오후 10:23 7월 18일 토요일 65%

App Development with Swift

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 five projects, beginning with a simple flashlight app that changes the background color of the screen and ending with a shopping app that works with network services to communicate with a web server. After you finish the course projects, you'll have a chance to build your own personal project, working through design, prototyping, and development phases.

You can work through *App Development with Swift* 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 he or she may use different parts of the course in different ways.

This course was designed for students with no prior programming experience. But 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 App Development with Swift are five progressively challenging guided projects, each preceded by multiple lessons that cover the concepts and skills required to build the app. The course culminates in a personal project, which is creating an app of your own design.

About the Lessons

This course features 46 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 App Development with Swift covers two very different types of content—Swift and app development—you'll see two 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.

guiding you, keep in mind that he or she may use different parts of the course in different ways.

This course was designed for students with no prior programming experience. But 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 App Development with Swift are five progressively challenging guided projects, each preceded by multiple lessons that cover the concepts and skills required to build the app. The course culminates in a personal project, which is creating an app of your own design.

About the Lessons

This course features 46 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 App Development with Swift covers two very different types of content—Swift and app development—you'll see two 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.

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.

List

The fourth project is **List**, a task-tracking app that allows the user to add, edit, and delete items in a familiar table-based interface. You'll be able to customize the app to track any type of information, such as a card collection, homework assignments, or a playlist. You'll learn how to build scroll views, table views, and complex input screens. You'll also learn how to save data, share data to other apps, and work with images in the user's photo library.

Restaurant

The last project is **Restaurant**, a menu app that displays the available dishes from a restaurant and allows the user to submit an order. This project comes with an easy-to-use local web service that allows you to customize the entire menu with your own menu items and photos. You'll learn about animations, concurrency, and working with the web.

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

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 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 Mojave or High Sierra.
- Xcode 10. Xcode 10 is compatible with macOS Mojave and High Sierra. Download it from the Mac App Store, if you need it.
- Project files for the course which you can download using the link below.

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

Download student materials

A Word of Advice

App Development with Swift 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 hi 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 *App Development with Swift*. By learning the fundamentals of the Swift programming language, you'll be on the fast track to developing your own apps.

This first unit introduces you to the fundamentals 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.

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 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

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
- WWDC 2016 Introduction to 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 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, both 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 starting out with a new programming language, it's a time-honored tradition to build a "Hello, world!" app, the simplest program 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 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 Enter.
3. Type the command `print("Hello, world!")` and press Enter to execute it.

Note that "Hello, world!" was printed on screen 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 Enter to exit the Swift REPL.
2. Quit Terminal.

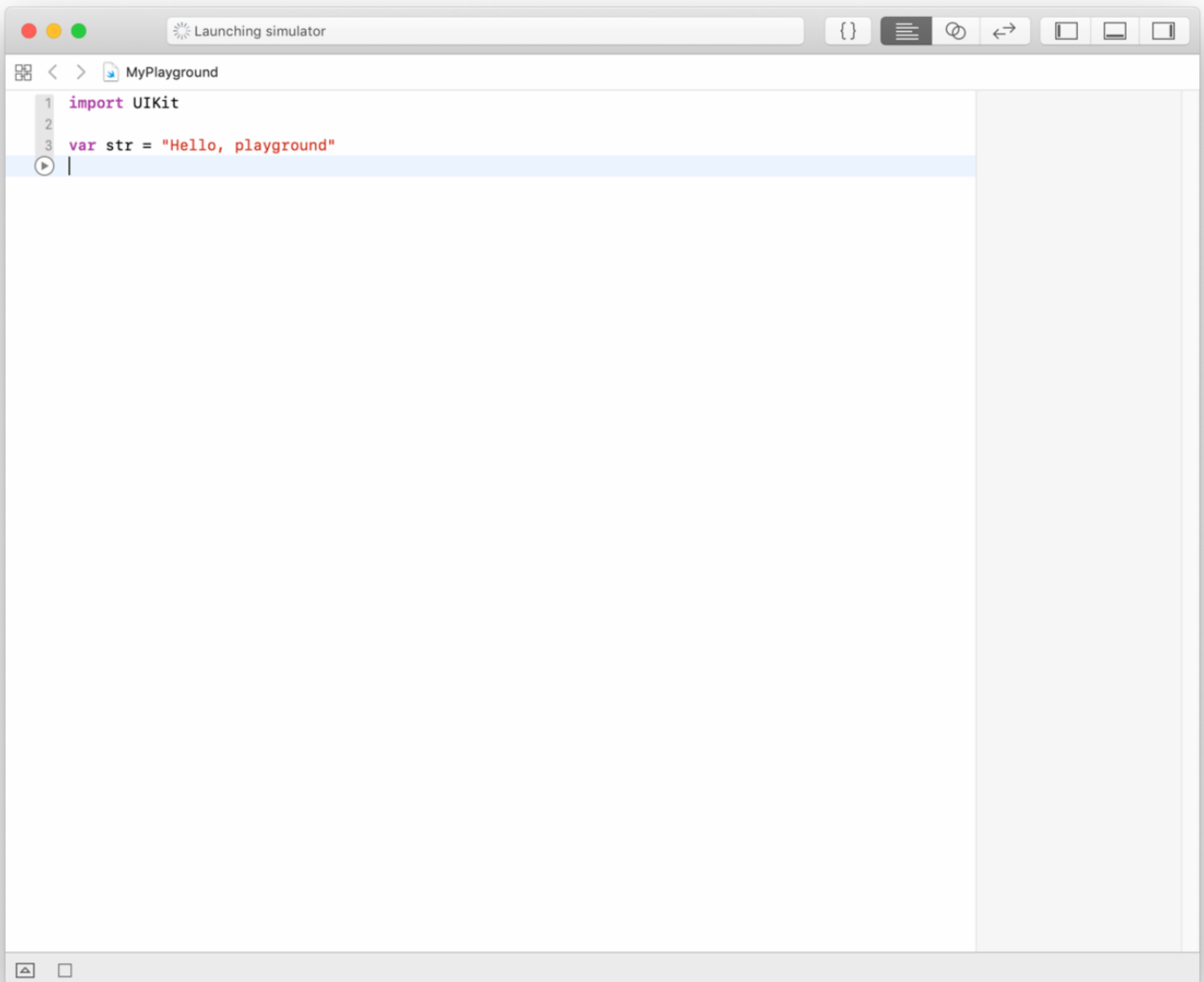
Playground

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.

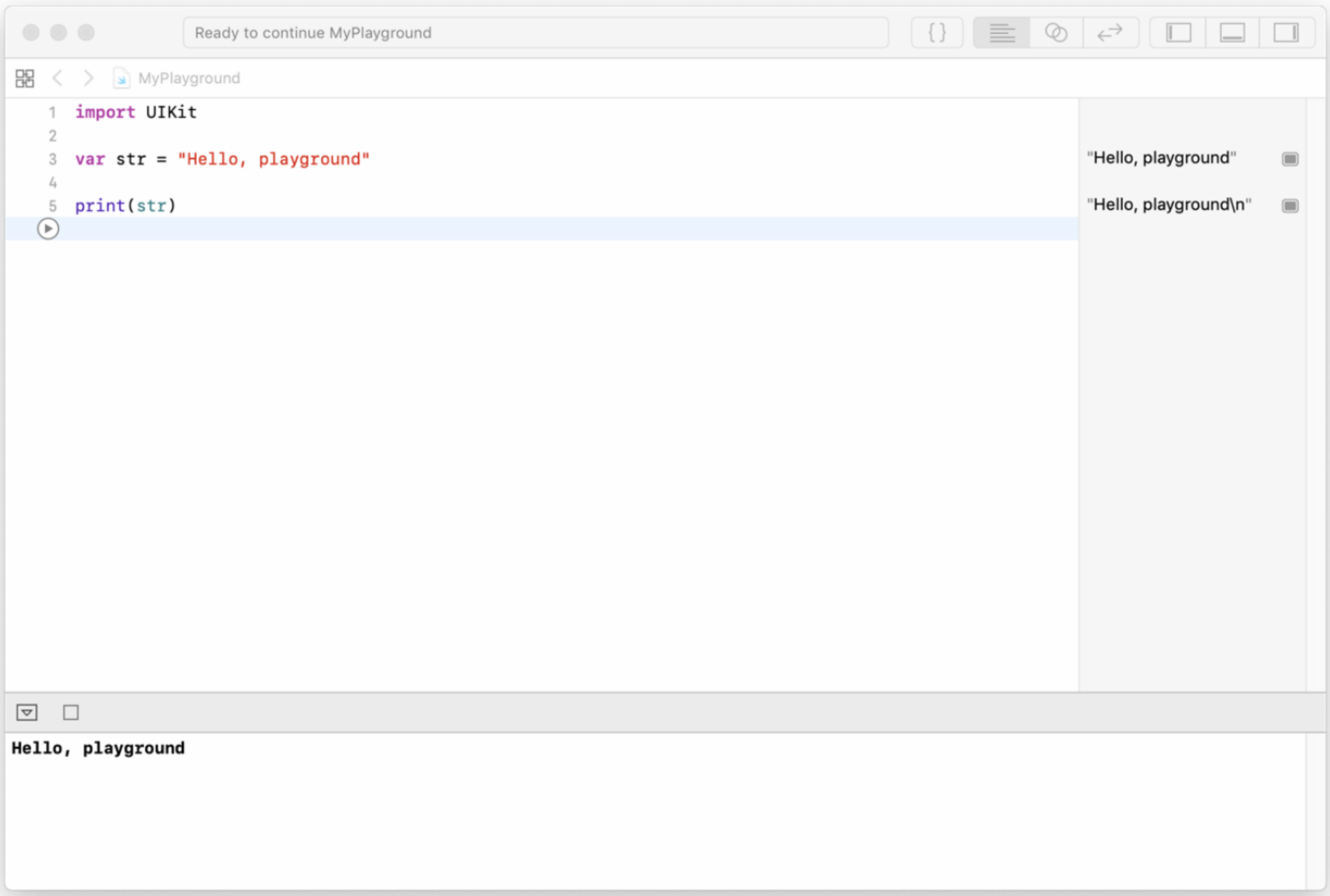
Working in playgrounds is preferable to working in a REPL for many reasons. For starters, they make writing Swift code fun and simple. Using a more familiar interface, you can type in a line of code and the results appear immediately in the sidebar. If your code runs over a period of time, you can watch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



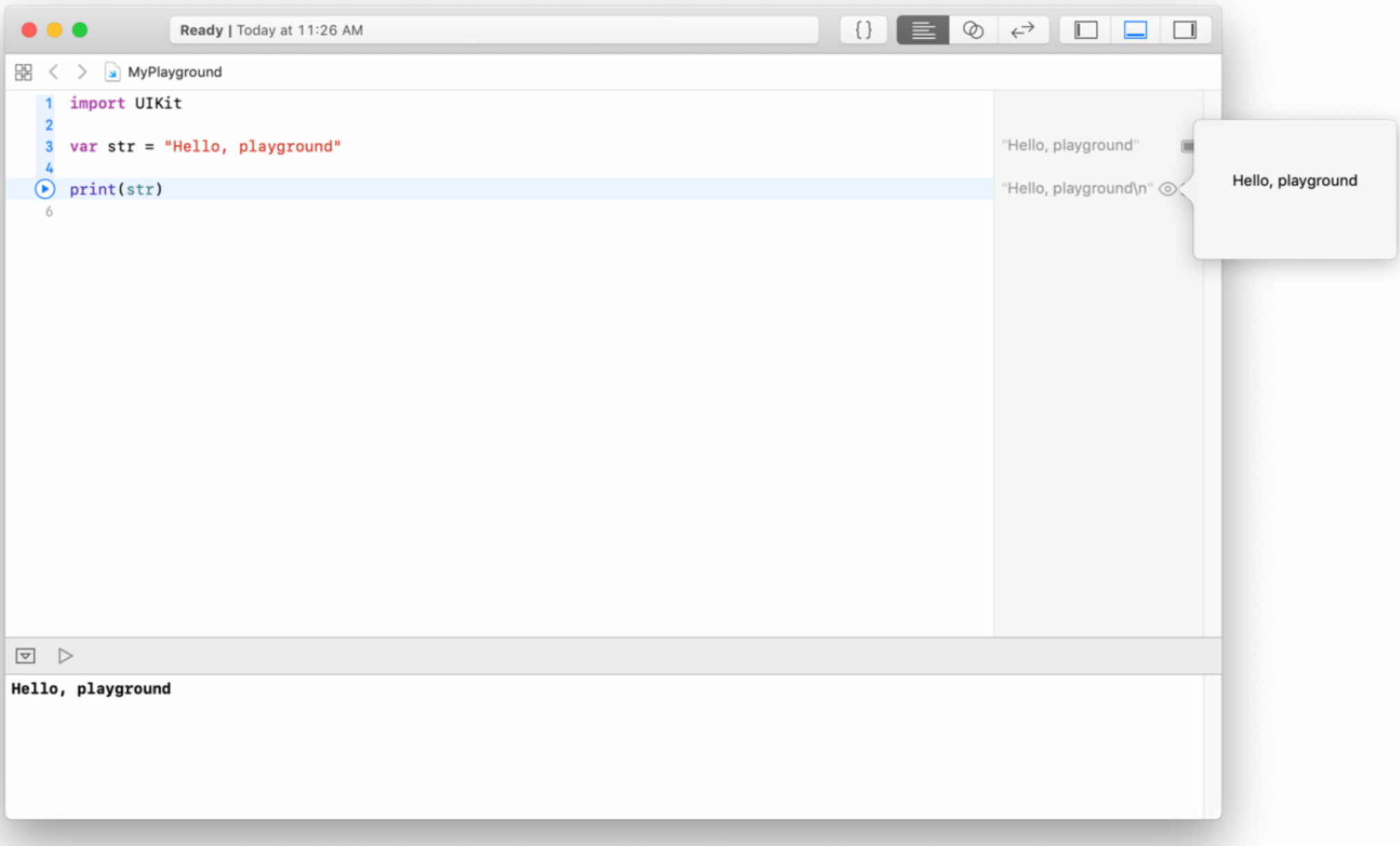
To view printed results from your Swift code, you can open a debug console area in a playground.



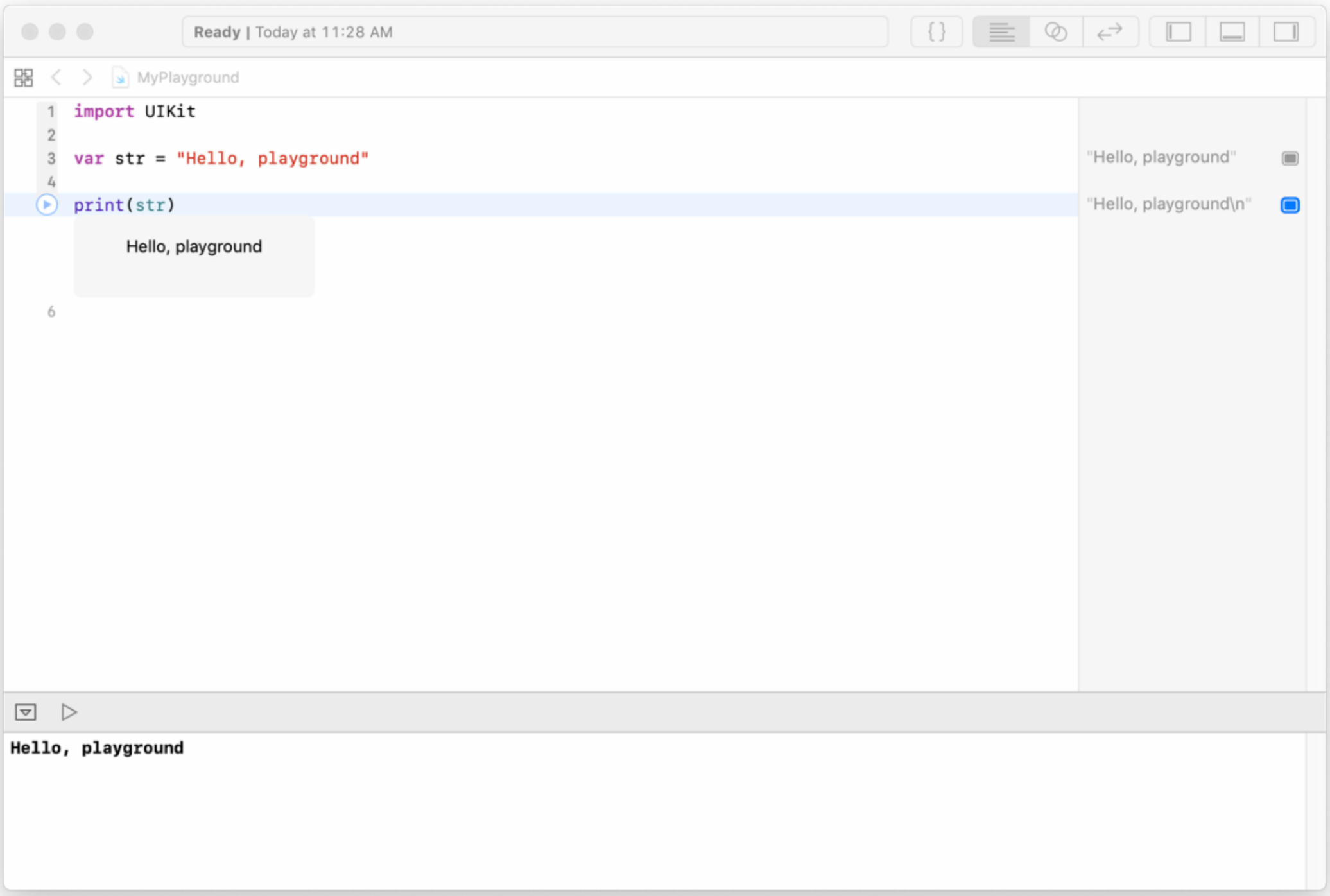
Remember, every time you edit the Swift code, the code is run from top to bottom. For every expression, results are shown on the right side.

오후 10:24 7월 18일 토요일 65%

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.



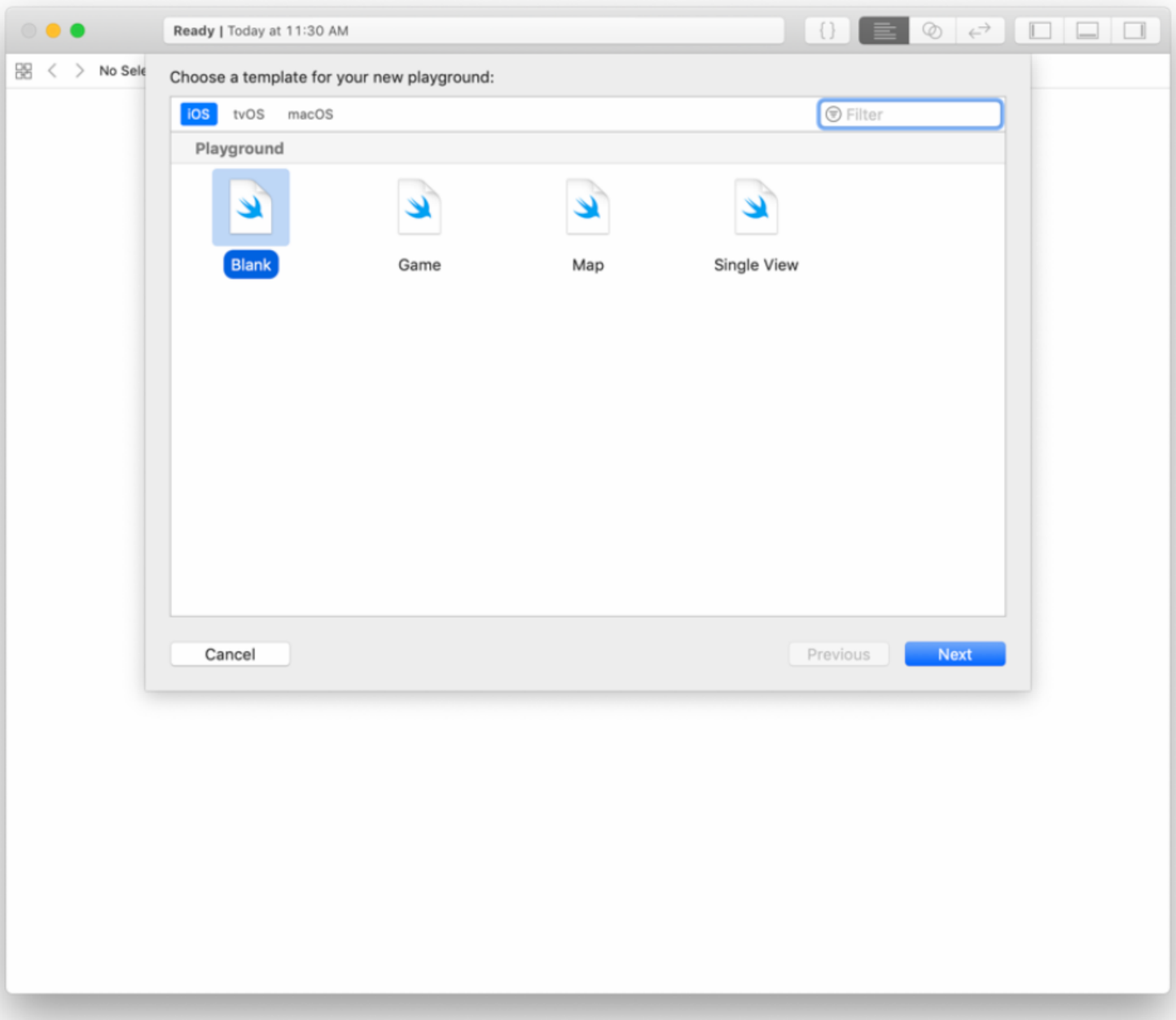
To view more details inline, you can use the Show Results button.



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

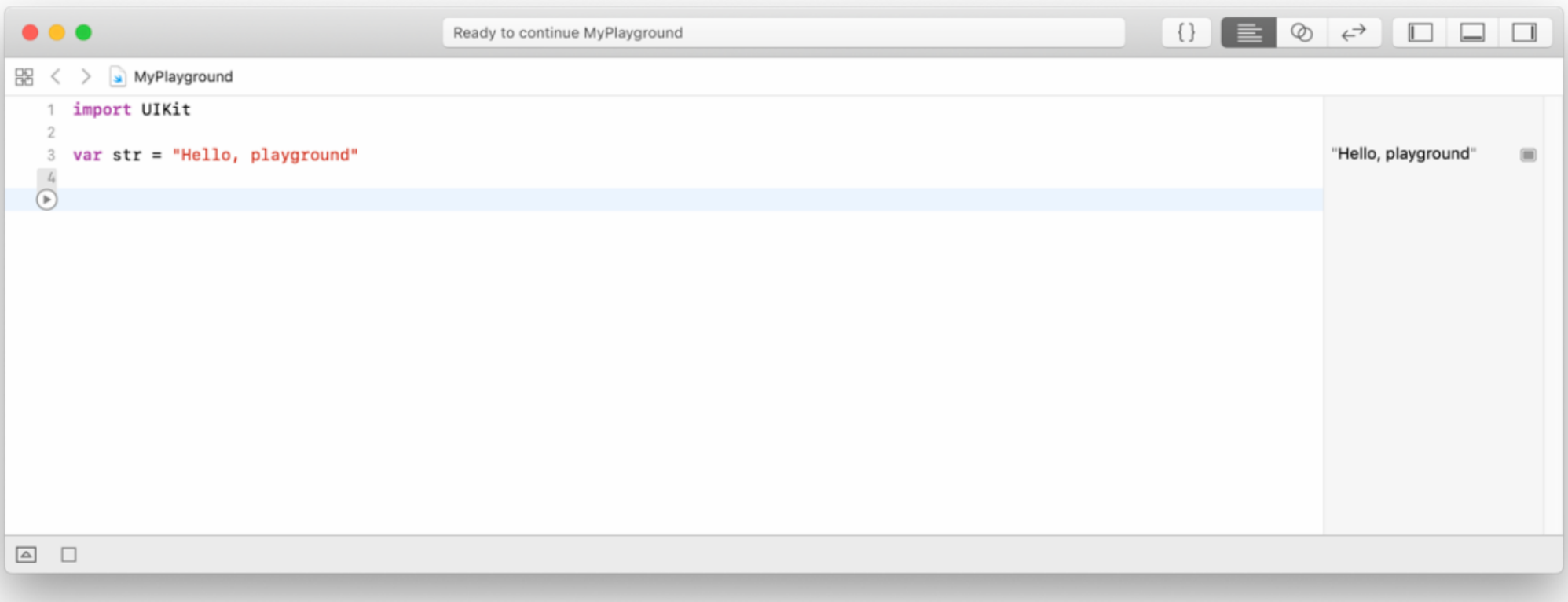
1. Open Xcode.
2. Create a new playground file by selecting File > New > Playground from the menu bar.
3. Choose the iOS platform and the Blank template.
4. Name your playground "Hello, world!" and save it to your course resources folder.

오후 10:25 7월 18일 토요일 65%



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

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 str = "Hello, playground"` in the playground file template created a variable named `str` and set it equal to a string of text containing "Hello, playground." Because the `str` 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 `str` variable instead of your "Hello, world!" text by replacing "Hello, world!" with `str`.

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 `str` 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

문제 1(총 4개)

What are the three primary goals of Swift?

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

정답 확인

Lesson 1.2

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`
- constant
- comment
- `Double`
- function
- immutable
- `Int`
- `let`
- mutable
- property
- type inference
- type safety
- variable
- `var`

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 compile.

```
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 correct, 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"
let anzahlDerBücher = 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, which means that you lowercase the first letter in the name and then capitalize the first letter of each new word. You've probably noticed examples of this already: `defaultScore` is the camel case treatment for the combination of `default` and `score`. Camel case is easier to read than all the words mushed together. For example, `defaultScore` is clearer than `defaultscore` and `restauranutsNearCurrentCity` is clearer than `restaurantsnearcurrentcity`.

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 Swift type comes with properties and functions that may be helpful when working with constants or variables of that type. A property represents information about an object, and a function performs some sort of action with the object.

Consider a simple `Person` type defined below.

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

  func sayHello() {
    print("Hello there! My name is \(firstName) \(lastName).")
  }
}
```

You probably aren't familiar with all the syntax in the code block above. That's OK. It's called a type definition, and it defines the properties and functions of a `Person` type.

You can think about a type definition as a blueprint for how objects should look and function. This example describes how objects of the type `Person` should look and perform.

By reading it, you can see there are two `String` values, `firstName` and `lastName`. These are called properties, which store information about the person. You also see `sayHello()`, which is a function and defines an action for the person.

Whenever you create a new variable, you're creating an instance of a type. 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.

Swift comes with many predefined types that make it easier to write clean code. Whether your program needs to include numbers, strings, or Boolean (true/false) values, you can use types to represent a specific kind of information.

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

Type name	Symbol	Purpose	Example
Integer	`Int`	Represents whole numbers, or integers.	`4`
Double	`Double`	Represents numbers requiring decimal points.	`13.45`
Boolean	`Bool`	Represents `true` or `false` values.	`true`
String	`String`	Represents text.	`"Once upon a time..."`

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

The Swift standard library includes all of the types above, so they're always available to you.

In addition, you can define your own types in Swift using the same structure you saw earlier in the `Person` example. 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 checks on all of your constants and variables, and flags any mismatched types as errors. If you mismatch types, you won't be able to 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 requires you to be careful about which types you work with, the example above is an obvious mismatch. Assigning a `String` value to an `Int` variable doesn't make sense. When you access `playerScore`, you know you want a score value represented by a number and not by text.

Type safety also applies to values that represent data that may seem compatible. For example, the compiler 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 numbers, 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.

There may be cases when it's useful, or even required, to explicitly specify the type of a constant or variable. This is called 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: Float = 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`.

REVIEW QUESTIONS

문제 1(총 7개)

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

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

정답 확인

Lesson 1.3

Operators

Operators are the symbols that make your code work. You'll use them to do things 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 so 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.

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.

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`.

REVIEW QUESTIONS

문제 1(총 6개)

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

정답 확인

Lesson 1.4

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 operator (`?:`) to conditionally assign different values to a constant or variable

Vocabulary

- conditional
- if statement
- If-else statement
- interval matching
- logical operators
- switch
- ternary 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 OPERATORS

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

Start by taking a look through this list of the most common logical operators:

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

You can mix and match logical operators to create what's called a Boolean value, or `Bool`. Boolean values are either `true` or `false` and can be combined 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 basically says "If this condition is true, then run this block of code." If the condition isn't true, the program will skip 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 will run 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 operator to a `Bool` constant or variable in order 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." because even though `isPluggedIn` is false, `hasBatteryPower` is true, and the `||` OR operator checks to see if either option is true.

`Switch` STATEMENT

You've seen `if` statements and `if-else` statements that allow you 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 hard to read after a small number of options. Swift has another tool for control flow called a `switch` statement, which is great 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 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 `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 difficult 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 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, called a ternary operator (`? :`), for writing more concise code.

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` constant; otherwise, assign `b`." In this case, `a` is greater than `b`, so it's assigned to `largest`.

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

LAB

Open and complete the exercises in `Lab—Control Flow.playground`.

REVIEW QUESTIONS

문제 1(총 10개)

Which of the following logical operators means "equals"?

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

정답 확인

Lesson 1.5

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

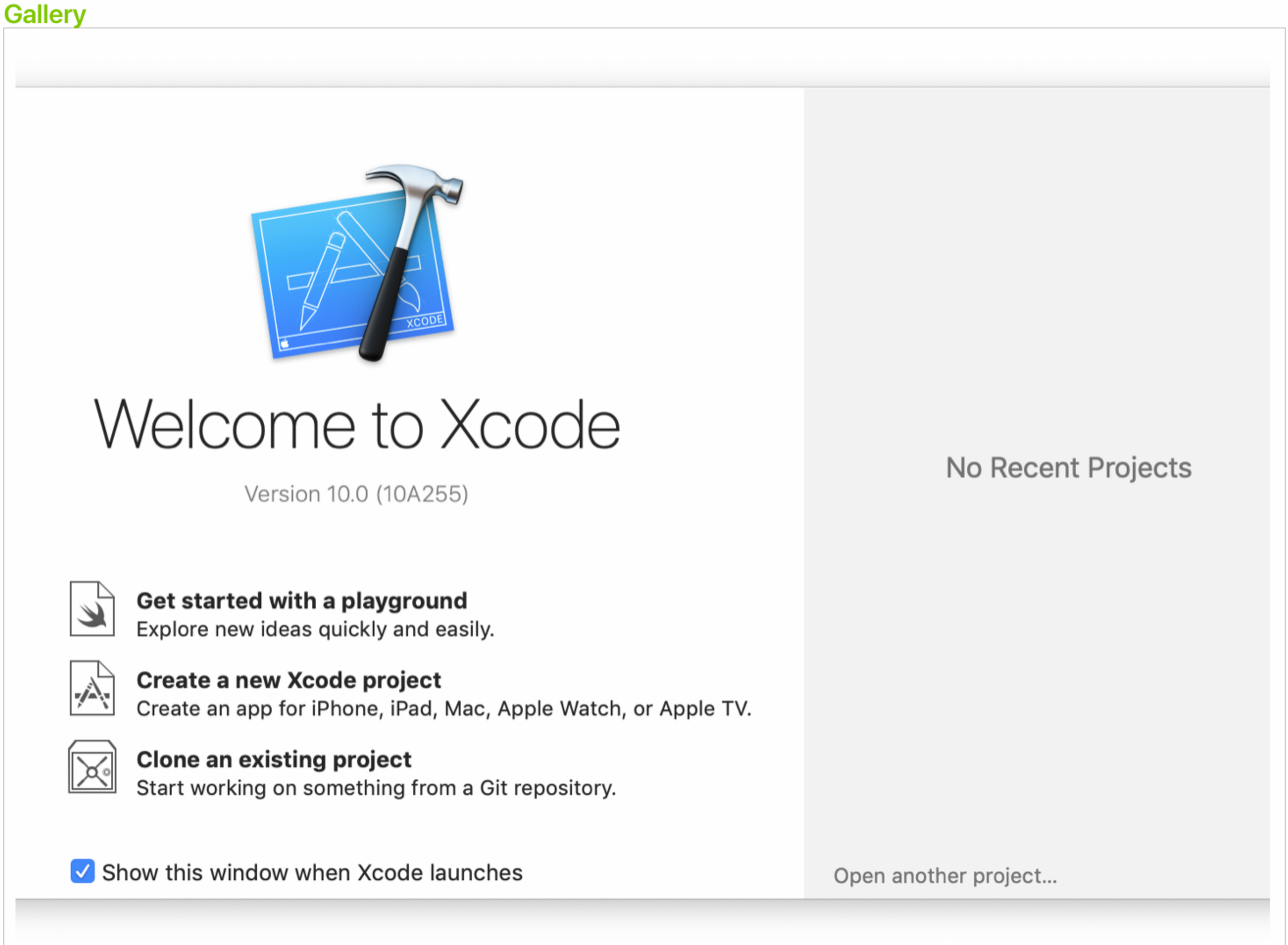
- active scheme
- assistant editor
- compiler
- console pane, or console area
- debug area
- executable apps
- Git
- IDE
- project
- project navigator
- project template
- push notifications
- standard editor
- storyboard
- target
- utilities area
- variables view
- version editor

Related Resources

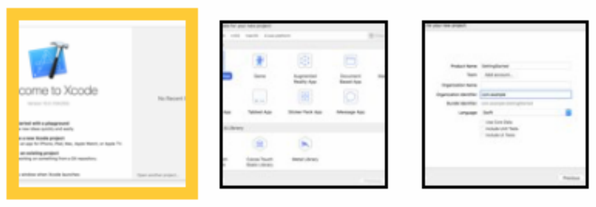
- Xcode Help
- Xcode Overview
- WWDC 2016: Introduction to Xcode

오후 10:29 7월 18일 토
도서 지금
1개의 알림
64%

Gallery

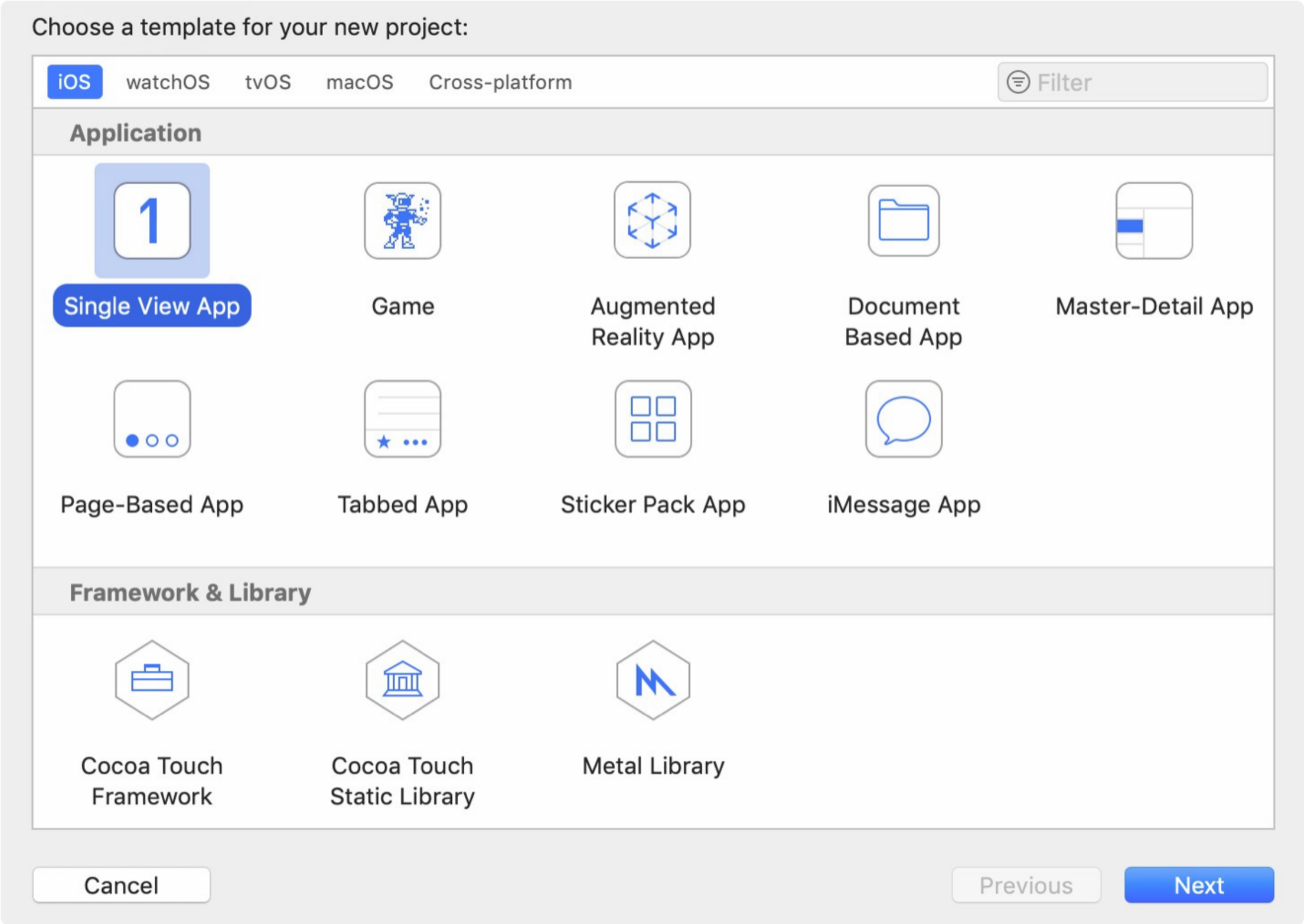


To begin your exploration of the Xcode IDE, you'll need to create a new project. Open Xcode and select "Create a new Xcode project."



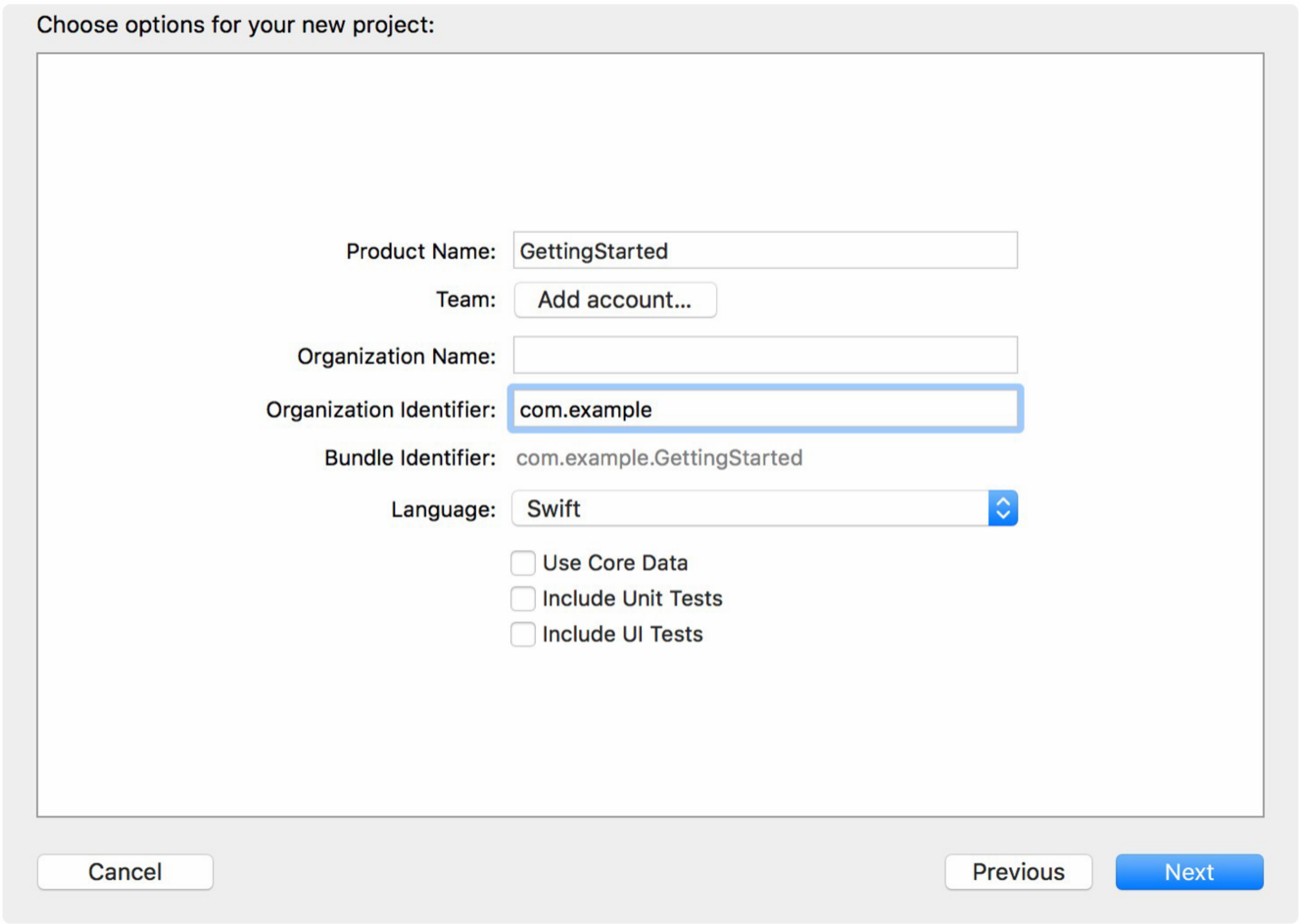
오후 10:30 7월 18일 토요일 64%

Gallery



On the next screen, you'll see that Xcode provides many starting templates for creating new applications, sectioned according to the platforms that Xcode supports: iOS, watchOS, tvOS, and macOS. Select the iOS Single View App template. Click Next.

Gallery



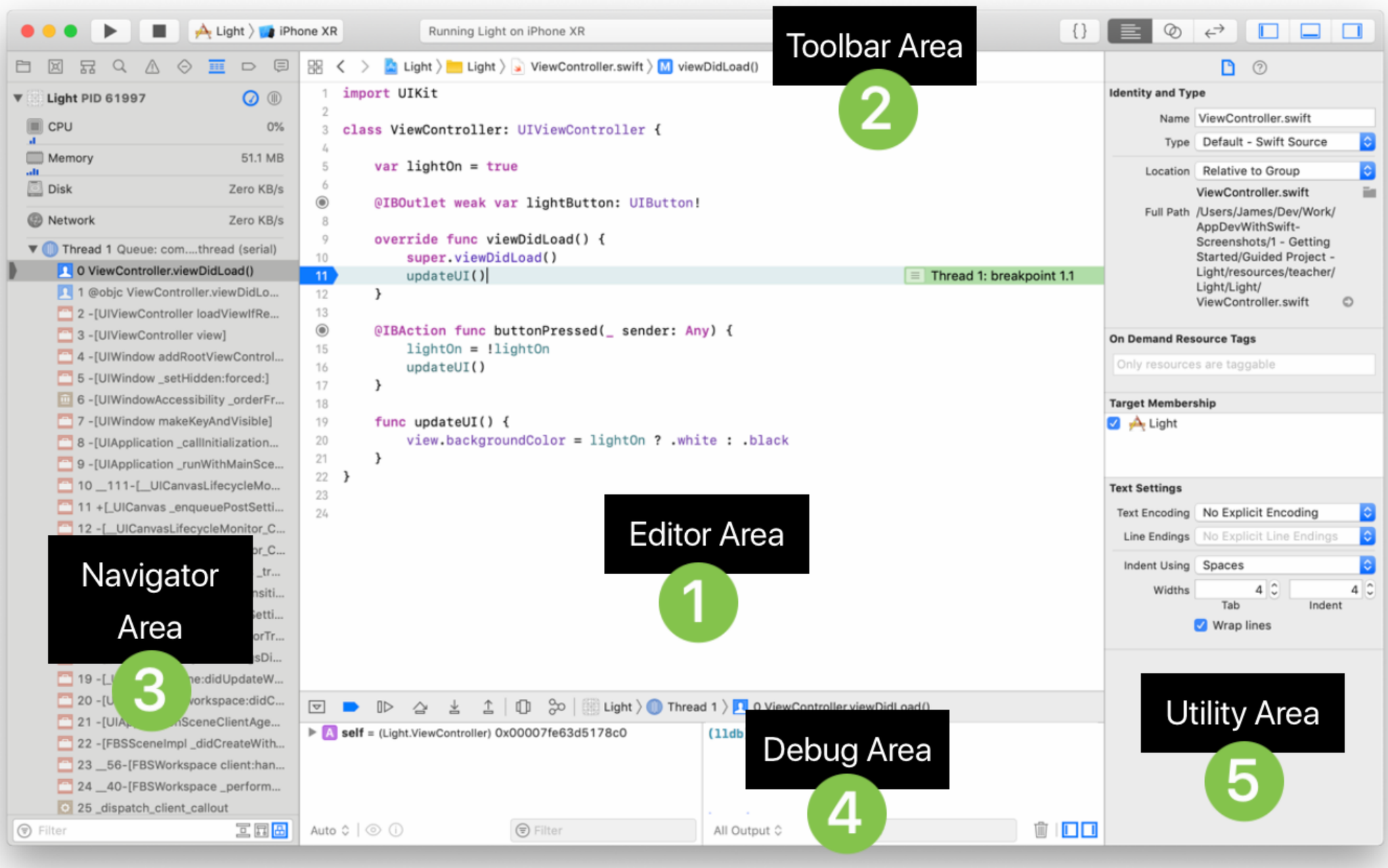
Enter "GettingStarted" in the Product Name field. Xcode will use this name for your project and your app. Under Organization Identifier, you may specify your organization's identifier, or just use "com.example." Leave Use Core Data, Include Unit Tests, and Include UI Tests all unselected. Click Next.

Select a location for saving your project and click Create. Until you work with a version control system, such as Git, you can leave the box that asks if you'd like to create a git repository unselected.

Now that you've created the project, you'll arrive at the main interface of Xcode. You'll notice that it takes up a lot of space. You might want to resize the window to fill the majority of your desktop—or take Xcode full-screen by clicking the green circle in the upper left of the window or using the Command-Control-F keyboard shortcut.

XCODE INTERFACE

The Xcode workspace is divided into five main sections.



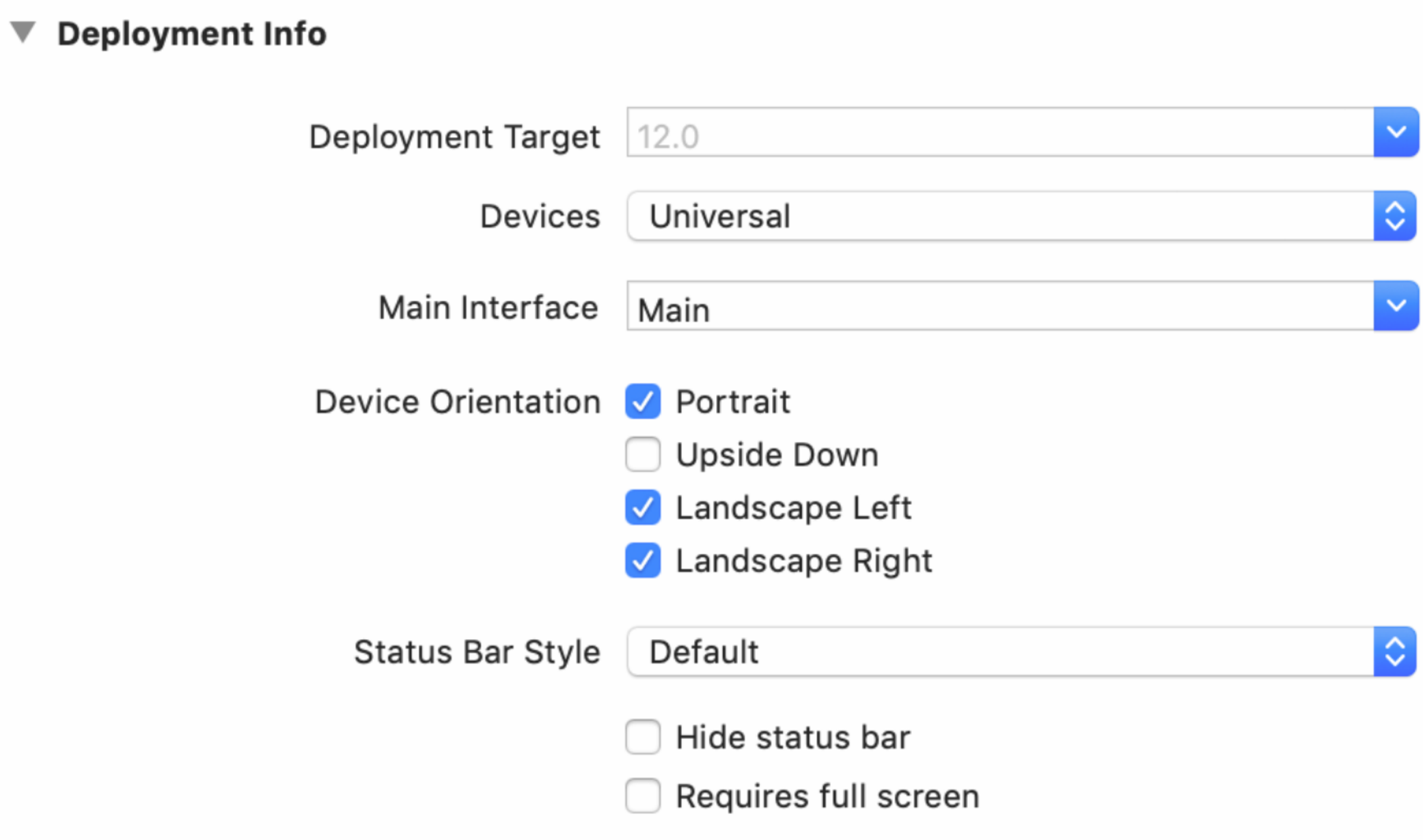
XCODE FILE TYPES

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.

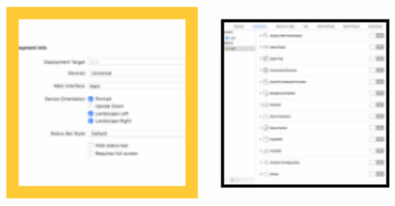
At the very top of the project navigator, you'll see a file with a tiny blue Xcode icon. Click the file to open it in the editor area. This is the `.xcodeproj` file, which includes all the settings for your project and its targets. Each target is a product that Xcode can build from the project. For now, 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.

오후 10:30 7월 18일 토요일 64%

Gallery

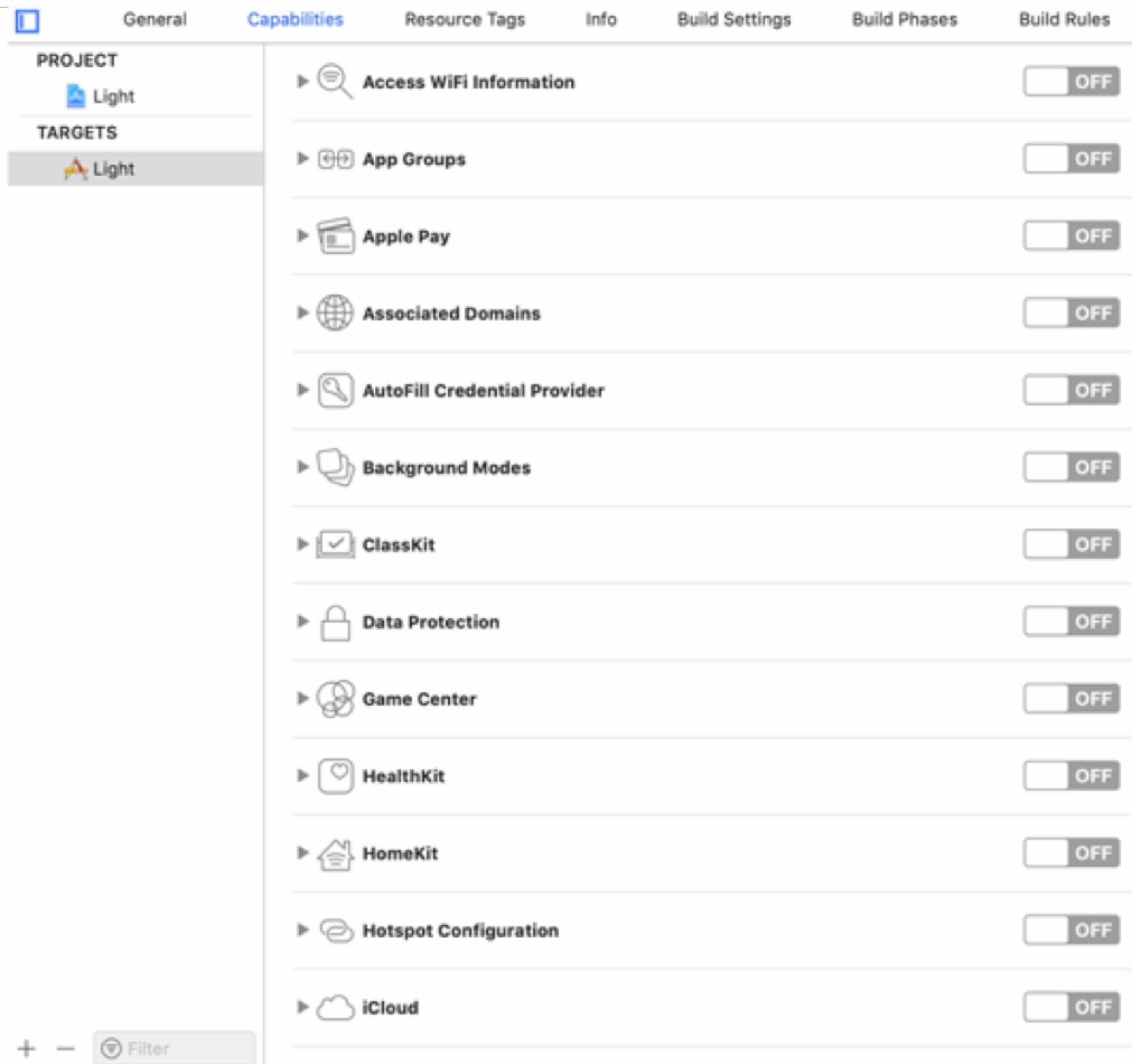


The particular project you're working with has only one target: an iOS app. Within the project file, you can change all the details of a particular target. For example, under Deployment Info, you can specify which version of iOS your code must support, change which devices your app is intended to run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

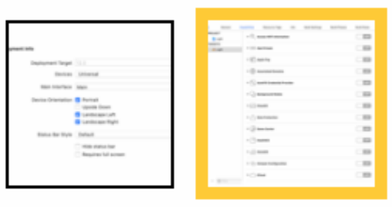


Files that include the `.swift` extension contain Swift code, as you'd expect. Whenever you build your app, Xcode gathers up all included `.swift` files and runs them through the Swift compiler, which converts the Swift code into a format your selected device understands.

Gallery



By selecting Capabilities at the top of the pane, you can enable different features within the selected target. For example, if your application will need to accept push notifications, you can move the on/off switch from off to on, and Xcode will configure everything necessary for your app to receive notifications from the Apple Push Notifications service. Click the triangle (▶) next to each capability to read more information about the ways your target will be affected.



Files that include the `.swift` extension 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 Swift code into a format your selected device understands.

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.

The `.xcassets` file makes it easy to maintain all your assets, without needing to work with individual icons and images. This asset catalog has two advantages: It reduces the list of items in the project navigator; and it shows which images will be used for devices of different screen sizes—helping you quickly identify that you're missing assets for a particular device.

The `Info.plist` file contains a list of properties and settings for your app. In earlier versions of Xcode, you needed to edit this file directly, but newer versions allow you to edit settings—including app version, app icon, and device orientation—using the `.xcodeproj` file that 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:

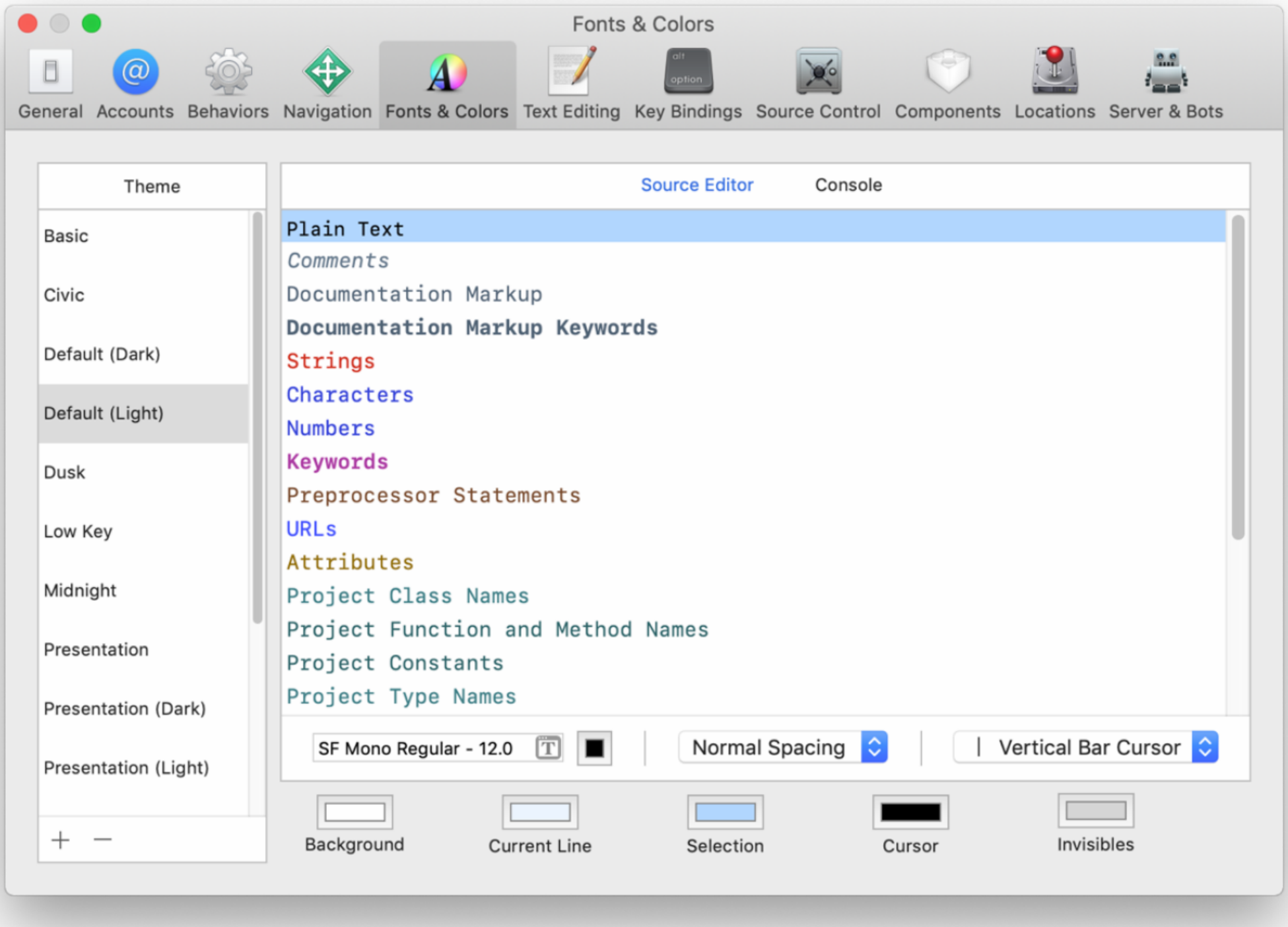
- Command-B—Build the project
- Command-R—Build and run the project
- Command-/—Toggle comments on selected rows of code
- Command-[—Shift the selected code left
- Command-]—Shift the selected code right
- Command-↑—Navigate to the top of the file

Refer to the Xcode Keyboard Cheat Sheet for a full list of shortcuts.

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 navigating to Xcode -> Preferences in 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 up the Xcode Preferences and navigate to the Fonts & Colors menu.



You can choose a new theme, or set of colors and fonts, by selecting different options in the list on the left. The Civic theme is great if you prefer a dark editing mode, and the Presentation or Presentation Large options are perfect when presenting code to a group.

You can read much more about Xcode and its various tools in the Xcode Help documentation, which you can open in Xcode by choosing Help -> Xcode Help in the menu bar.

REVIEW QUESTIONS

문제 1(총 4개)

Where is the "Build and run" button located in Xcode?

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

정답 확인

Lesson 1.6

Building, Running, and Debugging an App

While Xcode playgrounds are a great tool 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.

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

- breakpoint
- bug
- deprecated code
- developer mode
- compiler error, or error
- exception
- landscape
- portrait
- step control buttons
- warning

Related Resources

- Xcode Help: Run your app in Simulator
- Xcode Help: Debug your app

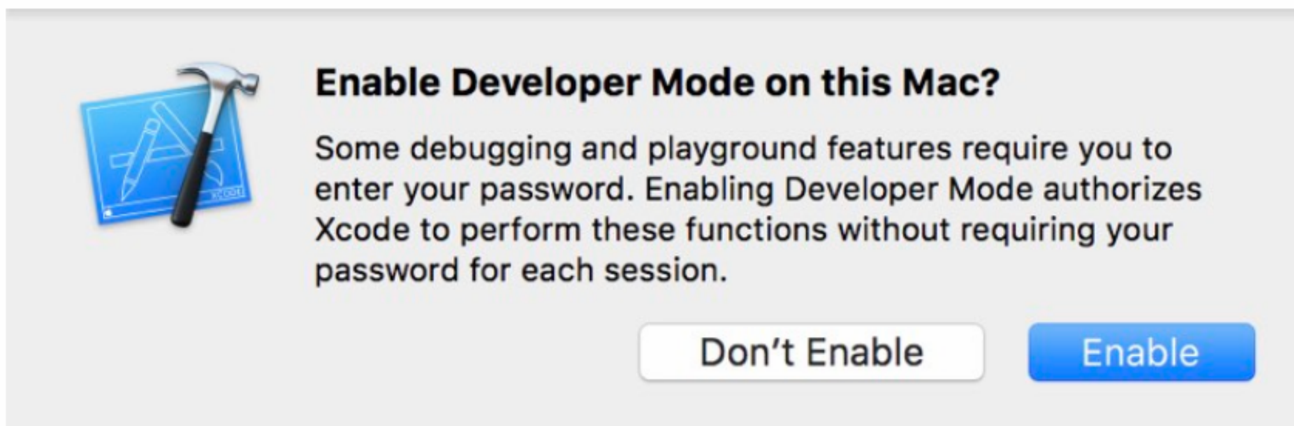
BUILDING AND RUNNING

Your first step is to create a new project in Xcode. You'll use the same iOS Single View App template as in the previous lesson. 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.

If you're running an app for the first time, Xcode asks whether you'd like to enable developer mode on your Mac. Developer mode gives Xcode access to certain debugging features without requiring you to enter your password each time. All the lessons in this course assume you've enabled developer mode.



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 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.

There are also some software limitations with Simulator. For example, push notifications can be delivered 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, you might try testing the code on your iPhone or iPad. Your issues may be resolved.

Overall, Simulator works very 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.

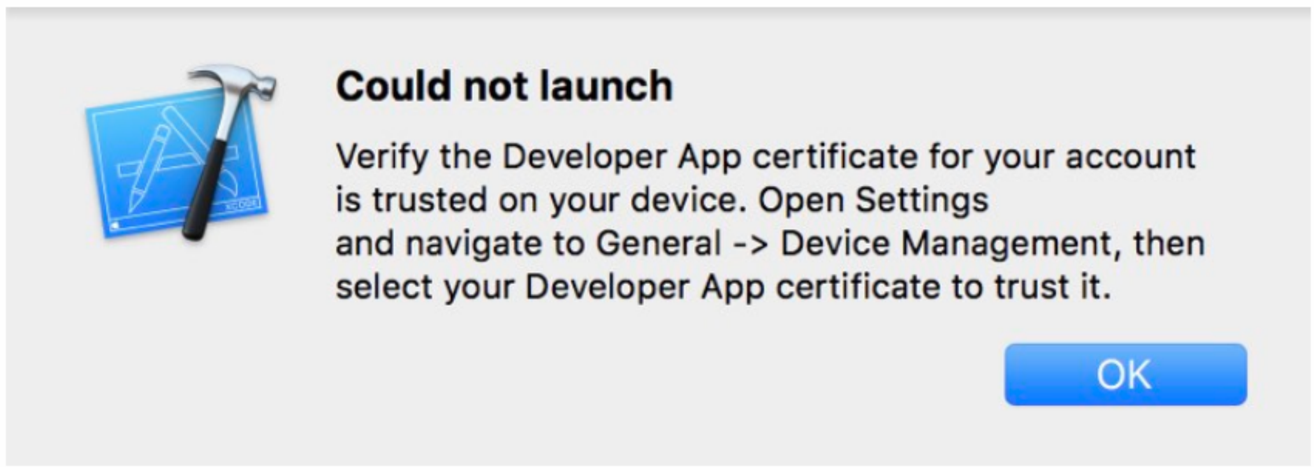
Using a web browser, go to developer.apple.com and click Account. You can sign up using your existing Apple ID. If you don't have an Apple ID, go ahead and create one—it's also free. Once you've entered your Apple ID, your developer account is good and you can head on back to Xcode.

(A free account enables you to run your iOS apps on only one physical device. If you want to distribute your apps to multiple devices or publish them to the App Store, you'll need to enroll in the Apple Developer Program.)

Now you'll need to tell Xcode about your new developer account. Open Xcode Preferences using the keyboard shortcut Command-Comma, and select the Accounts button near the upper-left corner. Click the "+" button in the lower-left corner, and 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 the following prompt, asking you to trust the developer certificate. Follow the instructions within the alert to allow the device to run your apps.

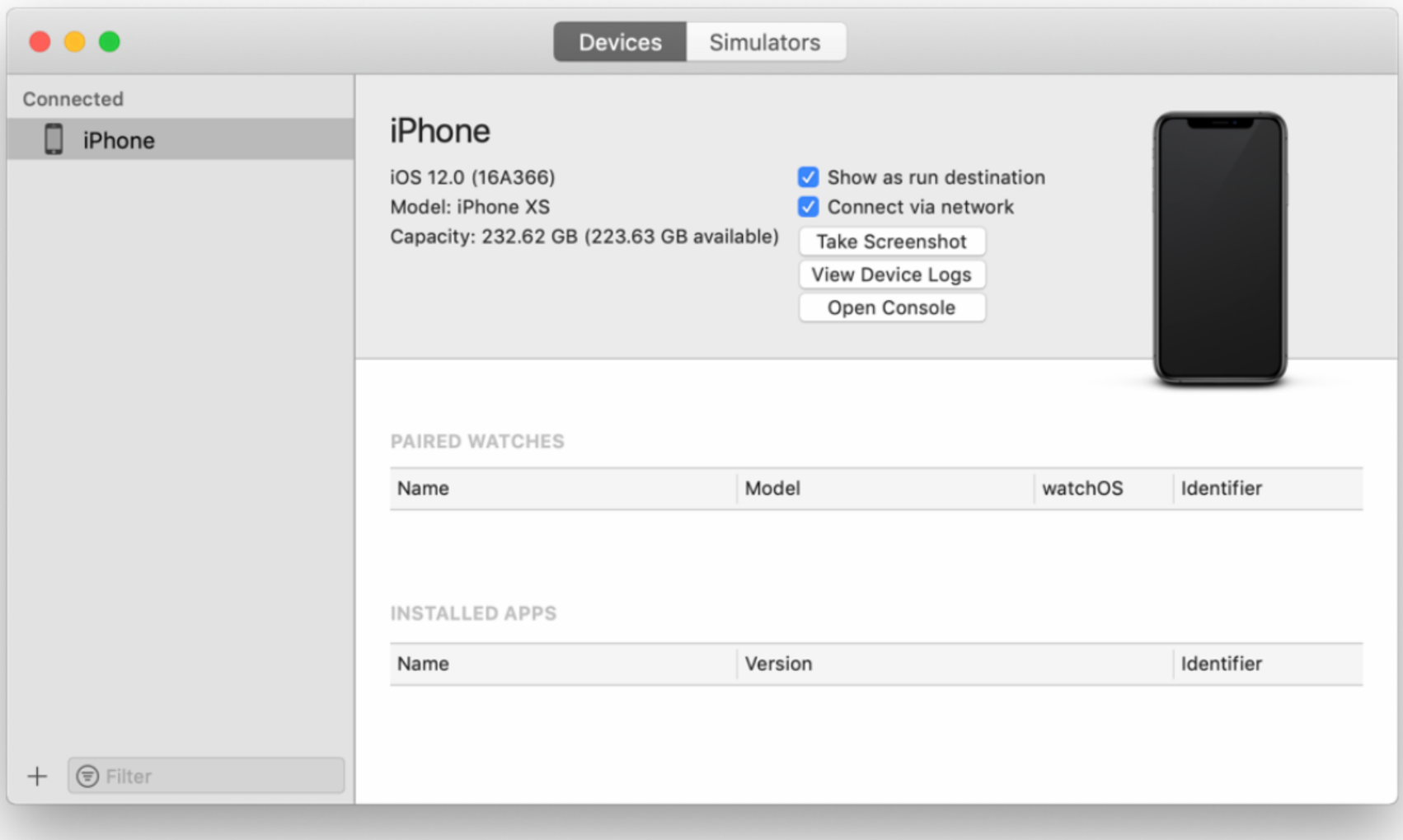


Build and run once more, and you should see the same simple white screen on your iOS device. To stop the app from running, click the Stop button near the left end of the Xcode toolbar.

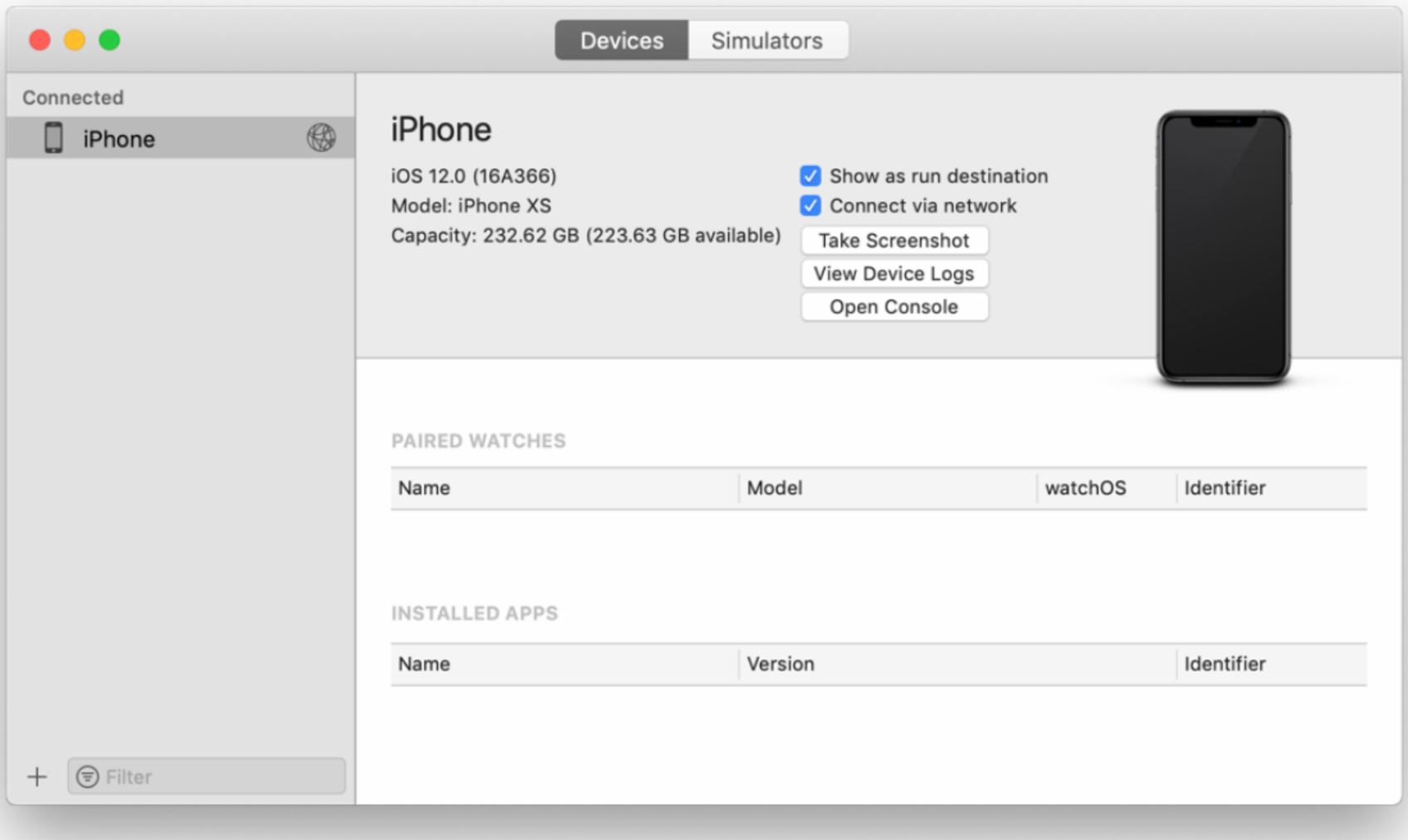
Building and Running Wirelessly

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.



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.



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 does not 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, simply open the Devices and Simulators window, hold Control and click on your device, then click Connect via IP Address in the dropdown presented. You'll 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.

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 the 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.swift` 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 on the line you just added.

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—since 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.swift`, 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.

One 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, the compiler cannot be relied upon to properly fix all errors—you need to be able to address errors on your own.

```
11  class ViewController: UIViewController {
12
13      override func viewDidLoad() {
14          super.viewDidLoad                                   Expression resolves to an unused function
15          navigationController.title = "Debugging"
16      }
17
18
19  }
20
21
```

Value of optional type 'UINavigationController?' must be unwrapped to refer to member 'title' of wrapped base type 'UINavigationController'

Chain the optional using '?' to access member 'title' only for non-'nil' base values — Fix

Force-unwrap using '!' to abort execution if the optional value contains 'nil' — Fix

The other error has an exclamation point in the center . Xcode doesn't offer a fix-it for this type of error, but it does provide 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 last error.

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.swift`, 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
```

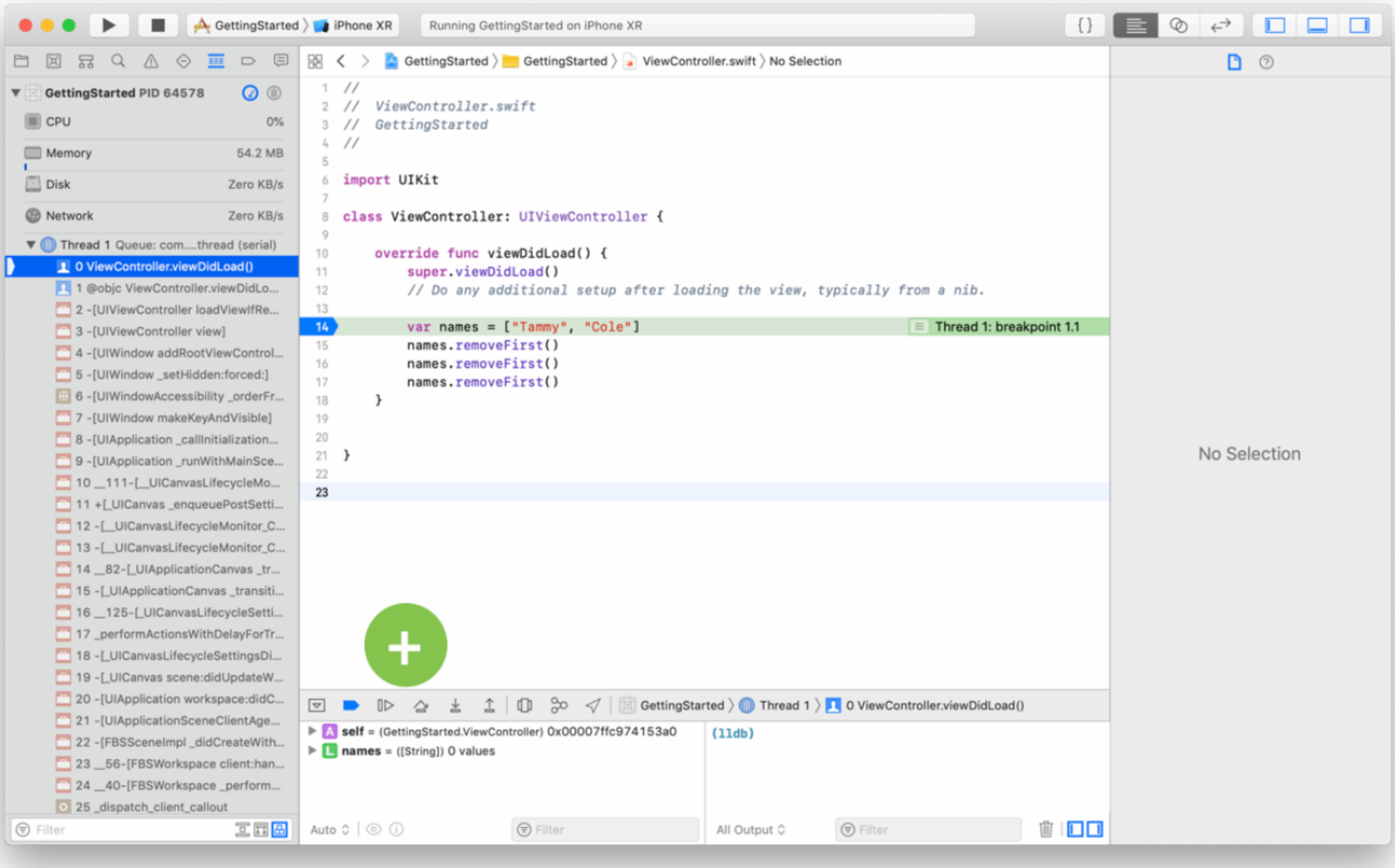
The program crashed when it tried to remove the remaining first element and couldn't find one. You've already guessed that. But imagine you're unsure how to fix the problem. Xcode can help you to step through the program one line at a time.

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 left-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  //
2  //  ViewController.swift
3  //  GettingStarted
4  //
5
6  import UIKit
7
8  class ViewController: UIViewController {
9
10     override func viewDidLoad() {
11         super.viewDidLoad()
12         // Do any additional setup after loading the view, typically from a nib.
13
14         var names = ["Tammy", "Cole"]
15         names.removeFirst()
16         names.removeFirst()
17         names.removeFirst()
18     }
19
20
21 }
22
```

Build and run your app. You'll see the program pause at the breakpoint. That's good. In the debug area, show the variables view to inspect the current values. Since the breakpointed line hasn't yet been executed, `names` contains no values.



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.

Click the "Step over" button again 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. (Remember that the compiler is very literal; it 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 around, notice that the app stops execution right after opening in the 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. Take a look at the text in the console area to learn what the issue might be. Go ahead and 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

문제 1(총 3개)

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

정답 확인

Lesson 1.7

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

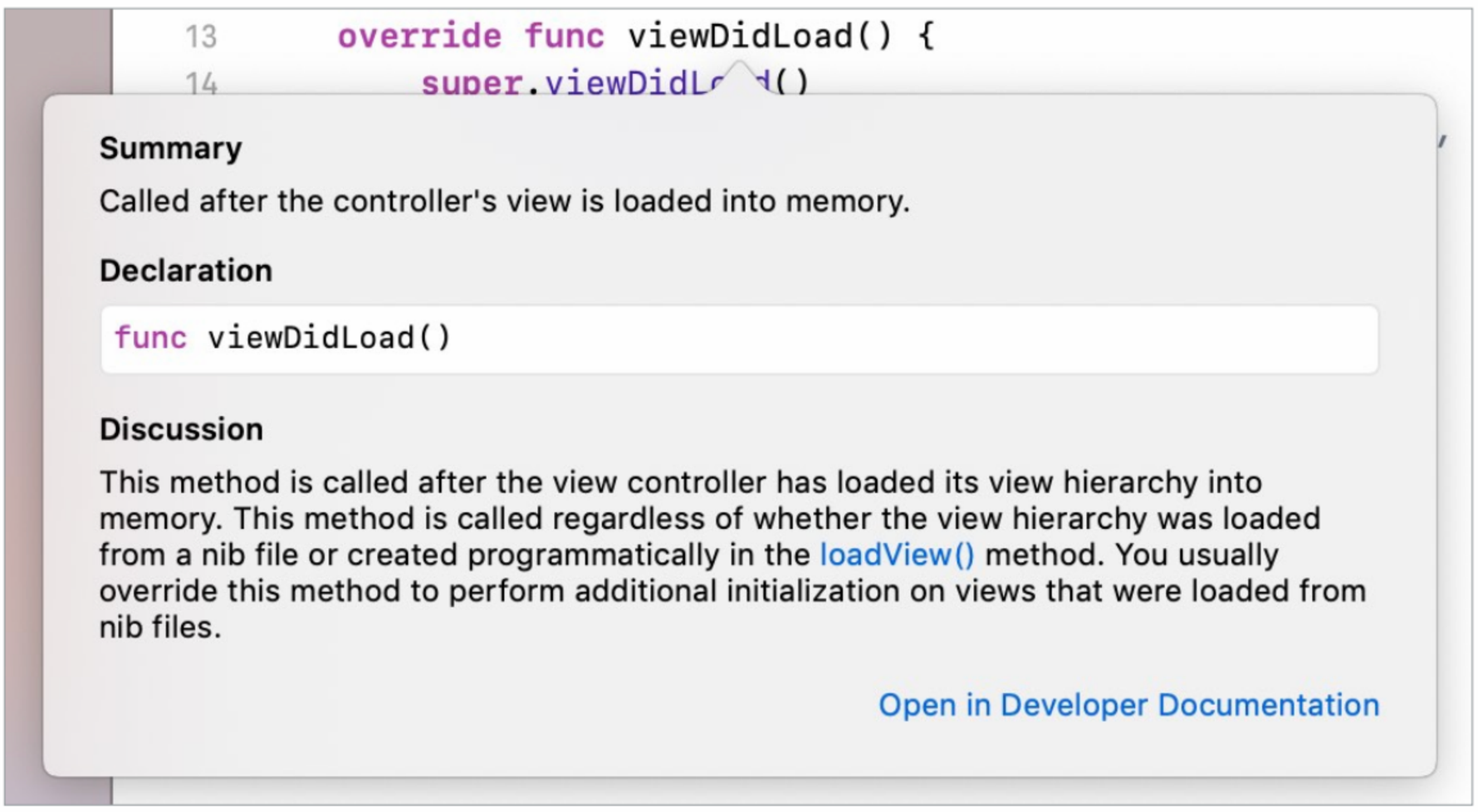
Related Resources

- API Reference
- Xcode Help: Search for developer documentation

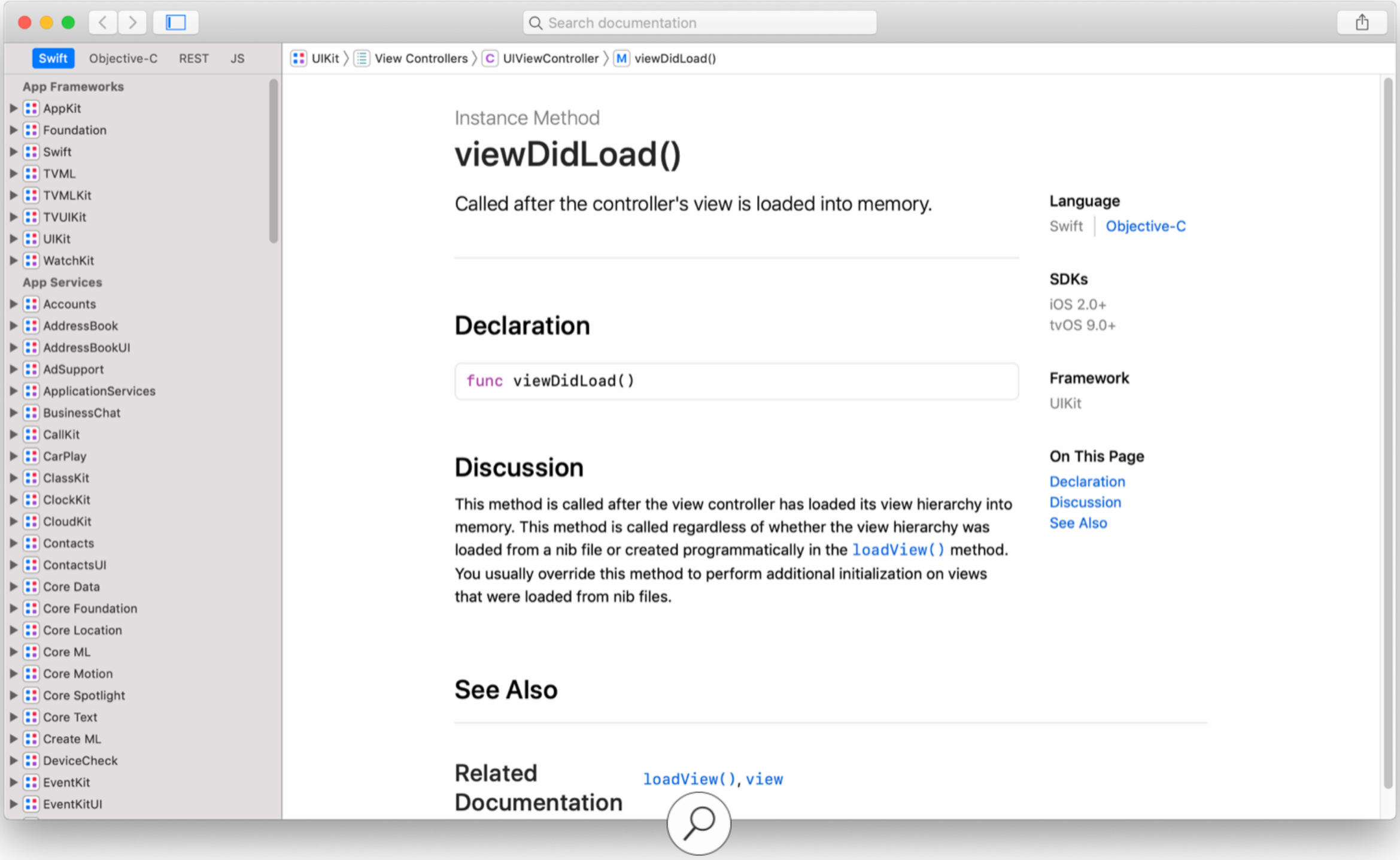
Every developer has a preferred method of accessing documentation. Some prefer to view it in a web browser, while others like to use the documentation browser provided by Xcode. By learning multiple ways to interact with documentation, you can decide which method works best for you. The faster you can 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 and the OS versions that support it. It also shows which framework the function belongs to. In this case, the framework is `UIKit`.



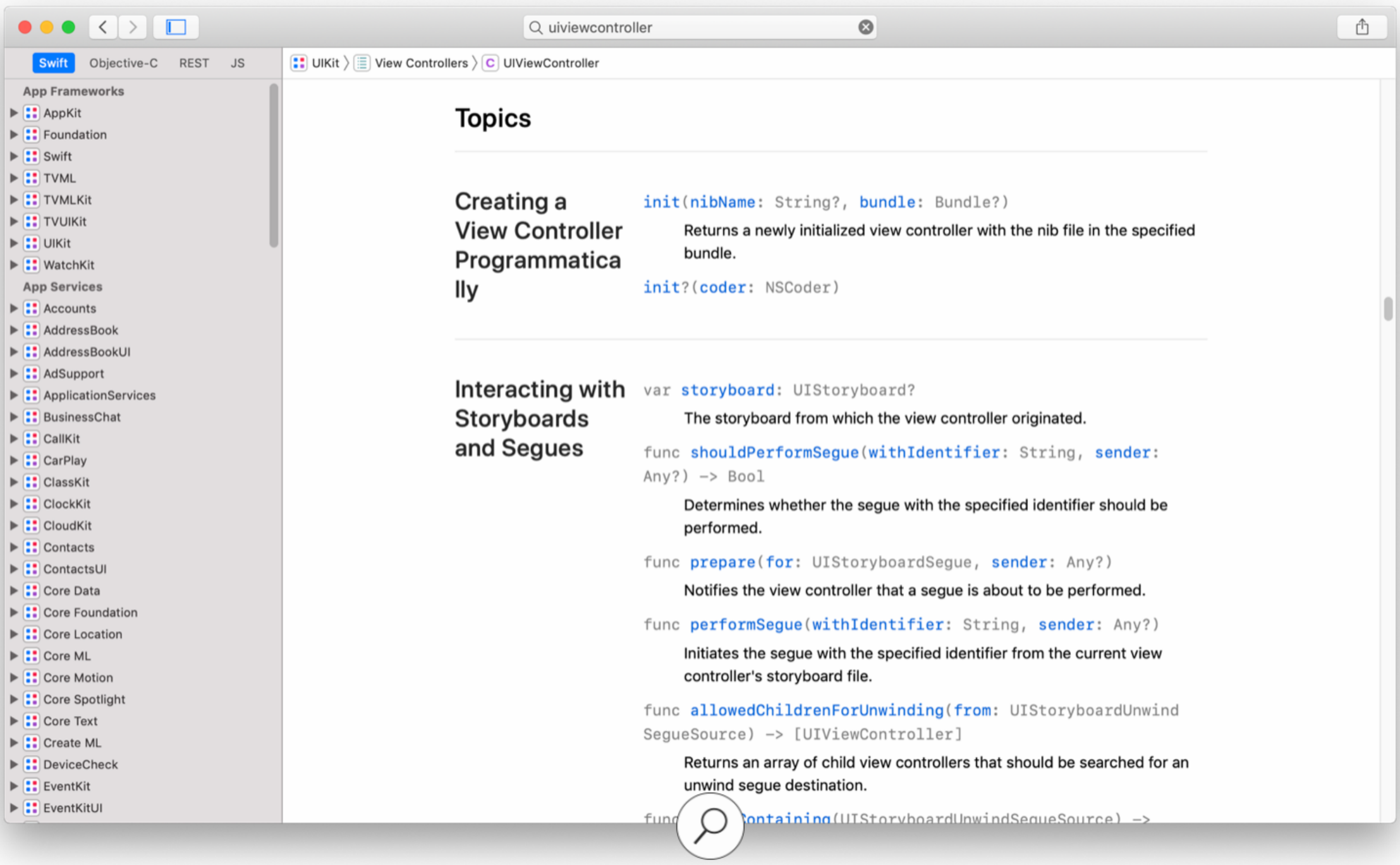
Within the Quick Help popover, click Open in Developer Documentation to access Xcode documentation, which includes more thorough explanations and references to related functions. (You can also access this window from the Xcode Help menu or by using the shortcut Command-Shift-0.) The links on the right of the documentation allow you to jump quickly to different information on the same page.



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. 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.

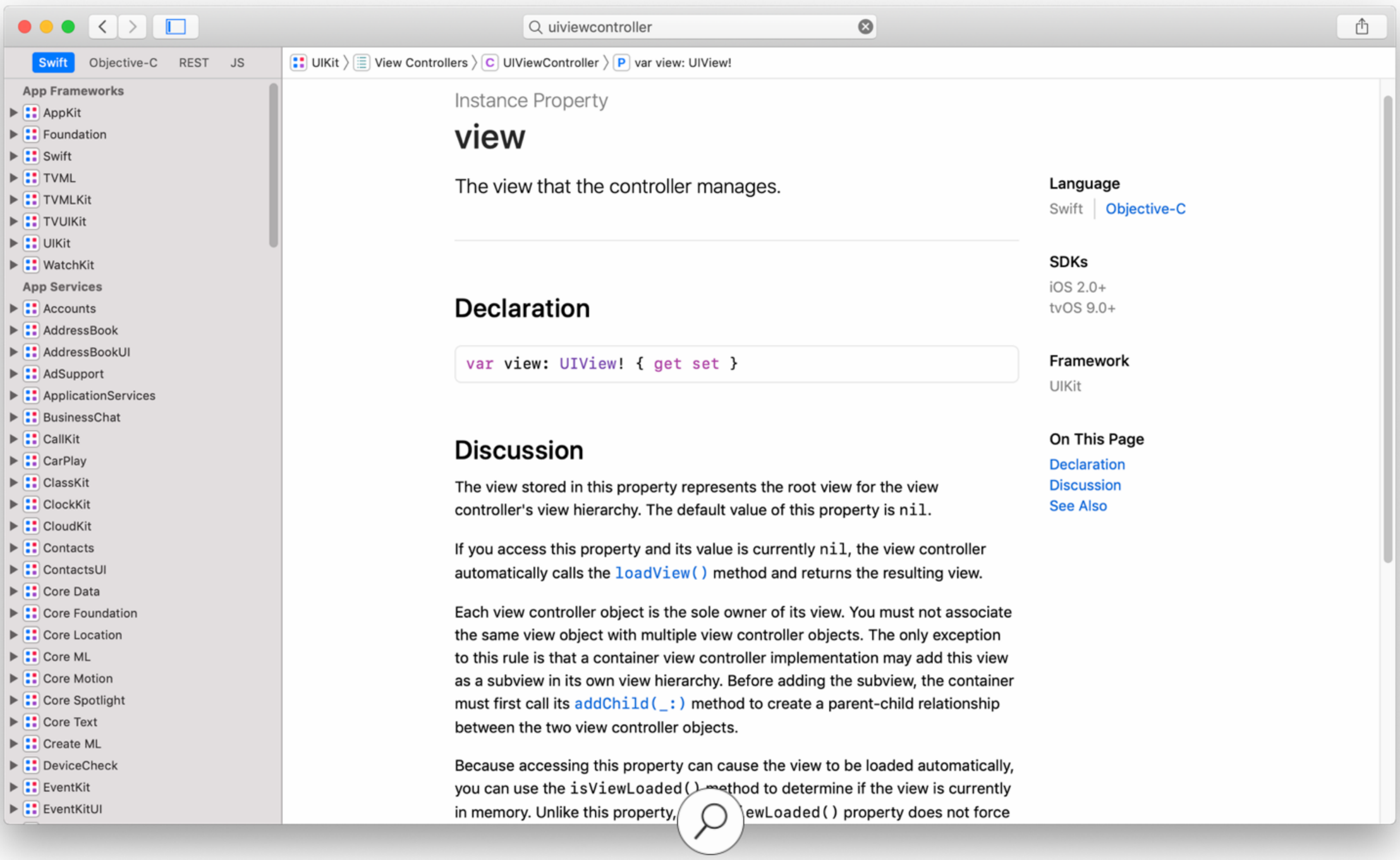
Try navigating through the Xcode documentation. Start typing `UIViewController`. As you type, Xcode will suggest matching options with autocompletion. As soon as you see `UIViewController` in the menu, select it to display the documentation for `UIViewController`.

In the top-right corner, 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.



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

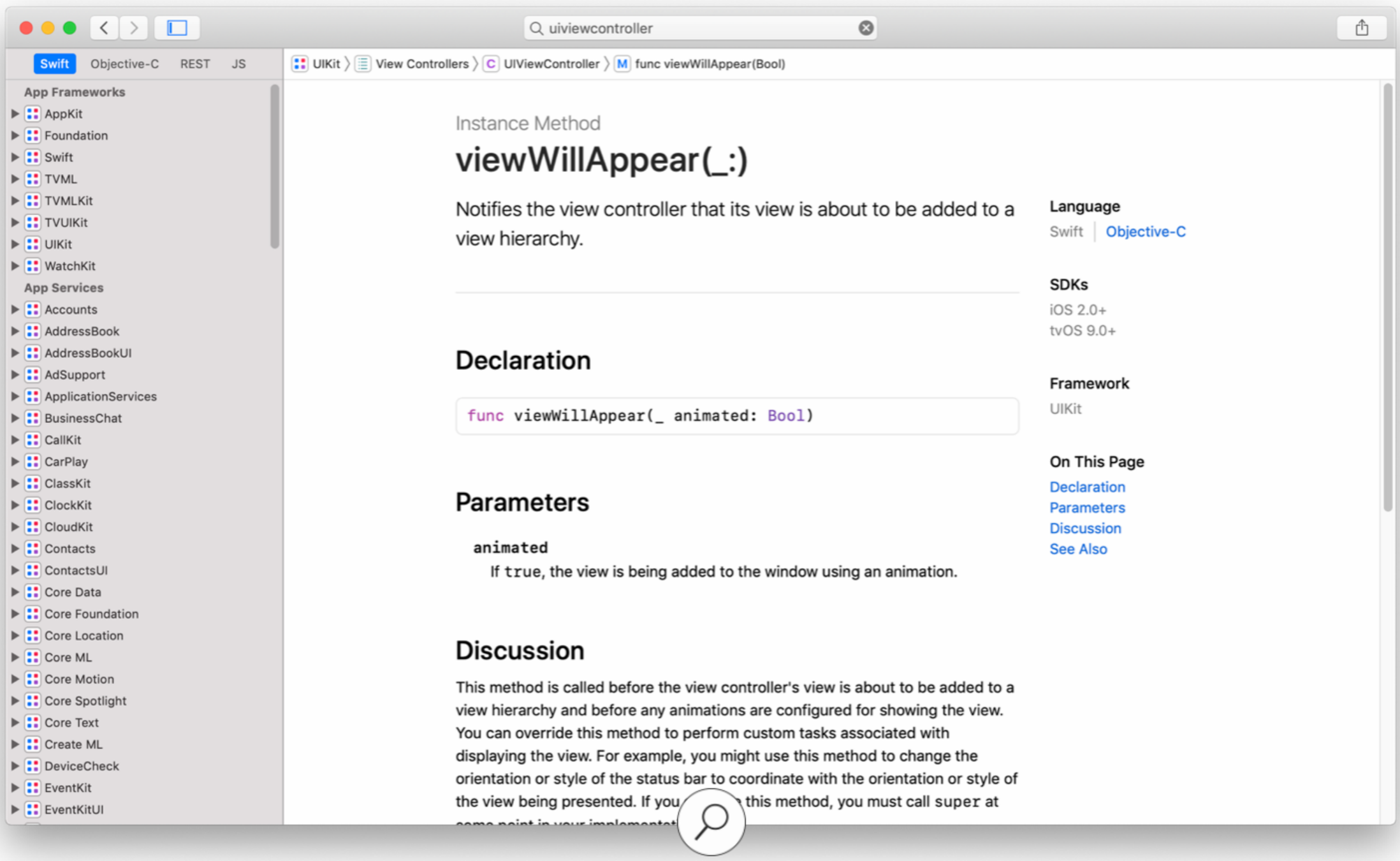
Look through the topics for `UIViewController` and try to find the symbol for the `view` property. Click this symbol.



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(_:)`.



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 GUIDES

Technical documentation isn't always the best learning resource for some programmers or in certain situations. As you work your way through coding concepts, it can be useful to se them in action. Apple provides numerous guides and sample code projects to help you quickly understand how to use frameworks and their functions. Visit the Apple Developer website, where you can view or download these guides and sample code projects.

You will 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 called "Lab - Use Documentation" and save it to your project folder. Use this document to write down your answers to the questions in the rest of this lab.

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 1.8

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

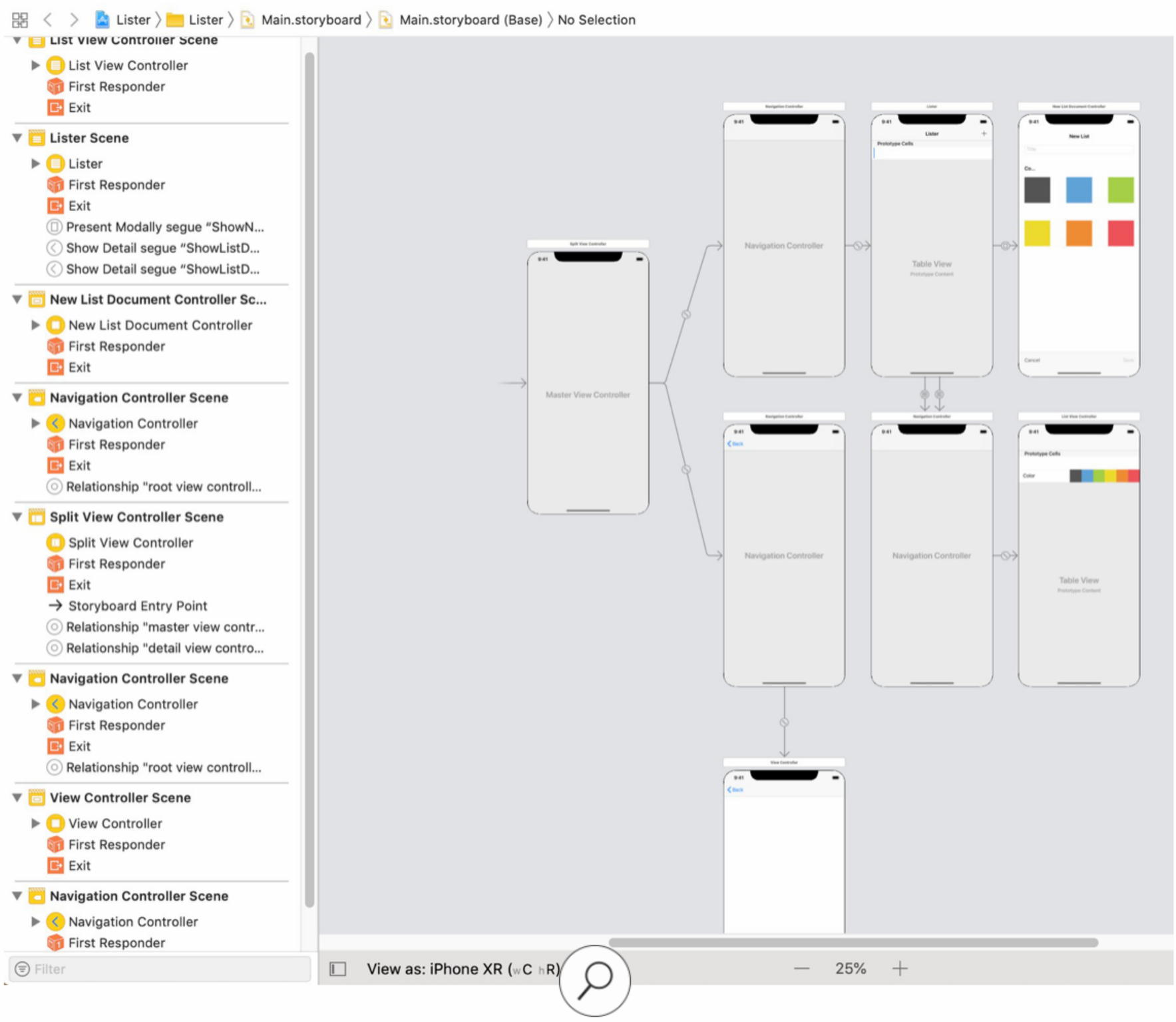
The best way to learn the basics of Interface Builder is to dive into Xcode and explore some of its features. Start by creating a new iOS project using the Single View App template. Name the project "IBBasics".

STORYBOARDS

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

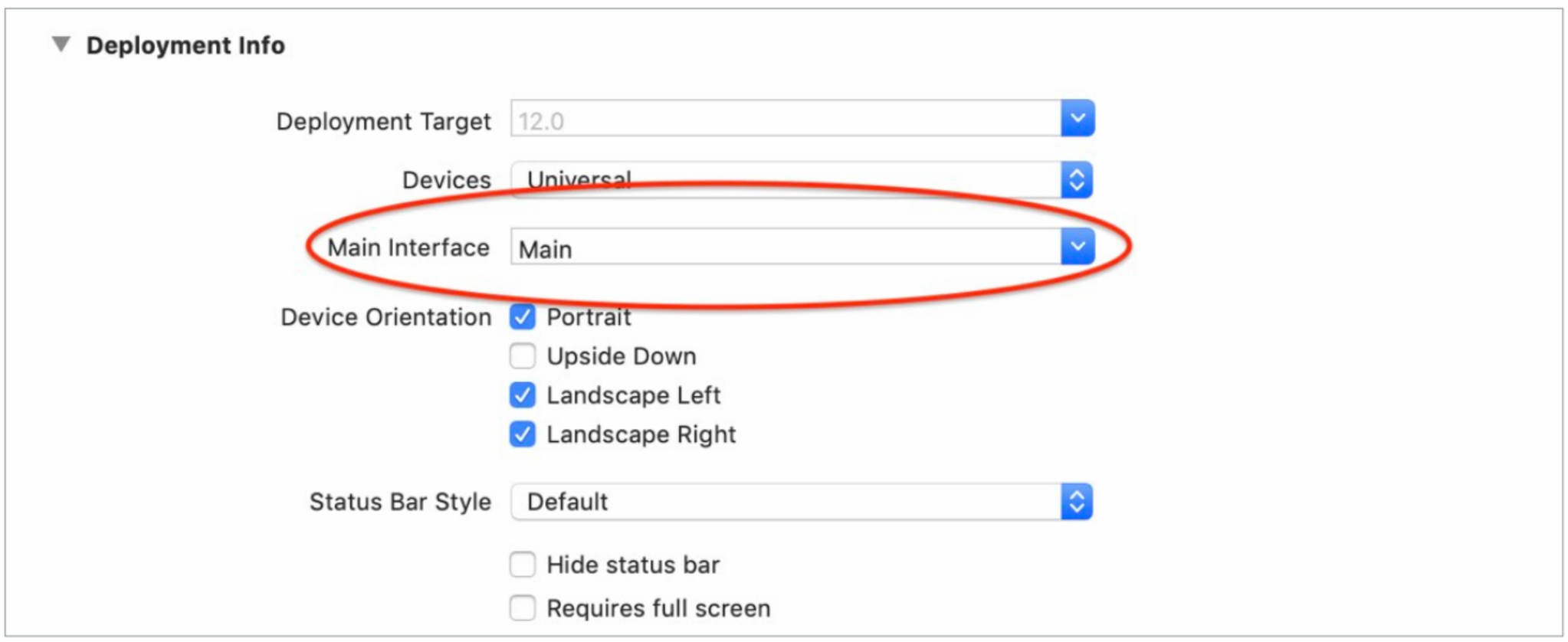
An XIB file contains the user interface for a single visual element, such as a full-screen view, a table view cell, or a custom UI control. XIBs were used more heavily before the introduction of storyboards. They're still a useful format in certain situations, but this lesson will focus on storyboards.

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.



Click `Main.storyboard` in the project navigator to open the file 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 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.

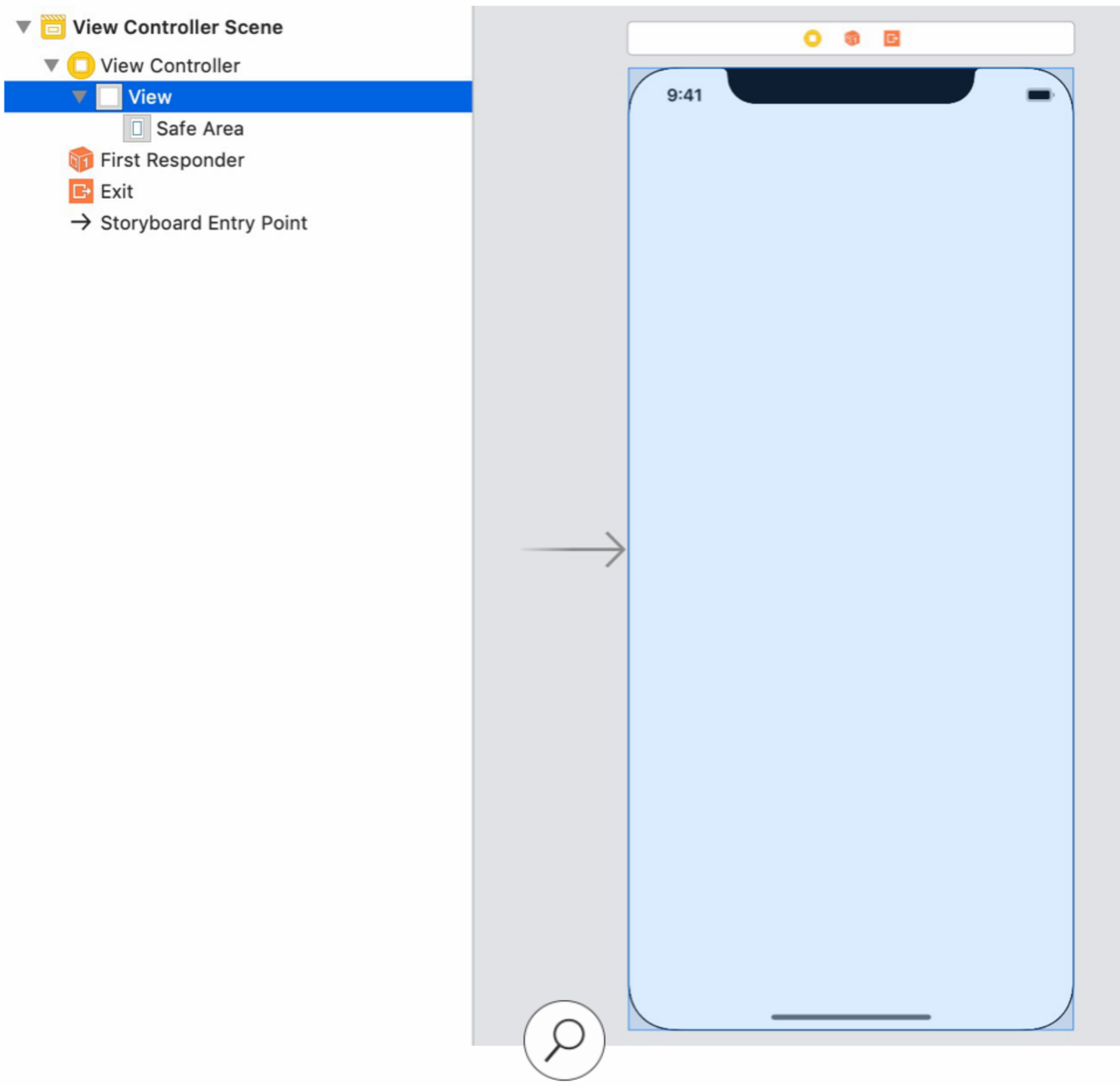
Build and run the project. Simulator displays the same white screen you saw in the storyboard. How did the app know to display this screen? To investigate, select the top-most file (`IBBasics`) in the project navigator, and find the Deployment Info section under the General header. The entry in Main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



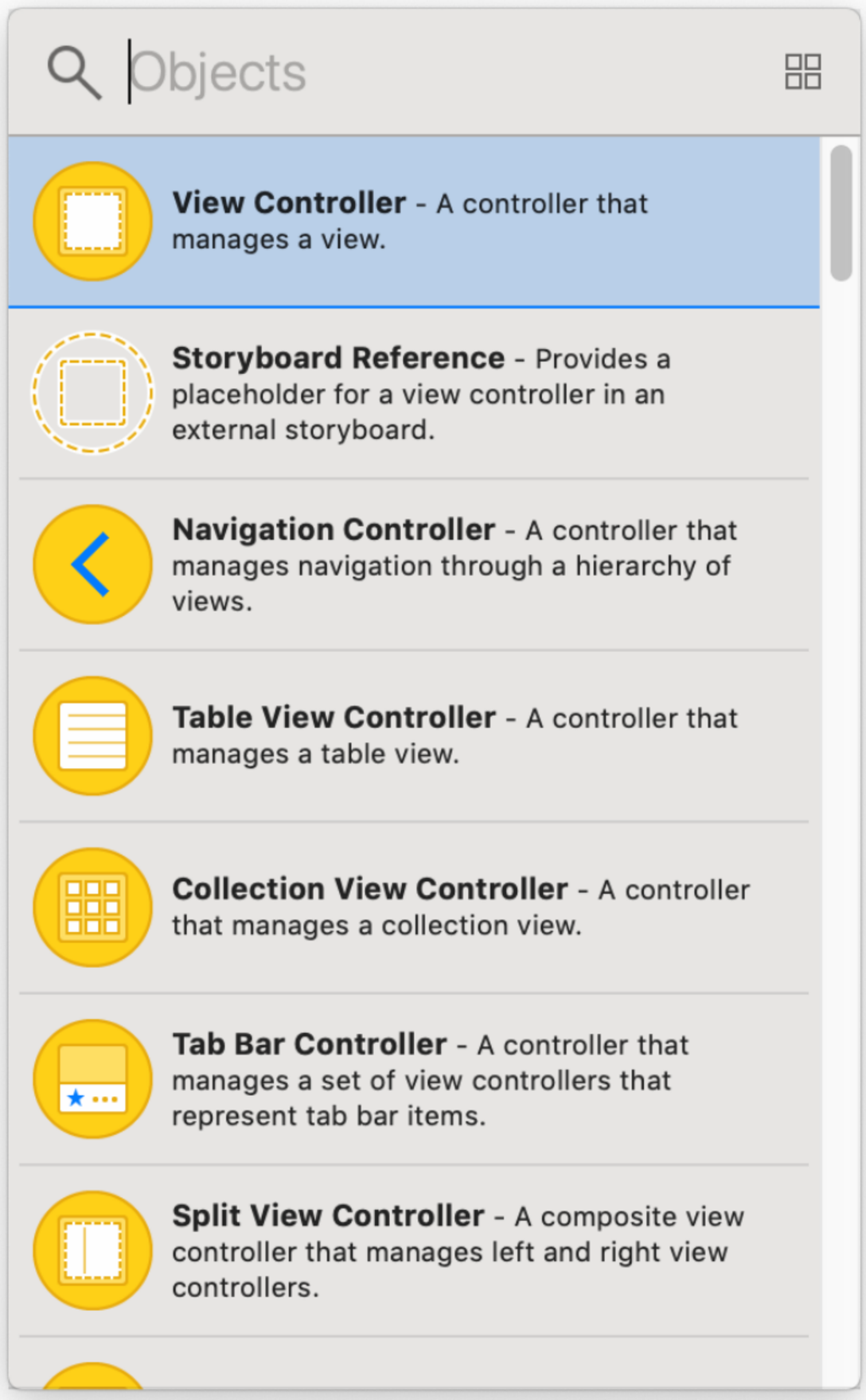
You can change the initial view controller by moving the entry point (the gray arrow) to the left of the desired view controller. Currently in this project, `Main.storyboard` has only one view controller, so you won't be able to change it.

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.

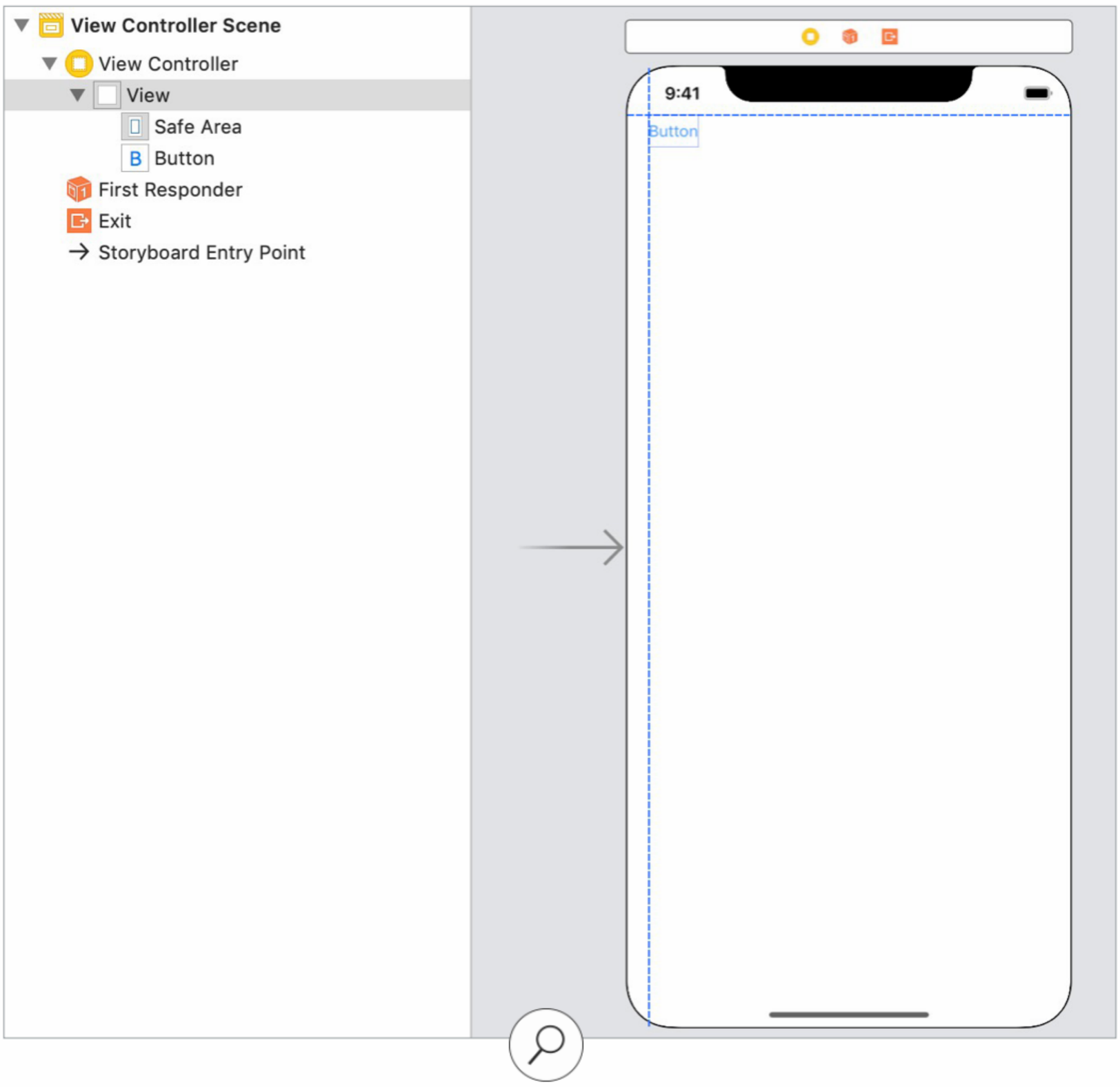


Select the Library button in the Xcode toolbar to display the Object Library. The Object Library contains buttons, text fields, labels, and other view controllers that you can add to your scene.



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.



In addition to the File and Quick Help inspectors (which are always available), the top of the utilities area displays four context-sensitive inspectors when you're in Interface Builder. 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.

The Identity inspector allows you to edit the properties related to an object's identity, such as what class it belongs to. In this example, the button belongs to the `UIButton` class. If you'd defined a custom button class elsewhere, you could use the Identity inspector to change the class of the button you just added. You'll learn more about classes in a future section.

The Attributes inspector provides a list of adjustable visual properties for the selected item. In the case of a `UIButton`, you can change attributes such as the button's text, text color, background, and alignment. Try changing the look of your button.

The Size inspector allows you to adjust the size and position of the selected element within the scene. You'll use the X and Y fields—where the top left of the screen corresponds to (0,0)—to change the position. The X value gets larger as you move the element farther to the right, and Y gets larger as you move down the screen. For `UIButton`, the Size inspector includes additional fields for adjusting the padding around the button's title or image.

The Connections inspector lists all the functions and variable names related to the object. Currently, your button isn't connected to anything, so you won't see any connections. In the next section, you'll learn how to reference the button in code so that it will perform functions when the user interacts with it.

OUTLETS AND ACTIONS

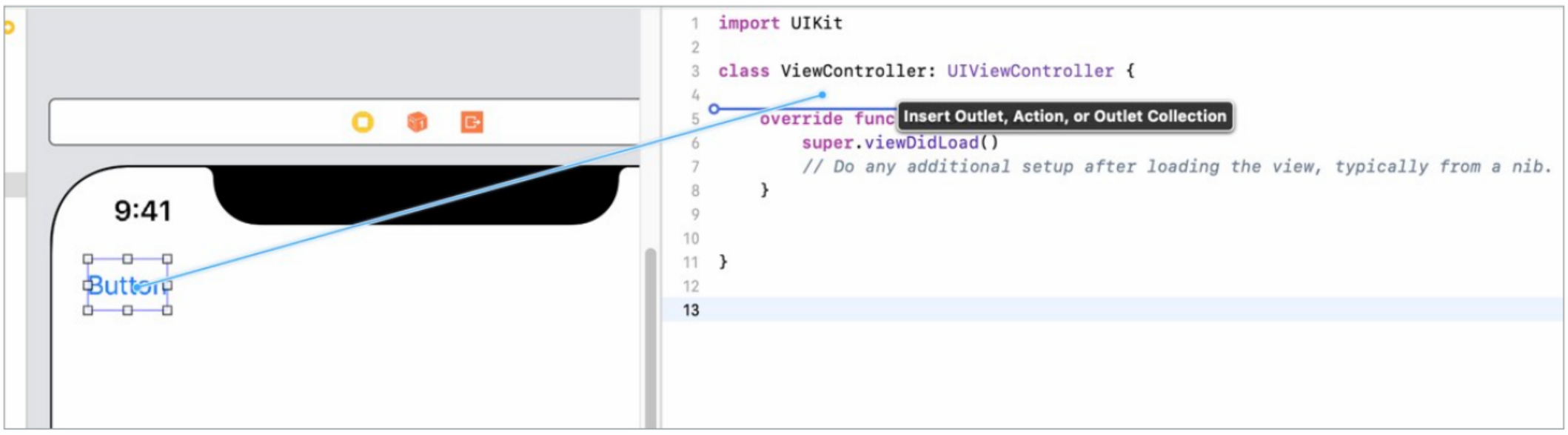
You'll often need a way to reference your visual elements in code so that they can be adjusted at runtime, or when the app is already running. This reference from Interface Builder to code is called an outlet. When you have an object that you want the user to interact with, you create an action—a reference 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 , and you'll see the definition of `ViewController` displayed alongside the storyboard (since 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, which contains the `ViewController` class definition. As you drag the cursor into the code, you'll see a blue line.



When you release the cursor, the "Outlets and Actions" dialog appears. Make sure 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.

```
@IBOutlet var myButton: UIButton!
```

The outlet definition has multiple parts. Here's what you're seeing from left to right:

- Circle — The filled circle indicates that the outlet is connected. The circle would be empty if the variable isn't connected to anything.
- `@IBOutlet` — Unique to Interface Builder, this keyword specifies a relationship between the variable and a storyboard. Without the `@IBOutlet` keyword, the circle doesn't display.
- `var myButton` — This declares a variable called "myButton."
- `: UIButton!` — The type of the property is a `UIButton`, the button type within the `UIKit` framework. The exclamation point warns you that the program will crash if you try to access this property and the outlet isn't connected. More about this in a later lesson.

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

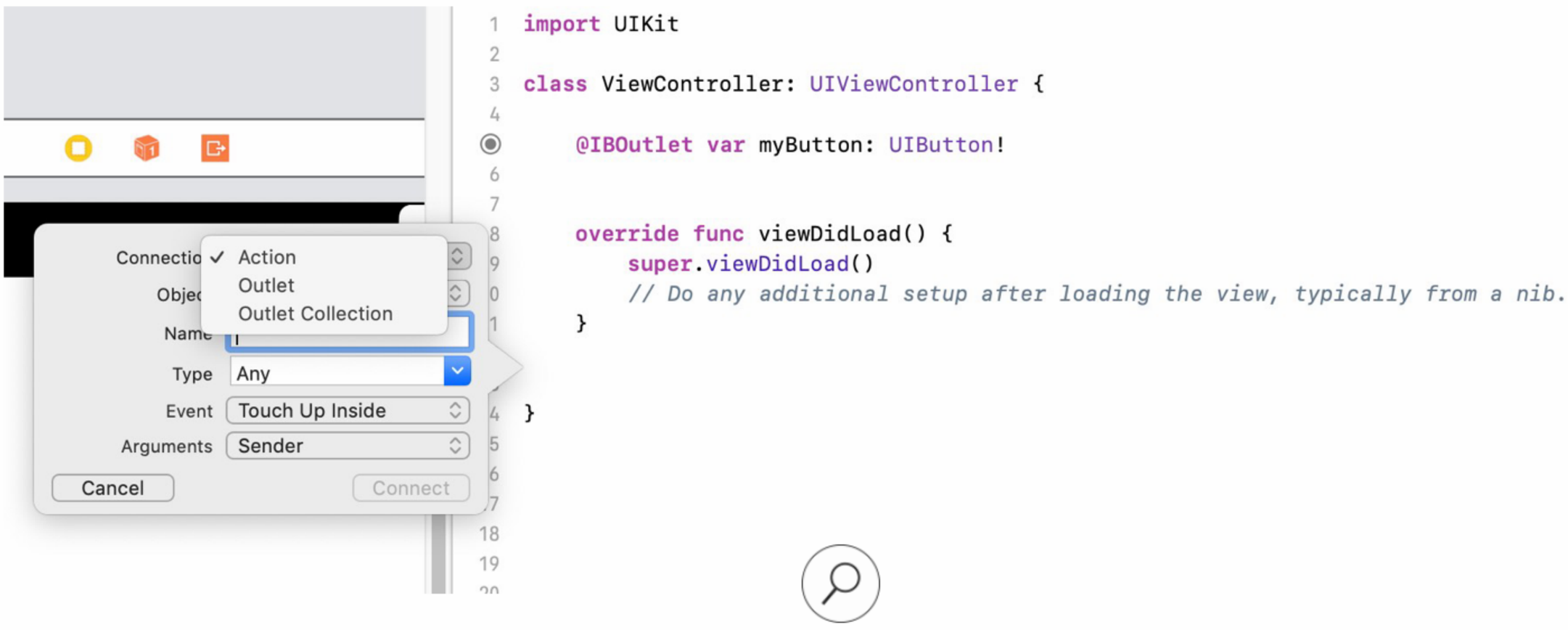
```
myButton.setTitleColor(.red, for: .normal)
```

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. 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.

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

}
```

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

- Circle — The filled circle indicates that the action is connected. The circle would be empty if the function isn't connected to anything in the storyboard.
- `@IBAction` — Unique to Interface Builder, this keyword specifies a relationship between an action and an object in a storyboard. Without the `@IBAction` keyword, the circle doesn't display.
- `func buttonPressed` — This declares a function called "buttonPressed."
- `(_ sender: Any)` — An action can be connected to a number of different objects: buttons, sliders, switches, etc. In this example, `sender` is the name of the object that triggers the action. Because `sender` could represent any of several user interface elements, it's of the `Any` type. You'll learn more about types and `Any` in a later unit.

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

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

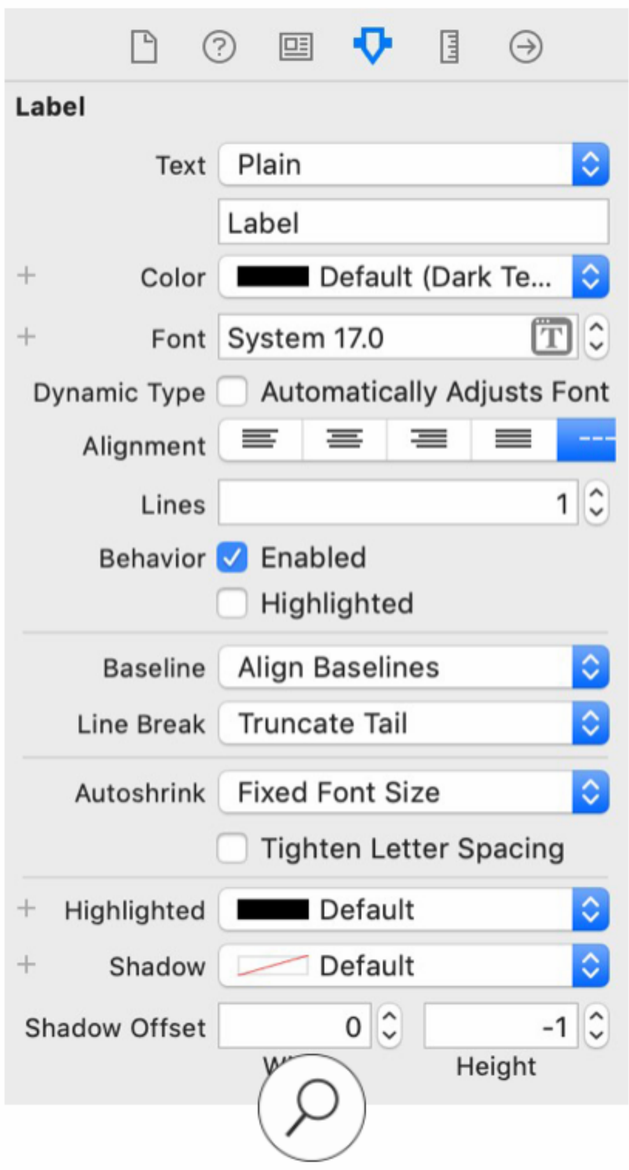
This code prints a message to the Xcode console whenever the function is executed. Build and run your app, then click the button to see the message print in the console at the bottom right of the screen.

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

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, then look at the Attributes inspector for the label.



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	`text`
Color	`textColor`
Font	`font`
Alignment	`textAlignment`
Lines	`numberOfLines`
Enabled	`enabled`
Highlighted	`isHighlighted`
Baseline	`baselineAdjustment`
Line Break	`lineBreakMode`
Autoshrink	`adjustsFontSizeToFitWidth`
Tighten Letter Spacing	`allowsDefaultTighteningForTruncation`
Highlighted	`highlightedTextColor`
Shadow	`shadowColor`
Shadow Offset	`shadowOffset`

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` 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()` function of a view controller, which gives you access to the view controller's main `view` property before it is 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

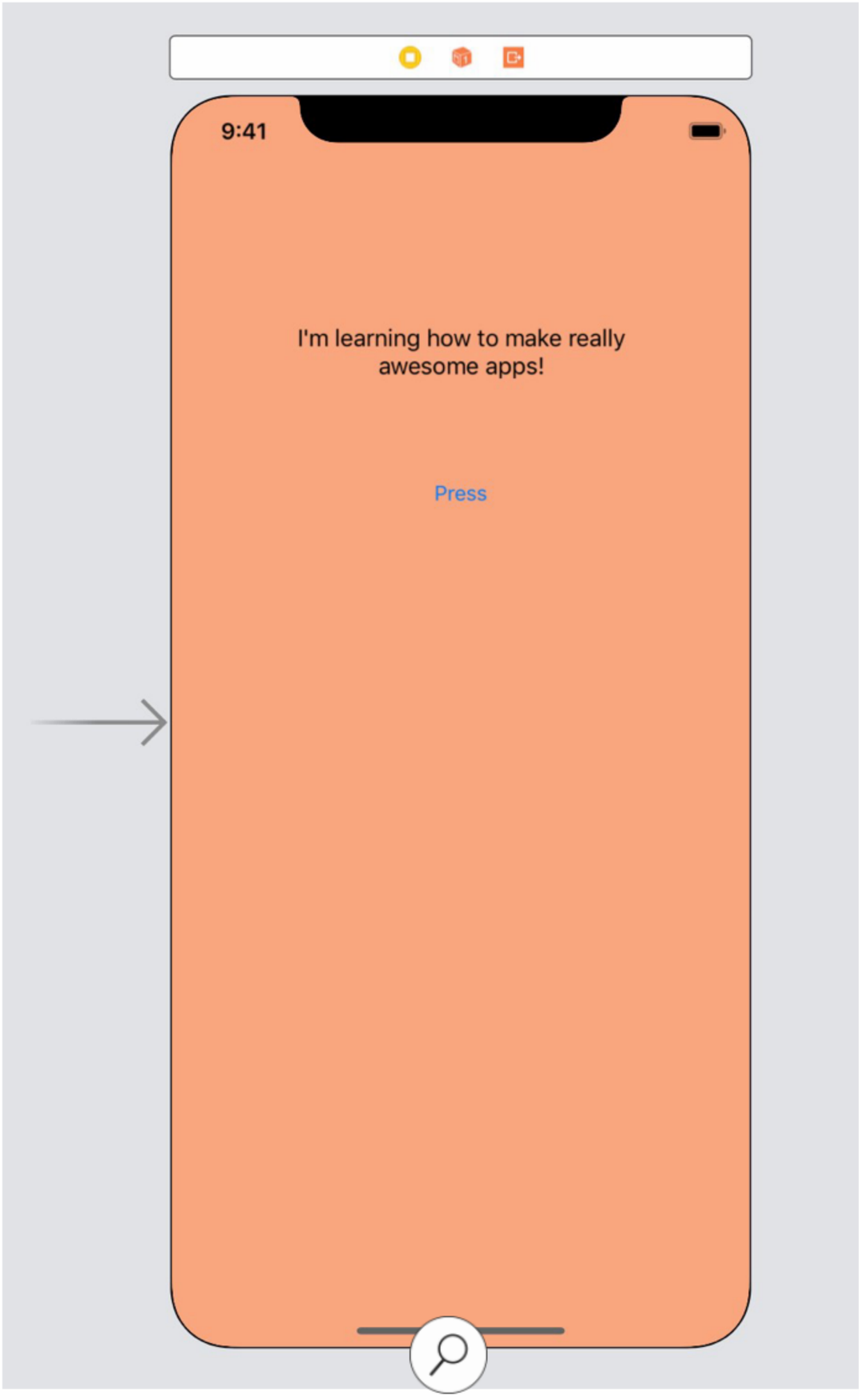
The objective of this lab is to use Interface Builder and the assistant editor to create a basic view.

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 `Main.storyboard` in the project navigator.
- Use what you learned in the lesson to recreate the following image. You'll need to use a label and a button.



STEP 3: USE THE ASSISTANT EDITOR TO CONNECT YOUR VIEW

- Now open the assistant editor and make sure that it's displaying the file named `ViewController.swift`. 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

문제 1(총 10개)

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

정답 확인

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.

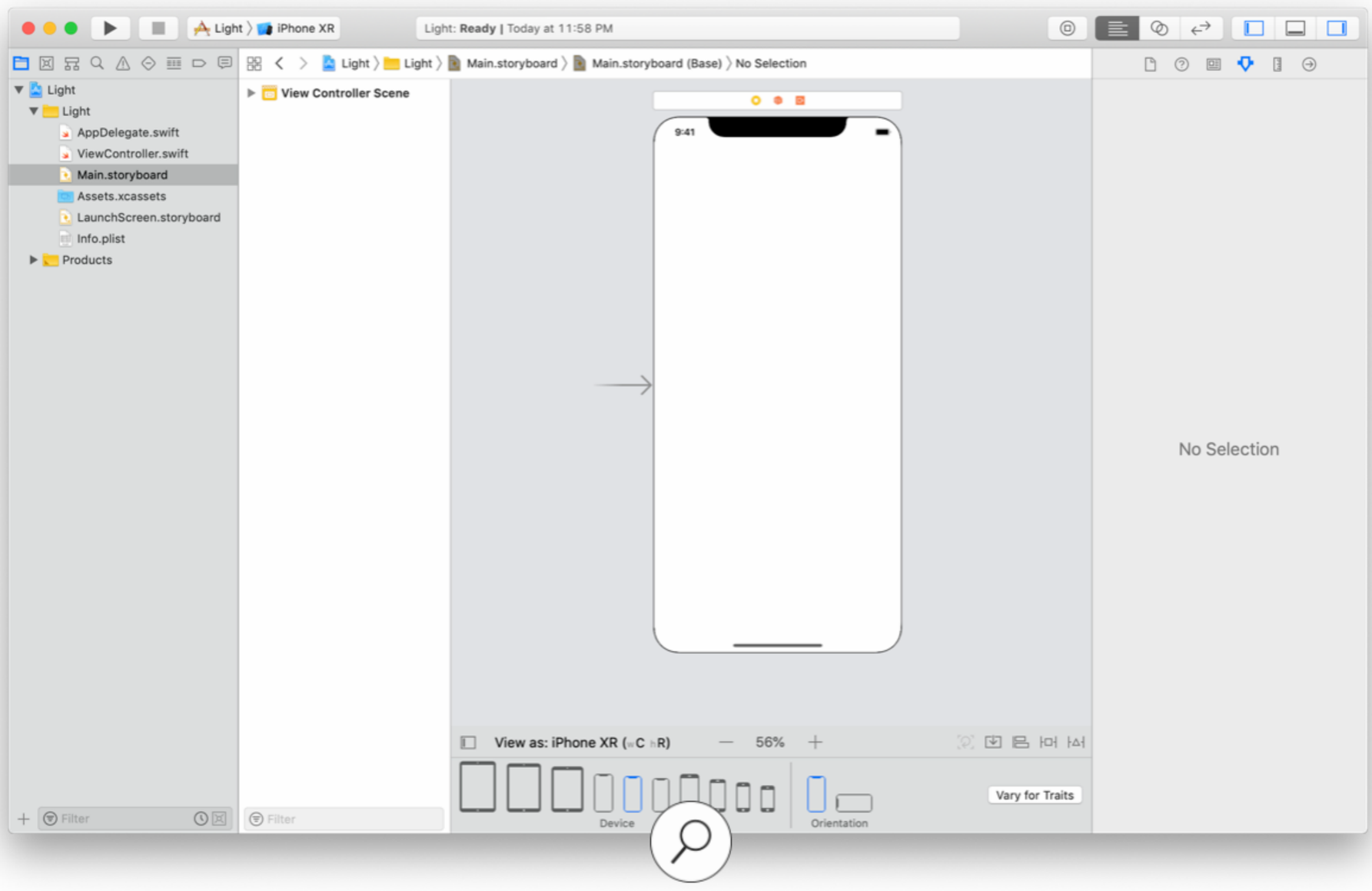
By the end of this Guided Project, you'll have created an app, called Light, that changes the screen from black to white, and back again, whenever the user taps a button. To successfully build the light, you'll need to use Xcode documentation, set breakpoints, and create outlets and actions.

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!

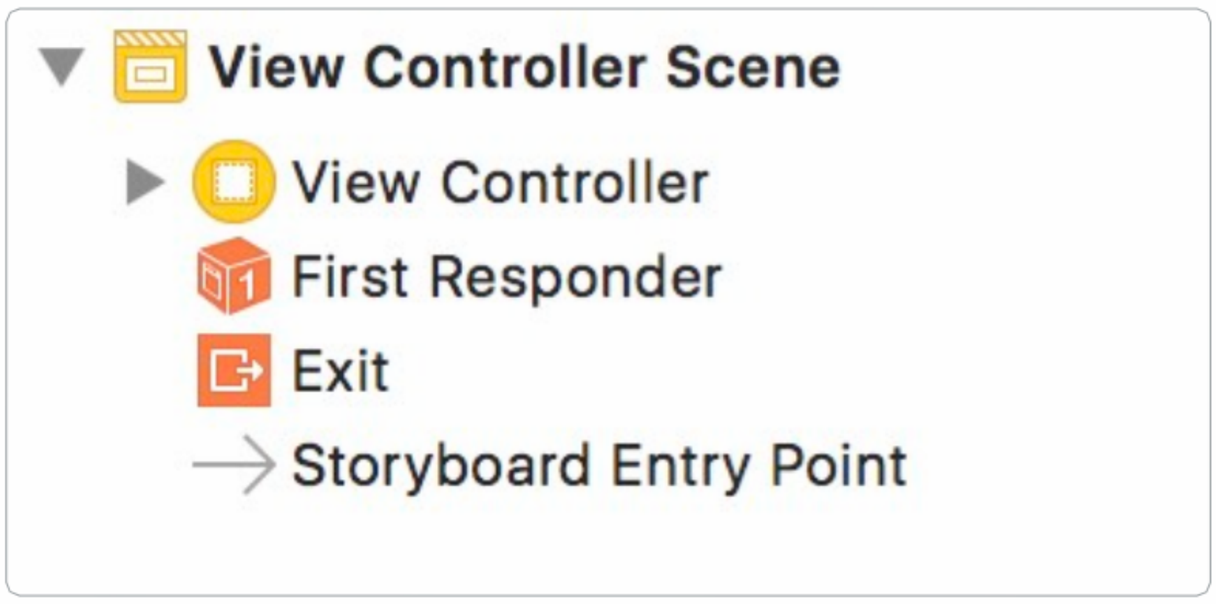
PART ONE—CREATE A BUTTON AND AN ACTION

Create a new Xcode project using the Single View App template that you've used in previous lessons. 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.

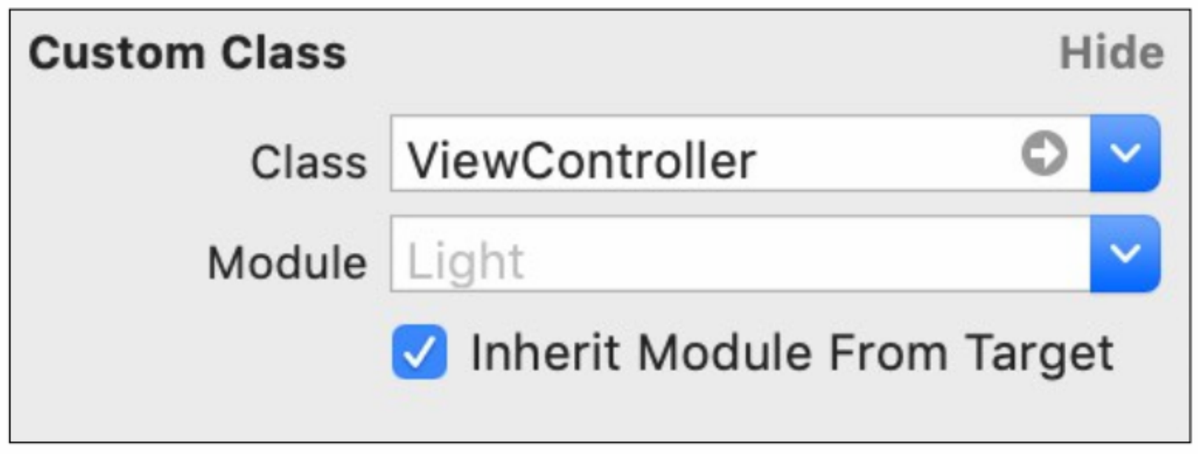
Select `Main.storyboard` in the project navigator to open your storyboard in Interface Builder.



The initial interface created by Xcode includes an instance of `ViewController`. `ViewController` is a subclass of `UIViewController` that comes predefined as part of the Single View 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 `Main.storyboard`.

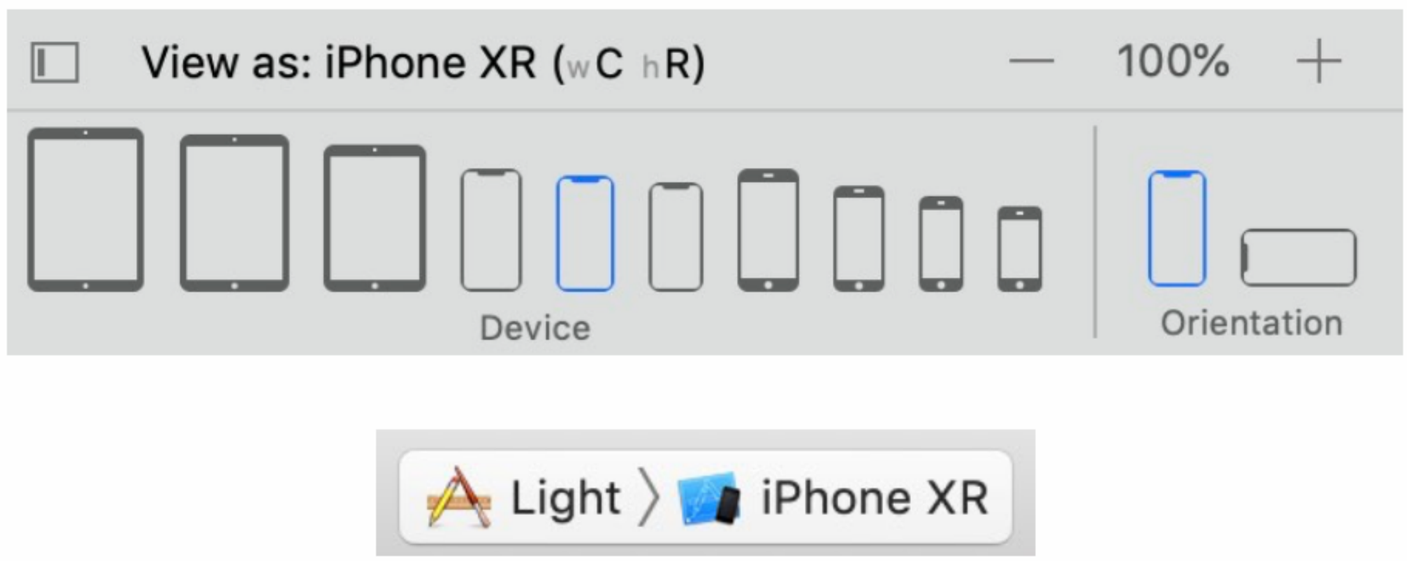


Select View Controller in the document outline, then click the Identity inspector button at the top of the utilities area to confirm that this particular view controller is of type `ViewController`.



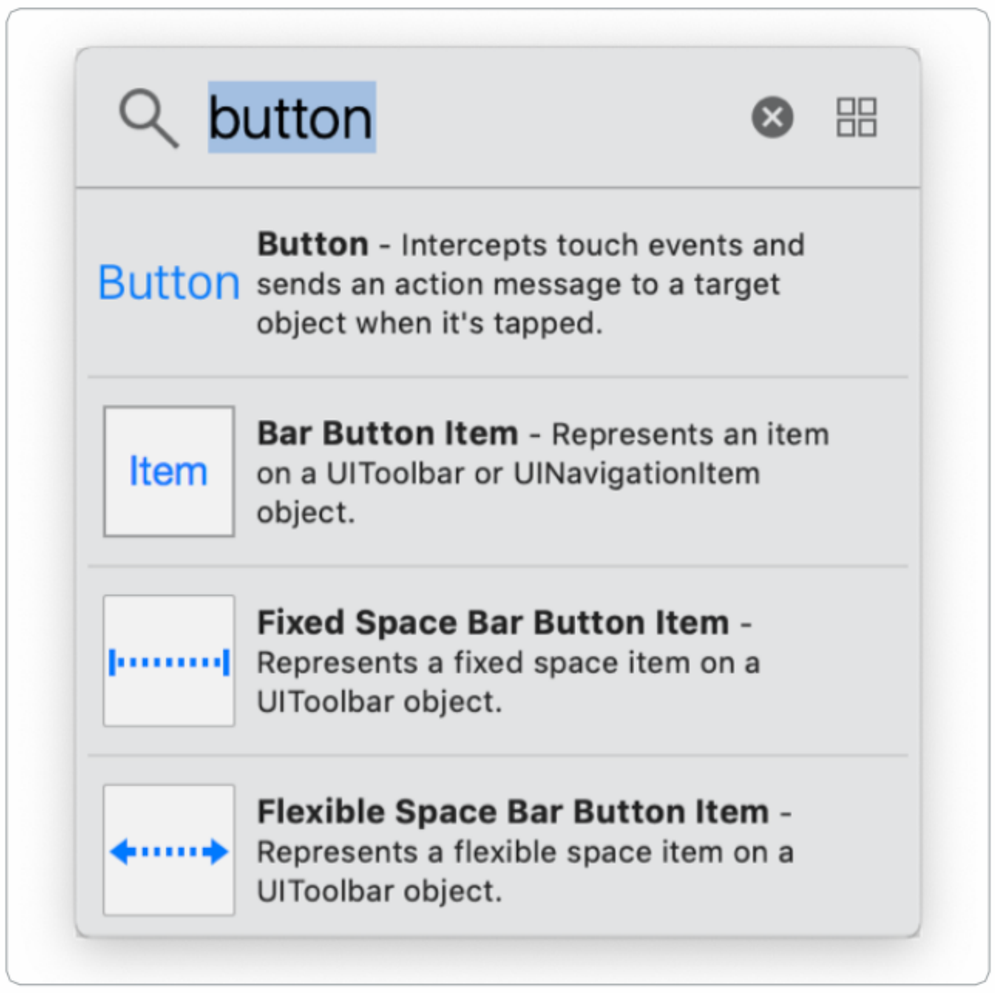
If you're using Simulator, use the "View as" button at the bottom of the canvas to adjust the size of the canvas. Set the canvas size to the iPhone XR configuration, and select the iPhone 8 simulator from the dropdown menu toward the left end of the Xcode toolbar.

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.

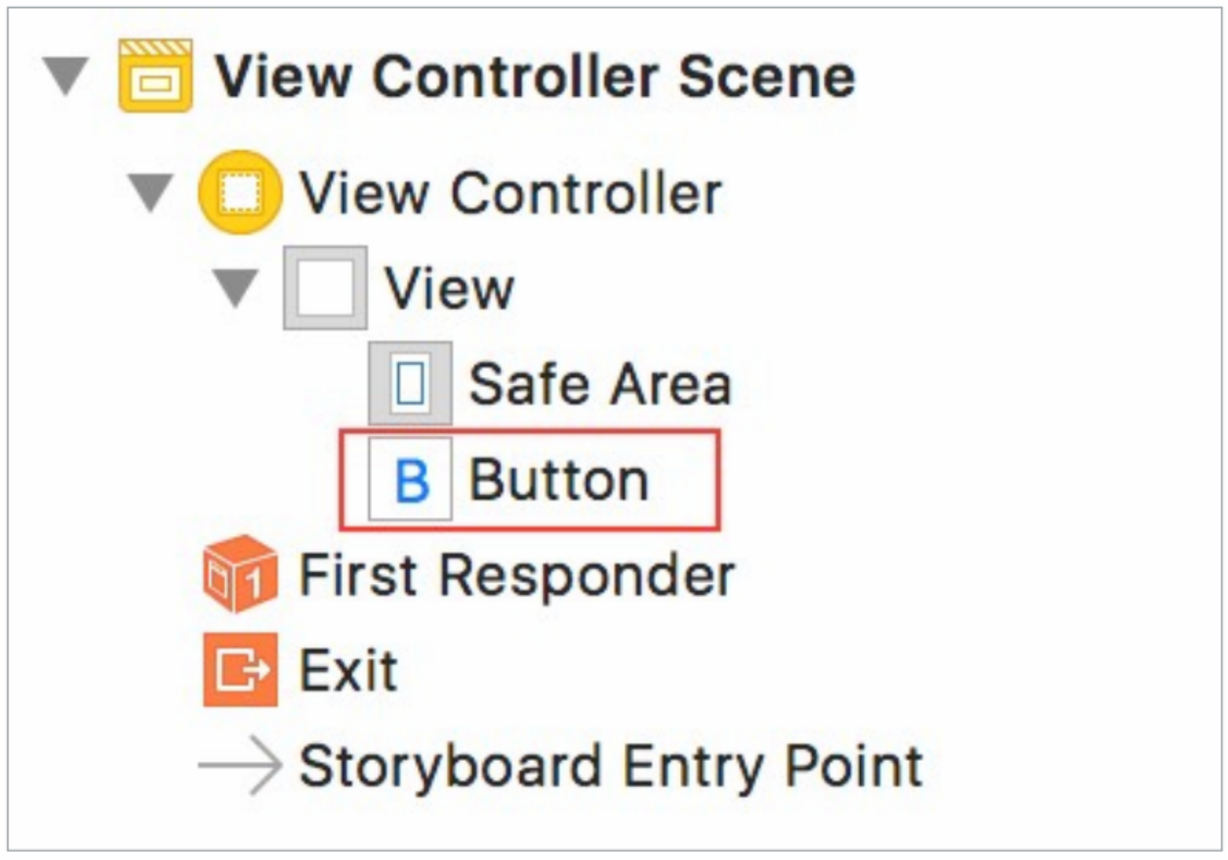


Select the view controller's view, then select the Attributes inspector , which will allow you to customize the attributes of any interface element. The background color for this view has already been set to white. That's the desired state of your app on launch, so you don't need to change anything right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

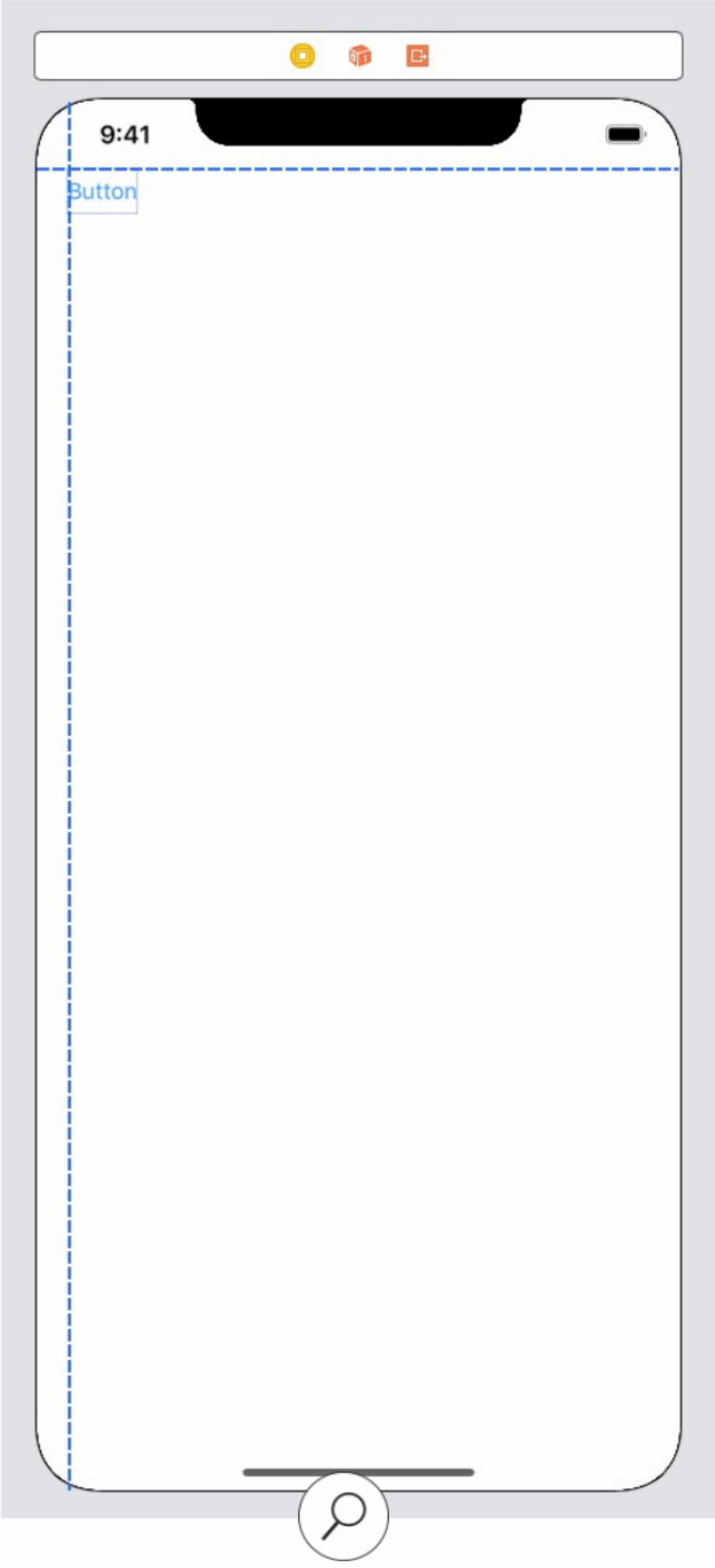
Next, you'll add a button that will be used to change the view's background color, simulating a light turning on and off. Locate the Object library in the bottom-right corner of the utilities panel. As you learned in an earlier lesson, this library includes a list of common objects that can be added to a storyboard. Scroll through the list, or use the search filter at the top of the library area.



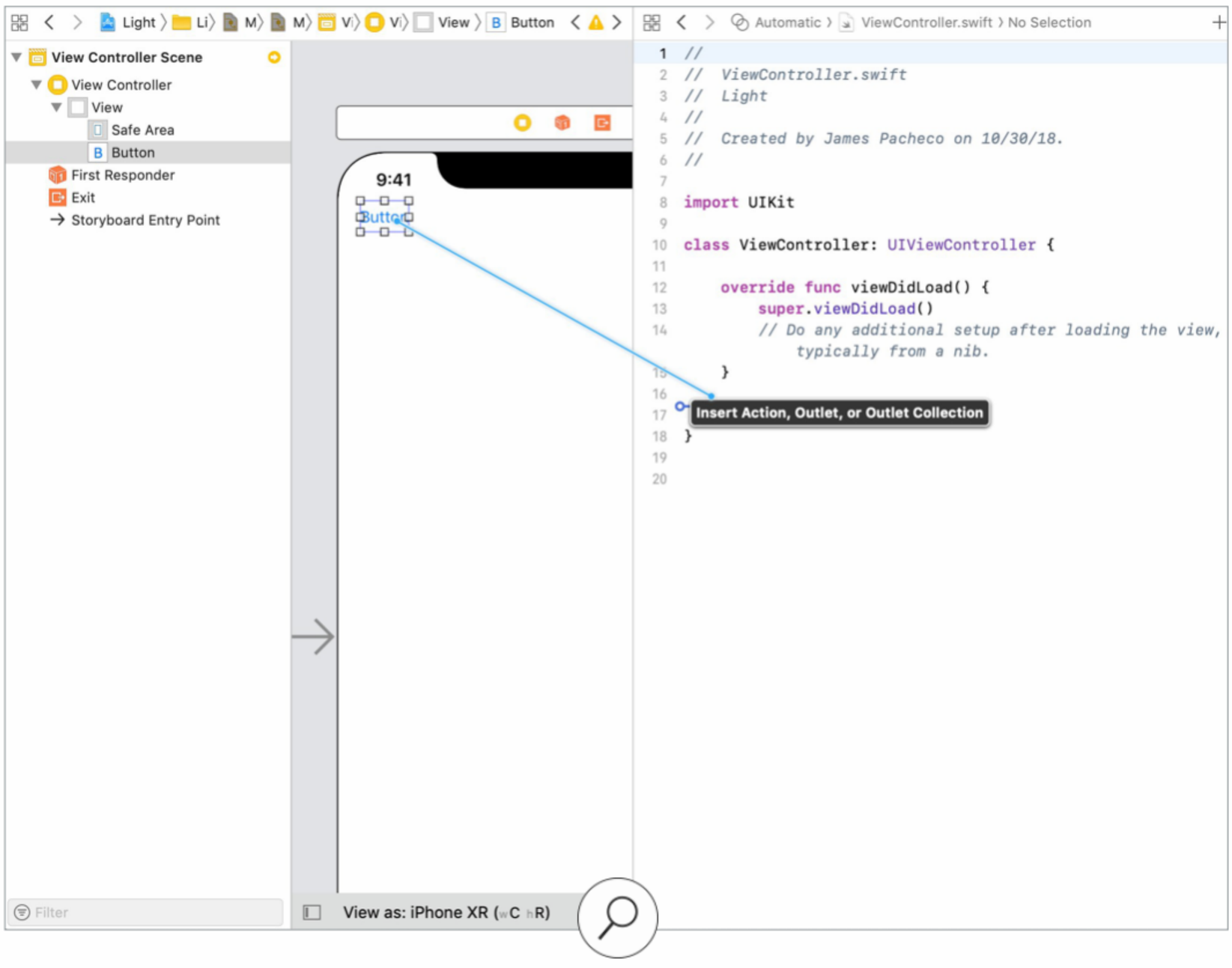
Find the "Button" object in the library and drag it over on top of the view. When you release the cursor, the document outline should indicate that the button has been added as a subview.



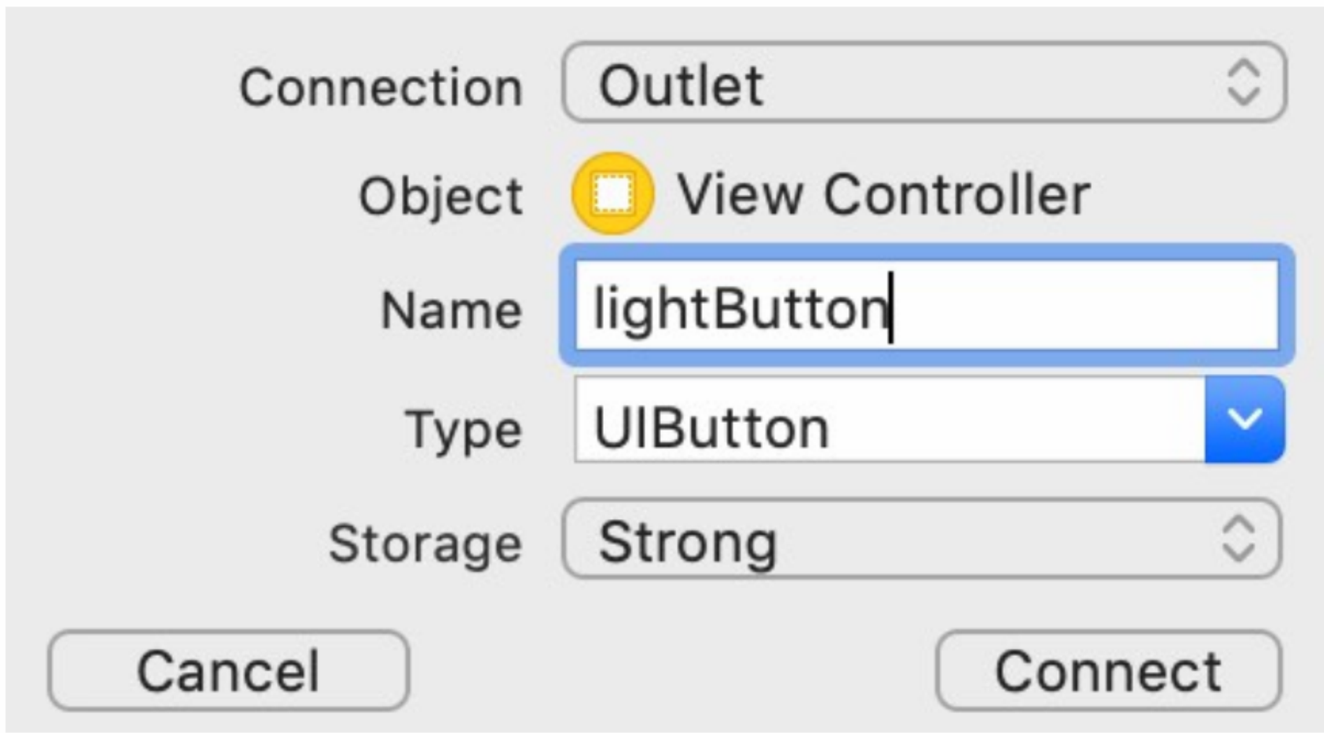
Move the button to the upper-left corner of the view. Margin-alignment guides will help snap the object into the correct position.



Your button now needs to be given an action to perform when it's clicked or tapped. Click the assistant editor button 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. 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". 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.

```
    @IBAction func buttonPressed(_ sender: Any) {
13
14  }
```

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 continues to grow in size, it becomes very important to make sure that your code stays nicely organized. Rather than putting the if-statement directly inside the `buttonPressed(_:)` method, you can move it to a new method that will handle updating the entire user interface. Create a new method called `updateUI()` and move the if-statement inside of it, then call the `updateUI()` method within `buttonPressed(_:)`.

```
@IBAction func buttonPressed(_ sender: AnyObject) {
    lightOn = !lightOn
    updateUI()
}

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

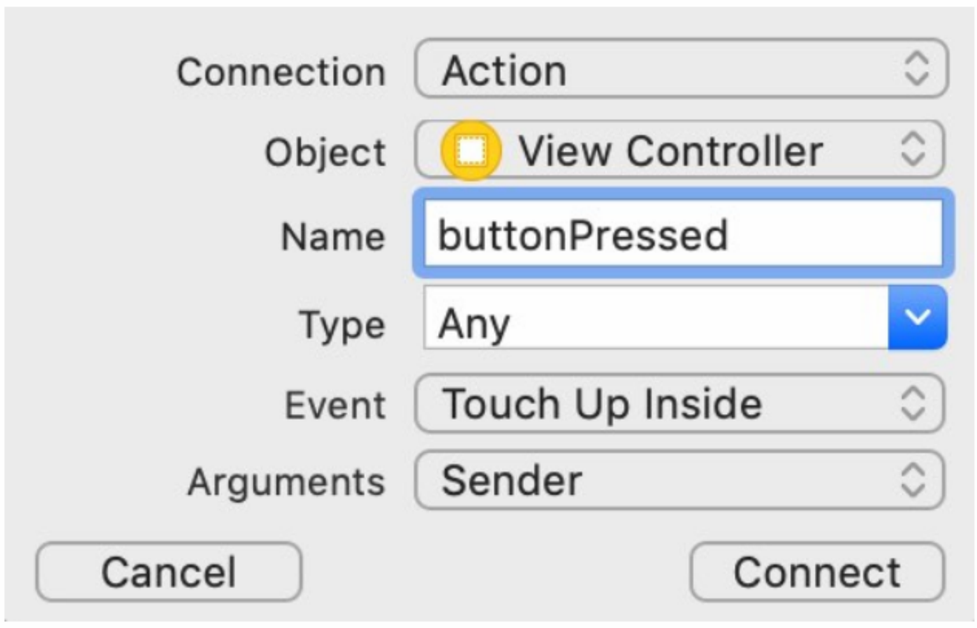
```
}
```

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.

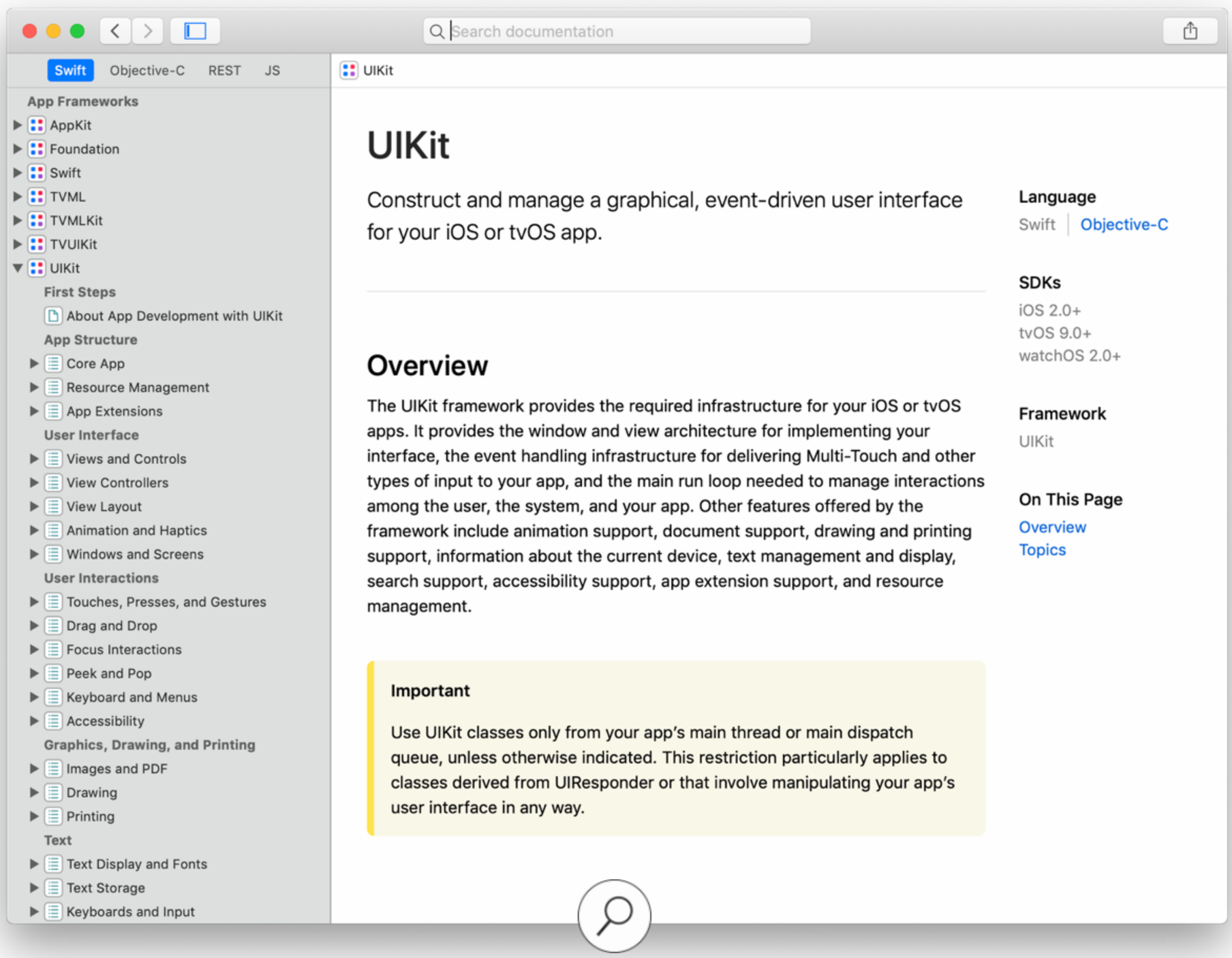
```
15  @IBAction func buttonPressed(_ sender: Any) {
16      lightOn.toggle()                          Thread 1: breakpoint 1.1
17      updateUI()
18  }
```

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

```
19      func updateUI() {
20          if lightOn {
21              view.backgroundColor = .white
22          } else {
23              view.backgroundColor = .black
24          }
25      }
26
27  }
```

Take a moment to try and read the body of the `updateUI()` method. 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."

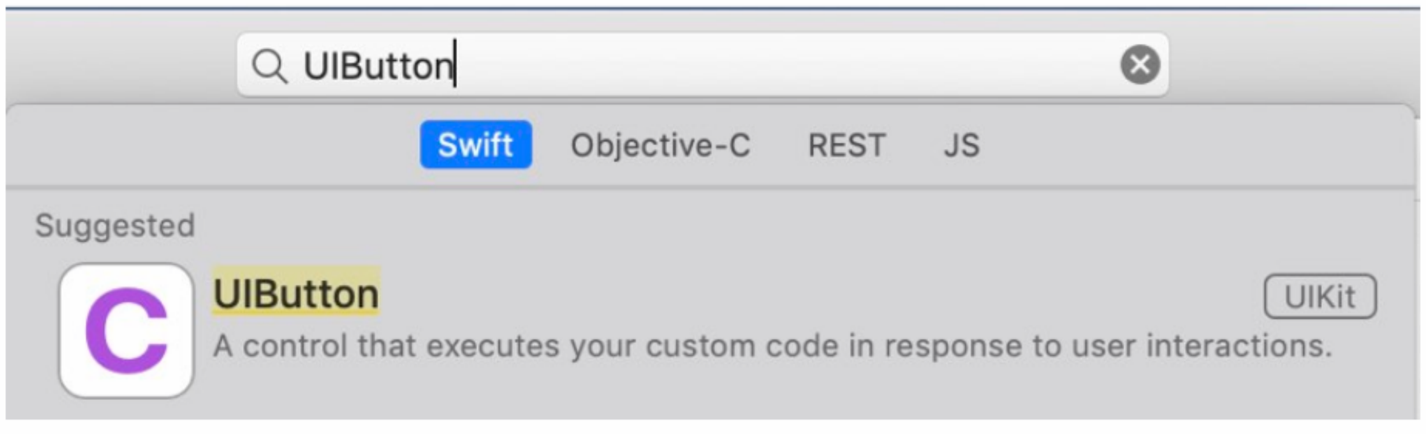
Now consider how the button's title should be decided. If the light is on, the button's title should read "Off," or else the button's title should read "On." It makes sense that the button's title should be updated in a similar way to the view's background color.



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.

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.



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 definition and a list of its parameters. 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.

Declaration

```
func setTitle(_ title: String?, for state: UIControl.State)
```

In this new list of symbols, the documentation defines a `normal` constant, 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.

Topics

Initializers

```
init(rawValue: UInt)
```

Constants

```
static var normal: UIControl.State
```

The normal, or default state of a control—that is, enabled but neither selected nor highlighted.

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 tex 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 code:

```
1  import UIKit
2
3  class ViewController: UIViewController {
4
5      var lightOn = true
       @IBOutlet var lightButton: UIButton!
7
8      override func viewDidLoad() {
9          super.viewDidLoad()
10         updateUI()
11     }
12
       @IBAction func buttonPressed(_ sender: Any) {
14         lightOn.toggle()
15         updateUI()
16     }
17
18     func updateUI() {
19         if lightOn {
20             view.backgroundColor = .white
21             lightButton.setTitle("Off", for: .normal)
22         } else {
23             view.backgroundColor = .black
24             lightButton.setTitle("On", for: .normal)
25         }
26     }
27 }
28
```

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

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.

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

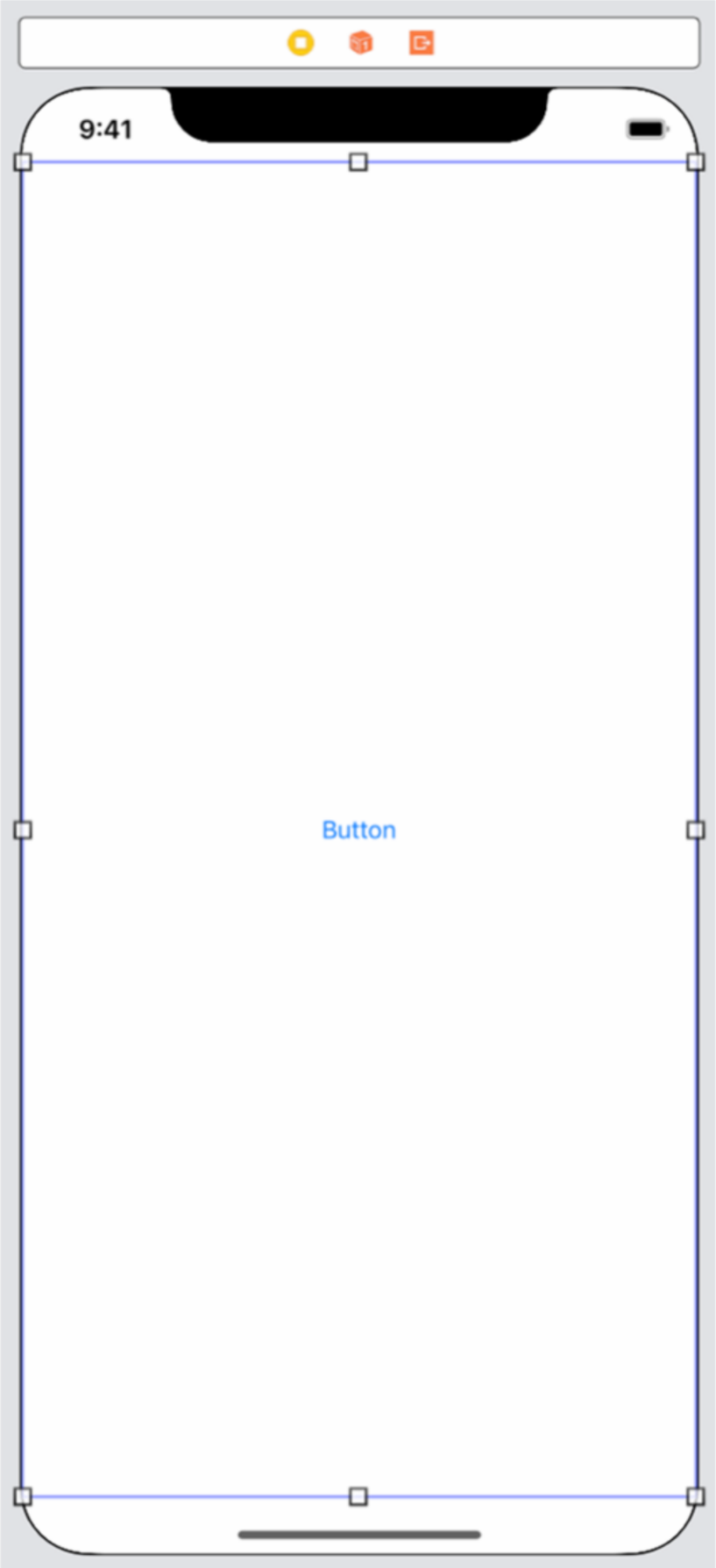
```
func updateUI() {
    if lightOn {
        view.backgroundColor = .white
    } else {
        view.backgroundColor = .black
    }
}
```

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 . Here you can see all the

오후 10:33 7월 18일 토요일 63%

Gallery

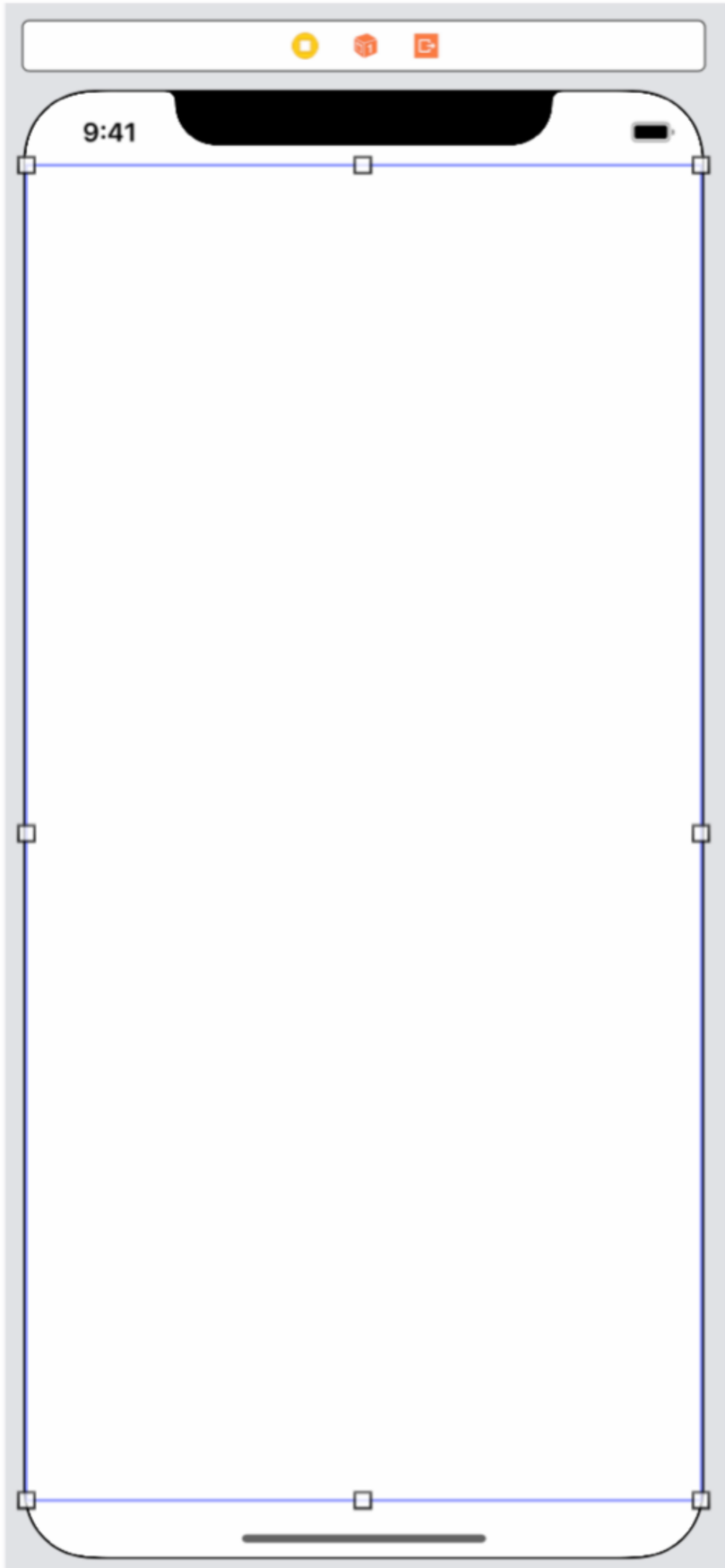


To begin, you'll need to resize the button to fill the screen. Select the button in Interface Builder, then drag its top-left and bottom-right corners, extending the button to cover the entire white view.

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.

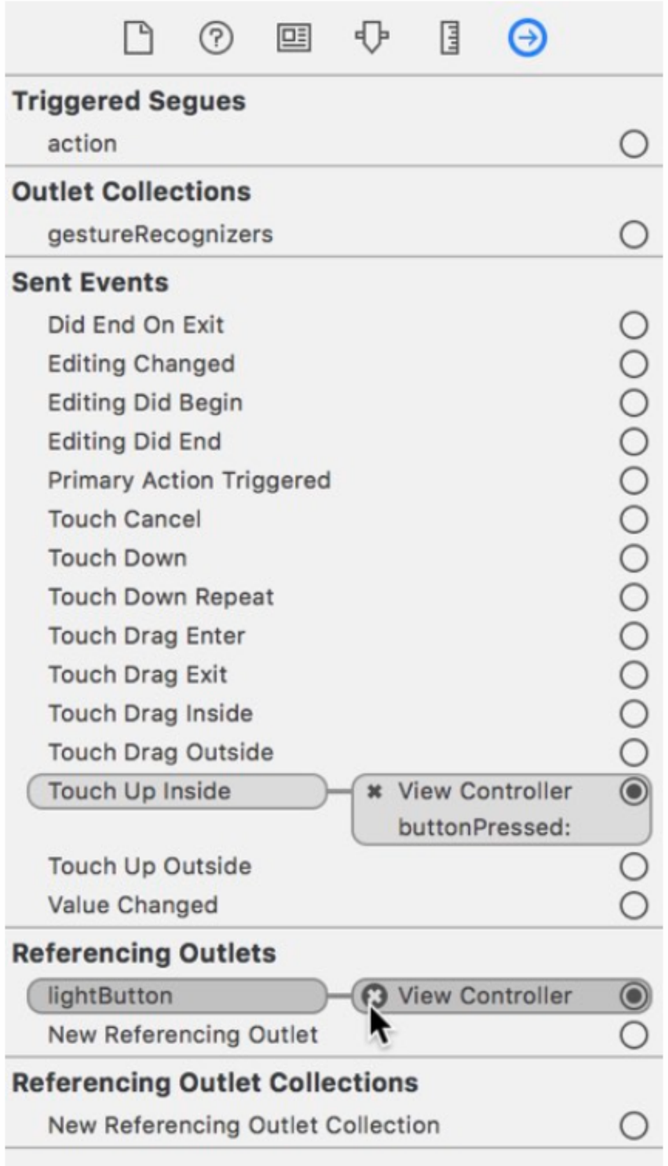
오후 10:33 7월 18일 토요일 63%

Gallery



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.

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.



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

```
○ @IBOutlet var lightButton: UIButton!
```

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.

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

WRAPUP

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, you might want to revisit the previous sections that discussed them in more detail. As you move onto the next unit, remember to keep the developer documentation handy. Developers are only as good as the tools and resources they use.

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 lesson, 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.