size the button grid while the tree image view can shrink and expand as necessary. ¹



Before you move on, verify that you don't have any warnings or errors in Interface Builder. You can build and run your app to ensure that the layout looks correct on different iPad simulators or by using the View As feature in Interface Builder.

Create Outlets and Actions

During gameplay, the text of the labels change whenever a letter is guessed correctly or whenever a new round begins. The image need to change whenever an incorrect letter is guessed, and the letter buttons need to be disabled whenever they're pressed and re-enabled before each round. To update these views, create outlets so that you can reference the views in code.

Open the assistant editor so that the `ViewController` class definition appears to the right of the storyboard. Next, select the image view on the left, **Control-drag** to an area within the class definition, and release the mouse button. In the pop-up menu that appears, name the outlet `treeImageView`. When you press the Connect button, the code for the outlet is created.

```
@IBOutlet var treeImageView: UIImageView!
```

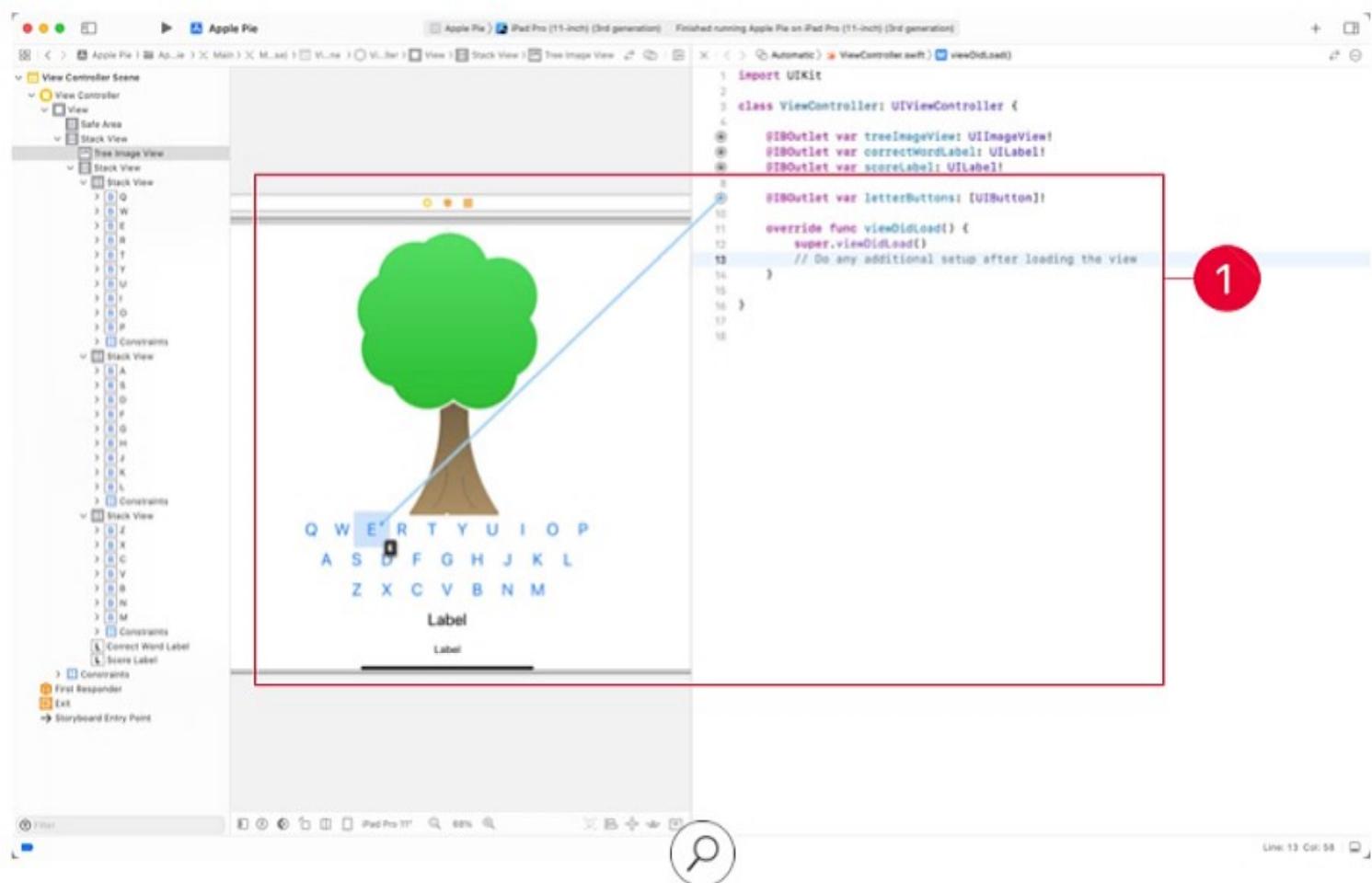
Repeat this process for the two labels. Name the top label `correctWordLabel`, and the bottom label `scoreLabel`.

```
@IBOutlet var correctWordLabel: UILabel!
@IBOutlet var scoreLabel: UILabel!
```

It's tedious to create a separate outlet for each `UIButton`. Instead, create one outlet that holds the collection of buttons. Begin by selecting the first button with the letter Q as the title. **Control-drag** to the class definition to create an outlet, then release the mouse button. In the pop-up menu that appears, change the Connection type from Outlet to Outlet Collection. Then set the name of the outlet collection to `letterButtons`. When you press Connect, an outlet is created that references a collection of buttons.

```
@IBOutlet var letterButtons: [UIButton]!
```

Now click and drag from the circle next to the outlet collection to another button. A blue rectangle will appear, indicating a valid connection. Release the mouse button, then repeat this step for each button in the scene. ①



Each button also needs to be tied to an action. Again, it's tedious to create a separate action for each button. Instead, create one action that all the buttons call when pressed. Begin by selecting the first button with the letter Q as the title. **Control-drag** to the class definition, and release the mouse button. In the pop-up menu that appears, change the Connection type to Action. Set the name of the action to `letterButtonPressed`, and change the type of the argument to `UIButton`. When you press Connect, the method is created. Whenever a letter button is tapped, it should be disabled (a player can't select a letter more than once in the same round).

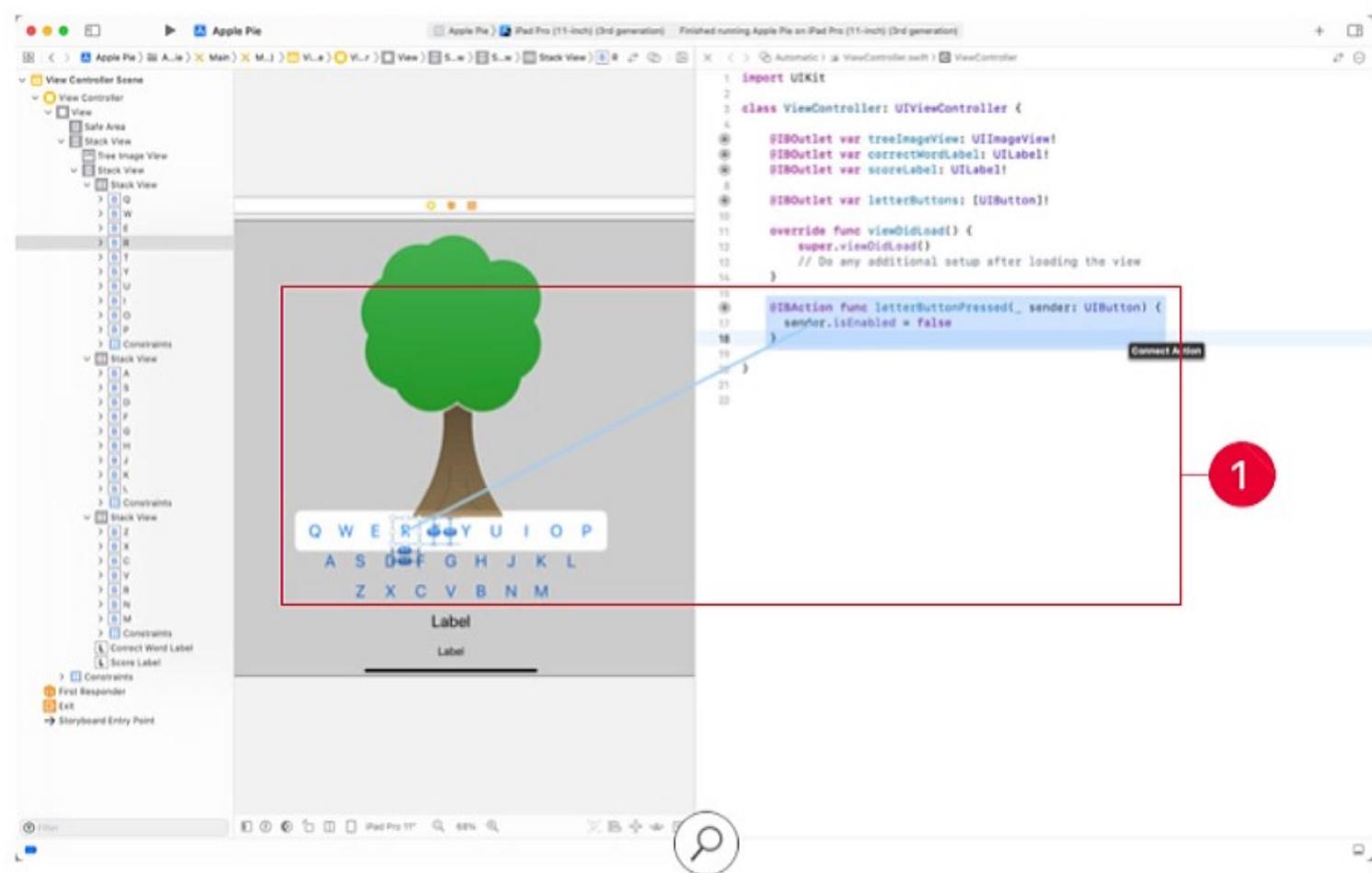
```

@IBAction func letterButtonPressed(_ sender: UIButton) {
    sender.isEnabled = false
}

```

Why change the argument type from `Any` to `UIButton` in the popup? Since there are twenty-six buttons connected to the same action, you'll need to use the `sender` to determine *which* button, specifically, triggered the method. If `sender` has the `Any` type, you can't access the `title` property (and we'll need that later).

Control-drag the rest of the buttons to the existing action. You'll know the connection is valid because a blue rectangle will appear as you hover the mouse near the method. 



Build and run your application to verify that tapping each button disables it. It'll become more apparent what else to do inside this method after you build other portions of the app.

Part Two

Beginning A Game

Now you're set to work on the Apple Pie game logic.

Define Words and Turns

Your first task is to supply a list of words for players to guess. At the top of `ViewController`, define a variable called `listOfWords`. Fill this array with words: food names, hobbies, animals, household objects, or whatever else. To keep things simple, use only lowercase letters.

```
var listOfWords = ["buccaneer", "swift", "glorious",
"incandescent", "bug", "program"]
```

Below `listOfWords`, define a constant called `incorrectMovesAllowed` which establishes how many incorrect guesses are allowed per round. The lower the number, the harder it will be for the player to win. There are seven different images of apple trees provided, so you'll want this value to be between 1 and 7.

```
let incorrectMovesAllowed = 7
```

Define Number of Wins and Losses

After each round, the bottom label will display an updated count of the number of wins and losses. Create two variables to hold each of these values, and set the initial values to 0.

```
var totalWins = 0
var totalLosses = 0
```

Begin First Round

When the application launches, the `viewDidLoad()` method of `ViewController` is called. This is a great place to start a new round. Define a method called `newRound`.

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

What does it mean to start a new round? Each round begins with the selection of a new word, and resetting the number of moves the player can make to `incorrectMovesAllowed`. It would be helpful to hold the state of the game inside of a `Game` struct. Create a new file in your project by selecting **File -> New -> File (Command-N)** from the Xcode menubar. Select “Swift File” as your template, then select Next. Name the file “Game.”

Inside of `Game`, define a struct called `Game`. For now, you know that an instance of a `Game` has two properties: the word, and the number of turns you have left to properly guess the word.

```
import Foundation  
  
struct Game {  
    var word: String  
    var incorrectMovesRemaining: Int  
}
```

Back in the `newRound()` method, you can create a new instance of a `Game`. You should create a property that holds the current game's value so that it can be updated throughout the view controller code. You can give the `Game` a new word in the initializer by removing the first value from the `listOfWords` collection, and set `incorrectMovesRemaining` to the number of moves you allow, stored in `incorrectMovesAllowed`.

```
var currentGame: Game!

func newRound() {
    let newWord = listOfWords.removeFirst()
    currentGame = Game(word: newWord, incorrectMovesRemaining:
incorrectMovesAllowed)
}
```

Why does the `currentGame` variable have an exclamation mark at the end? For a brief moment between the app launch and the beginning of the first round, `currentGame` doesn't have a value. This is a concept you'll learn in the next unit. For now, know that the exclamation mark means that it's OK for this property not to have a value for a short period.

Now that you've started a new round, you need to update the interface to reflect the new game. Create a separate method called `updateUI()` that will handle the interface updates, then call it at the end of `newRound`. Inside of this method, there are two pieces of the interface that you can update with the code you've written thus far: the score label, and the image view. The score label uses simple string interpolation to combine `totalWins` and `totalLosses` into a single string. Since each of the tree images are named "Tree X," where X is the number of moves remaining, you can use string interpolation once more to construct the image name.

```
func newRound() {  
    let newWord = listOfWords.removeFirst()  
    currentGame = Game(word: newWord, incorrectMovesRemaining:  
        incorrectMovesAllowed)  
    updateUI()  
}  
  
func updateUI() {  
    scoreLabel.text = "Wins: \(totalWins), Losses: \  
(\ntotalLosses)"  
    treeImageView.image = UIImage(named: "Tree \  
(currentGame.incorrectMovesRemaining)")  
}
```

Build and run your application. The score label and the image view should update to reflect the beginning of a new round. Great job!

Part Three

Update Game State

So far, you've built a working interface, and you've successfully started a new round. Now you need to add some code that progresses the game further towards a win or a loss.

Extract Button Title

Currently, the `letterButtonPressed(_ :)` method disables whichever button the player used to guess, but it doesn't update the game state. Whenever a button is clicked, you should read the button's title, and determine if that letter is in the word the player is trying to guess. Begin by reading the title from the button's configuration property. Not all buttons have configurations or titles, so you'll need to add exclamation marks to tell the Swift compiler that your button *does* have a configuration that has a title. (You'll learn more about the exclamation mark in the next unit.) You should lowercase the letter because it's be much easier to compare everything in lowercase, and then convert it from a `String` to a `Character`.

```
@IBAction func letterButtonPressed(_ sender: UIButton) {  
    sender.isEnabled = false  
    let letterString = sender.configuration!.title!  
    let letter = Character(letterString.lowercased())  
}
```

Guess Letter

Now that you have the letter that the player guessed, what should you do with it? A `Game` manages how many more moves are remaining, but it doesn't know which letters have been selected during the round. Add a collection of characters to `Game` that keeps track of the selected letters, named `guessedLetters`. Then add a method in `Game` that receives a `Character`, adds it to the collection, and updates `incorrectMovesRemaining`, if necessary. (By adding the `guessedLetters` property, you'll need to update the initialization for `currentGame`.)

```
struct Game {
    var word: String
    var incorrectMovesRemaining: Int
    var guessedLetters: [Character]

    mutating func playerGuessed(letter: Character) {
        guessedLetters.append(letter)
        if !word.contains(letter) {
            incorrectMovesRemaining -= 1
        }
    }

    func newRound() {
        let newWord = listOfWords.removeFirst()
        currentGame = Game(word: newWord, incorrectMovesRemaining:
            incorrectMovesAllowed, guessedLetters: [])
        updateUI()
    }

    @IBAction func letterButtonPressed(_ sender: UIButton) {
        sender.isEnabled = false
        let letterString = sender.title(for: .normal)!
        let letter = Character(letterString.lowercased())
        currentGame.playerGuessed(letter: letter)
        updateUI()
    }
}
```

Build and run your application. As you press each button, the character gets added to the list of letters the player has guessed, and `incorrectMovesRemaining` decreases by 1 for each incorrect letter.

Part Four

Create Revealed Word

Using `word` and `guessedLetters`, you can now compute a version of the word that hides the missing letters. For example, if the word for the round is "buccaneer" and the user has guessed the letters "c," "e," "b," and "j," the player should see "b_cc__ee_" at the bottom of the screen.

To begin, create a computed property called `formattedWord` within the definition of `Game`. Here's one way to compute `formattedWord`:

- Begin with an empty string variable.
- Loop through each character of `word`.
- If the character is in `guessedLetters`, convert it to a string, then append the letter onto the variable.
- Otherwise, append `_` onto the variable.

```
var formattedWord: String {
    var guessedWord = ""
    for letter in word {
        if guessedLetters.contains(letter) {
            guessedWord += "\\"(letter)"
        } else {
            guessedWord += "_"
        }
    }
    return guessedWord
}
```

Now that `formattedWord` is a property that your UI can display, try using it for the text of `currentWordLabel` inside of `updateUI()`.

```
func updateUI() {  
    correctWordLabel.text = currentGame.formattedWord  
    scoreLabel.text = "Wins: \(totalWins), Losses: \  
    (totalLosses)"  
    treeImageView.image = UIImage(named: "Tree \  
    (currentGame.incorrectMovesRemaining)")  
}
```

Build and run your application. Letters will be added to the `guessedLetters` collection as they're selected, and `formattedWord` will be re-calculated whenever it's accessed in `updateUI()`.

You may notice a new issue: Because multiple underscores appear as a solid line in the interface, it can be difficult to tell how many letters are in the word. A solution could be to add spaces during your computation of `formattedWord`. But this issue is purely about improving the interface, not meddling with the computed data. A better solution is to add the spaces when you update the text of `correctWordLabel`.

To properly set the text of `correctWordLabel`, you can use a Swift method named `joined(separator:)` that operates on an array of strings. This function concatenates the collection of strings into one string, separated by a given value. Here's an example:

```
let cast = ["Vivien", "Marlon", "Kim", "Karl"]  
let list = cast.joined(separator: ", ")  
print(list) // "Vivien, Marlon, Kim, Karl"
```

How can you imagine using the `joined(separator:)` method to set `correctWordLabel`? Start by converting the array of characters in `formattedWord` into an array of strings. Use a for loop to store each of the newly created strings into a `[String]` array. Then you can call the `joined(separator:)` method to join the new collection together, separated by blank spaces.

```
func updateUI() {  
    var letters = [String]()  
    for letter in currentGame.formattedWord {  
        letters.append(String(letter))  
    }  
    let wordWithSpacing = letters.joined(separator: " ")  
    correctWordLabel.text = wordWithSpacing  
    scoreLabel.text = "Wins: \(totalWins), Losses:  
    \(totalLosses)"  
    treeImageView.image = UIImage(named: "Tree  
    \(currentGame.incorrectMovesRemaining)")  
}
```

Build and run your application to see a clear break between the letters and the underscores.

Part Five

Handle A Win Or Loss

Apple Pie is starting to feel like a real game. The round is progressing along with each button pressed, but there are two obvious issues. The first is that `incorrectMovesRemaining` can go below 0, and the second is that the player can't win a game, even if they guess all letters correctly.

When `incorrectMovesRemaining` reaches 0, `totalLosses` should be incremented, and a new round should begin. It would be nice to have a single method that checks the game state to see if a win or loss has occurred, and if so, update `totalWins` and `totalLosses`. Create a method called `updateGameState` that will perform this work, and call it after each button press instead of calling `updateUI()`.

```
@IBAction func letterButtonPressed(_ sender: UIButton) {  
    sender.isEnabled = false  
    let letterString = sender.title(for: .normal)!  
    let letter = Character(letterString.lowercased())  
    currentGame.playerGuessed(letter: letter)  
    updateGameState()  
}  
  
func updateGameState() {  
}
```

How do you determine if a game is won, lost, or if the player should continue playing? A game is lost if `incorrectMovesRemaining` reaches 0. When it does, increment `totalLosses`. You can determine that a game has been won if the player has not yet lost, and if the current game's `word` property is equal to the `formattedWord` (`formattedWord` won't have any underscore if every letter has been successfully guessed). When that happens, increment `totalWins`. If a game has not been won or lost yet, then the player should be allowed to continue guessing, and the interface should be updated.

```
func updateGameState() {  
    if currentGame.incorrectMovesRemaining == 0 {  
        totalLosses += 1  
    } else if currentGame.word == currentGame.formattedWord {  
        totalWins += 1  
    } else {  
        updateUI()  
    }  
}
```

Build and run your application. Is the game functioning properly? It's really close, but a new round does not begin after a win or loss. Whenever `totalWins` or `totalLosses` changes, a new round can be started, so this is a great time to add `didSet` property observers to `totalWins` and `totalLosses`.

```
var totalWins = 0 {
    didSet {
        newRound()
    }
}

var totalLosses = 0 {
    didSet {
        newRound()
    }
}
```

Now try running your application, verifying that both wins and losses are tallied up accordingly, and that a new round begins.

Re-enable Buttons and Fix Crash

It looks like you're approaching the finish line! You're successfully able to complete a round of Apple Pie, but a new round doesn't re-enable the letter buttons. Also, if you try to start a new round and there's no more words to choose from, the game will crash.

The `newRound` logic needs to be a little smarter. If `listOfWords` isn't empty, then you should perform the same work that you did previously, but also re-enable all of the buttons. If there are no more words to play with, disable all of the buttons so that the player cannot continue playing the game.

```
func newRound() {  
    if !listOfWords.isEmpty {  
        let newWord = listOfWords.removeFirst()  
        currentGame = Game(word: newWord,  
                            incorrectMovesRemaining: incorrectMovesAllowed,  
                            guessedLetters: [])  
        enableLetterButtons(true)  
        updateUI()  
    } else {  
        enableLetterButtons(false)  
    }  
}
```

The `enableLetterButtons(_:)` method is fairly straightforward. It takes a `Bool` as an argument, and it uses the parameter to enable or disable the collection of buttons by looping through them.

```
func enableLetterButtons(_ enable: Bool) {  
    for button in letterButtons {  
        button.isEnabled = enable  
    }  
}
```

If you build and run the application, you should be able to get through all of the Apple Pie rounds until the end is reached and the buttons are disabled.

Wrap-Up

Congratulations on building your first mobile game with Swift!

By successfully completing Apple Pie, you've demonstrated an understanding of the building blocks of the Swift language. This project isn't easy, and you should be proud of what you've accomplished. If you struggled to follow along with any of the steps, set aside some time to rebuild Apple Pie on your own, without this guide. A second run-through will point out areas that you may not have understood—and you can look to the guide and earlier lessons to reinforce your knowledge.

Stretch Goals

If you'd like to continue working on Apple Pie, go ahead and try adding some features to the game. Here are a few ideas to play with. You can build most of these features using your existing knowledge of Swift, but a few may require you to use the Xcode documentation. Good luck!

Challenge yourself by adding these features to Apple Pie:

- Learn about the `map` method, and use it in place of the loop that converts the array of characters to an array of strings in `updateUI()`.
- Add a scoring feature that awards points for each correct guess and additional points for each successful word completion.
- Allow multiple players to play, switching turns after each incorrect guess.
- Allow the player to guess the full word using the keyboard instead of guessing one letter at a time using the interface buttons.
- Support letters with special characters. For example, the E button could check for “e” and “é” within a word.
- The keyboard layout doesn't work well when the app is in one-third Split View mode on iPad—the buttons get flattened. To resolve this issue, use trait variations to adjust the layout when in compact width.

Lesson 2.12

Finish Your App Prototype

You have learned tools to help you work with information in your app, like structures and loops. And you've learned how to start implementing some of the visual elements of your app with views and controls. All these tools can help you start building your actual app once you have the finalized app structure and style.

In this lesson, you'll finish up your app prototype using the App Design Workbook. You'll take the map you created and apply basic UI elements to create a wireframe. Then you'll refine your prototype using common design guidelines and define the personality of your app with color, icons, and more.

What You'll Learn

- How to create a wireframe prototype in a Keynote prototype
 - How to refine your prototype by focusing on UI elements
 - How to define a personalized style for your app
-

Related Resources

- [WWDC 2021 Discoverable Design](#)
- [WWDC 2021 The Practice of Inclusive Design](#)
- [Apple Style Guide for Writing Inclusively](#)
- [Human Interface Guidelines](#)

Guide

Wireframe

Now that you have a skeleton for your prototype, it's time to make it more formal. Continue to ask yourself what the key interactions and UI elements for the goals of your app are. These are the elements you want to build into your prototype.

Work through the Wireframe section of the App Design Workbook. You'll build a wireframe from your app's architecture map by converting screen outlines into a sketch of the interface. By the end of this stage, you'll have a functioning Keynote prototype that simulates the behavior of your app.

Refine

After developing a basic wireframe, it's time to mimic the experience of an iOS app by considering how users will expect it to look and feel.

Use the Refine section of the App Design Workbook to apply important interface design guidelines to your functioning prototype. By the end of this stage, your prototype will feel at home on iOS and in the hands of your users.

Style

From your icon to the fonts and colors you use, each design element is an opportunity to influence the user's experience. Now it's time to define the personality of your app to set it apart from its peers. Use your imagination to create a cohesive identity for your app.

Complete the Style section of the App Design Workbook. By the end of this stage, you'll have a Keynote prototype that feels like a real app, which will enable your testers to provide good feedback.

Summary

You made it! In this unit, you learned about many new programming concepts and how the Swift language puts them to use. You've learned the basics of `UIKit`, the foundation of iOS development. You've learned how to display simple information with views, and respond to user input with a variety of controls. You also learned about the powerful tools in Xcode that enable you to build slick-looking interfaces that fit within the iOS environment.

After completing this unit, you should be ready to build much more interesting apps. Find out what's next.

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.

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.

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.

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.

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.

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

Question 1 of 3

Which of the following would be best represented with an enumeration?
(Choose all that apply.)

- A. Names of people in a room
- B. Political parties
- C. Addresses
- D. Compass degrees

Check Answer



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

- | | |
|--|---|
| <ul style="list-style-type: none">• bar button• modal presentation• modal segue• navigation bar• navigation controller• pop | <ul style="list-style-type: none">• 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.

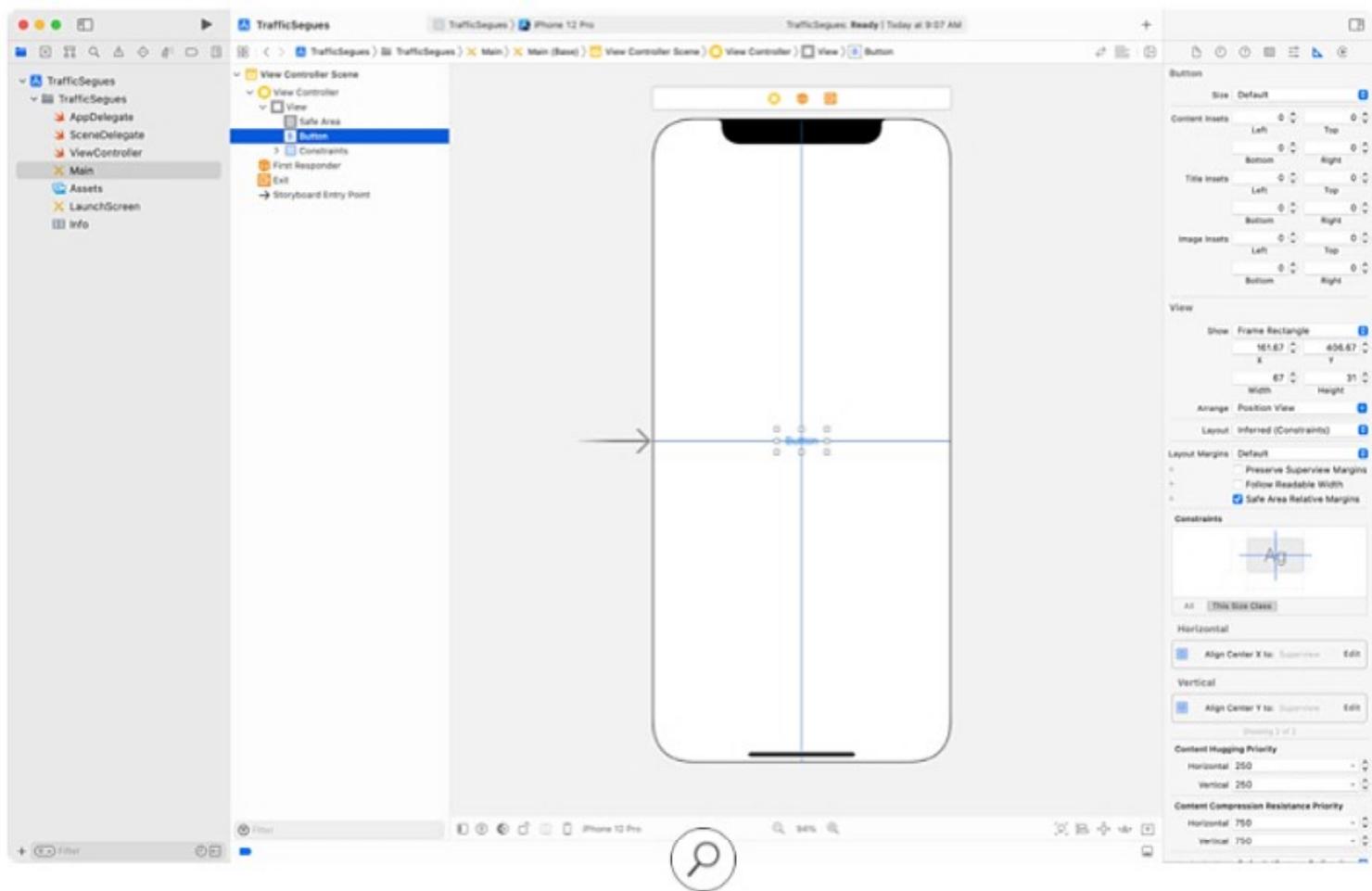
In addition to the transition, a segue also defines the presentation method of the view controller. One common method is a modal presentation, which places a new view controller on top of the previous one. On smaller screens, a modal presentation will always appear at full screen. To adapt the UI for 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.

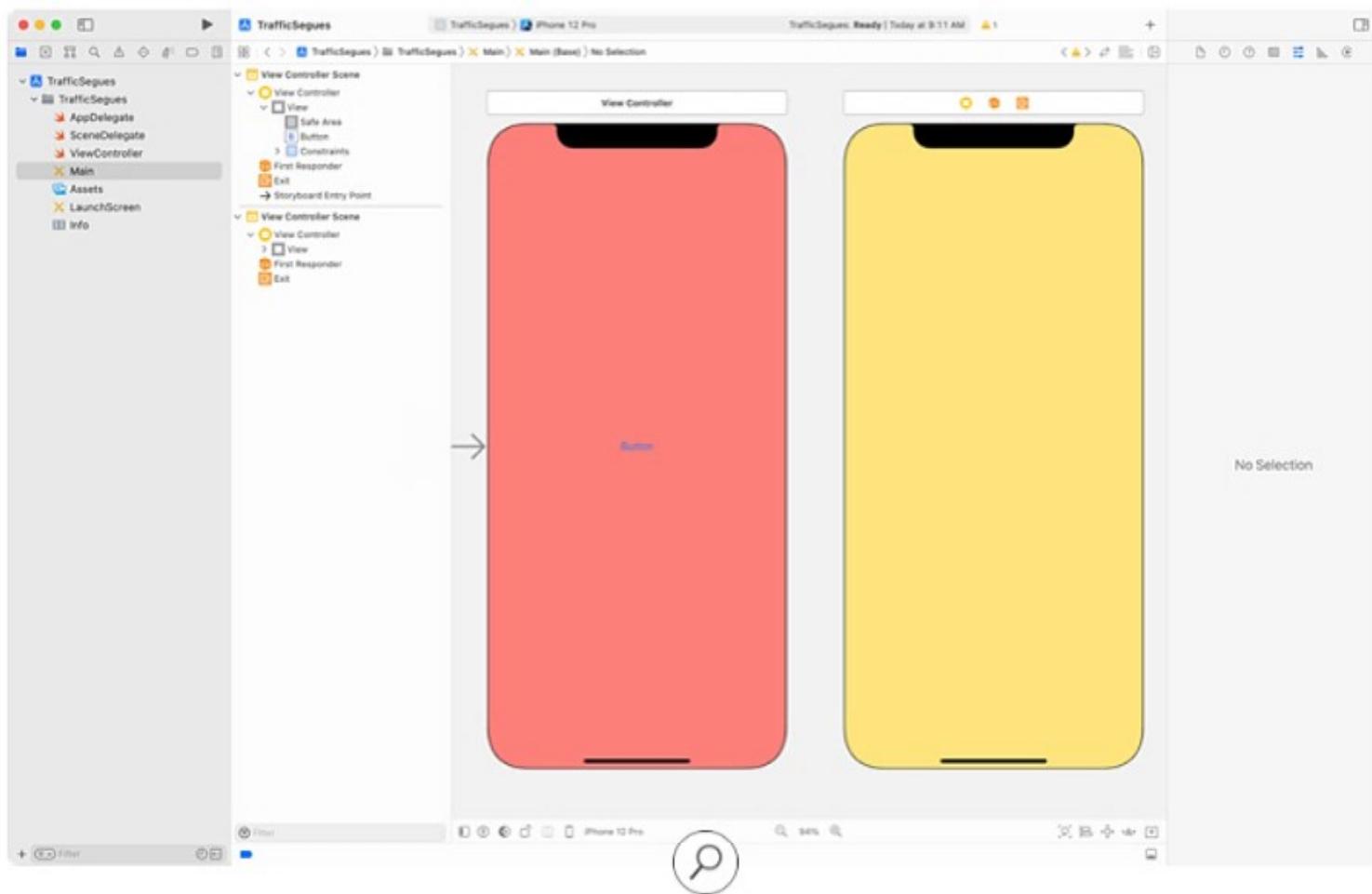
When a new view controller is presented modally, you can use an unwind segue to allow the user to dismiss the new view controller and return to the previous one.

Create Triggered Segues

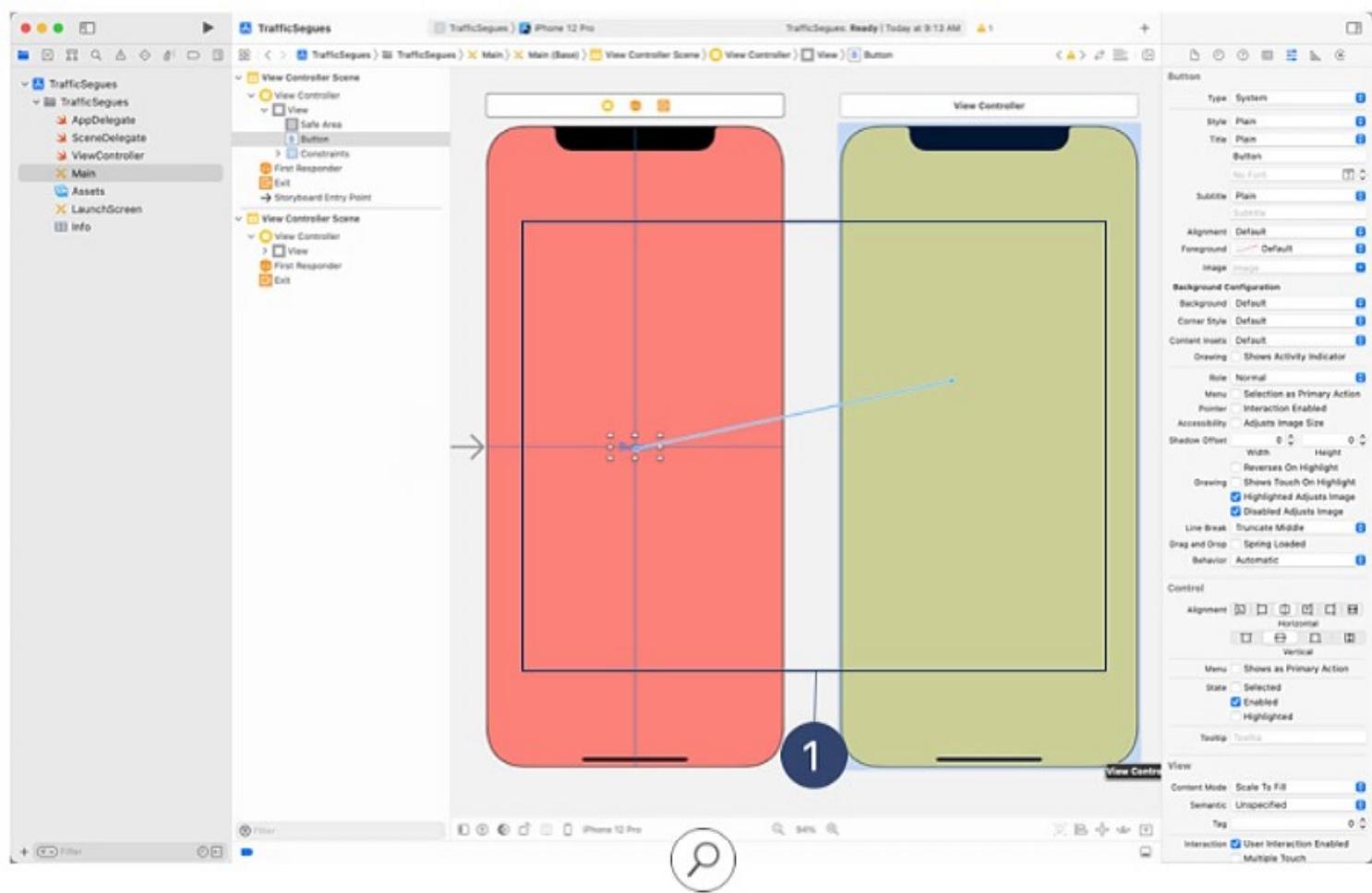
To practice transitioning between view controllers, you'll create a simple app that cycles through the different colors of a traffic light. Start by creating a new Xcode project using the iOS App template. Name the project "TrafficSegues." When creating the project, make sure the interface option is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



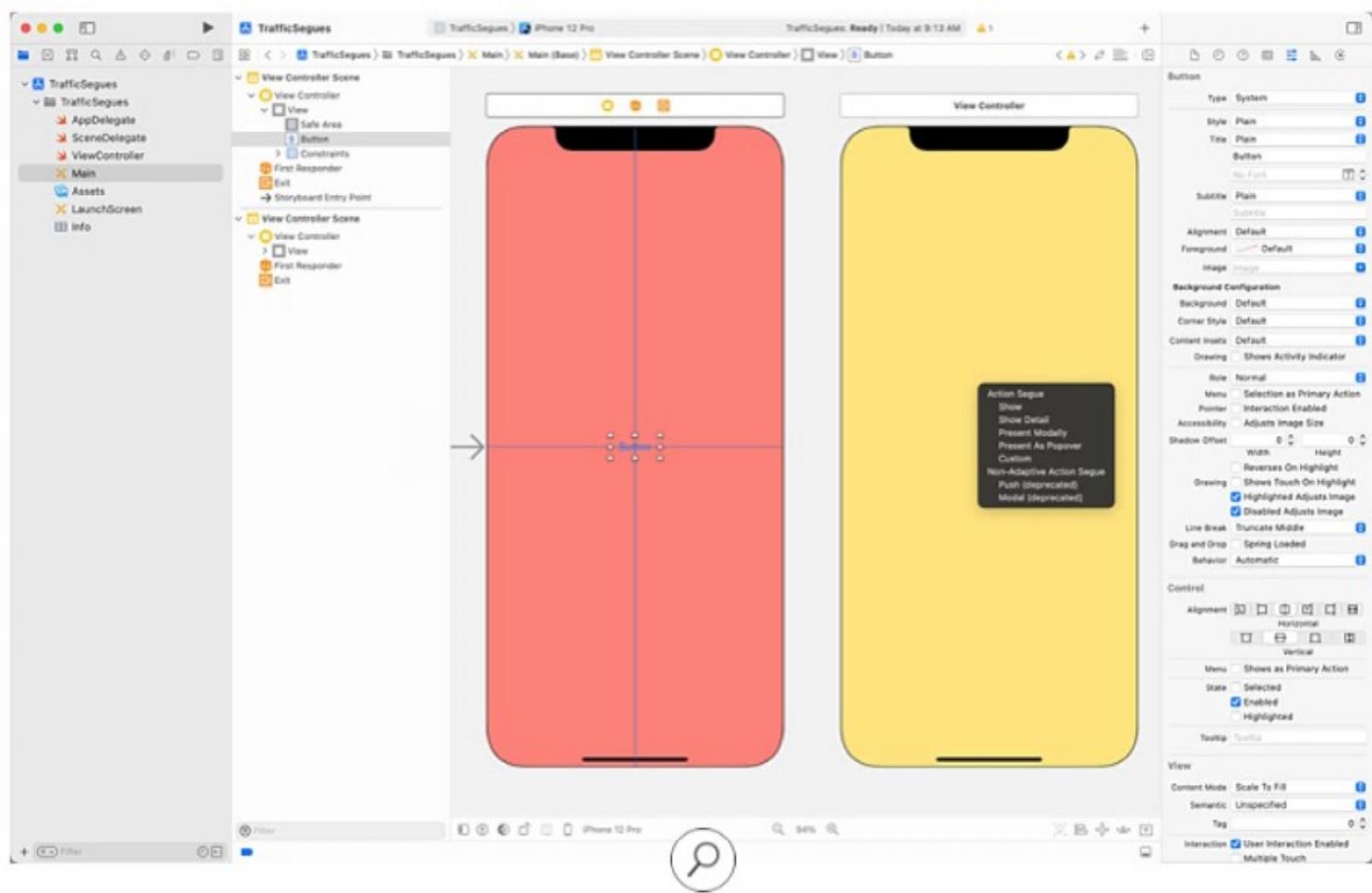
Add a `UIButton` to the center of the view, using the alignment guides to help position it. Click the Align button and select “Horizontally in Container” and “Vertically in Container” to create two constraints that center the button for all screen sizes.



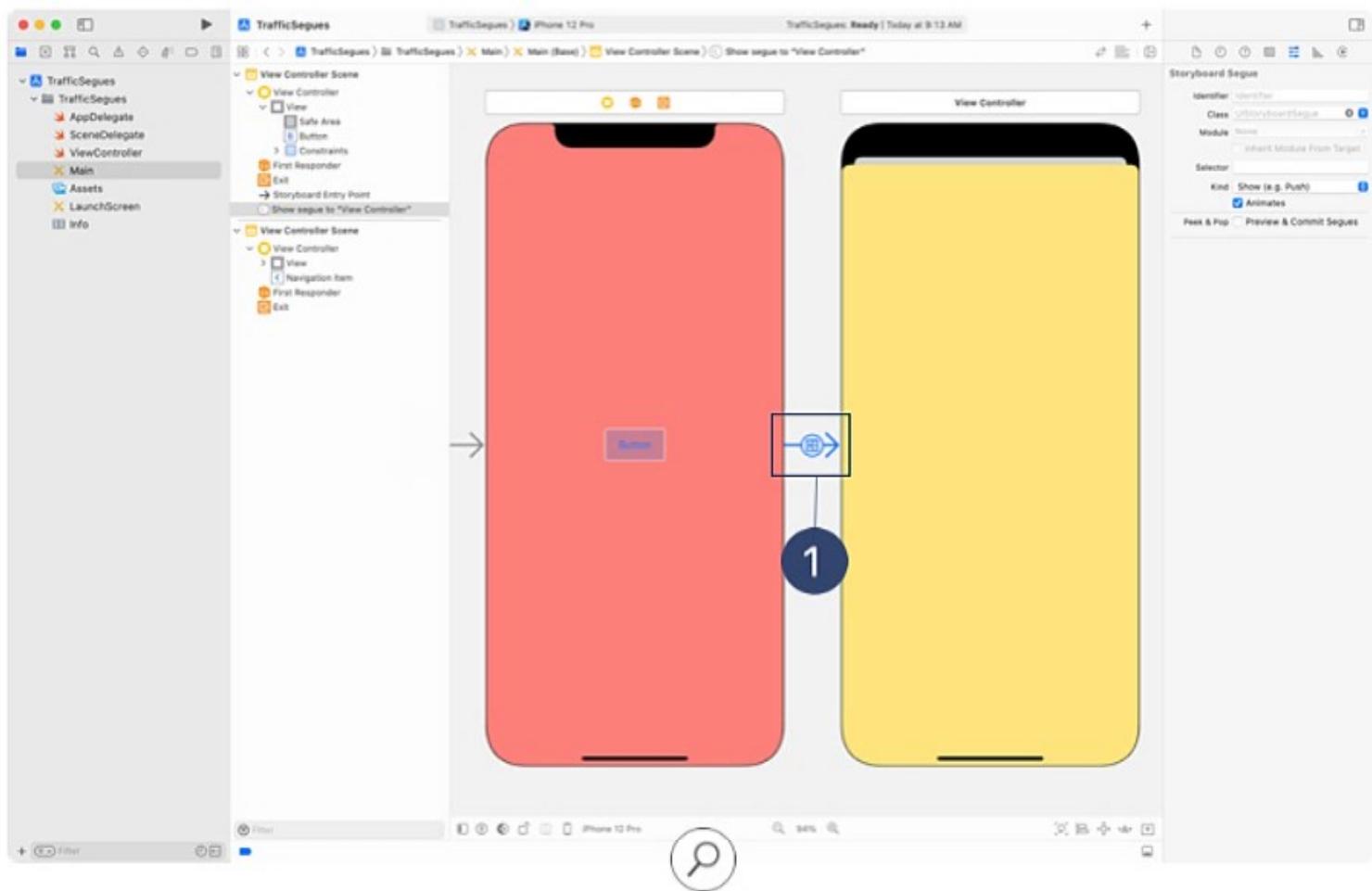
Locate View Controller in the Object library, and drag this object onto the canvas, positioning it to the right of the first view controller. Using the Attributes inspector, give the left view controller a red background and the right view controller a yellow background. (The screenshots use lighter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



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.

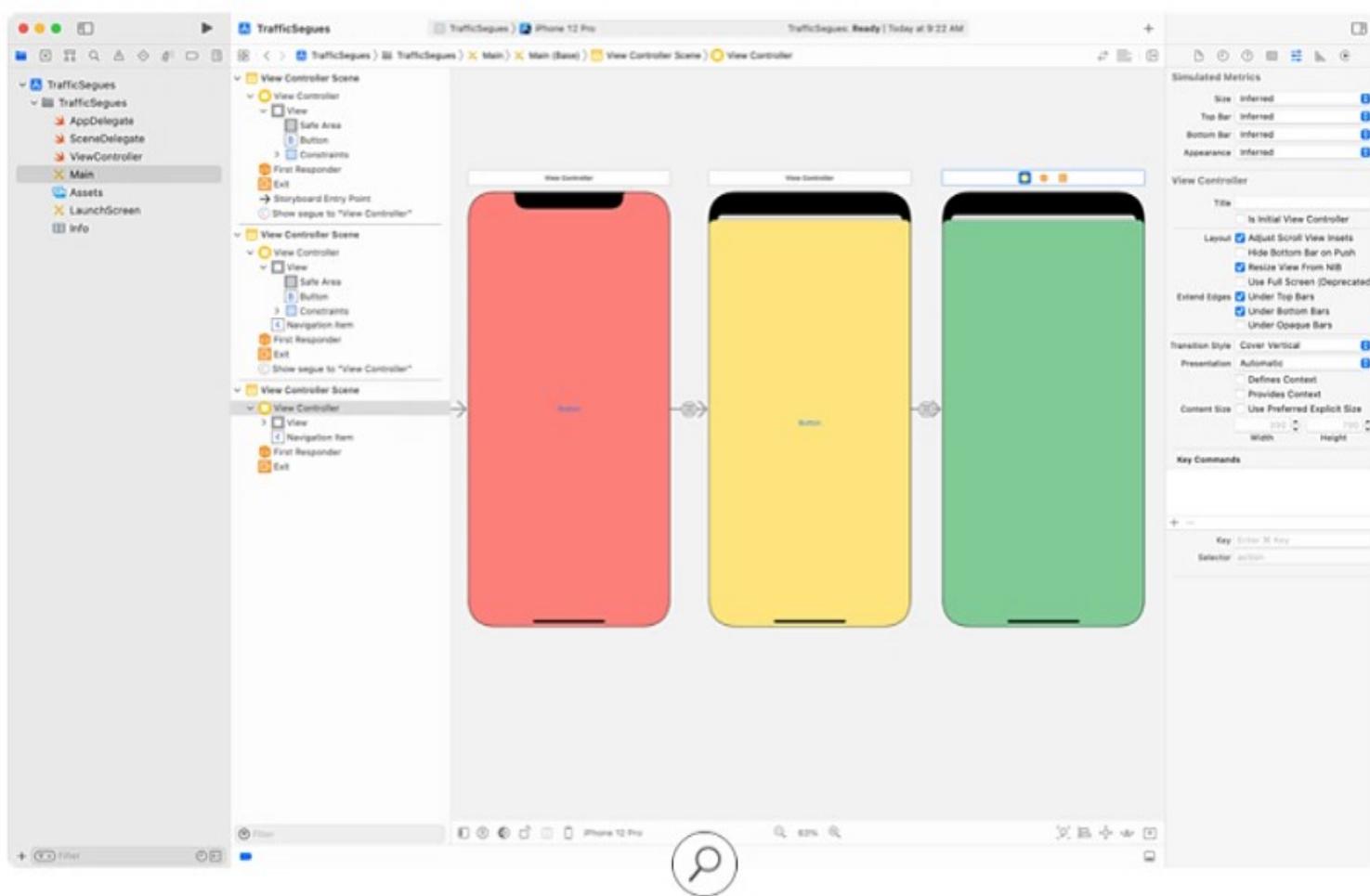


The Show segue also presents modally until a navigation controller is added to the storyboard scene. You'll add one of these later in the lesson. For now, select Show. An arrow appears from the red view controller to the yellow view controller indicating the segue. ①

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.

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.

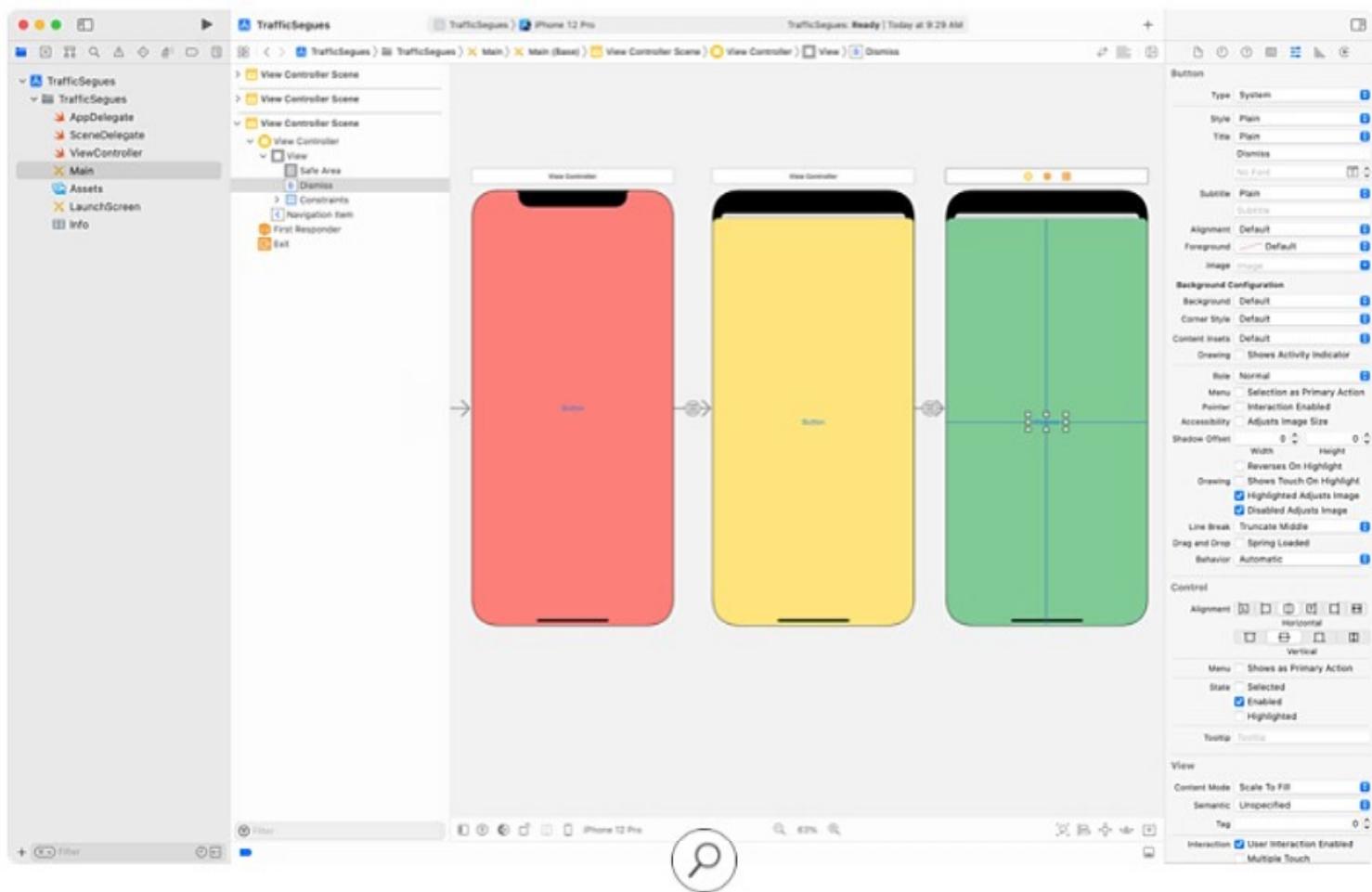
To begin, select `ViewController` in the Project navigator and add the following method just below the `viewDidLoad()` function:

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

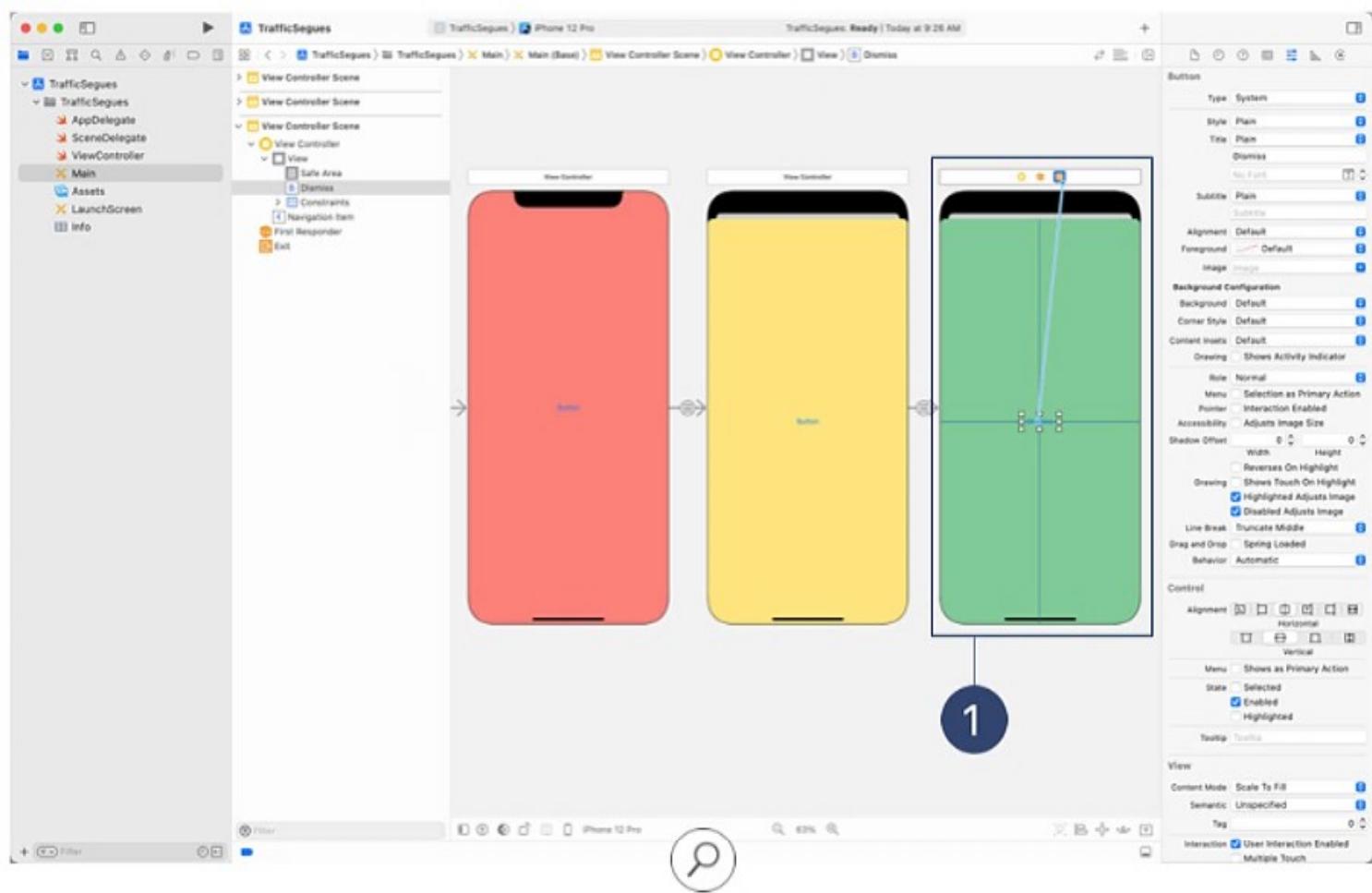
You can name the method anything you like, but it must take `UIStoryboardSegue` as its only parameter.

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

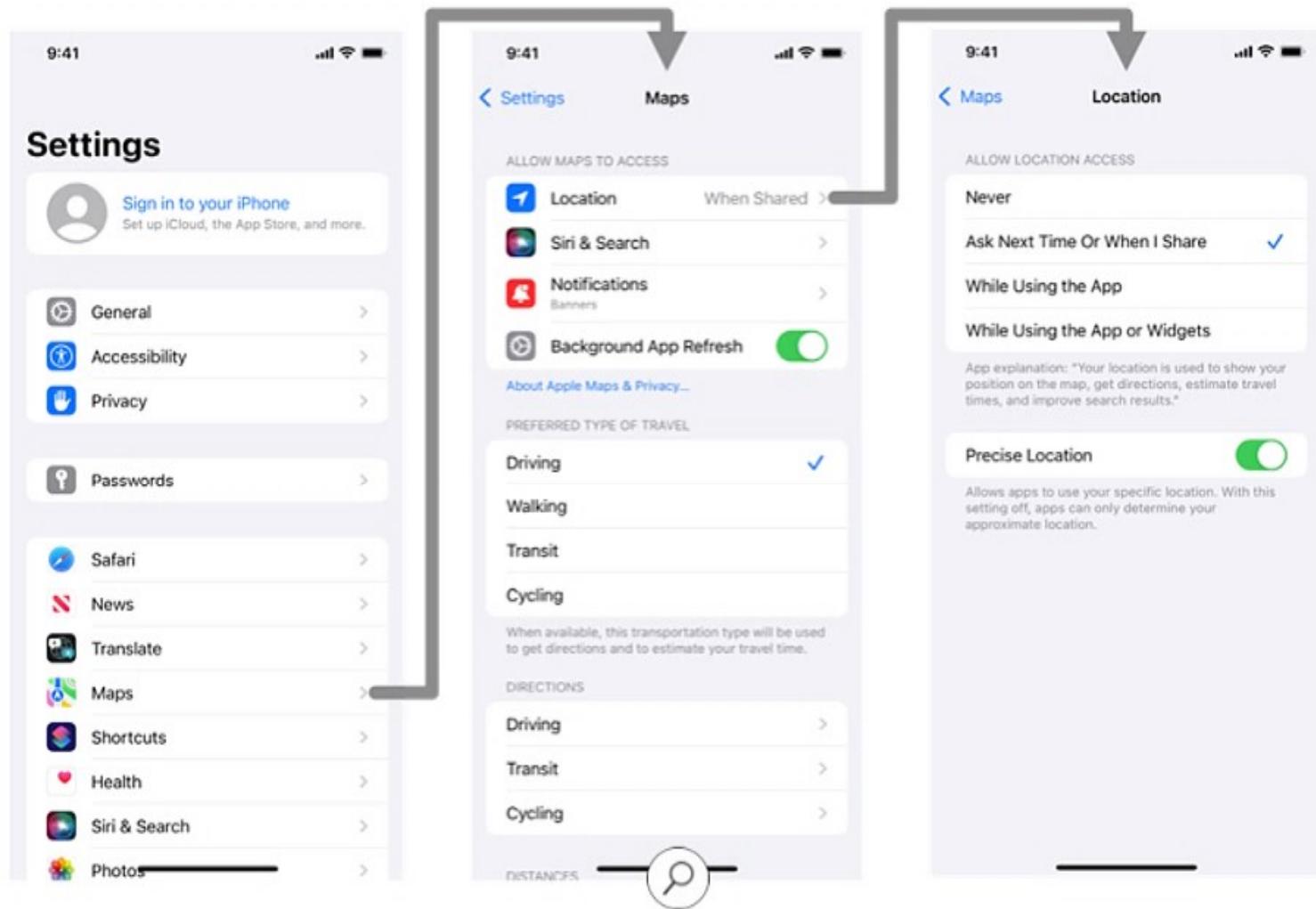
Modal segues are the preferred method of transitioning from one context to another within your app. For example, in the iOS Mail app, tapping the Compose button  transitions from reading messages to writing messages. The Cancel button is always available if the user chooses to return to a previous context.

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.



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.



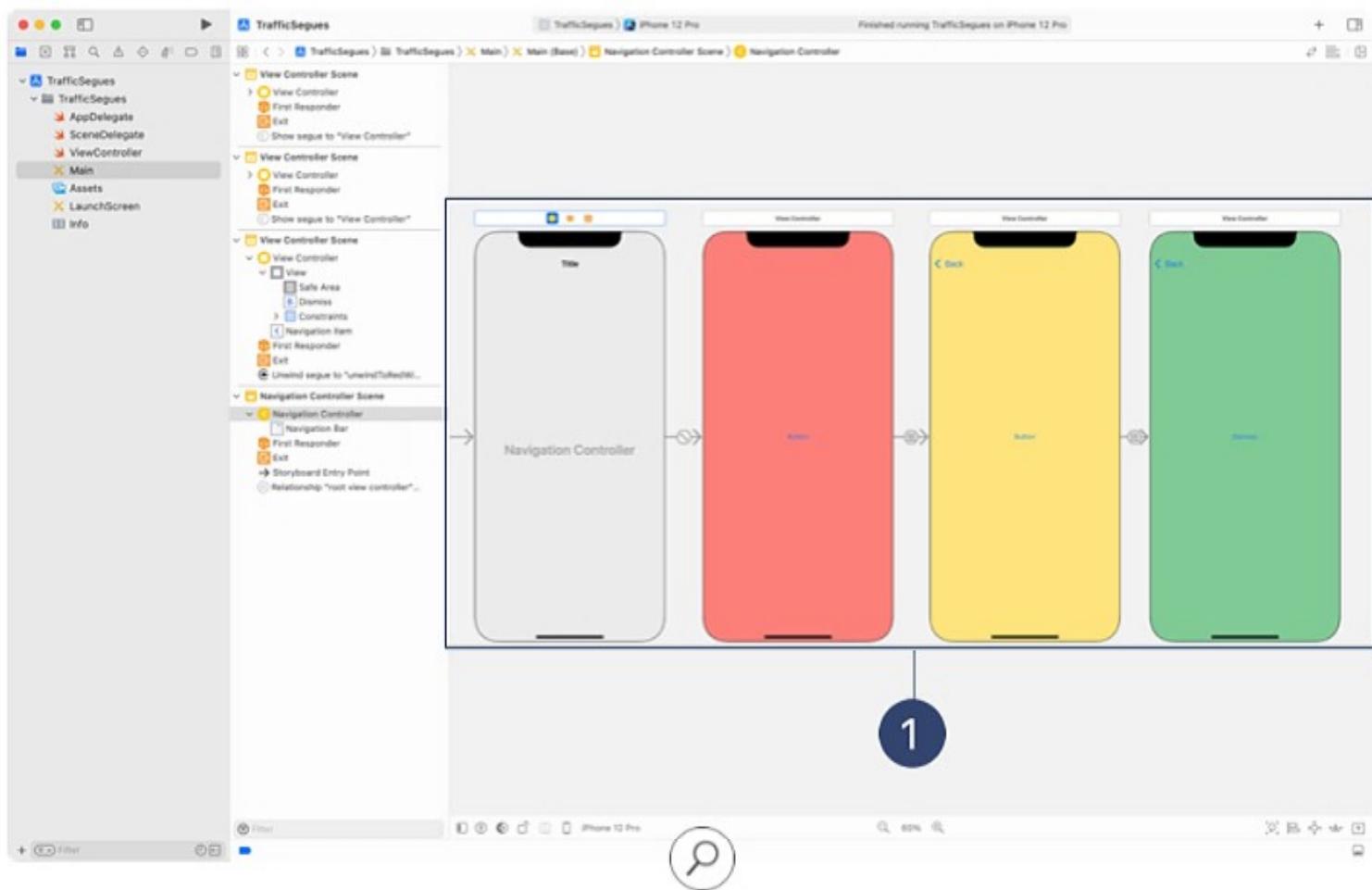
This push-and-pop structure is like washing a stack of dirty plates. After you wash and dry each plate, you place it in the cabinet. The first plate you wash will be at the bottom of the stack, and the last plate will be on the top. Later, when you grab a clean plate from the cabinet, you'll be grabbing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

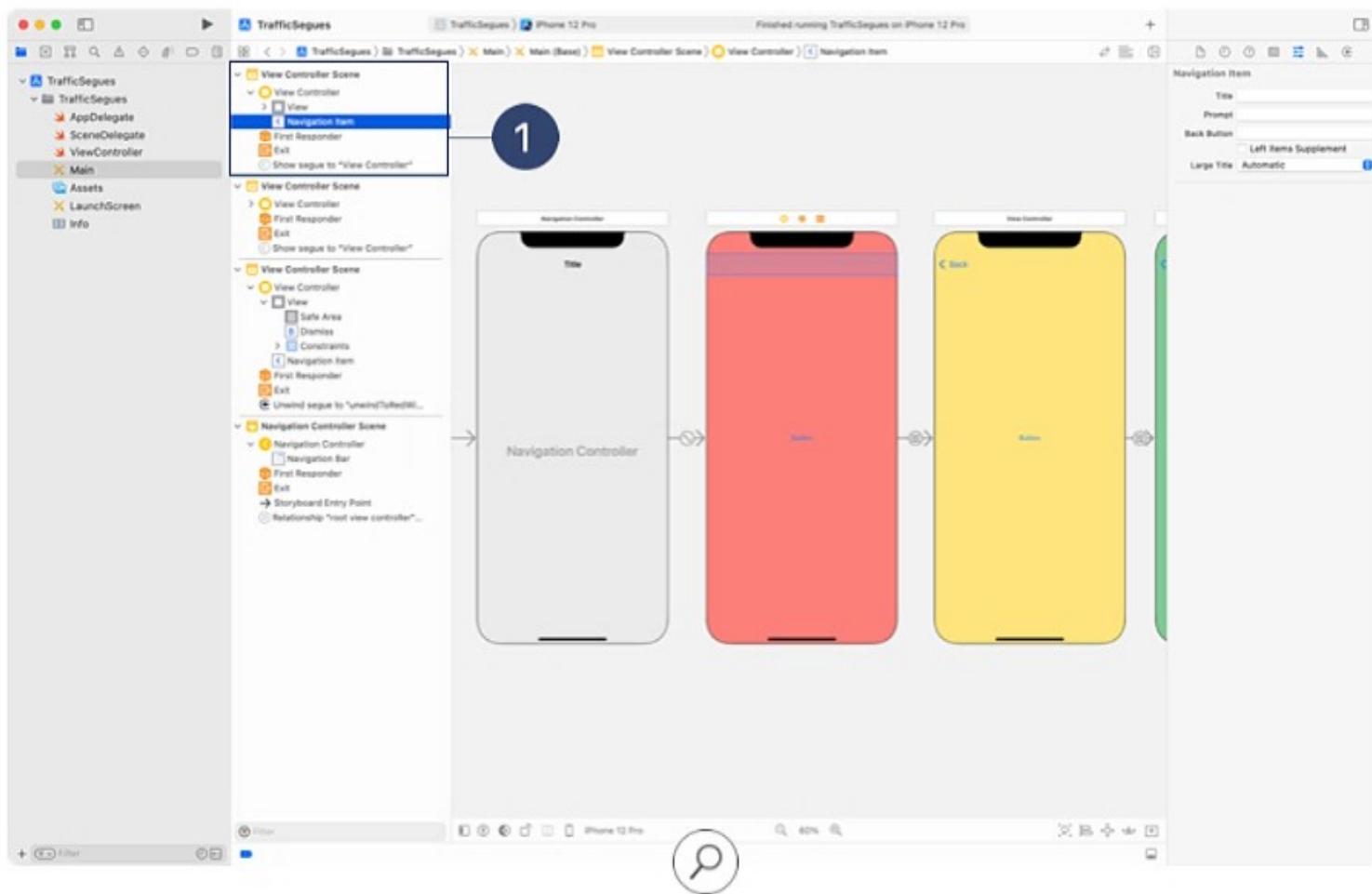
Navigation Bar

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.

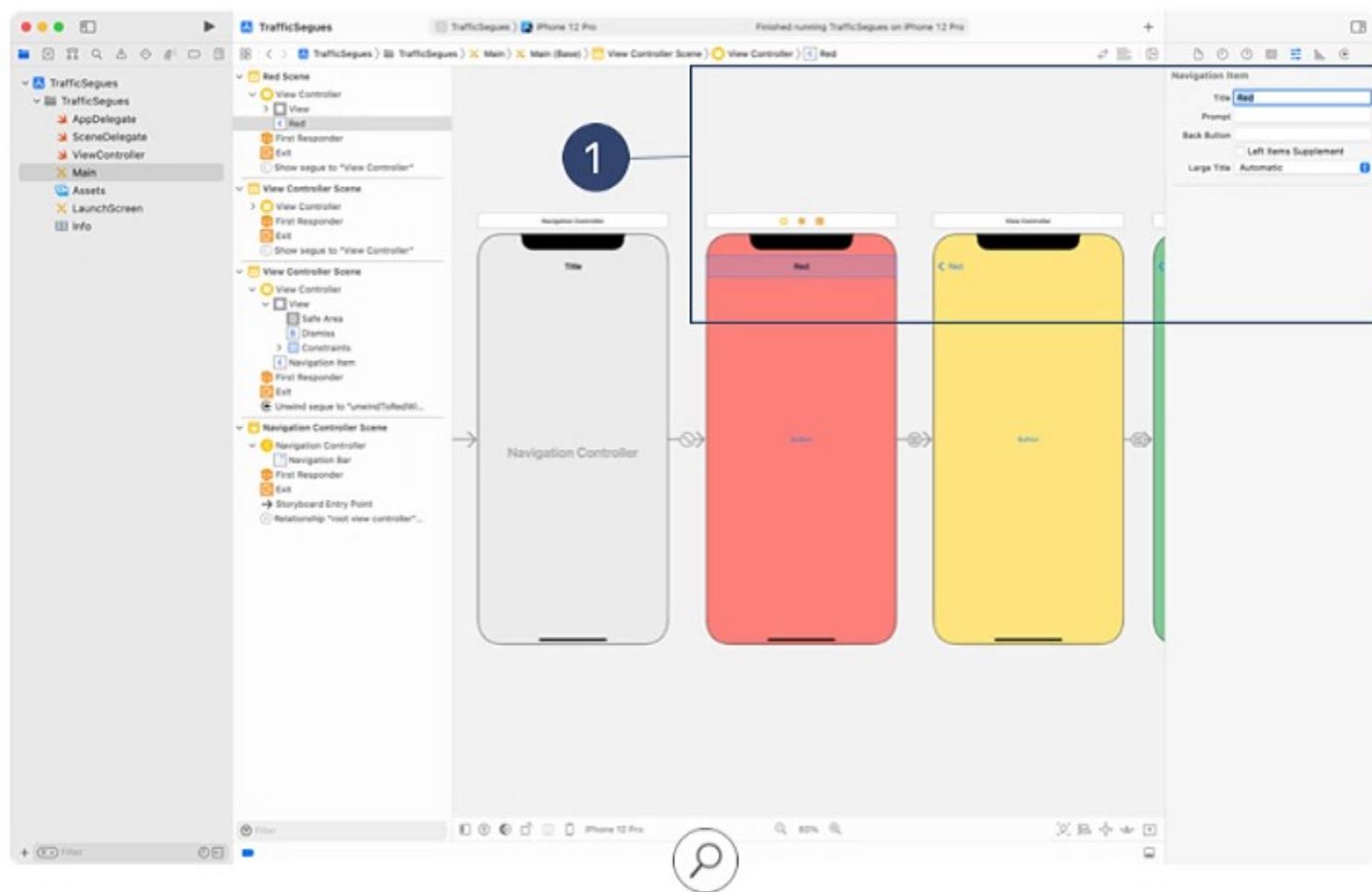
Select the navigation bar in the Document Outline, and check out the Attributes inspector to see what properties you can customize, such as the bar's tint color, title color, and title font. (You might also choose to modify these properties in code.) View the documentation for `UINavigationBar` for a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

Navigation Item

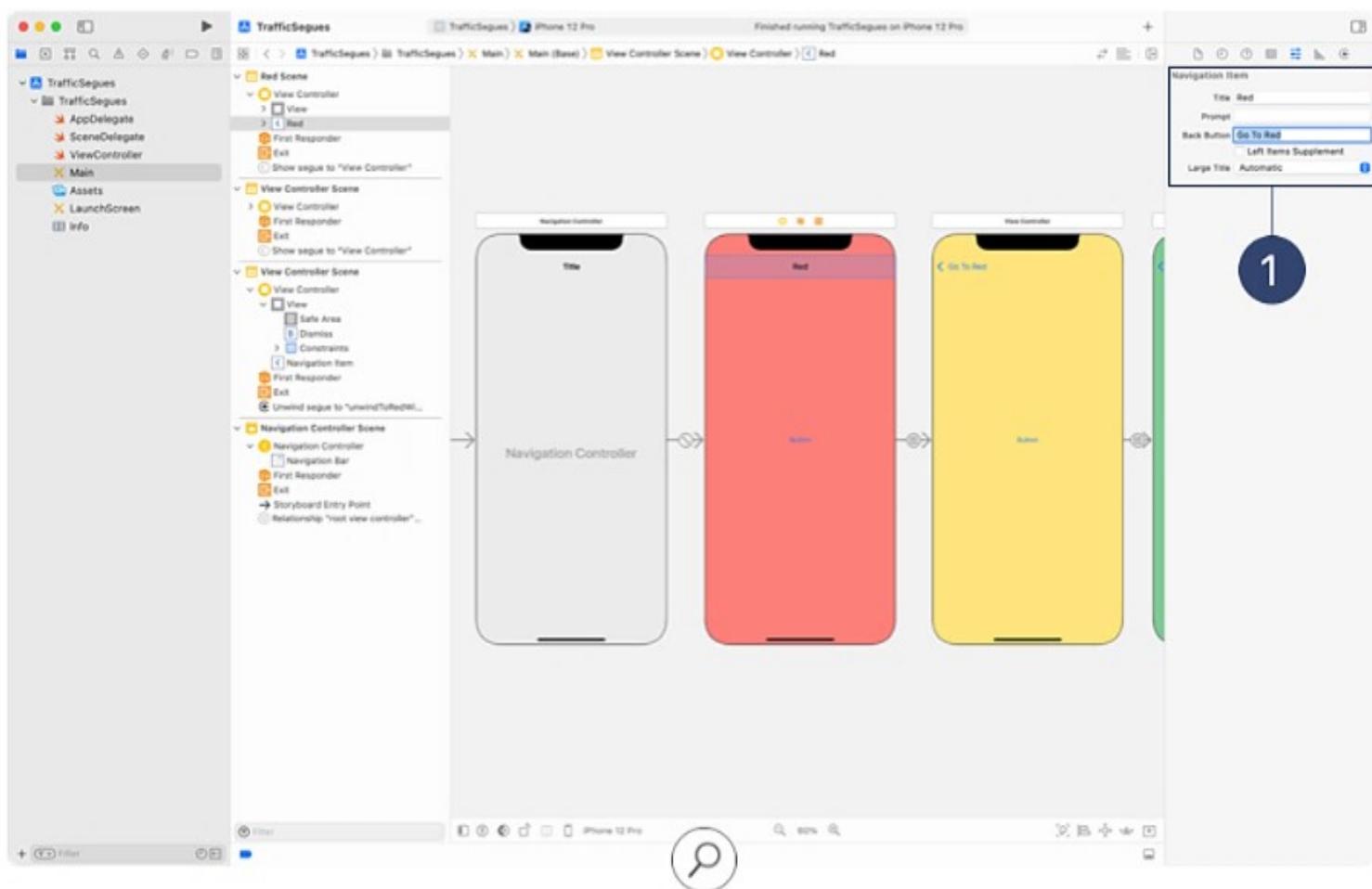
Every `UIViewController` has a `navigationItem` that you can use to customize its navigation bar.^① When you added the navigation controller in the earlier step, Interface Builder automatically added a navigation item to the root (red) view controller.



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



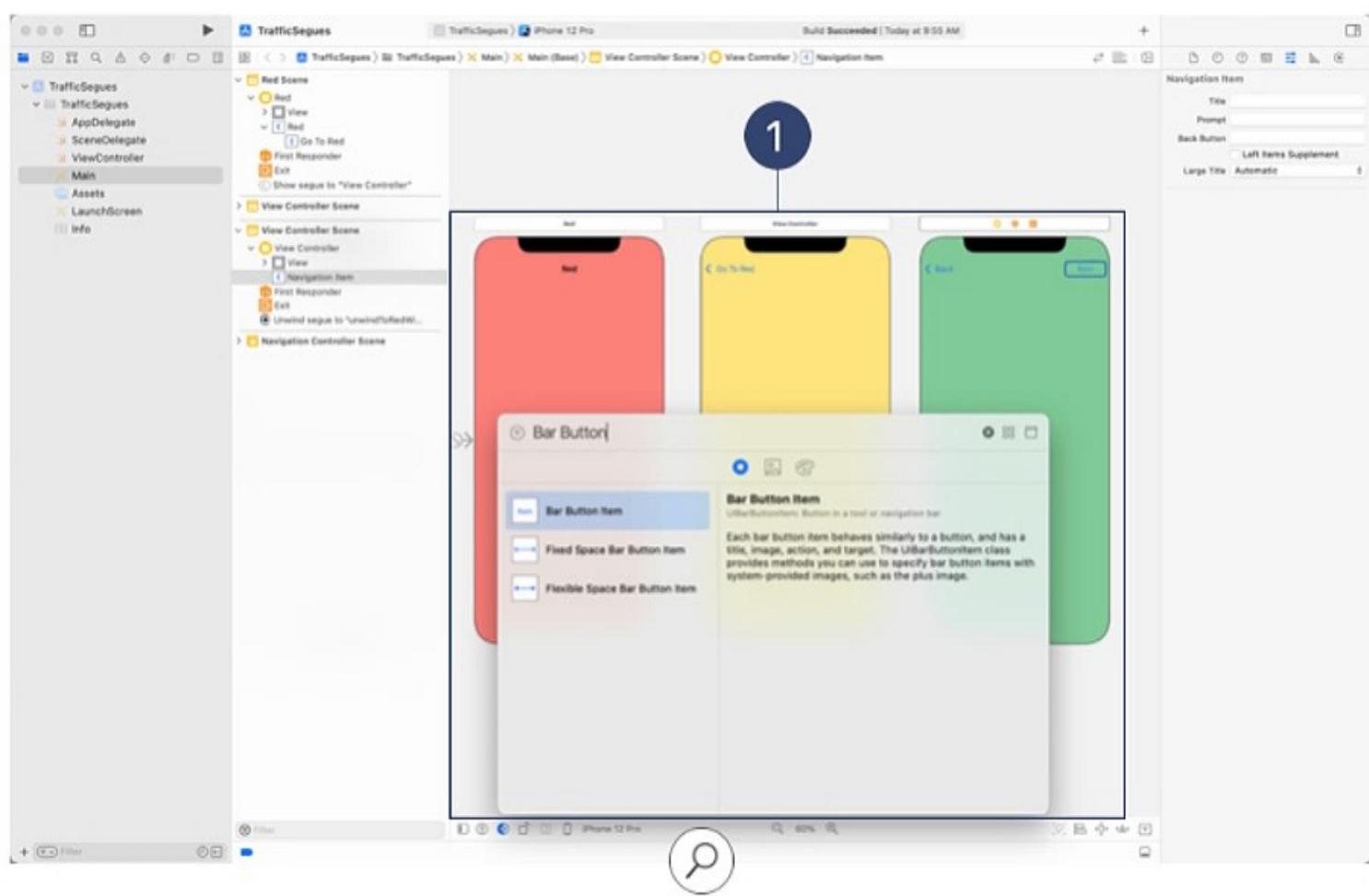
Build and run the app. When you push to the yellow screen, you'll notice that the Back button now displays "Red." How did that happen? The Back button used the title of the preceding view controller as its text. However, if the preceding view controller doesn't have a title, the Back button simply displays "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

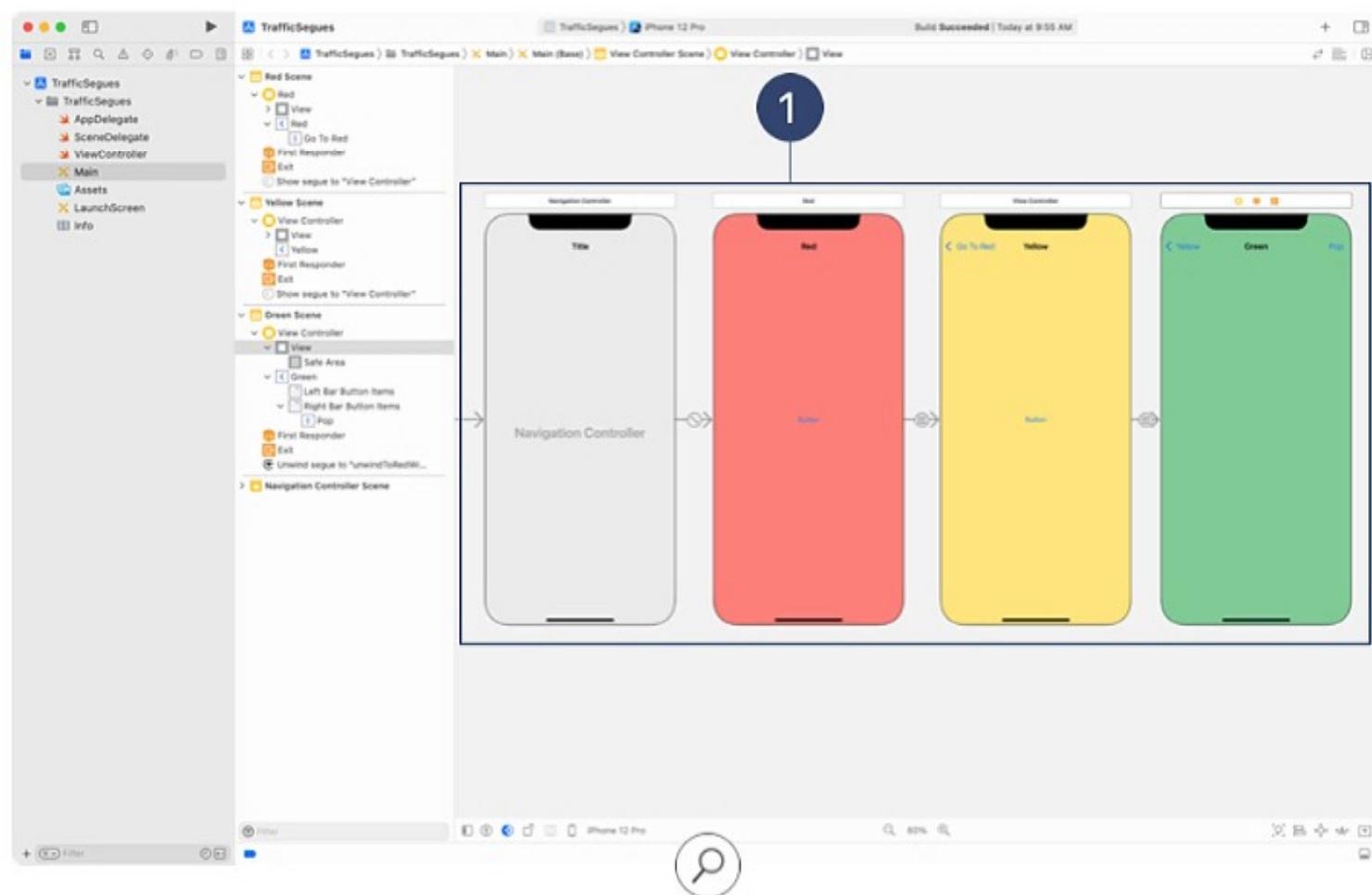


Click the bar button you just added, and open the Attributes inspector. In the System Item popup menu, you'll see commonly used button choices, such as Add, Save, and Cancel. Choose one or two to see the button text change. Play with the Style and Tint attributes as well. The Bar Item properties allow 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.)

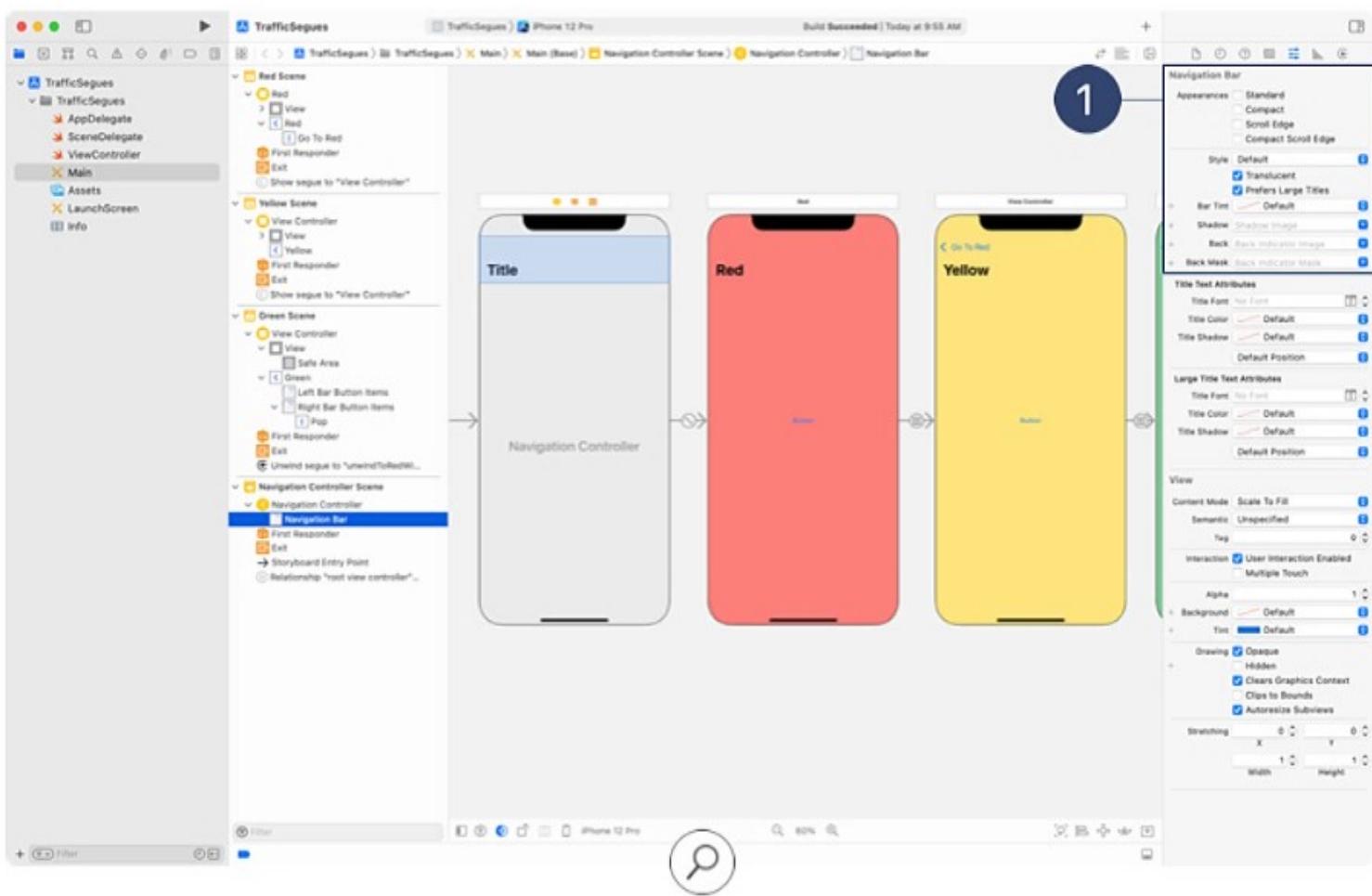
Next, you can wire this new bar button to the unwind segue. Refer to the same steps you used to connect the Dismiss button. Once you're done, you can delete the Dismiss button.

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.

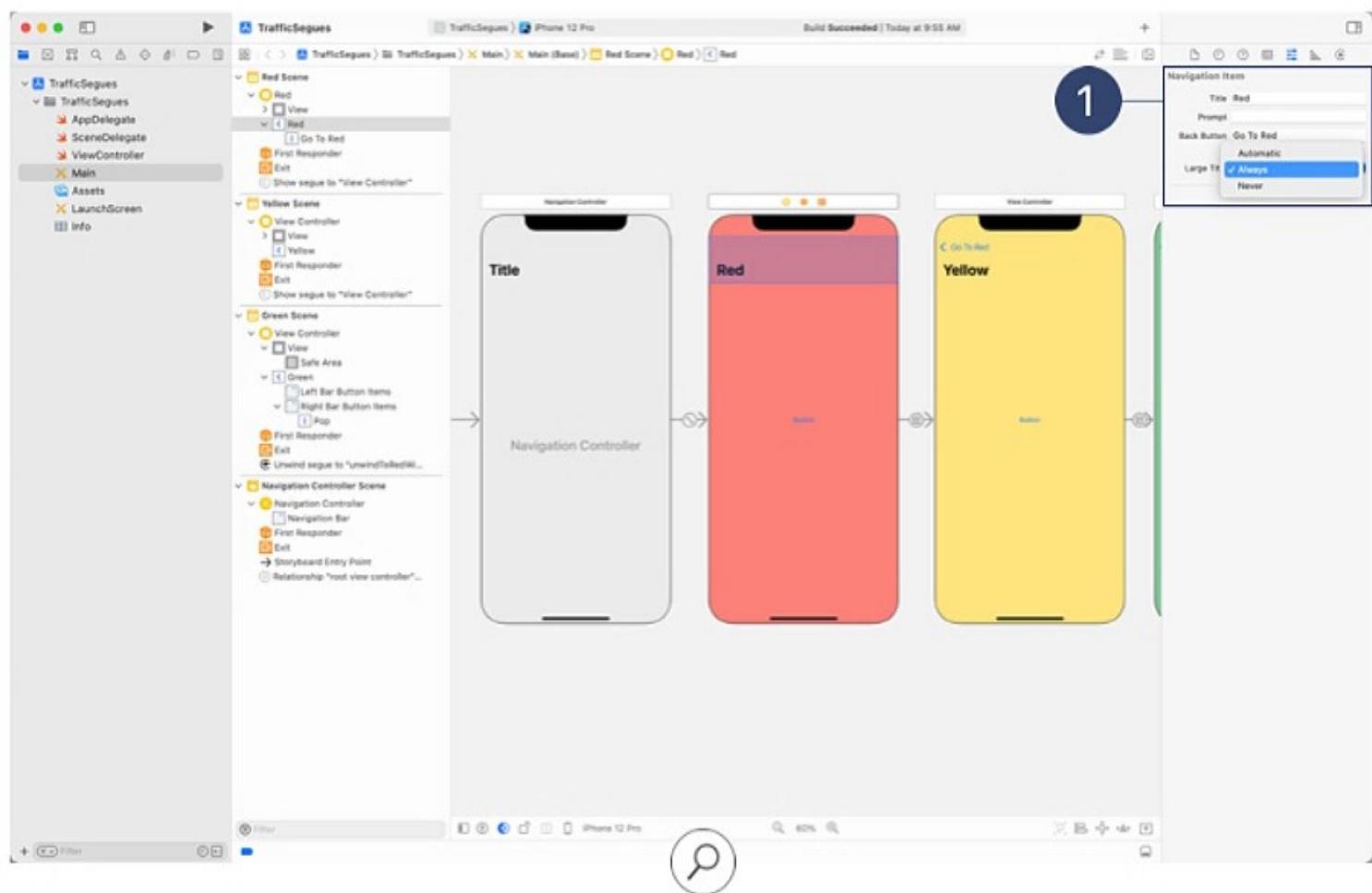


Large Titles

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



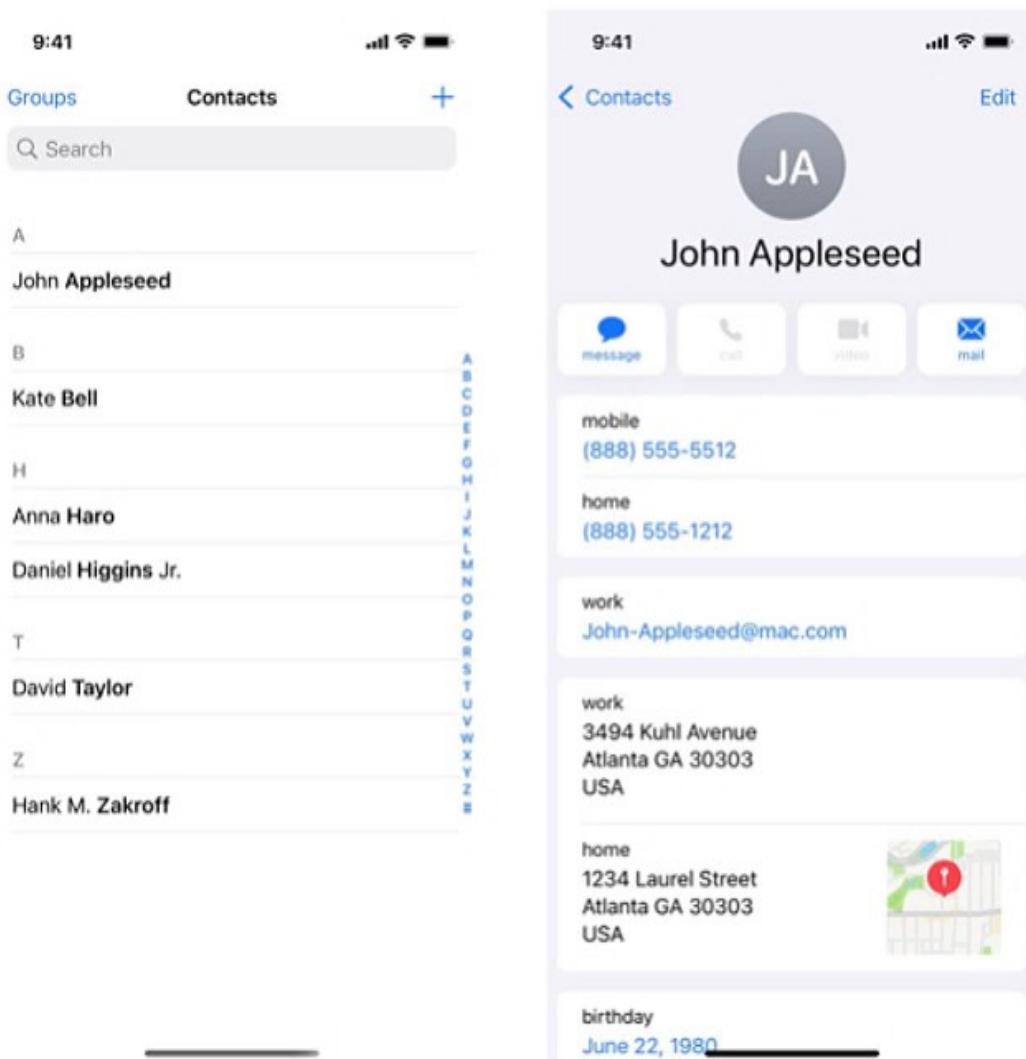
Note that all of your titles, not just the title of the first view controller, have now adopted a large title. You can adjust which view controllers use the large title option by selecting the navigation item in question and selecting one of the options from the Large Title dropdown in the Attributes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

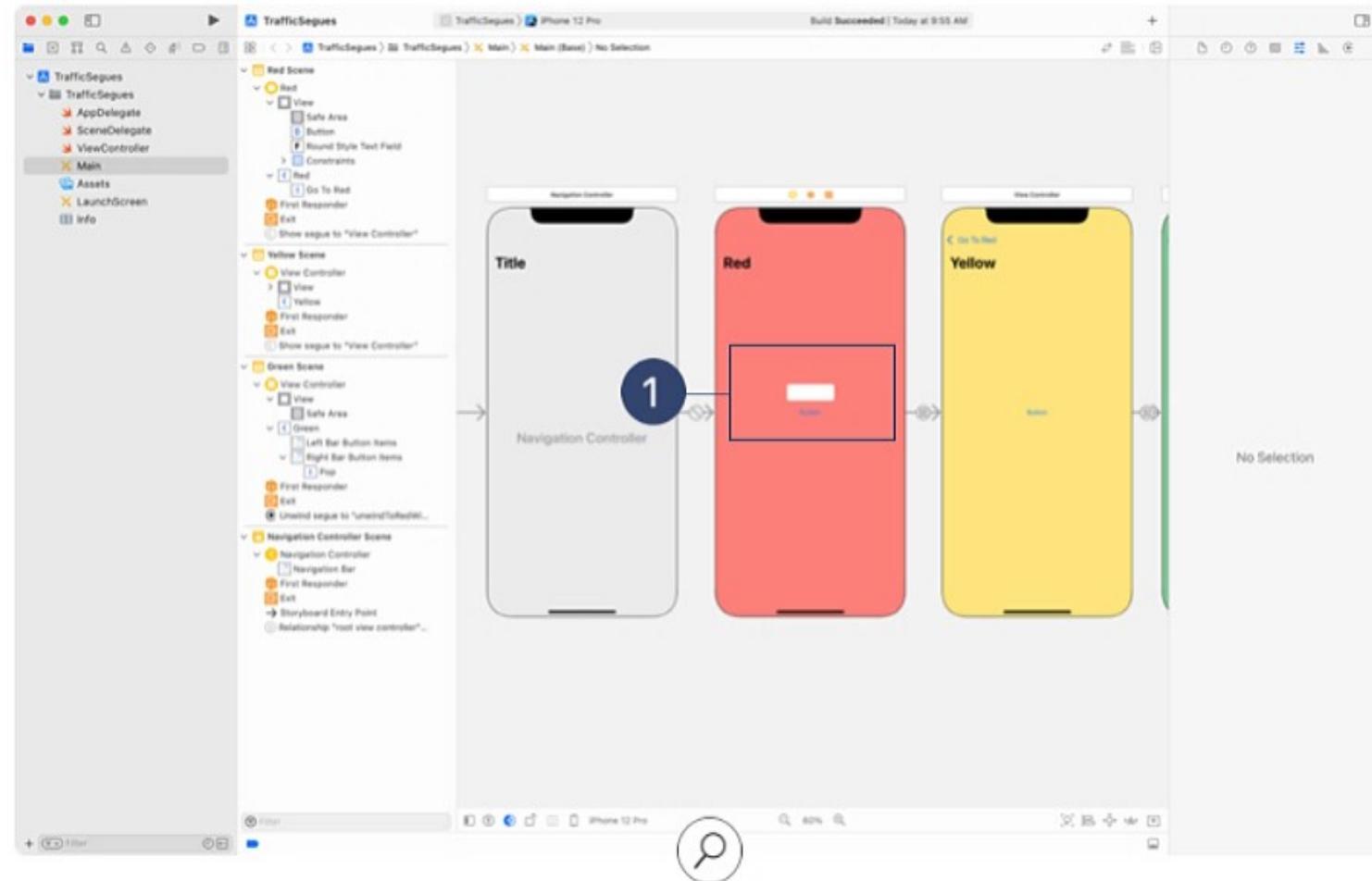
Pass Information

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.

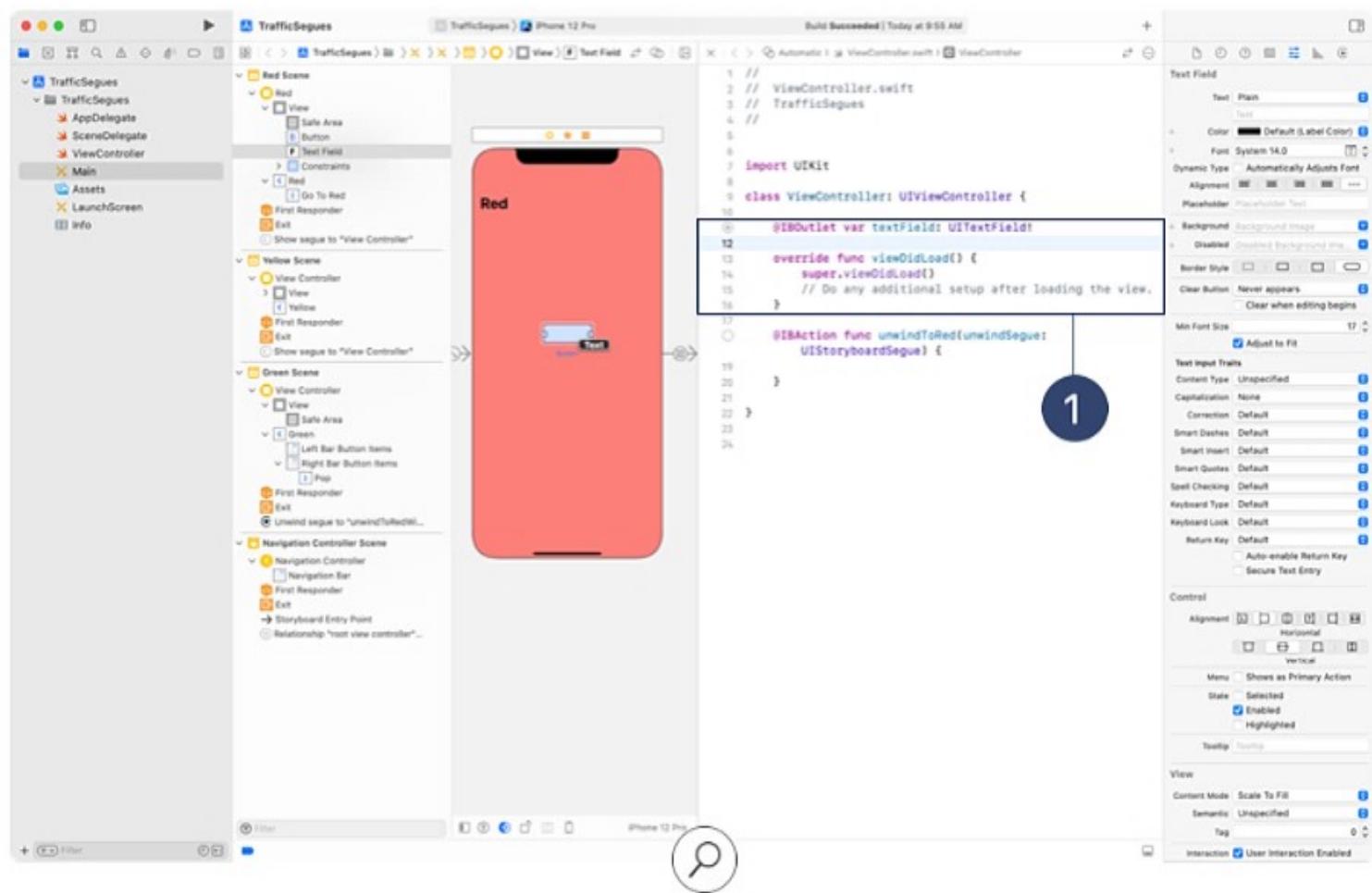


To learn how this works, you'll update the red view controller to include a text field in addition to the button. When the button is pressed, triggering the segue, Interface Builder will use any text in the text field as the title for the yellow view controller's navigation item.

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.

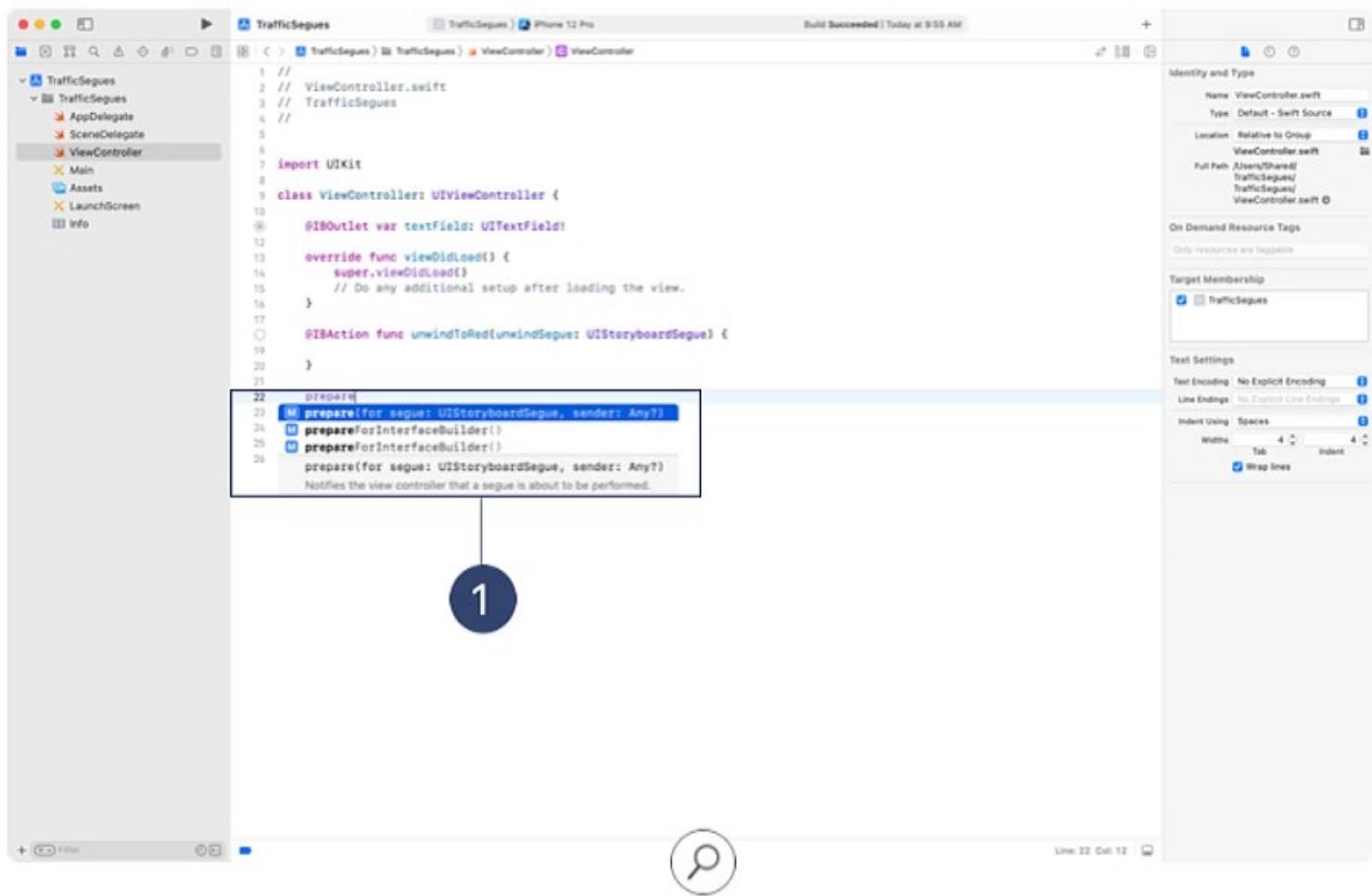


Create an outlet for the text field, naming it "textField."^① You'll need to refer to the text field in code so that you can access the text and update the destination's title accordingly.



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  
}
```

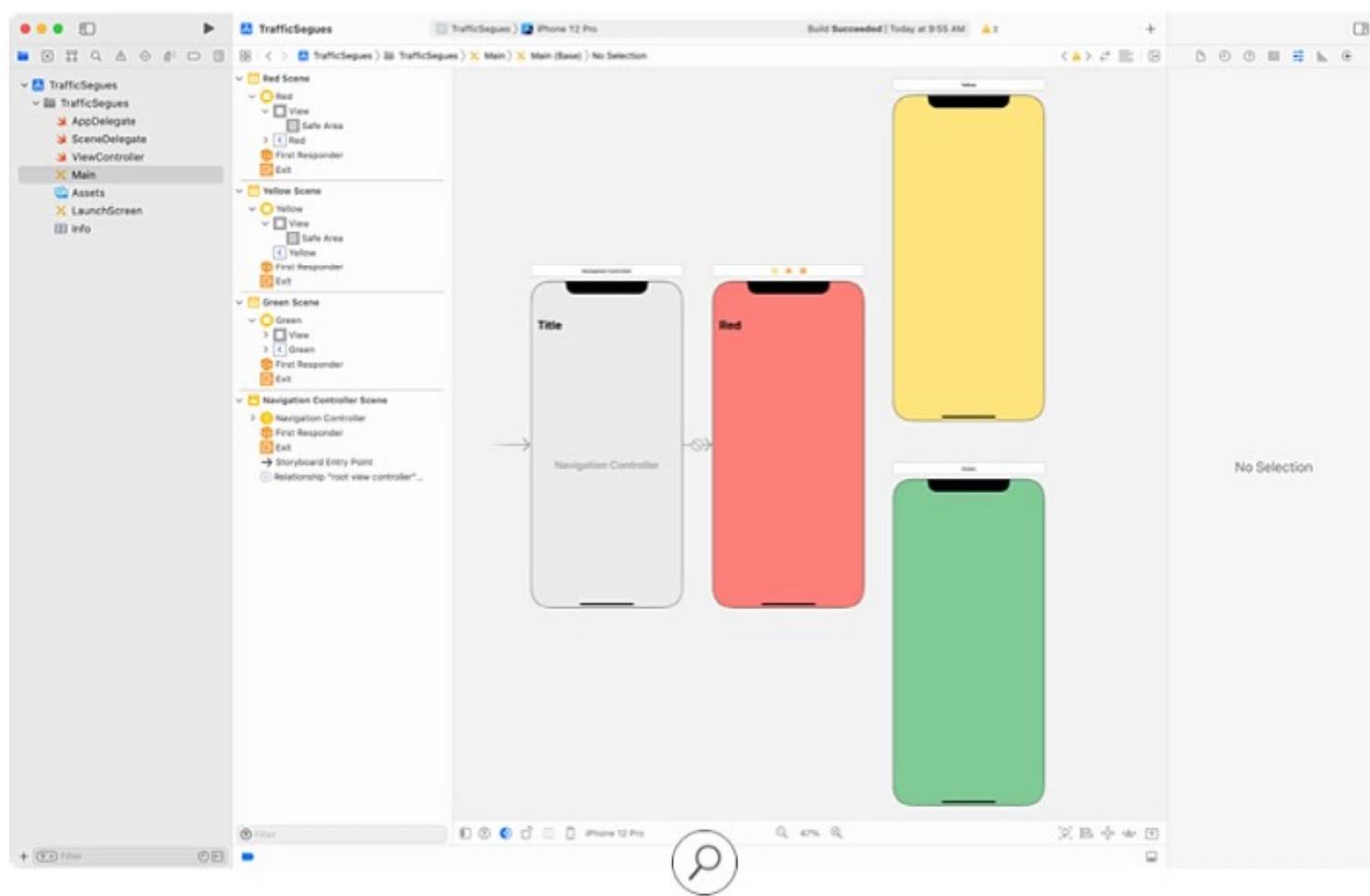
Build and run your app. Then enter a short string into the text field. When you press the button, the method you just added will update the title on the yellow screen.

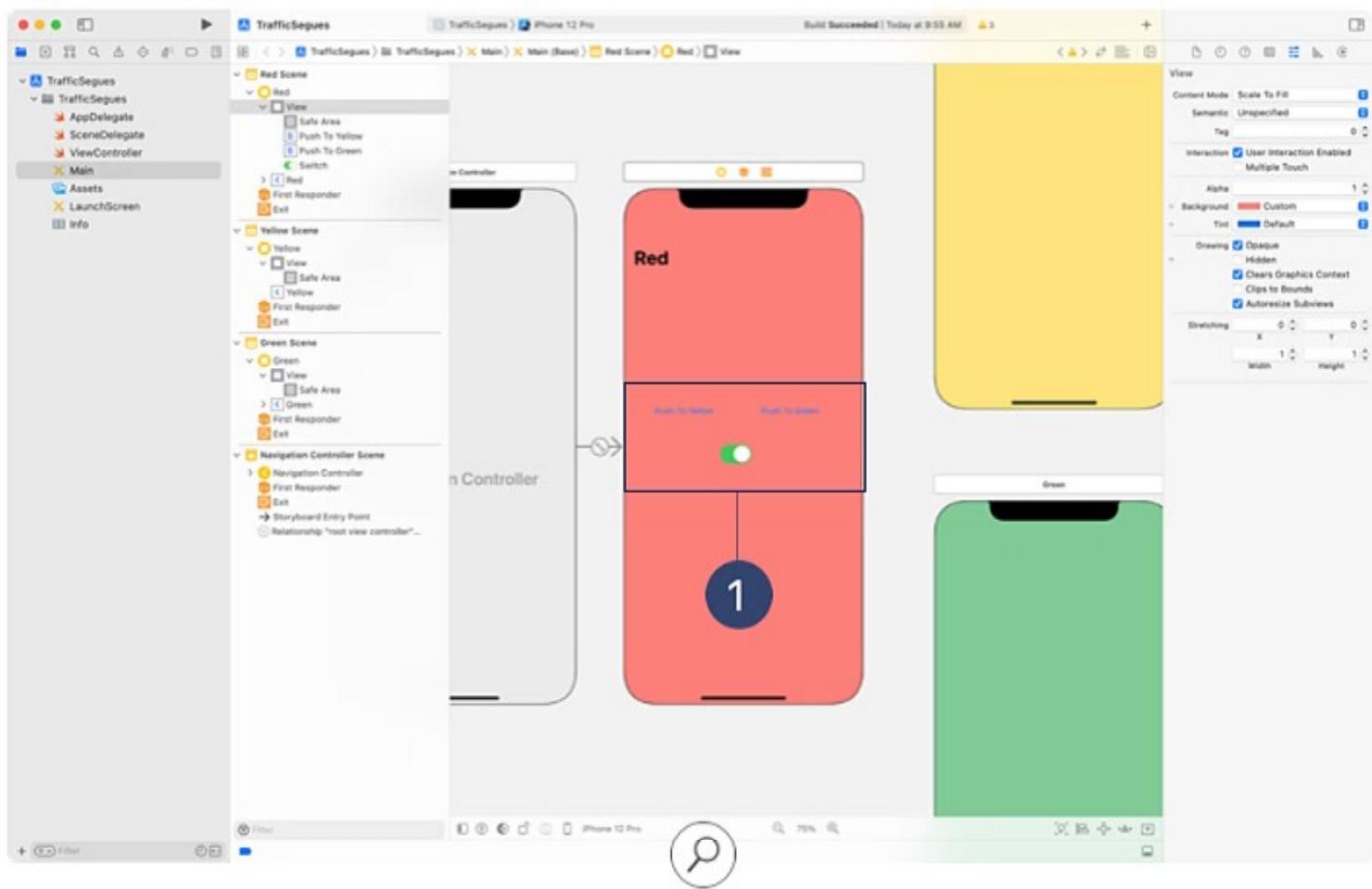
Writing the code for passing data between scenes can be a little tricky at first. But you'll discover that passing data between screens is extremely useful and will allow you to write very flexible code. You'll continue working with passing information with segues in the Quiz project.

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.

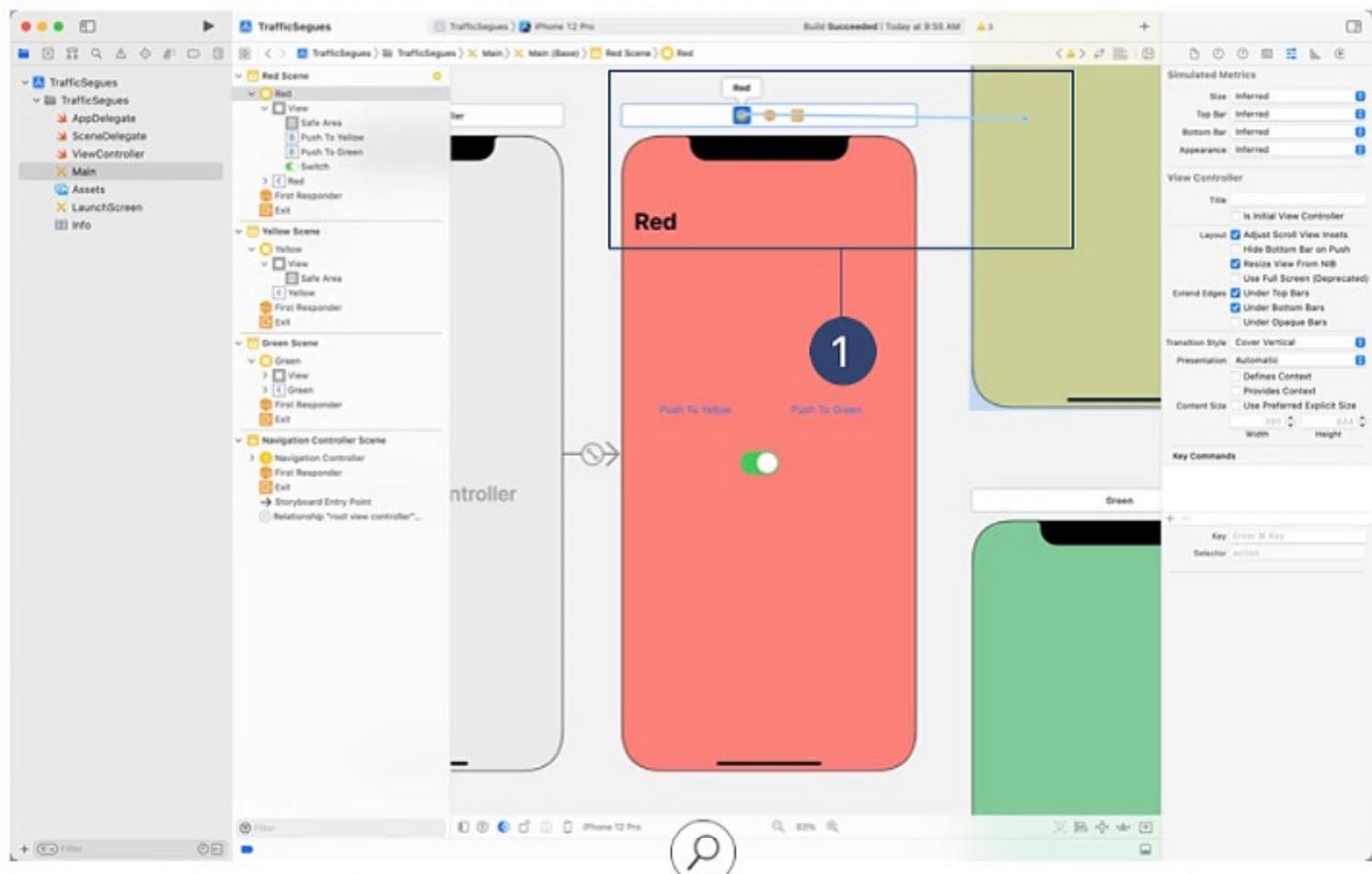
Before you begin, restore your application to a clean slate. Remove all existing segues, controls, and labels from your storyboard. You also need to remove any outlets you've created, as well as the `prepare(for:sender:)` and `unwindToRed(segue:)` methods. Your app should look like this when you're ready:





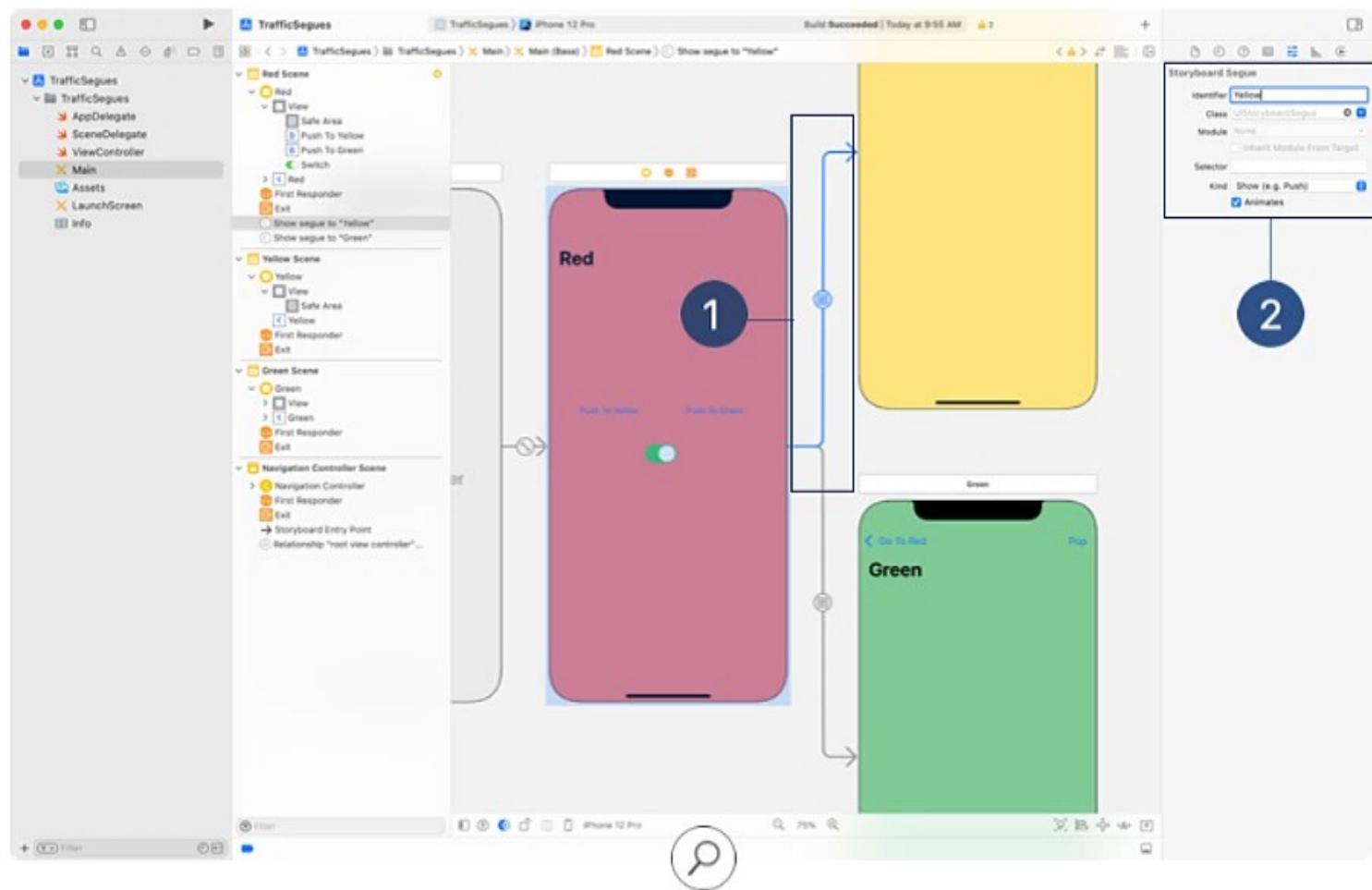
In this example, you'll place two buttons on the red view controller. One button will segue to the yellow screen, and the other will segue to green. The UI will also include a UISwitch that will determine whether or not the segues will be performed. If the switch is enabled, the segue can be performed; 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

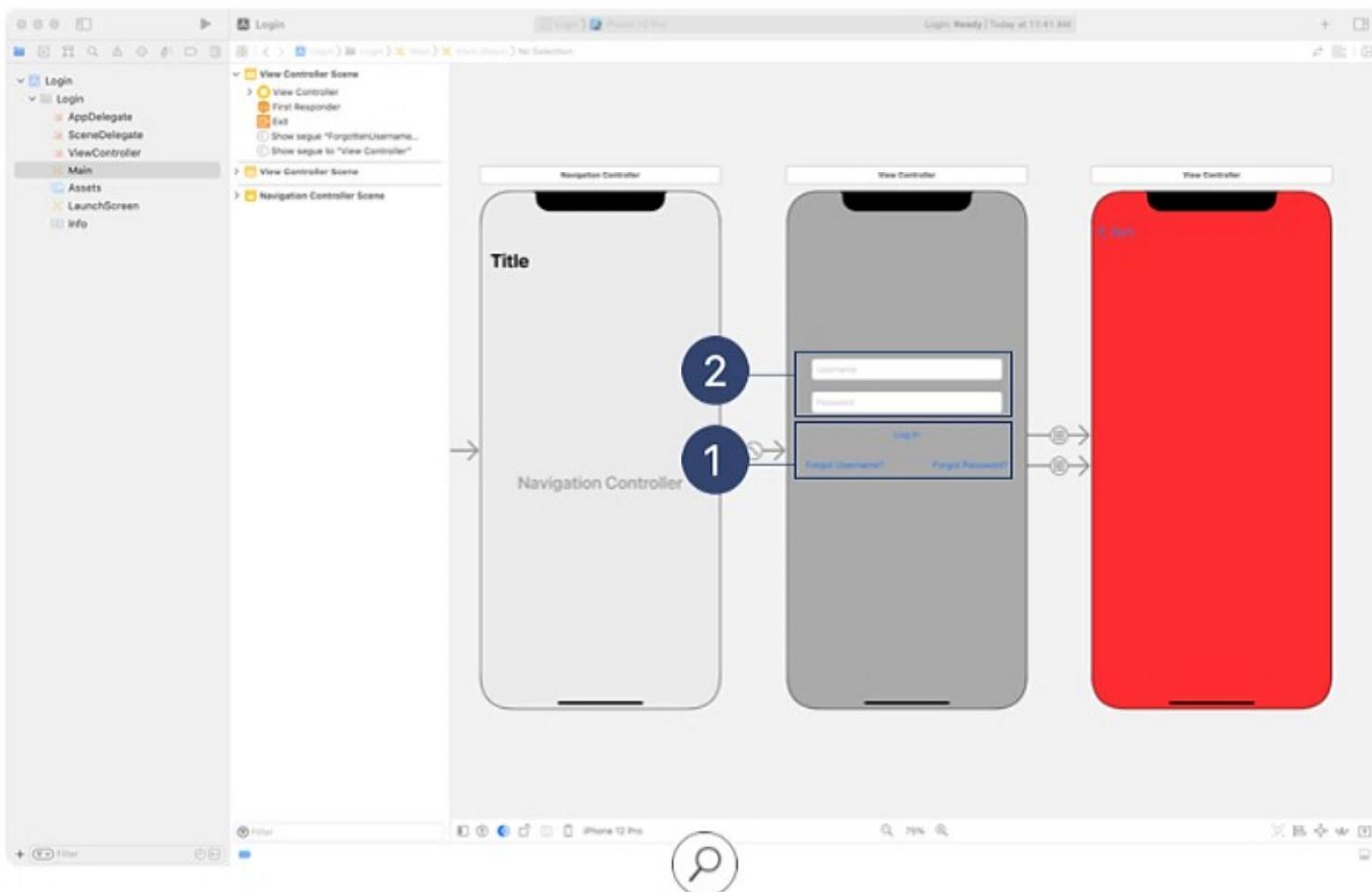
The objective of this lab is to create a login screen that passes a username between view controllers. You'll use view controllers, a navigation controller, and segues to create both the login screen and a simple landing screen that displays in its title either the username or text related to a forgotten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

```
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.

Review Questions

Question 1 of 4

Which segue adapts its presentation style from modal to push if a navigation controller is present?

- A. Unwind
- B. Show
- C.Popover
- D. Replace

[Check Answer](#)



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
-

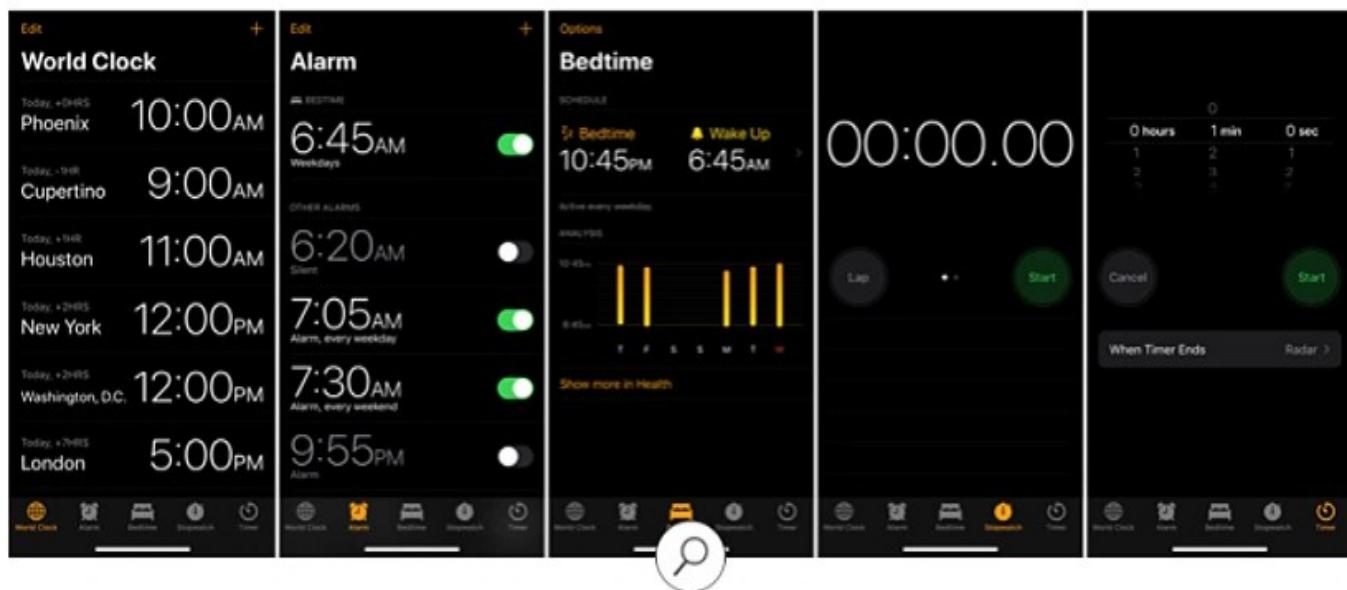
Vocabulary

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

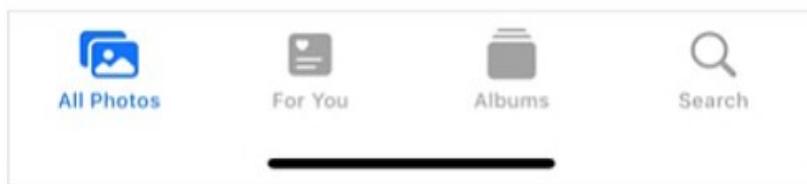
Related Resources

- 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 of 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.

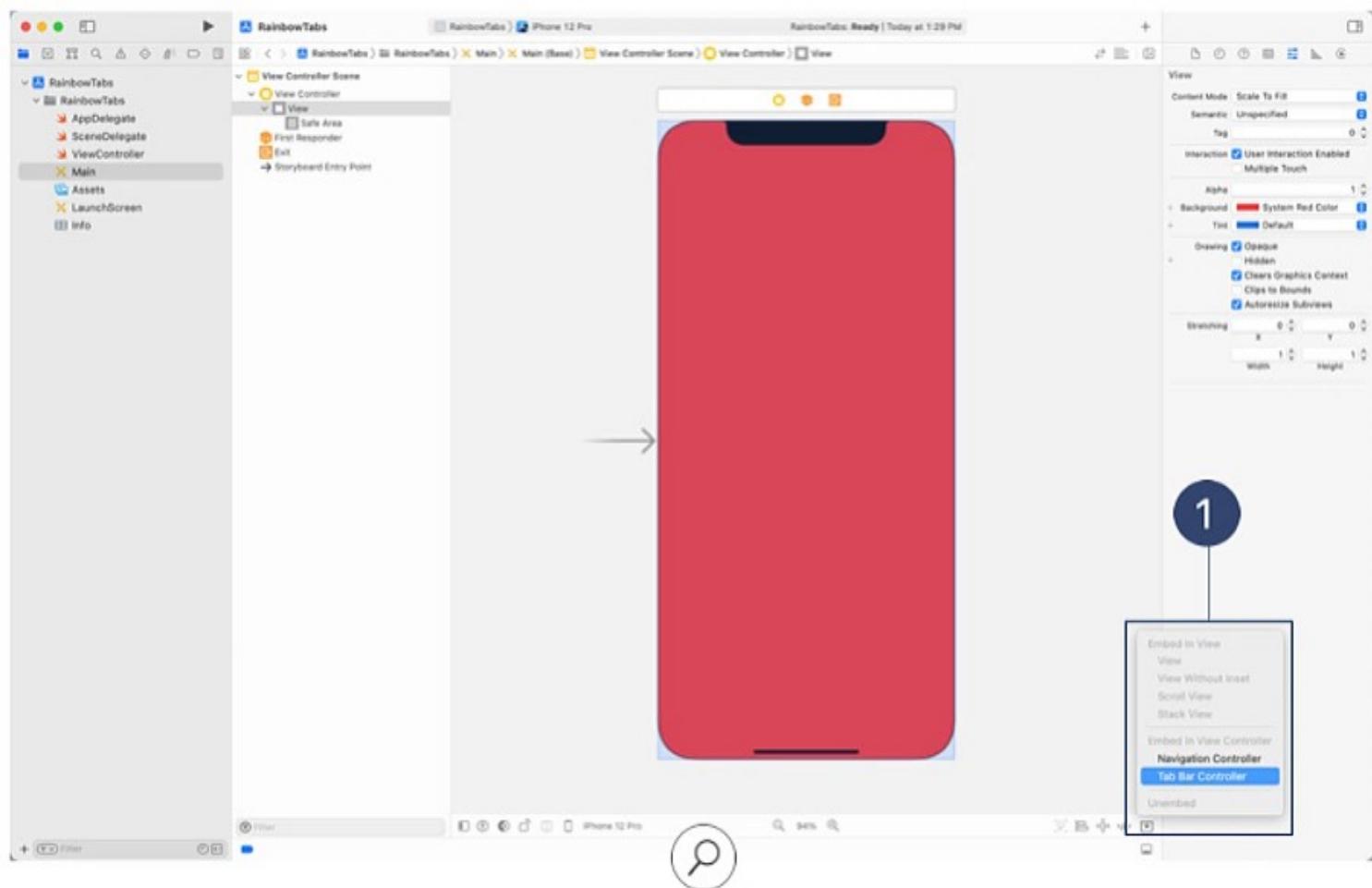


Add A Tab Bar Controller

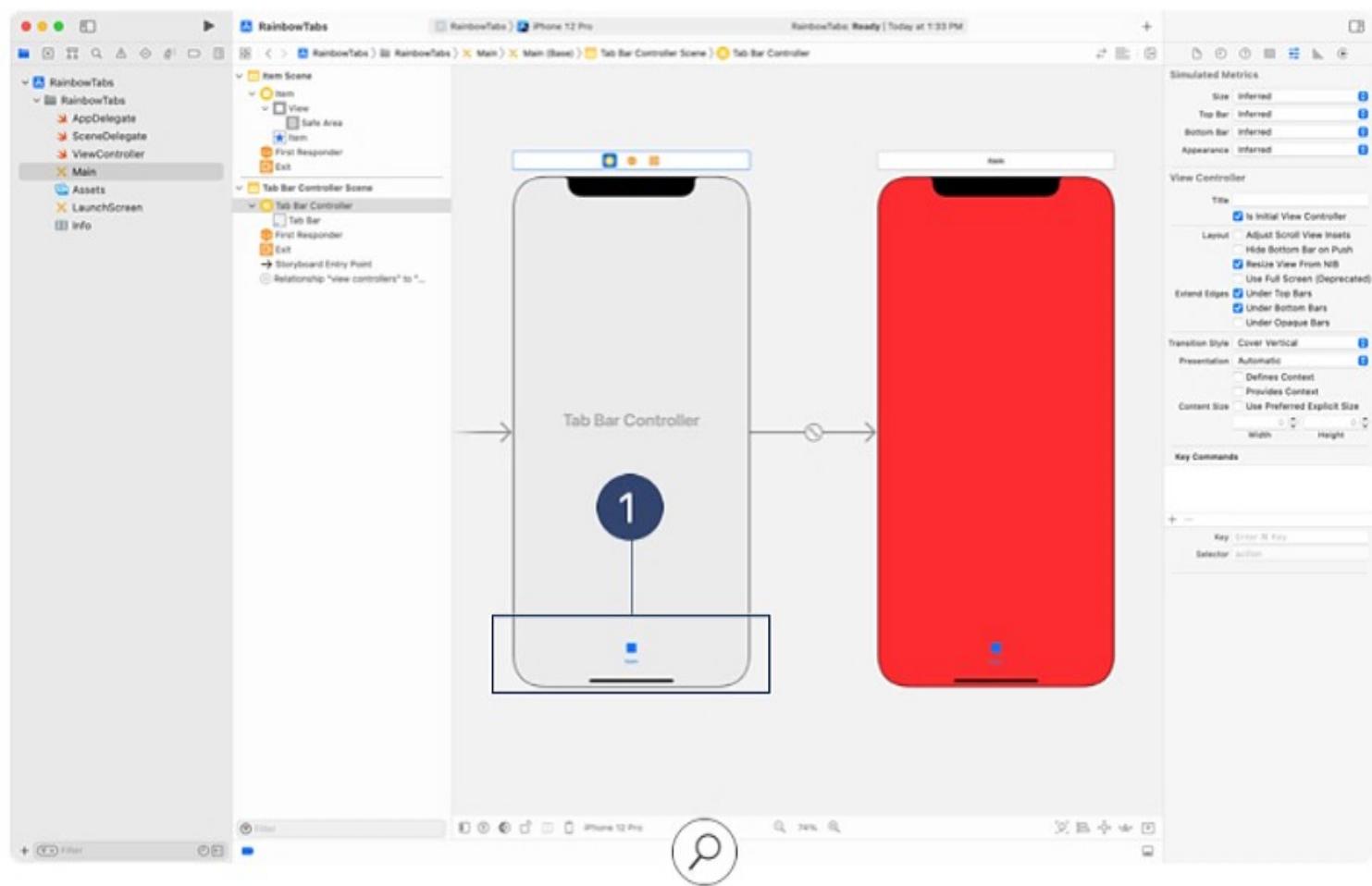
To practice building a tab bar interface, you'll create a simple app that navigates between several root view controllers. (This could be a good template for future projects that also use a tab bar controller.)

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**.



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

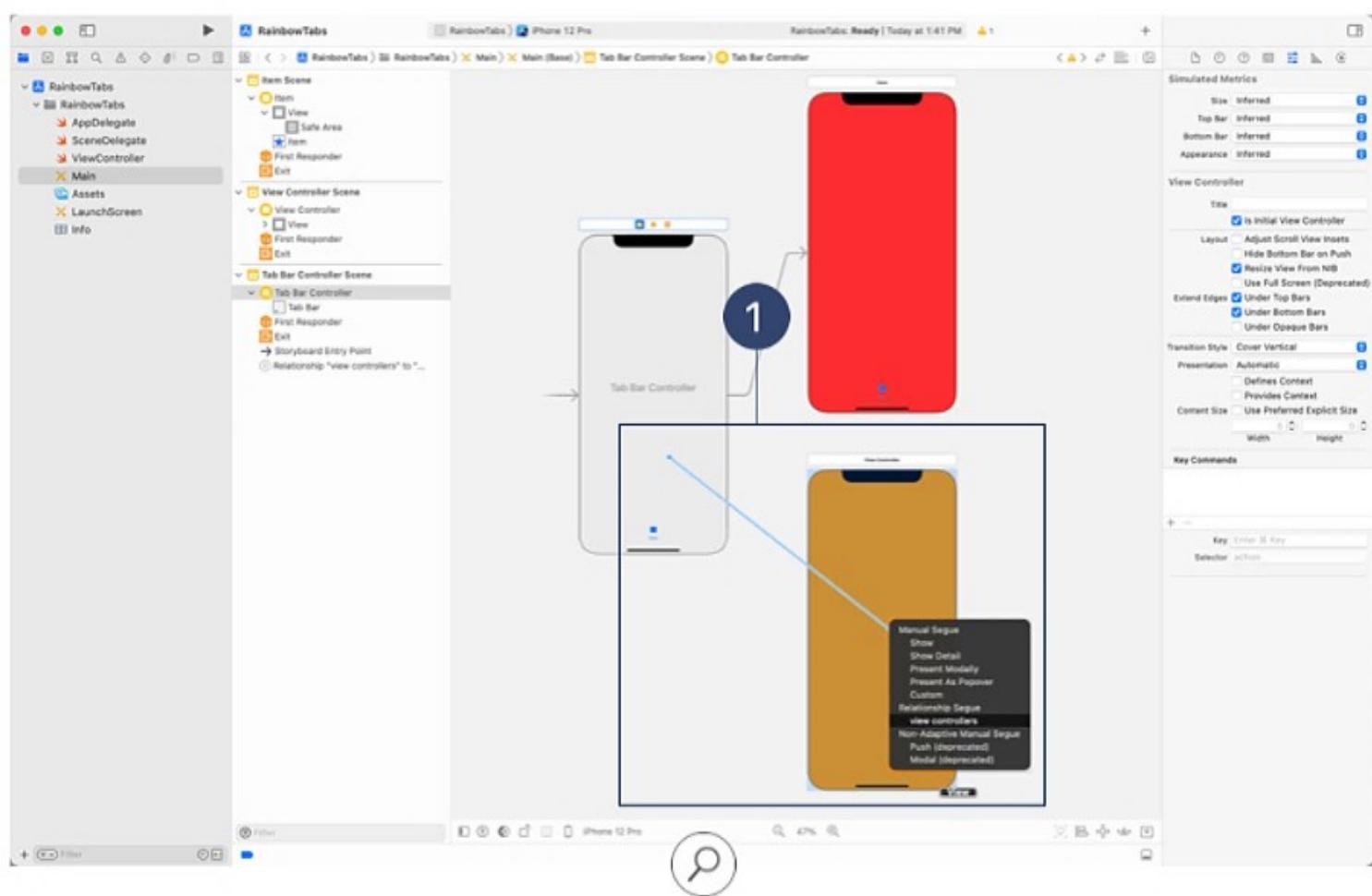


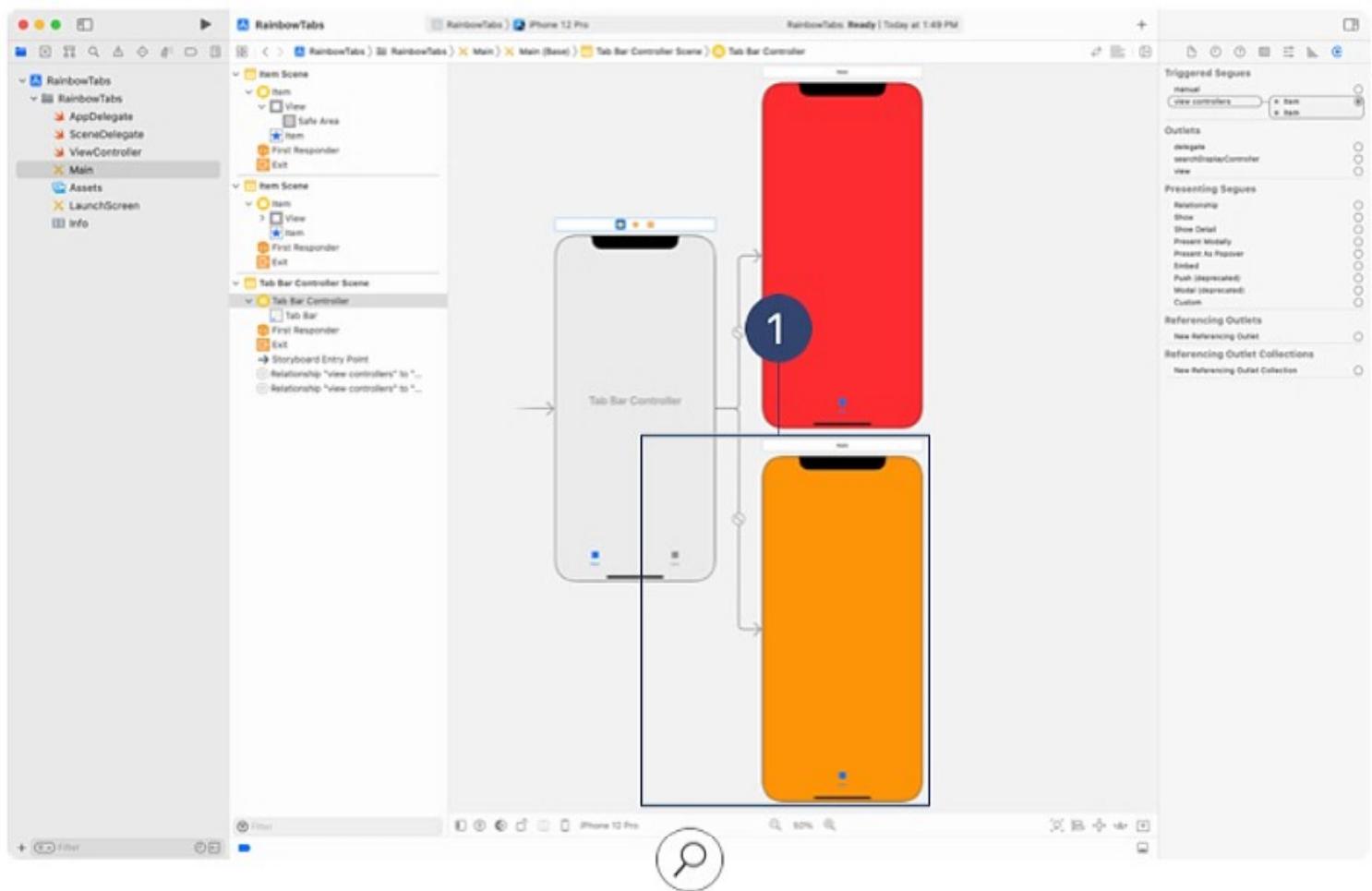
The tab bar controller maintains a list of tabs through its `viewControllers` property, an array of the root view controllers displayed by the tab bar interface.

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 the 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



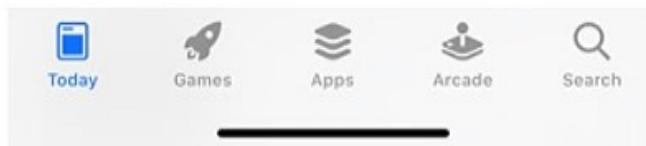


You should now see a second tab bar item on the tab bar controller. ①

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

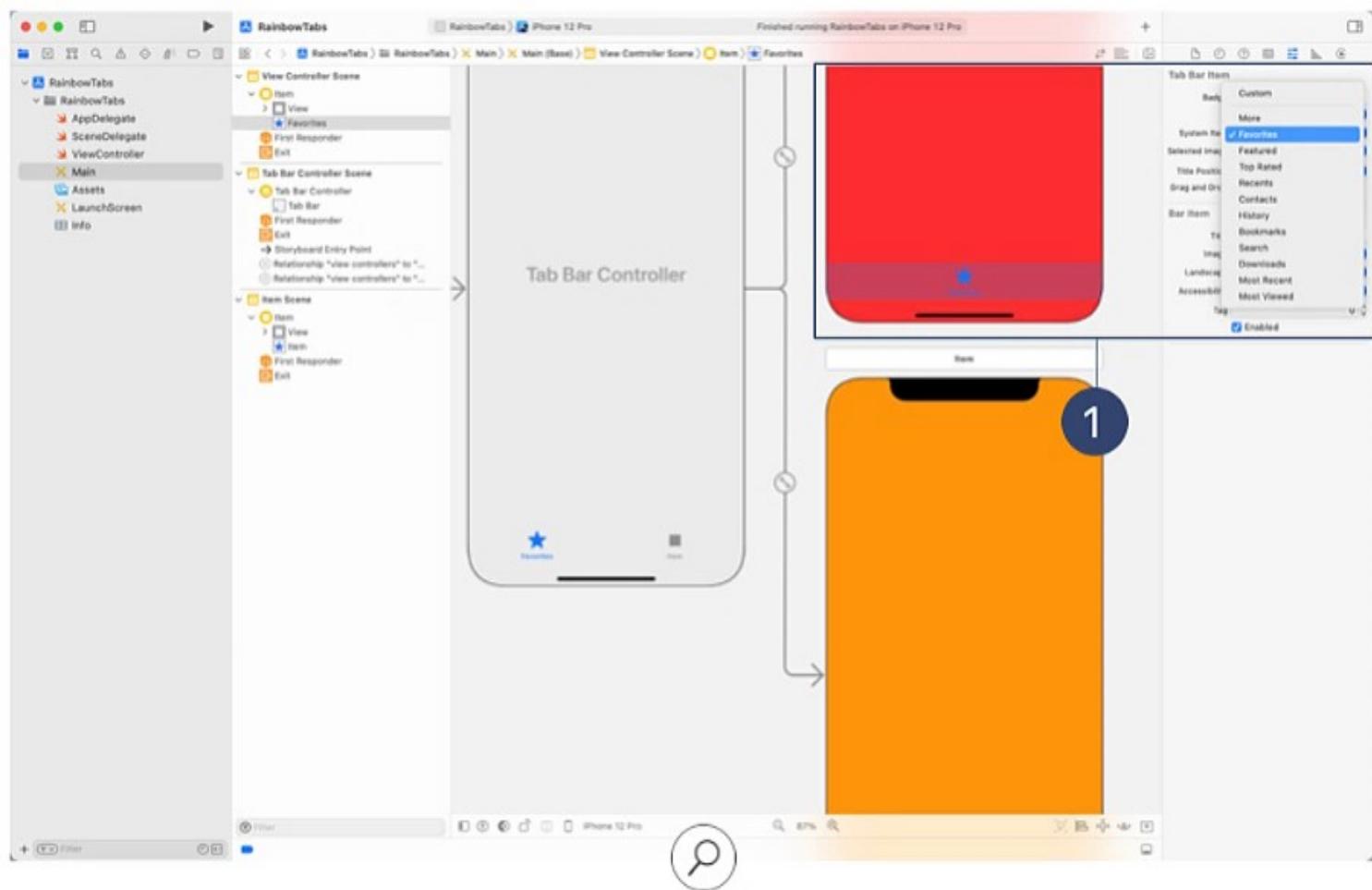
A tab bar item consists of two visual elements: an image and a label. The iOS SDK provides several iOS-style icons paired with system-defined text, referred to as system items. One example is the search icon that you see in the App Store tab bar.



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	1 2 3

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.

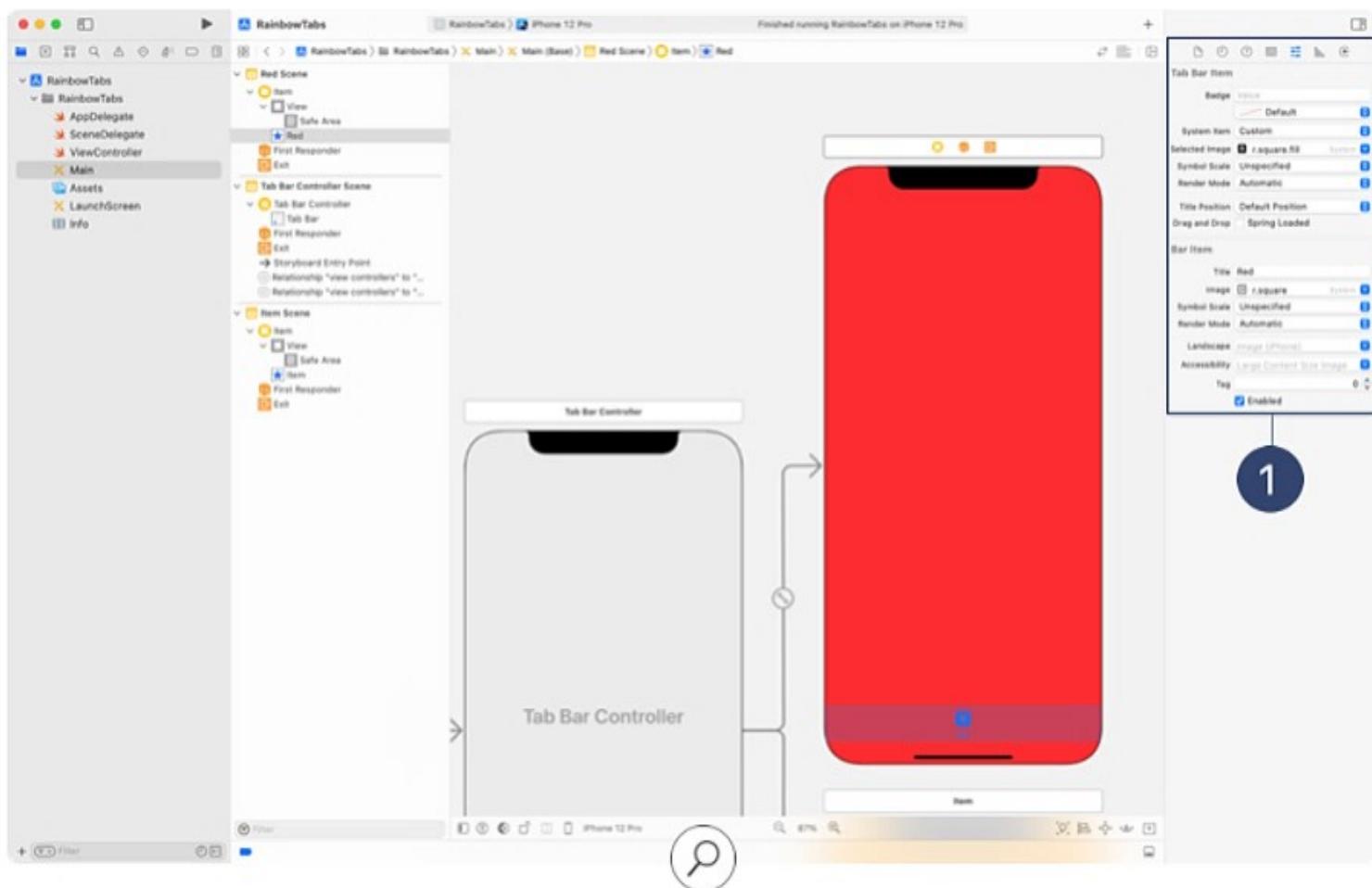


Customize Tab Bar Items

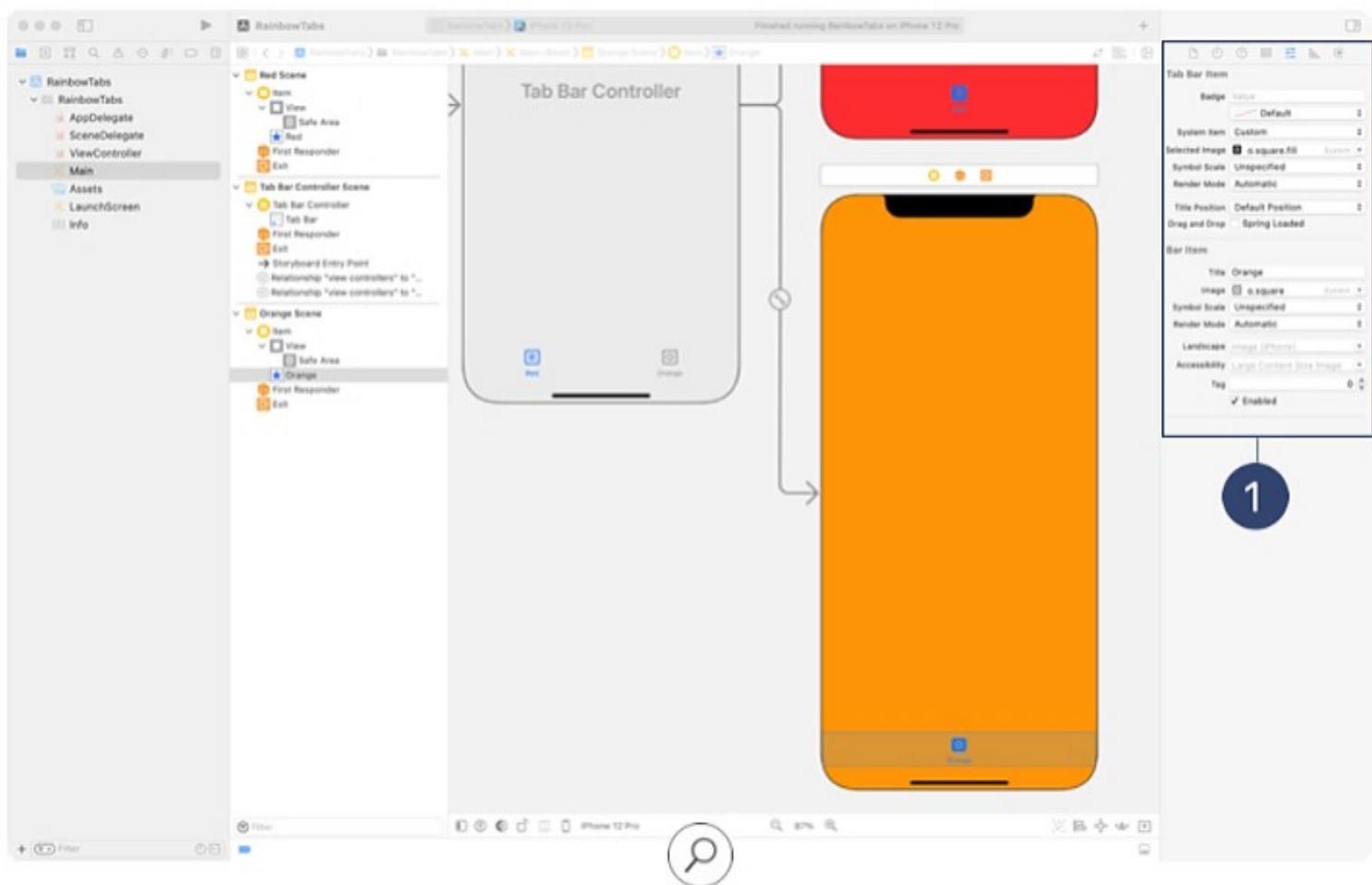
But maybe the system items don't make sense for your app. You can use the Attributes inspector to customize an item's label and its image, for both unselected and selected states. When providing icons of any kind for an iOS app, you should review the [Human Interface Guidelines for Custom Icons](#) to 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.

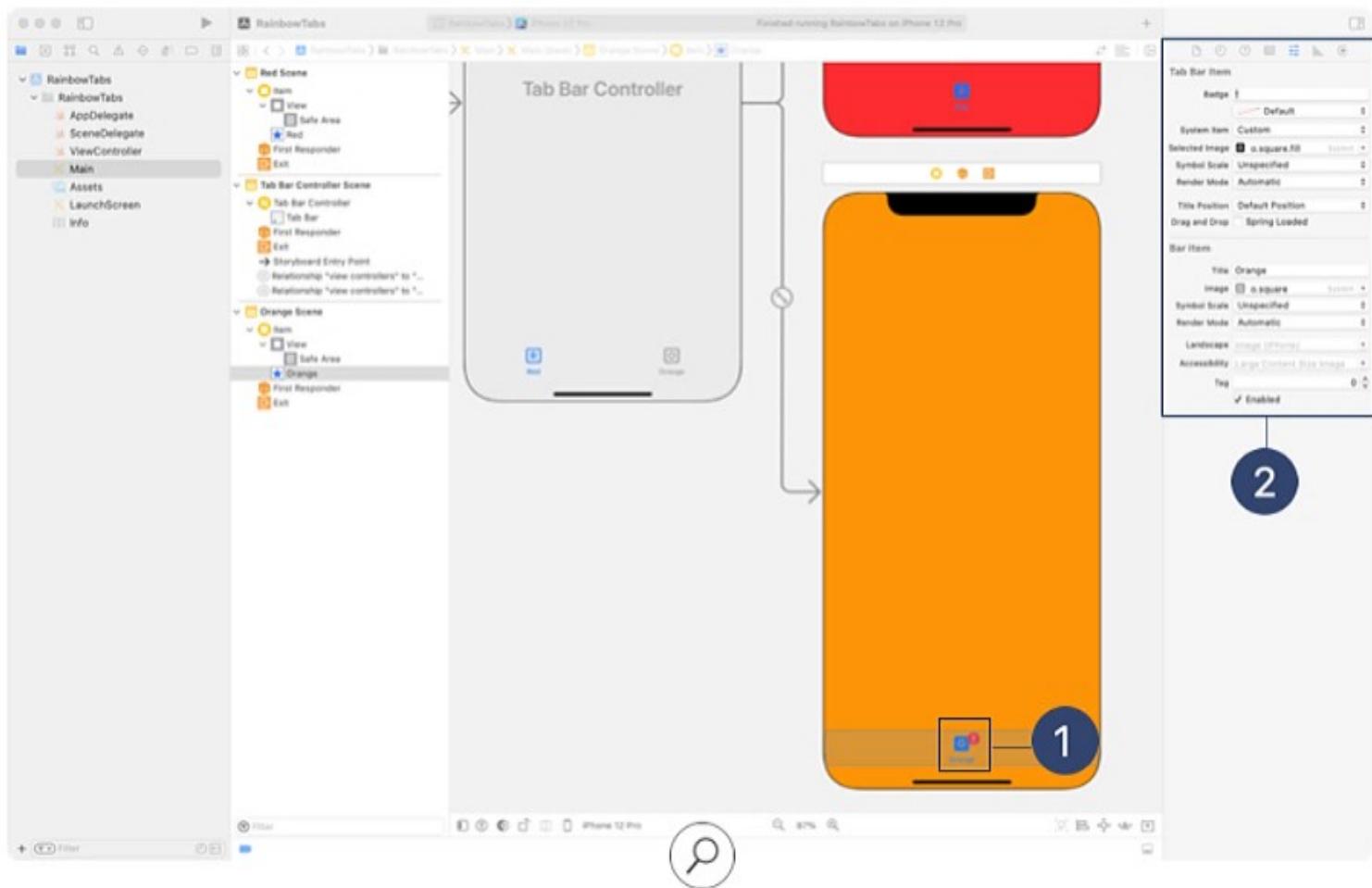
With the red view's Tab Bar Item still selected, type "Red" in the Title field and type "r.square" into the Image field. From the menu, choose the matching symbol. The image is provided through SF Symbols as denoted by the "System" label next to the arrow.^① You can also customize the Selected Image 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

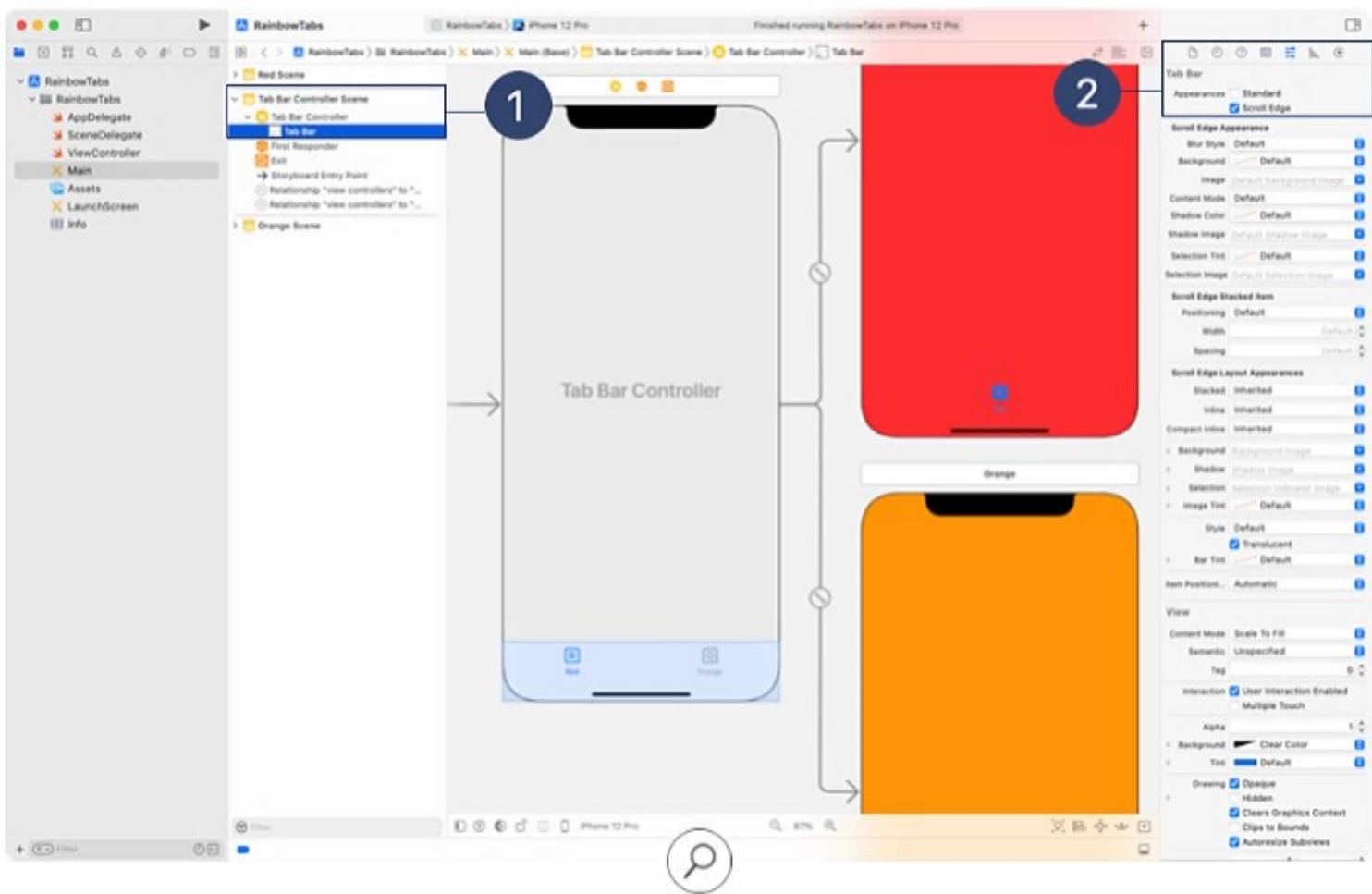


What if you want your tab bar item to indicate that new information is available for that view or mode? You can add a small red marker with white text, known as a badge, to the tab bar item. ① Use the Badge attribute to enter information. ②



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  
}
```

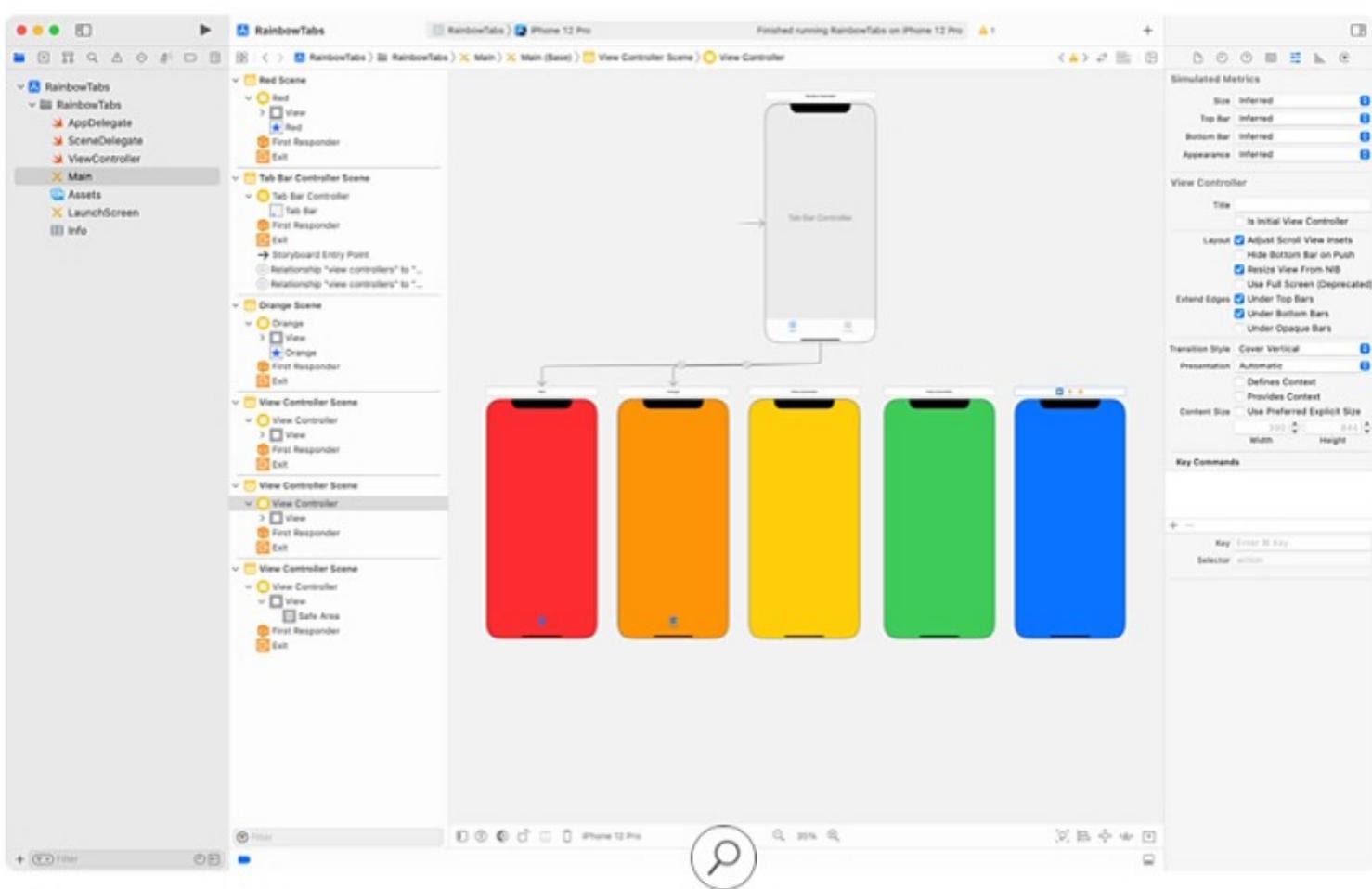
Build and run the app again, when you navigate to the Orange tab and you will see the badge on the red tab disappear.



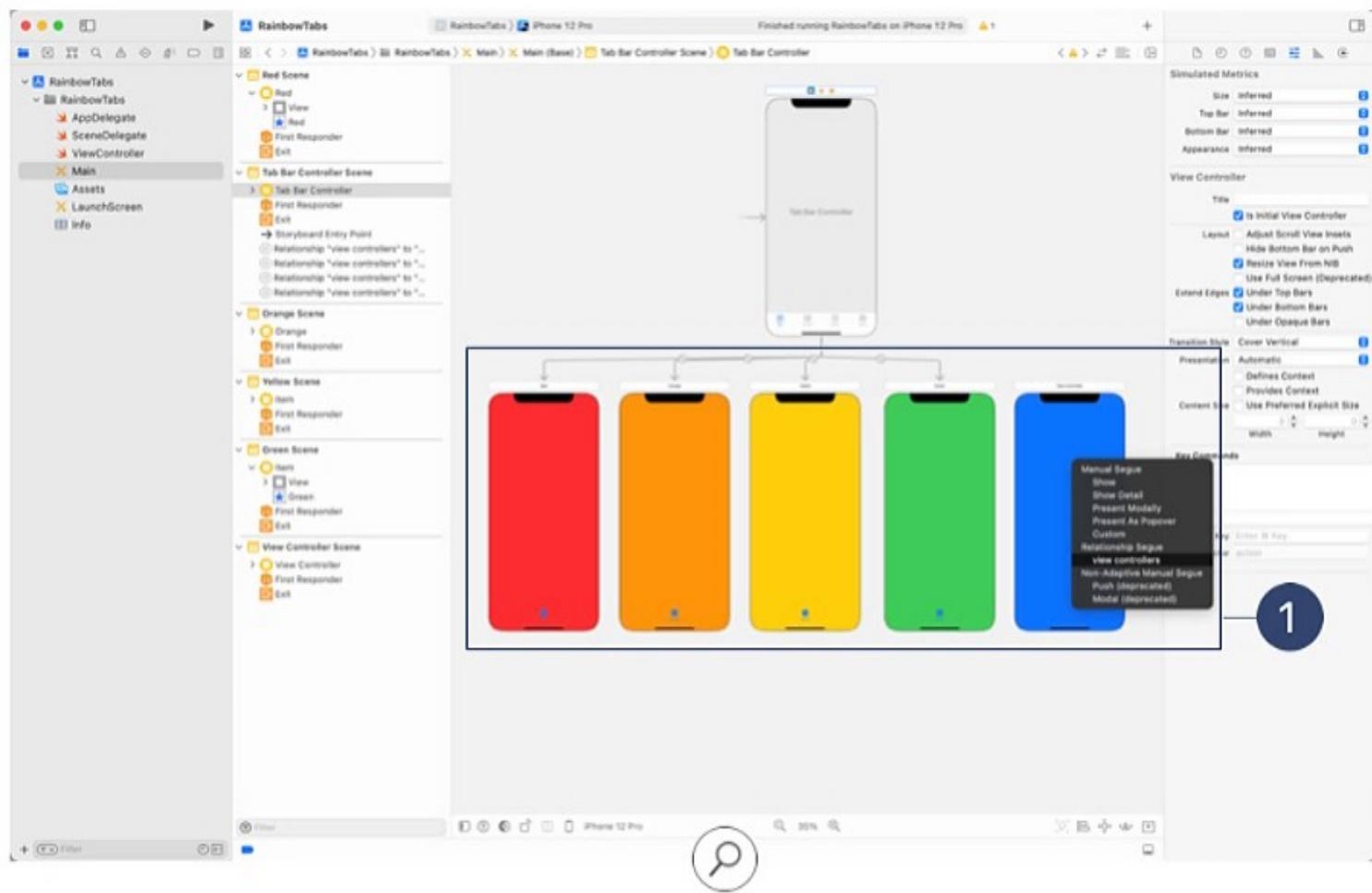
Even More Tab Items

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.

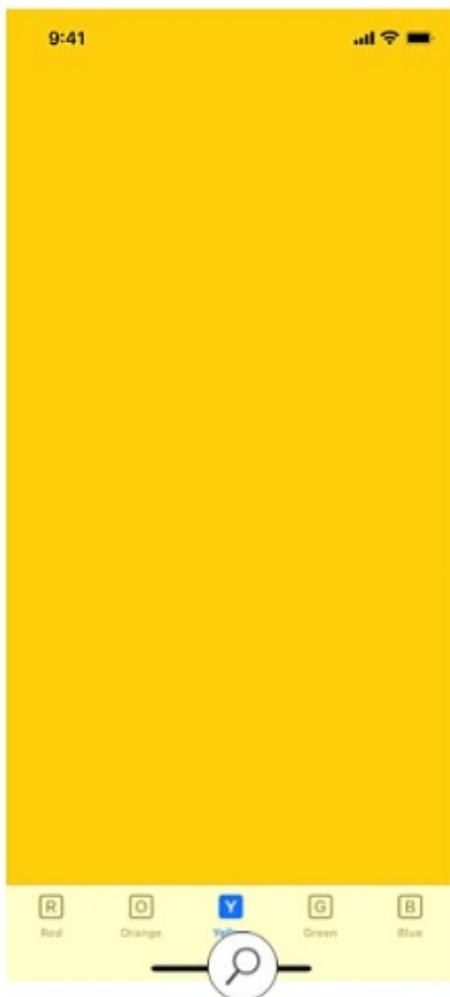


Add each of these view controllers to the `viewControllers` property of the tab bar controller. ①



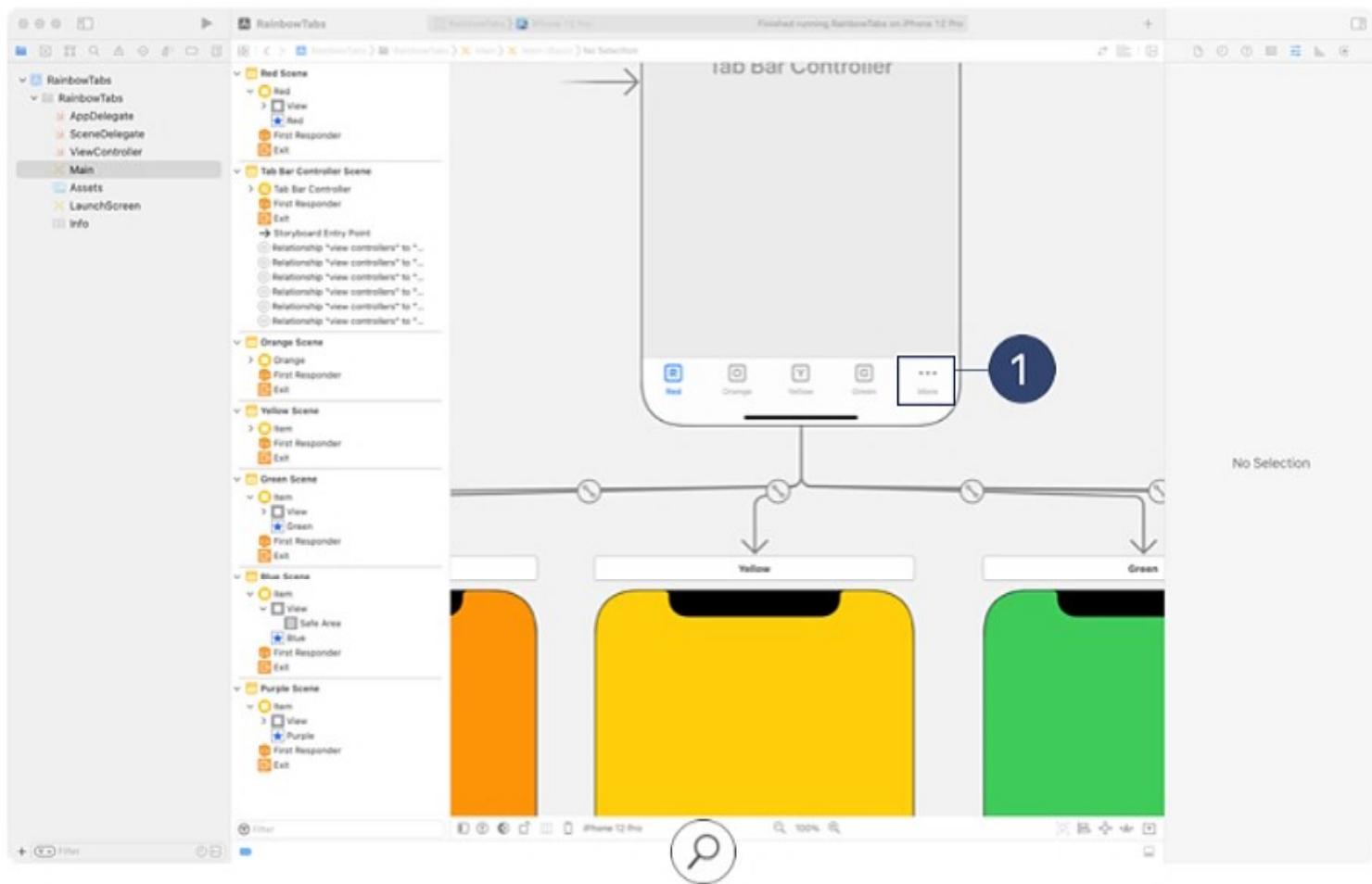
You can update the new tab bar items with images from the assets catalog or from SF Symbols.

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?

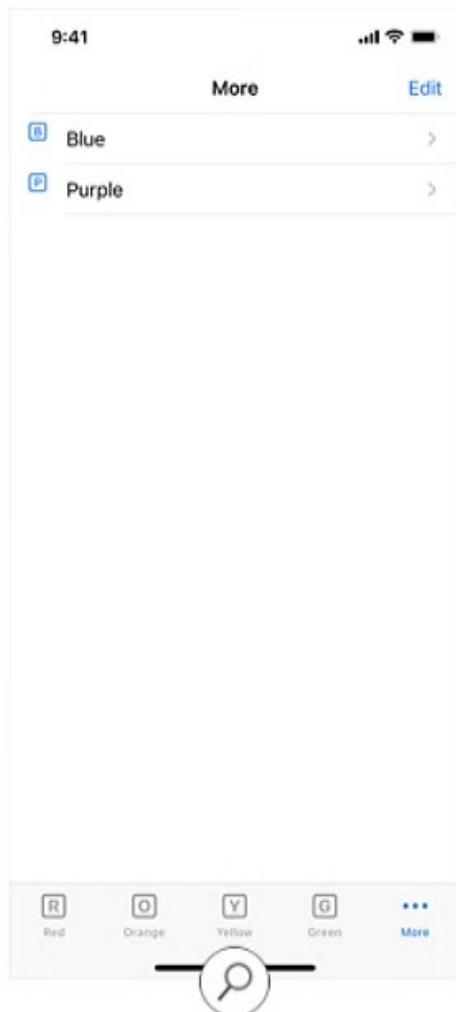


Drag another view controller onto the canvas, update its background color to purple, and add it to the tab bar controller's array of view controllers. You'll notice, in your tab bar controller scene, the fifth tab is replaced with a More tab item.¹

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.

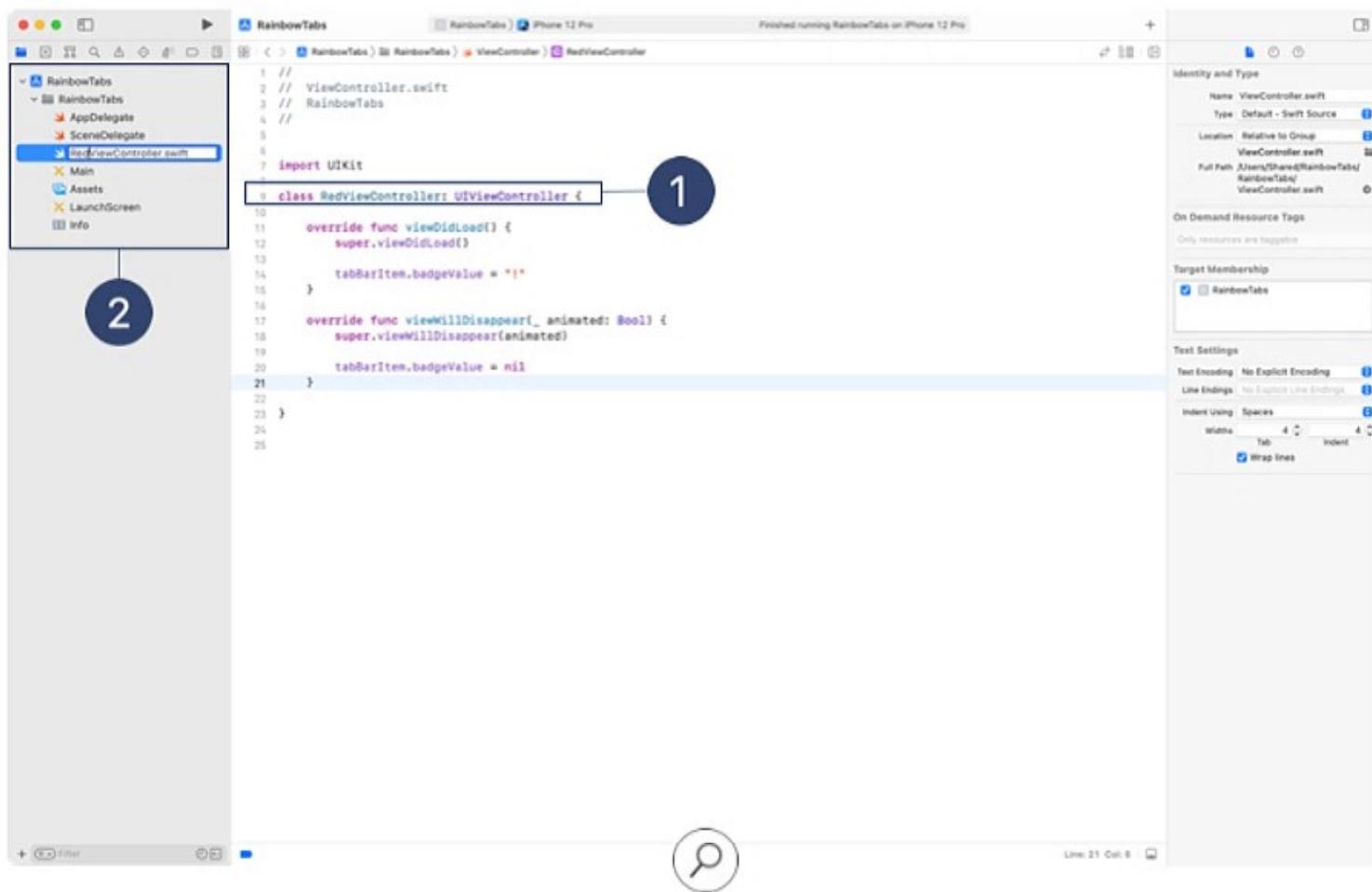


The More view controller is unusual. It can't be customized or selected. It doesn't appear among the view controllers managed by the tab bar controller. It appears when needed and is otherwise separate from the rest of your content.

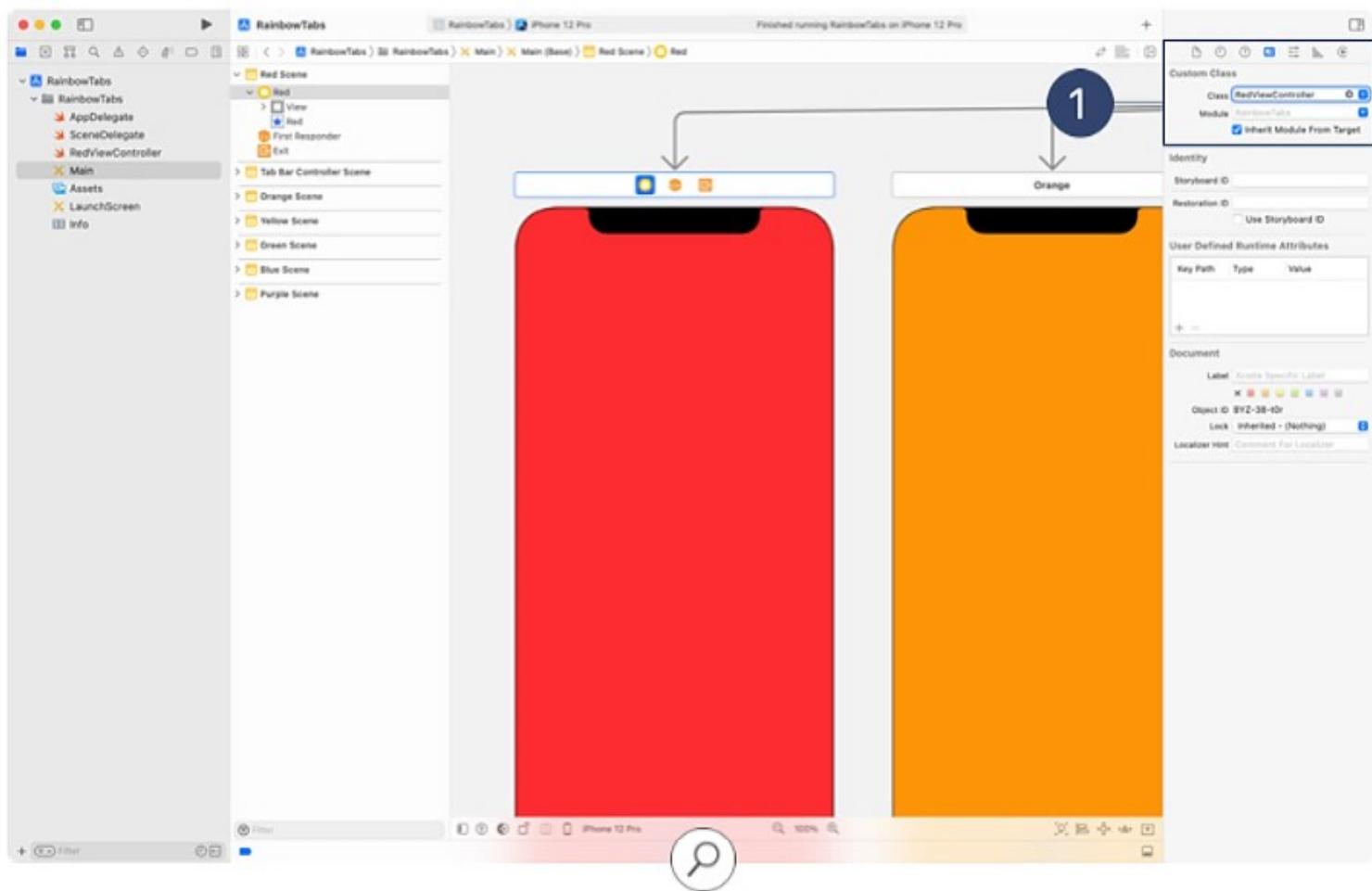
The More view controller can be quite useful for displaying additional tab items, but also consider that a More tab requires more time and effort from the user. A much better practice is to plan your app carefully so that you include only essential tabs—the minimum number necessary for your app and its 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

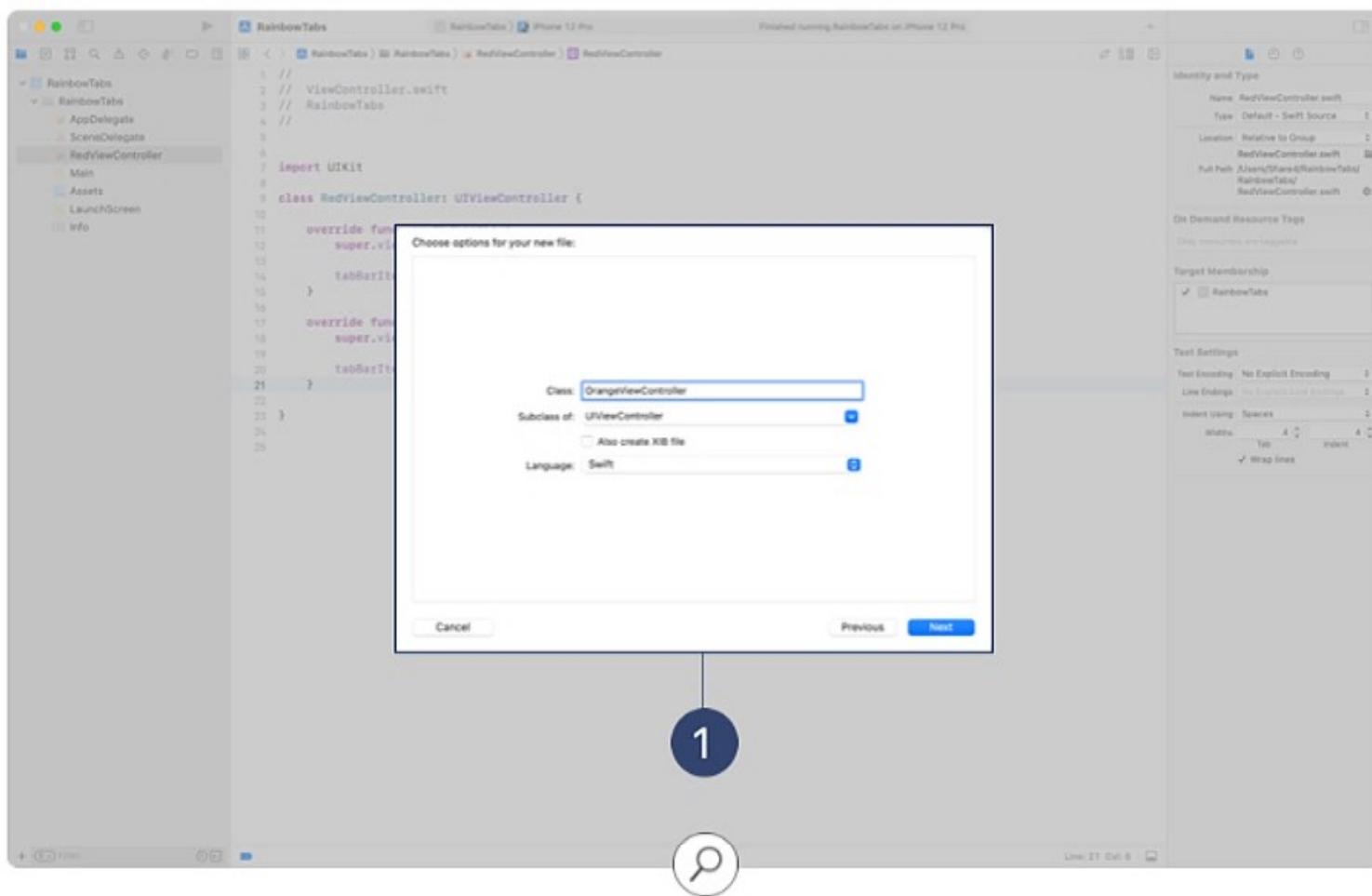
Currently, only the controller with the red view can be customized with code, because it's the only controller that uses a `UIViewController` subclass. That subclass is called `ViewController`, and it was created as part of the iOS App template. At some point, you're going to want to add new functionality 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 SF Symbols, 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.

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.



MORE



CONTACTS



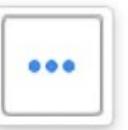
SEARCH



MOST RECENT



FAVORITES



Check Answer



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](#)

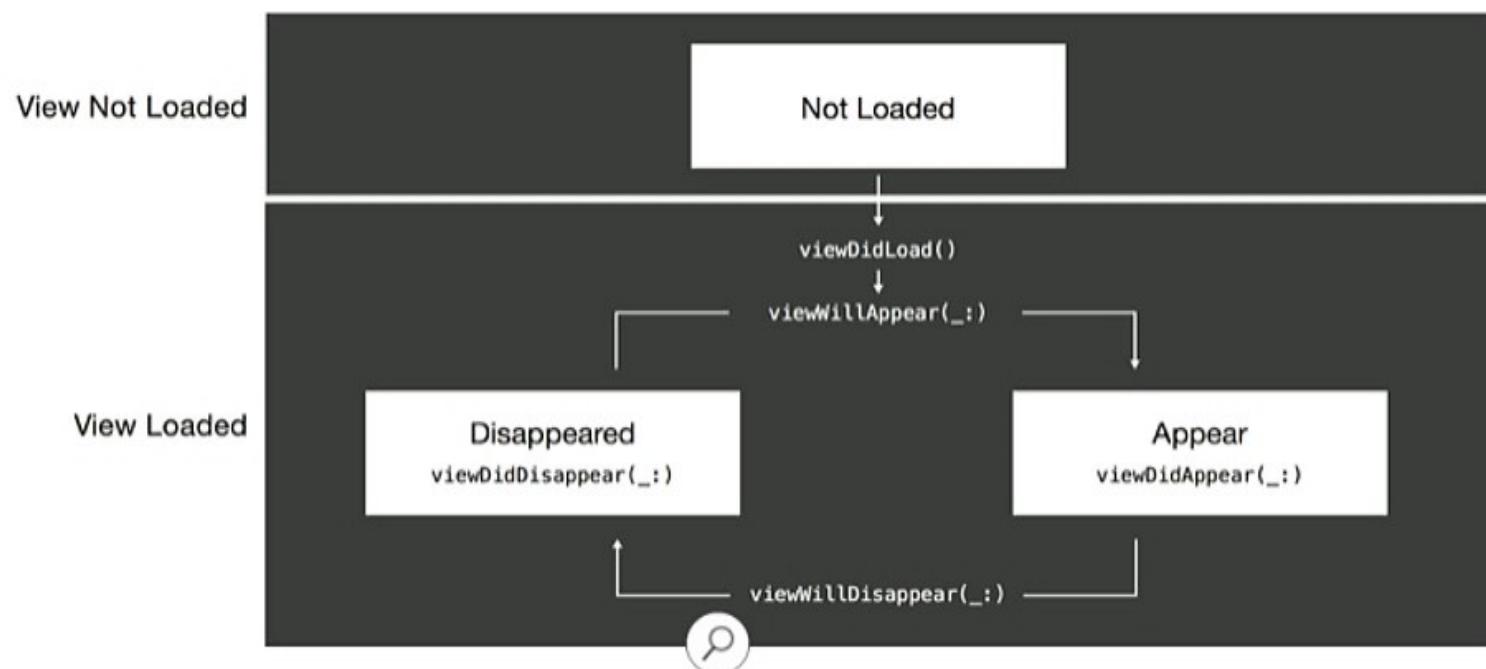
When you've finished setting up views and view hierarchy, you'll need to move on to writing the logic behind your app. Your view controller classes are responsible for displaying your user's data and handling user interactions. In the Light project, you used a `UIViewController` subclass to manage the 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.

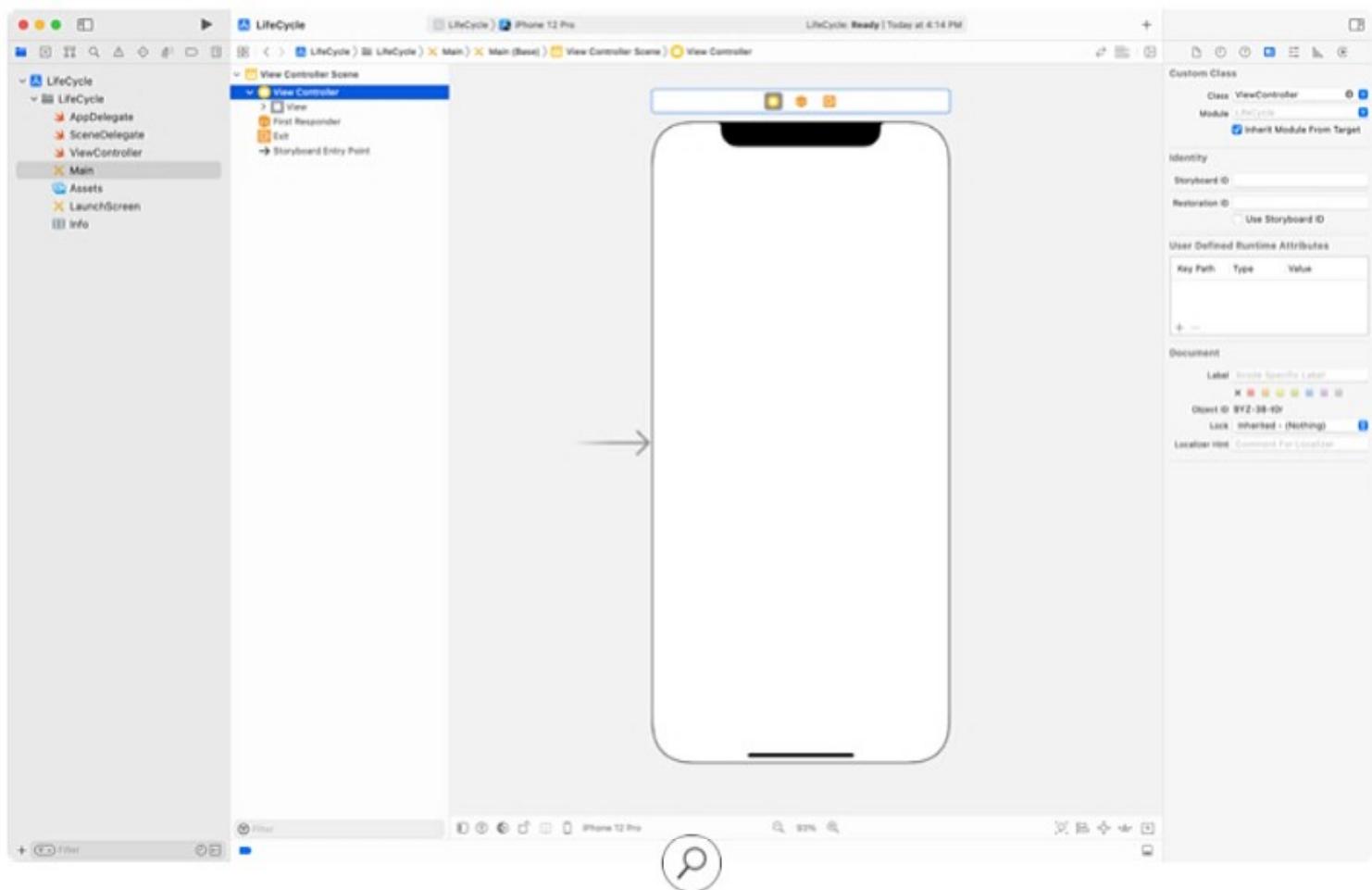
In Xcode, create a new project using the iOS App template. Name the project “Lifecycle.”

View Did Load

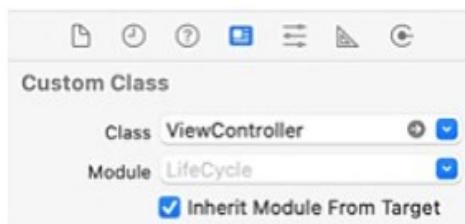
After you’ve instantiated a view controller, whether from a storyboard or programmatically, the view controller will load the view into memory. This process creates the views that the controller will manage.

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`.

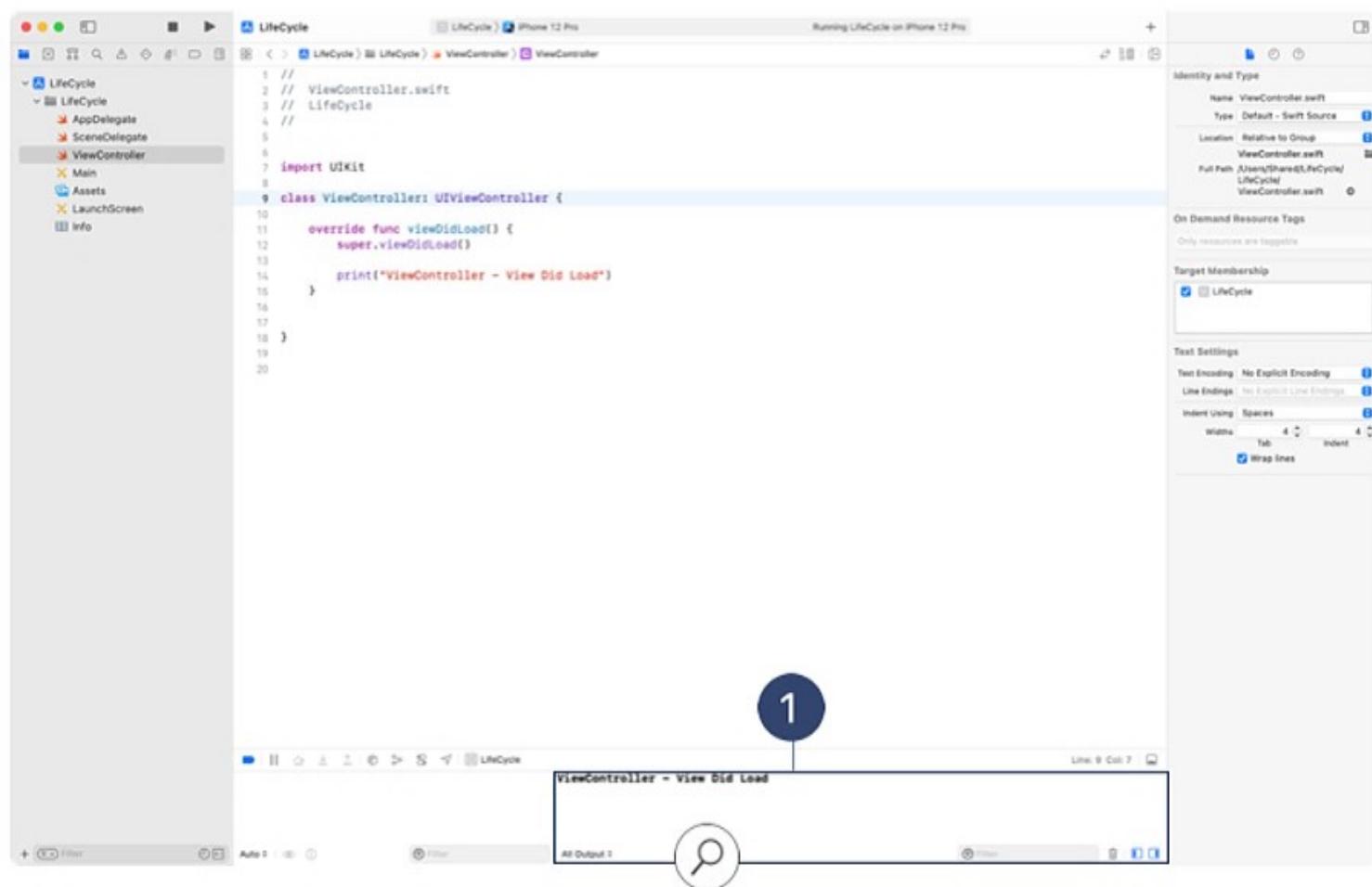


Add the following to the end of the `viewDidLoad()` function:

```
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 viewDidAppear(_ animated: Bool) {  
    super.viewDidAppear(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(_:)`.

The screenshot shows the Xcode interface with the following details:

- Project Navigator:** Shows a project named "Lifecycle" with files: AppDelegate.swift, SceneDelegate.swift, ViewController.swift, Main.storyboard, LaunchScreen.storyboard, and Info.plist.
- Editor:** Displays the content of ViewController.swift. The code defines a ViewController class that overrides viewDidLoad(), viewWillAppear(_ animated: Bool), and viewDidAppear(_ animated: Bool) methods. Each method prints a log message: "ViewController - View Did Load", "ViewController - View Will Appear", and "ViewController - View Did Appear".
- Utilities Navigator:** Shows the file's identity and type: "ViewController.swift" (Default - Swift Source). It also lists target membership: "Lifecycle".
- Output Window:** Shows the console output with three lines of text: "ViewController - View Did Load", "ViewController - View Will Appear", and "ViewController - View Did Appear". A circled '1' is placed above the first line of output.

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.

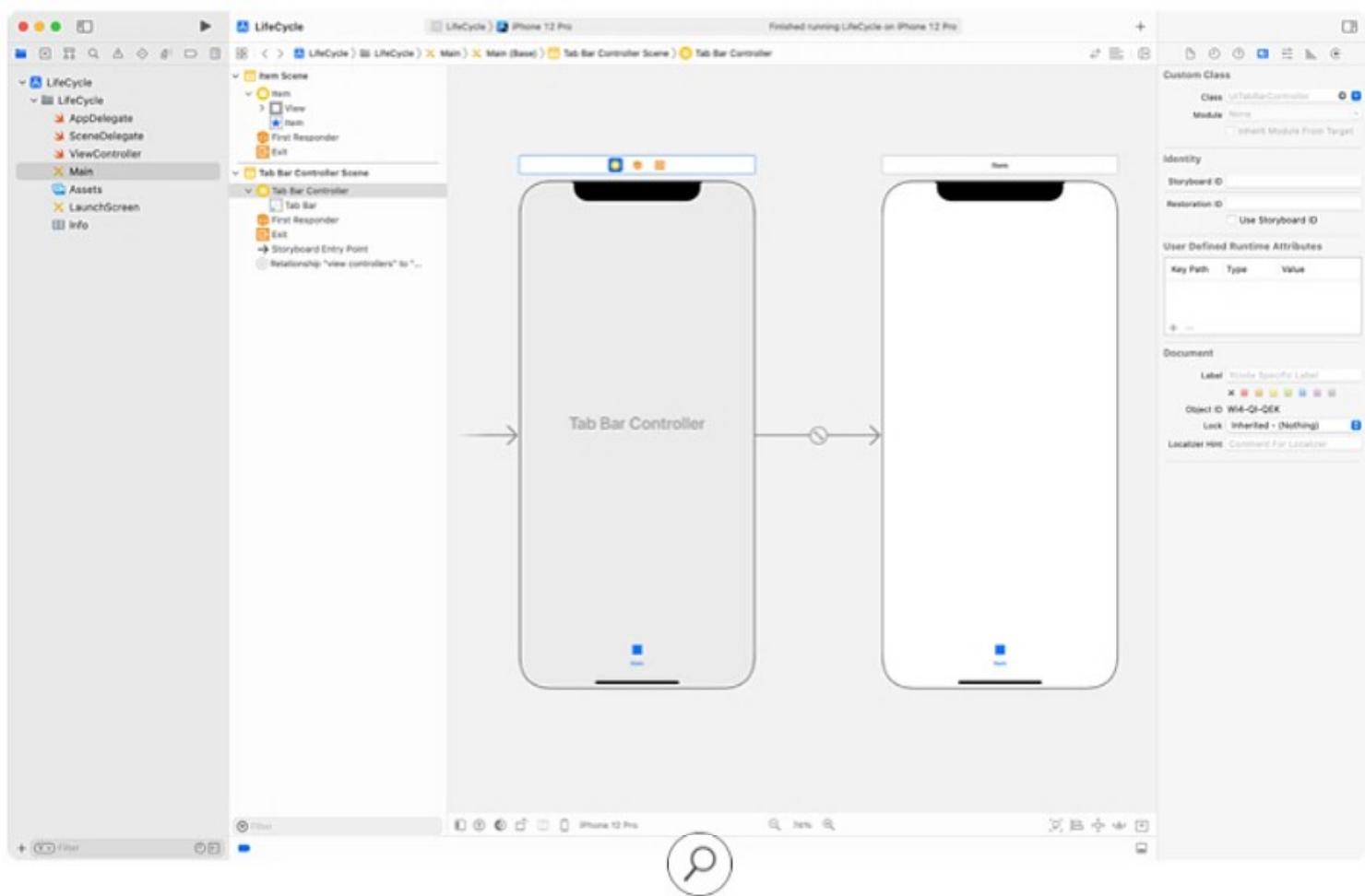
View Will Disappear and View Did Disappear

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.

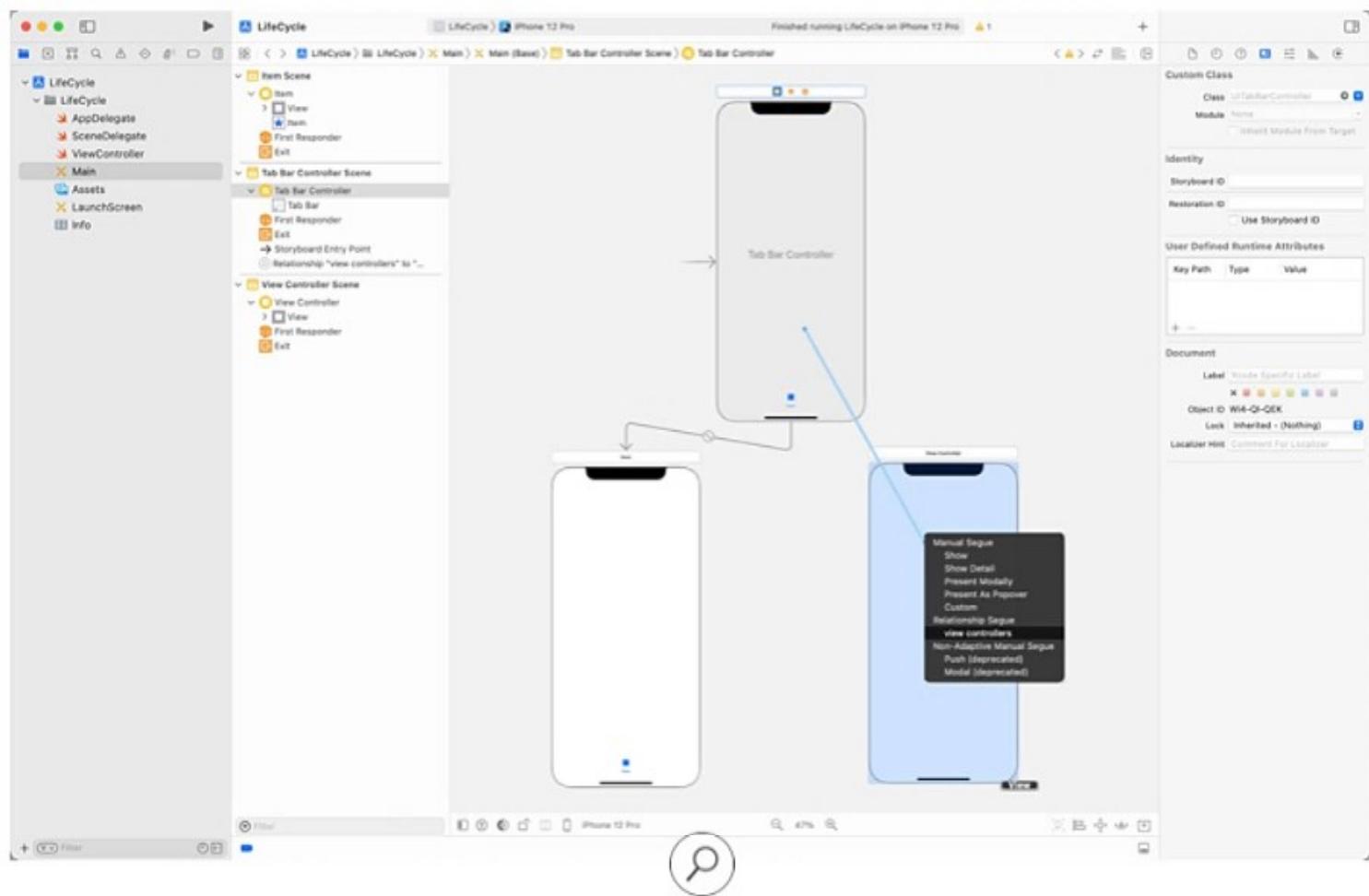
The last method in the life cycle is `viewDidDisappear(_ :)`, which is called after the view disappears from the screen—typically after the user has navigated to a new view. If this method executes, it is certain the view has disappeared. As such, this method gives you an opportunity to stop services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

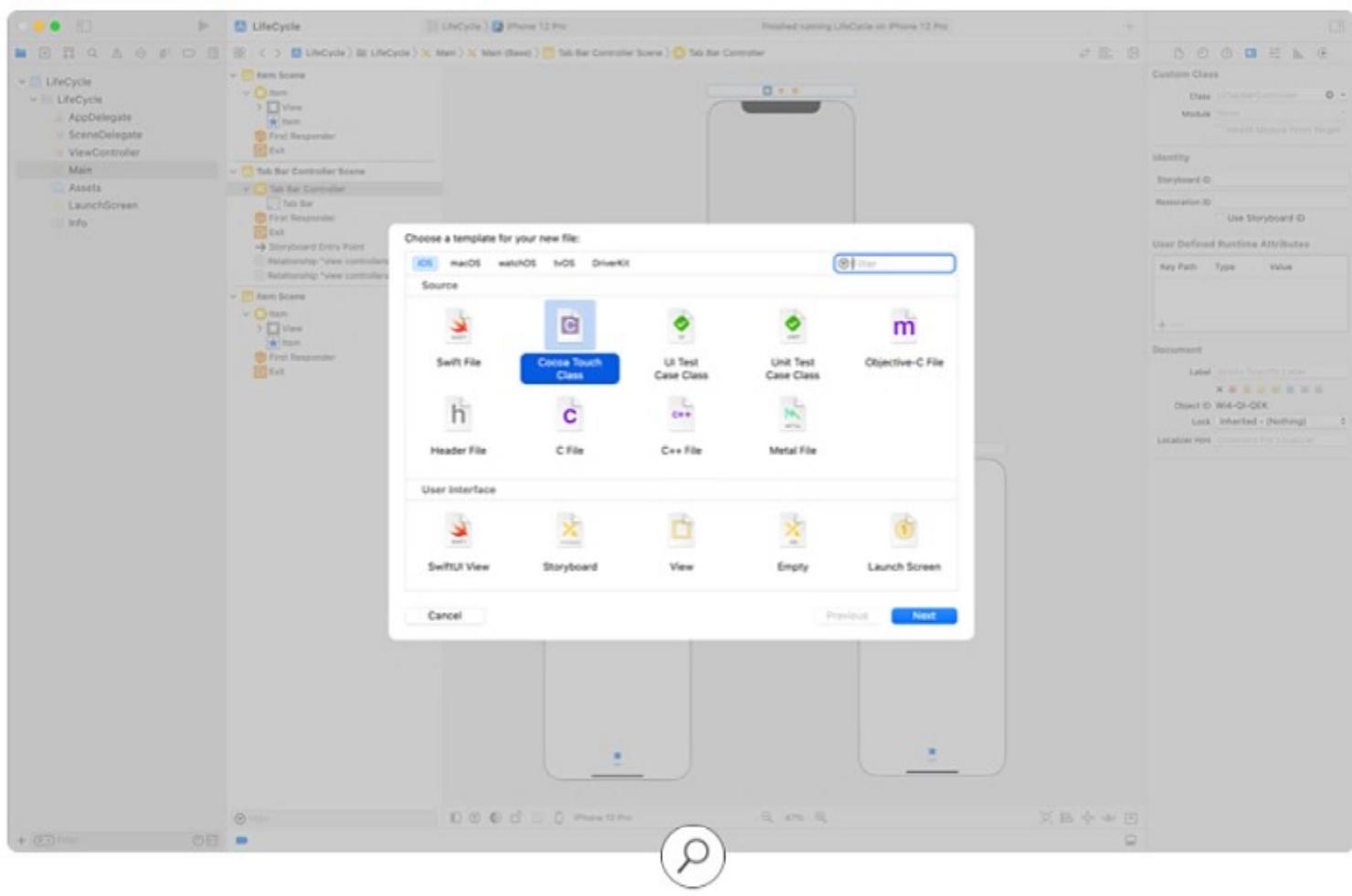
For your view to disappear, you’ll need to add a second view controller. Otherwise, there’d be no way to navigate *from* the current view. But first, you’ll add a tab bar controller to handle navigation.



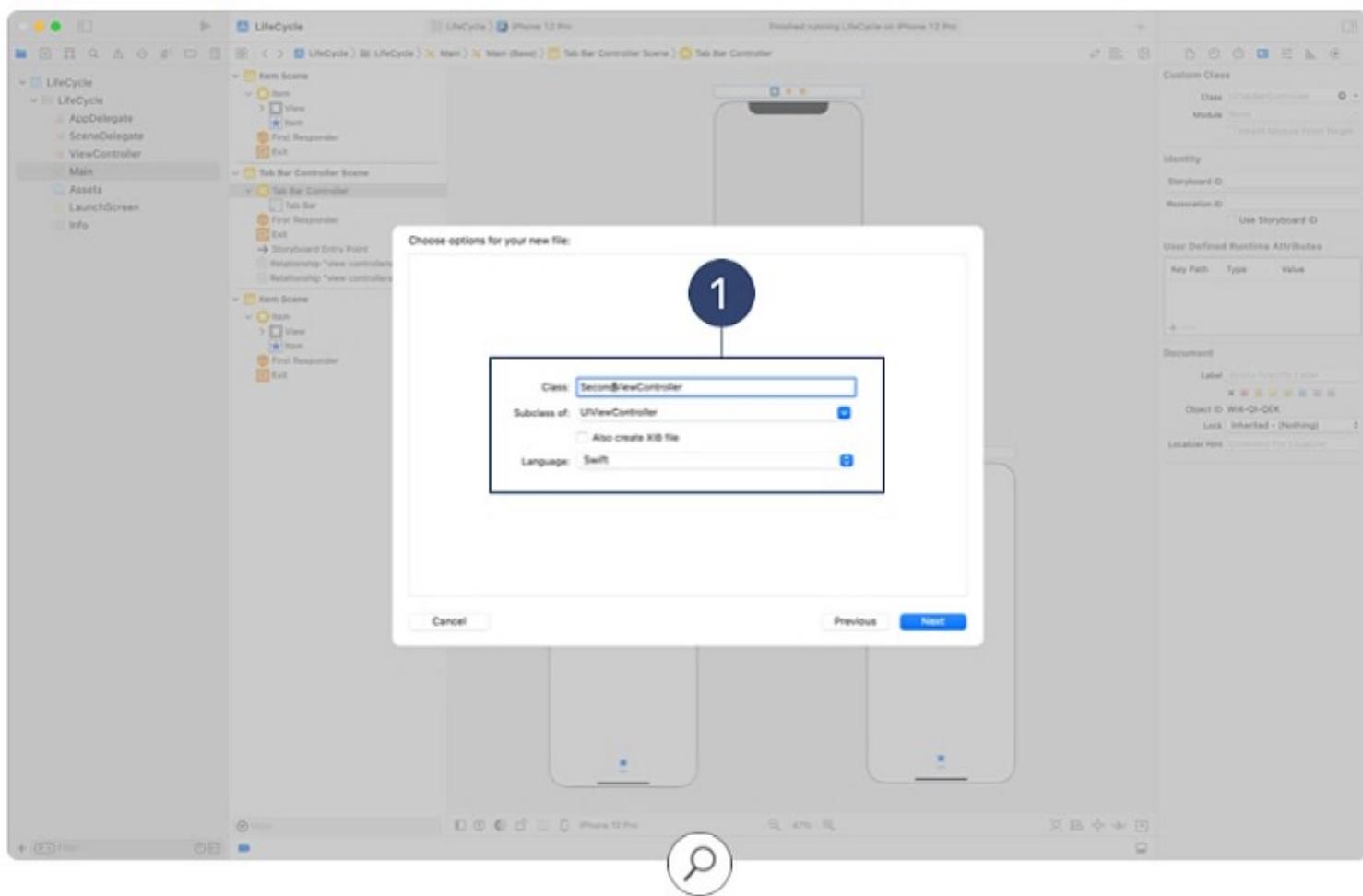
Open the **Main** storyboard. Select the **ViewController** scene and embed it in a tab bar controller.



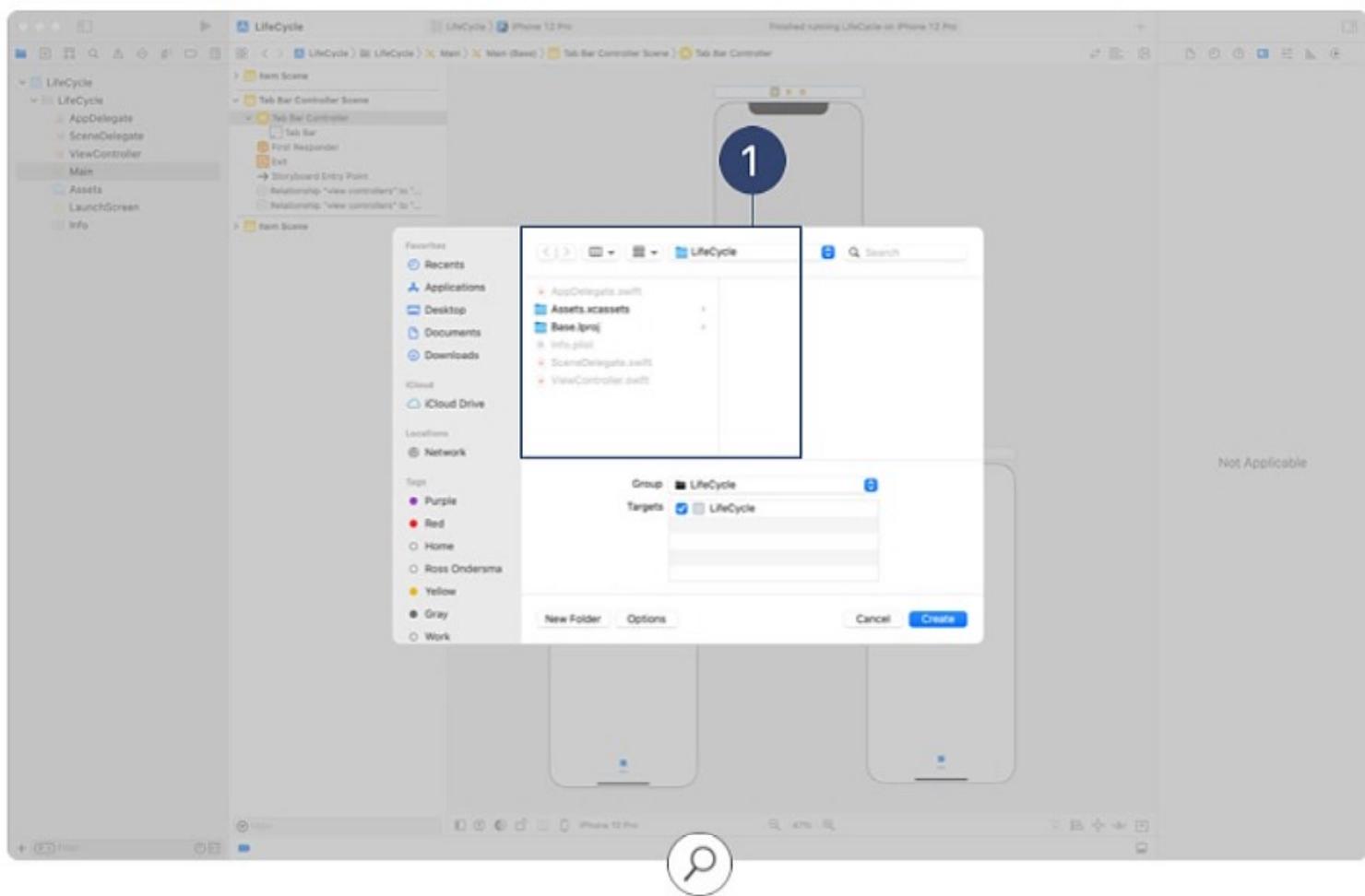
Next, drag a view controller from the Object library, and add it as a second tab in the tab bar controller. Feel free to organize the storyboard and update the background color of the view controllers.



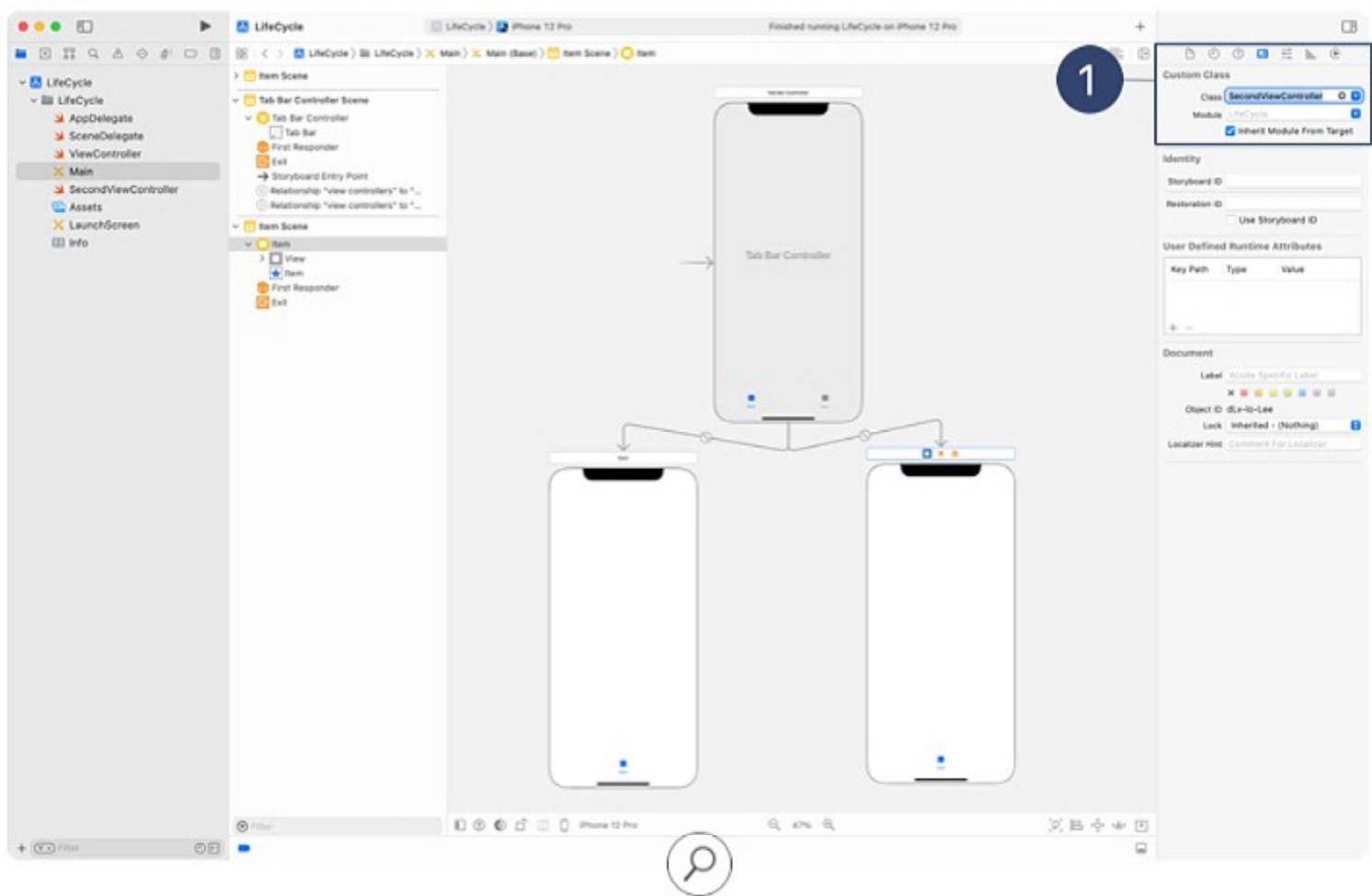
To write custom code for the new view controller, you'll have to add a new file. From the Xcode menu bar, choose **File > New > File** (or **Command-N**). Then select **Cocoa Touch Class** and click **Next**.



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



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

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.

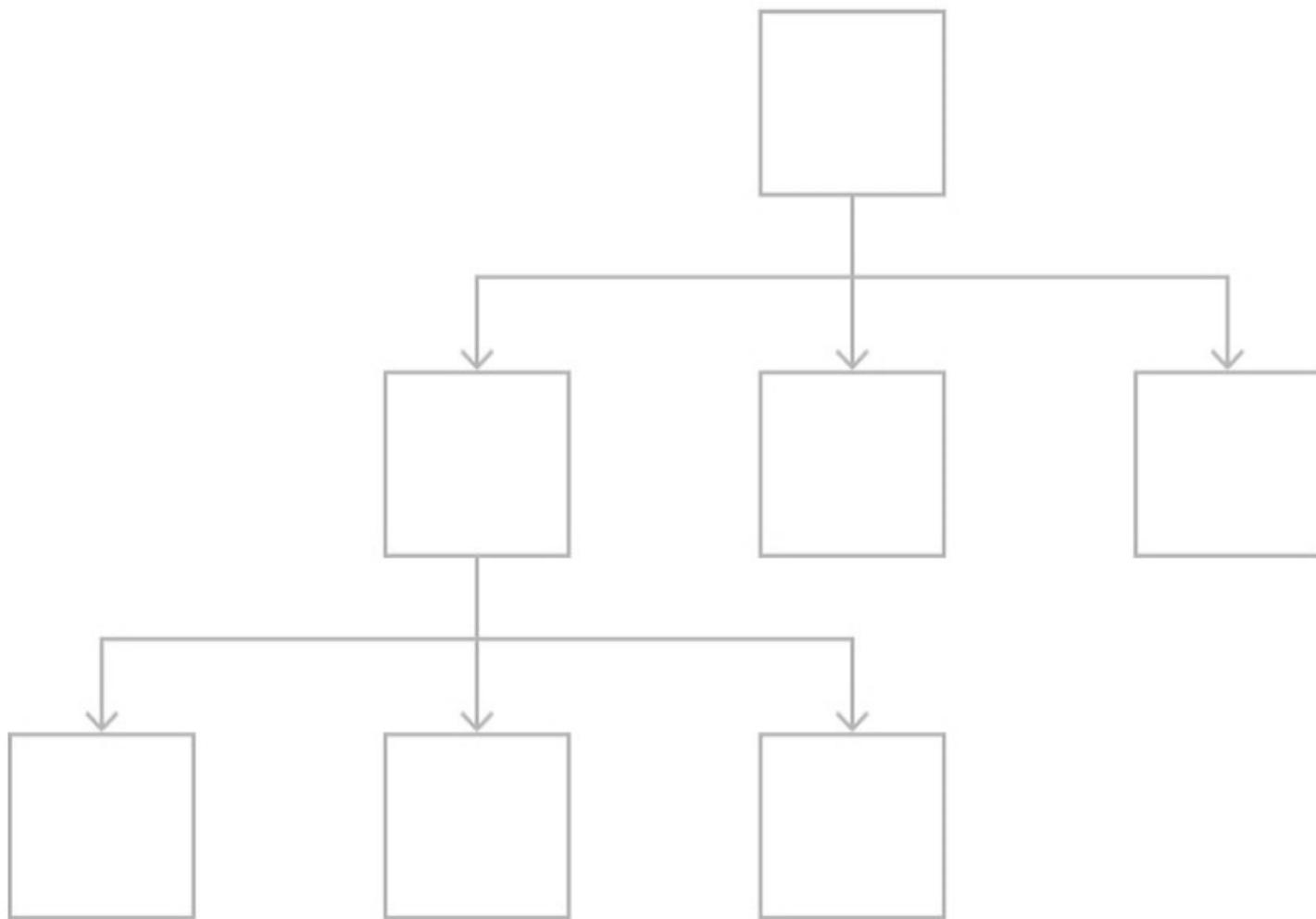
Navigation Hierarchy

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.



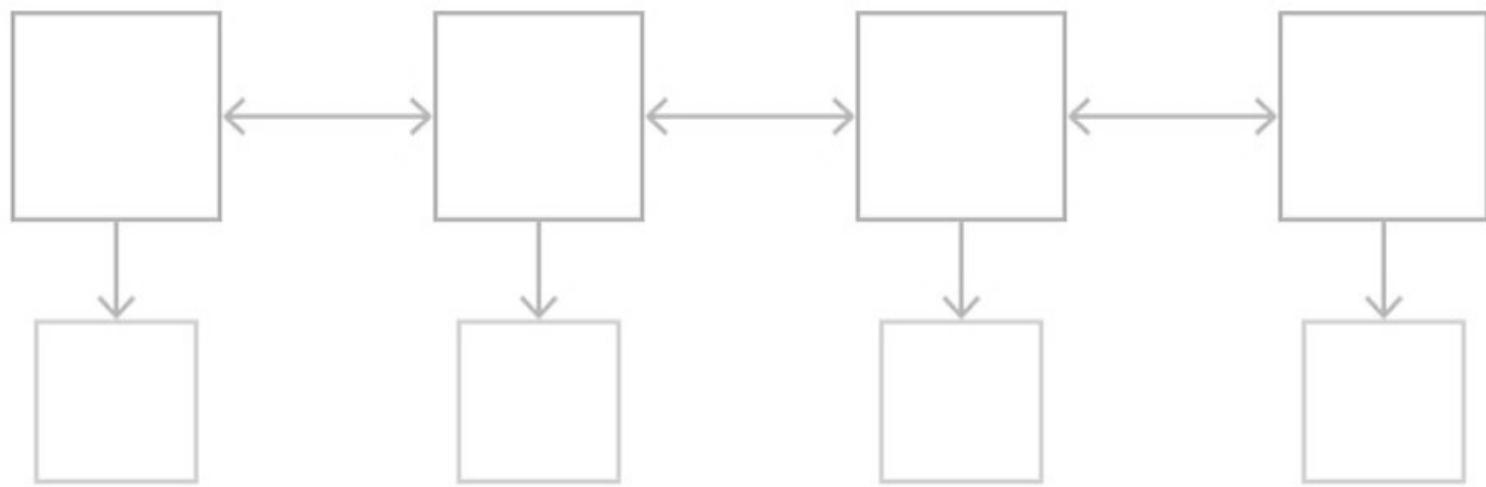
For this style, you'll typically use navigation controllers to enable navigation through a series of hierarchical screens. When a new screen is displayed, a Back button on the leading end of the navigation bar allows an easy return. (To give the user context, the bar may display the title of the current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



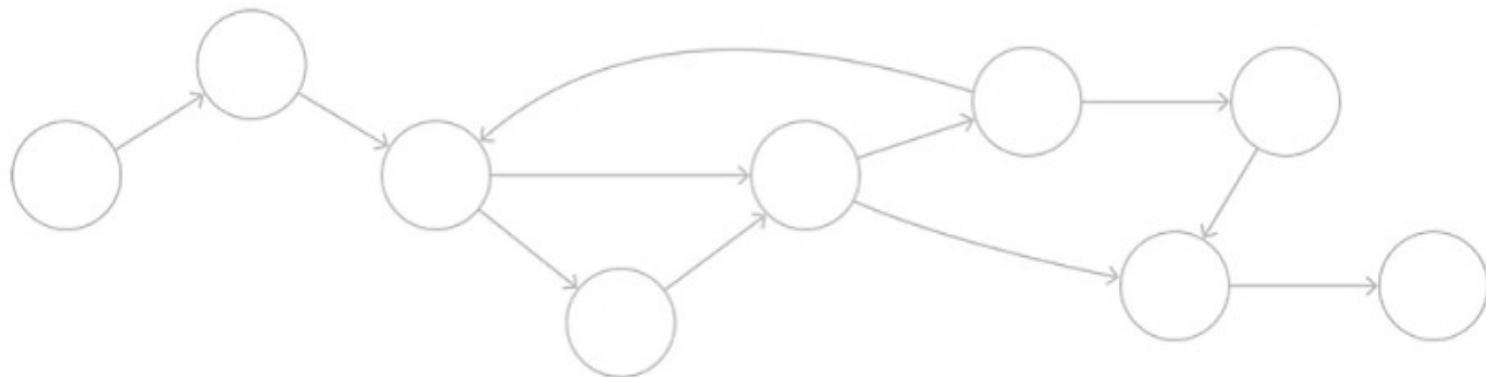
For flat navigation, you'll typically use a tab bar controller to organize information at the app level. A tab bar is a good way to provide access to several workflows or distinct modes of operation. You might also use a page controller.

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.

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:

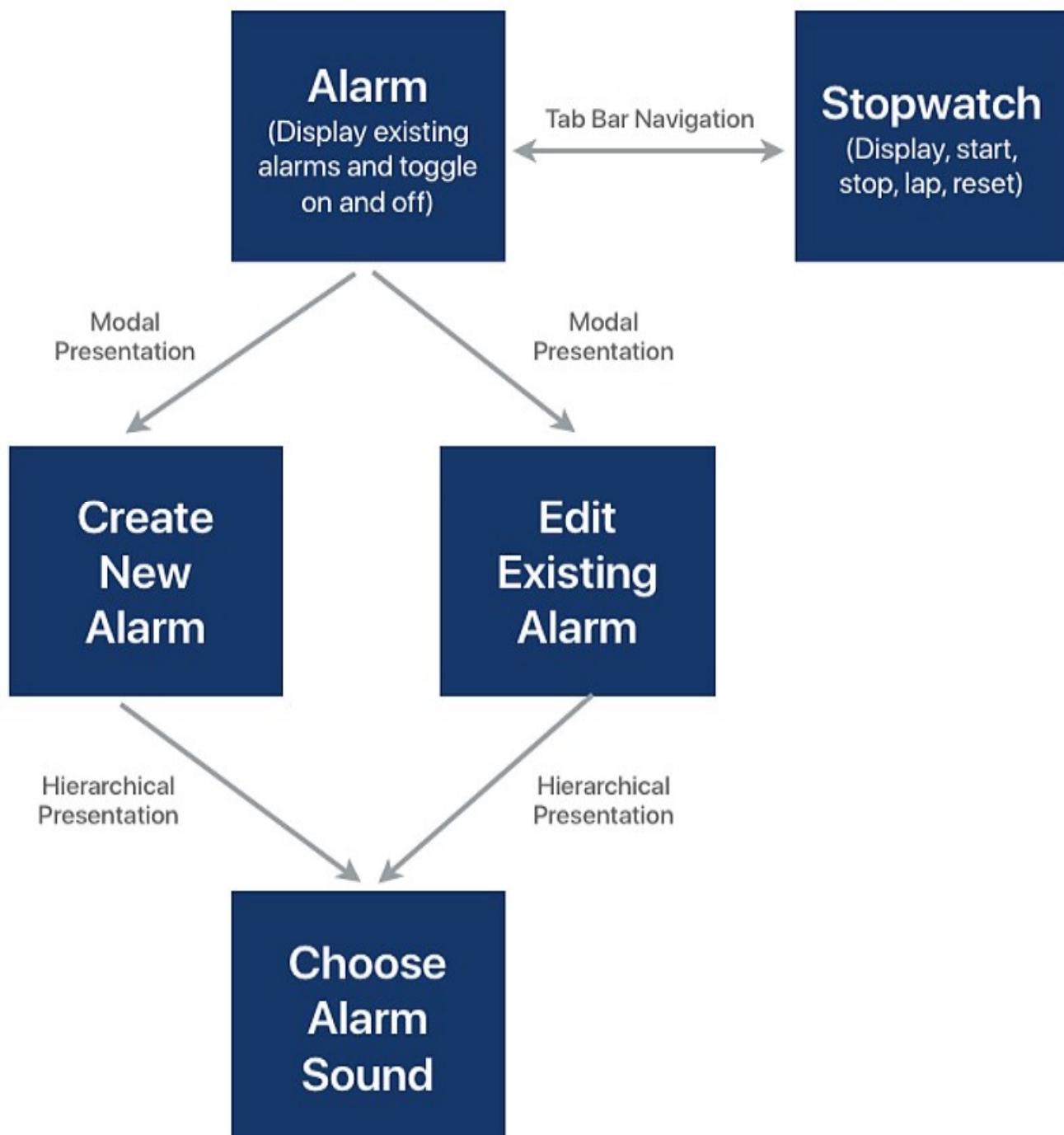
- Display alarms
- Toggle alarms on and off
- Create alarms
- Edit alarms
- Change the sound of alarms
- Basic stopwatch functionality (display, start, lap, stop, reset)

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.

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.

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

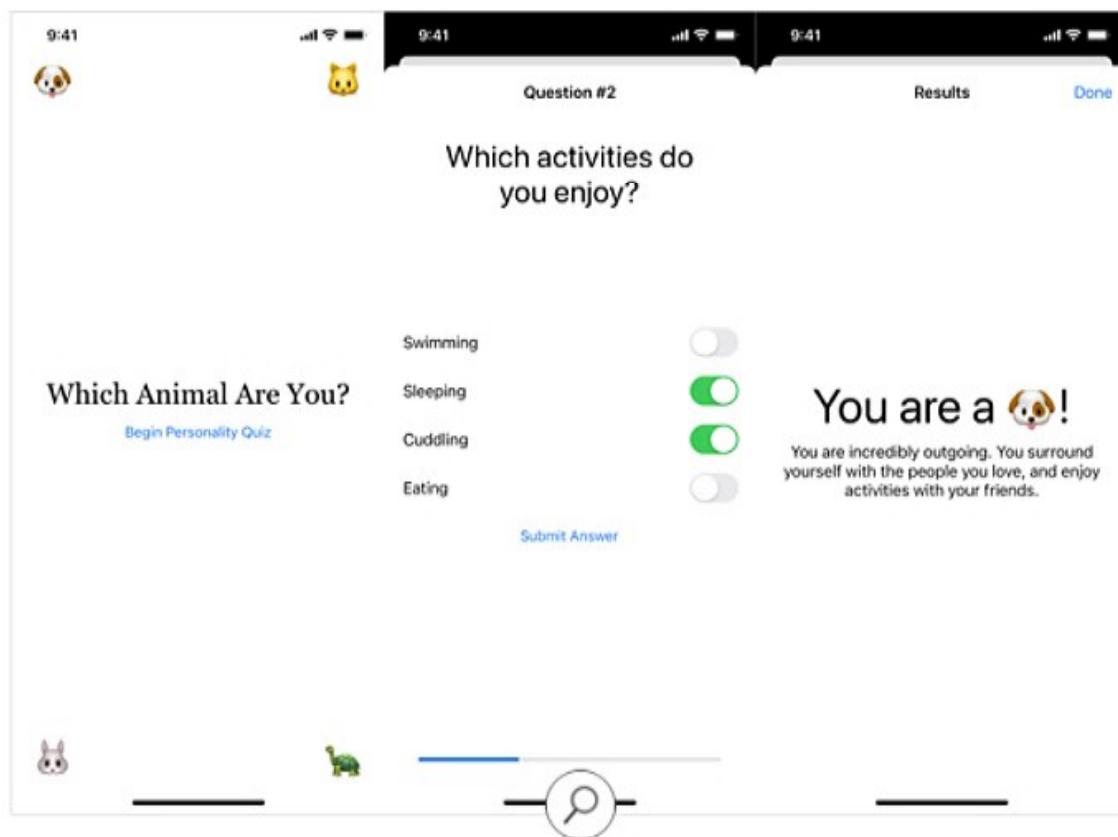
What type of data do you need for a personality quiz? At first, you might consider creating an array of strings to hold your questions, but where would you store the answers? A better idea would be to create a `Question` struct that holds a collection of answers. The collection of answers also needs 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

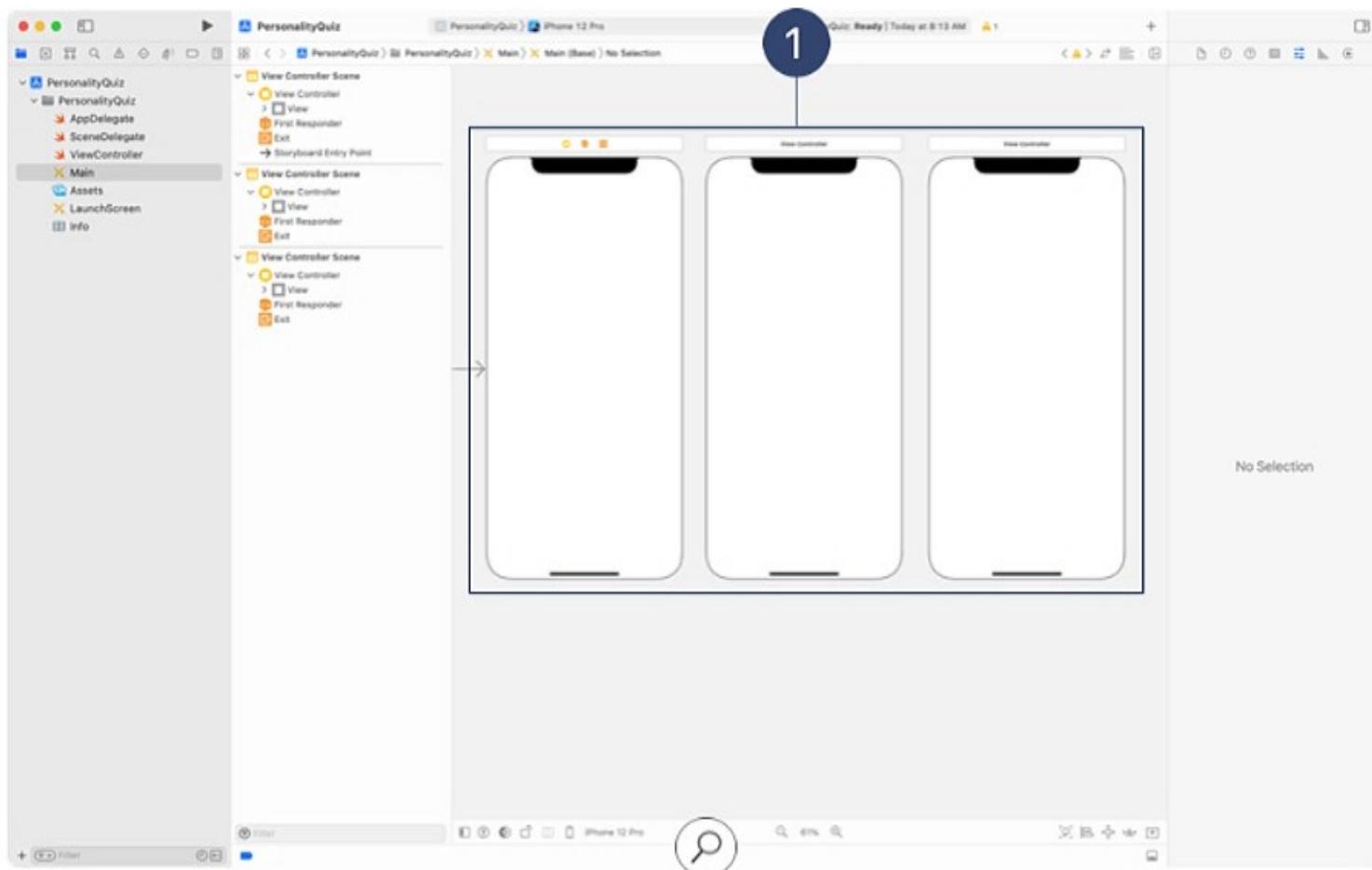
Now that you've analyzed which components you'll need, it's probably easier to see how the project will come together. You'll use three view controllers for your quiz:

- 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

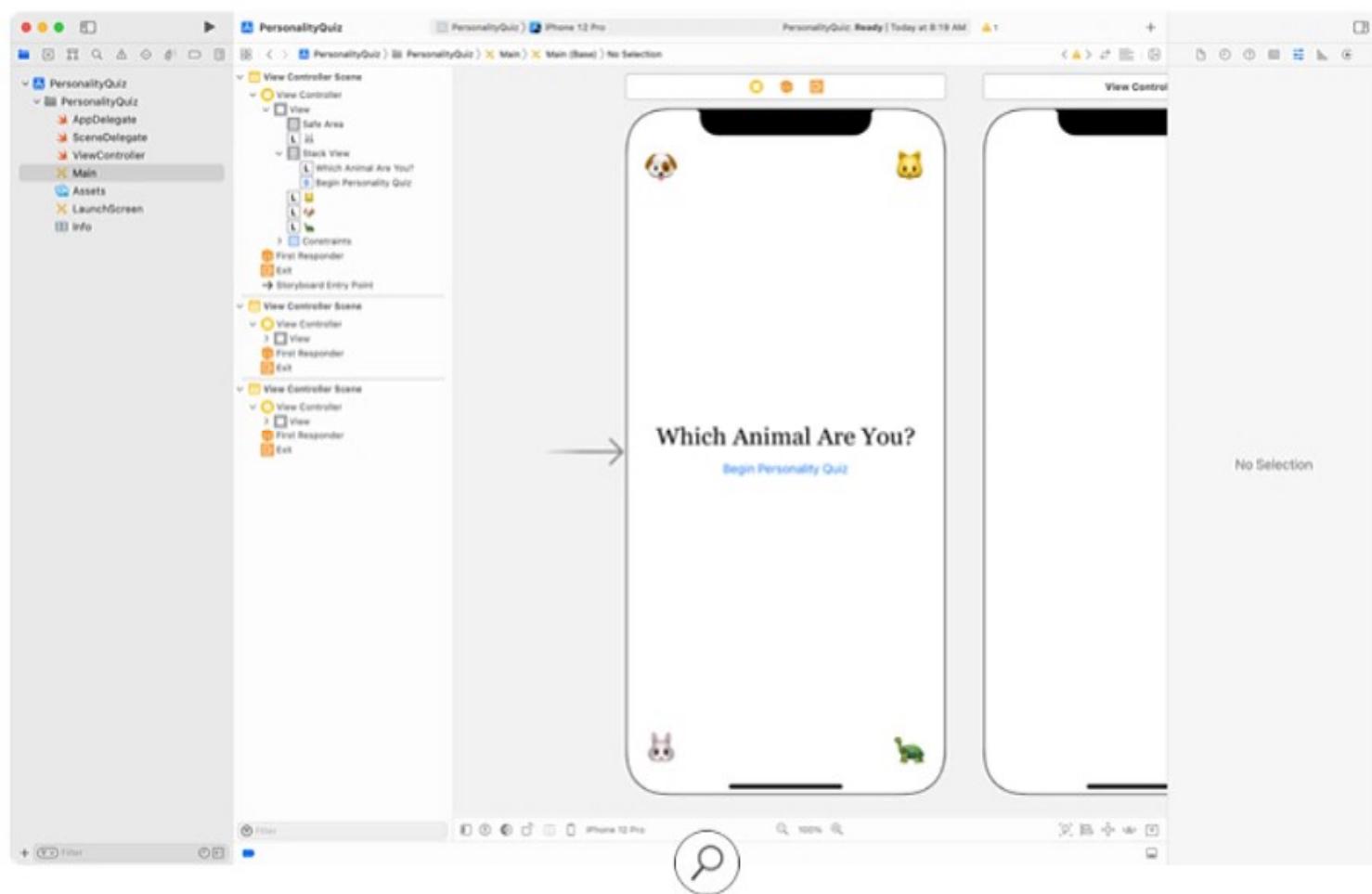
Create a new project using the iOS App template. Name it “PersonalityQuiz,” and open the Main storyboard. The storyboard already contains one view controller, but you’ll need two more. Drag two view controllers from the Object library onto the canvas, and position all three in a horizontal row. ①



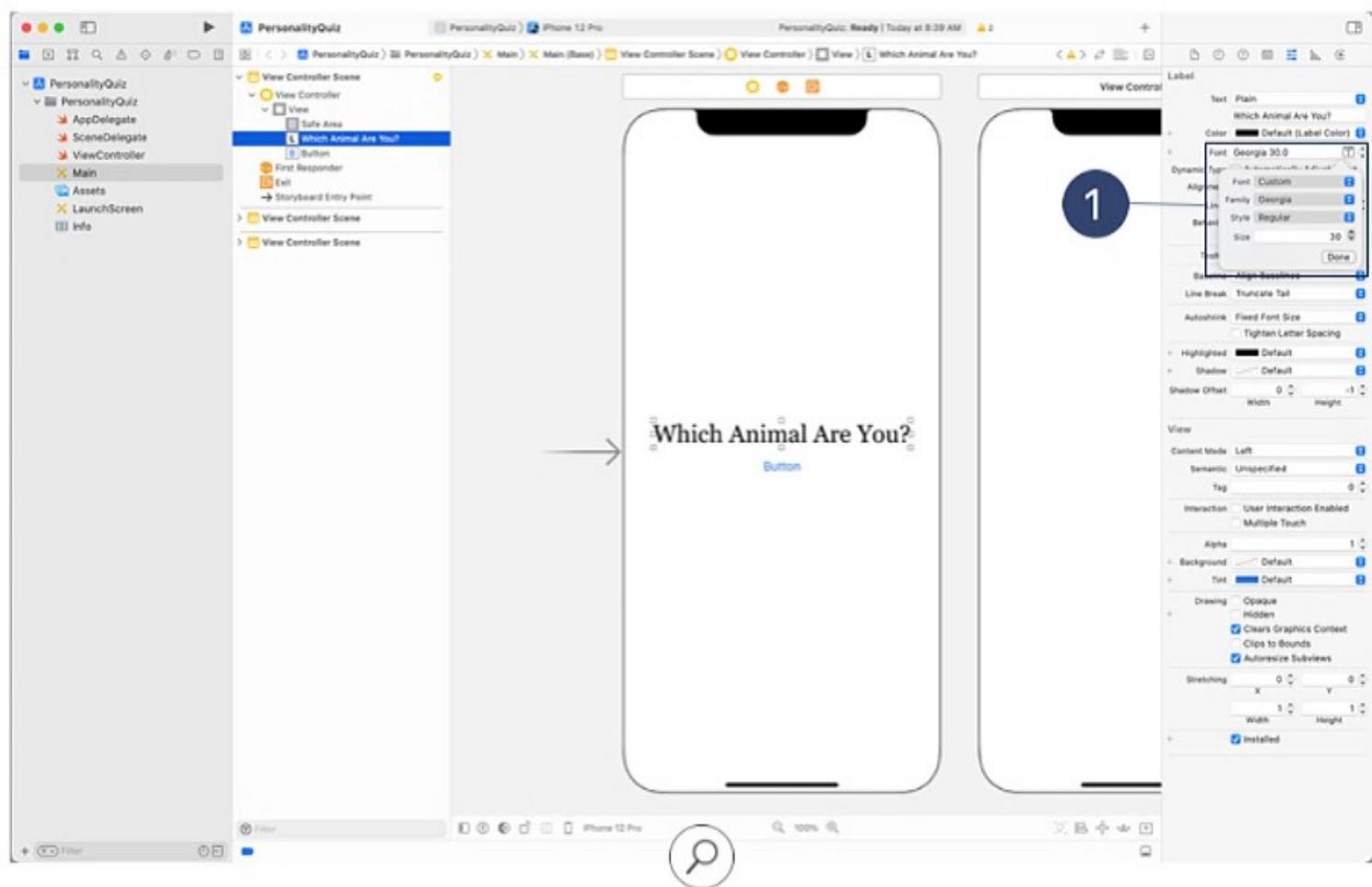
Create the Introduction Screen

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.

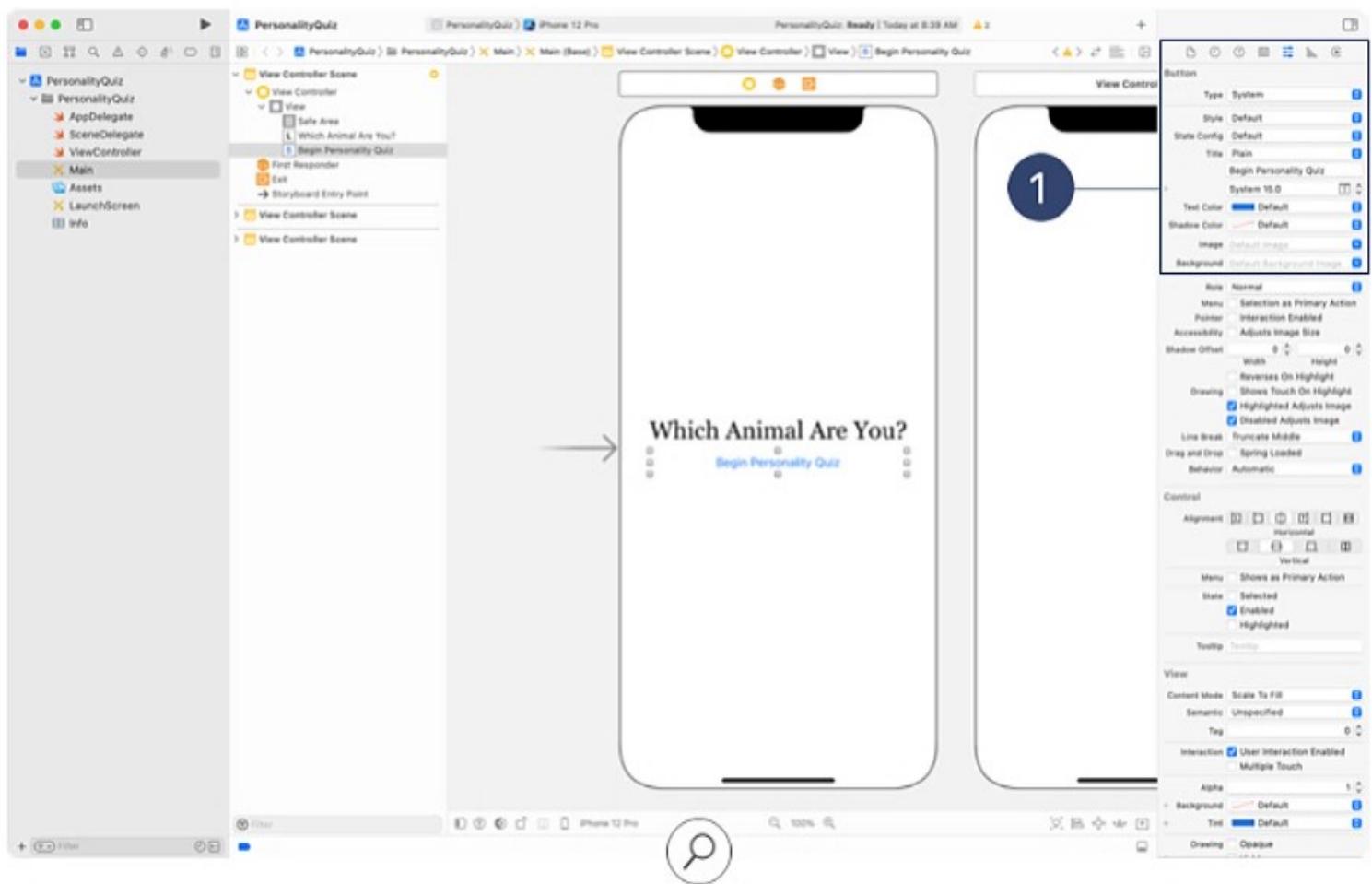
As an example, take a look at the introduction screen below, then follow the instructions to learn how to build it. You can use the same screen, or something similar, as long as the topic of the quiz is clear to the player.



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

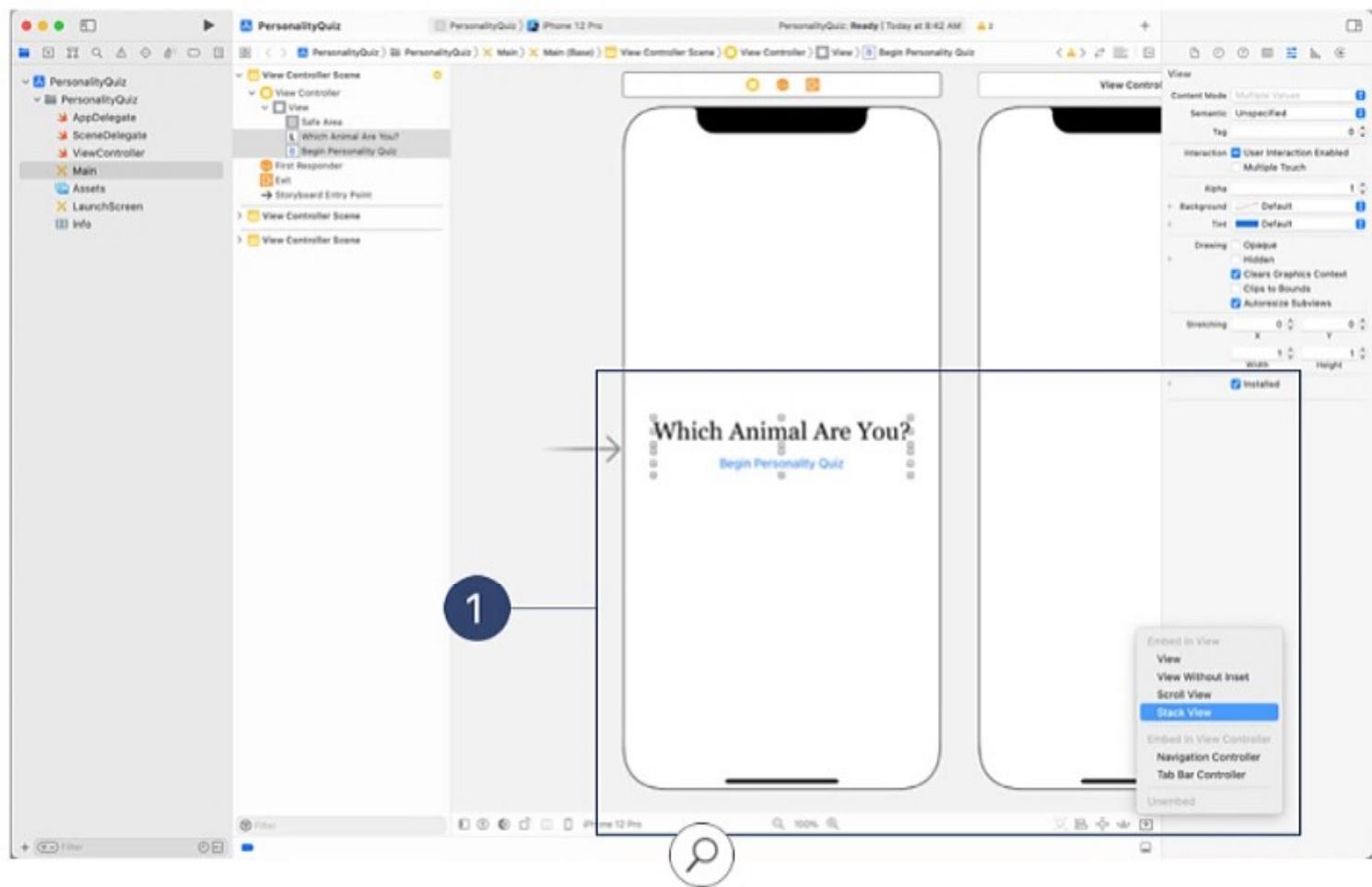


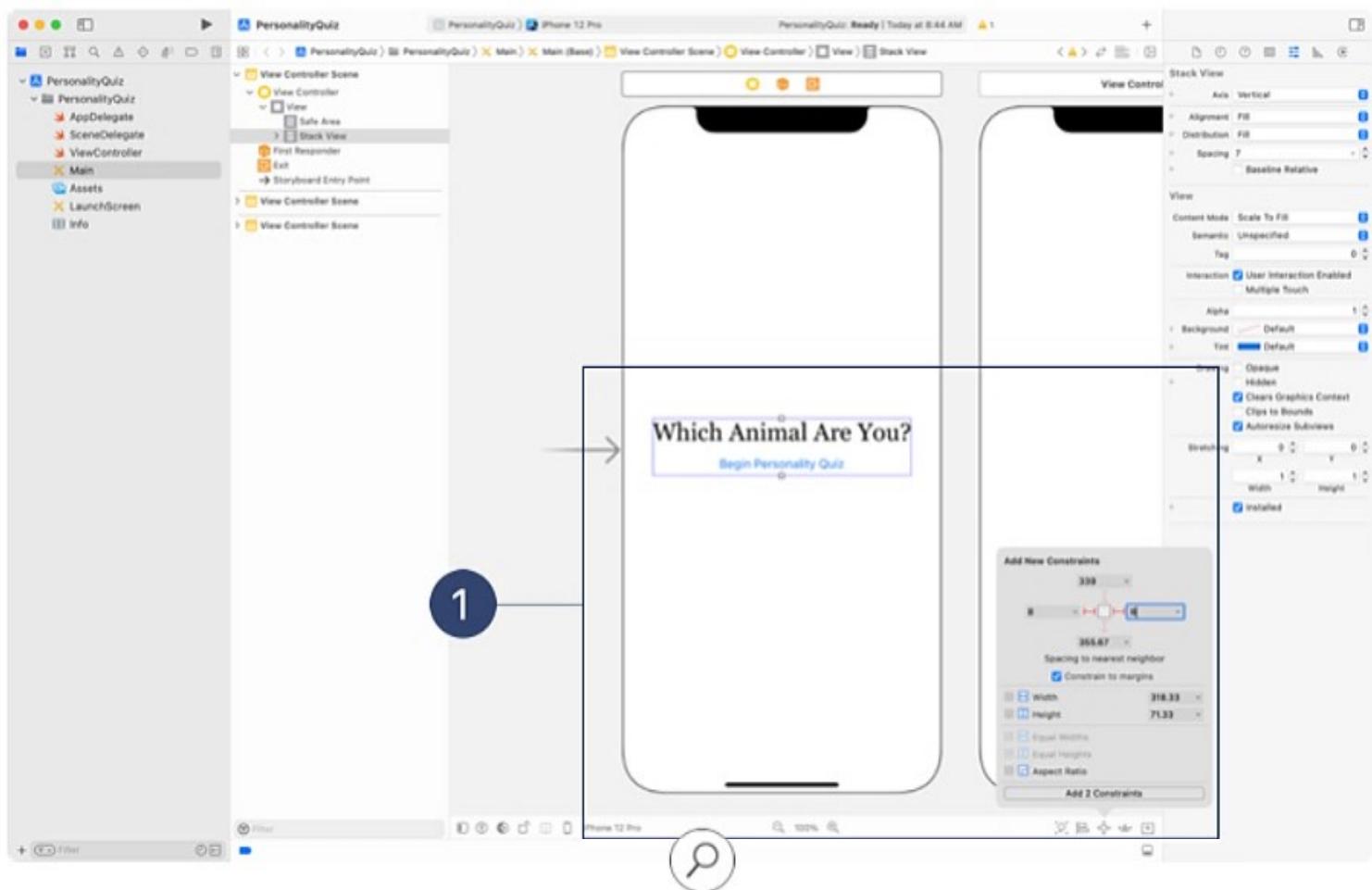
Select the button and update the title to read "Begin Personality Quiz" using the System font 15.0. ①



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

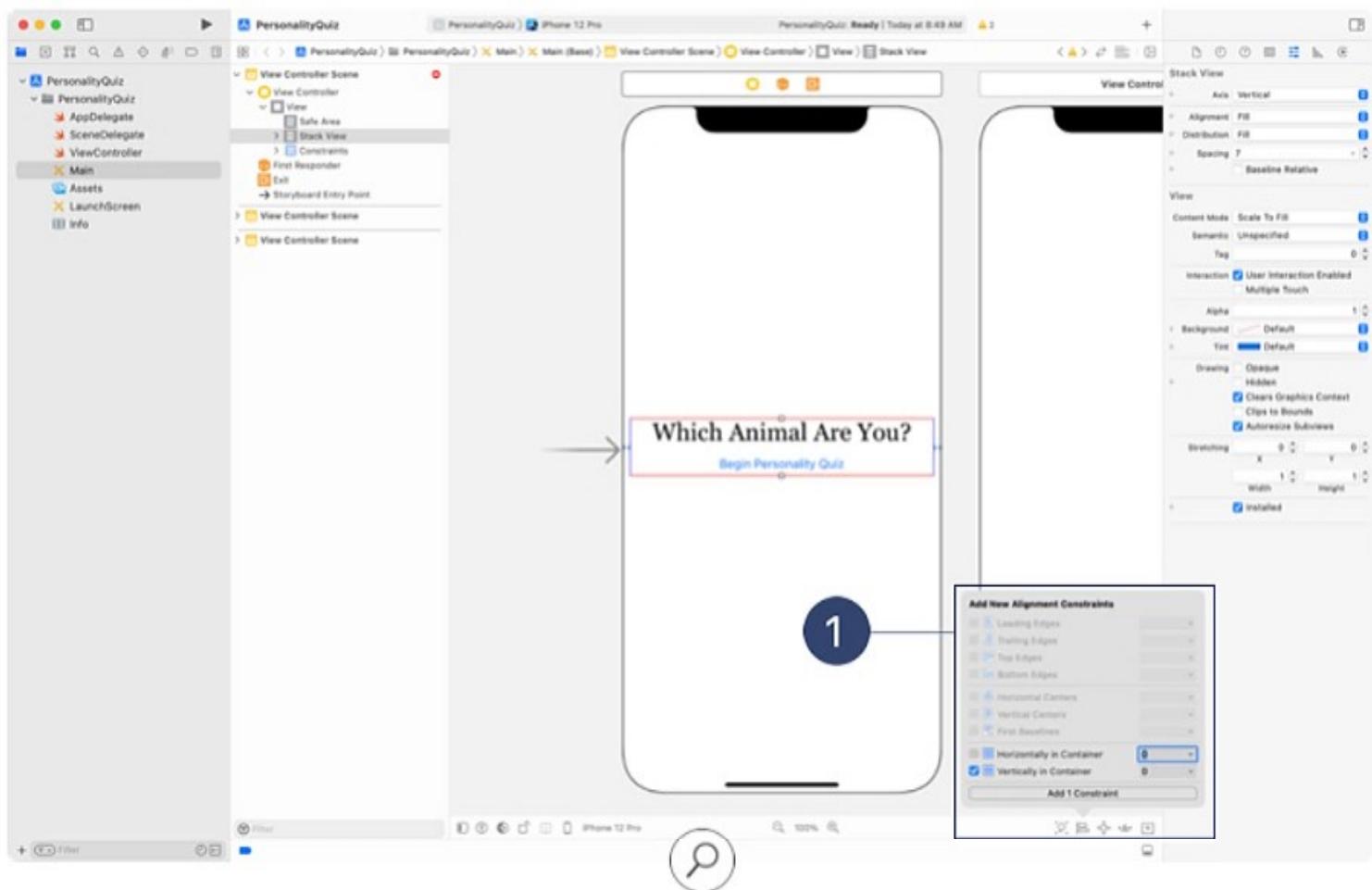




With the stack still selected, use the Attributes inspector to check that the Axis is set to Vertical, and set both the Alignment and Distribution to Fill. These settings ensure that elements in the stack are positioned vertically and that they fill all available space along the stack view's axis.

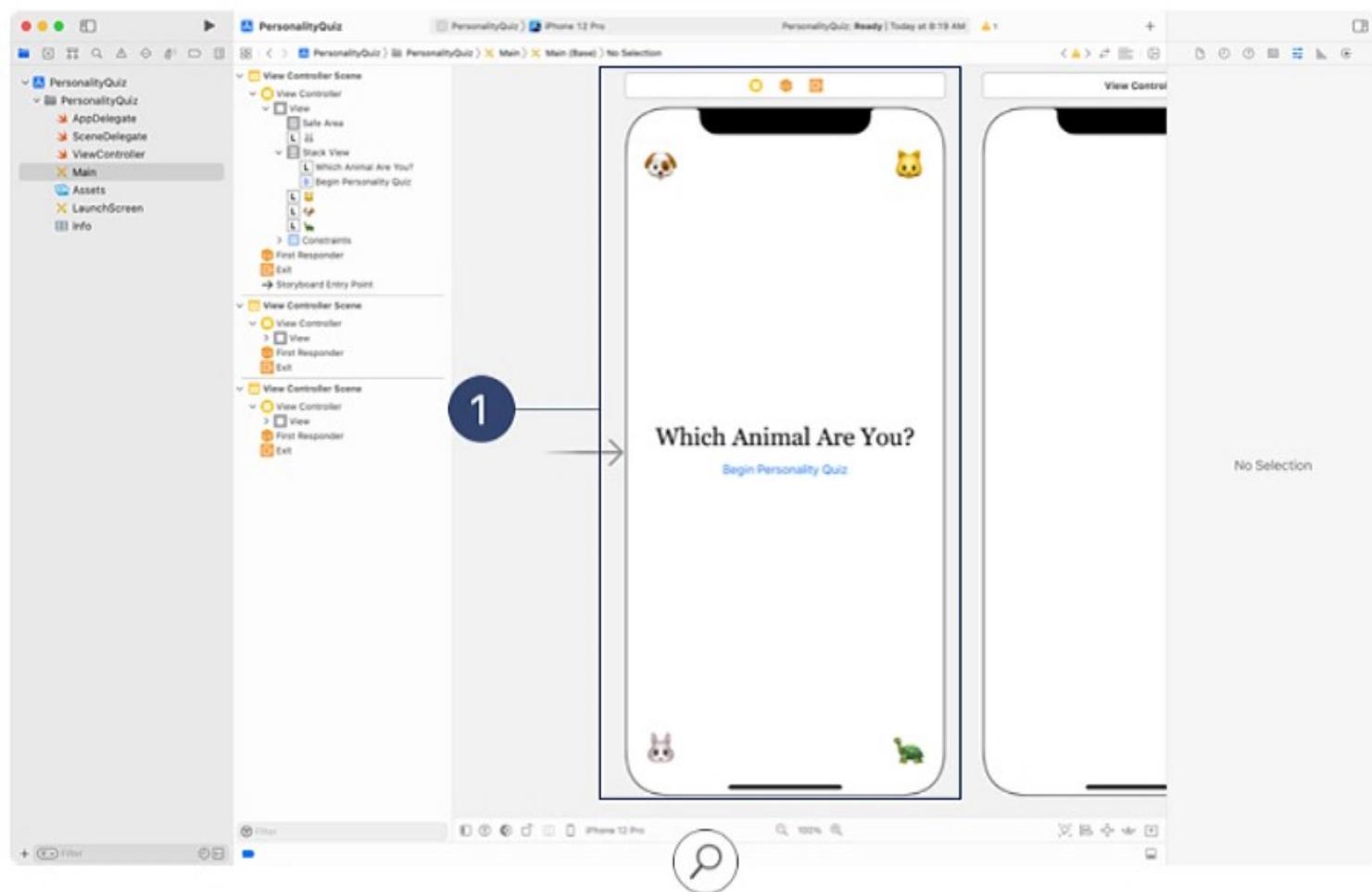
The label and button should be centered both horizontally and vertically. Although you can create two constraints to do that, the label is too close to the screen's horizontal bounds. Use leading and trailing constraints to ensure that it doesn't run off the screen no matter the device size.

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

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.



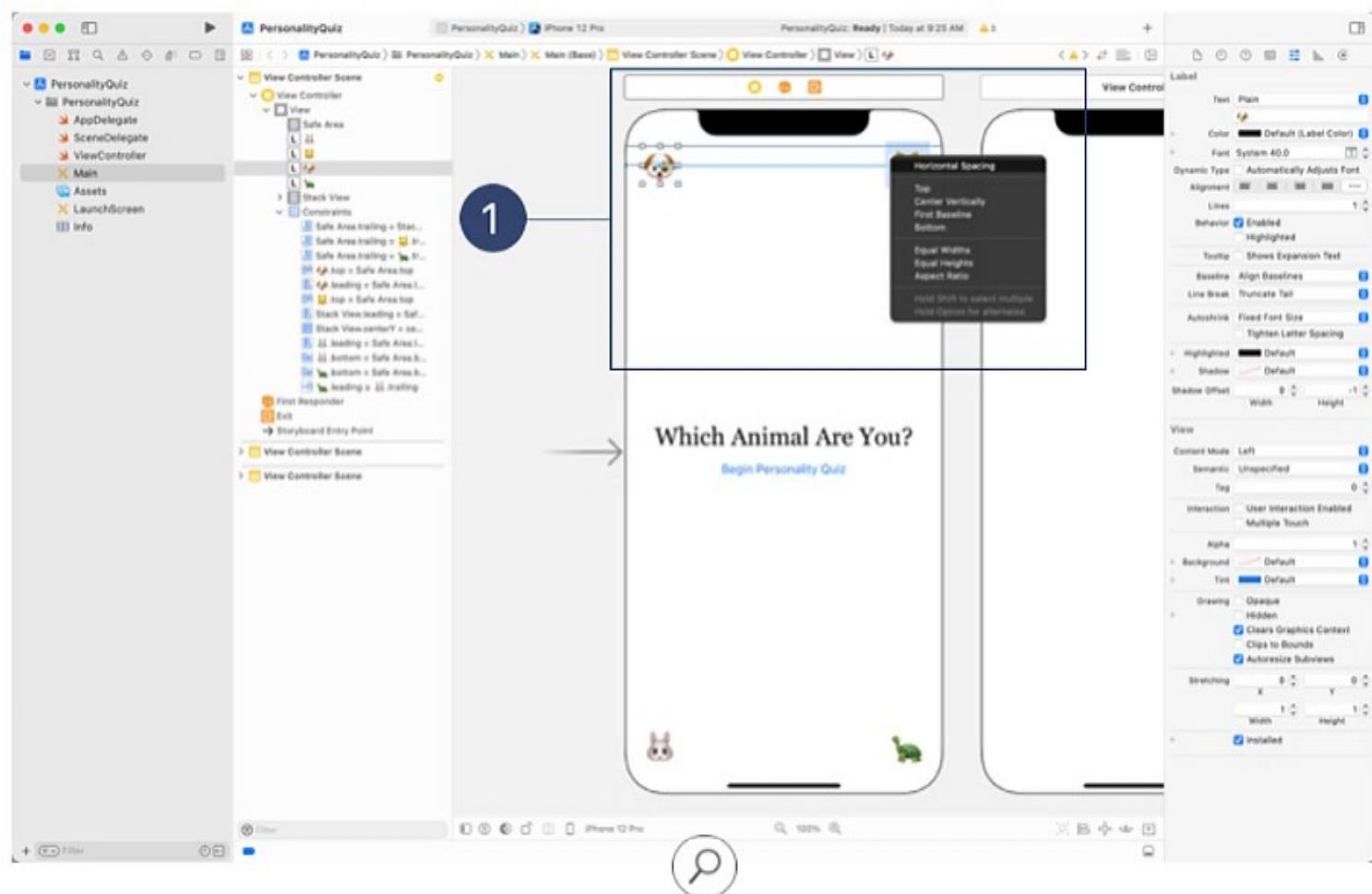
If you haven't already, drag four more labels from the Object library onto the view. Replace the text with the emoji for each animal. To bring up the emoji picker, highlight the label text in the Attributes inspector, then press **Control-Command-Space**. Enlarge all the emoji by setting the Font to 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.

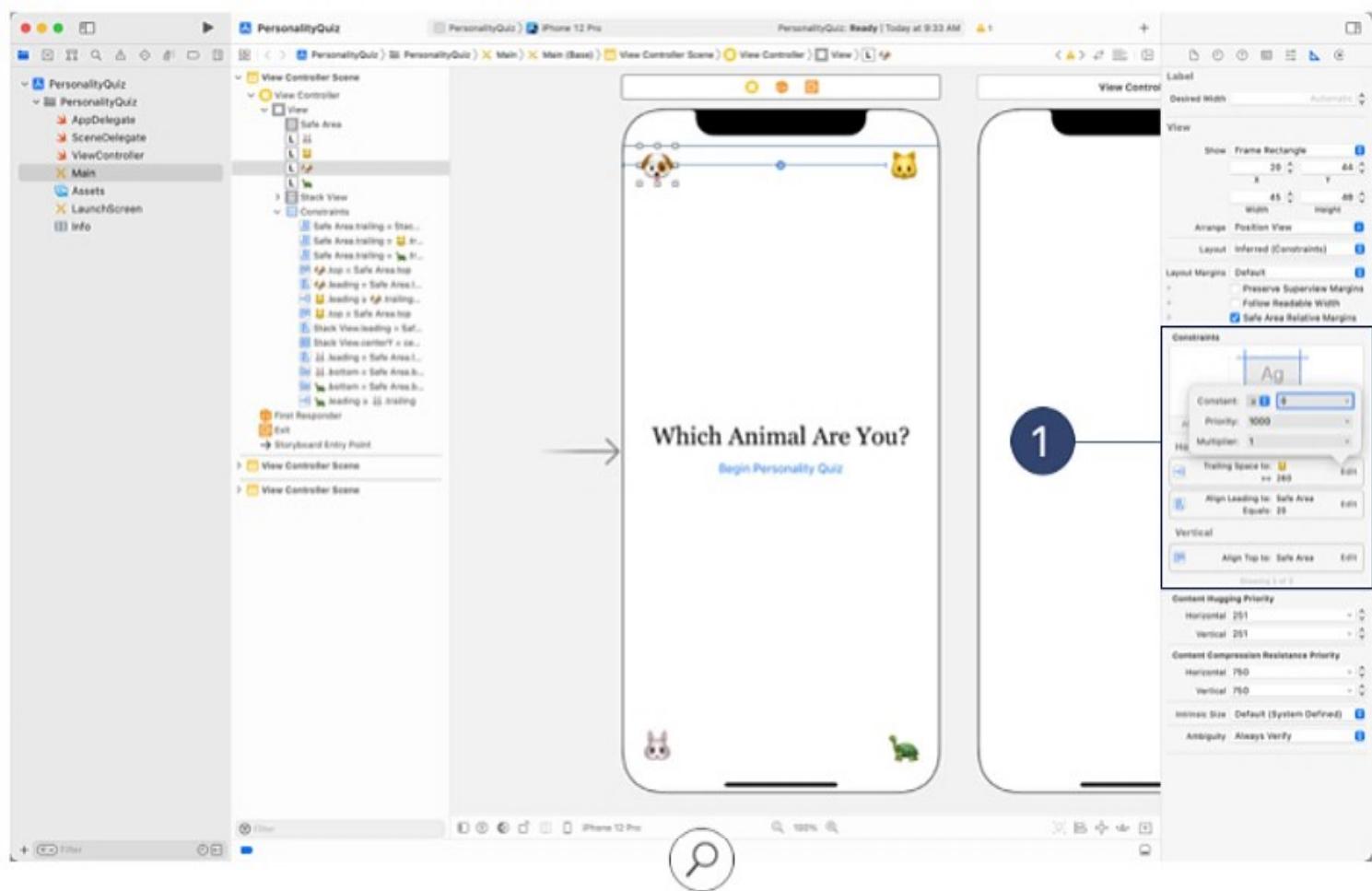


Xcode isn't satisfied with only these constraints. It produces a warning notifying you that trailing and leading constraints are missing, which may cause overlapping with other views. The reason is that labels often have dynamic values or sizes. For this interface design, you're fine leaving it as is.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

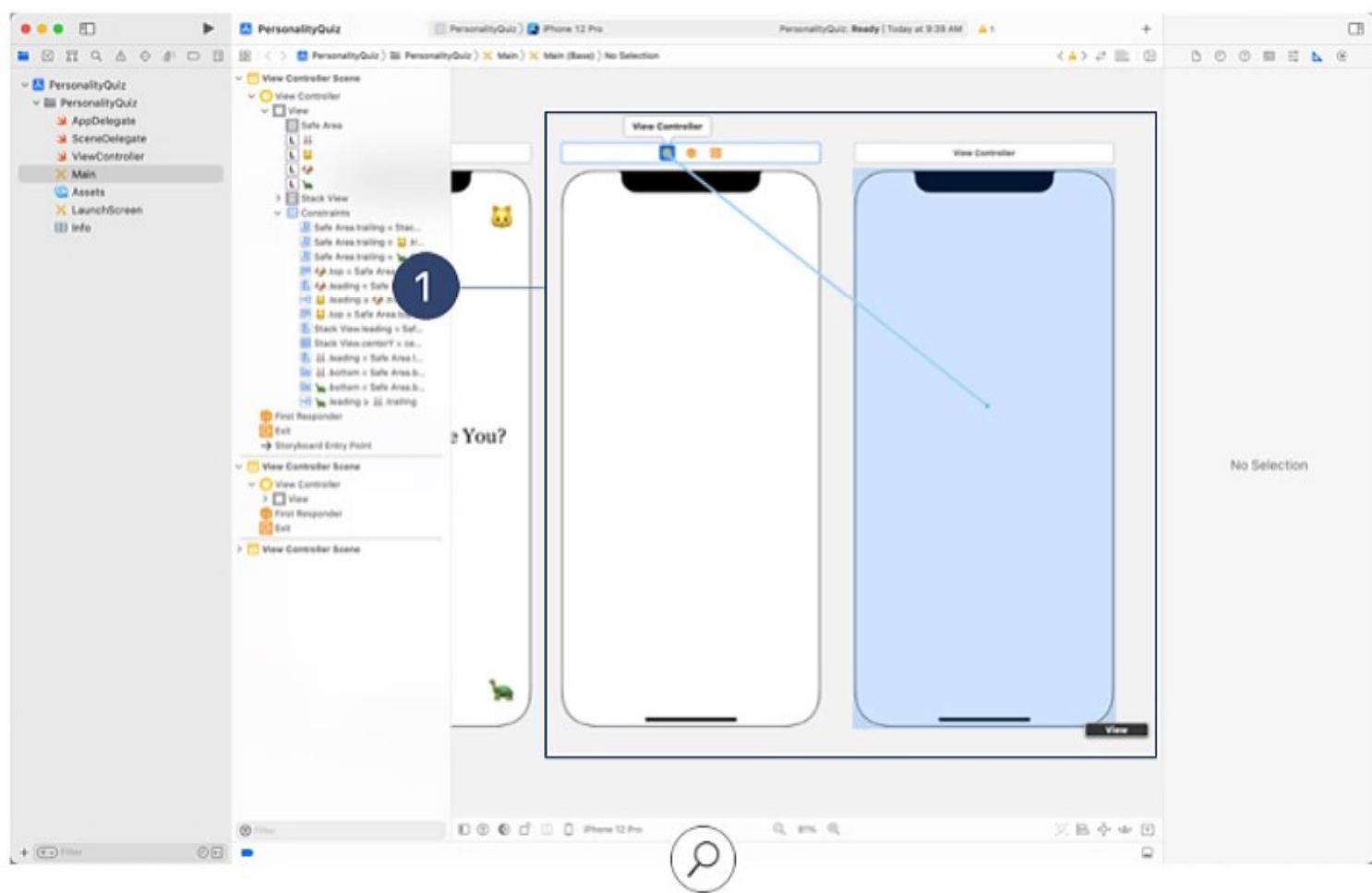
The second view controller will display each question, one at a time, along with input controls that allow the player to respond. The controls you use—buttons, switches, or sliders—need to make sense for the questions and answers in your quiz. You’ll think through the controls a bit later in the project.

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.

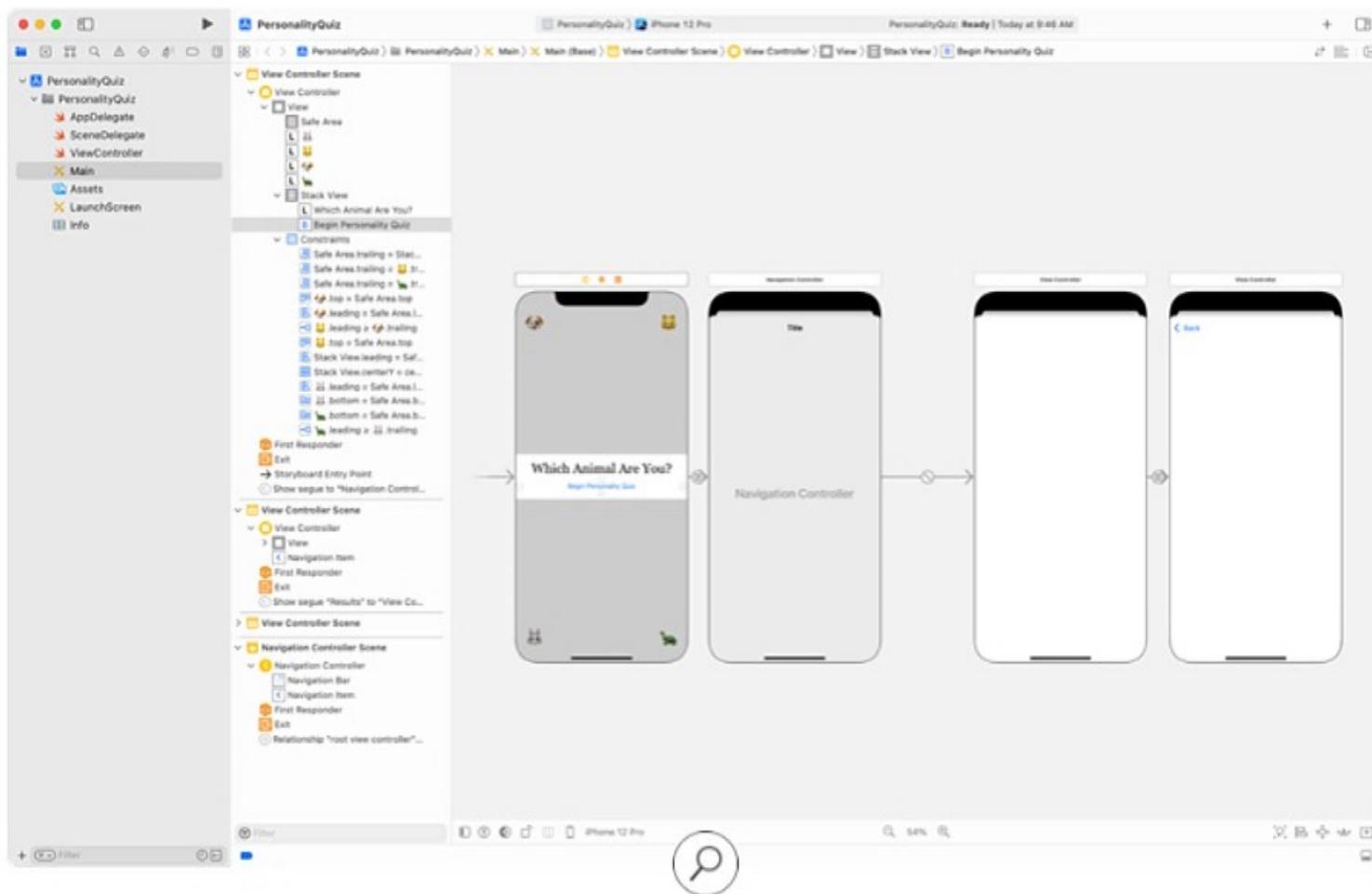
Instead, you can invoke a segue programmatically between the second and third view controllers. **Control-drag** from the view controller icon, above the second view controller's view, to the third view controller, ① and create a Show segue. Highlight the segue in the storyboard. Then use the Attributes 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

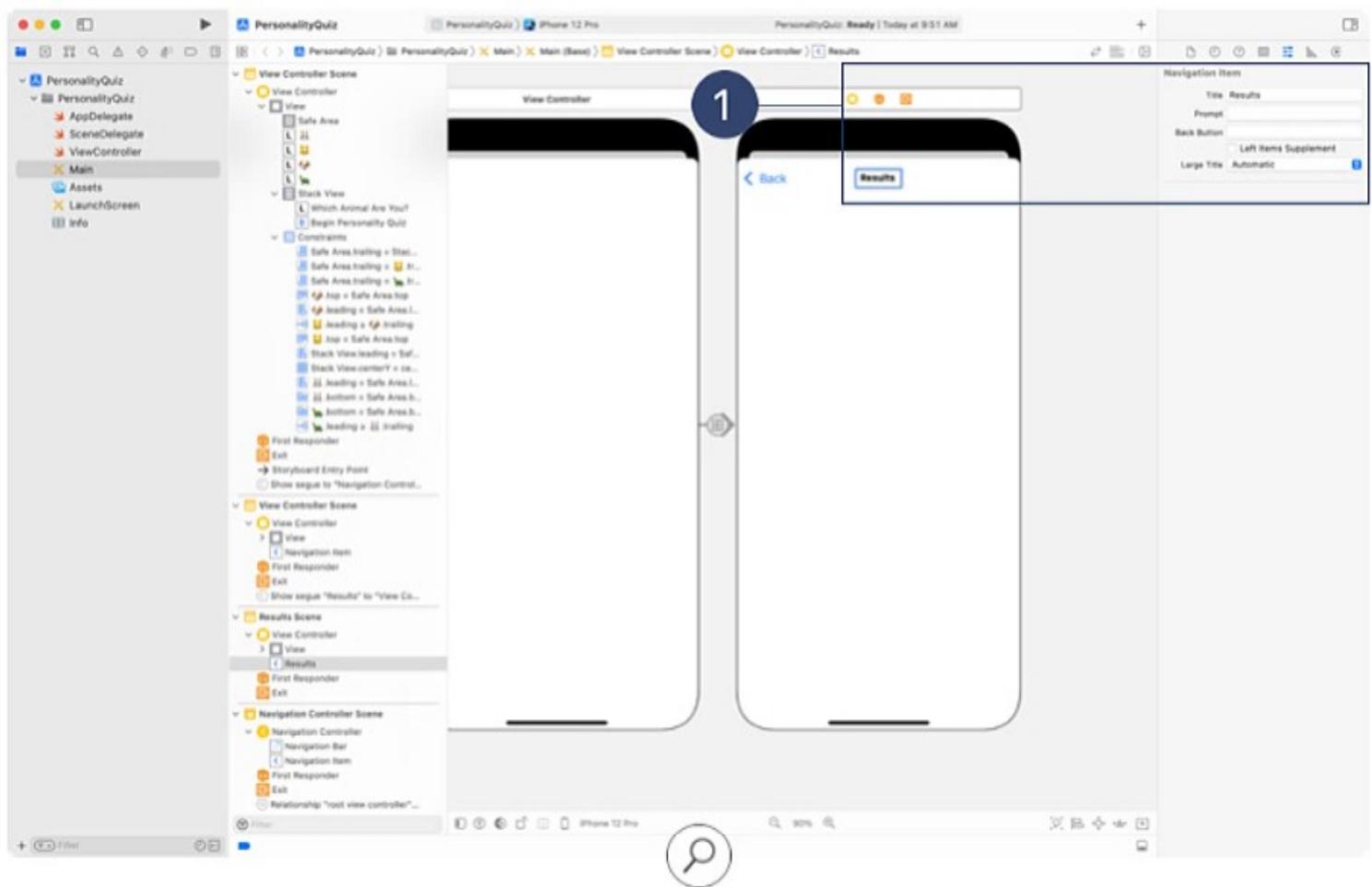
After the player has finished the quiz, the app will display the results, along with a short description. The player should have some way to dismiss the results and return to the introduction screen so that another player can take the quiz. Here's an example of this interface:



You are a 🐶!

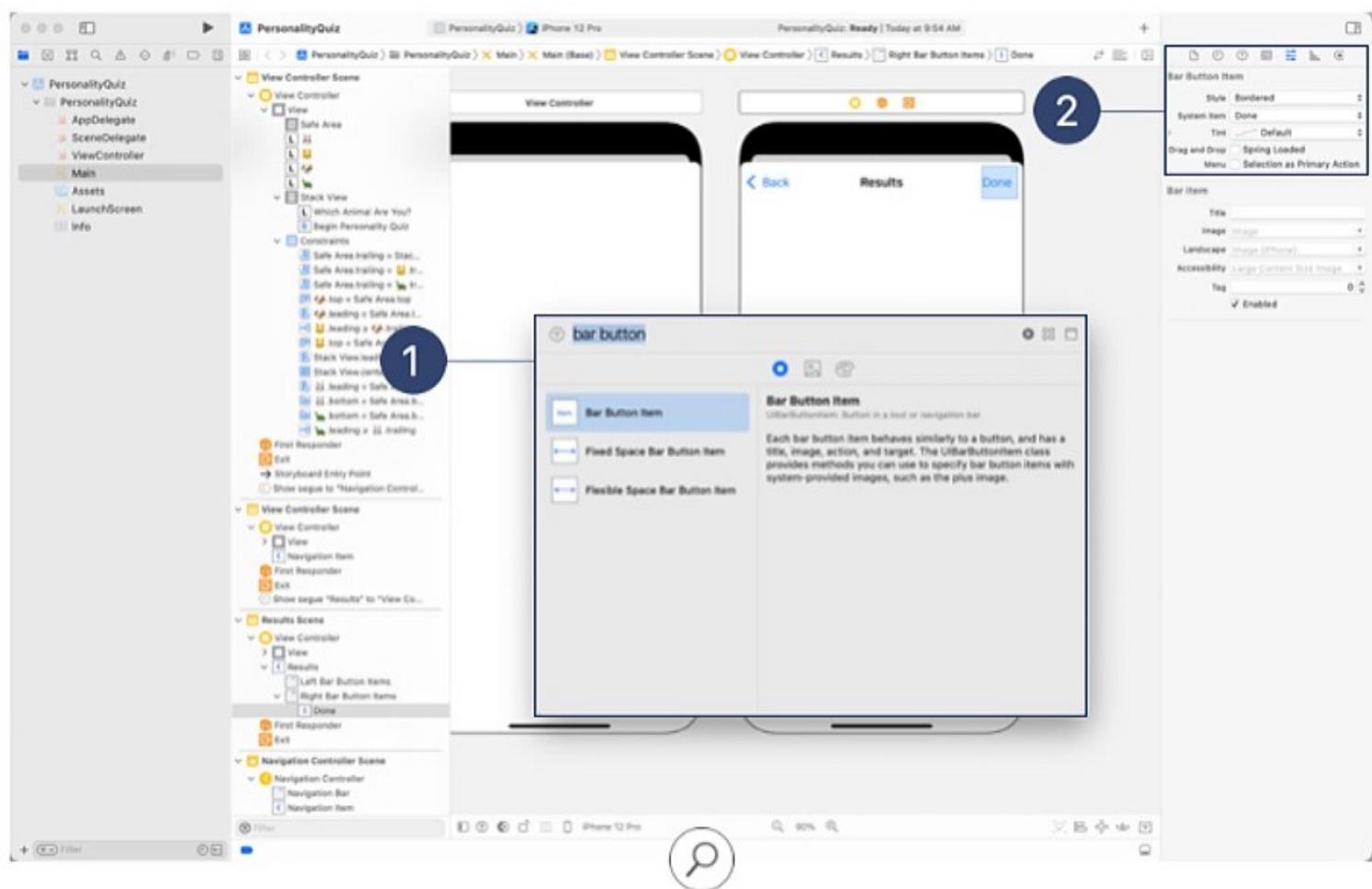
You are incredibly outgoing. You surround yourself with the people you love, and enjoy activities with your friends.

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.

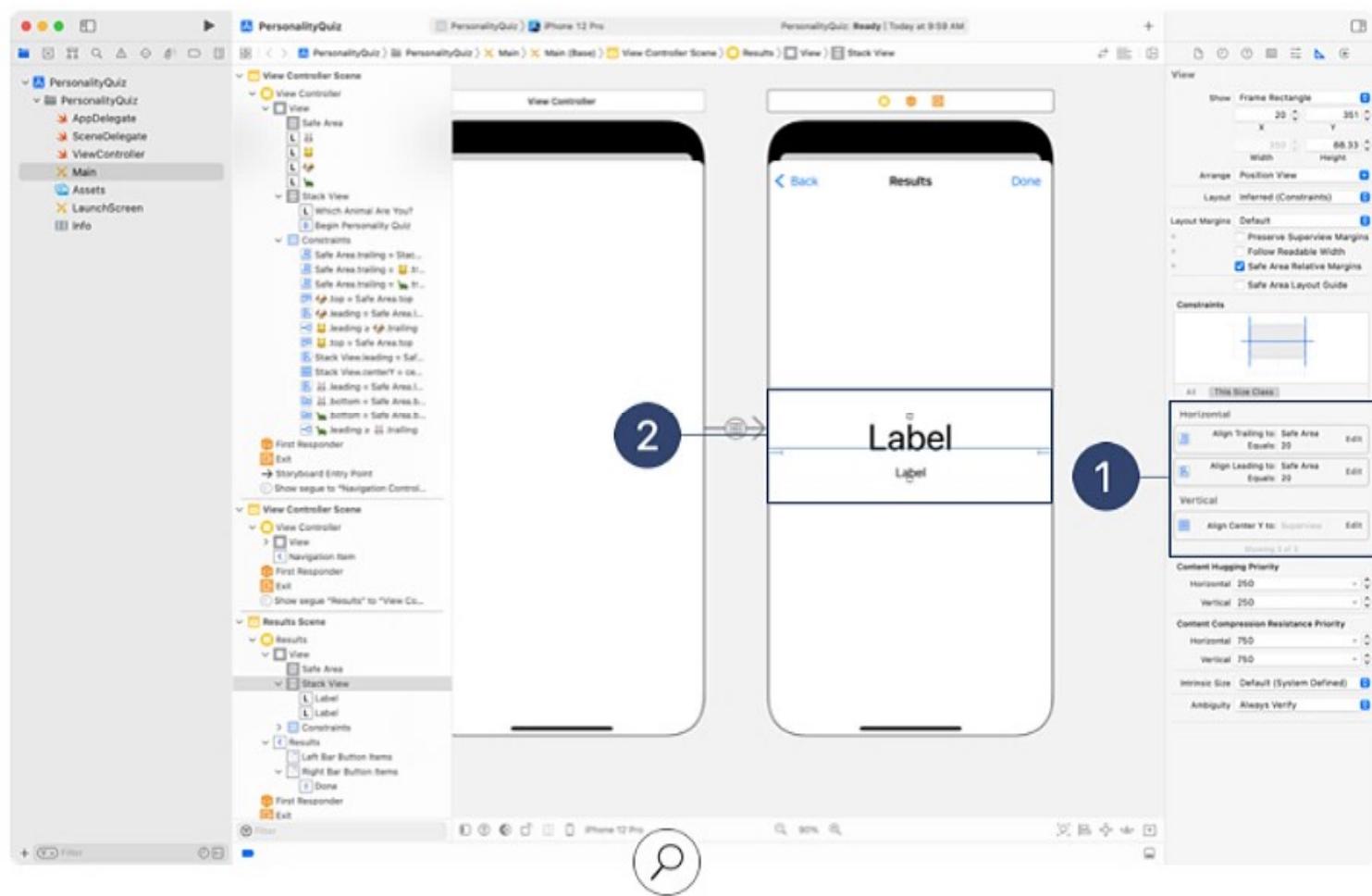


Update the final view controller's title to "Results" in the Attributes inspector for its Navigation Item, or simply double-click in the navigation item. ①

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



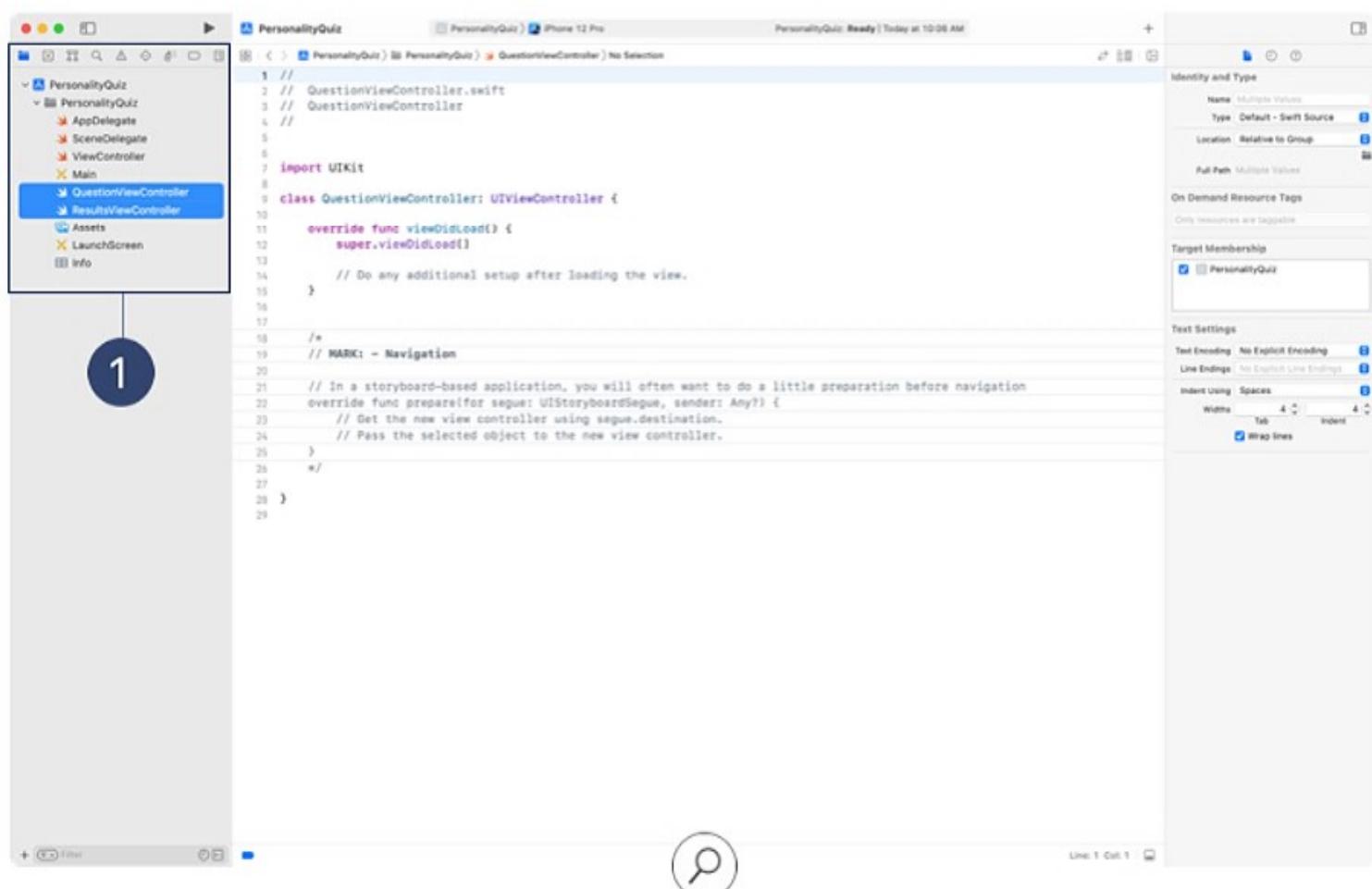
By now you're a pro at positioning stack views with constraints. To replicate the results screen, create a vertical stack view with two labels. Use the Align tool to center the stack vertically, and use the Add New Constraints tool to set the leading and trailing edges to 20. ① Use System Font 50.0 for 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



```

1 // QuestionViewController.swift
2 // QuestionViewController
3 // AppDelegate
4 // SceneDelegate
5 // ViewController
6 // Main
7 import UIKit
8
9 class QuestionViewController: UIViewController {
10
11     override func viewDidLoad() {
12         super.viewDidLoad()
13
14         // Do any additional setup after loading the view.
15     }
16
17
18     /*
19     // MARK: - Navigation
20
21     // In a storyboard-based application, you will often want to do a little preparation before navigation
22     override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
23         // Get the new view controller using segue.destination.
24         // Pass the selected object to the new view controller.
25     }
26     */
27
28 }
29

```

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 `ResultsController`.

Part Three

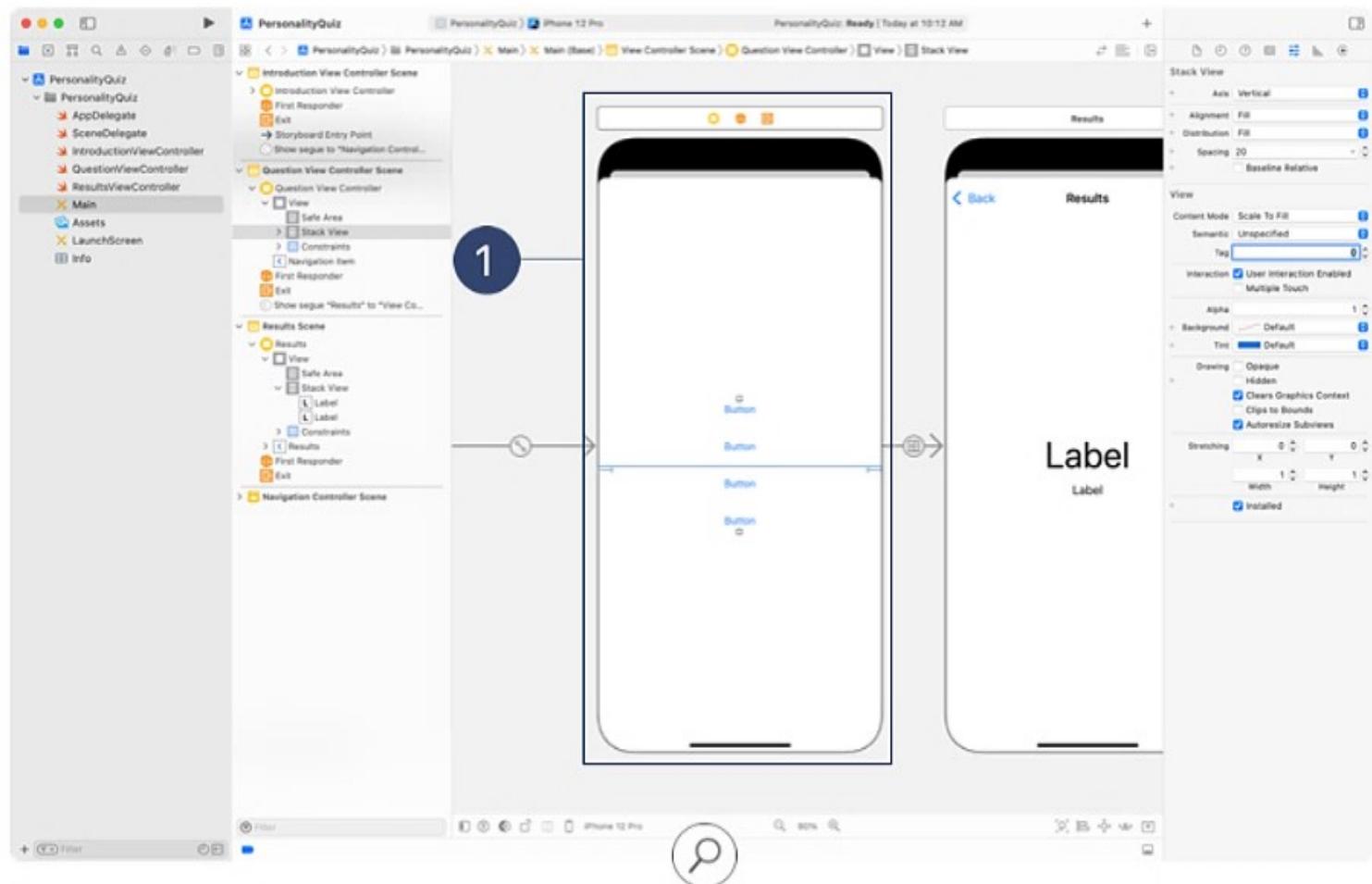
Create Questions And Answers

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.

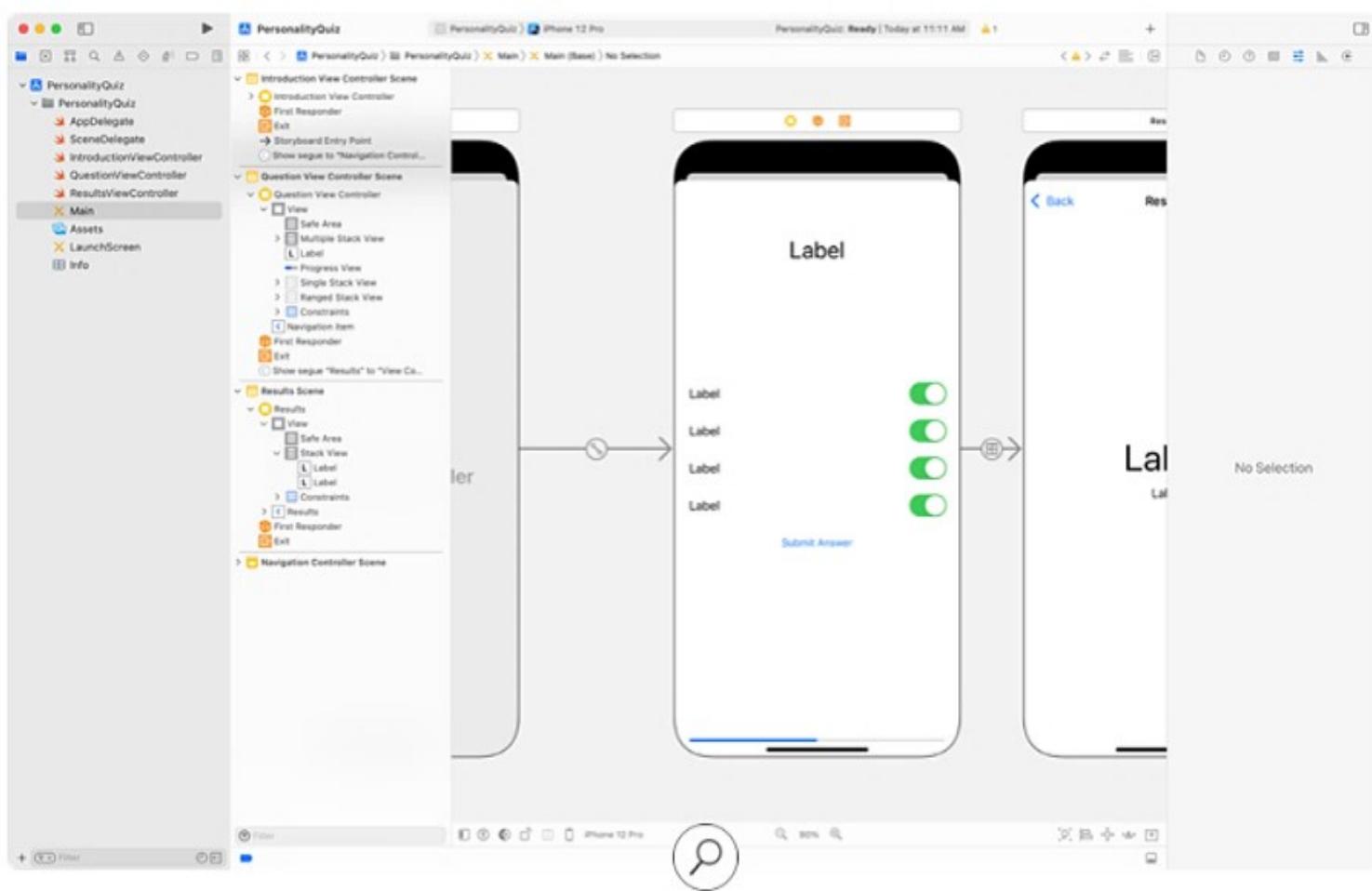
Begin by dragging a vertical stack view from the Object library to the QuestionViewController. Now add four buttons to the stack view. Use the Align tool to center the stack vertically within the view, then use the Add New Constraints tool to set its leading and trailing edges to 20 pixels. Add space 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

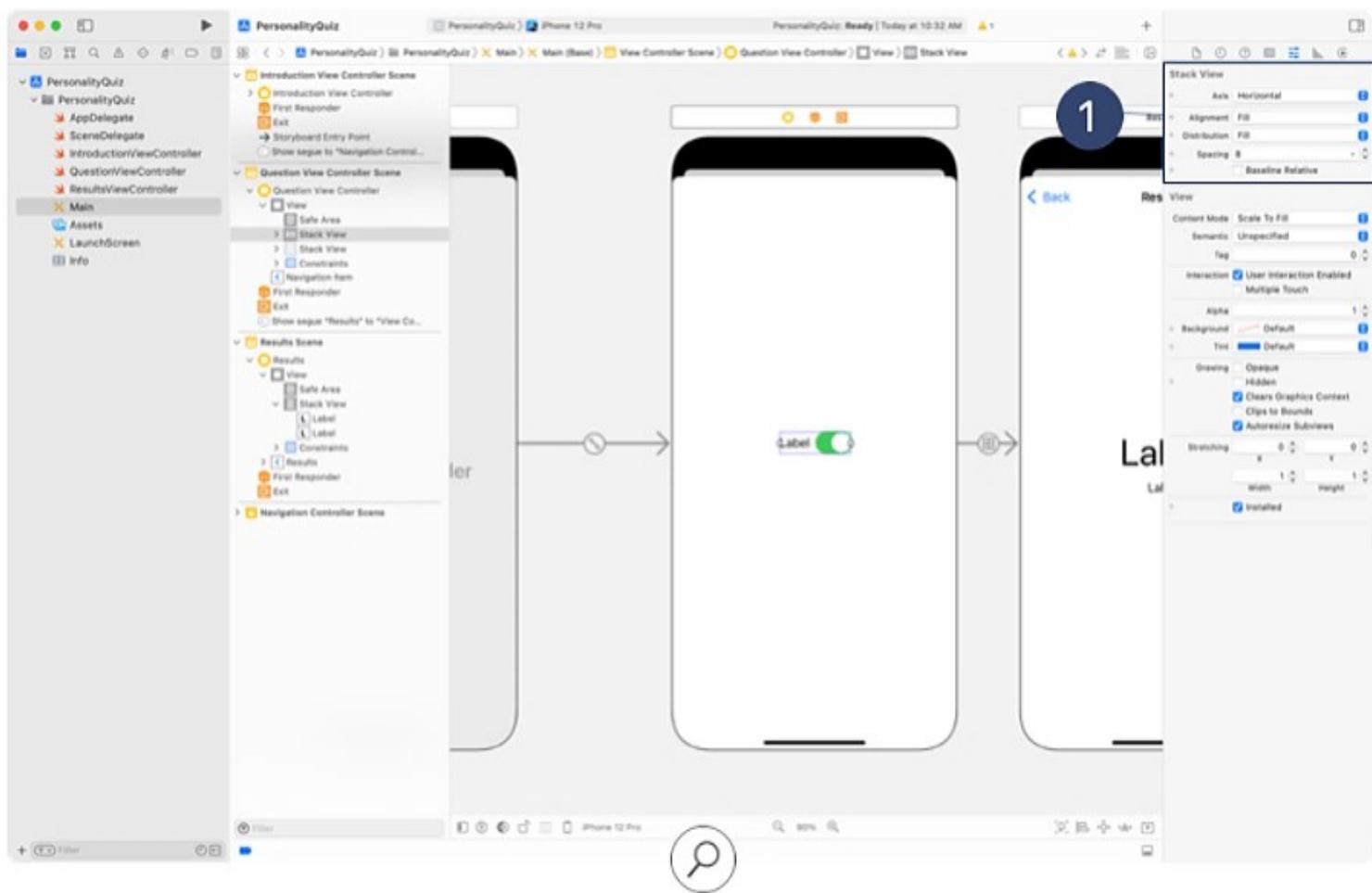
The question "Which of the following foods do you like?" suggests that the player can choose multiple answers. Rather than using buttons for the answers, it would make more sense to create pairs of labels and switches—so the player can switch on all positive answers. When the player has made their selections,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



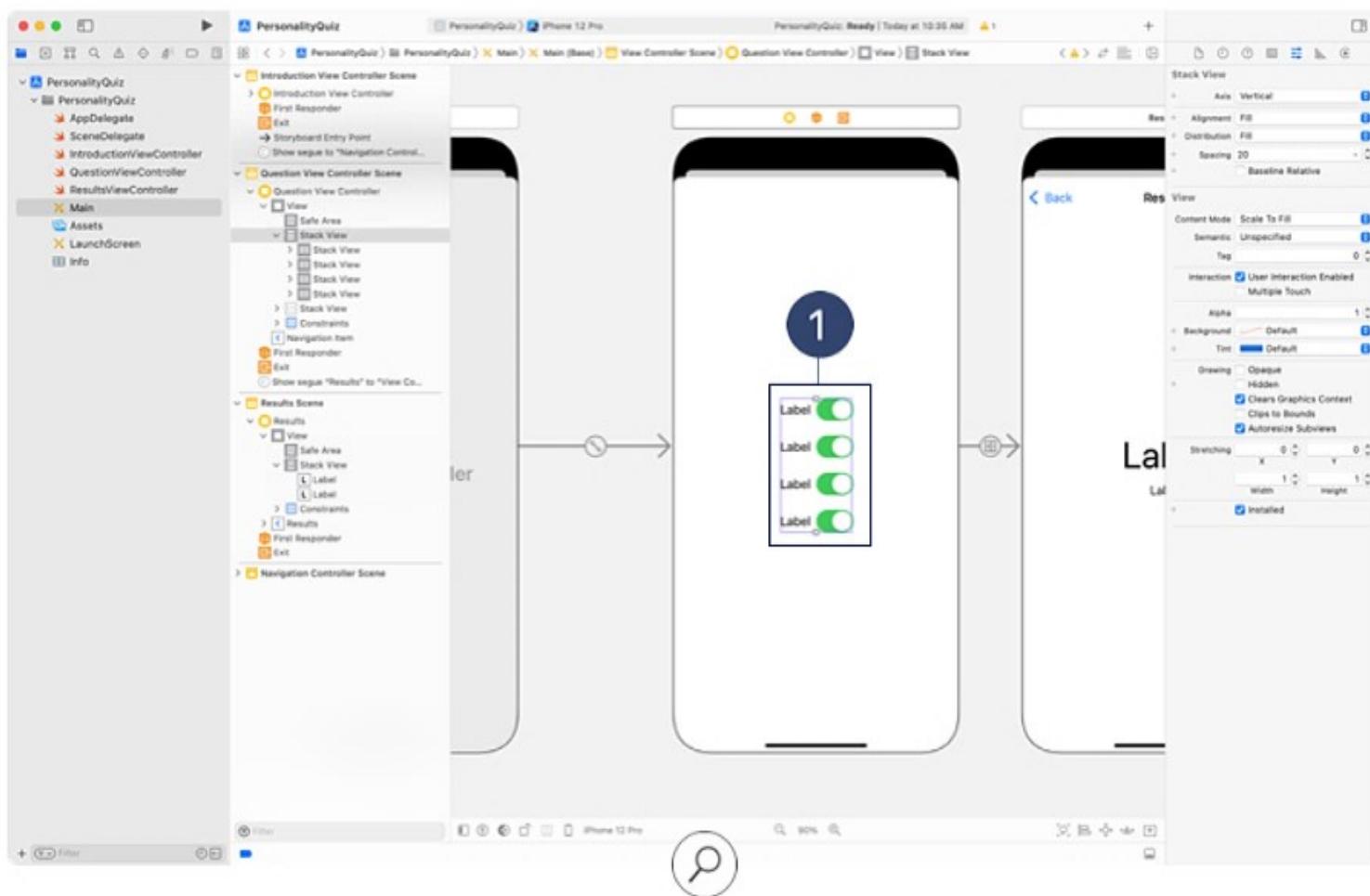
Before you begin, you might want to hide the single-answer stack view. Select the stack in the storyboard or the outline, then open the Attributes inspector, and deselect Installed at the bottom of the pane.

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.

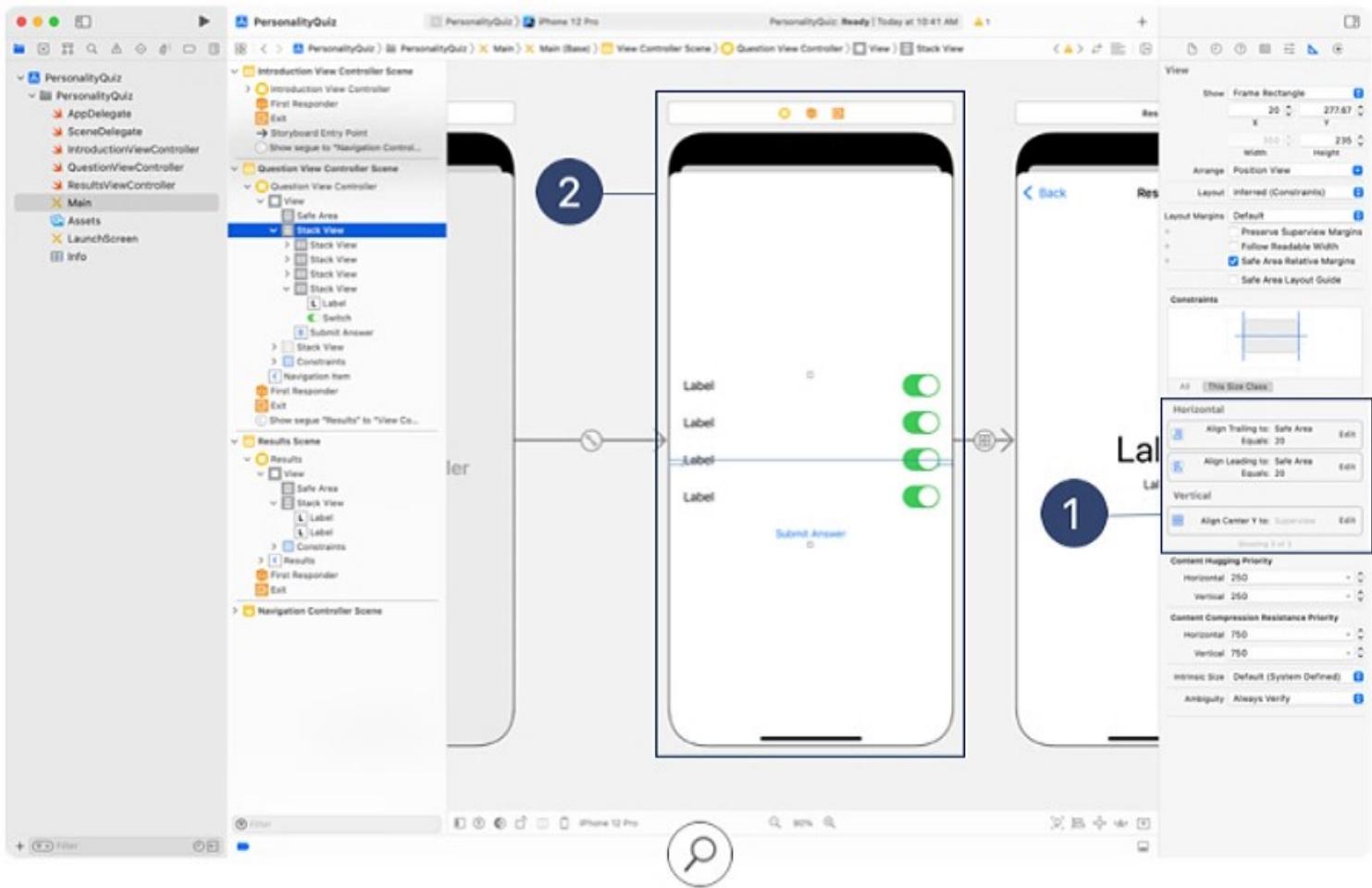
Begin by adding a label and a switch from the Object library. Highlight both of them, then click the Embed In button and choose Stack View. In the Attributes inspector for the stack view, make sure that the Axis is set to Horizontal and that Alignment and Distribution are both set to Fill. ①



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

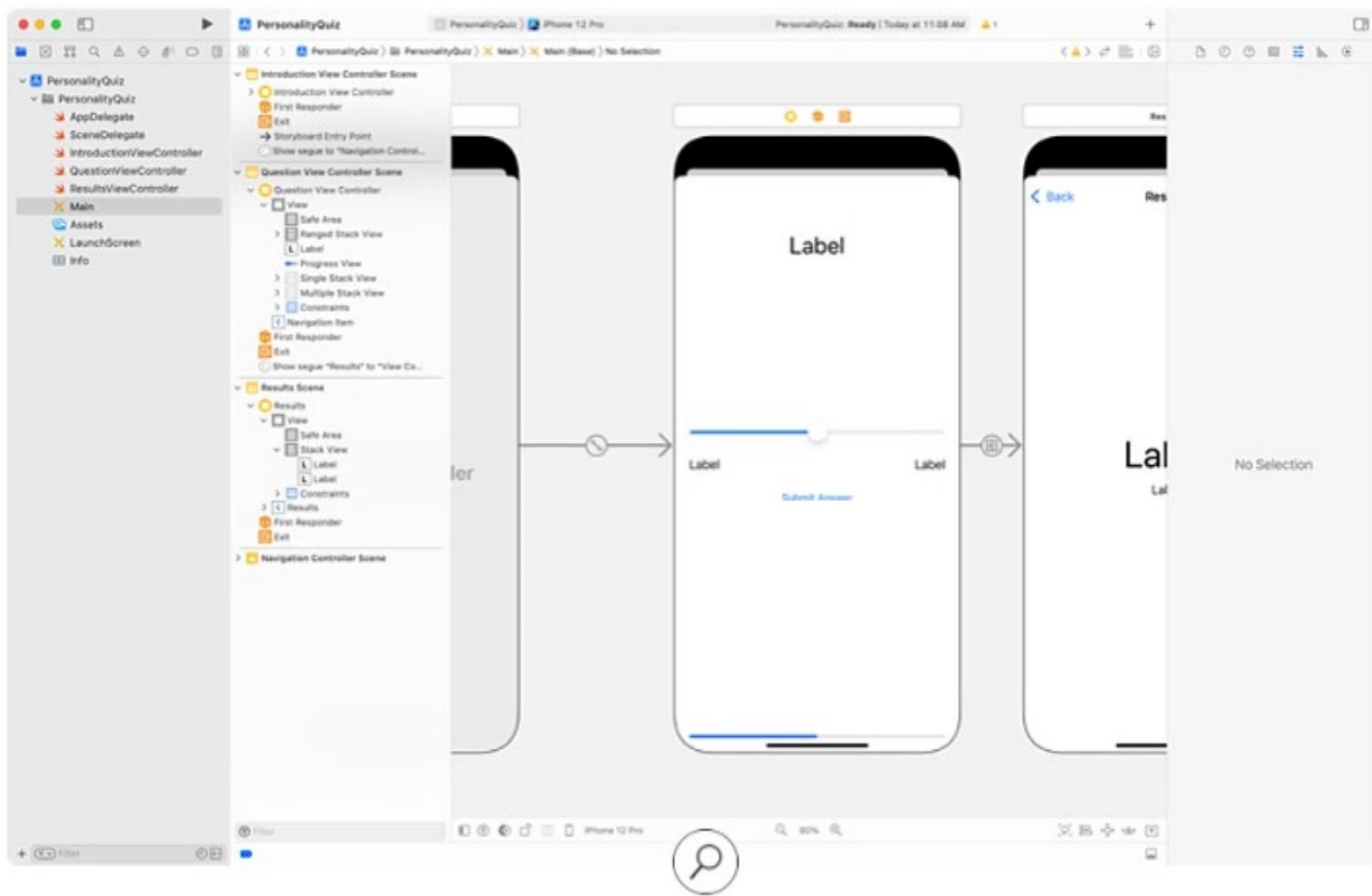


Add a button to the bottom of the stack view, and set its title to "Submit Answer." Finally, use the Align tool to center the stack vertically within the view, then use the Add New Constraints tool to set the leading and trailing edges to 20 pixels from each margin.^① If necessary, use the Update Frames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



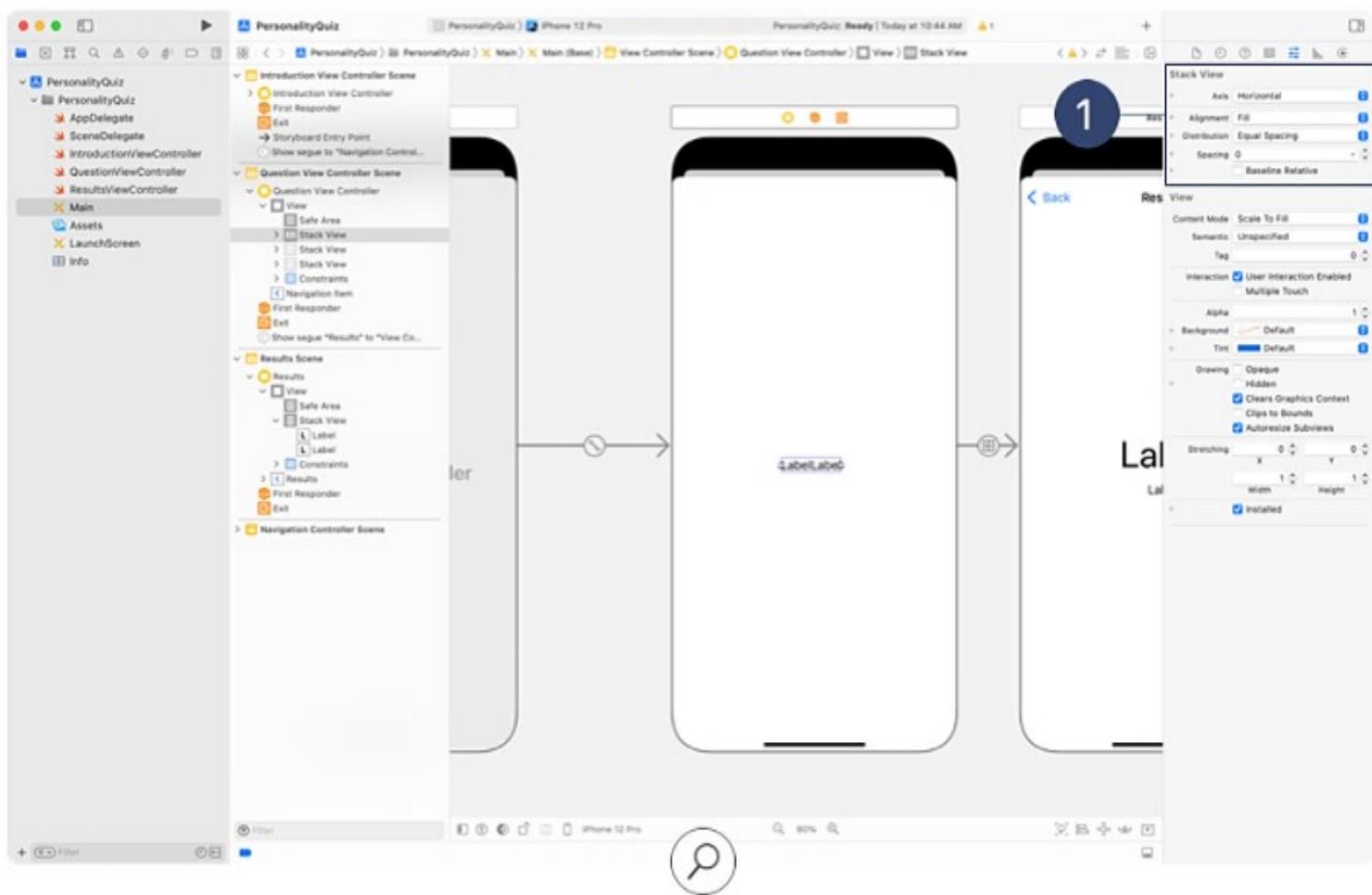
Range Questions

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.

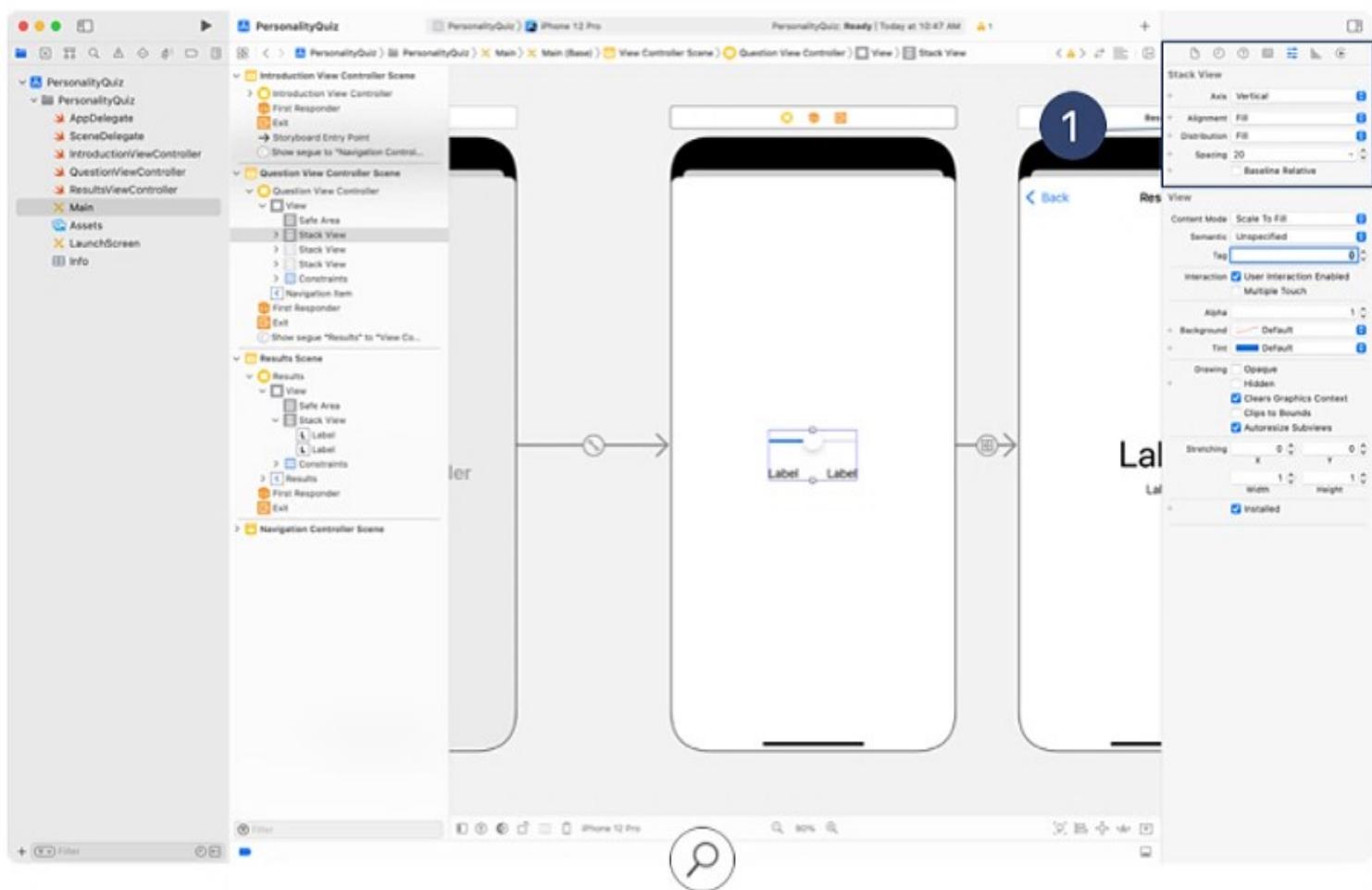


To make things easier, you might want to hide the multiple-answer stack, just as you did earlier with the single-answer stack. Select it in the storyboard, open the Attributes inspector, and deselect Installed at the bottom of the pane.

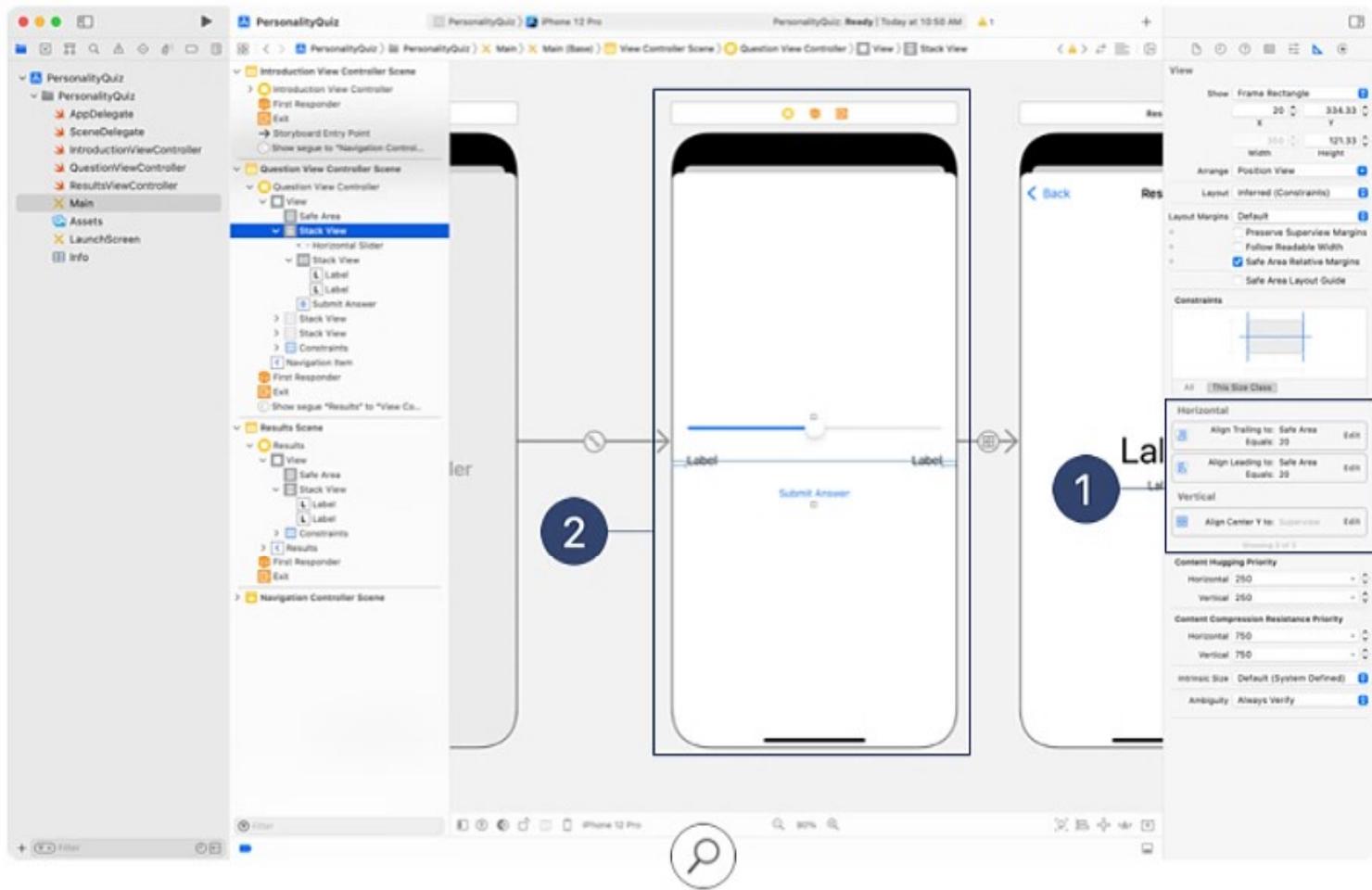
You can use stack views to create this interface without having to define very many constraints—similar to the switch approach. Begin by adding two labels to the canvas from the Object library, then select both, click the Embed In button and choose Stack View. In the Attributes inspector, check that 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



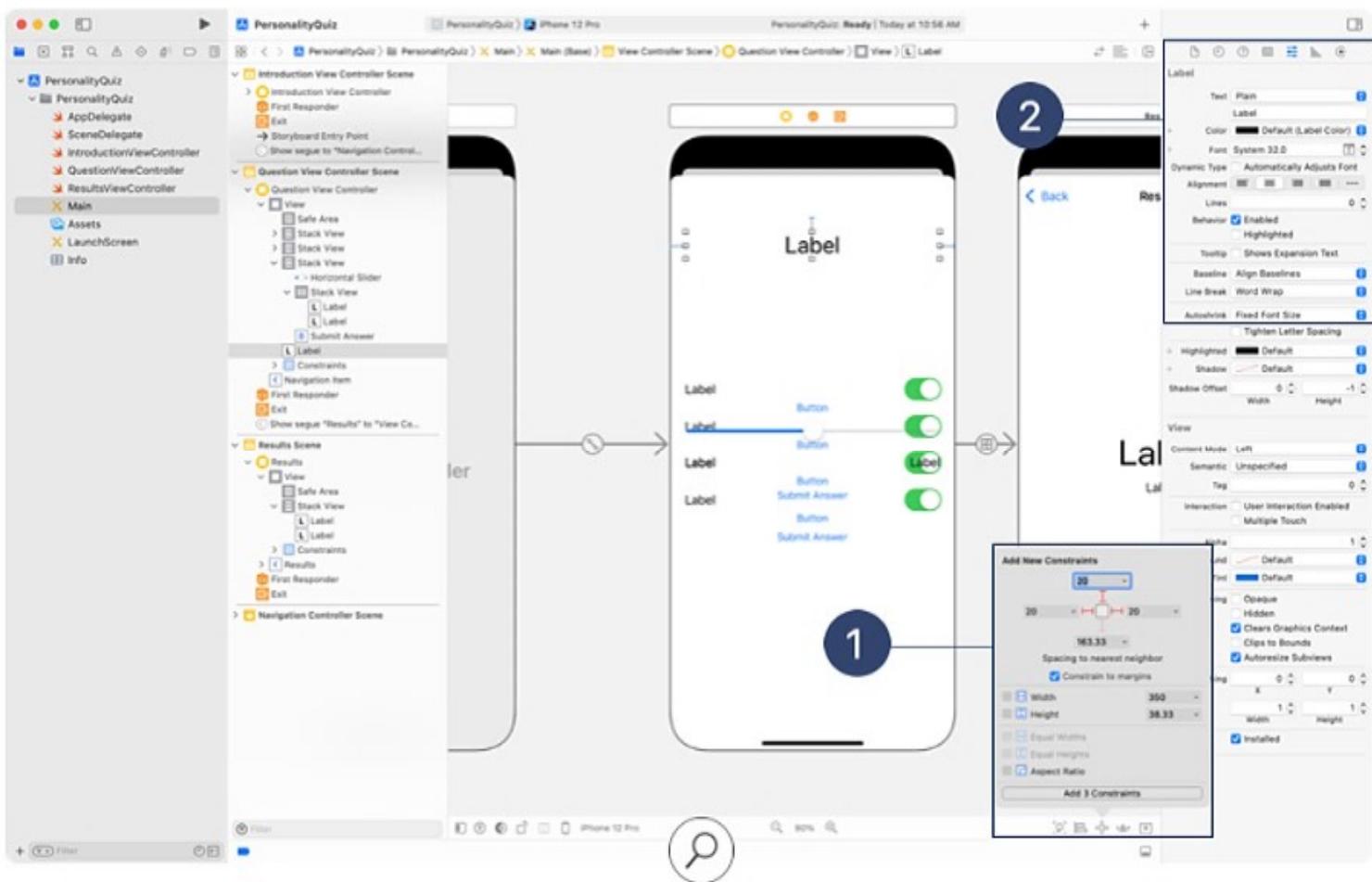
Add a button to the bottom of the stack, and set its title to “Submit Answer.” Use the Align tool to center the stack vertically within the view, then use the Add New Constraints tool to align the leading and trailing edges with 20 pixels of spacing to each margin. ^①If necessary, use the Update Frames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



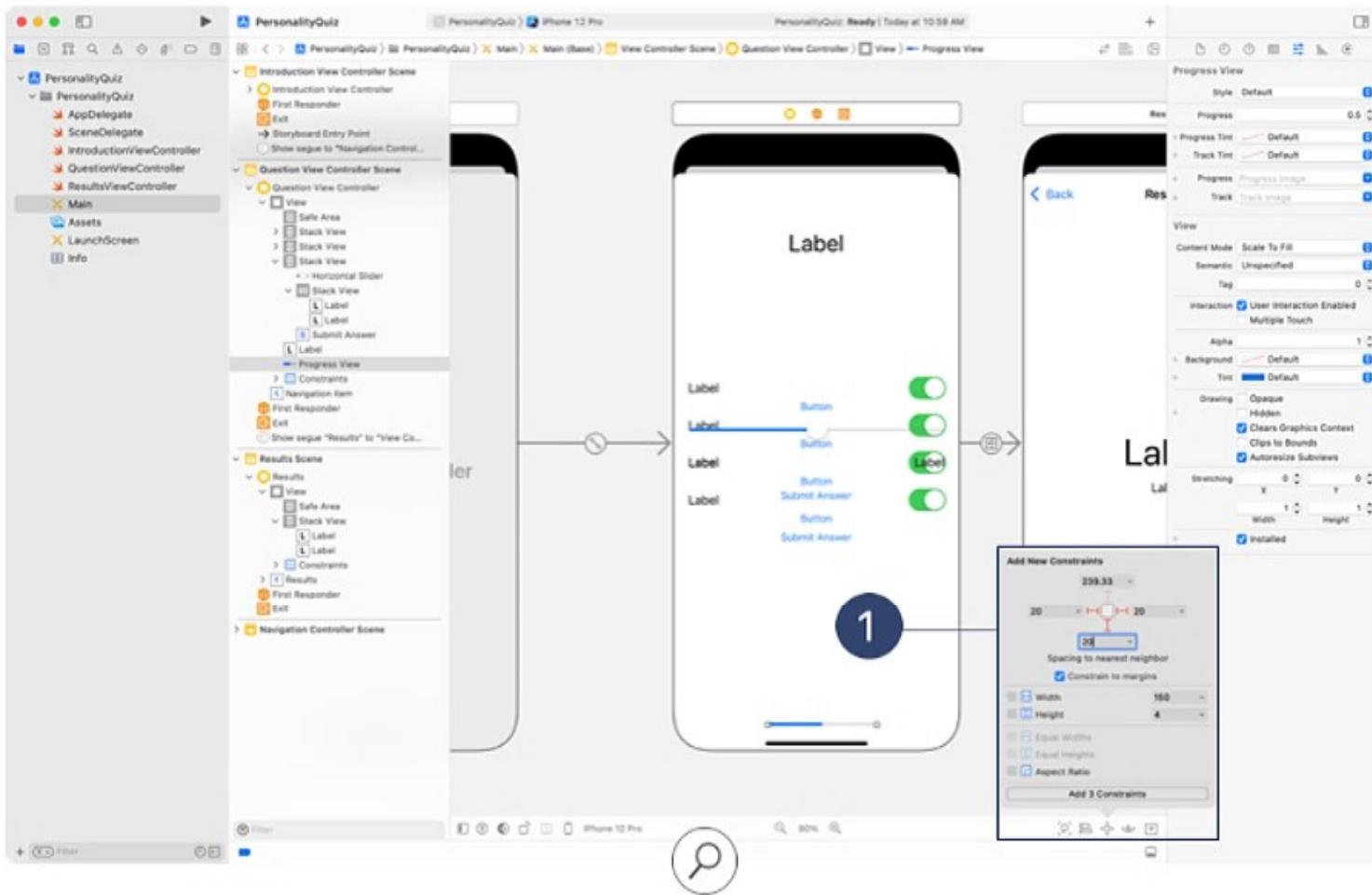
Before you move on, you’ll need to re-enable the stack views that you uninstalled during the building process. In the Document Outline, select each stack view, then select the Installed checkbox in the Attributes inspector.

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.



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
}
```

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 = "\ud83d\udc3f", cat = "\ud83d\udc3e", rabbit = "\ud83d\udc3d", turtle = "\ud83d\udc3a"

    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)
        ]
    )
]
```

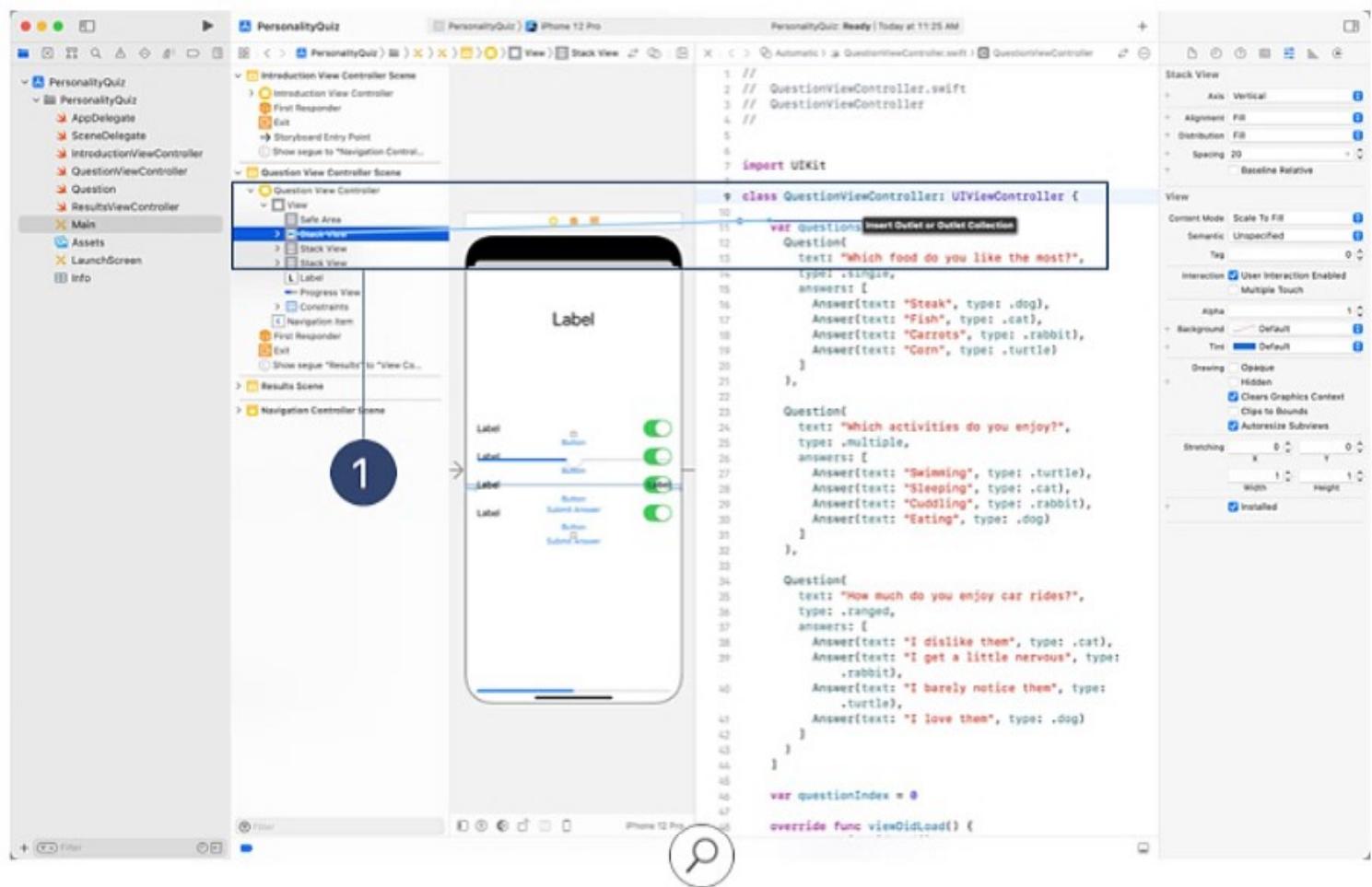
Display Questions with the Right Controls

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.



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, ① 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: UIPr
```

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.

Retrieve Answers with Actions

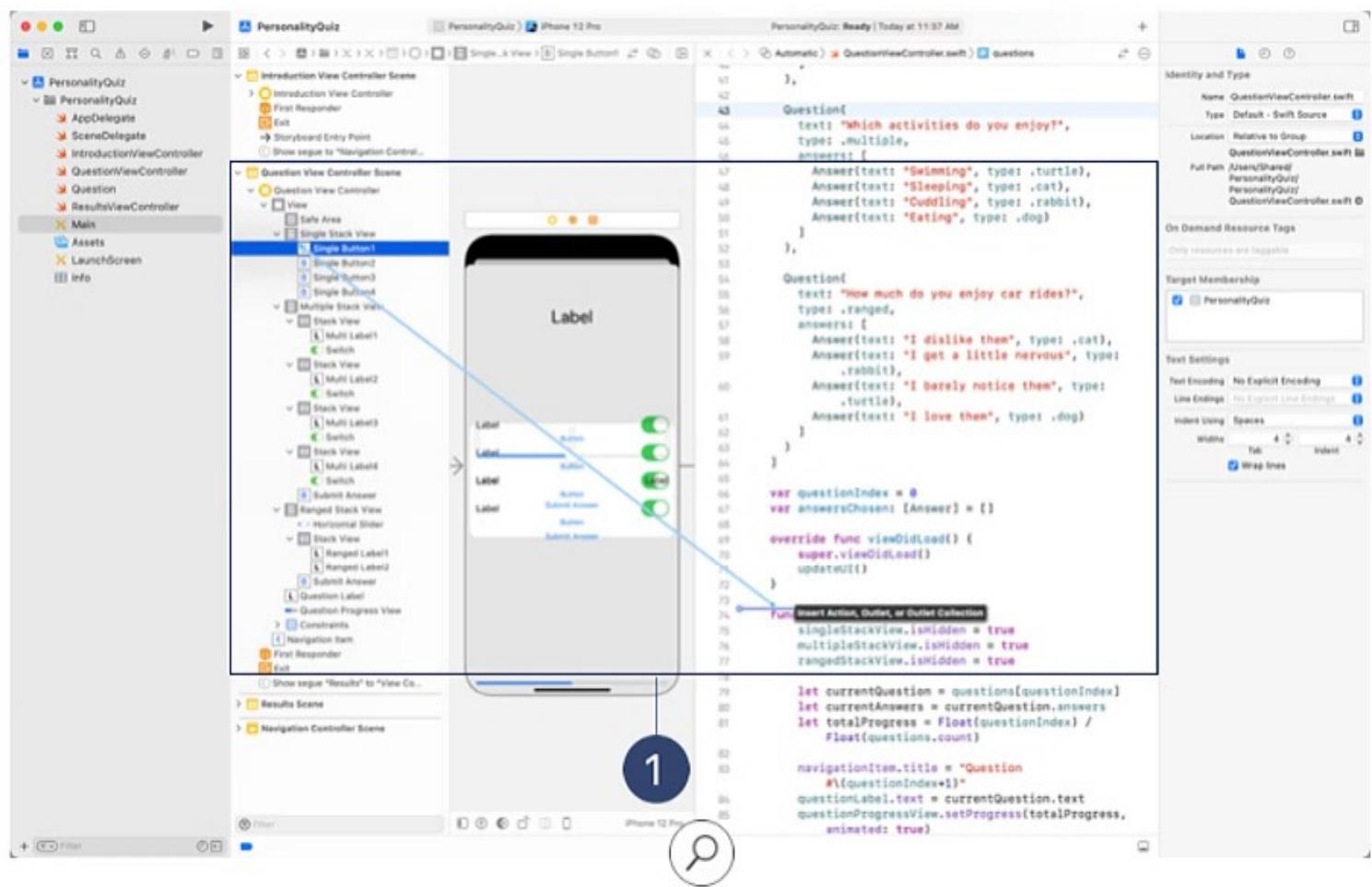
So far, you've generated a list of questions, and your app is displaying the correct user interface for the first question. That's an excellent start. In this section, you'll record the player's answers and move to the next question. When the player has answered every question in the collection, you'll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

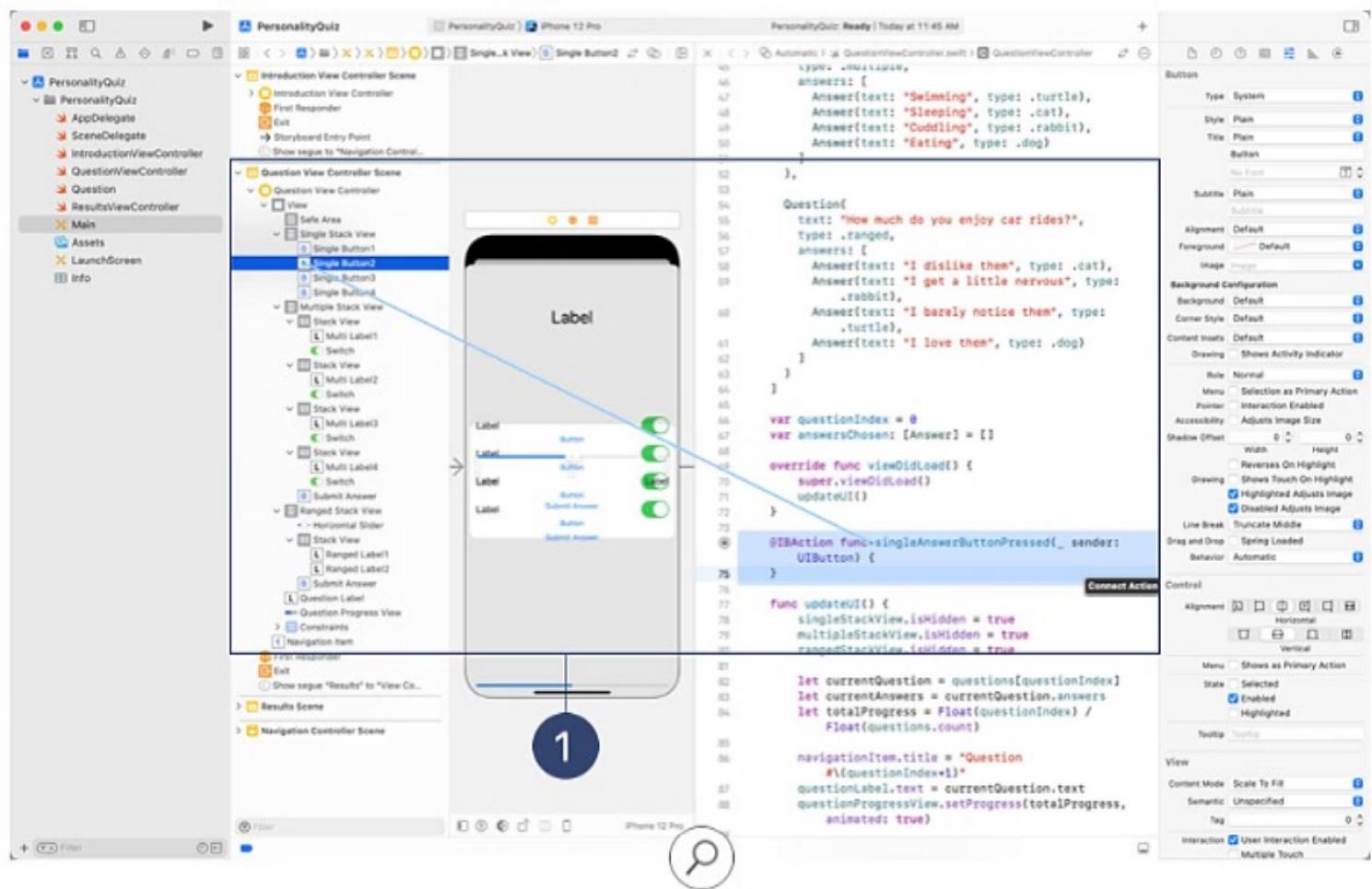
No matter which set of controls you use, you'll need to initialize an empty collection that can store the player's answers.

```
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`.





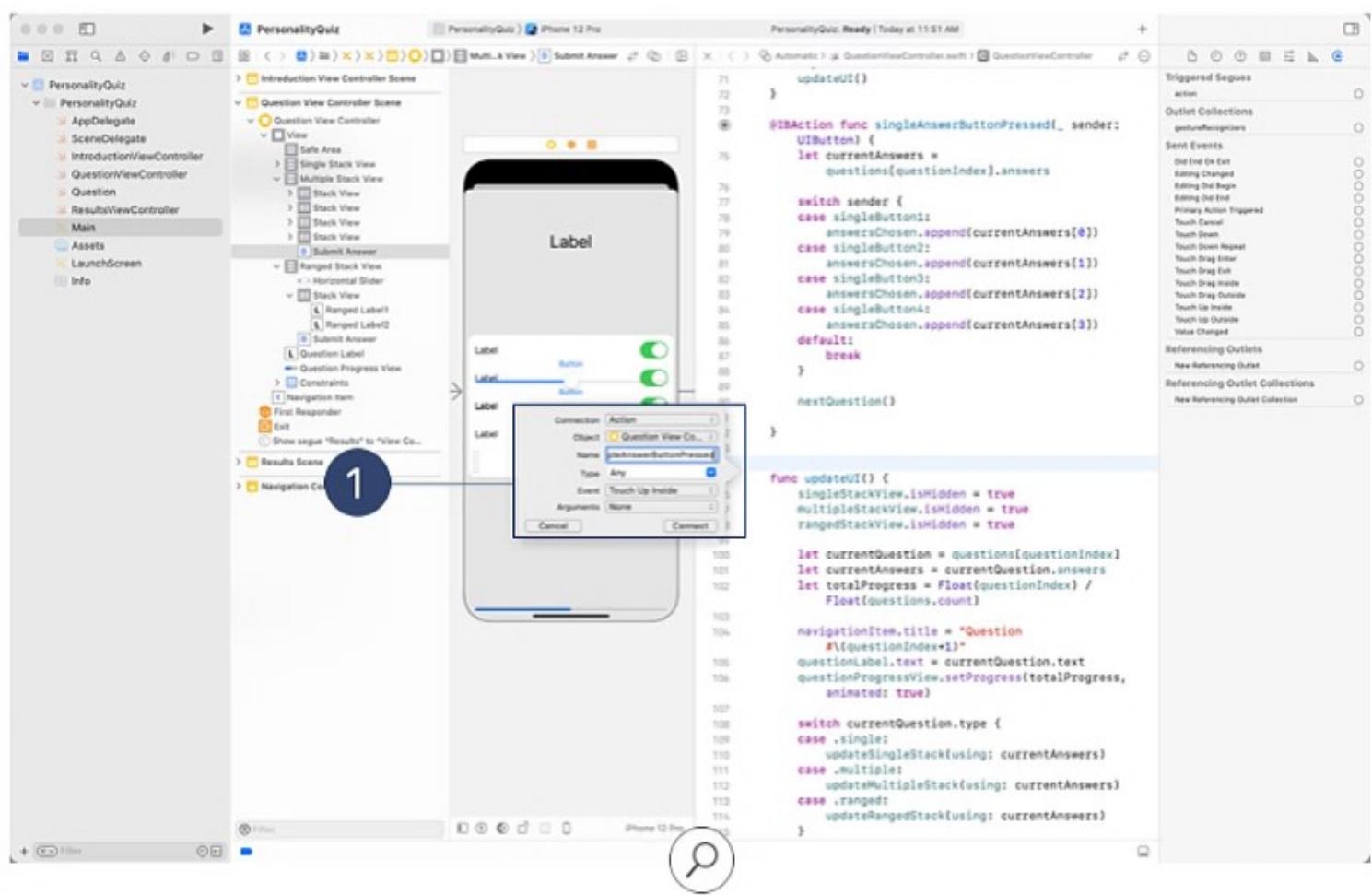
Connect the three remaining buttons in the single-answer stack view to this newly created `@IBAction`.¹

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.

```

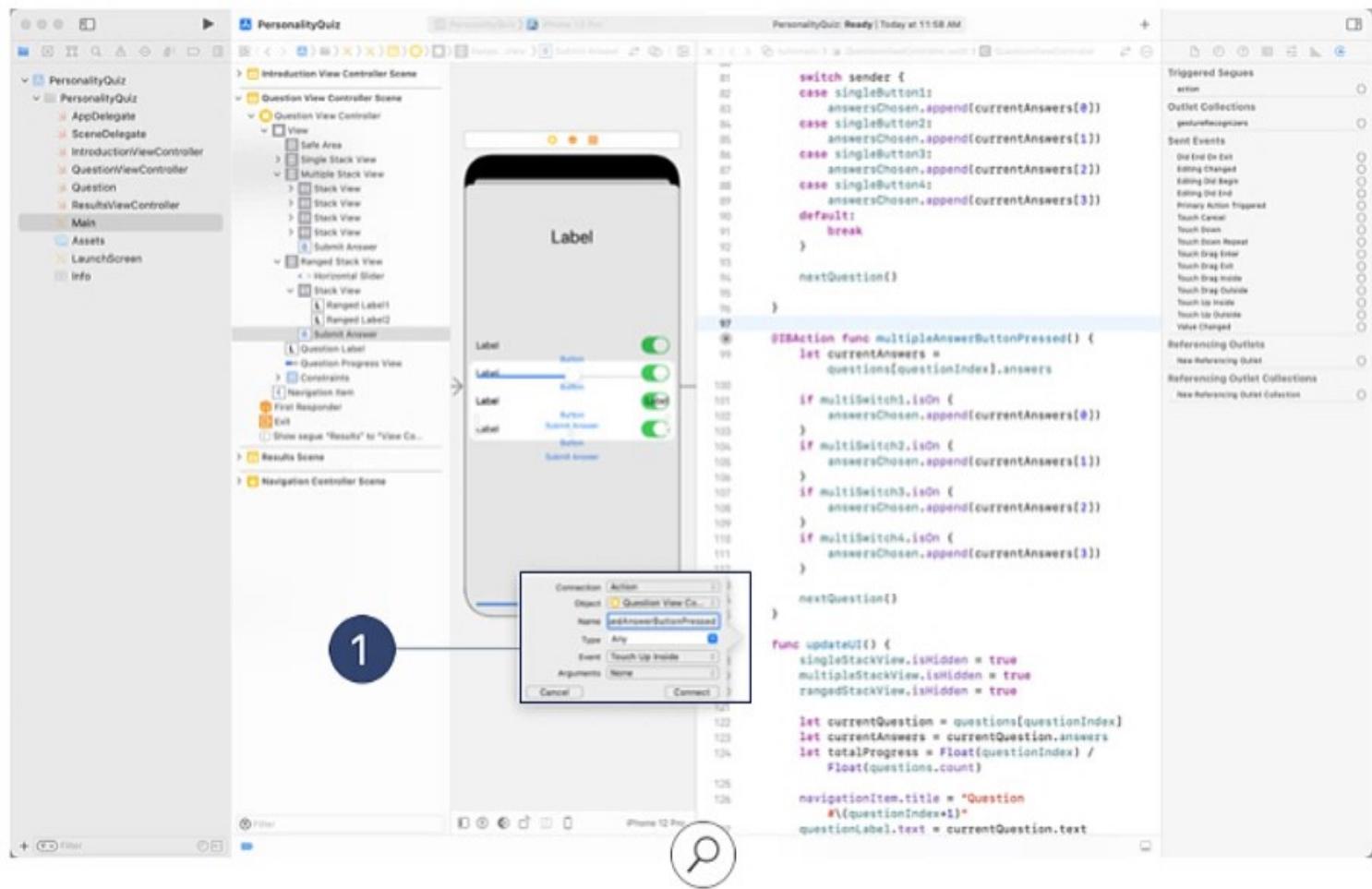
@IBOutlet var multiSwitch1: UISwitch!
@IBOutlet var multiSwitch2: UISwitch!
@IBOutlet var multiSwitch3: UISwitch!
@IBOutlet var multiSwitch4: UISwitch!

```

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

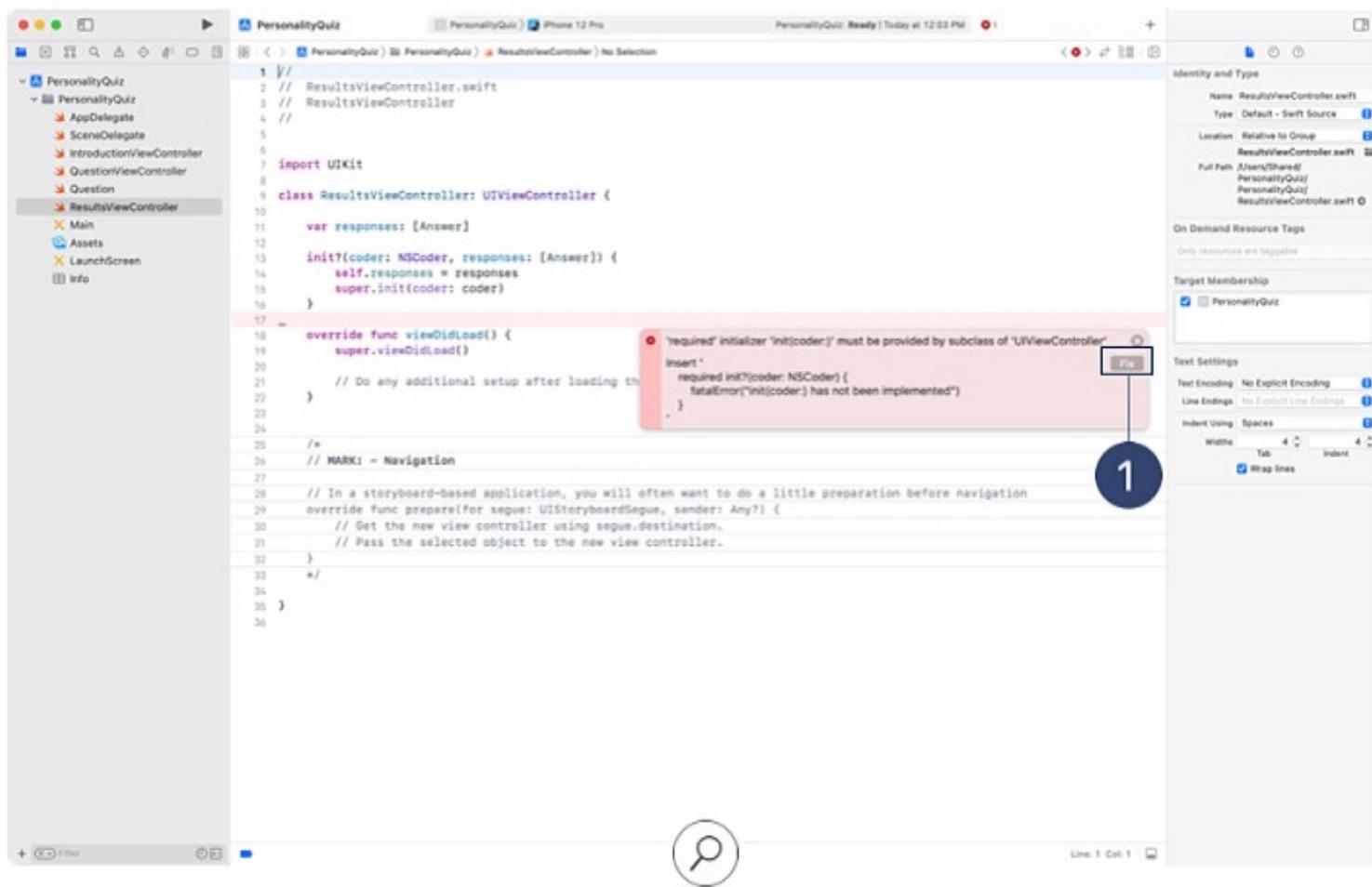
When it is time to show the results screen you will need to pass the user's responses to it. Open `ResultsController` and add a property to hold the user's responses.

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

This will cause Xcode to report a compilation error stating "Class 'ResultsController' has no initializers". This is because `responses` does not have an initial value nor is it even assigned a value in an initializer. To resolve this you will create a custom initializer for `ResultsController` 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. ①

```
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 `ResultsController`, 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.

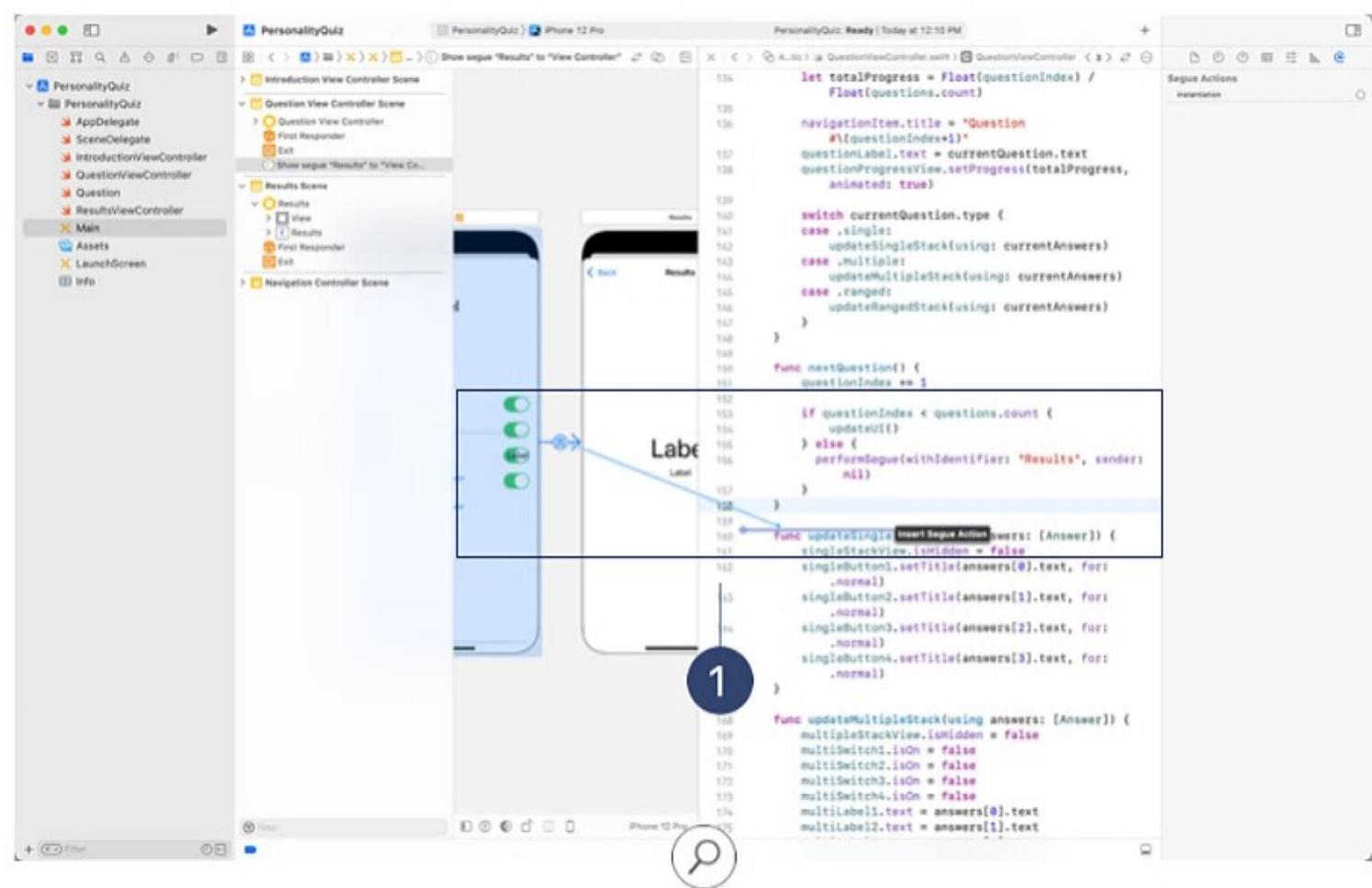
To resolve this problem, you can reset the positions of the switches and slider to logical defaults when the next question is displayed.

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

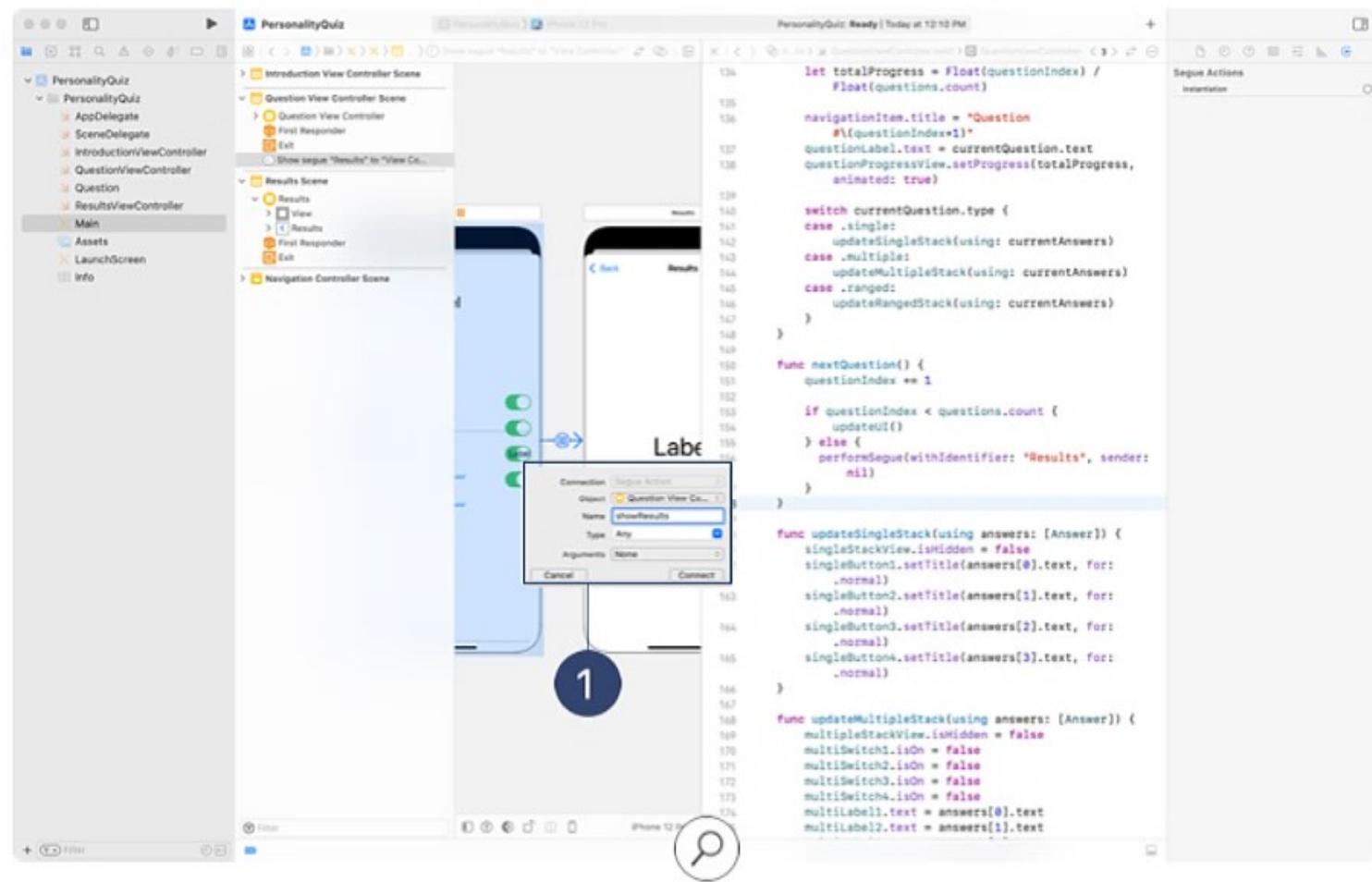
```
func updateMultipleStack(using answers: [Answer]) {  
    multipleStackView.isHidden = false  
    multiSwitch1.isOn = false  
    multiSwitch2.isOn = false  
    multiSwitch3.isOn = false  
    multiSwitch4.isOn = false  
    multiLabel1.text = answers[0].text  
    multiLabel2.text = answers[1].text  
    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 `ResultsController`. 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 `ResultsController`. It's your job within the implementation to initialize and return one. Using the custom initializer you created earlier, return a new instance of `ResultsController` 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 `ResultsController` 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  
}
```

To begin, within `ResultsController` 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](#).)

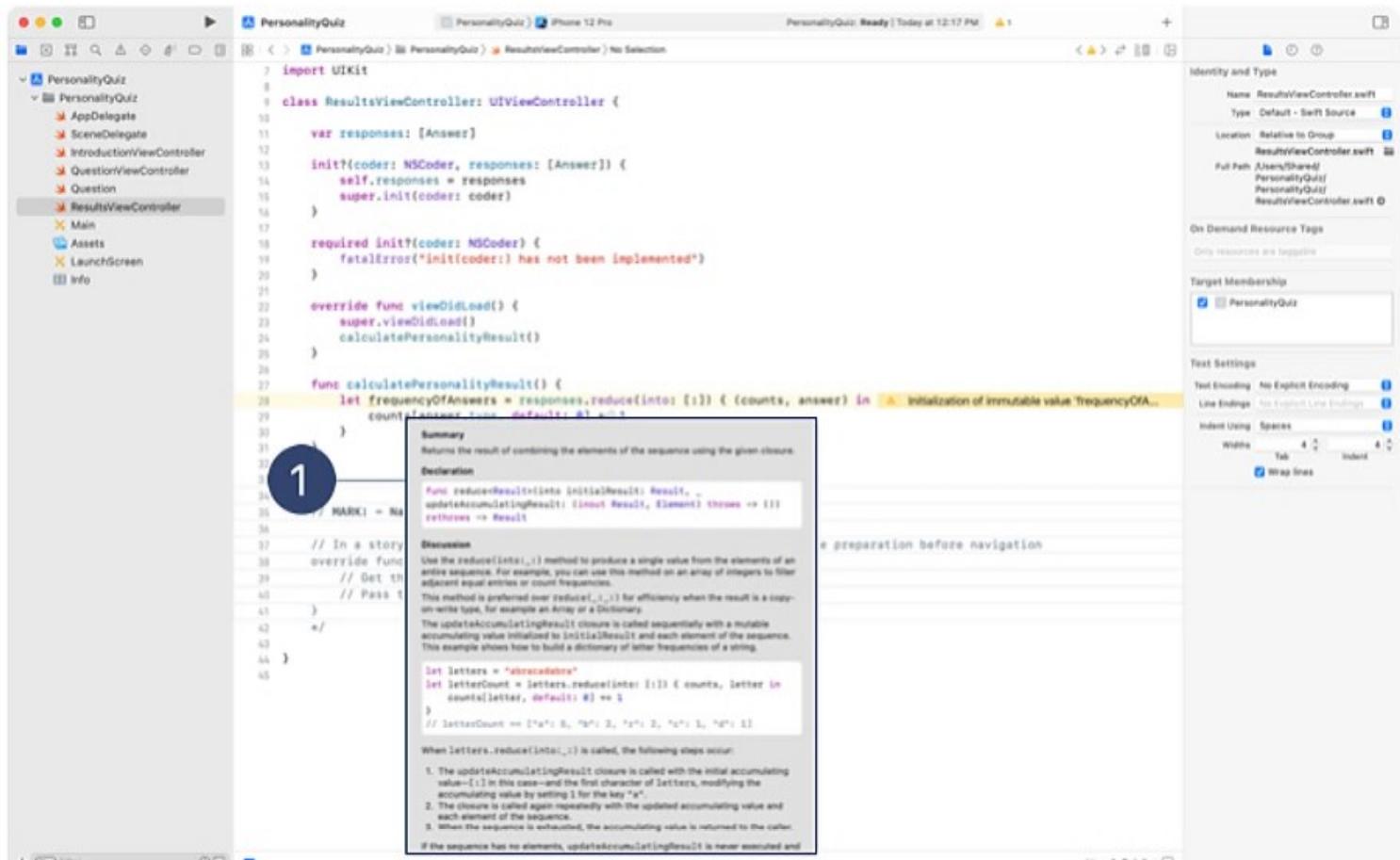
In this case, you're reducing `responses` into a dictionary that's initially empty—the argument passed for `into:`—and then incrementing the value in the dictionary that corresponds to the key for each response. Notice the `default:` parameter in the subscript to `counts`. This is a way to guard against 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.¹ 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 `ResultsController` 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 `ResultsController` 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 `ResultsController`:

```
navigationItem.hidesBackButton = true
```

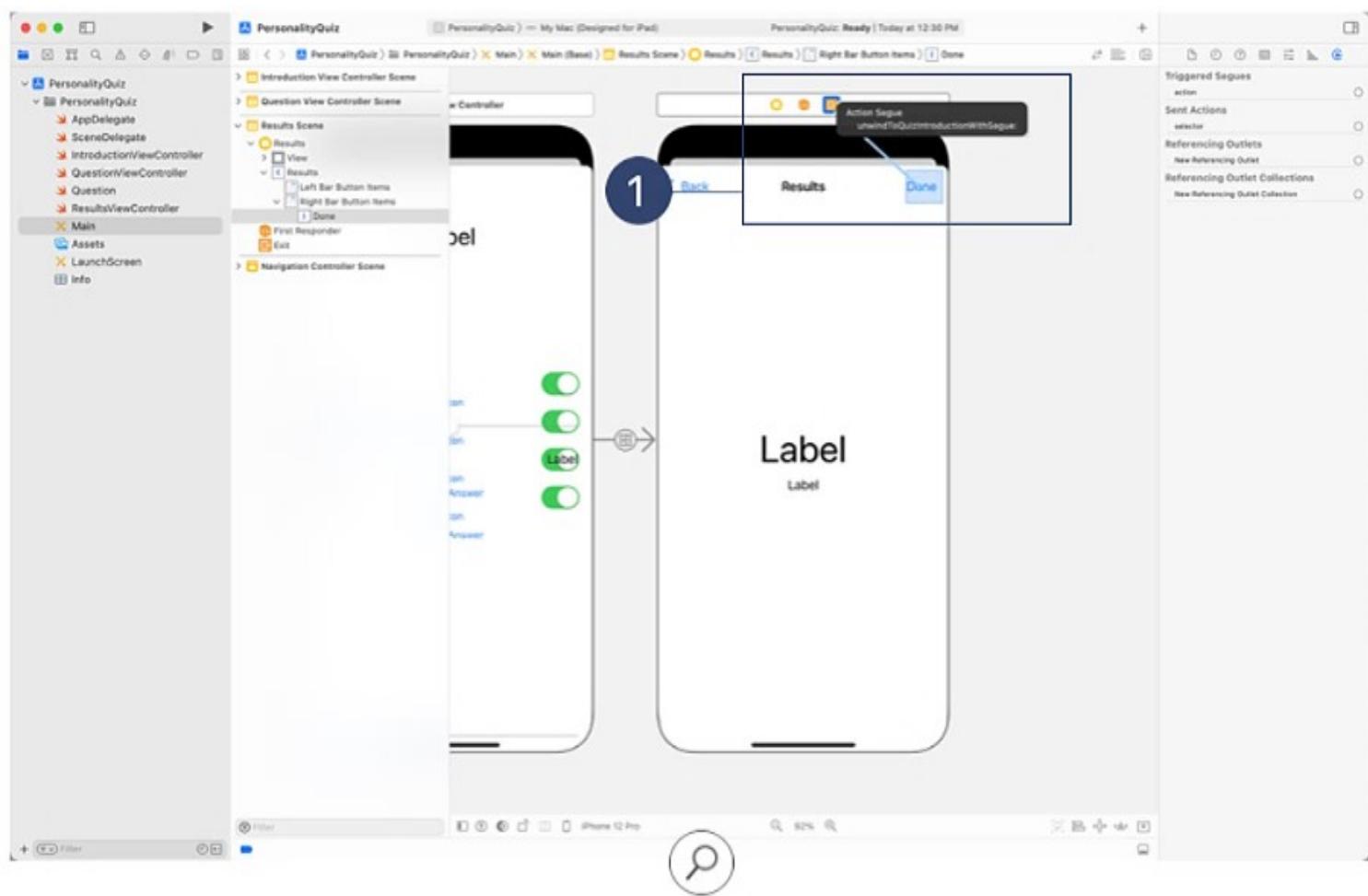
Instead of changing previous responses, the player should be able to dismiss the results and start with a clean slate. A tap of the Done button can return to the `IntroductionViewController`, making it very clear that the quiz is over. But at the moment, the Done button doesn't connect to any sort of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

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.

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](#).



© 2021 Apple Inc. All rights reserved. Apple, the Apple logo, Apple Books, Apple TV, Apple Watch, Cocoa, Cocoa Touch, Finder, Handoff, HealthKit, iPad, iPad Pro, iPhone, iPod touch, Keynote, Mac, macOS, Numbers, Objective-C, Pages, Photo Booth, Safari, Siri, Spotlight, Swift, tvOS, watchOS, and Xcode are trademarks of Apple Inc., registered in the U.S. and other countries. App Store and iBooks Store are service marks of Apple Inc., registered in the U.S. and other countries.

The Bluetooth® word mark and logos are registered trademarks owned by Bluetooth SIG, Inc. and any use of such marks by Apple is under license.

iOS is a trademark or registered trademark of Cisco in the U.S. and other countries and is used under license.

Other product and company names mentioned herein may be trademarks of their respective companies.