Unit 2

Introduction to UIKit

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

This unit will take you many steps further. You'll learn about structures, collections, loops, and different ways to work with the information that makes up an app. At the end of the unit, you'll build an ambitious project. Don't worry if it feels hard—because it is! Every coder in the world went through the same challenges. Keep at it, and you'll start to enjoy solving problems with code.

Swift Lessons

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

SDK Lessons

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

What You'll Build

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

Lesson 2.1

Strings

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

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

What You'll Learn

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

Vocabulary

- case sensitivity
- concatenation
- equality
- escape character
- index
- literal
- range
- string interpolation
- substring
- Unicode

Related Resources

- Swift Programming Language Guide: Strings and Characters

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

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

String literals can be used to provide an initial value to a constant or variable:

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

If you assign a string to a constant, using `let`, the string is immutable and can't be modified. By assigning a string to a variable, you're allowing the string to change.

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

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

Console Output:

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

The whitespace before the closing quotes (`"""`) tells Swift what whitespace to ignore before all of the other lines. Even though both lines of your string literal included whitespace at the beginning, that indentation was stripped off when you printed your string literal since your final three double quotation marks were indented equally as far.

If the string will *include* double quotes, you'll need to use the backslash (`\`), known in Swift as the escape character. That's because you're escaping the string.

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

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

- Double quote: `\"`
- Single quote: `\'`
- Backslash: `\\`
- Tab: `\t`
- Carriage return (return to beginning of the line): `\r`

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

```
var myString = ""
```

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

```
var myString = ""

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

As you might expect, individual characters are of type `Character`. But since strings are much more common in programming than individual characters, Swift will always infer

the type of a collection of characters — or even a single character — as `String`, unless you specify otherwise with a type annotation:

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

CONCATENATION AND INTERPOLATION

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

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

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

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

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

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

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

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

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

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

Console Output:

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

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

STRING EQUALITY AND COMPARISON

Developers often need to compare `String` values to see if they're equal to each other.

Just as you do with numbers, you can check for equality between two strings using the `==` operator. As you might expect, `==` checks for identical characters in the same order. Since uppercase characters aren't identical to their lowercase counterparts, the strings have the same value if the case of each character also matches.

```
let month = "January"
let otherMonth = "January"
```

```
let lowercaseMonth = "january"

if month == otherMonth {
  print("They are the same")
}

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

Console Output:

```
They are the same.
They are not the same.
```

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

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

Console Output:

```
The two names are equal.
```

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

```
let greeting = "Hello, world!"

print(greeting.hasPrefix("Hello"))
```

```
print(greeting.hasSuffix("world!"))
print(greeting.hasSuffix("World!"))
```

Console Output:

```
true
true
false
```

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

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

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

```
let name = "Ryan Mears"
let count = name.count //10
```

```
let newPassword = "1234"

if newPassword.count < 8 {
    print("This password is too short. Passwords should have at
    least eight characters.")
```

```
}
```

Console Output:

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

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

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

Console Output:

```
e is a vowel.
```

MORE ADVANCED STRING TOPICS

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

- `startIndex`
- `endIndex`

- `index(before:)`
- `index(after:)`
- `index(_:, offsetBy:)`
- `insert(_:, at:)`
- `insert(contentsOf:, at:)`
- `remove(at:)`
- `removeSubrange(_:)`
- `replaceSubrange(_:, with:)`

UNICODE

Every Swift `String` adheres to an international computing standard called Unicode. Unicode compliance allows Swift to go beyond the short list of letters and symbols in the English language. Instead, Unicode encompasses over 128,000 different characters used across multiple languages. This includes accents on characters (é), emoji (🐮), symbols (∞), Kanji (七), and other specialized characters. Unicode ensures that the length of these characters are rational, so that "e," "é," and "🐮" each have a correct length of 1. In addition, Unicode supports text that reads right to left, as well as left to right.

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

LAB

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

REVIEW QUESTIONS

문제 1(총 5개)

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

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

정답 확인

Lesson 2.2

Functions

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

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

What You'll Learn

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

Vocabulary

- argument label
- parameter
- return type
- return value

Related Resources

- Swift Programming Language Guide: Functions

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

A function is made up of three things: its name, an optional list of parameters, and an optional return type. Calling a function is similar to making a phone call. You can pass information along (parameters), but you can also get information back (return values). A function in Swift can take zero, one, or multiple parameters, but it can also return zero, one, or multiple values. When a function *does* return a value, you have the option to ignore it.

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

DEFINING A FUNCTION

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

Think about your daily routine. You can probably identify plenty of repetitive tasks you perform each day, such as tying your shoes, making breakfast, and driving to work or school. Some of these tasks require no additional input. You tie your shoes the same way every time. For functions without inputs, there will be no function parameters:

```
tieMyShoes()
```

However, each breakfast may have a list of ingredients. So you'd use parameters to specify the list when you call the function:

```
makeBreakfast(food: ["eggs", "orange juice"])
makeBreakfast(food: ["cereal", "toast"])
```

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

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

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

```
displayPi()
```

Console Output:

```
3.1415926535
```

PARAMETERS

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

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

In the case above, `value` is a constant that can be used within the function.

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

Console Output:

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

To pass multiple arguments to a function, you'll need to separate each parameter with a comma (`,`). The following is an example of a function that takes in two `Int` parameters, multiplies them together, and then prints the result:

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

The function is then called like this:

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

Console Output:

```
The result is 50.
```

ARGUMENT LABELS

So far, the label for each argument within a function has the same name internally as it does externally. Consider the following example, where `firstName` is used both within the function and when the function is called.

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

```
sayHello(firstName: "Aidyn")
```

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

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

The argument labels `to` and `and` make the function read very clearly: "say hello to Luke and Dave". However, check out the implementation of this function:

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

You can probably see that, within the body of the function, `to` and `and` are very poor names. If you want the name of the constant within the function to differ from the argument label, you can specify an external name before the local name:

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

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

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

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

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

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

To omit the argument label, use _ in place of the external name:

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

sayHello("Luke", "Dave")
```

DEFAULT PARAMETER VALUES

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

As part of the function definition, you can provide a default value for any parameter. This way, you can call the function with or without the parameter. If the parameter is unspecified, the function will simply use the default value. You'll need to place all parameters with default values at the end of the list, and these parameters must all have argument labels.

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

display(teamName: "Wombats", score: 100) //"Wombats: 100"
display(teamName: "Wombats") //"Wombats: 0"
```

RETURN VALUES

It's unlikely that you'll always want to print "The result is" before the result from `multiply`. Instead, it might make more sense if the function simply returned the new value. To do so, you'll need to adjust the function declaration to have a return value and

to specify the value's type. You've learned that multiplying two `Int` types will always result in an `Int`, so that's the return type you'll use.

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

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

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

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

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

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

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

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

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

LAB

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

REVIEW QUESTIONS

문제 1(총 3개)

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

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

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

정답 확인

Lesson 2.3

Structures

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

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

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

What You'll Learn

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

Vocabulary

- computed property
- function
- initializer
- initialization
- instance method
- memberwise initializer
- method
- property
- self
- structure
- type

Related Resources

- Swift Programming Language Guide: Classes and Structures

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

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

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

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

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

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

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

Console Output:

```
Jasmine
```

As the name in the example implies, there may be more than one `Person` that gets created throughout the lifetime of your program. The next instance you create could be called `secondPerson`, or you could create a collection called `people` that holds multiple `Person` objects.

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

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

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

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

Console Output:

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

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

INSTANCES

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

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

```
struct Shirt {
  var size: Size
  var color: Color
```

```
}
// Defines the attributes of a shirt.

let myShirt = Shirt(size: .xl, color: .blue)
// Creates an instance of an individual shirt.
let yourShirt = Shirt(size: .m, color: .red)
// Creates a separate instance of an individual shirt.
```

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

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

  func startEngine() {
    print("The \(year) \(make) \(model)'s engine has started.")
}

  func drive() {
    print("The \(year) \(make) \(model) is moving.")
  }

  func park() {
    print("The \(year) \(make) \(model) is parked.")
  }
}

let firstCar = Car(make: "Honda", model: "Civic", year: 2010,
topSpeed: 120)
```

```
let secondCar = Car(make: "Ford", model: "Fusion", year: 2013,
topSpeed: 125)

firstCar.startEngine()
firstCar.drive()

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

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

INITIALIZERS

All Swift types come with an initializer, which is similar to a function that returns a new instance of the type. The default initializer is `init()`.

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

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

But there's a shorthand version of the default initializer that's much more common. The following snippet is more concise, but works the same as the example above:

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

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

Default Values

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

One approach is to provide default property values in your type definition. Each instance will be initialized with that value. This is useful when defining objects that have a consistent default state, such as a 0 reading on an odometer.

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

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

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

Console Output:

```
0
```

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

Memberwise Initializers

Sometimes you'll want to create a new instance with a non-default value. For example, you may need to initialize an `Odometer` with mileage from an earlier trip or from a previous owner. When you define a new structure, Swift creates a special initializer, called a memberwise initializer, that includes all properties of a structure. The memberwise initializer allows you to set initial values for each property of the new instance.

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

Console Output:

```
27000
```

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

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

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

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

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

```
struct Person {
  var name: String

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

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

```
struct Shirt {
  var size: Size
  var color: Color
}

let myShirt = Shirt(size: .xl, color: .blue) // Memberwise
Initializer
```

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

let firstCar = Car(make: "Honda", model: "Civic", year: 2010,
topSpeed: 120) // Memberwise initializer
```

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

Custom Initializers

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

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

```
struct Temperature {
  var celsius: Double
}

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

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

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

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

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

```
struct Temperature {
  var celsius: Double

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

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

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

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

Console Output:

```
18.5
100.0
```

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

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

```
struct Temperature {
  var celsius: Double

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

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

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

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

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

Console Output:

```
18.5
100.0
0
```

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

INSTANCE METHODS

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

As you learned earlier, you add an instance method by adding a function within a type definition. You can then call that function on instances of the type. You may have noticed this syntax in the `Person` or `Car` structures earlier.

Consider a `Size` struct with an instance method `area()` that calculates the area of a specific instance by multiplying its width and height:

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

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

let someSize = Size(width: 10.0, height: 5.5)
let area = someSize.area() // Area is assigned a value of 55.0
```

The `someSize` instance is of the `Size` type, and `width` and `height` are its properties. The `area()` is an instance method that can be called on all instances of the `Size` type.

MUTATING METHODS

Occasionally you'll want to update the property values of a structure within an instance method. To do so you'll need to add the `mutating` keyword before the function.

In the following example, a simple structure stores mileage data about a specific `Car` object. Before looking at the code, consider what data the mileage counter needs to store and what actions it needs to perform.

- Store the mileage count to be displayed on an odometer
- Increment the mileage count to update the mileage when the car drives
- Potentially reset the mileage count if the car drives beyond the number of miles that can be displayed on the odometer.

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

  mutating func increment() {
    count += 1
  }

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

  mutating func reset() {
    count = 0
  }
}

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

The `odometer` instance is of the `Odometer` type, and `increment()` and `increment(by:)` are instance methods that add miles to the instance. The `reset()` instance method resets the mileage count to zero.

COMPUTED PROPERTIES

Swift has a feature that allows a property to perform logic that returns a calculated value.

Consider the `Temperature` example. While most of the world uses the Celsius scale of measurement for temperature, some places (such as the U.S.) use Fahrenheit, and certain professions use Kelvin. So it might be useful for the `Temperature` structure to support all three measurement systems.

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

Imagine that you'd used the memberwise initializer for this structure:

```
let temperature = Temperature(celsius: 0, fahrenheit: 32.0,
kelvin: 273.15)
```

Any time you would write code to initialize a `Temperature` object, you would need to calculate *each* temperature and pass all those values as parameters.

An alternative would be to add multiple initializers that handle the calculations.

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

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

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

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

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

The approach above, using multiple initializers, involves managing a lot of state, or information. Any time the temperature changes, you would be required to update all three properties. This approach is error-prone, and would be frustrating for any programmer.

Swift provides a safer approach. With computed properties, you can create properties that can compute their value based on other instance properties or logic.

```
struct Temperature {
  var celsius: Double

  var fahrenheit: Double {
```

```
    return celsius * 1.8 + 32
  }

  var kelvin: Double {
    return celsius + 273.15
  }
}
```

To add a computed property, you declare the property as a variable (because its value can change). You must also explicitly declare the type. Then you use an open curly brace (`{`) and closing curly brace (`}`) to define the logic that calculates the value to return.

You can access computed properties using dot syntax, just as you would with any other property.

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

Console Output:

```
32.0
273.15
```

The logic contained in a computed property will be executed each time the property is accessed, so the returned value will always be up to date.

PROPERTY OBSERVERS

Swift allows you to observe any property and respond to the changes in the property's value. These property observers are called every time a property's value is set, even if the new value is the same as the property's current value. There are two observer closures, or blocks of code, that you can define on any given property: `willSet`, and `didSet`.

In the following example, a `StepCounter` has been defined with a `totalSteps` property. Both the `willSet` and `didSet` observers have been defined. Whenever `totalSteps` is modified, `willSet` will be called first, and you'll have access to the new value that will be set to the property value in a constant named `newValue`. After the property's value has been updated, `didSet` will be called, and you can access the previous property value using `oldValue`.

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

Here's the output when modifying the `totalSteps` property of a new `StepCounter`:

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

Console Output:

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

TYPE PROPERTIES AND METHODS

You learned that instance properties are data about the individual instance of a type, and instance methods are functions that can be called on individual instances of a type.

Swift also supports adding type properties and methods, which can be accessed or callec on the type itself. Use the `static` keyword to add a property or method to a type.

Type properties are useful when a property is *related* to the type, but not a characteristic of an instance itself.

The following sample defines a `Temperature` struct that has a static property called `boilingPoint`, which is a constant value for all `Temperature` instances.

```
struct Temperature {
  static var boilingPoint = 100
}
```

You access type properties using dot syntax on the type name.

```
let boilingPoint = Temperature.boilingPoint
```

Type methods are similar to type properties. Use a type method when the action is related to the type, but not something that a specific instance of the type should perform.

The `Double` structure, defined in the Swift Standard Library, contains a static method called `minimum` that returns the smaller of its two parameters..

```
let smallerNumber = Double.minimum(100.0, -1000.0)
```

COPYING

If you assign a structure to a variable or pass an instance as a parameter into a function, the values are copied. Separate variables are therefore separate instances of the value, which means that changing one value doesn't change the other.

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

someSize.width = 500

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

Console Output:

```
500
250
```

Note that the `width` property of `someSize` changed to have a value of 500, but the `width` property of `anotherSize` did not, because although we set `anotherSize` equal to `someSize`, this created a copy of `someSize`, and the copy's width did not change when the original width was changed.

SELF

You may have noticed the word `self` in this section. In Swift, `self` refers to the current instance of the object. It can be used within an instance method to point to the current instance of the object.

```
struct Car {
  var color: Color

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

```
    }
}
```

Many languages require the use of `self` to refer to instance properties or methods within the type definition. The Swift compiler recognizes when property or method names exist on the current object, and makes using `self` optional.

This example is functionally the same as the previous example.

```
struct Car {
  var color: Color

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

The use of `self` is required within initializers that have parameter names that match property names.

```
struct Temperature {
  var celsius: Double

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

This concept is called shadowing, and you will learn more about it in a future lesson.

VARIABLE PROPERTIES

Did you notice that all of the structure examples in this lesson used variable properties instead of constants? Consider the `Car` definition used at the beginning of the lesson.

```
struct Car {
  var make: String
  var year: Int
  var color: Color
  var topSpeed: Int
}
```

A car's color could easily be changed with a paint job, and the top speed might get updated if the owner upgrades the engine. But the make and year of a car will never change, so why not define the properties using `let`?

Variable properties provide a convenient way to create new data from old data. In the following code, making a blue, 2010 Ford is as simple as making a copy of the 2010 Honda and modifying the `make` property. If the `make` property were constant, the second car would need to be created using the memberwise initializer.

```
var firstCar = Car(make: "Honda", year: 2010, color: .blue,
topSpeed: 120)
var secondCar = firstCar
secondCar.make = "Ford"
```

Earlier in this lesson, you learned that whenever an instance of a struct is assigned to a new variable, the values are copied. In the previous example, `firstCar` is still a Honda. If you want to explicitly prevent `firstCar`'s properties from ever changing, simply declare the instance using a `let`.

```
let firstCar = Car(make: "Honda", year: 2010, color: .blue,
topSpeed: 120)
firstCar.color = .red //Compiler error!
```

Even if the properties are declared using `var`, the values of a constant cannot be changed. As a general rule, you should use `let` whenever possible to define an instance of a structure, use `var` if the instance needs to be mutated, and use `var` when defining the properties of a structure.

오후 10:34 7월 18일 토요일 62%

LAB

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

REVIEW QUESTIONS

문제 1(총 3개)

When is it useful to define a custom structure?

- **A.** When you want to represent simple data already handled by a Swift type
- **B.** When you want to represent a new type of data with a collection of properties and functions

정답 확인

Lesson 2.4

Classes, Inheritance

You've now learned about structures as a way to compile data and functionality into a type. Many programming languages also support another feature, called classes, that perform similar functionality. In special cases, classes can (and should) be used instead of structures.

Classes and structures are very similar, and either can be used as the building blocks for your program or app. In this lesson, you'll learn what makes classes different than structures and when to use classes instead of structures. You'll also learn about inheritance, superclasses, and subclasses.

What You'll Learn

- The difference between a structure and a class
- How to define a class
- The concept and importance of inheritance
- How to write a class that inherits from another class
- How to use a class to manage complex states in an application

Vocabulary

- base class
- class
- inheritance
- state
- subclass
- superclass

Related Resources

- Swift Programming Language Guide: Classes and Structures
- Swift Programming Language Guide: Inheritance

Classes are very similar to structures. Among other similarities, both can define properties to store values, define methods to provide functionality, and define initializers to set up the initial state. The syntax for doing these things is almost always identical.

You define a class using the `class` keyword along with a unique name. You then define properties as part of the `class` by listing the constant or variable declarations with the appropriate type annotation. As with structures, it's best practice to capitalize the names of types and to use lowercase for the names of properties:

```
class Person {
  let name: String

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

You can add functionality to a class by adding functions to the class definition:

```
class Person {
  let name: String

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

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

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

```
person.sayHello()
```

Console Output:

```
Jasmine
Hello, there!
```

You've seen an almost-identical example in the previous lesson on structures. But how do classes differ from structures?

INHERITANCE

The biggest difference between structures and classes is that classes can have hierarchical relationships. Any class can have parent and child classes. A parent class is called a superclass, and a child class is called a subclass.

Just as in family relationships, subclasses inherit properties and methods from superclasses. However, subclasses can augment or replace the implementation of superclass properties and methods.

In the sections below, you'll read how to define a base class, which is a class that has no parent classes, how to define subclasses, and how to override properties or methods from the base class.

Defining a Base Class

A class that doesn't inherit from a superclass is known as a base class. All the classes that you've seen so far are base classes.

Here's an example of a base class for a `Vehicle` object:

```
class Vehicle {
    var currentSpeed = 0.0

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

    func makeNoise() {
```

```
        // do nothing - an arbitrary vehicle doesn't necessarily
        make a noise
    }
}

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

Console Output:

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

The `Vehicle` class has a property called `currentSpeed` with a default value of `0.0`. The `currentSpeed` property is used in the computed `description` variable that returns a human readable description of the vehicle. The class also has a method called `makeNoise()`. This method does not actually do anything for a base `Vehicle` instance, but will be customized by subclasses later.

The `Vehicle` class defines common characteristics, like properties and methods, for any type of vehicle–like cars, bicycles, or airplanes. It is not very useful on it's own, but it will make it easier to define other more specific types.

Create a Subclass

Subclassing is the act of basing a new class on an existing class. The subclass inherits properties and methods from the superclass, which you can then refine and make more specific. You can also add new properties or methods to the subclass.

To define a new type that inherits from a superclass, write the type name before the superclass name, separated by a colon:

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

The following example defines a subclass called `Bicycle`, with a superclass of `Vehicle`:

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

The new `Bicycle` class automatically inherits all of the properties and methods of `Vehicle`, such as its `currentSpeed` and `description` properties and its `makeNoise()` method.

In addition to the inherited properties and methods, the `Bicycle` class defines a new boolean property `hasBasket`. By default, new instance of `Bicycle` will not have a basket. You can set the `hasBasket` property to `true` for a particular `Bicycle` instance after it has been created, or within a custom initializer for the type.

```
let bicycle = Bicycle()
bicycle.hasBasket = true
```

As you'd expect, you can also update the `currentSpeed` property, which will impact the description of the instance.

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

Console Output:

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

Subclasses can themselves be subclassed. For example, you can create a class that represents a two-seater `TandemBike`.

```
class Tandem: Bicycle {
```

```
    var currentNumberOfPassengers = 0
}
```

`Tandem` inherits all of the properties and methods from `Bicycle`, which in turn inherits all of the properties and methods from `Vehicle`. The `Tandom` subclass also adds a new property called `currentNumberOfPassengers`, with a default value of `0`.

If you create a new instance of `Tandem`, you'll have access to all of the inherited properties and methods.

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

Console Output:

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

Override Methods and Properties

One advantage of subclassing is that each subclass can provide its own custom implementation of a property or method. To override a characteristic that would otherwise be inherited, like writing a new implementation for a function, you prefix your new definition with the `override` keyword.

The following example defines a new `Train` class, which overrides the `makeNoise()` method that `Train` inherits from `Vehicle`:

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

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

Console Output:

```
Choo Choo!
```

You can also override properties by providing a getter, or a block of code that returns the value, like a computed property. The following example defines a new class called `Car`, which is a subclass of `Vehicle`. The new class has a new `gear` property, with a default value of `1`. The `Car` class also overrides the `description` property to provide a custom description that includes the current gear:

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

Notice that the new implementation accesses the description from the superclass by calling `super.description`. The new implementation then adds some extra text onto the end of the description to provide information about the gear.

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

Console Output:

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

Override Initializer

Suppose you have a `Person` class with a `name` property. The initializer sets the `name` to the parameter specified.

```
class Person {
  let name: String

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

Now imagine you want to create a `Student`, which is a subclass of `Person`. Each `Student` includes an additional property, `favoriteSubject`.

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

If you try to compile this code, `Student` will fail, because you haven't provided an initializer that sets the `favoriteSubject` property to an initial value. Since the `Person` superclass already does the work of initializing the `name` property, the `Student` initializer can call the superclass' initializer using `super.init()`. You should call this initializer after you've provided values for any properties added within your subclass.

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

  init(name: String, favoriteSubject: String) {
    self.favoriteSubject = favoriteSubject
    super.init(name: name)
  }
}
```

By calling the superclass' initializer, the `Student` class doesn't have to duplicate the work that was already written in the `Person` initializer.

REFERENCES

A special feature of classes is their ability to reference values assigned to a constant or variable. When you create an instance of a class, Swift picks out a region in the device's memory to store that instance. That region in memory has an address. Constants or variables that are assigned to that instance store that address to refer to the instance.

So the constant or variable does not contain the value itself, it points to the value in memory.

When you assign a class to multiple variables, each variable will reference, or point to, the same address in memory. So if you update one of the variables, both variables will be updated.

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

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

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

jack.age += 1

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

Console Output:

```
25
25
```

In contrast, when you create an instance of a structure, you're assigning a literal value to that variable. If you create a variable that's equal to another structure variable, the value is copied. Any operations you perform on either variable will be reflected only on the modified variable.

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

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

jack.age += 1

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

Console Output:

```
25
24
```

MEMBERWISE INITIALIZERS

You've learned that all property values must be set when you create an instance of a class Unlike for structures, Swift does *not* create a memberwise initializer for classes. However, it's common practice for developers to create their own memberwise initializer for classes they define, ensuring that all initial values are set for each instance of any object.

CLASS OR STRUCTURE?

Because classes and structures are so similar, it can be hard to know when to use classes and when to use structures for the building blocks of your program.

As a basic rule, you should start new types as structures until you need one of the features that classes provide.

Start with a class when you're working with a framework that uses classes or when you want to refer to the same instance of a type in multiple places.

Working with Frameworks That Use Classes

You'll often work with resources that you didn't write, such as frameworks like `Foundation` or `UIKit`. In these situations, you may start with a base class, then create a subclass to add your own specific functionality. For example, when you're working with `UIKit` and want to create a custom view, you create a subclass of `UIView`.

When working with frameworks, it's often an expectation that you'll pass around class instances. Many frameworks have method calls that expect certain things to be classes. So in these cases, you'll always choose to use a class over a structure.

Stable Identity

There are times when you want to use a single instance of an object in multiple places, but it doesn't make sense to copy the instance. Because each class constant or variable is an address that points to the same data, the data has a *stable identity*.

Consider a `UITableViewCell` subclass `MessageCell` that represents a row in a table view. The cell is designed to display information about an email message. Each instance of a `MessageCell` will be accessed in many places in code, as you will see.

```
class MessageCell: UITableViewCell {

  func update(message: Message) {
    // Update `UITableViewCell` properties with information
    about the message
    textLabel.text = message.subject
    detailTextLabel.text = message.previewText
  }
}
```

When a table view is preparing to display cells, it calls a function `cellForRow(at: IndexPath)` to request the cell it should display at each position in the list. All cells for a table view are initialized (and put into memory) during this function call.

```
override func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {

  let cell = MessageCell(style: .default, reuseIdentifier:
  "MessageCell")
  cell.update(message: message)

  // Returns the cell to the table view that called this method
  to request it
  return cell
}
```

When the user scrolls through the list of messages in the table view, a method is called that allows your program to set up, update, or modify the cell as it is about to be displayed. The function passes a reference to the cell as a parameter.

```
override func tableView(_ tableView: UITableView, willDisplay
cell: UITableViewCell, forRowAt indexPath: IndexPath) {
  // Perform any operations you want to take on the cell here
}
```

There's some unfamiliar syntax here, which you'll learn more about later. For now, focus on the fact that the function is called for each cell that's about to appear on the table view. Also notice that the function passes three parameters: a reference to the `tableView`, which already exists; the `cell` that already exists but is about to be displayed; and something called `indexPath`.

Because `tableView` and `cell` are both classes, the parameters are references to the `tableView` and `cell` in memory. If you update the `cell` parameter, you're updating the

same cell that was initialized in the `cellForRow` function, the same cell that's about to be displayed.

Consider how a class works differently from a structure. If `cell` were a structure, or a value type, the parameter would be a *new copy* of the cell that's about to be displayed. So if you updated `cell`, you wouldn't see your changes, because you updated the copy and not the cell that's about to be displayed.

Having a hard time understanding references in the context of stable identity? That's OK. You'll learn more in future lessons, when you'll need to use these concepts to complete a project.

LAB

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

REVIEW QUESTIONS

문제 1(총 3개)

Which of the following statements are true about these classes?

```
class Scientist {}
class Geologist: Scientist {}
class Physicist: Scientist {}
class Astrophysicist: Physicist {}
```

- **A.** `Scientist` is a base class.
- **B.** There is only one subclass of `Scientist`.
- **C.** `Geologist`, `Physicist`, and `Astrophysicist` are all descendants of `Scientist`.
- **D.** `Astrophysicist` does not inherit from `Scientist`.

정답 확인

Lesson 2.5

Collections

So far in this course, you've learned to work with single items, whether constants or variables. But sometimes you'll find that it makes more sense to work with a group of objects as a whole. A collection allows you to reference multiple objects at once. For example, you could handle a dozen eggs using one constant instead of managing twelve separate constants. In this lesson, you'll learn about the various collection types available in Swift and how to choose the appropriate one for your program.

What You'll Learn

- How to declare constant and variable collections
- How to add and remove values from arrays and dictionaries
- How to choose the appropriate collection type

Vocabulary

- array
- dictionary

Related Resources

- Swift Programming Language Guide: Collection Types

Swift defines two collection types you will frequently work with: arrays and dictionaries. Each type provides a unique method for interacting with multiple objects. As you progress through learning the different types, you'll notice that they share certain functionality: adding/removing items, accessing individual items, and providing type information about the data within the collection.

ARRAYS

The most common collection type in Swift is an array, which stores an ordered list of same-typed values. When you declare an array, you can specify what type of values will be held in the collection, or you can let the type inference system discover the type.

An array is often initialized using a literal, similar to an `Int` or `String` literal. To declare an array literal, surround the collection of values with brackets, with commas separating the values:

```
[value1, value2, value3]
```

You'll use similar syntax to declare an array that stores strings. Notice in the following example that the variable's type, `String`, is surrounded by brackets:

```
var names: [String] = ["Anne", "Gary", "Keith"]
```

Since type inference can determine that `"Anne"`, `"Gary"`, and `"Keith"` are strings, you could actually skip a step and declare the array without specifying the array's type:

```
var names = ["Anne", "Gary", "Keith"]
```

But there might be situations when you want to specify the array's type even though type inference can discover it. Imagine, for example, you want a collection of 8-bit integers (numbers between -127 and 128). You begin by assigning a variable to the following collection of numbers:

```
var numbers = [1, -3, 50, 72, -95, 115]
```

Swift will infer all the values to be of type `Int` and set `numbers` to be an array of integers, or `[Int]`. While this inference is certainly correct, there's a problem: An `Int` can hold positive and negative numbers that exceed beyond the range from -127 to 128. To help Swift understand that you want to restrict the array to smaller integers, you can specify the type as `[Int8]`, an array of 8-bit integers whose values are restricted to the aforementioned range:

```
var numbers: [Int8] = [1, -3, 50, 72, -95, 115]
```

What if you tried to include a larger number, such as 300, in an array literal of type `[Int8]`? Swift will return an error because 300 is greater than the maximum number for an `Int8`. You should specify the type if the inferred type is not specific enough, and it will help to restrict the values that you don't want to allow.

It's often useful to check if a certain value exists in an array. To do so, you can use the `contains(_:)` method, passing in the desired value. If the array contains the value, the expression is true; otherwise, it's false:

```
let numbers = [4, 5, 6]
if numbers.contains(5) {
  print("There is a 5")
}
```

Early in this course, you learned that constants can't be modified once they're declared. It's the same when you assign a collection to a constant using `let`: You can't add, remove, or modify any items within the collection. However, if you store the collection within a variable using `var`, you'll be able to add to, remove from, or modify items in the collection. In addition, you'll be able to empty the collection or set the variable to a different collection entirely.

ARRAY TYPES

An array is like a basket: It can start out empty and you can fill it with values at a later time. But if an array literal doesn't contain any values, how can its type be inferred?

You can declare the type of an array using type annotation, collection type annotation, or an array initializer.

This example defines an array with traditional type annotation.

```
var myArray: [Int] = []
```

This example defines an array using a special collection type annotation. This is a less common practice you should be familiar with in case you run across it in code you're working with.

```
var myArray: Array<Int> = []
```

Just like all objects can be initialized by adding a `()` after the type name, you can add a `()` after `[Int]` to initialize an empty array of `Int` objects. You should also be familiar with this code in case you run across it in the future.

```
var myArray = [Int]()
```

WORKING WITH ARRAYS

You may find some situations when it's tedious or error-prone to use array literals. For example, say you want an array filled with 100 zeros. In array literal form, you'd have to enter all 100 zeros, and it would be easy to miscount them.

That's OK. Swift has a solution. Instead of counting out the 100 zeros, you can use the following initializer to create an array of count `100` with repeating default values:

```
var myArray = [Int](repeating: 0, count: 100)
```

To find out the number of items within an array, you can use its `count` property. What if you wanted to check if the array is empty? Rather than comparing `count` to 0, you can check the array's `isEmpty` property, which returns a `Bool`:

```
let count = myArray.count
if myArray.isEmpty { }
```

Once you've defined an array, you can use various methods and properties to access or modify it. You can also use an array's subscript syntax. Subscript syntax uses square brackets after a collection to refer to a specific element or range inside the collection. To retrieve a particular value from a collection, specify inside square brackets the index of the value you wish to access. An array's values are always zero-indexed, meaning that the first element of an array has an index of zero. The following example will retrieve the first value in a collection of first names:

```
let firstName = names[0]
```

You can also use subscript syntax to change a value at a given index. By putting the subscript on the left side of the `=` sign, you can *set* a particular value rather than *access* the current one. In the example below, you're changing the second name in the collection to `"Paul"`.

```
names[1] = "Paul"
```

When subscripting an array, you need to be sure that the index you specify exists in the array. For example, if you specify `3` as the index, the array must have at least four elements. If it doesn't, your program will crash when that line of code is executed.

After defining an array and setting some values, you'll often want to `append` new values. You can use a variable array's append function to add a new element at the end of the array. To append multiple elements at once, use the `+=` operator.

```
var names = ["Amy"]
```

```
names.append("Joe")
names += ["Keith", "Jane"]
print(names) //["Amy", "Joe", "Keith", "Jane"]
```

What if you didn't want the new element to show up at the end of the array? A variable array also has an `insert(_: at:)` function that allows you to specify the value and the index.In this scenario, as with subscripting, you'll need to be sure that you supply a valid index.

Since an array is zero-indexed, the first element always has an index of 0, and the last element's index is equal to `count`-1. In the following example, "Bob" will be inserted at the beginning of the array:

```
names.insert("Bob", at: 0)
```

Similarly, the `remove(at:)` function works to remove an item at a specified index. Unlike `insert(_: at:)`, you don't need to enter the value; the function returns the removed item by default, as in the example below. Another method is `removeLast()`, which removes the last item from the array, eliminating the need to calculate the index. Finally, the `removeAll()` method will remove all elements from the array.

```
var names = ["Amy", "Brad", "Chelsea", "Dan"]
let chelsea = names.remove(at:2)
let dan = names.removeLast()
names.removeAll()
```

Swift also allows you to to create a new array by adding together two existing arrays of the same type. You can specify which array comes first within the resulting array:

```
var myNewArray = firstArray + secondArray
```

What about arrays within an array? Use the array literal syntax to contain two smaller arrays inside a larger one. You can then use subscript syntax on the container array to

access one of the nested arrays. To access a particular element within one of the nested arrays, you can use two sets of subscripts.

```
let array1 = [1,2,3]
let array2 = [4,5,6]
let containerArray = [array1, array2] //[ [1,2,3], [4,5,6] ]
let firstArray = containerArray[0]  //[1,2,3]
let firstElement = containerArray[0][0] //1
```

DICTIONARIES

The second type of collection in Swift is a dictionary. Like a real-world dictionary that contains a list of words and their definitions, a Swift dictionary is a list of keys, each with an associated value. Each key must be unique, just like each word in the dictionary is unique. And just as an English dictionary is in alphabetical order to make the words easy to look up, a Swift dictionary is optimized to make key lookups very fast.

You can set up a dictionary using a dictionary literal: a list of comma-separated key/value pairs surrounded by square brackets. A colon separates each key and its resulting value:

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

Just as with an array, the dictionary's type can be inferred based on the types used in the dictionary literal. Say you want to store a list of high scores in a game. You'll use the player's name as the key and the player's score as the corresponding value. The dictionary's type will be inferred as a dictionary where the keys are of type `String` and the values are of type `Int`. This can be represented as either `[String: Int]` or `Dictionary<String, Int>`:

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

As with the two other collection types, a dictionary has a `count` property to determine the number of key/value pairs and an `isEmpty` property for determining whether or not the dictionary has no key/value pairs. The syntax for creating an empty dictionary will probably seem familiar by now:

```
var myDictionary = [String: Int]()
```

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

```
var myDictionary: [String: Int] = [:]
```

All of these examples result in the same type of empty dictionary. You'll likely have a preferred method that you use regularly, but you should be familiar with all of them.

ADD/REMOVE/MODIFY A DICTIONARY

Subscript syntax is particularly handy with a dictionary. Since the order in a Swift dictionary doesn't matter, there's no index and there's no risk of subscripting errors associated with indices.

The following example adds a new score to the dictionary of high scores. If the key "Oli" already exists in the dictionary, this code will replace the old value with the new one:

```
scores["Oli"] = 399
```

But what if you want to know if there's an old value in the dictionary *before* replacing it? You can use `updateValue(_:, forKey:)` to update the dictionary, and the value returned from the method will be equal to the old value, if one existed. In the following example, `oldValue` will be equal to Richard's old value before the update. If there was no value, `oldValue` will be `nil`. You'll learn more about the `nil` keyword later. It's a special way of representing the *absence* of a value.

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

Swift uses if-let syntax to let you run code *only if* a value is returned from the method. If there wasn't an existing value, the code within the brackets won't be executed:

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

To remove an item from a dictionary, you can use subscript syntax, setting the value to `nil`. Similar to updating a value, dictionaries have a `removeValue(forKey:)` method if you need the old value returned before removing it:

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

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

ACCESSING A DICTIONARY

Swift dictionaries provide two properties not included in other collection types. You can use `keys` to return a list of all the keys within a dictionary and `values` to return a list of all the values. If you want to use these collections subsequently, you'll need to convert them to arrays:

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

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

To look up a particular value within a dictionary, use if-let syntax. If the key you specify is in the dictionary, the result will be the key's corresponding value. However, if the key isn't in the dictionary, the code within the brackets won't be executed.

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

if let henrysScore = scores["Henry"] {
  print(henrysScore) //not executed; "Henry" is not a key in
     the dictionary
}
```

Console Output:

```
400
```

LAB

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

REVIEW QUESTIONS

문제 1(총 2개)

Given the following dictionary, what is the result of executing `numberOfLegs["snake"] = 0`?

```
var numberOfLegs = ["spider": 8, "human": 2, "dog": 4,
"cat": 4]
```

- **A.** Nothing; `numberOfLegs` cannot be modified.
- **B.** "snake" is added as a key, with a value of 0.
- **C.** "snake" is updated to have a value of 0 instead of 4.
- **D.** An error is thrown because two keys in the dictionary have the same value.

정답 확인

Lesson 2.6

Loops

You can probably think of many everyday tasks that you continue performing until a condition is met. You might continue filling a water bottle until it's full or continue doing homework until the assignment is complete.

Scenarios that require completion and repetition of a task can be done in code using loops. In this lesson, you'll learn how to create loops in Swift, control the conditions for looping, and specify when to stop.

What You'll Learn

- How to step through each value of a collection using a for loop
- How to iterate through a range of values
- How to write a loop that continues until a condition is no longer true

Vocabulary

- for-in loop
- while loop

Related Resources

- Swift Programming Language Guide: For-In Loops
- Swift Programming Language Guide: While Loops

Computer programs are great at repeating things. Developers often write code to perform the same work across multiple objects, perform a task a huge number of times, or continue to perform work until specific conditions have been met. Swift provides three different ways to loop through, or repeat, blocks of code.

FOR LOOPS

The first loop you'll learn is the for loop, also known more specifically as a for-in loop. A for loop is useful for repeating something a set number of times or for performing work across a collection of values.

A for-in loop executes a set of statements for each item within a range, sequence, or collection. Suppose you have a range of numbers between 1 and 5, and you want to print each value within the range. Rather than writing out five `print` statements, you can use for-in over the range and write one `print` statement. The syntax looks like this:

```
for index in 1...5 {
  print("This is number \(index)")
}
```

Console Output:

```
This is number 1
This is number 2
This is number 3
This is number 4
This is number 5
```

In the code above, `index` is a constant that's available to customize the work performed within the braces (the for-in loop). The first time the statement within the braces is executed, `index` has a value of 1, the first value in the range. When execution is complete, the value of `index` is updated to 2, the next value in the range. As `index` is updated to each of the values, the `print` statement is executed five times. After the entire range has been exhausted, the loop is complete, and the code will move on to statements after the loop.

But let's say your result doesn't need to use the values in the range. If you just need a way to perform a series of steps a certain number of times, you can skip assigning a value to a constant and replace its name with a _:

```
for _ in 1...3 {
  print("Hello!")
}
```

Console Output:

```
Hello!
Hello!
Hello!
```

You can use the same for-in syntax to iterate over each item in an array.

```
let names = ["Joseph", "Cathy", "Winston"]
for name in names {
  print("Hello \(name)")
}
```

Console Output:

```
Hello Joseph
Hello Cathy
Hello Winston
```

Since `String` is a collection type, it has a similar functionality to an array. You can use a for-in loop to iterate over each character in a String:

```
for letter in "ABCD" {
  print("The letter is \(letter)")
}
```

Console Output:

```
The letter is A
The letter is B
The letter is C
The letter is D
```

What if you need the index of each element in addition to its value? You can use the `enumerated()` method of an array or string to return a tuple containing both the index and the value of each item:

```
for (index, letter) in "ABCD".enumerated() {
  print("\(index): \(letter)")
}
```

Console Output:

```
0: A
1: B
2: C
3: D
```

If you use a for-in loop with a dictionary, the loop generates a tuple–a special type that can hold an ordered list of values wrapped in parentheses– that holds the key and value of each entry. Since a dictionary is typically accessed by specifying a key, the loop doesn't guarantee any particular order of the items as it works through the dictionary:

```
let vehicles = ["unicycle" : 1, "bicycle" : 2, "tricycle" : 3,
"quad bike" : 4]
for (vehicleName, wheelCount) in vehicles {
  print("A \(vehicleName) has \(wheelCount) wheels")
}
```

Console Output:

```
A unicycle has 1 wheels
```

```
A bicycle has 2 wheels
A tricycle has 3 wheels
A quad bike has 4 wheels
```

WHILE LOOPS

A while loop will continue to loop until its specified condition is no longer true. Imagine you want to keep playing a game until you run out of "lives." The condition under which you continue looping is that the number of lives is greater than 0:

```
var numberOfLives = 3

while numberOfLives > 0 {
  playMove()
  updateLivesCount()
}
```

Swift checks the condition before each loop is executed, which means it's possible to skip the loop entirely if the condition is never satisfied. If `numberOfLives` had been initialized to 0 in the above example, the while loop would determine that `0 > 0` is false and would never proceed to the body of the loop.

For this reason, the body of the while loop should perform work that will eventually change the condition. In the example below, nothing is updating the value of `numberOfLives`, so the condition always resolves to `true` and continues forever.

```
var numberOfLives = 3

while numberOfLives > 0 {
  print("I still have \(numberOfLives) lives.")
}
```

Instead, the body should execute statements that will, at some point, result in a `false` condition:

```
var numberOfLives = 3
var stillAlive = true
while stillAlive {
  numberOfLives -= 1
  if numberOfLives == 0 {
    stillAlive = false
  }
}
```

CONTROL TRANSFER STATEMENTS

There may be situations when you want to stop execution of a loop from within the loop's body. The Swift keyword `break` will break the code execution within the loop and start executing any code defined after the loop.

In the code below, the loop breaks if the counter reaches 0:

```
//Prints -3 through 0
for counter in -3...3 {
  print(counter)
  if counter == 0 {
    break
  }
}
```

Console Output:

```
-3
-2
-1
0
```

There may also be situations in which you want to skip to the next iteration in a loop. While the `break` keyword will end the loop entirely, `continue` will move onto the next iteration. For example, you may have an array where each element is of type `Person`, and you want to loop through the array and send an email to everyone 18 years of age and older:

```
for person in people {
  if person.age < 18 {
    continue
  }

  sendEmail(to: person)
}
```

Computers are incredibly good at performing tasks as many times as requested. Whether you need to iterate through a collection or repeat a series of steps until a condition is met, loops are an important concept to understand well. You'll use them frequently throughout your programming career.

LAB

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

REVIEW QUESTIONS

문제 1(총 2개)

Which of the following are used to loop over a section of code?

- **A.** `for`, `if`, `while`
- **B.** `while`, `Range`, `if`
- **C.** `for`, `while`, `do`
- **D.** `for`, `while`

정답 확인

Lesson 2.7

Introduction to UIKit

As a user of computers, appliances, and all sorts of devices, you already know that user interface is important. Now that you're creating your own apps on iOS, you'll rely heavily on `UIKit`, a foundational framework for building and managing user interfaces, or UIs. `UIKit` defines how you display information to the user and how you respond to user interactions and system events. It also allows you to work with animations, text, and images. Basically, if you can see it on an iOS device, it was likely built with `UIKit`.

In this lesson, you'll learn about some of the most commonly used interface elements in `UIKit` and where you can go to find out more.

What You'll Learn

- Why `UIKit` is such an important part of app development
- The name and appearance of five common views in apps
- The name and functionality of five controls in apps
- Where to find out more

Vocabulary

- button
- control
- control event
- date picker
- image view
- `UIKit`
- label
- navigation bar
- scroll view
- segmented control
- slider
- switch
- tab bar
- table view
- text field
- toolbar

Related Resources

- Human Interface Guidelines

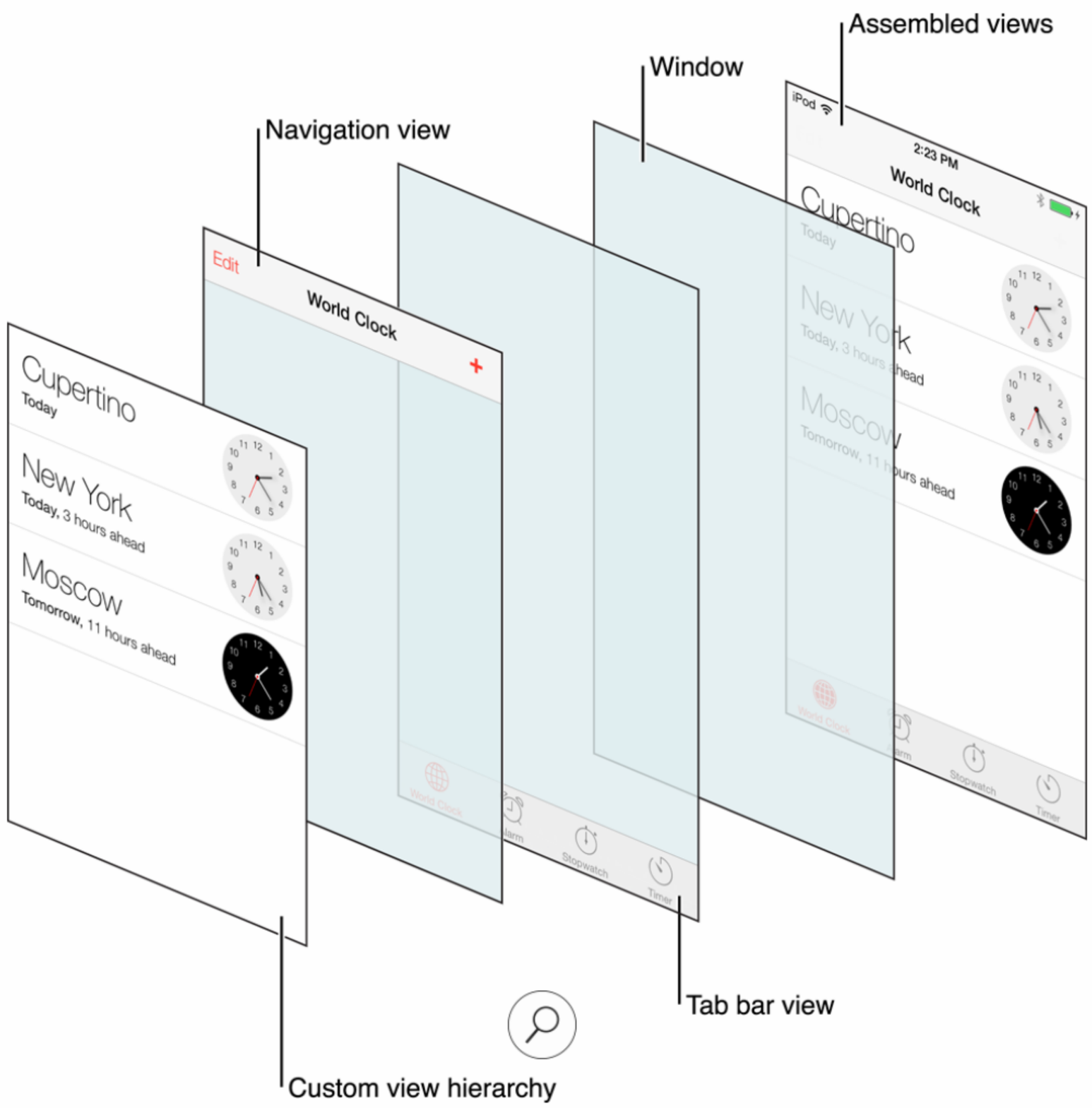
The `UIKit` framework provides the crucial pieces you need to build and manage iOS apps. It includes definitions for all user interface objects, the event-handling system that responds to user input, and the entire model that allows apps to run in iOS.

COMMON SYSTEM VIEWS

The foundational class for all visual elements defined in `UIKit` is the `UIView`, or view. A view defines a rectangular shape that can be customized to display anything on the screen. Text, images, lines, and graphics all depend on `UIView` as the base.

`UIKit` defines dozens of special `UIView` subclasses that perform specific tasks. For example, `UILabel` displays text, `UIImageView` displays an image, and `UIScrollView` allows you to put scrollable content onto the screen.

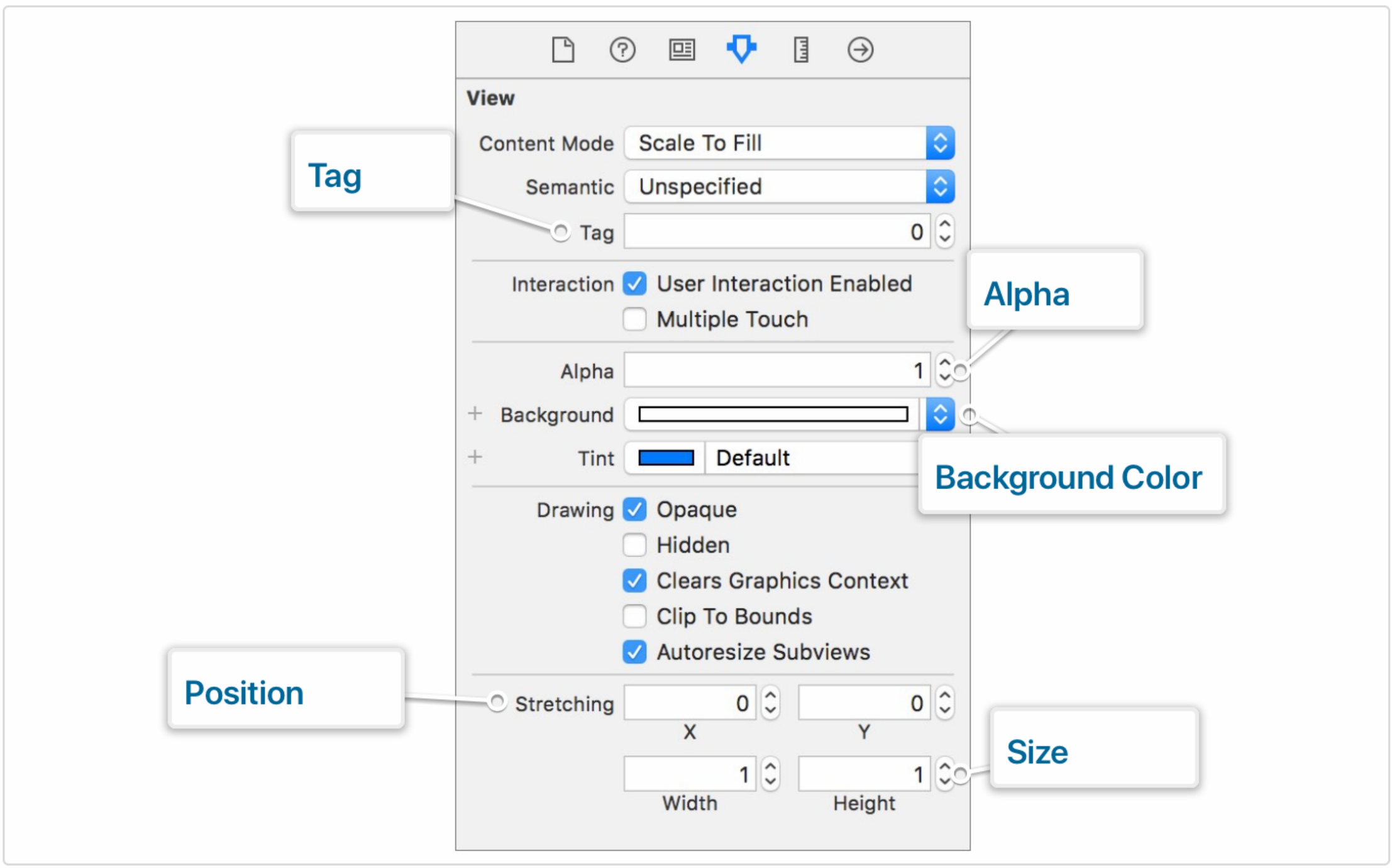
Almost all screens in an app consist of many views, which together make up a view hierarchy.



Views are often nested, as in the the previous view hierarchy diagram. A view that's contained in another view is called a child view. A view that contains one or more views is called a parent view. In this example, each cell in the list is a child view of a table view, and each cell is a parent to three views: a label that displays a city name, another label that displays how the time zones are related, and an image view that displays an image of a clock.

To display a view onscreen, you need to give it a frame—which consists of a size and a position—and add it to the view hierarchy. The area within the view is its bounds. When adding a view in Interface Builder, its background color is white by default. When creating a view in code, the background is transparent by default. You can set a new background color in either scenario.

Here are some of the attributes you can change when working with views:



Next, take a look at some of the most common views defined in `UIKit`.

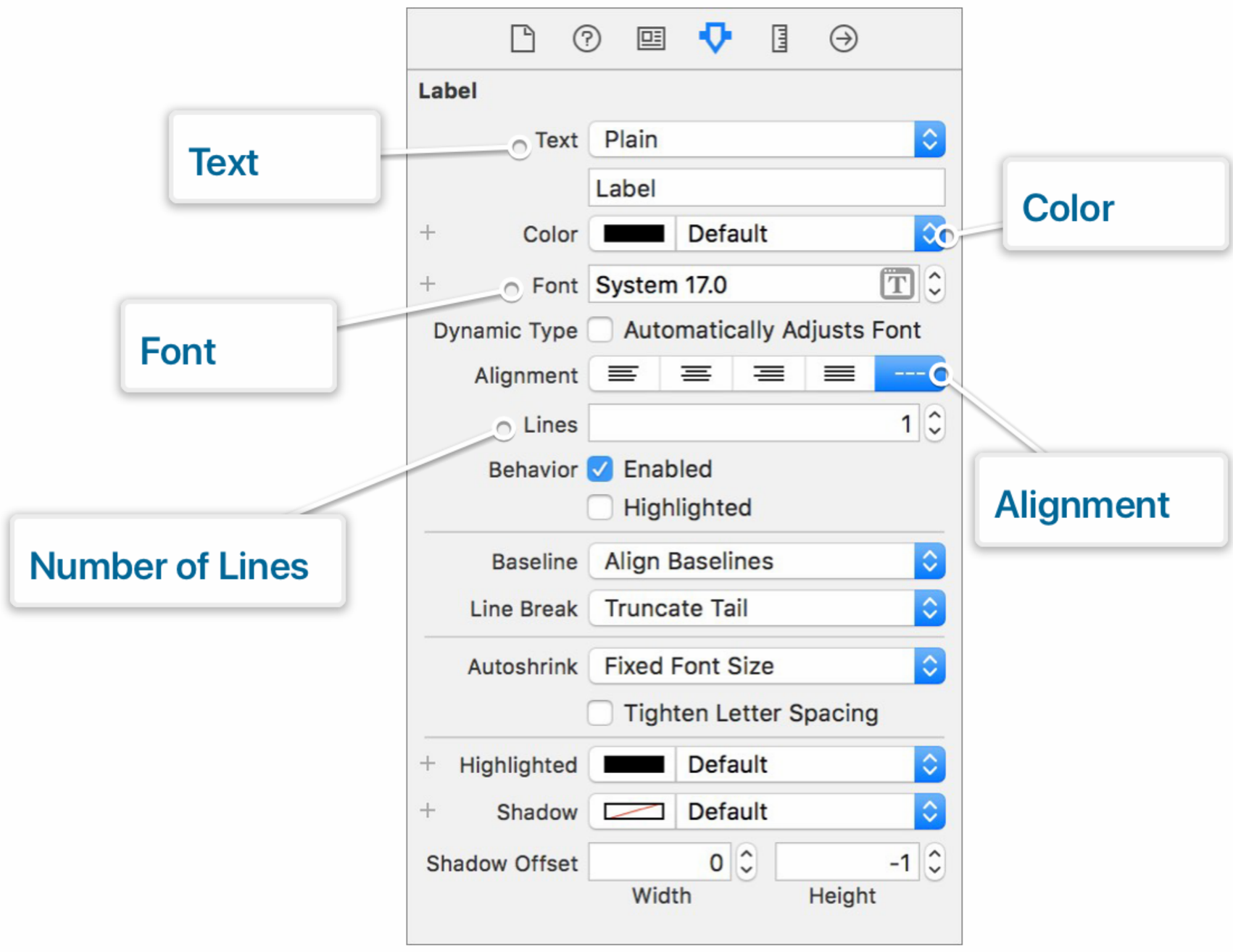
This section covers some common views you'll work with using `UIView`. As you learn about the different views, think about apps you use and which views might be used to create their interfaces.

Label (UILabel)

Labels use static text to relay information to the user.

The volume of the ringer and alerts can be adjusted using the volume buttons.

Configuration



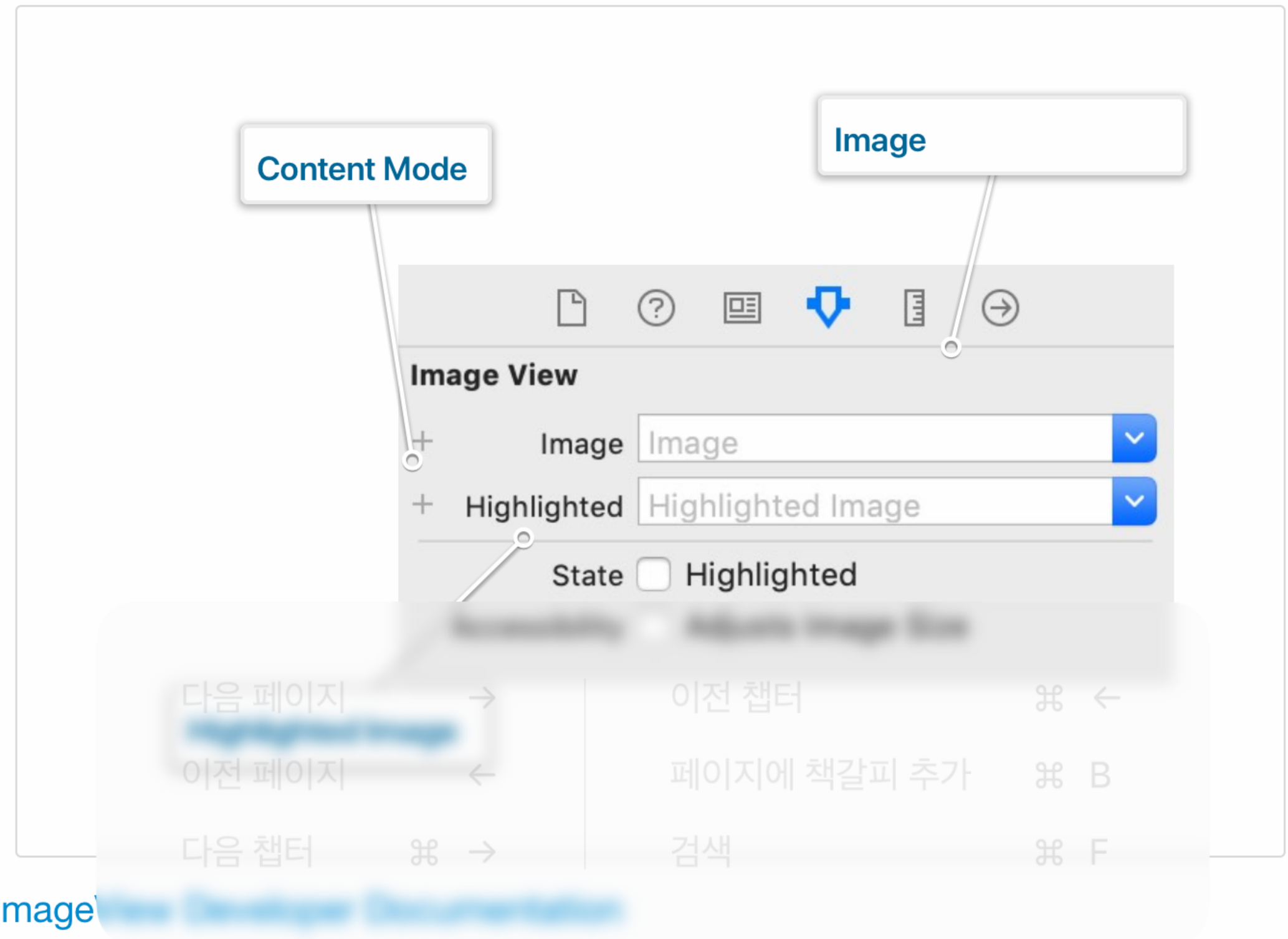
UILabel Developer Documentation

Image View (UIImageView)

An image view displays an image or an animated sequence of images. Some people confuse the image view with the image itself. Think of it this way: Just like a label displays text, an image view displays an image.

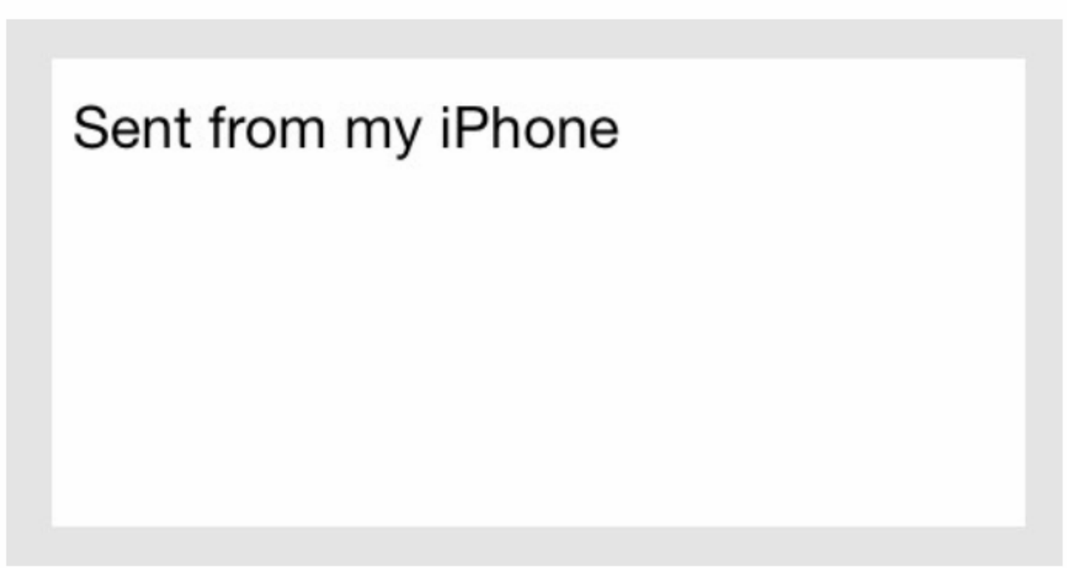
오후 10:36 7월 18일 토요일 62%

Configuration



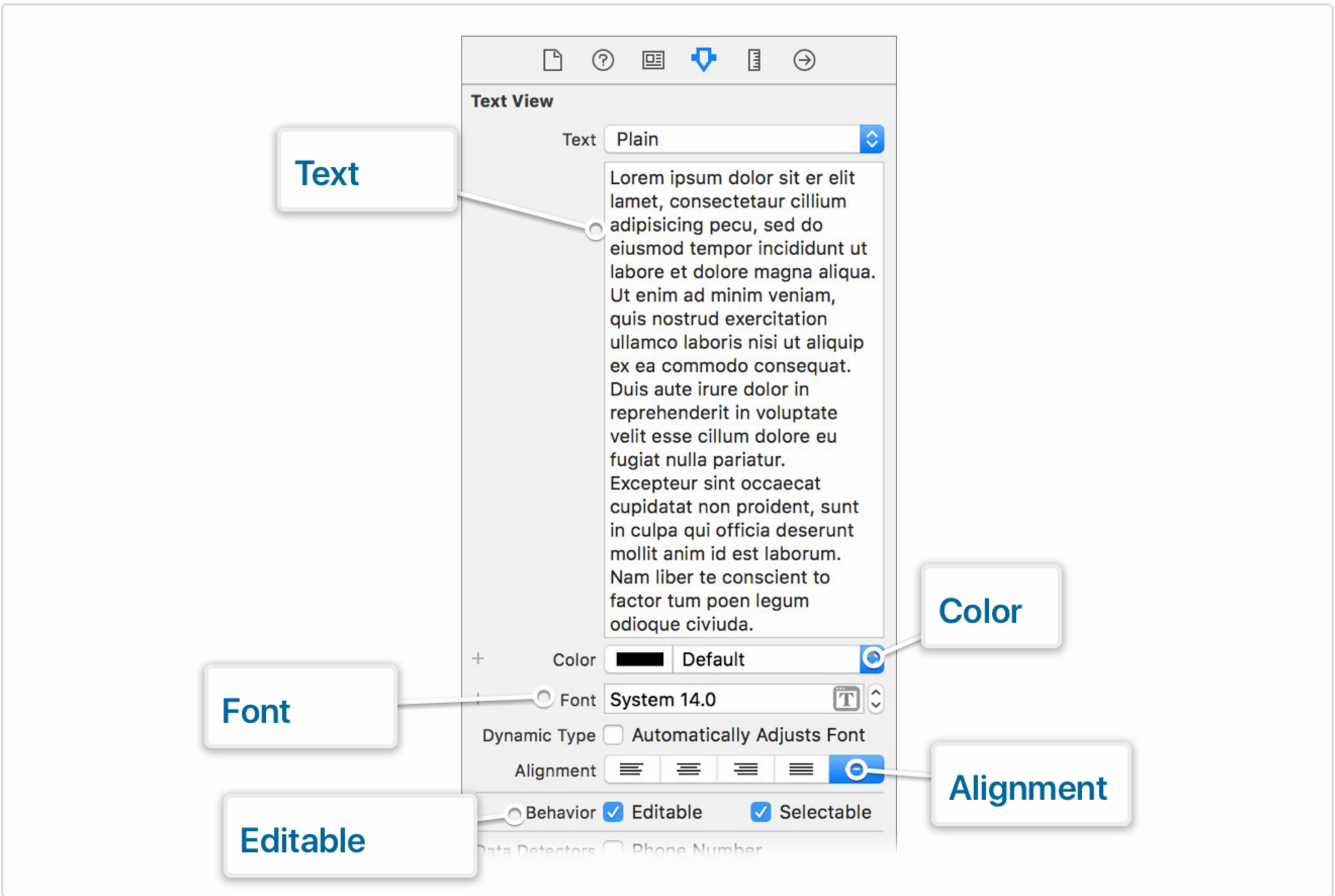
UIImageView Developer Documentation

Text View (`UITextView`)



A text view allows the user to input text in your app. Text views accept and display multiple lines of text, with support for scrolling and editing. You'll typically use a text view to display a large amount of text, such as the body of an email message.

Configuration



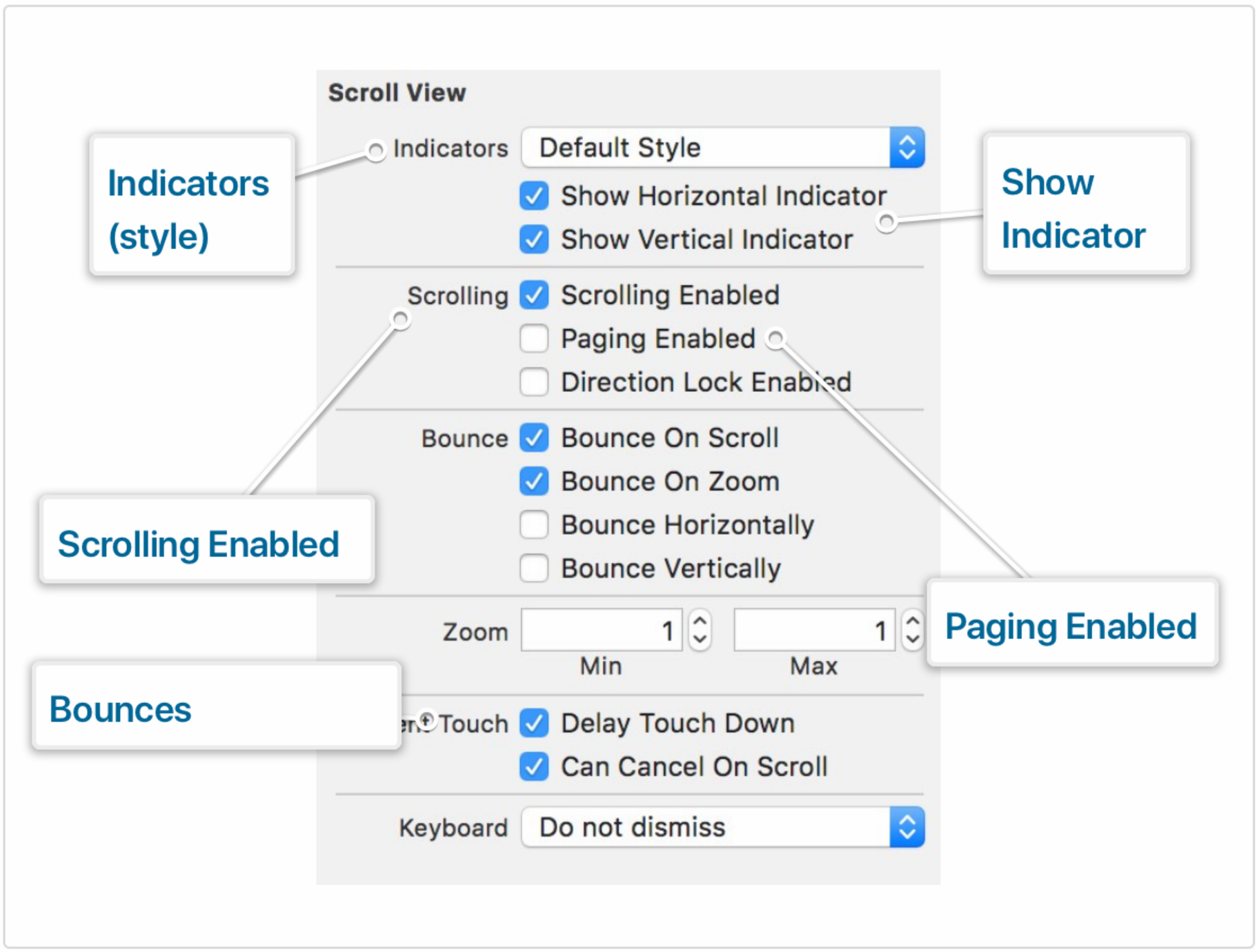
UITextView Developer Documentation

Scroll View (`UIScrollView`)

A scroll view allows the user to see content that runs beyond the boundaries of the view. You'll typically use a scroll view when the information you want to display is larger than the device's screen.

When the user interacts with a scroll view, a vertical or horizontal scroll indicator briefly appears to indicate there's more content to view.

Configuration



UIScrollView Developer Documentation

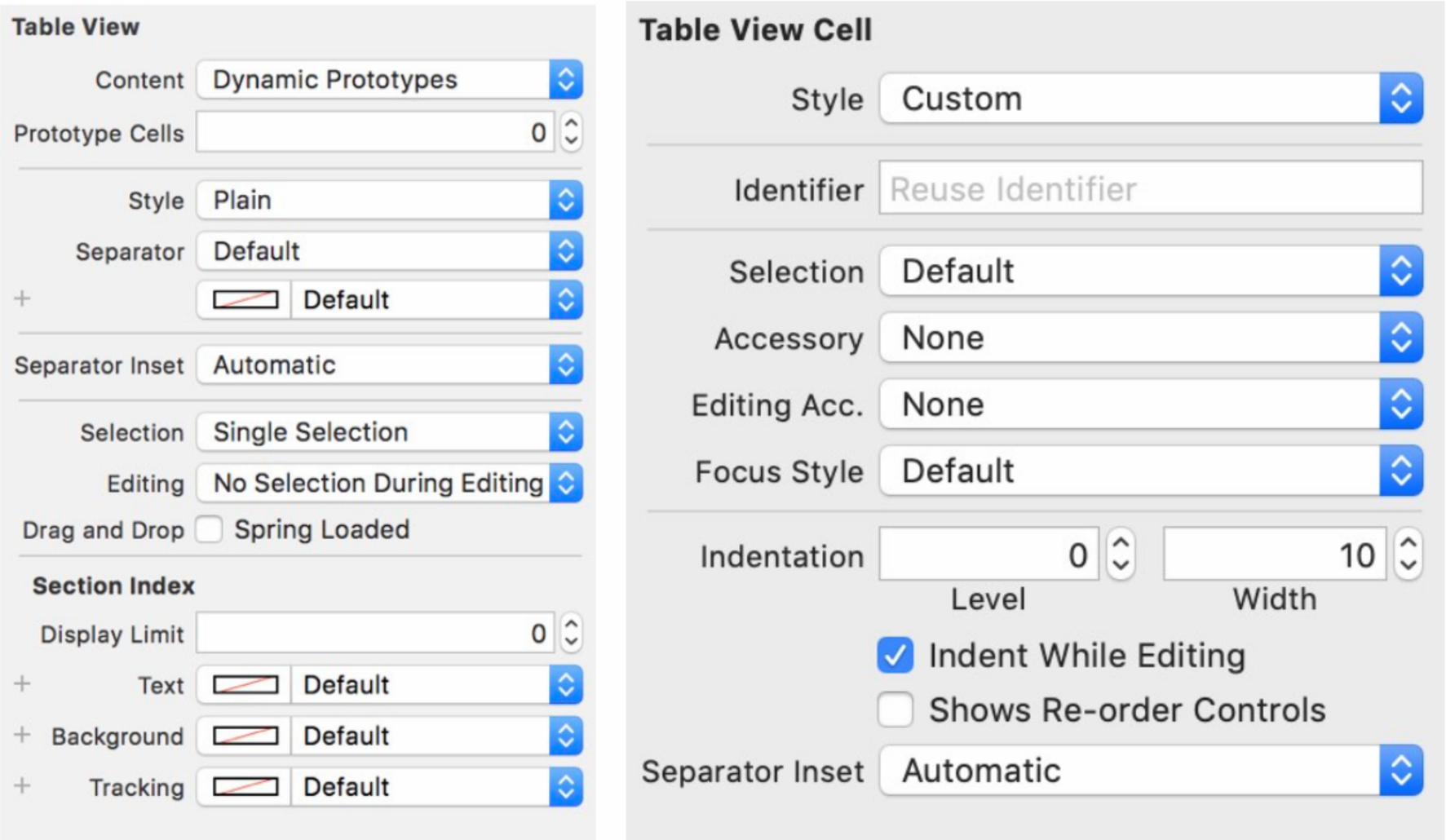
Table View (`UITableView`)

A table view presents data in a single scrollable column of rows and sections, allowing users to navigate easily through groups of information. Table views are an excellent format for displaying and editing hierarchical lists of information.

For example, the Mail app uses a table view to display email messages in the user's inbox, and the Messages app uses a table view to display message threads organized by contacts.

Configuration

Table views are interesting because you can customize the look and feel of the table view itself, as well as the look and feel of the rows (or cells) that it displays.



You'll learn more about table views and how to customize them in future lessons.

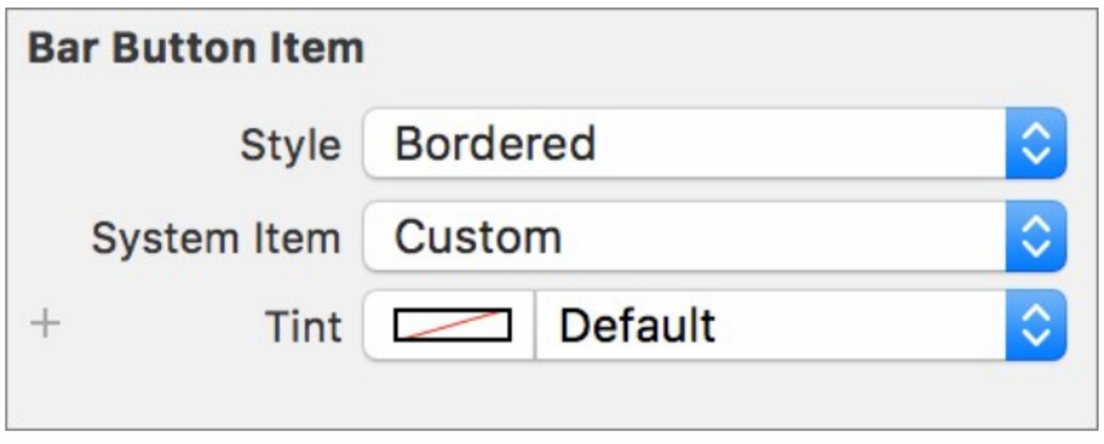
UITableView Developer Documentation

Toolbars (`UIToolbar`)

A toolbar usually appears at the bottom of a screen and displays one or more buttons, called bar button items. Users can select a button, or tool, to perform an action within a given view.

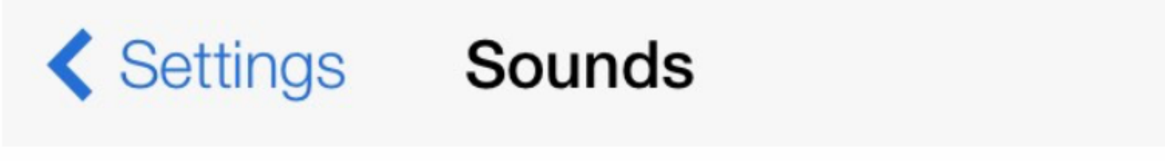
Configuration

You can configure the toolbar itself, or the bar items it displays. Each bar item consists of a title and an image, and can be enabled or disabled programmatically.



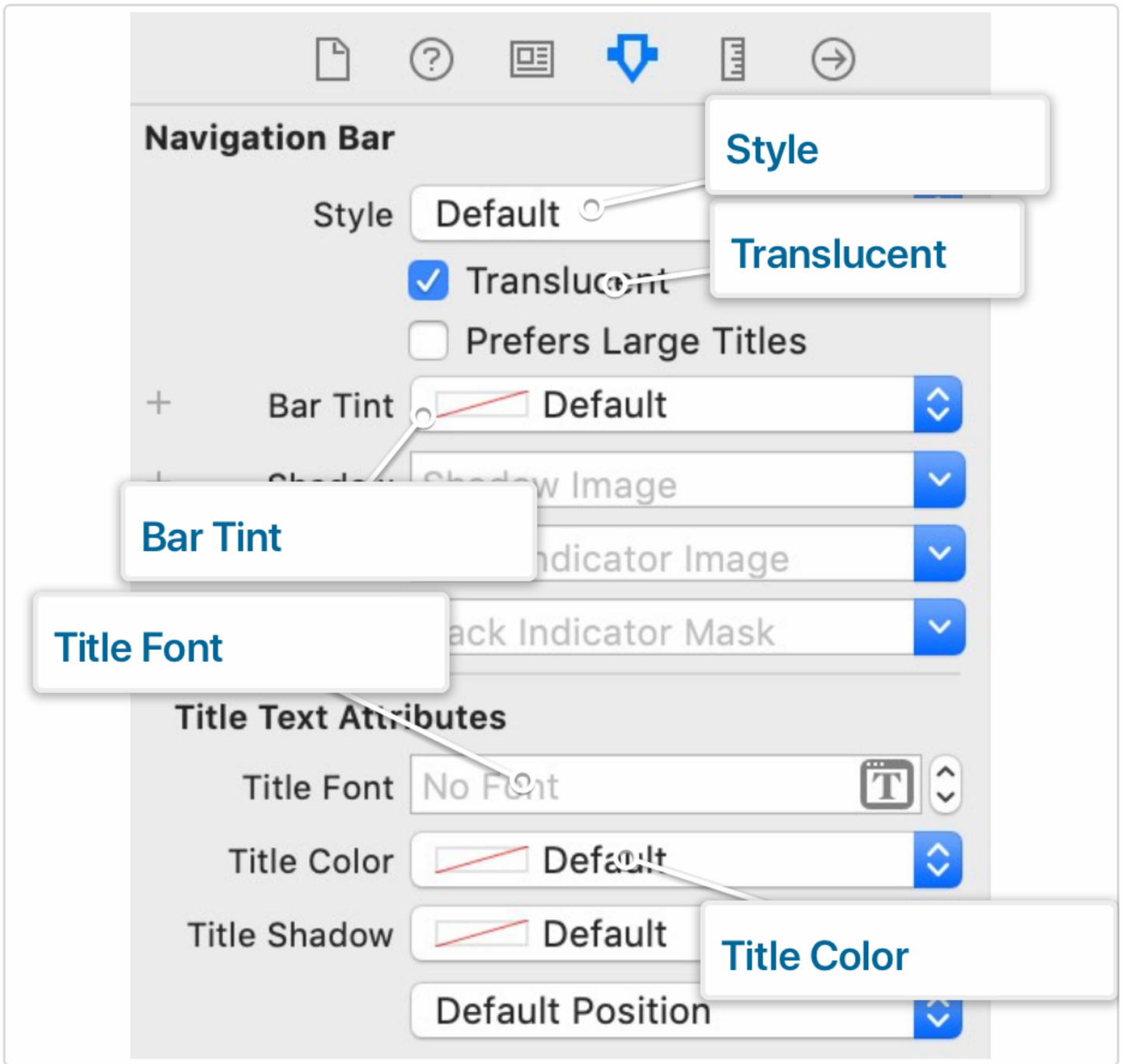
UIToolBar Developer Documentation

Navigation Bars (`UINavigationBar`)



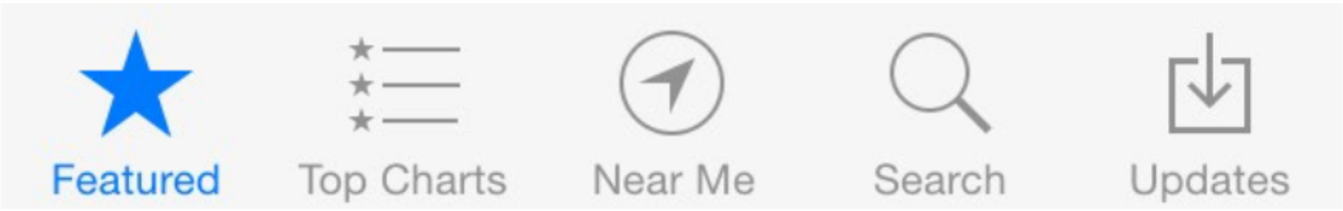
You'll use the navigation bar to present your app's primary content in an organized way, one that you hope will be intuitive to the user. Navigation bars are typically displayed at the top of the screen, with bar buttons for navigating through a hierarchy of screens. They'll most likely include a title and a back button.

Configuration



UINavigationBar Developer Documentation

Tab Bars (`UITabBar`)



A tab bar provides easy access to different views in an app. It displays multiple tab bar items, each made up of an icon image and text. You'll use tab bars in your app to organize information by a specific feature or task. The most common way to use a tab bar is with a tab bar controller, which holds a property of each view controller that represents each scene you want presented in the tab bar. When the user taps an item, a new view related to the new task is displayed.

Tab bars are frequently used in apps that present multiple workflows or courses of action. For example, the App Store app has separate tabs to browse featured apps, search for a specific app, or check for available updates.

Configuration

Because you'll most often use the tab bar in conjunction with a tab bar controller, you'll add scenes to be displayed to the controller. The view controller for each scene has a `UITabBarItem` property that defines the text and optional image that will be displayed by the tab bar. To add a scene to a tab bar controller, and the tab bar view, you link it to the tab bar controller's `viewControllers` property in a storyboard.

UITabBar Developer Documentation

CONTROLS

In the previous section, you learned about common views, which display information to the user. What about responding to user input? You'll use tools in `UIKit`, called controls, to tell the app what to do.

Think of a control as a communication tool between the user and the app. When the user interacts with a control, the control triggers a control event. Different controls trigger different control events.

After setting up a control in Interface Builder, you set up an `IBAction` that responds to a specific control event and allows you to execute a block of code. Most often you will use the Primary Action Triggered (`UIControl.Event.primaryActionTriggered`) control event. This control event is triggered when a button is tapped or when the value of a control changes.

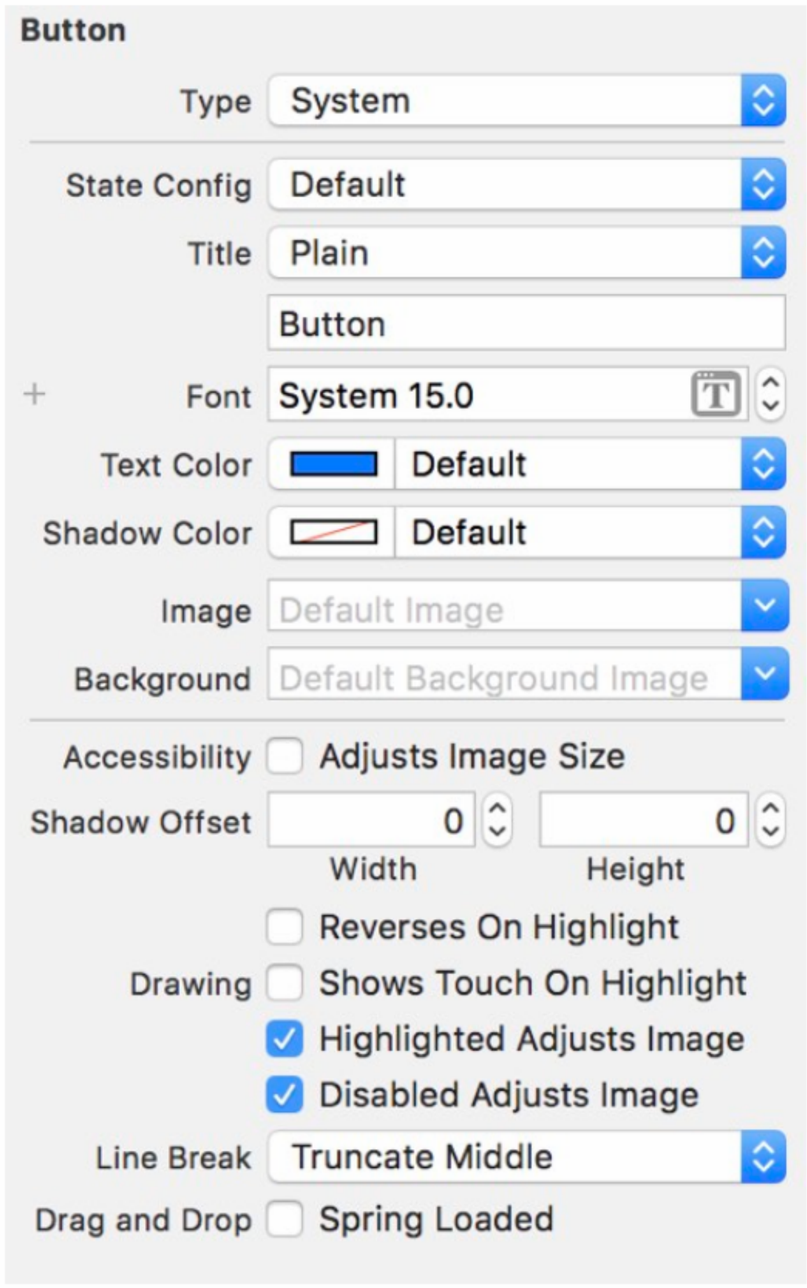
Controls are simple, straightforward, and familiar to users because they appear throughout most iOS apps. As you learn about some of the different subclasses of `UIControl`, on the following pages, think about your favorite apps and how you use controls to interact with to them.

Buttons (`UIButton`)

Button

Users of iOS devices can initiate a control event with the tap of a button. When you set up a button, you give it a title or an image that conveys what the button will do when tapped. The appearance of the button changes with different states of being tapped: tapping down and lifting up.

Configuration



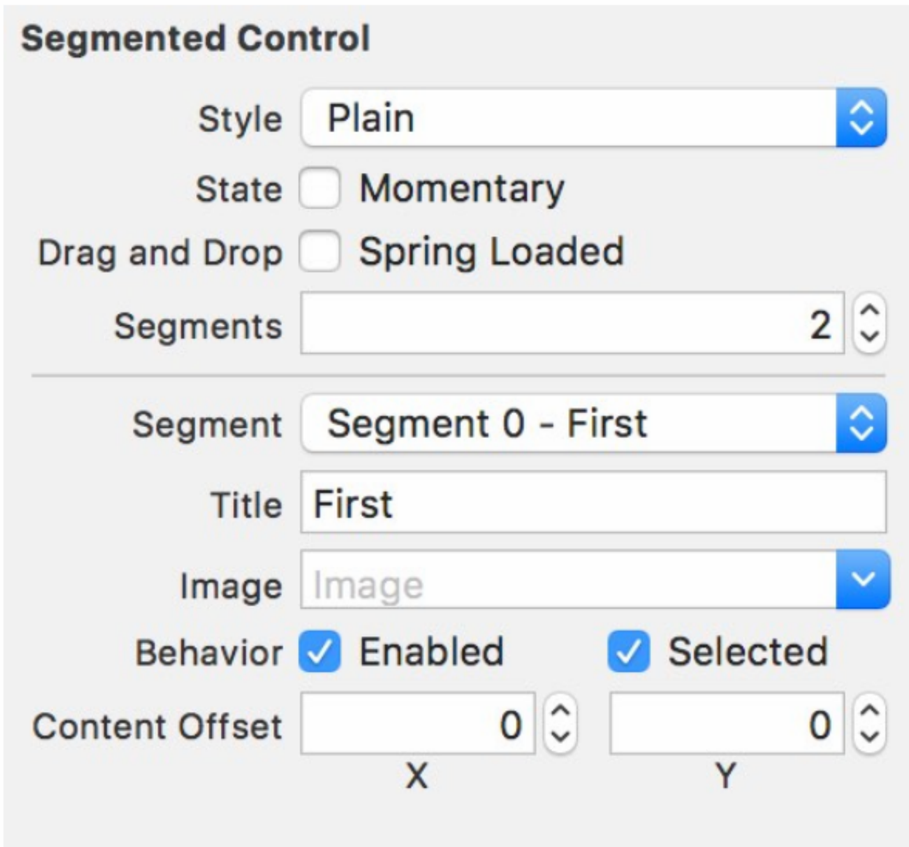
The primary control event is triggered when the user releases a button after tapping it. Buttons can also execute code at different stages of a tap, such as when the user first touches the button, holds down the button, or cancels the tap by dragging their finger outside of the frame of the button before lifting their finger.

UIButton Developer Documentation

Segmented Controls (`UISegmentedControl`)

A segmented control is a horizontal set of multiple segments. Each segment functions as a discrete button, allowing the user to choose from a limited, compact set of options. This example, from the App Store, allows the user to change the Top Charts view, with three choices: paid apps, free apps, or top grossing apps.

Configuration



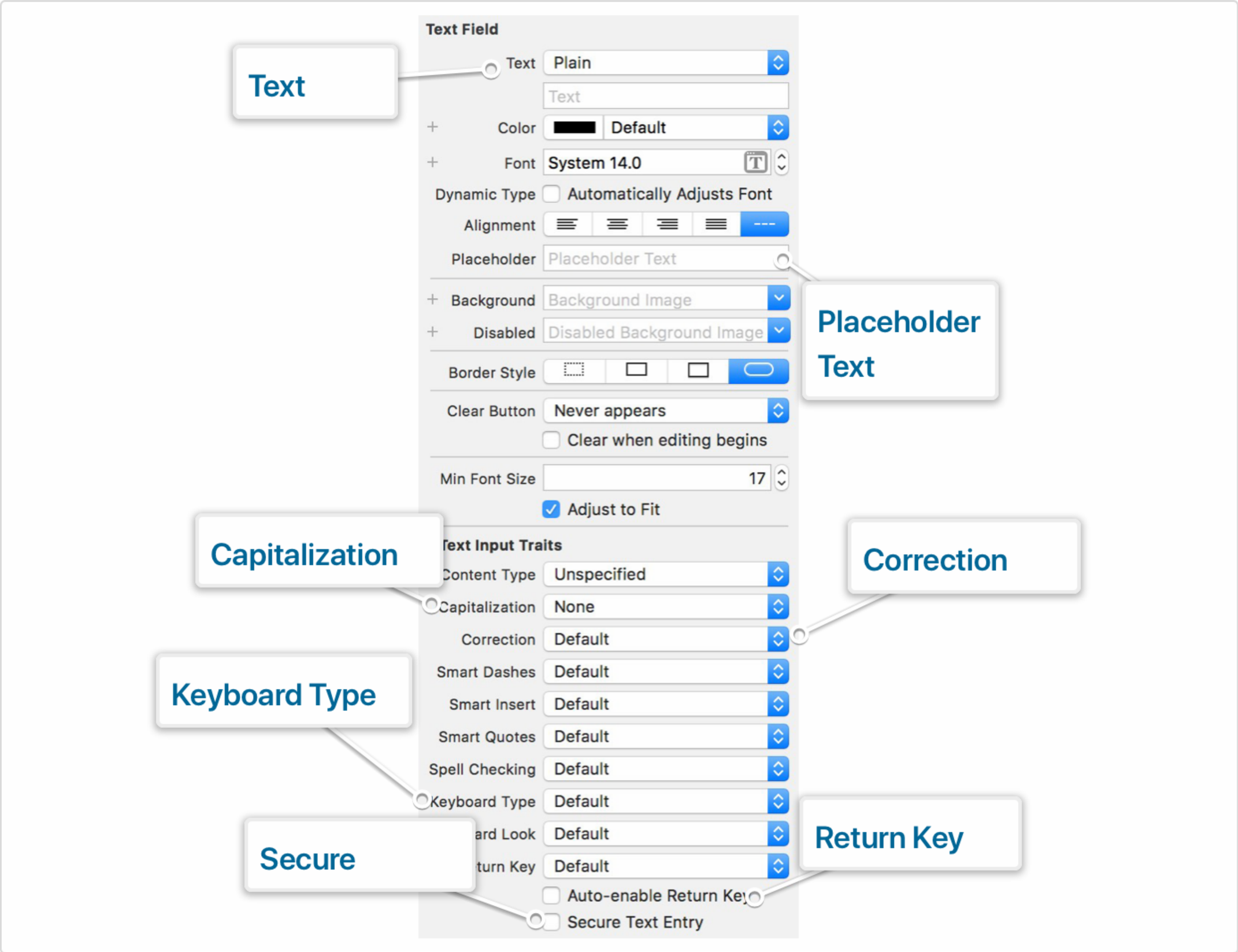
Segmented controls execute code when the control's value changes. The value represents which segment of the control is selected.

UISegmentedControl Developer Documentation

Text Fields (`UITextField`)

Text fields allow the user to input a single line of text into an app. You'll use them to gather a small amount of text and to perform some action based on that text.

Configuration



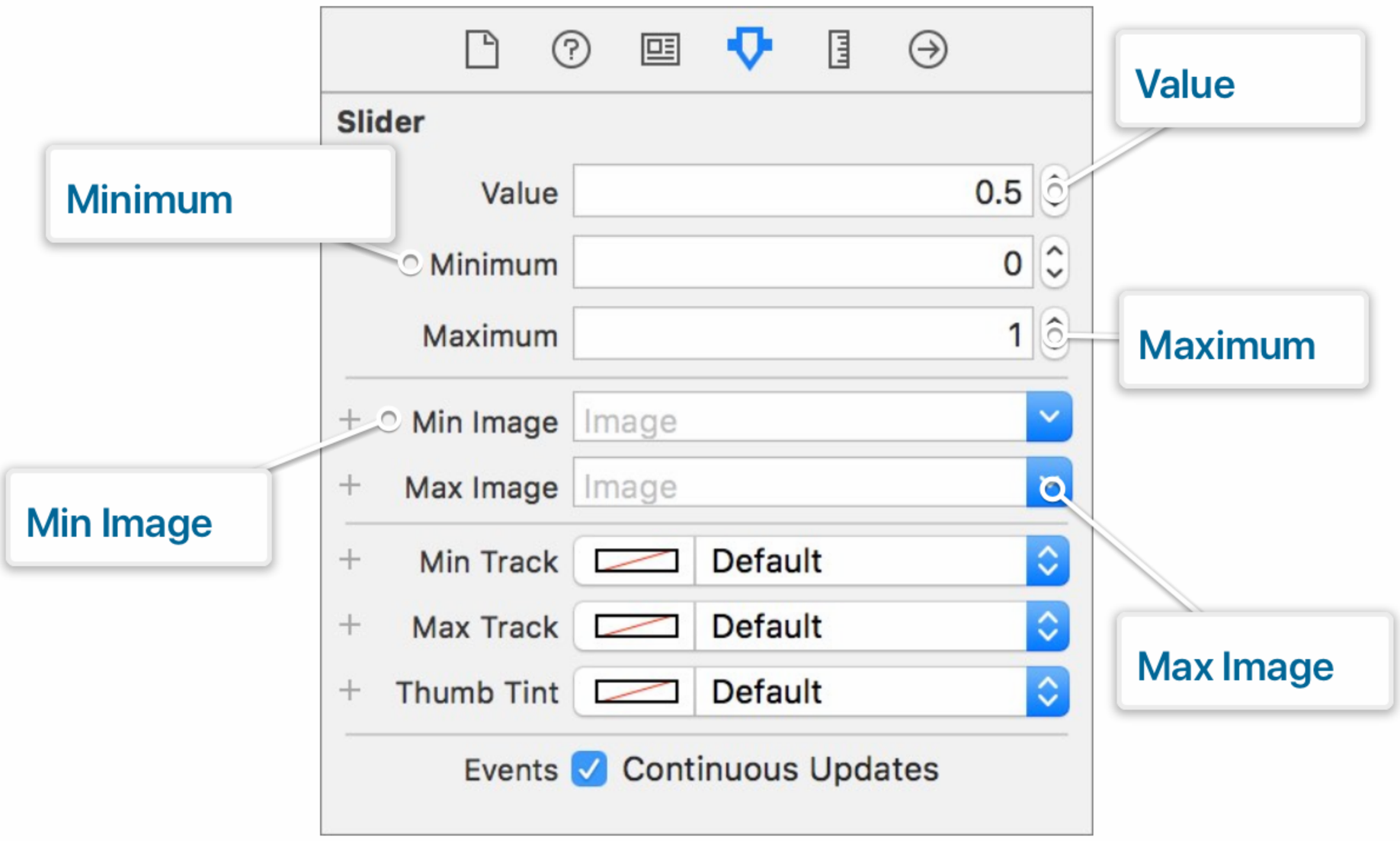
Text fields execute code when the user presses the 'Return' or 'Done' key on the keyboard or when the user edits the text.

UITextField Developer Documentation

Sliders (`UISlider`)

Sliders allow users to make smooth and gradual adjustments to a value—useful for controls that modify things like speaker volume, screen brightness, or color values. The user controls a slider by moving from a starting value along a continuous range between minimum and maximum values.

Configuration



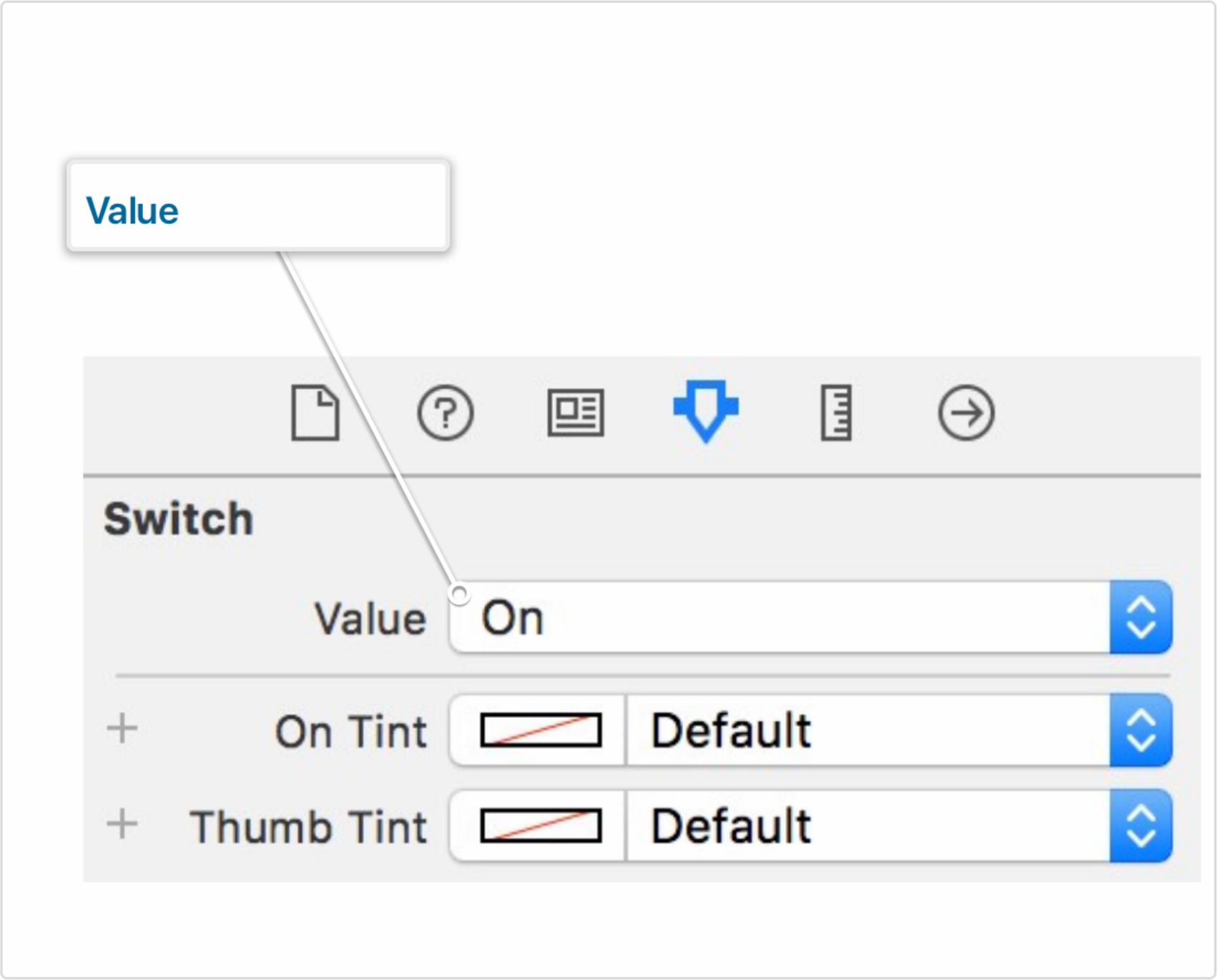
Switches execute code as the user changes the value of the slider.

UISlider Developer Documentation

Switches (`UISwitch`)

A switch lets the user turn an option on or off. You've probably noticed—and used—switches throughout the Settings app.

Configuration



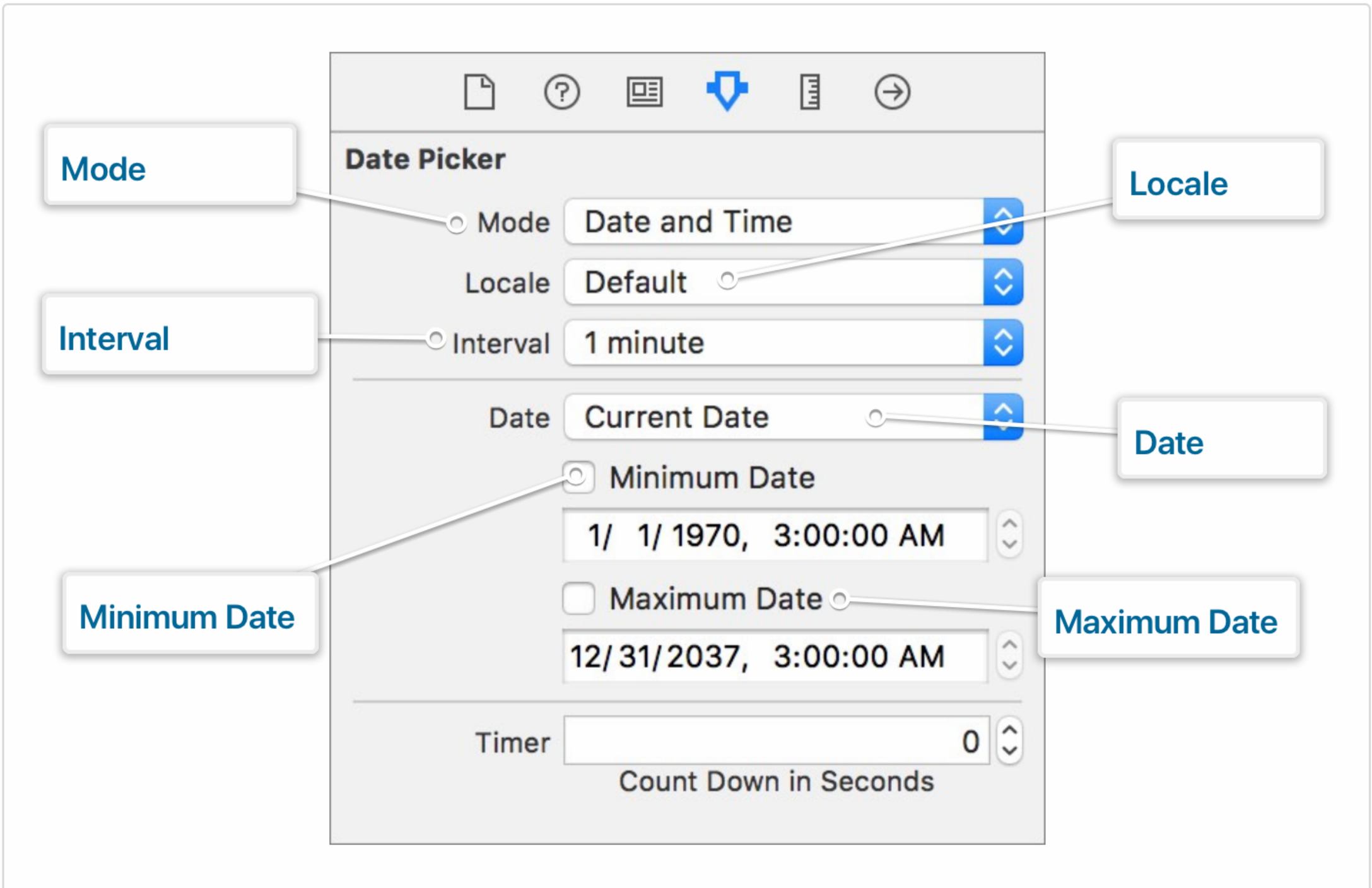
Switches execute code as the user toggles the control.

UISwitch Developer Documentation

Date Pickers (`UIDatePicker`)

Date pickers provide a straightforward interface for managing date and time selection, allowing users to specify a particular date quickly and efficiently.

Configuration



Date pickers execute code whenever the value of the selected date or time is changed.

UIDatePicker Developer Documentation

VIEW CONTROLLERS

`UIKit` defines a special class that controls a view, sets up child views, controls what they display, and responds to user interaction. This class is called the `UIViewController`.

Most commonly, each screen in an app is represented by a scene in a storyboard, and each scene in a storyboard is associated with a subclass of `UIViewController`. The associated `UIViewController` subclass is defined in a `.swift` file that holds all of the logic that controls the scene. Each `UIViewController` class has a `view` property that represents the parent view of the scene.

Think back to the Light project you built in the first unit. The project has a single storyboard with a single scene. That scene was linked to a `UIViewController` subclass called `ViewController`. The light was toggled by adjusting the `view` property

of `ViewController`. That `view` property is the same instance of `UIView` as the parent view of the scene in the storyboard.

When you added a button to the screen, the button became a child view of the scene's main view. When you wired up actions and outlets, you linked them to the `ViewController.swift` file, which defined the view controller for that scene.

You'll learn how to make complex view controllers in later lessons. But for now, it's enough to understand that each different type of screen you see in an app is managed by a different type of view controller.

WHERE TO LEARN MORE

You've just learned about the most common views and controls in `UIKit`. But where would you go to find out more about these and other `UIKit` tools? As you learned earlier, Apple has large teams dedicated to writing documentation to help developers like you learn more about the system tools for building apps.

Most of the information for this section is from a set of guidelines called the Human Interface Guidelines, which was written and maintained by one of those teams. When you want to learn more about `UIKit`, you can go to developer.apple.com and search for more information about the topics in this lesson.

LAB—UIKIT SURVEY

The objective of this lab is to identify different views and controls in some of the most common system apps on iOS. You'll use Simulator to look through Settings, Contacts, News, and Calendar, and then use Pages to complete a survey of your findings.

STEP 1: CREATE A PAGES FILE FOR YOUR SURVEY

- Create a new Pages document called "1.8 UIKit Survey," and save it to your project folder.
- Add section headers for six views: label, image view, text view, table view, navigation bar, and tab bar.
- Add section headers for five controls: button, segmented control, text field, slider, and switch.

As you go through the following steps, one app at a time, you'll add screenshots to the sections you just created. For each app, you don't need find *all* the views and controls, but make sure that each section in your survey includes at least one screenshot.

STEP 2: SURVEY THE SETTINGS APP

- Open the Simulator application and click Settings.
- Navigate through the Settings app to find examples of at least five views and at least three controls.
- For each example, take a screenshot using the Shift-Command-4 keyboard shortcut, and add the screenshot to your document under the correct heading.
- If you come across an unfamiliar view or control, use the UIKit User Interface Catalog to identify it and add it to your document under a new section heading.

STEP 3: REPEAT FOR CONTACTS, NEWS, AND CALENDAR

- Repeat the search for Contacts, News, and Calendar.

STEP 4: REVIEW THE EXAMPLES

- Review the document with a partner or with someone who isn't familiar with iOS development.

REVIEW QUESTIONS

문제 1(총 5개)

What is the main distinction between a view and a control?

- A. Views respond to user interaction; controls display information.
- B. Views display information; controls respond to user interaction.

정답 확인

Lesson 2.8

Displaying Data

In the last lesson, you learned about common views and controls. Now you can put that knowledge to use.

In this lesson, you'll use Interface Builder to create the beginnings of an app, called Hello, that you can use to introduce yourself. You'll plan out the content for the app, and begin to build it by adding labels and a profile image.

What You'll Learn

- How to configure views using Interface Builder
- How to customize a label
- How to customize an image view

Vocabulary

- aspect ratio
- clipping
- content mode
- dynamic data
- frame
- static data

Related Resources

- API Reference: UILabel
- API Reference: UIImageView

In this lesson, you'll plan out and build a simple app called Hello, an app that you could use to introduce yourself. As you work, you'll apply what you've learned about some of the common views in `UIKit`.

PLANNING THE APP

Most apps start with a simple idea, which can usually be summed up in one short sentence. Think of apps that you use every day. They may have started with the following straightforward ideas:

- "Send and receive messages."
- "Share photos with my friends and family."
- "Post short messages, and see the short messages that others have written."

As an app developer, you'll learn how to take a one-sentence overview of an app or a feature and plan out how to make it a reality.

In this project, your task is to build an app for introducing yourself. This guide will give you a framework for what to include, but the app will introduce *you*. Bring in your own ideas, and make it yours.

Think about how you would introduce yourself to someone you haven't met before. What information would you want to make sure someone knew about you? Start by writing down your name, then take a few minutes to write down what you might share about yourself. Use the following prompts to help you come up with enough information for your app.

- Who are you? What is your life all about?
- What does a day in your life look like?
- If you had all the time and money you needed, what would you do with your life?
- What are some of your favorite things or activities?

Include any other information you think other people would want to know.

Sometimes the information you want to display in an app is consistent and never changes. In programming, this information is called static data. As you plan your app, you're brainstorming static data about yourself. In future apps, you'll work with data that changes, which is called dynamic data.

Example

```
## About Me
My name is Alexis Kay. I'm a dedicated student, I play lacrosse,
I love animals, and I'm looking for an internship in Marine
Biology.
Location: Half Moon Bay, California
Website: www.example.com

## Day in the Life
Eat breakfast
Go to school
Go to lacrosse practice
Eat dinner
Hang out with friends
Watch TV

## When I Grow Up
I want to sail around the world, spend time in new countries,
learn new languages, and meet awesome people everywhere I go.

## Favorites
Food: Ice cream
Sport: Lacrosse
Book: Harry Potter
```

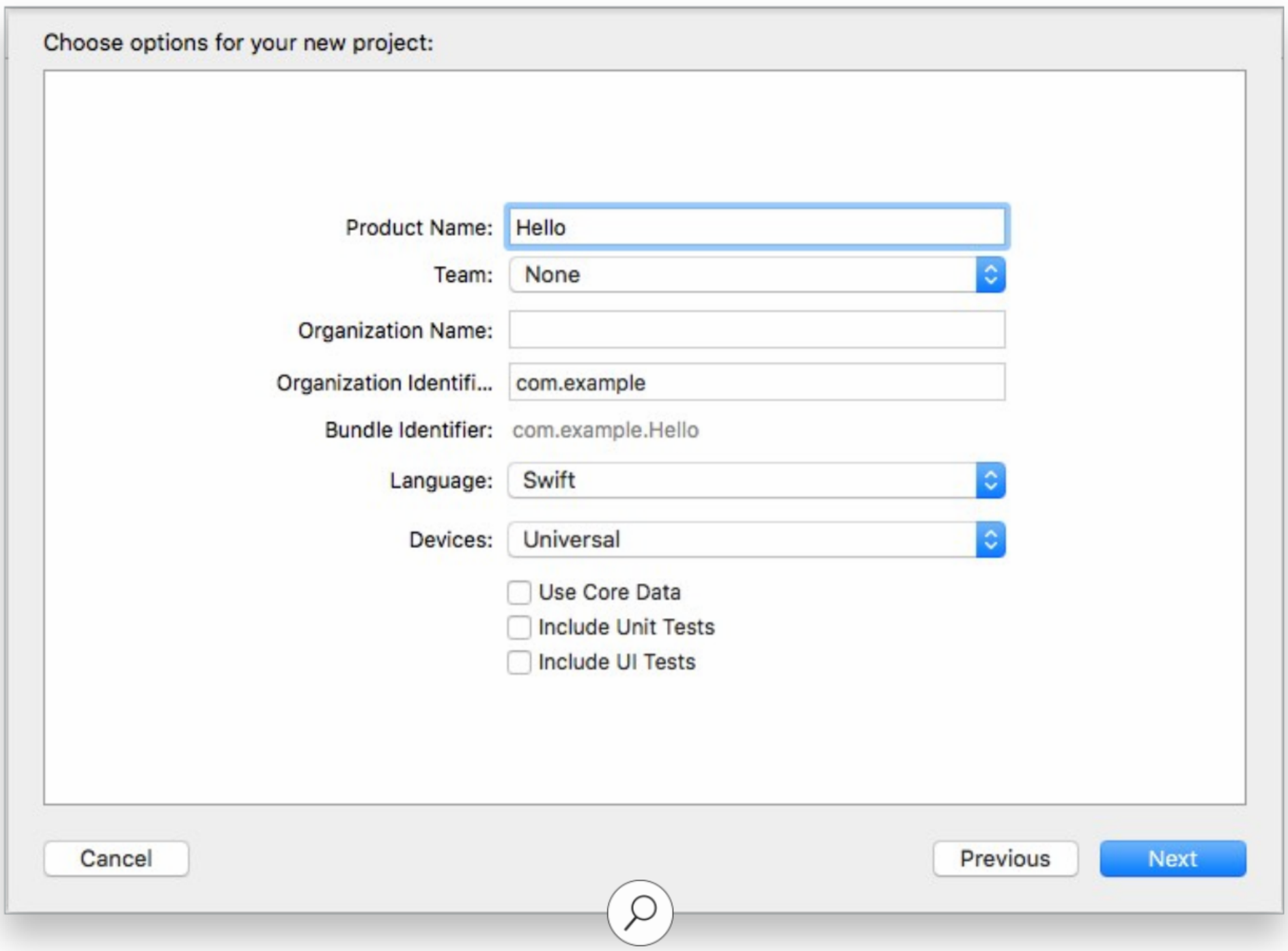
As you come up with information that's unique to you, you can probably imagine many ways to display it and share it with others. For this project, try to think about how you might display this information on a single screen.

Since you're still in the planning phase, you might find it helpful to open Pages or Keynote and create a rough outline of the data you want your app to display.

You'll probably discover that all the information you want to include won't fit on a single iOS device screen—which means you'll need to enable scrolling. In this lesson, you'll create part one of the project, adding your name, a profile image, and a few sentences about yourself. In a future lesson, you'll complete part two, using more advanced layout features in Interface Builder to add images and labels in a scroll view.

CREATE THE PROJECT

Create a new Single View App, name it "Hello," and save it to your project folder.

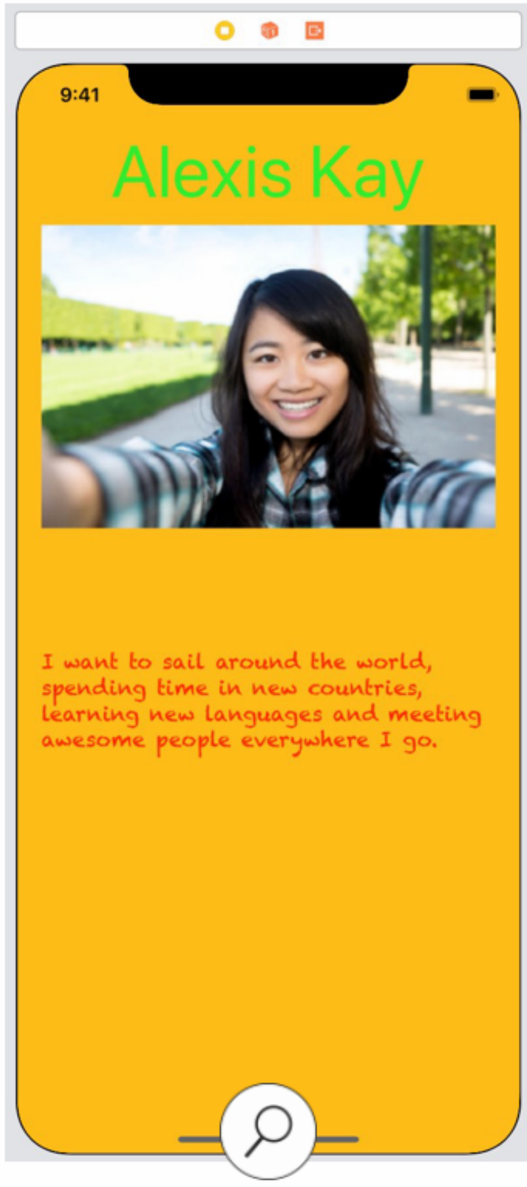


ENTER YOUR INFORMATION

Open the `Main.storyboard` file. You'll use the initial scene to add your information. Because your app will always display the same data, you can build your static data directly into your screens using labels or images in Interface Builder. In future lessons, you'll learn to write code for displaying dynamic data.

As you choose fonts, colors, and other display settings, do your best to keep a clean-looking presentation that fits on an iOS device. Avoid unreadable fonts, conflicting color schemes, and confusing text layouts.

Bad Example

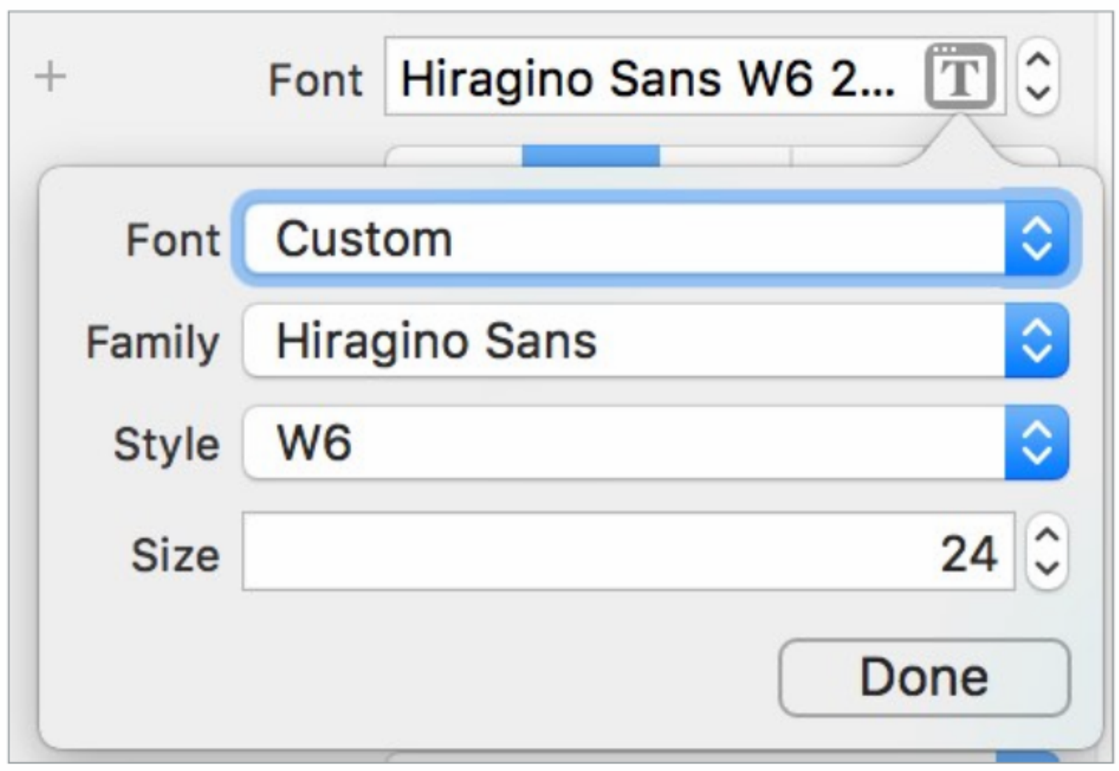


As you place views, take advantage of the layout guides to center your content and be careful to observe margins and allow sufficient spacing between objects. You'll also want to avoid putting content in the status bar at the top of the screen.

Add Labels for Displaying Text

For each of the text items you want to include, drag a Label object from the Object library to the scene. At a minimum, you'll want to add labels for your name and one or two sentences about yourself. You can add the rest when you finish this project in a future lesson.

Selecting one label at a time in the storyboard, use the Attributes inspector to set the label's text and choose an appropriate font. If you want a label to display as a header, make the font bigger or bolder than the default text. Do you want to use a custom font? Open the Font pop-up menu and choose Custom to select from all the fonts installed on your Mac.



Experiment with different attributes in the inspector to see how each of them modifies the appearance of your labels. For example, you can change the alignment of the text or the number of lines you want the label to display. By default, labels display only one line of text, which may cut off any remaining text at the end of the line.

Alexis is a dedicated student looking for int...

When working with static information, you can keep increasing the number of lines until you see that the label can display all your text. But there's a better way that allows the text to flow to as many lines as it needs to display the entire text you provide.

Look at the documentation for `UILabel` and find the `numberOfLines` symbol. In the Description section, you'll see the following text:

> This property controls the maximum number of lines to use in order to fit the label's text into its bounding rectangle. The default value for this property is 1. To remove any maximum limit, and use as many lines as needed, set the value of this property to 0.

The `numberOfLines` property is useful when you want a label to display text that flows beyond a single line. Feel free to explore the documentation for `UILabel` to discover other strategies for fitting text into a label.

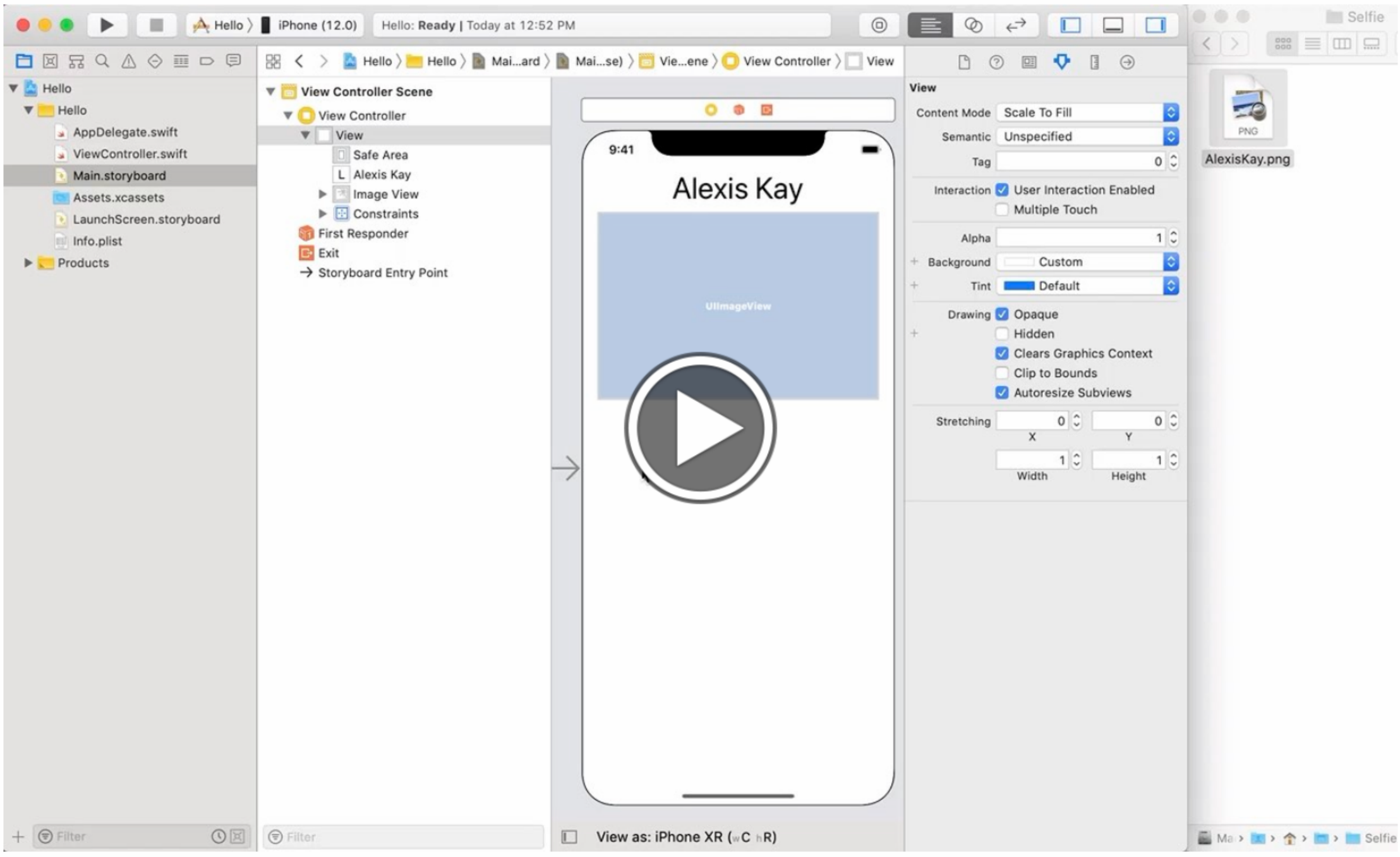
Finish up adding text to all the labels you want to include in your Hello app.

Add an Image

Your introduction wouldn't be complete without a photograph of yourself. If you don't already have an image you like, you can use the Photo Booth app to take a selfie. Make sure you have the image file handy on the desktop or in the Finder. If you have more images, you'll have a chance to add them to the app in a future lesson.

Now go ahead and add the image file to your scene so that the app will display it.

Here's how to add an image to your Xcode project:



1. *Open the navigator area*. Make sure the navigator area on the left is open and that the project navigator shows a list of your files. If you're working on a small screen, you may want to close the utilities area on the right.
2. *Select* `Assets.xcassets`. Xcode has automatically added an asset catalog to your project. This is the best place to keep the images you'll use in your app.
3. *Drag in the image file*. Add the image to your storyboard and give it a name in the document outline. The image is now part of your app.

Now that you've added an image of yourself, you can set up an image view in Interface Builder. Go back to `Main.storyboard`. From the Object library, drag an Image View object to the scene.

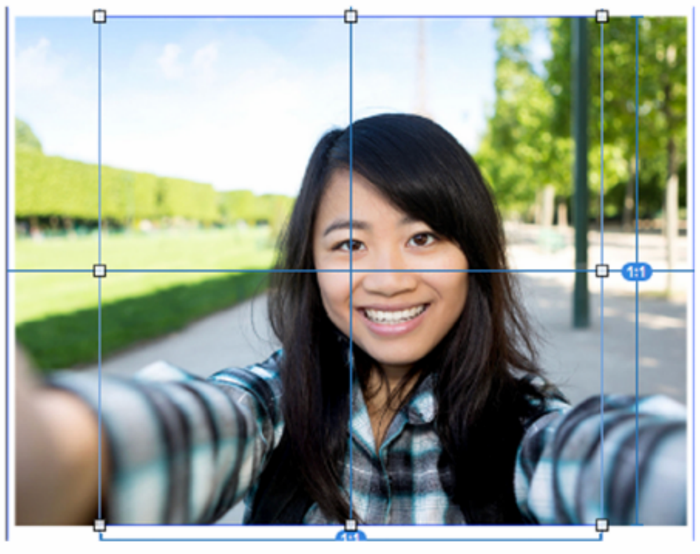
Using the Attributes inspector, choose the name of your image from the Image pop-up menu. This will assign that image to the image view you just created.

In the previous lesson, you learned that an image *view* is like a frame around the *image* it will display. The size of the image view determines the total potential display size of the

오후 10:36 7월 18일 토요일 62%

image. By setting the content mode of the image view, you can control if the image will display within the frame, potentially leaving empty white space, if it will stretch to fit within the frame, potentially distorting the image, or if it will stretch to fill the entire frame, potentially cropping some of the image.

In the table below, you can see the most common `contentMode` options for a square `UIImageView` displaying a non-square image.

Scale to Fill	Aspect Fit	Aspect Fill
Resized to image view bounds	Maintains aspect ratio, resized to fit whole image into the image view bounds	Maintains aspect ratio, resized to fill the image view bounds, overflows the image view frame
		

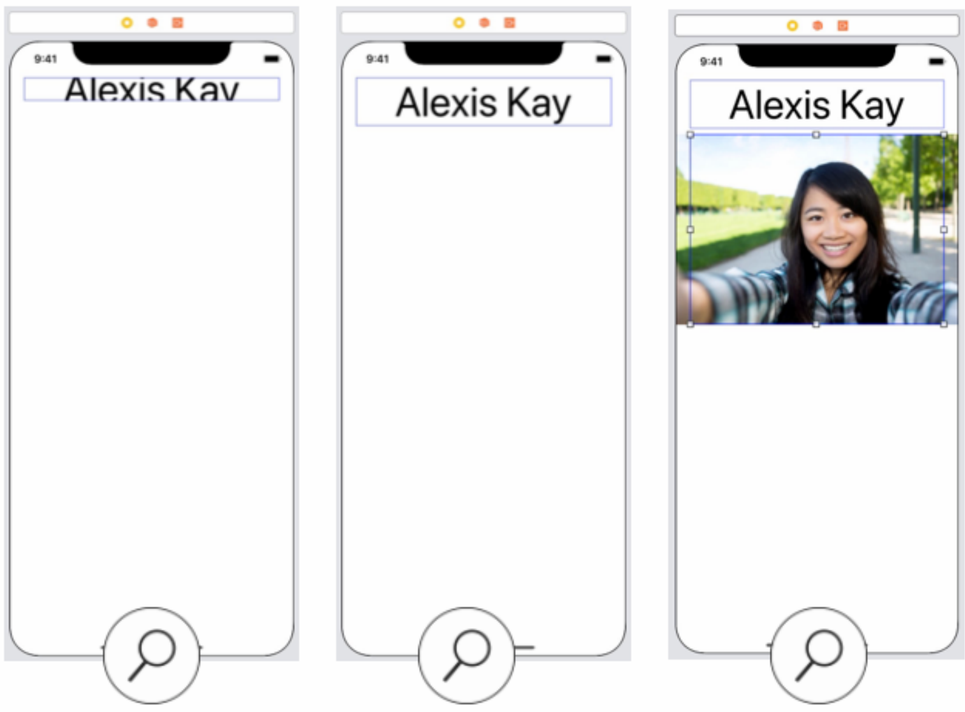
Choose the correct content mode for the image you added to your project.

ADDRESS LAYOUT ISSUES

Unexpected Clipping

It's possible you've encountered occasional warnings or bugs with your layout. Don't worry about these right now. In a later lesson, you'll learn how to resolve them and how to use more advanced layout options, including Auto Layout, in Interface Builder.

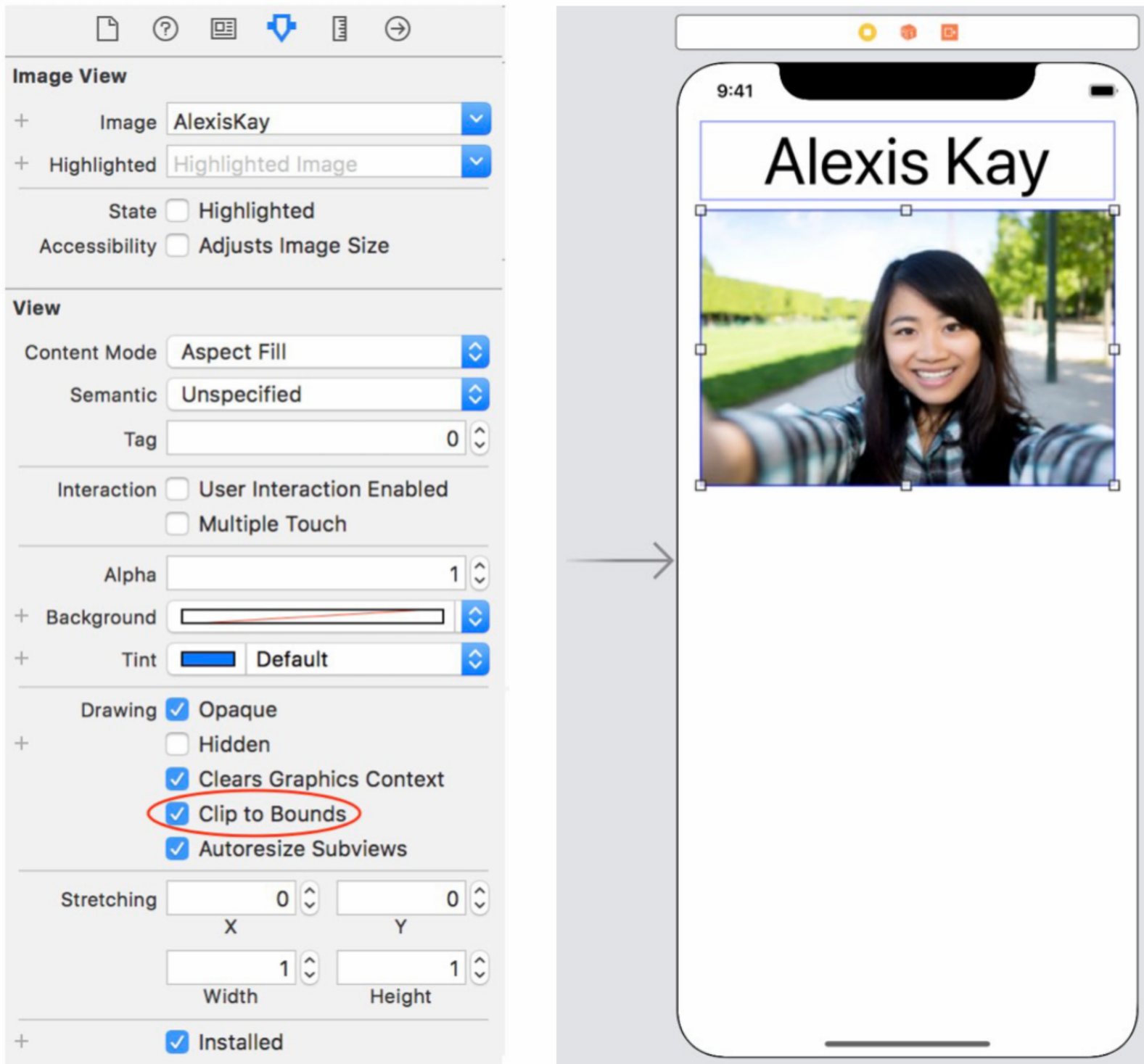
However, if you've run into unexpected clipping (a very common issue), you can easily resolve it now. Just as an image view frames the image it's displaying, a label frames the text it displays. If a label's content is larger than the label itself, it result in clipped text.



Images can also have clipping issues. For example, if you have an image view with a different aspect ration than the image it displays and you choose the Aspect Fill content mode, some of the image will overflow the bounds of the image view.

To fix clipped text, you have two choices: decrease the size of the font or increase the size of the label.

Interface Builder allows you to modify this behavior, or choose whether or not content outside the bounds of the view will be visible.



Placement Errors with Different Screen Sizes

When you learn about Auto Layout, you'll learn how to make your interface scale across various device sizes. But for now, make sure you set up your interface for the same device size that you're using in Simulator. This will ensure that you see a consistent view across your projects.

Save Your Project

You've now built a simple Single View App with a few labels and one image. Be sure to save your project in your project folder. You'll continue building this project in the "Auto Layout and Stack Views" lesson, where you'll learn how to use advanced layout tools to position your images and labels, and how to use a scroll view to add more content.

LAB—TUTORIAL SCREEN

The objective of this lab is to create a tutorial about a hobby you enjoy. You'll use Interface Builder to set up a view with relevant images and text.

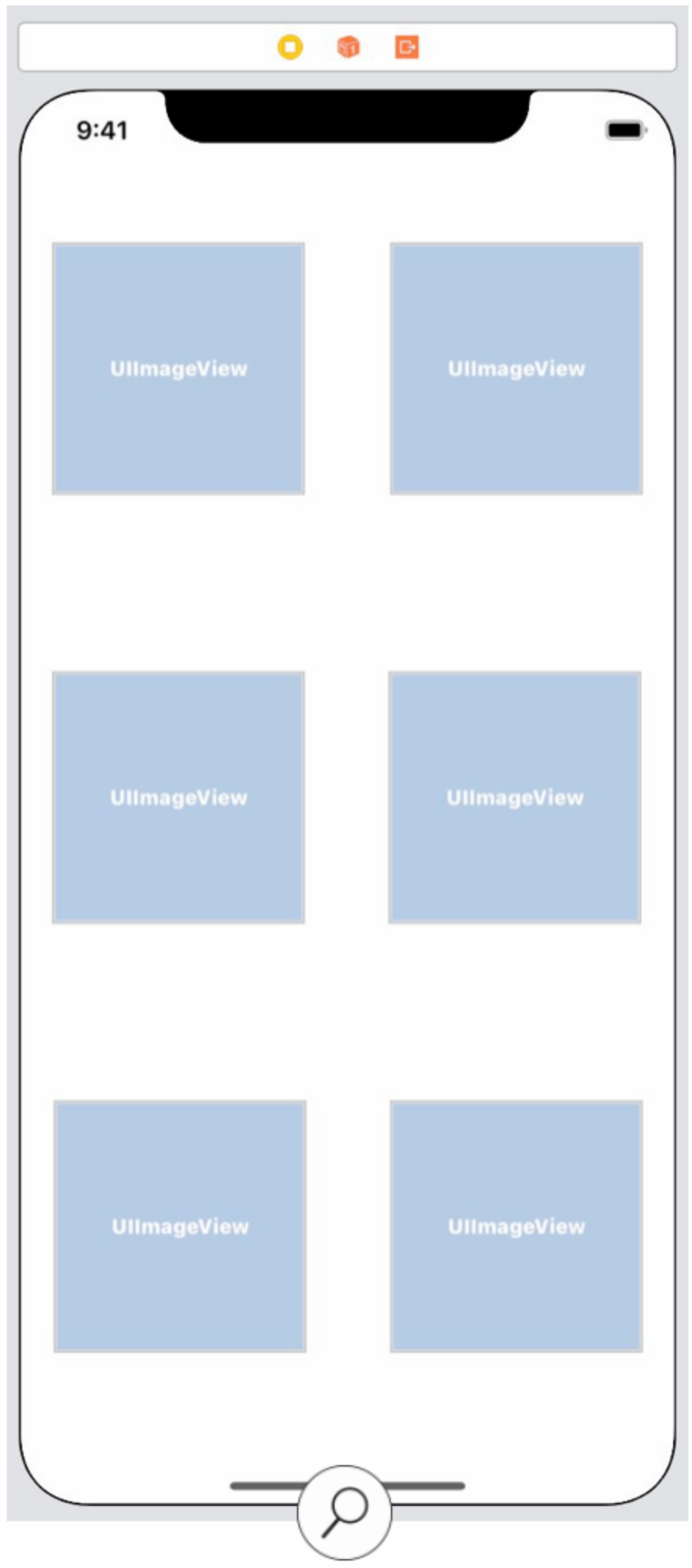
Create a new project using the iOS template for a Single View App and name it "Hobby Tutorial."

STEP 1: CREATE A PROJECT AND ADD IMAGES

- Choose three to six images that explain something about your hobby.
- Add the images to your project by dragging them into `Assets.xcassets`.
- Assign appropriate names for each image in the asset catalog.

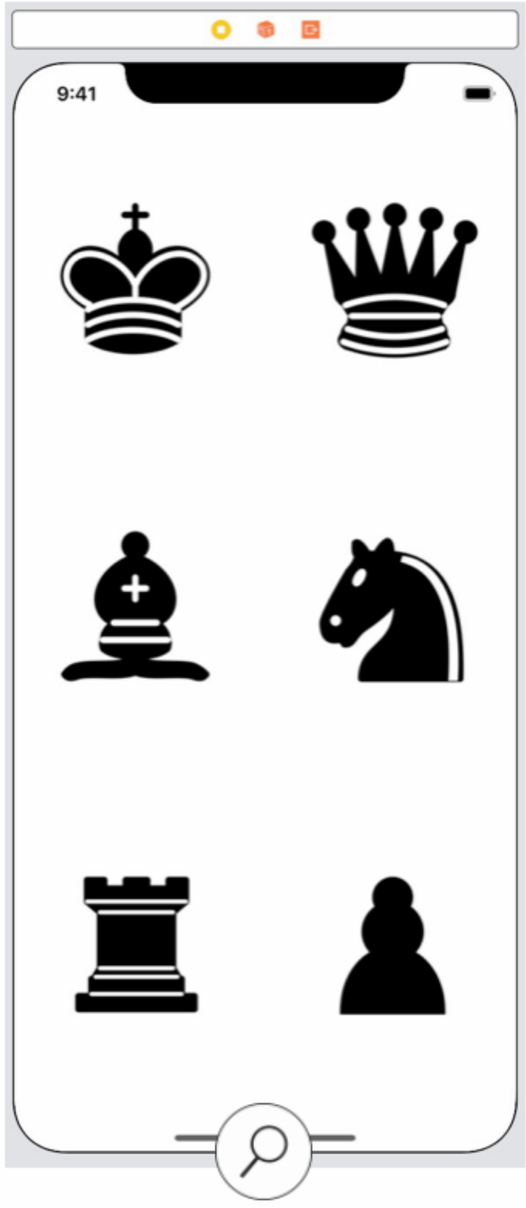
STEP 2: SET UP THE IMAGE VIEWS

- Add an image view in storyboard for each of the images. Use the layout guides to set up a clean, well-balanced interface.



오후 10:37 7월 18일 토요일 62%

- Assign the images to the image views using the Attributes inspector.



STEP 3: SET UP THE LABELS

- For each image, add a label and enter a brief description of the image as the text. Use an appropriate font.

Excellent work! Labels and image views are two of the most common views for displaying data in an app. Practice using labels and images in new ways to display different kinds of information. Be sure to save your project to your project folder.

Lesson 2.9

Controls In Action

Two lessons ago, you learned about common views and controls. In the last lesson, you had a chance to practice creating labels and image views. Now you can take another step forward by setting up controls and responses to control events.

In this lesson, you'll use Interface Builder to add buttons, switches, and sliders to a scene. You'll also create actions and outlets, write some basic code, and gain an understanding of how these tools work together.

What You'll Learn

- How to use a button to execute code
- How to use a switch to execute code
- How to access the value of a switch
- How to use a slider to execute code
- How to access the value of a slider
- How to use a text field
- How to access the value of a text field
- How to response to user interactions with gesture recognizers
- How to connect controls to actions programmatically

Vocabulary

- gesture recognizer

Related Resources

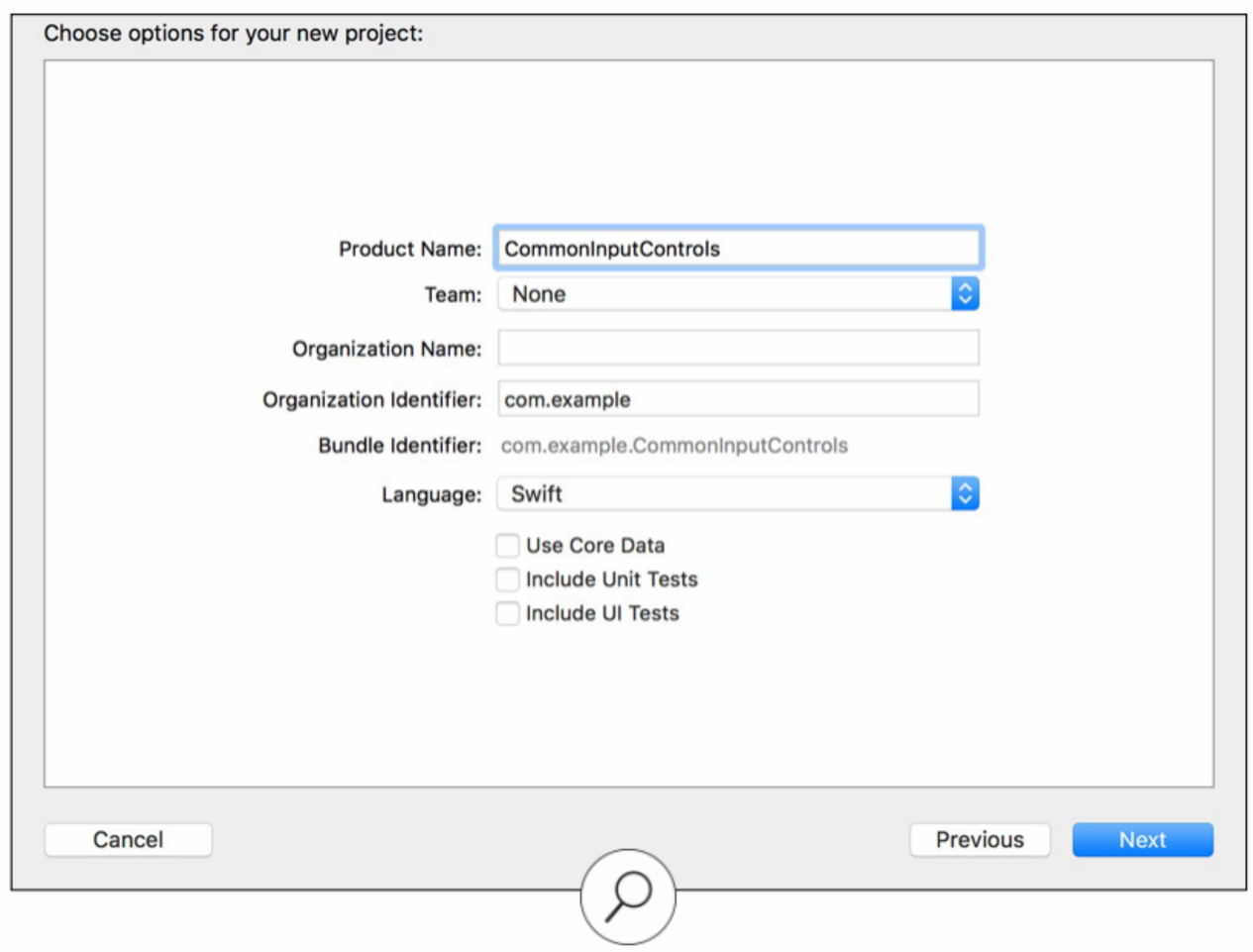
- API Reference: UIButton
- API Reference: UISwitch
- API Reference: UISlider
- API Reference: UITextField
- API Reference: Gesture Recognizers

When you first learned about Interface Builder, you added a button to the screen and printed when the button was tapped. In this lesson, you'll review that exercise with your new knowledge of views and controls, and you'll repeat it with switches and sliders.

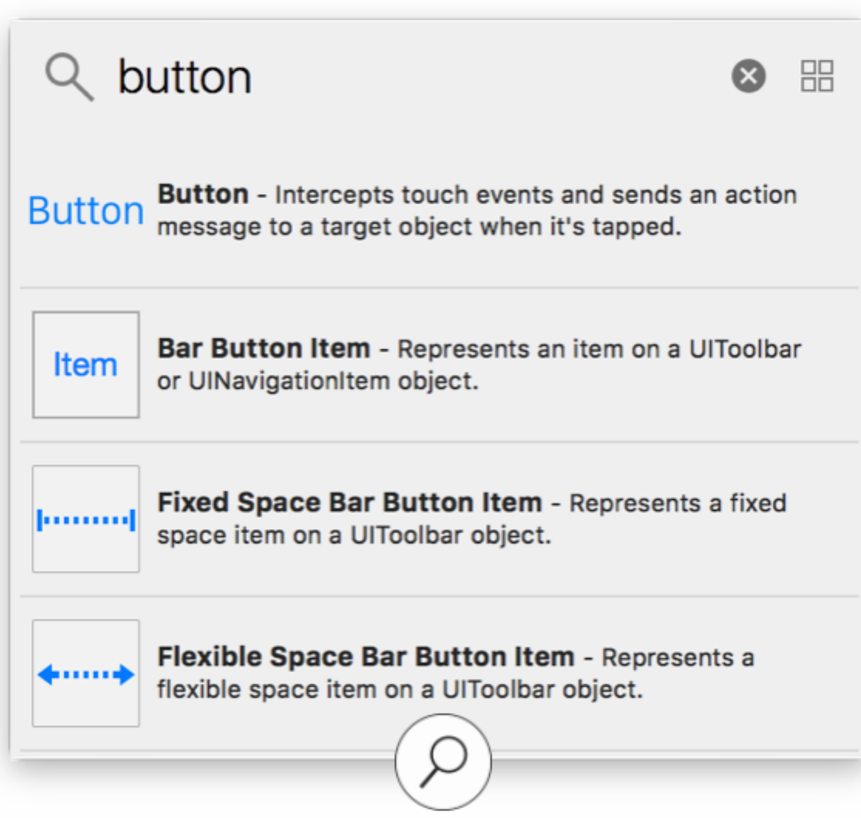
BUTTONS

As you know from your own direct experience with apps, buttons are the most common type of input control. Add a button to your main scene and have it print a statement to the console when it's tapped.

1. Create a new project called "CommonInputControls" using the Single View App template.

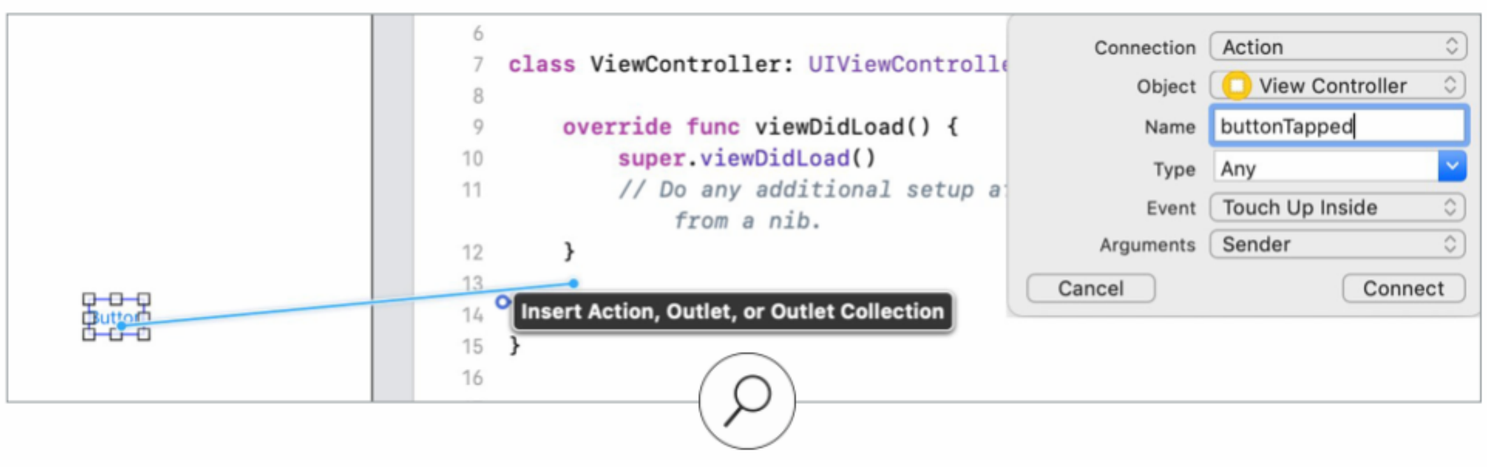


2. Open `Main.storyboard`. Then find a Button object in the Object library.



3. Drag the button to the scene, placing it about a third of the way from the top of the screen. Open the assistant editor to see the corresponding `ViewController.swift` file.

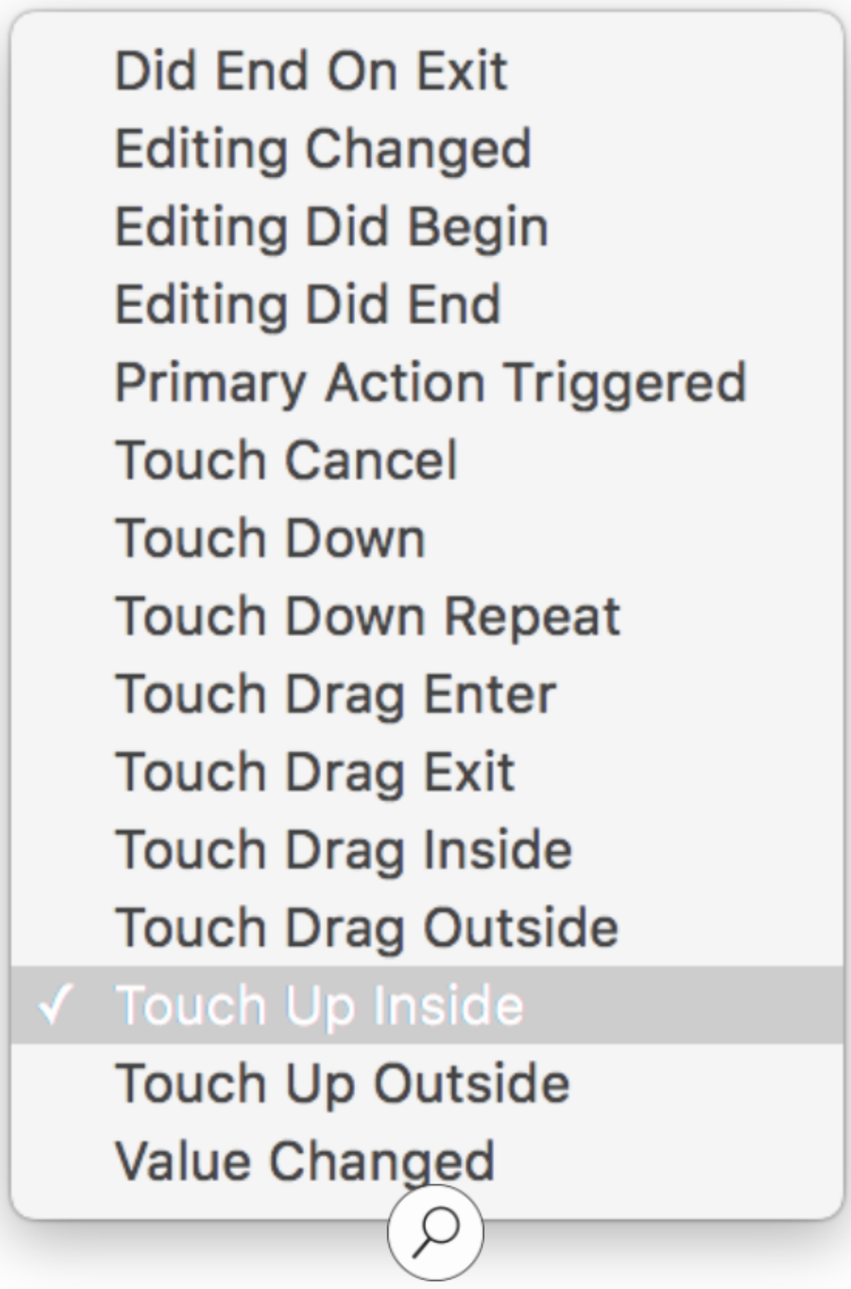
4. Using the Control-drag shortcut, add an `IBAction` from the button to the view controller code. Remember to add the action within the curly braces that define the `ViewController` class. On fresh view controller files, like this one, most developers add new actions below the `viewDidLoad()` block of code.



Use a descriptive name for your action, such as `buttonTapped`. Later on, when you look at the code in the `ViewController.swift` file, you'll be glad that the name of the action gives you a clue as to what will trigger it.

The most natural time for a button to execute code is when the user taps it and releases the touch from within the bounds of the button. That's why "Touch Up Inside" is the default control event when you create an `IBAction` from a button. You can also use the

"Primary Action Triggered" control event, which will be triggered at the same time as "Touch Up Inside." If you want code to execute in response to a different control event, you can choose from the options in the popover.



Remember that when you create an `IBAction` the function is passed to the `sender` as a parameter. The `sender` refers to the specific control that triggered the action. In this case, the `sender` is the tapped button.

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

}
```

When you know that the `sender` will *always* be the same type of object (like the button in this example), you can change `Any` to the specific control type (`UIButton` in this example). To make this change, you'll access properties on `sender` within the `IBAction`.

5. Use the `print()` function to print a line of text to the console when the user taps the button.

```
print("Button was tapped!")
```

You can put any code or logic inside the control's action using this same workflow.

6. Run the app in Simulator, and try clicking the button you just created. You should see the printed text in the console area.

Right now, the text on the button doesn't relate to the action it will trigger. Use Interface Builder to update the button's text to something more descriptive of the code it will execute.

You can also set other button attributes, including its font, alignment, and shadow. To get a feel for the things you can adjust, take a moment to play around with the various options in the Attributes inspector.

오후 10:37 7월 18일 토요일 62%

SWITCHES

Switches are used to toggle on a singular option. You can use an `IBAction` to execute code when a switch is toggled one way or another. Or you can check if the switch is currently toggled to the on or off position by accessing the `isOn` value from the `sender` parameter or from an `IBOutlet`.

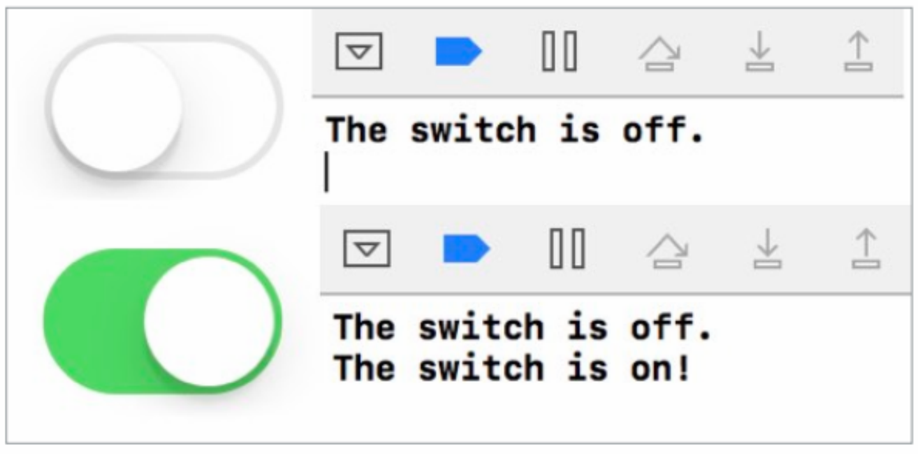
Add a switch to your CommonInputControls project, and print whether the switch is on or off when the user toggles the control.

1. Find a Switch object in the Object library, and drag it to your scene.

2. Using the assistant editor, add an `IBAction` from the switch to the `ViewController.swift` file. As with button actions, use a descriptive name, such as `switchToggled`, and set the sender type to `UISwitch`. Add code that will print to the console whether the switch was turned on or off.

```
@IBAction func switchToggled(_ sender: UISwitch) {
    if sender.isOn {
        print("The switch is on!")
    } else {
        print("The switch is off.")
    }
}
```

3. Run the app in Simulator, and toggle the switch. You should see the printed text in the console area.



SLIDERS

Sliders allow the user to have smooth control over a value, such as adjusting a setting. For example, in Control Center, you use sliders to adjust the volume and the display brightness of your device.

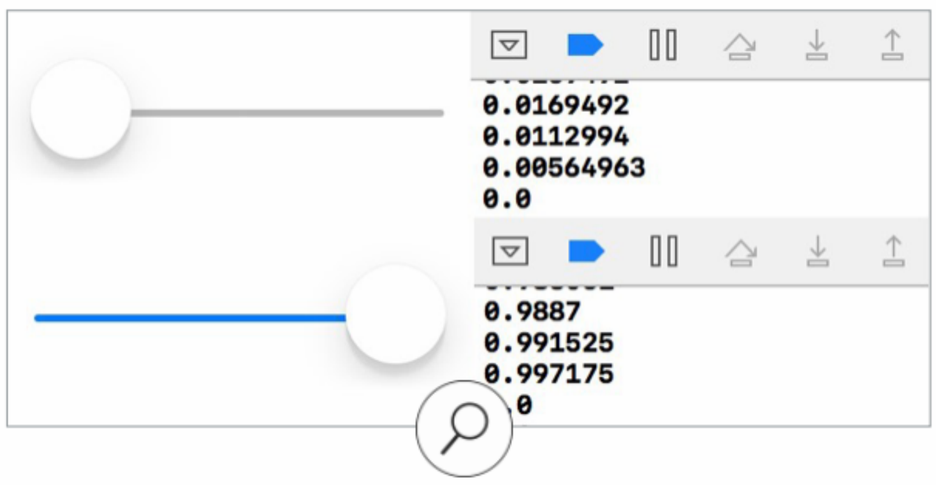
Add a slider to your CommonInputControls project, and print the slider's value as it changes.

1. Find a Slider object in the Object library, and drag it to your scene.

2. Using the assistant editor, add an `IBAction` from the slider to `ViewController.swift`. Use a descriptive name, such as `sliderValueChanged`, and set the sender type to `UISlider`. Add some code that will print the value of the slider to the console.

```
@IBAction func sliderValueChanged(_ sender: UISlider) {
    print(sender.value)
}
```

3. Run the app in Simulator, and move the slider. You should see the printed value in the console.

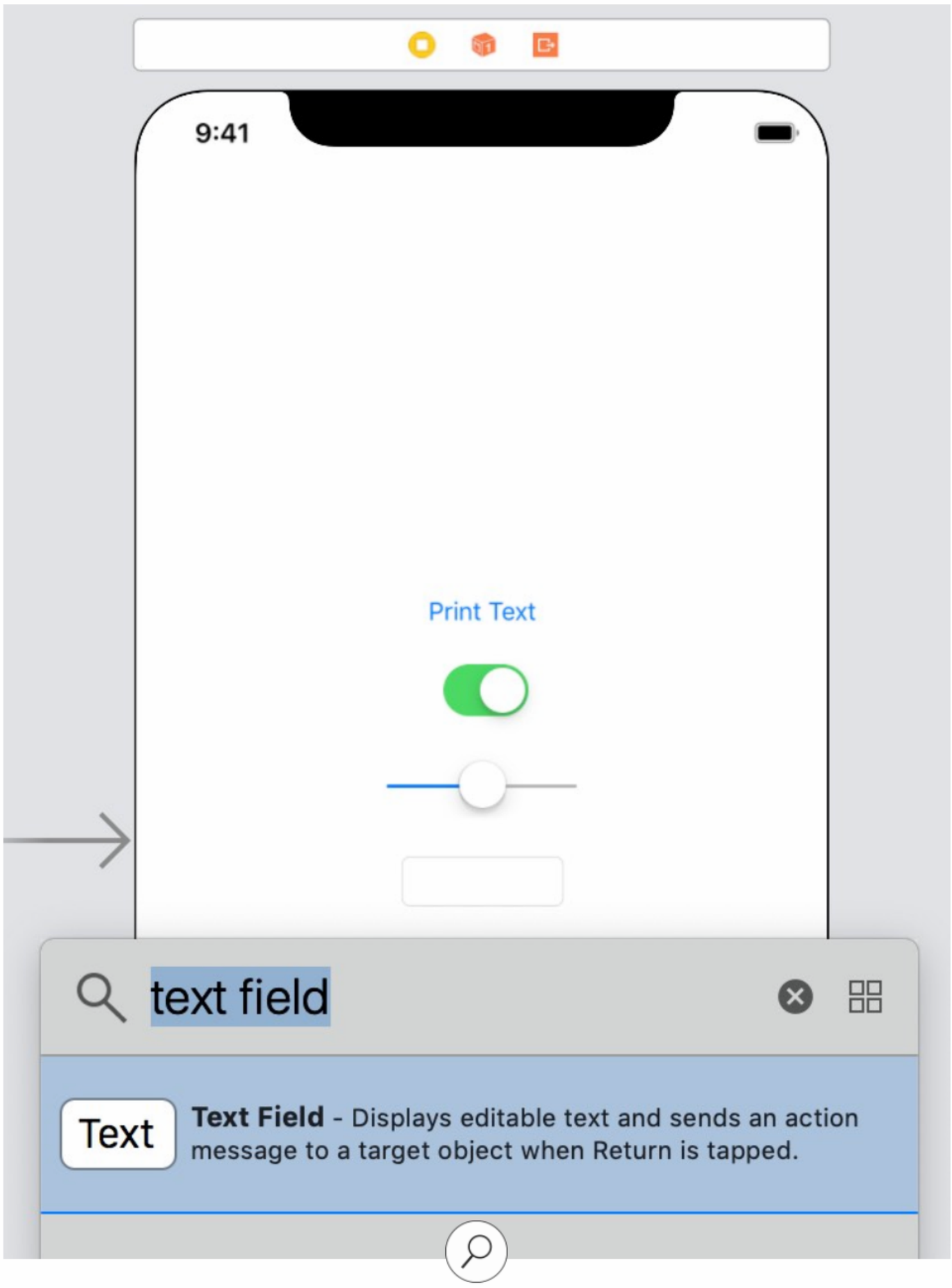


TEXT FIELDS

Text fields allow the user to enter a small amount of text, such as entering a username or password. For example, you use a text field when entering a subject on a new email draft.

Add a label and a text field to your CommonInputControls project, and set the label's text to the current text in the text field when the user taps the Enter or Done key on the keyboard.

1. Find a Text Field object in the Object library, and drag it to your scene.



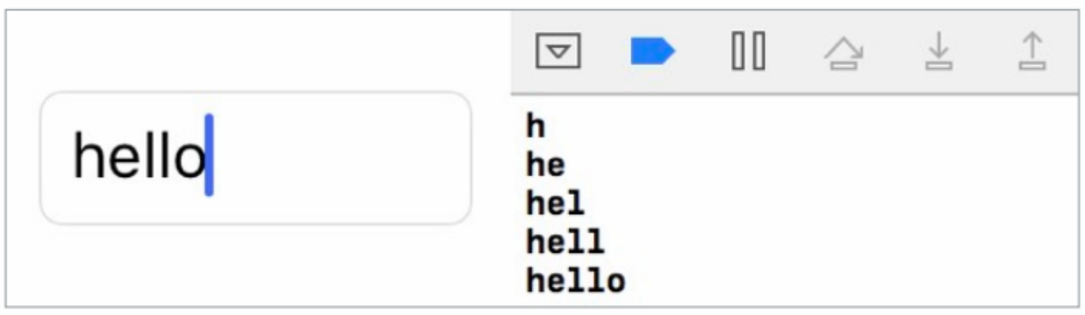
2. Using the assistant editor, add an `IBAction` using the "Primary Action Triggered" event from the text field to `ViewController.swift`. Use a descriptive name, such as `keyboardReturnKeyTapped`, and set the sender type to `UITextField`. Use the action to print the current text of the field.

```
@IBAction func keyboardReturnKeyTapped(_ sender: UITextField) {
    if let text = sender.text {
        print(text)
    }
}
```

3. Remember that text fields can also trigger code when the user edits the contents of the field. Using the assistant editor, add an `IBAction` using the "Editing Changed" control event from the text field to `ViewController.swift`. Use a descriptive name, such as `textChanged`, and set the sender type to `UITextField`. Use the action to print the current text of the field.

```
@IBAction func textChanged(_ sender: UITextField) {

    if let text = sender.text {
        print(text)
    }
}
```



Text fields are very flexible and should be configured to best fit the situation they are used in. For example, you can set the keyboard to display different keys based on whether the user is entering an email address or a URL. You can also turn autocorrect on or off, or set the text field to a secure input field to hide the characters as the user types them. Take some time to explore the options and use them appropriately when building your app.

ACTIONS AND OUTLETS

You can create outlets that allow actions to access properties on your views and controls —even if the view or control isn't the sender.

For example, you may have an outlet for a label that gets updated by a button's action.

In the steps below, you'll change the `buttonTapped` function triggered by the tapped button to access the current `isOn` state of the switch and the current `value` of the slider .

1. Using the same Control-drag technique in the Assistant Editor, add `IBOutlet` references for the switch and slider. Developers usually place `IBOutlet` references above the `viewDidLoad()` function in a view controller file. Just as when naming an `IBAction`, you should choose a descriptive name for your `IBOutlet`. You might think to choose `switch` and `slider` for this `IBOutlet`, but there's a problem with `switch`. It turns out that "switch" is a reserved keyword in Swift, so the best practice is to use `toggle` to name an `IBOutlet` for a switch.

```
@IBOutlet var toggle: UISwitch!
@IBOutlet var slider: UISlider!
```

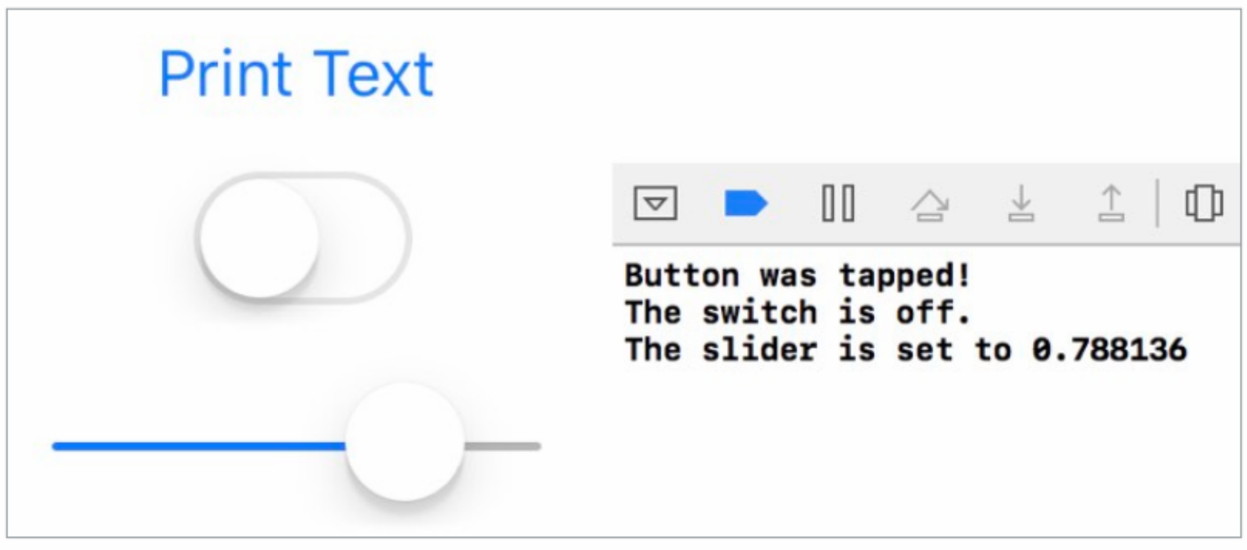
2. Update the `buttonTapped` function in the `IBAction` from the button to print the current `isOn` state and the value of slider.

```
@IBAction func buttonTapped(_ sender: UIButton) {
    print("Button was tapped!")

    if toggle.isOn {
        print("The switch is on!")
    } else {
        print("The switch is off.")
    }

    print("The slider is set to \(slider.value)")
}
```

3. Run the app in Simulator and click the button. You should see the printed text in the console.

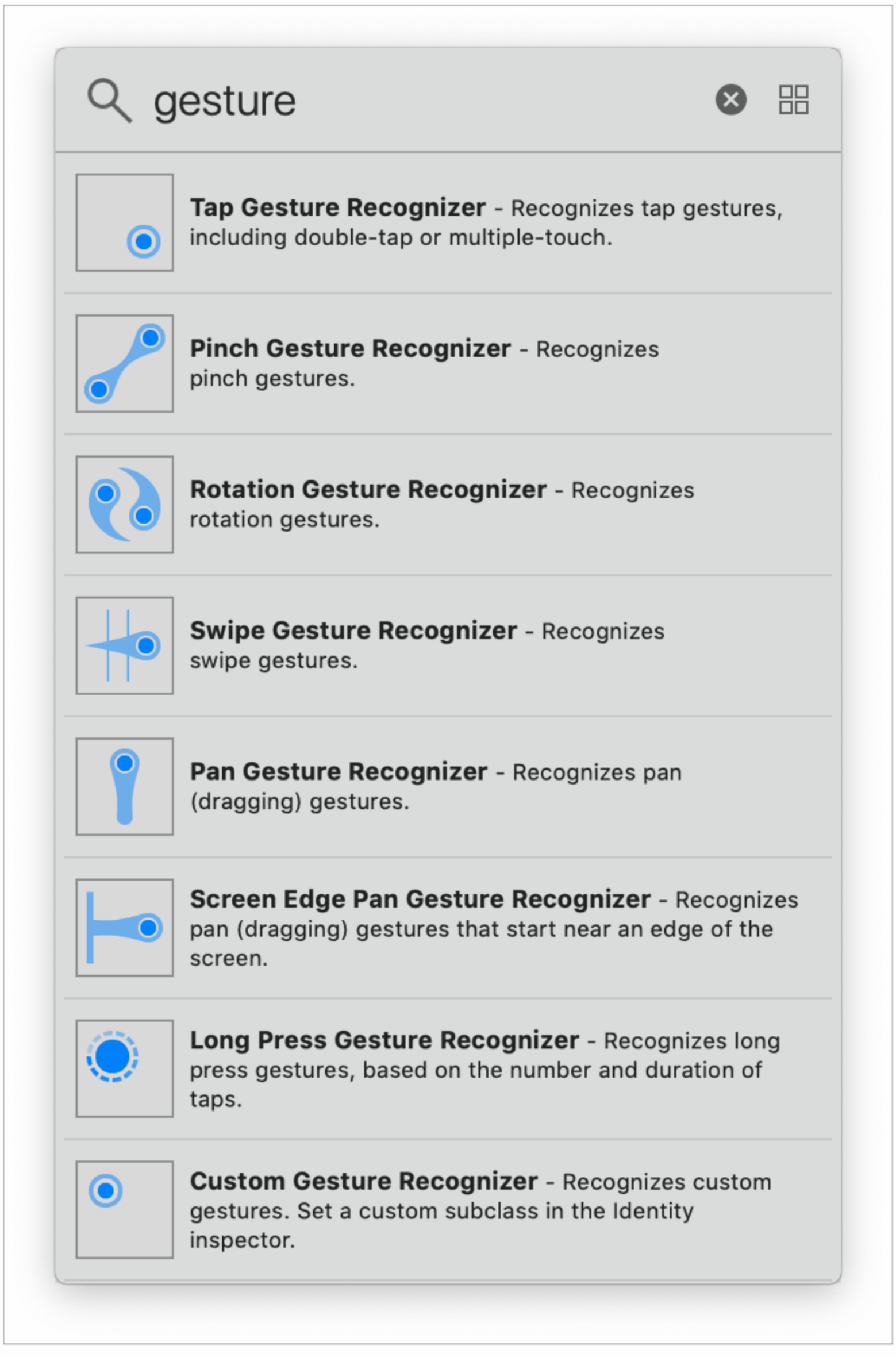


GESTURE RECOGNIZERS

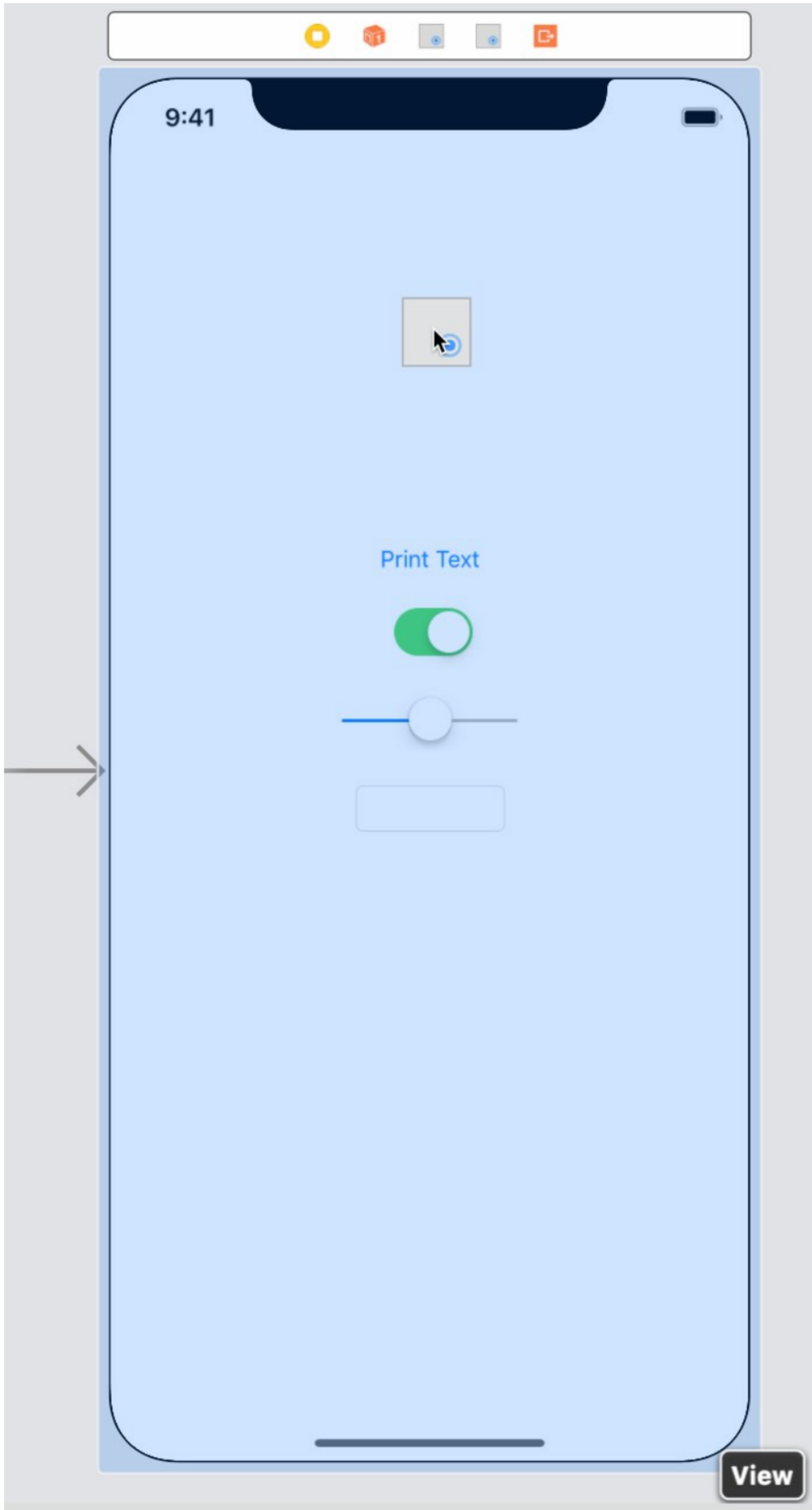
Each of the controls interact with gestures: buttons and switches ares configured to work with taps, sliders work with swipes, etc. A gesture is tied to a single view, but a view can have multiple gestures. For example, a switch responds to a tap, but the user can also swipe a switch left and right to turn it on and off. You can add your own gesture recognizers on top of any view you wish, from simple taps to complex pinches, long-presses, and rotations.

In this example, you'll add a tap gesture recognizer on top of your view controller's view. To begin, search the Object library for "gesture" to see the full list of available gesture recognizers.

오후 10:37 7월 18일 토요일 61%



Grab the Tap Gesture Recognizer, and drag it on top of the view that should respond to the tap event. In this case, drag it on top of the view controller's view.

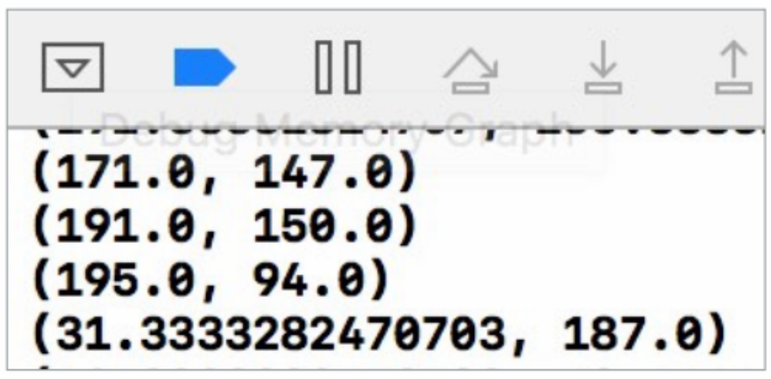


You will see the recognizer added to the Document Outline. Select the recognizer from the Document Outline and open the Attributes inspector. Different gestures will contain a different list of properties. You can adjust the number of taps that the recognizer requires, such as a single or double-tap, as well as the number of touches (or fingers) required to trigger any actions connected to the recognizer. In this example, you want to recognize a single tap using a single finger, so leave both values at 1.

Using the Assistant editor, add an `IBAction` from the recognizer to `ViewController.swift`. Use a descriptive name, such as `respondToTapGesture`, and set the sender type to `UITapGestureRecognizer`. The following implementation of `respondToTapGesture` will determine where the user tapped when the recognizer's action was called, and will print out the x/y value to the console.

```
@IBAction func respondToTapGesture(_ sender:
UITapGestureRecognizer) {
    let location = sender.location(in: view)
    print(location)
}
```

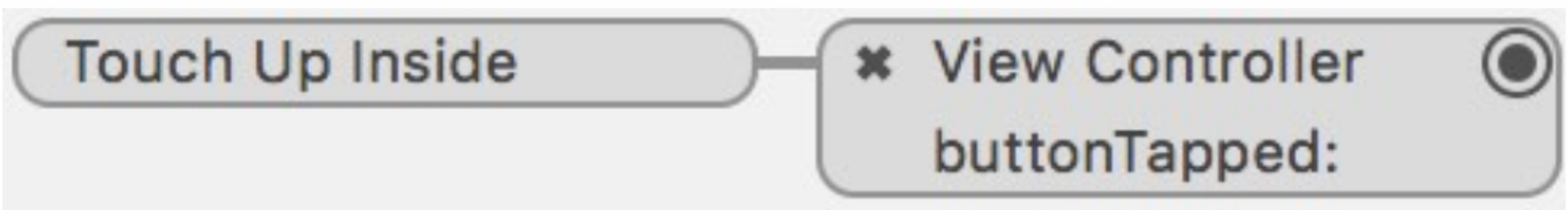
Build and run your application, then try tapping on the screen. The coordinates of your tap will be printed. Notice how the action does not trigger when you interact with the other controls on-screen, because the gesture is only tied to the view controller's view.



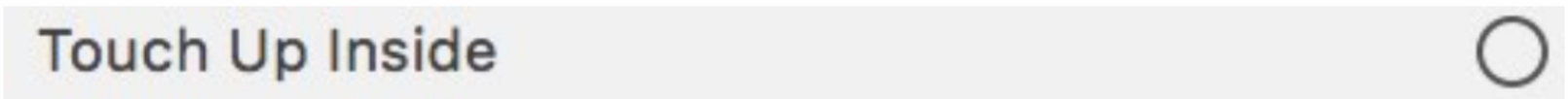
PROGRAMMATIC ACTIONS

How *does* Interface Builder connect controls to actions? It is helpful to understand how to connect controls to actions through code. At the very least, it will help you understand the work that Interface Builder is doing on your behalf.

To demonstrate connecting a control to a method programmatically, you will disconnect the existing button from the `buttonTapped(_:)` action and re-connect the two in code. Highlight the button in your storyboard, then open the Connections inspector. In the list of control events, you can see that the button executes the `buttonTapped(_:)` method when your finger presses down, then up, inside the control.



Tap the 'x' next to the method name, breaking the connection between the button and the action. Touch Up Inside will no longer be connected to any methods.



In order to connect the button to a method programmatically, you'll need a reference to the button in code. Use Interface Builder to create an `IBOutlet` for the button.

```
@IBOutlet var button: UIButton!
```

The best place to connect the button to the action is right after the view has finished loading. Add the following code to the bottom of `viewDidLoad()`:

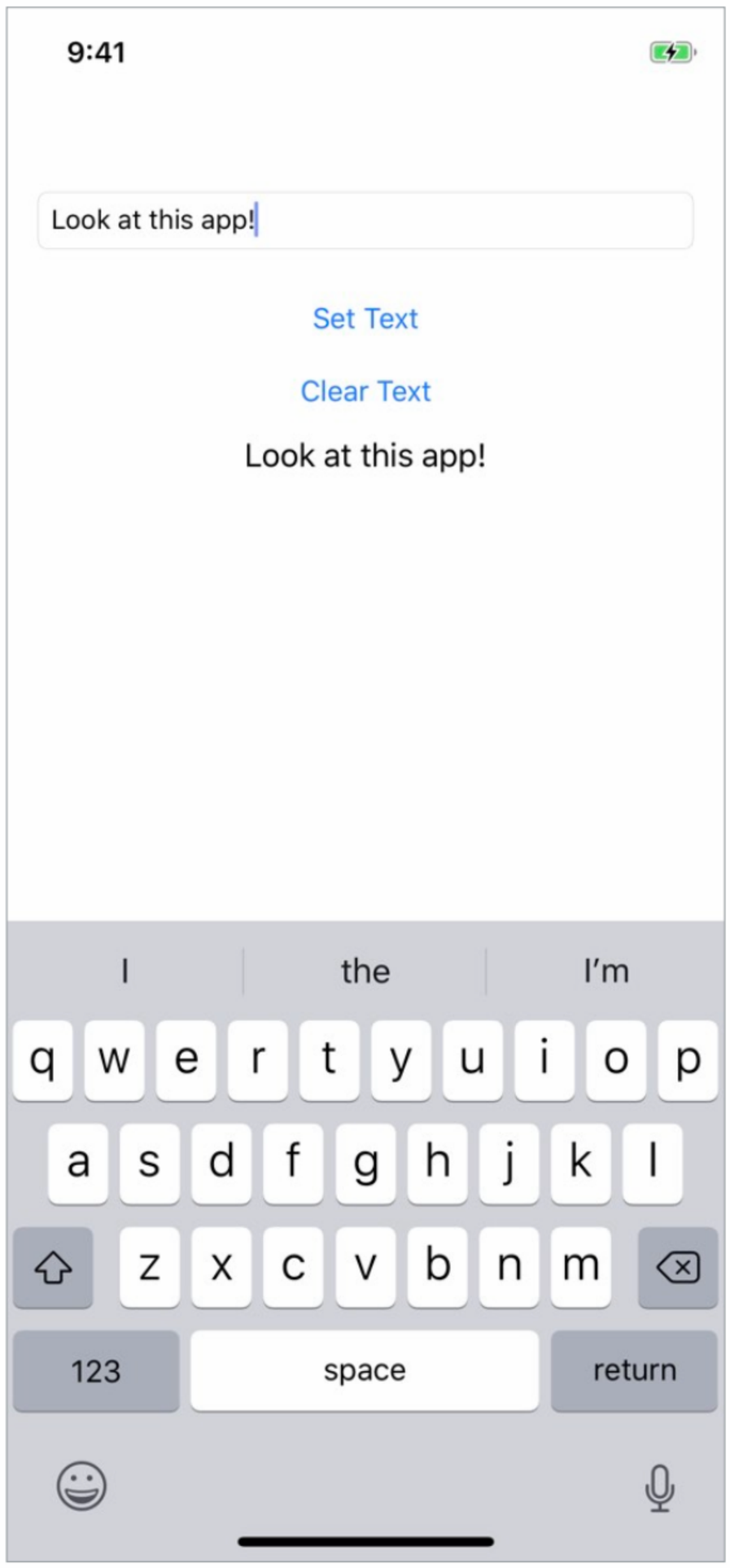
```
button.addTarget(self, action: #selector(buttonTapped(_:)),
for: .touchUpInside)
```

The `addTarget(_:action:for:)` method will connect the control to a particular action, and it requires three arguments. The first argument is the the owner of the function you want to execute. The owner of the `buttonTapped(_:)` method is the `ViewController`, or `self`. The second argument is a "selector": the name used to select a method to execute for an object. Swift uses `#selector` as its syntax to locate a particular method. The last argument is the event that should trigger the action. Just like you saw earlier in the Connections inspector, you should tie your actions to the Touch Up Inside event. (Other controls, such as switches and sliders, should utilize the Value Changed event.)

Build and run your application, and verify that when you tap the button the method still executes. Save your CommonInputControls project to your project folder.

LAB—BASIC INTERACTIONS

The objective of this lab is to create and compile an app with two buttons that, when tapped, change the contents of a label.



Create a new project called "Two Buttons" using the Single View App template.

STEP 1: CREATE YOUR VIEW IN INTERFACE BUILDER

- In the storyboard, drag a text field from the Object library and place it at the top of the view. Use the layout guides to position the label so it extends from the left margin to the right margin of the view. Set the placeholder text to "Enter text to display in the label below".

- Drag two buttons from the Object library. Place one just below the text field, and the other just below that. They should be centered in the screen. Change the title of the top button to "Set Text" and change the title of the bottom button to "Clear Text".
- Drag a label from the Object library and place it under the last button. Use the layout guides to position the label so it extends from the left margin to the right margin of the view.
- In the Attributes inspector, set the label's text alignment to centered. Double-click the label and delete the existing text. Set new text to say "Placeholder." New text will be set dynamically when the user taps one of the buttons. You'll get to that soon.

If you run the app, you should see your text field and two buttons. But if you click them, nothing will happen. That will change soon.

STEP 2: CREATE OUTLETS AND ACTIONS

- With `Main.storyboard` still selected, open the Assistant editor. The implementation file for your view controller should appear next to the canvas for the storyboard.
- Create an `IBOutlet` for the label in the `ViewController.swift` file.
- A popover will appear. Making sure the connection is set to Outlet, name the outlet `label`, and set the type to `UILabel`. Then click Connect.
- Create an `IBOutlet` for the text field in the `ViewController.swift` file. Name the outlet `textField`.
- Create an `IBAction` from the Set Text button with the name `setTextButtonTapped`, and from the Clear Text button with the name `clearTextButtonTapped`.

STEP 3: ADD CODE FOR YOUR ACTIONS

Your buttons now have actions, but they don't execute any code yet. In this step, you'll finally get your actions to do something.

- Select `ViewController.swift` in the project navigator. You'll implement the action in this file.

- Implement the `setTextButtonTapped` action to set the `label.text` to the current text in the textfield.

What did you just do? You assigned a new string to the text property of your label. The label will now display the string when the Set Text button is tapped or clicked.

- Implement the `clearTextButtonTapped` action to set the `textField.text` and the `label.text` to empty strings.

Now the Clear Text button will clear both the text label, and the text in the text field.

- Run the app, and test it by adding text in the text field and tapping the Set Text button. The text of the label should reflect the text in the text field. Now tap the Clear Text button. The text in both the text field and the label should be cleared.

Well done! You've now built an app that uses multiple controls to execute code that updates properties on another view. You're ready to move on to the next lesson. Be sure to save your work in your project folder.

REVIEW QUESTIONS

문제 1(총 6개)

Which of the following allows you to execute code for a specific control event?

- A. `IBAction`
- B. `IBOutlet`

정답 확인

Lesson 2.10

Auto Layout and Stack Views

When you build an app, you want to make sure it looks good on every iOS device. Xcode includes a powerful system called Auto Layout which makes it easy to build intricate interfaces that work on various screen sizes.

Auto Layout relies on constraints, or rules, to dynamically calculate the size and position of all views in a view hierarchy. So your interfaces will look and work the same—no matter what device your users have in their hands or how they're holding it.

In this lesson, you'll learn the fundamentals of Auto Layout for building precisely designed user interfaces.

What You'll Learn

- How to use Auto Layout to build precise views
- How to create constraints
- How to use stack views to simplify Auto Layout

Vocabulary

- Auto Layout
- constraint
- sibling
- size class
- stack view

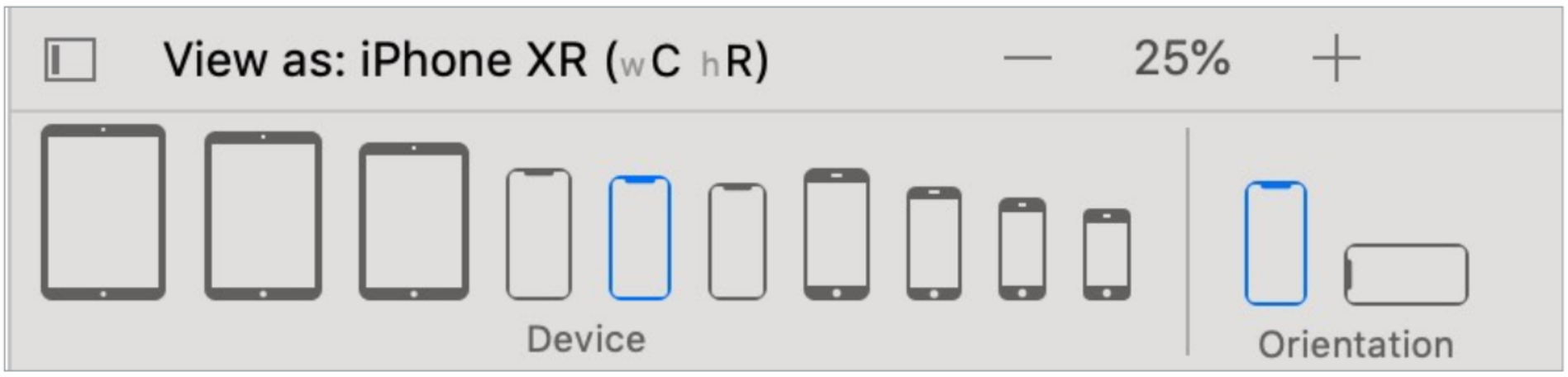
Related Resources

- Auto Layout Guide
- API Reference: UIStackView
- API Reference: UITraitCollection
- WWDC 2015 Getting Started with Multitasking on iPad

As you begin developing apps for iOS, it's important to think of the different devices that your apps will run on. You've learned that every user interface element has a size and a position on the screen. How might the size and position of those elements change in relation to the size and orientation of the device?

Create a new project called "AutoLayoutPractice" using the iOS template for a Single View App. Open `Main.storyboard` and use the layout guides to add a Button from the Object Library to the top-left corner of the screen.

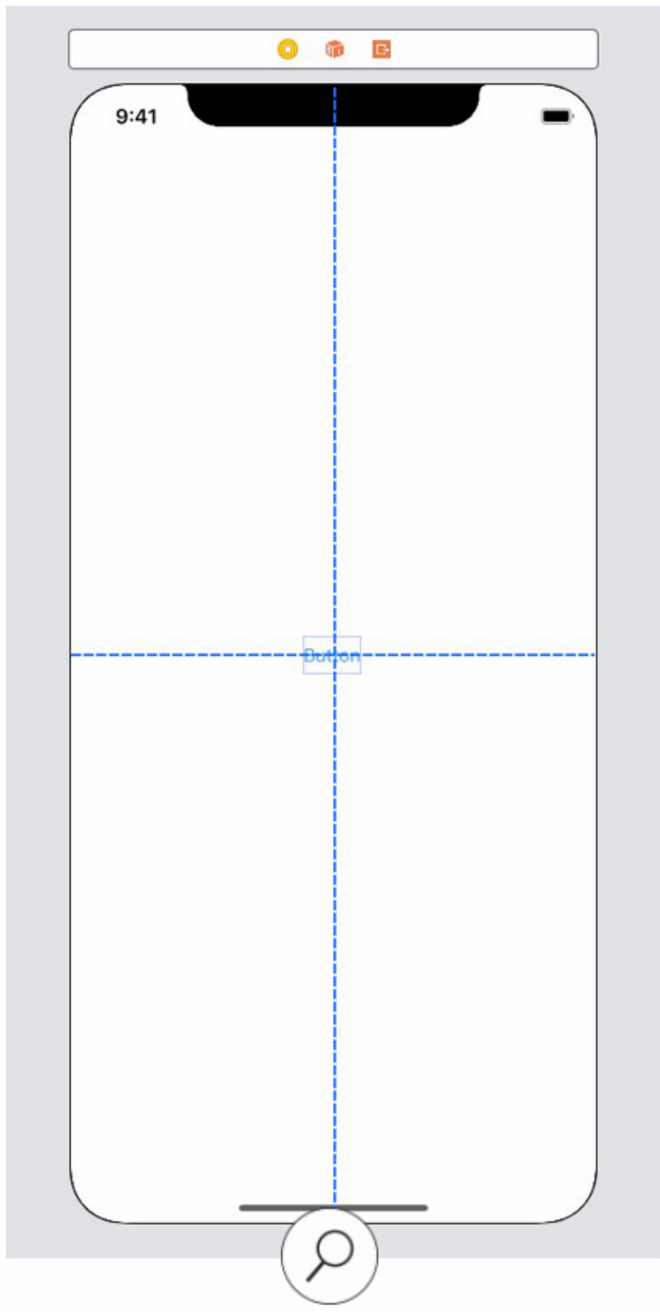
Interface Builder provides a quick way to check what your interface will look like on different screen sizes. Click the "View as" button (next to the Show/Hide Document Outline button along the bottom button bar) to reveal a menu listing multiple iOS devices and their orientations.



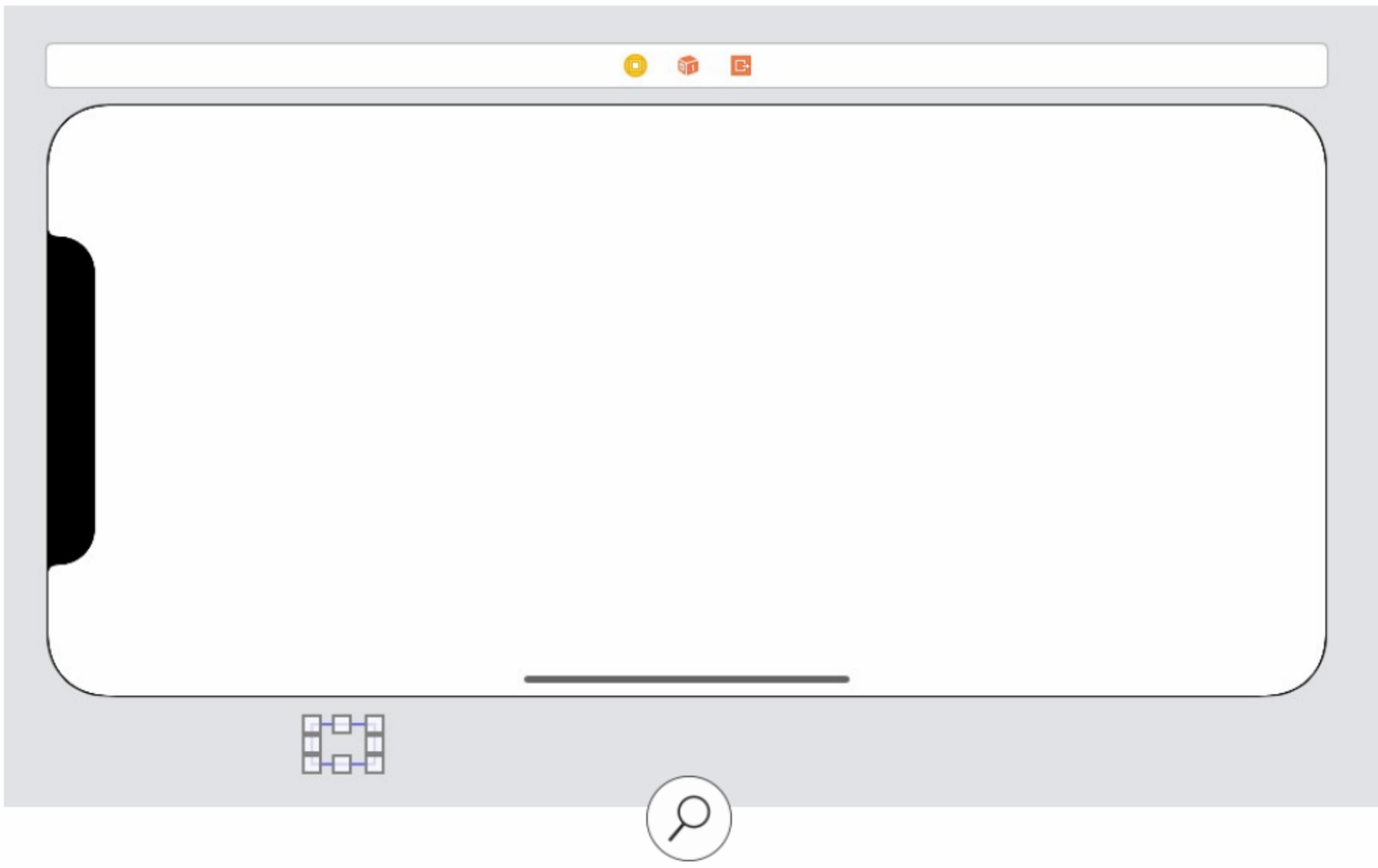
Go ahead and play around with the list. As you choose different screen sizes and orientations, the canvas will redraw your user interface accordingly. If you've positioned a button in the upper left of the canvas, it will remain in that same position—no matter which device or orientation you've selected.

WHY AUTO LAYOUT?

Now return the canvas to the iPhone XR screen size in portrait orientation, then move the button to the center of the screen. Use the blue alignment guides to ensure that the button is exactly in the center of the view.



Now switch the orientation to landscape mode. You'll see that the button is no longer centered. In fact, it isn't even on the screen!



What's going on? The button has stayed in the same X/Y position, based on the portrait orientation when you created the button. If you want the button to stay in the exact center, regardless of screen size or orientation, you'll need to create a set of constraints, or rules, that can be used to determine the size and position of your button. This system of using constraints to make adaptive interfaces is called Auto Layout.

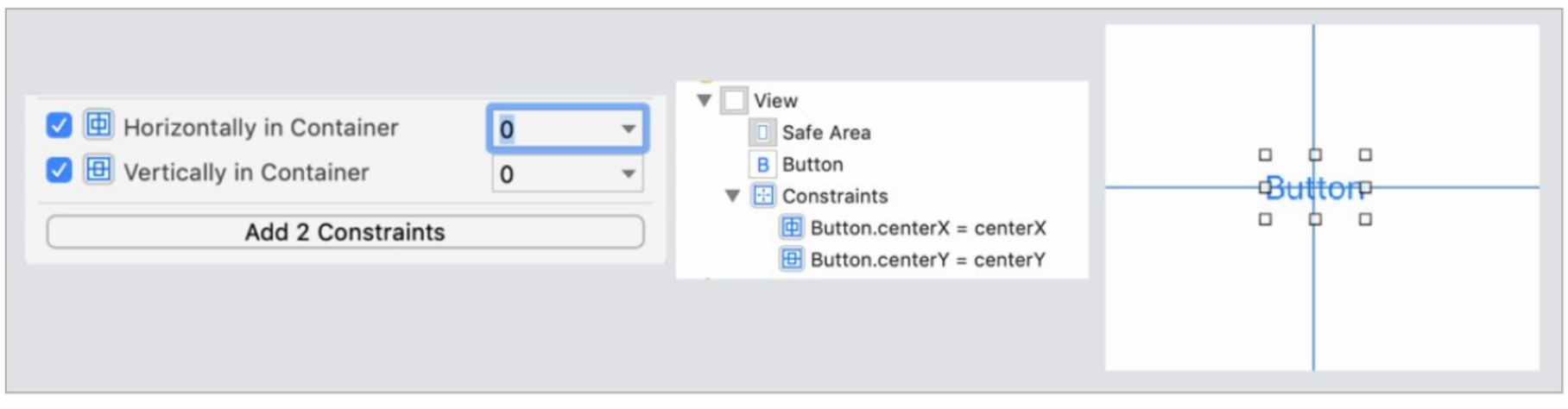
Create Alignment Constraints

At the right of the bottom button bar is a set of tools for creating and managing constraints. To lock the button in the center of the screen, you'll create two constraints that define the position of the button:

- The horizontal center of the button is equal to the horizontal center of the view.
- The vertical center of the button is equal to the vertical center of the view.

Start by returning to the portrait orientation and ensuring that the button is centered in the view. With the button selected, click the Align button , the third button in the bottom bar of constraint tools.

A popover displays a list of alignment constraints, which define the relationship between the selected object and the parent view. Select the bottom two options, "Horizontally in Container" and "Vertically in Container," then click "Add 2 Constraints." Once you've created these constraints, you should see crossing blue lines over the top of your button, indicating horizontal and vertical alignment with the parent view.

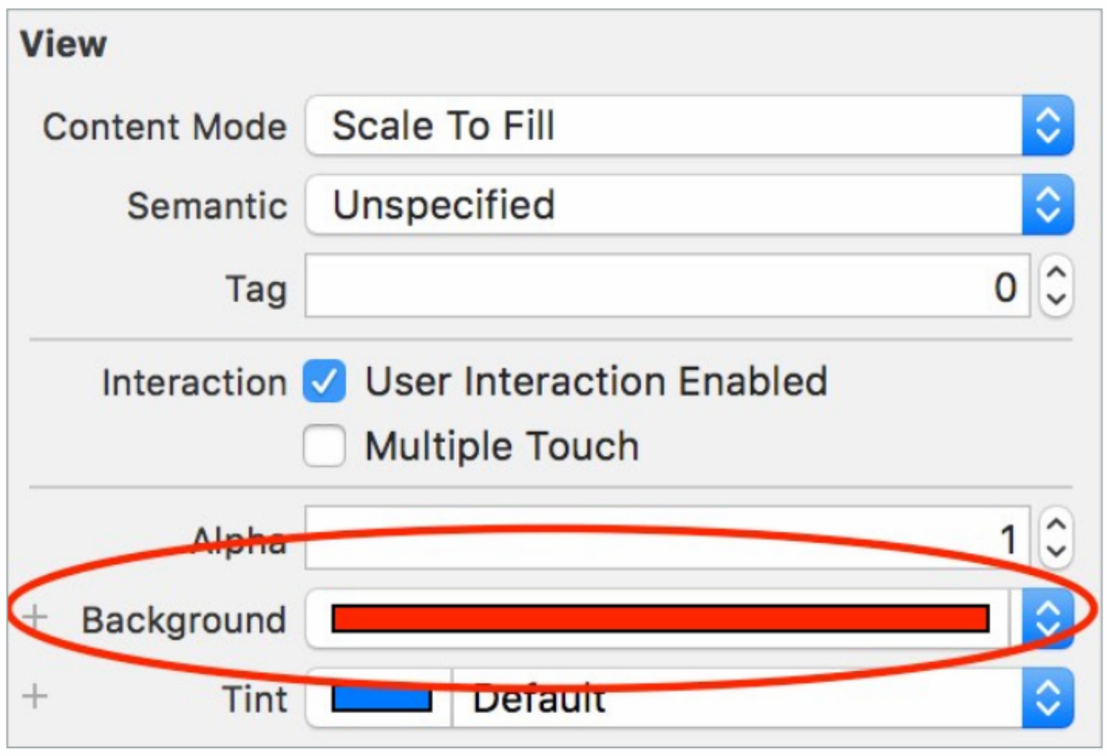


Use the "View as" button to select different devices and orientations. You'll see that the alignment constraints you added are keeping the button centered in all views.

Create Size Constraints

What about the width and height of the button? Since there aren't currently any size constraints on the button, its size is dictated by the button text and font. But Interface Builder provides multiple ways to use constraints to set the width and height of interface elements.

Start by using the Attributes inspector to change the background color of the button to something other than clear. This will make the button's size easier to see.



Now assume you want the height of the button to always be 60, no matter the screen size or orientation of the device. Click the Add New Constraints button , the button immediately to the right of the Align tool.

The popover displays a list of text fields and checkboxes to help you add constraints. Select the Height checkbox and adjust the value to 60. Click "Add 1 Constraint" to create the height constraint.

Note again that if you use the "View as" button to select different devices and orientations, you'll see that the size constraints added are keeping the button size identical in all views.

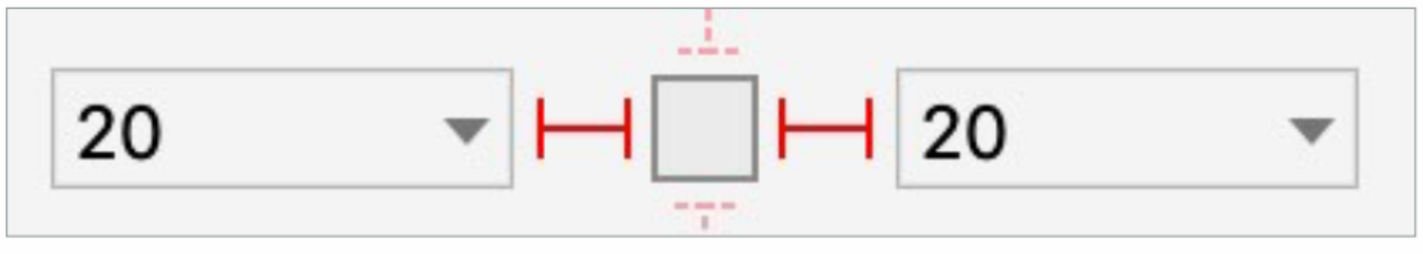
Constraints Relative to the Screen

You've fixed the height of the button, but what if you want the button's width to vary based on the screen size? Assume you want the button's left and right edges to always be 20 pixels from the left and right edges of the view, respectively.

Click the Add New Constraints button again, and notice the four fields at the top of the popover. These values define the distances from the top, leading, bottom, and trailing edges of the button to the nearest view. In this case, since there are no other views on the screen, they're dictating the distances from the button edges to the view's edges.

Leading refers to the left edge of the screen, while trailing refers to the right edge. These are labeled "leading" and "trailing" instead of "left" and "right" since not all languages read in the same direction. For certain users, you'll want your app flipped. Leading and trailing constraints make this process easier.

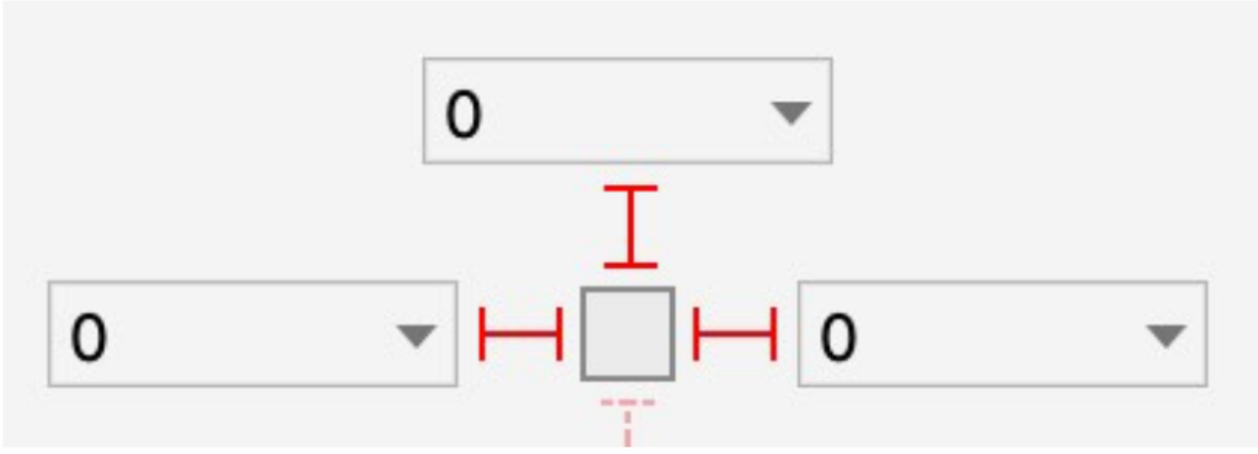
Adjust the left and right-edge values to 20. The red indicators will illuminate to show which edges are being constrained. Click "Add 2 Constraints."



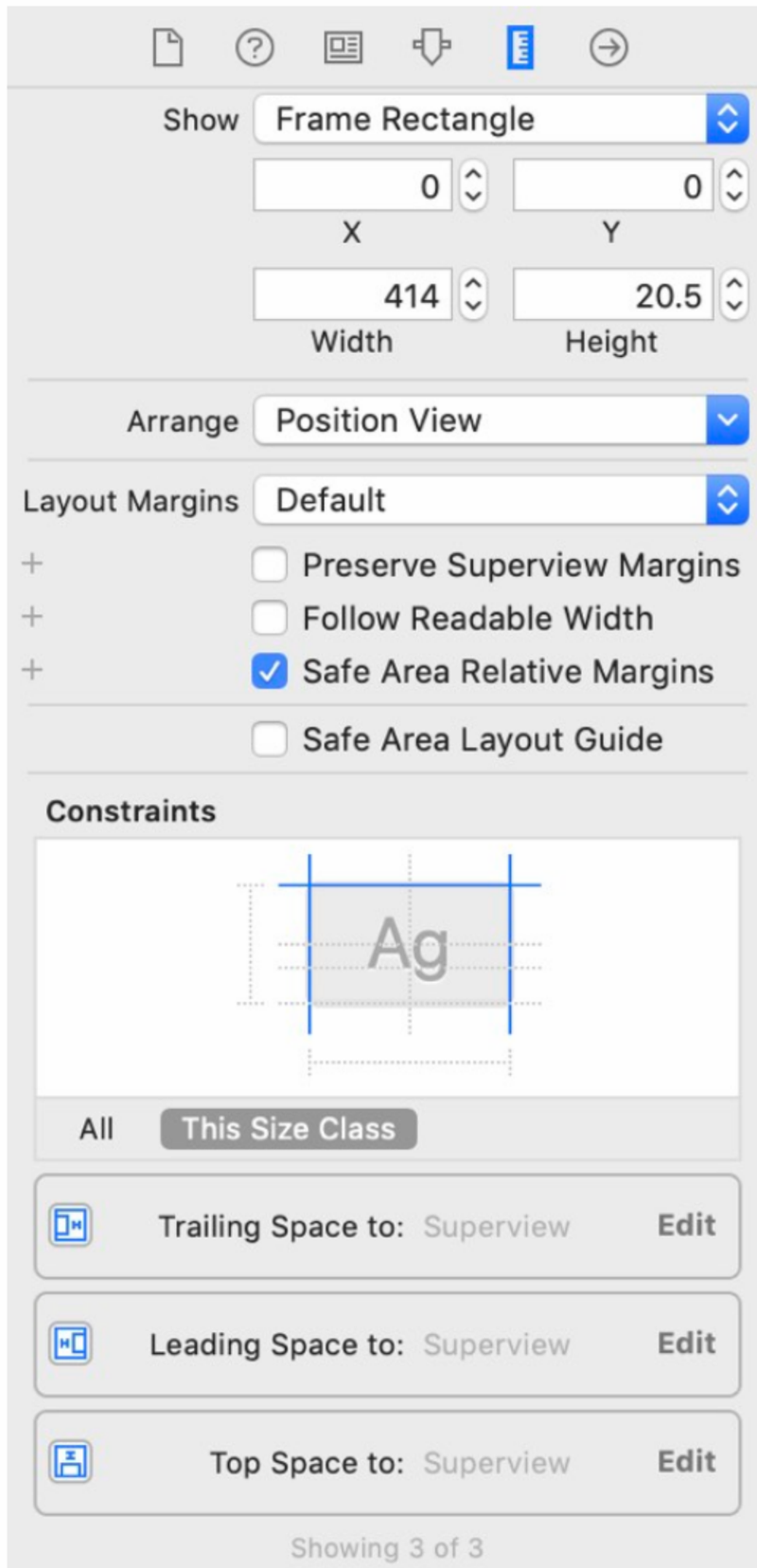
Notice that the button has now expanded to be 20 pixels from each edge of the screen. Use the "View as" button to select different devices and orientations. You'll see that the width of the button varies such that it always maintains the same distance from the edges of the screen. If for some reason the button didn't update, update the button's frame using the Update Frames tool

Safe Area Layout Guide

Remove the button from the screen by selecting it and pressing the Delete key. Now add a label from the Object Library to the very top left corner of the screen under the status bar. Using the Add New Constraints tool, create constraints with values of 0 on the top, left, and right of the label to the view.

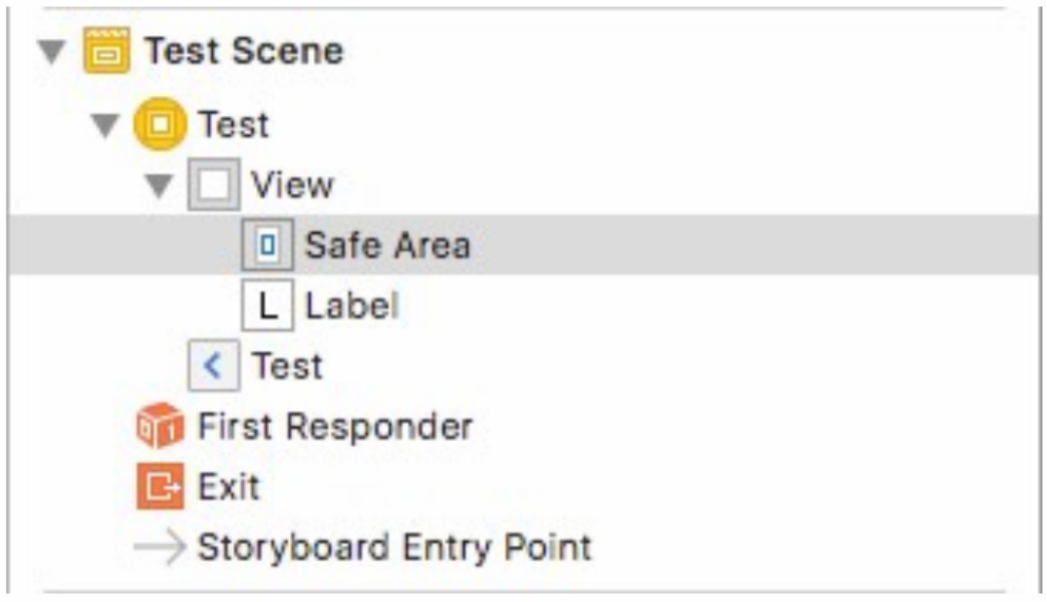


Now select the label and open the Size inspector. You can view the three constraints that you just added.



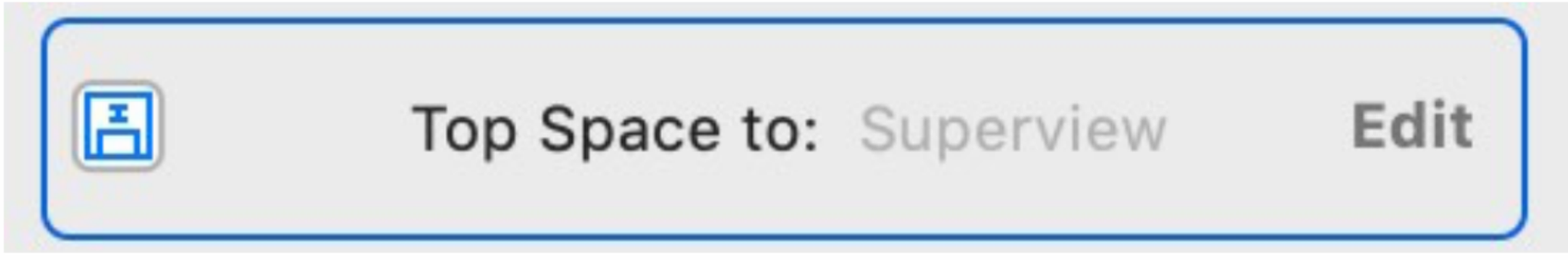
Notice that the constraints are between the label (which you have selected) and the Superview. This means the edges of the label are constrained to the edges of the main view controller's view. You can see in Interface Builder that this places the label under the status bar and obscures some of the label's text. You may be tempted to simply change the value of the top constraint so that the distance from the top is equal to the height of the status bar. However, what if later that view controller is embedded in a navigation controller? The label would no longer be covered by the status bar, but it would then be covered by a navigation bar.

To fix this, you'll need to create constraints relative to the Safe Area of your view controllers. The Safe Area is displayed in Interface Builder as a subview of the view controller's primary view.



If you select the Safe Area, you'll notice that the entire view controller is selected except for portions of the screen that are taken up by system views like the status bar. This is because Interface Builder anticipates the existence of these common system views. However, there are other system bars that can turn up at run-time, and the Safe Area will adjust for those as needed. This allows your content to adapt to system overlays such that it won't be hidden.

Delete your current constraints by selecting them one at a time in the Size inspector and pressing the Delete key.

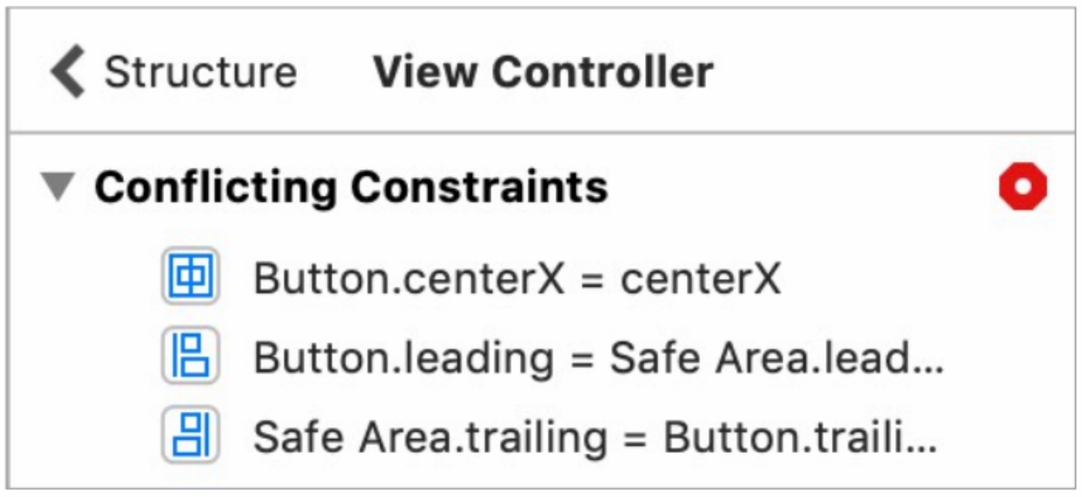


Drag your label a bit lower on the screen so it is below the status bar, and create a constraint between the label's top edge and the top edge of the Safe Area using the Add New Constraints tool. Also add constraints for the leading and trailing edges of the label. Since the Add New Constraints tool constrains your view to the nearest view, creating these constraints with the tool will constrain the top, leading, and trailing edges of the label relative to the Safe Area.

Resolve Constraint Issues

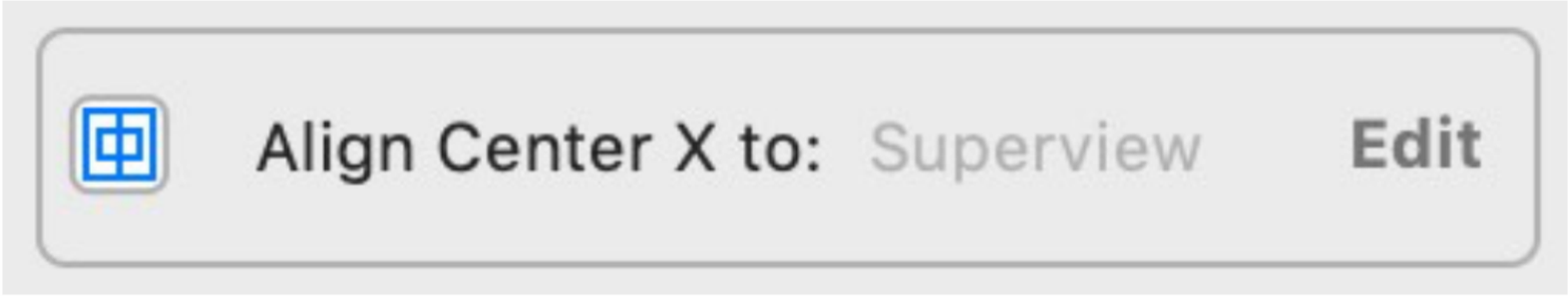
Now delete the label that you added. Add a button to the center of the view controller and give it two alignment constraints - one to center it vertically on the screen, and one to center it horizontally on the screen. Now use the Add New Constraints tool to constrain its left edge 20 pixels from the left edge of the screen, and to constrain its right edge 30 pixels from the right edge of the screen.

You'll notice there's a problem, indicated by the red lines on the canvas. You've now created two conflicting constraints, each of which is trying to dictate the X position of the button. To see a list of constraint conflicts, click the little red arrow in the upper-right corner of the document outline.



One of the constraints says, "Set the horizontal center of the button equal to the horizontal center of the view"; and the other says, "The left edge of the button is 20 pixels away from the left edge of the view." Since these two constraints can't coexist, you'll need to remove one.

To remove the constraint that centers the button, select the button and open the Size inspector . Choose "Align Center X to: Superview" from the Constraints list, and press the Delete key.



From time to time, you may run into a constraint issue that's tricky to resolve. Or maybe you're adding interface elements to the scene and you need to reconfigure the view's layout. In either situation, you can click the Resolve Auto Layout Issues button , next to the Add New Constraints tool, and use the options to try to automatically address constraint issues.

Update Constraint Constants will attempt to update the rules to match what is currently displayed in the storyboard scene.

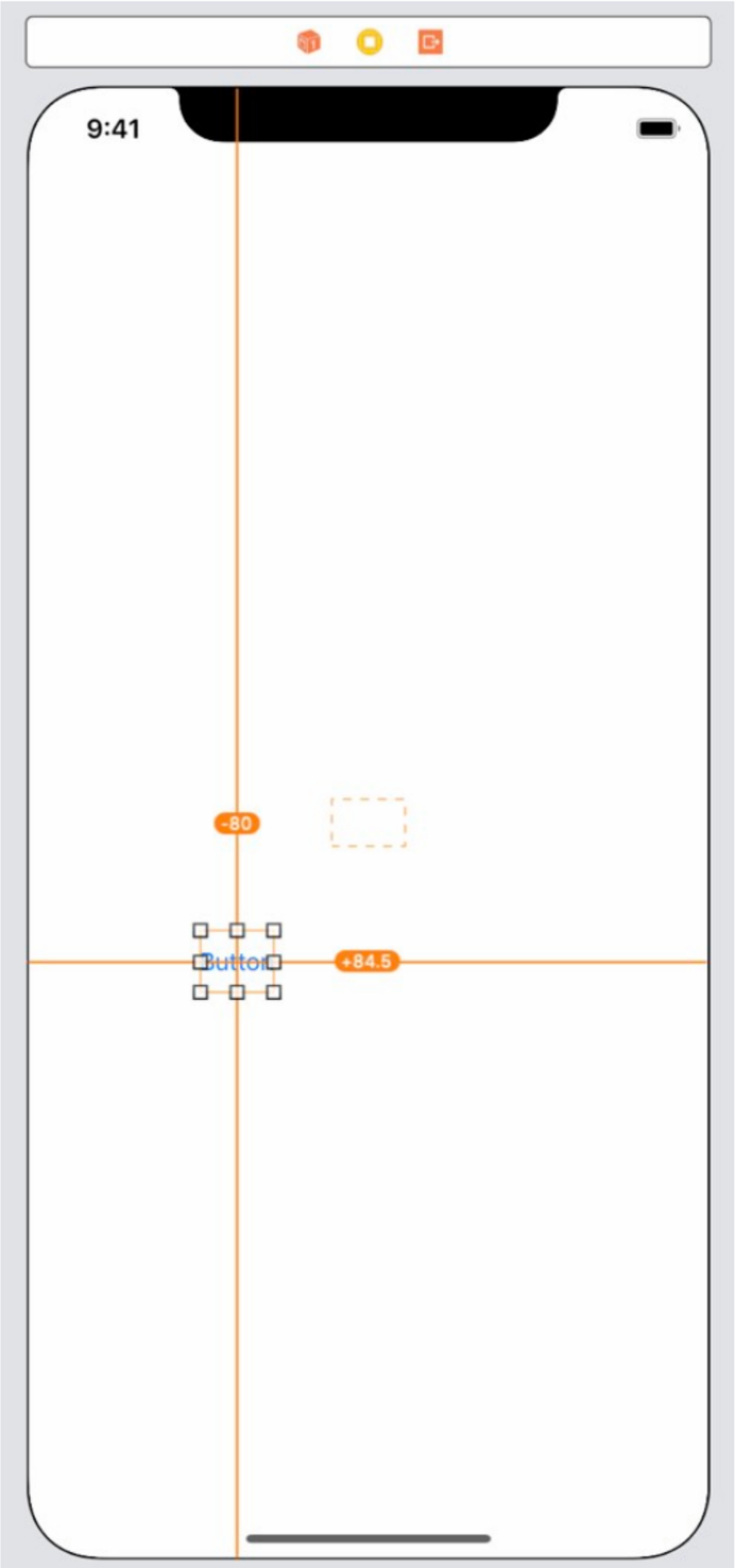
Add Missing Constraints will attempt to add new constraints that match what is currently displayed in the storyboard scene.

Reset to Suggested Constraints will clear all constraints and attempt to accurately assign new constraints that match what is currently displayed in the storyboard scene.

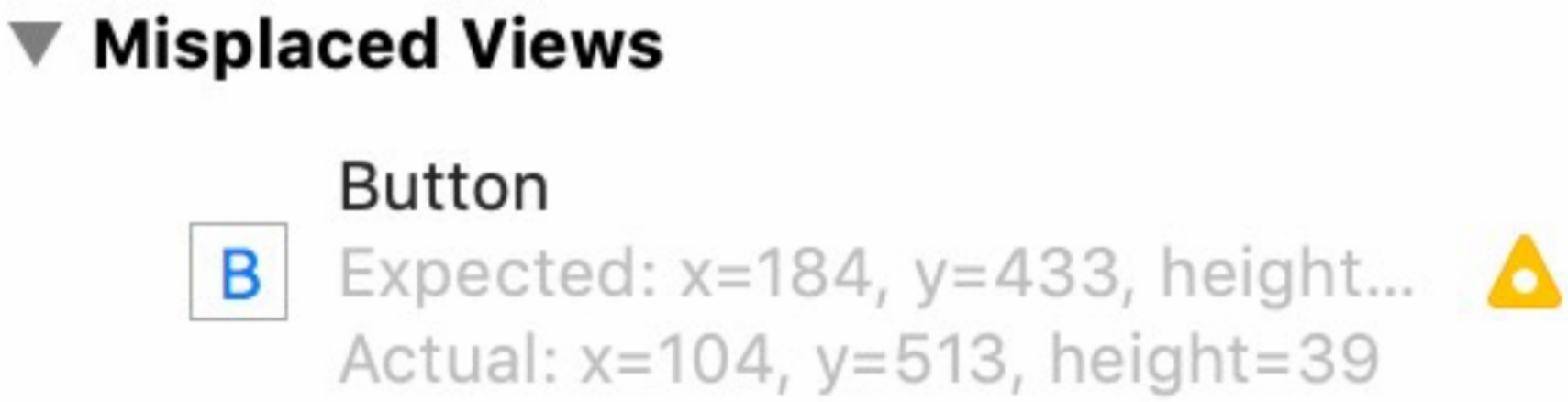
Clear Constraints will remove any rules associated with the position and size of the selected view or all views.

Resolve Constraint Warnings

Delete the button you've been working with. Add a new button to the scene, and create the original two constraints that position the button in the center of the view, both vertically and horizontally. Next select the button and drag it down and to the left. You should see an Auto Layout warning.



To understand what's happening, click the yellow arrow in the upper-right corner of the document outline. The warning informs you that the position of the button is no longer in sync with the position defined by the constraints.

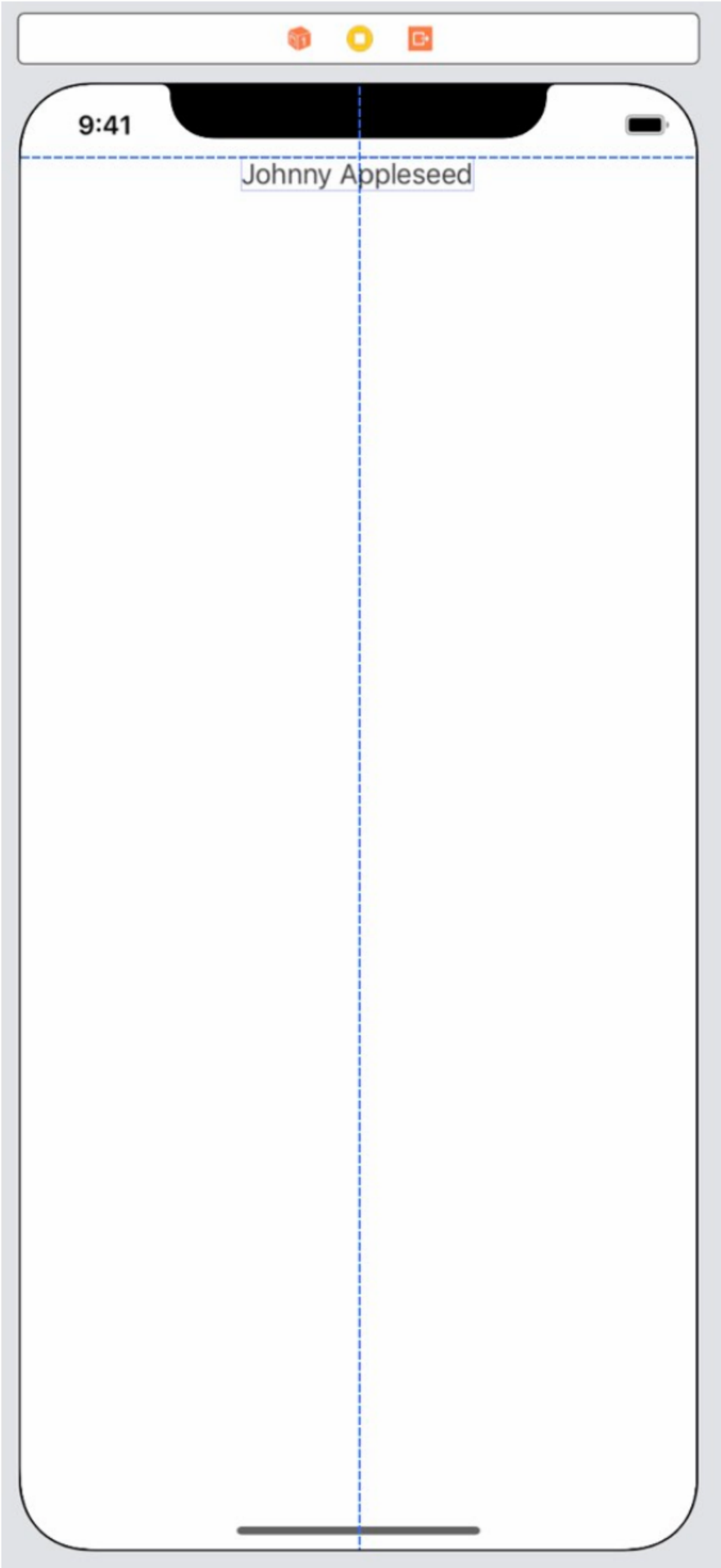


To adjust the button's position to fit the constraints, select the button on the canvas, then click the Update Frames button .

Constraints Between Siblings

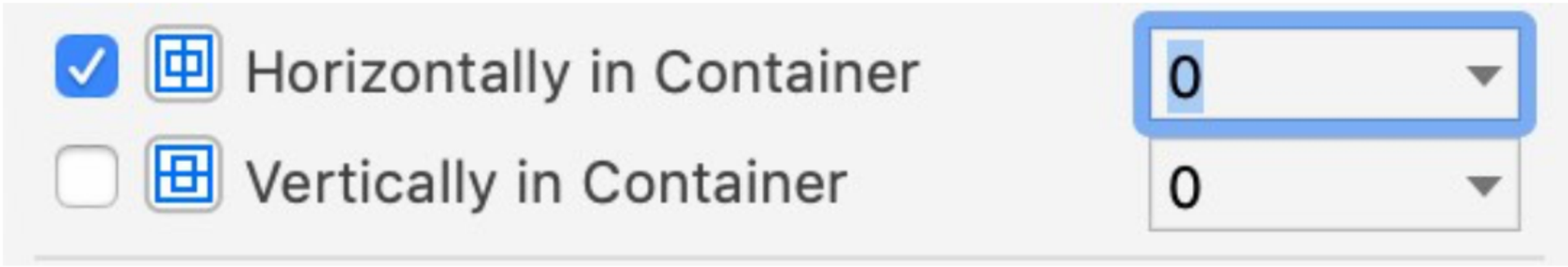
So far in this lesson, you've learned how to create constraints that define relationships between a view and its parent view. But there may also be times you want to create constraints between sibling views. In the following exercise, you'll work with multiple labels that display text information about yourself.

Once again, delete the button you've been working with, and add a label to the scene. Double-click the label and type in your name. Drag the label near the top of the view, using the guides to center it horizontally.



With the label selected, use the Add New Constraints tool to constrain it to be 0 pixels from the top of the Safe Area. To enable the constraint, you may need to select the red indicator line below the top field in the popover.

Use the Align tool to align the horizontal center of the label with the center of the parent view.



With one label set, you can add a second label and base its position on the position of its sibling. Drag another label from the Object library onto the view, just below the existing label. Enter some information about yourself, perhaps one of your favorite hobbies, into the new label's text.

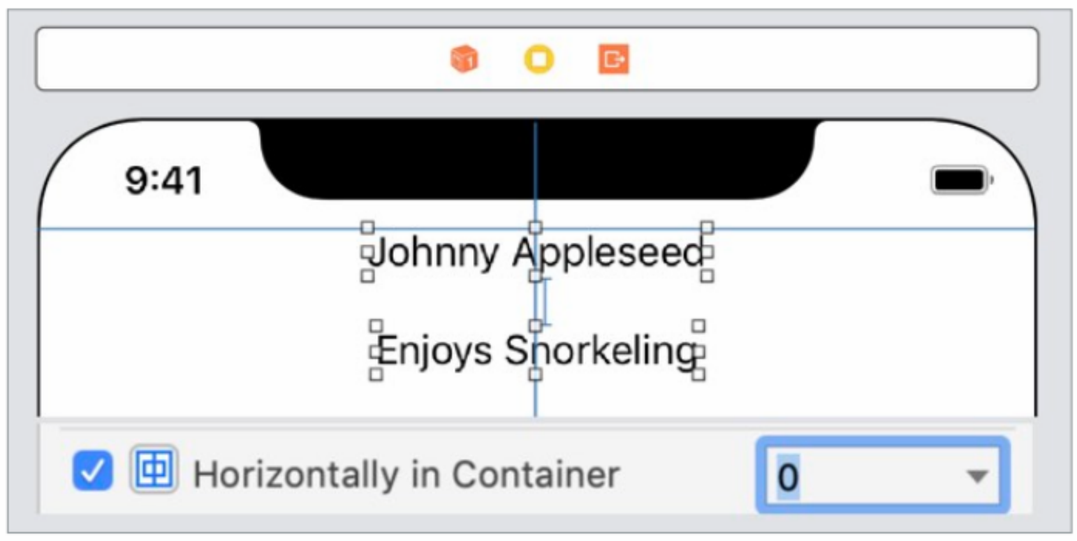
Now assume you want the new label to have the same horizontal center as the first label and to be positioned 20 pixels below it. With the new label selected, click the Add New Constraints button. In the popover, enter 20 into the top text field, and click "Add 1 Constraint."

At this point, Interface Builder will display an error, indicating that the new label has a constraint describing its vertical position, but no information about its horizontal position. Interface Builder can help you resolve this error.

Select both labels in the scene by Command-clicking each of them, and click the Align button. In the popover, select the Horizontal Centers checkbox, set its value to 0, and click "Add 1 Constraint." This tells Interface Builder that you want the selected views to have the same center position, with 0 pixels of offset.

If Interface Builder still displays a warning, click the Update Frames button to update the labels' frames to match the constraints you just specified.

오후 10:38 7월 18일 토요일 61%



To practice what you just learned, add three new labels and repeat the above steps three more times to position them on the scene. When you're done, each label should be 20 pixels below the previous one, and they all should be centered horizontally.

Stack Views

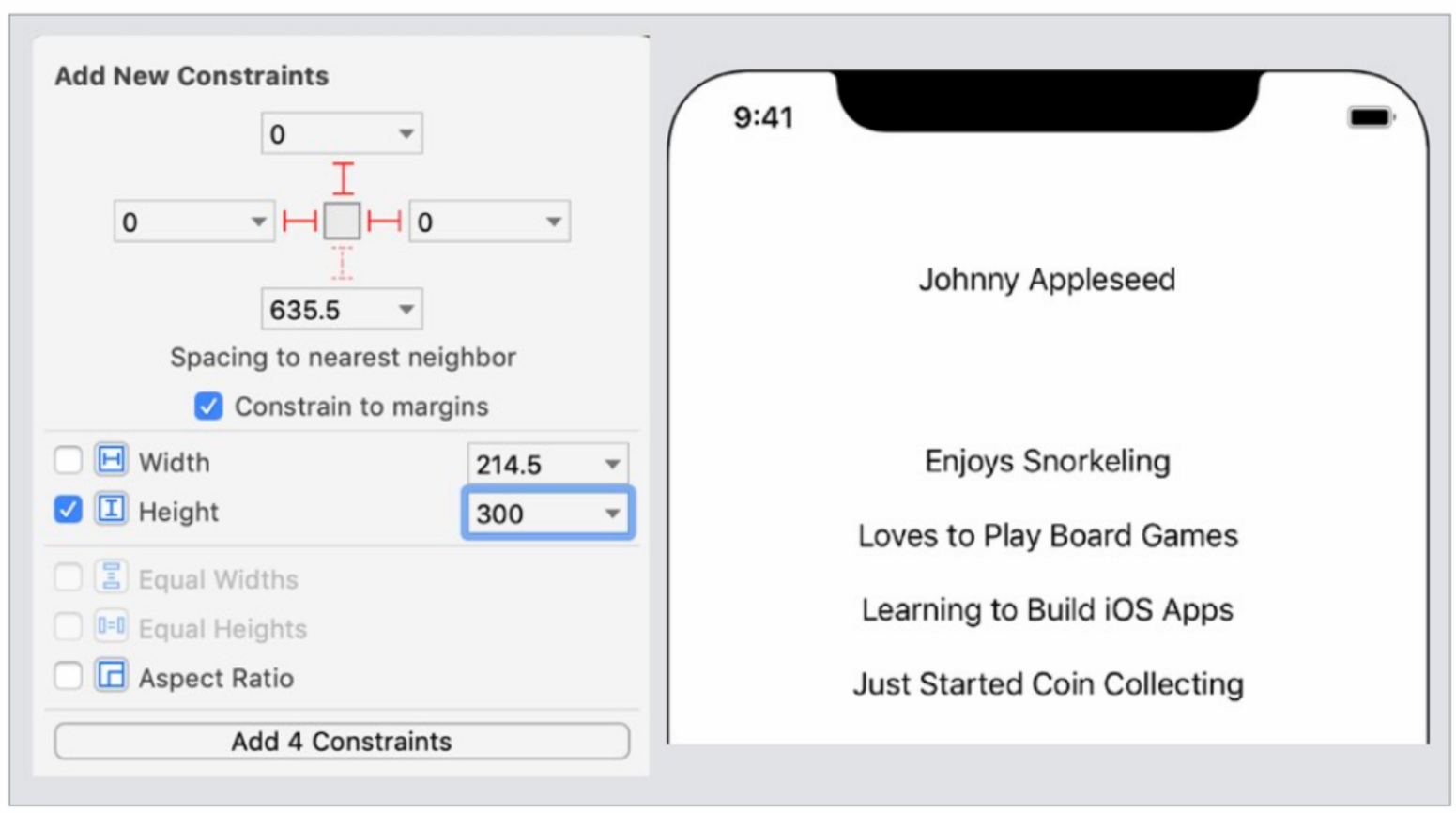
Did you find that last exercise a bit repetitive? Managing the constraints of multiple objects can be tedious, especially when the position of each element is the same distance from the previous one. What if you decide later to have 30 pixels of separation between labels instead of 20? Or what if you want all of the labels to be positioned on the left side of the parent instead of in the center? You would have to spend a lot of time updating each and every constraint.

Interface Builder provides a smarter approach. Rather than creating and updating lots of individual constraints, you can use a stack view to automatically manage the constraints of its child views.

A single stack view manages either a row or a column of interface elements, and arranges those elements based on the properties set on the stack view. Start by selecting all the labels you just created, then click the Embed In Stack button to the left of the Align tool and choose Stack View from the list that appears. This will

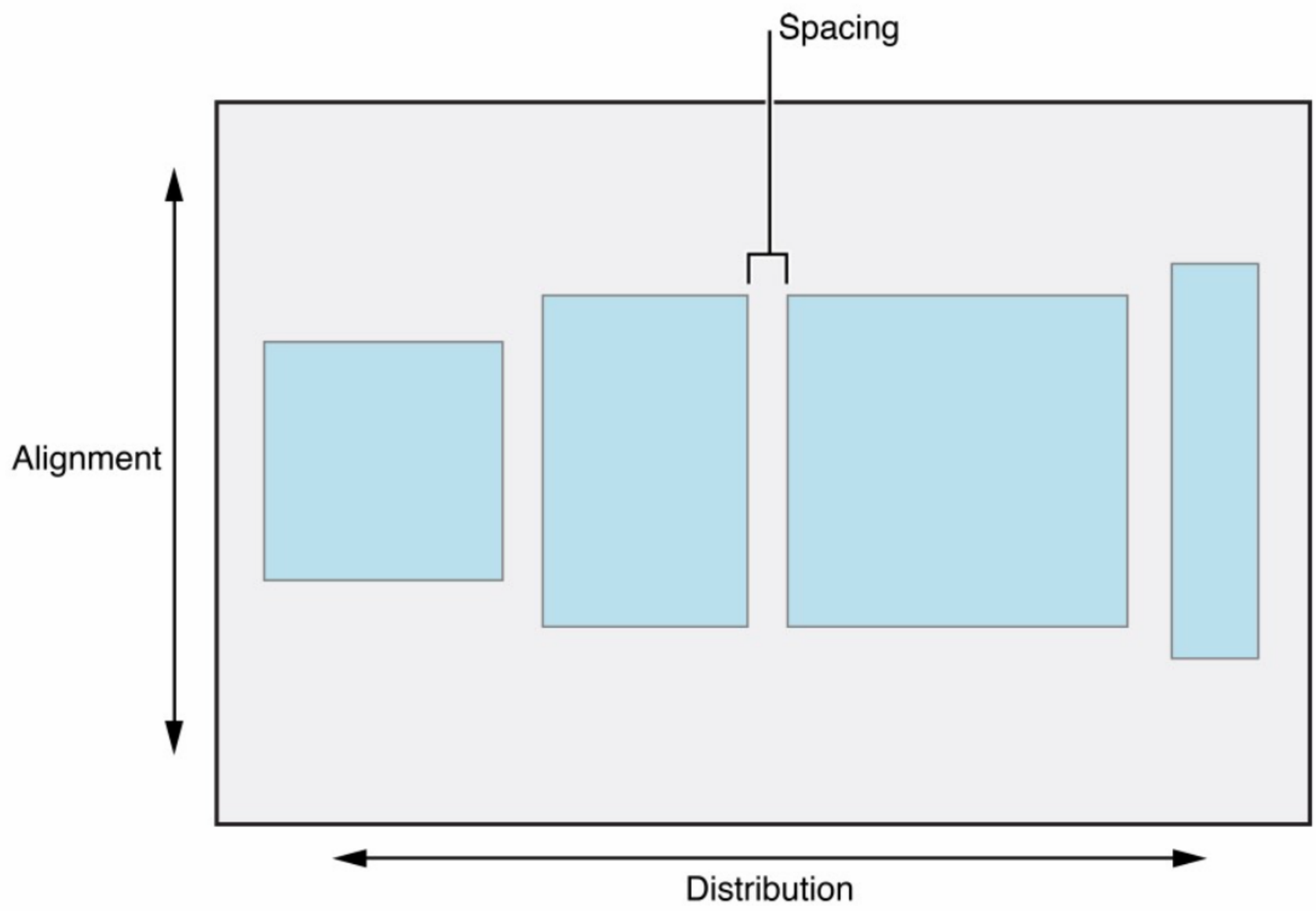
organize the selected views into a single stack and will remove any constraints on the individual labels.

At the moment, your new stack view doesn't have any constraints that define its size or position. To change that, select the stack view, then click the Add New Constraints button. Set the top, leading, and trailing edges of the stack view to 0 pixels from the parent view's edges. (You'll need to select the red indicators associated with each text field to enable the constraints.) Give the stack view a constraint height of 300. Click "Add 4 Constraints," and click the Update Frames button, if necessary.

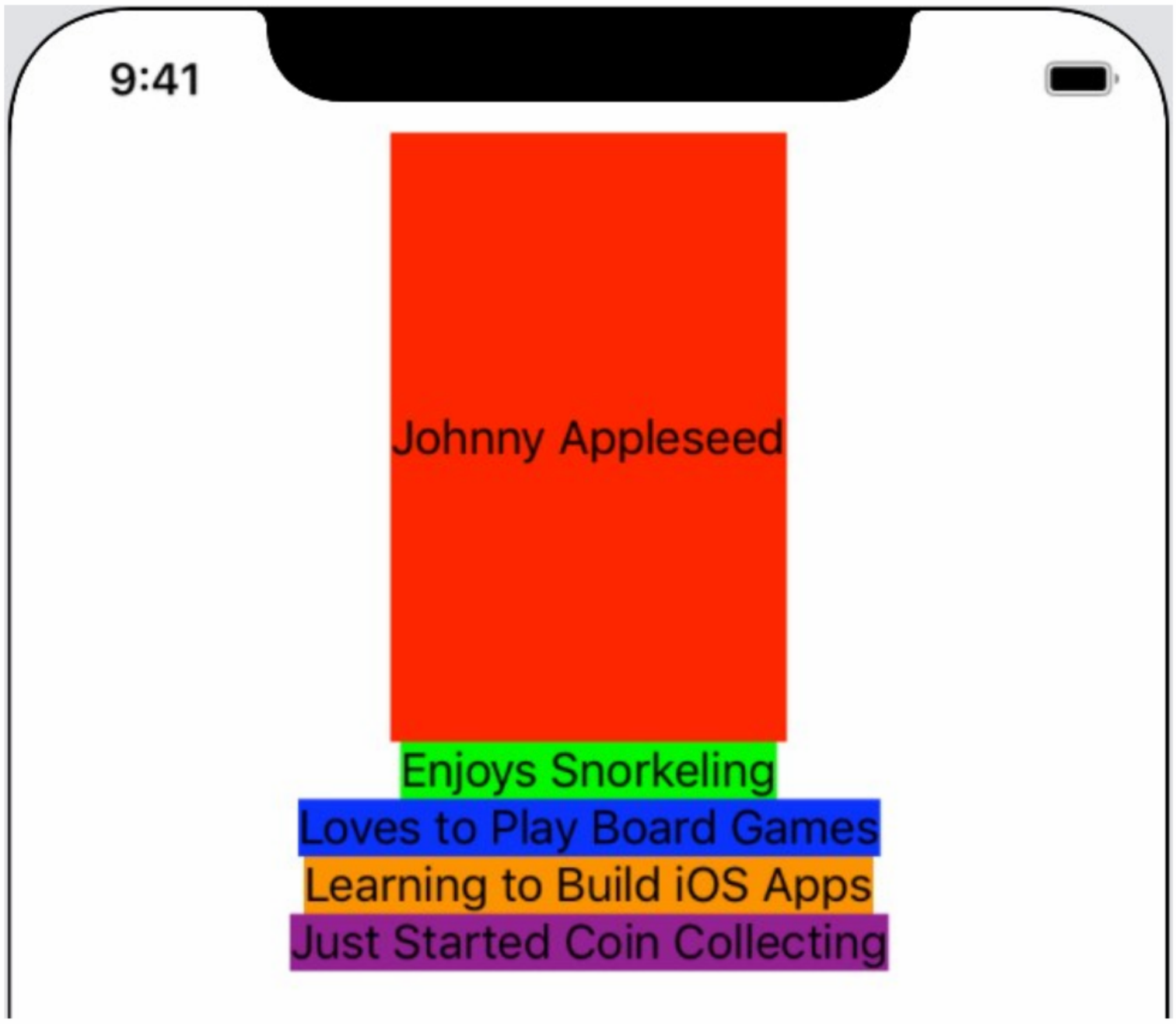


오후 10:38 7월 18일 토요일 61%

Stack View Attributes



Now that you have all your labels in a stack, you can learn how to manage the stack view and define the positions of its subviews. But before you begin, set the background of each label to a unique color. This will make it clear how your adjustments are affecting the interface.

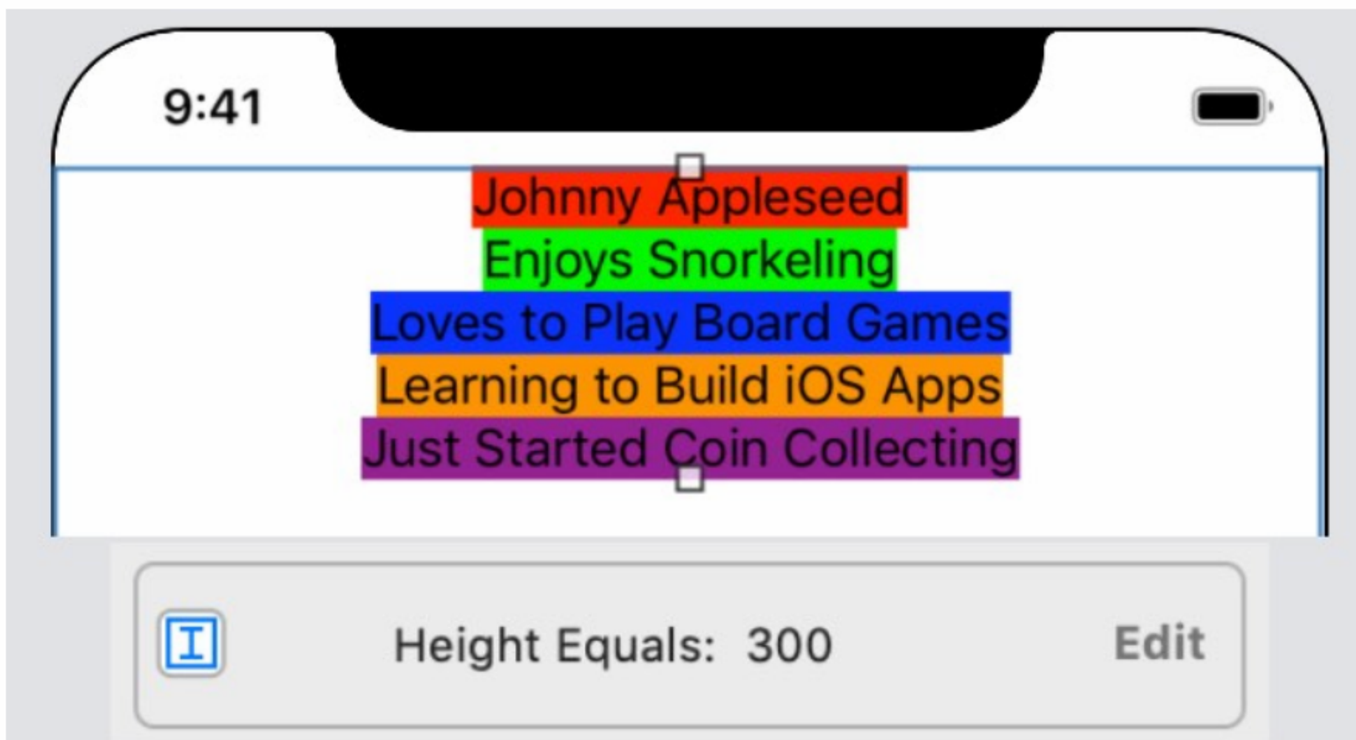


Select your stack view from the document outline, and open the Attributes inspector to reveal four main properties. Go ahead and explore these attributes, making adjustments along the way.

1. Axis determines whether the elements within the view are stacked vertically or horizontally.
2. Alignment describes how the elements within the stack are positioned. You can choose one of the following:
   - Fill—Each of the elements fills the size of the stack. For example, if the stack view has a width of 320 pixels, each element in a vertical stack will also have a width of 320 pixels.
   - Leading—The leading edge of each element is aligned to the leading edge of the stack view.
   - Center—The center of each element is aligned to the center of the stack view.
   - Trailing—The trailing edge of each element is aligned to the trailing edge of the stack view.

3. Distribution defines how the elements are distributed within the stack view. You can choose one of the following:
   - Fill—The stack view resizes the arranged subviews so that they fill all of the available space along the axis you specified. The stack view will size the first label as large as possible to completely fill the stack.
   - Fill Equally—The stack view resizes the arranged subviews so that they fill all of the available space along the axis. The views are resized so that they are all the same size along the stack view's axis.
   - Fill Proportionally—The stack view resizes the arranged subviews so that they fill all of the available space along the axis. If the size of the stack view changes, the views are resized proportionally to one another. For example, if two labels have heights of 50 and 100, respectively, and you increase the stack view's height by 50 percent, the labels would then have heights of 75 and 150.
   - Equal Spacing—The stack view doesn't resize the subviews but positions them at an equal distance from one another.
   - Equal Centering—The stack view doesn't resize the subviews, but ensures that the center of each subview is an equal distance to the centers of the other subviews.
4. Spacing describes the amount of space between each element in the stack view. For example, you can change the spacing from 0 to 20 to add 20 points of spacing between labels.

Use the Size inspector to remove the height constraint on the stack view. If a stack view doesn't have a specified height, it will resize based on its subviews.



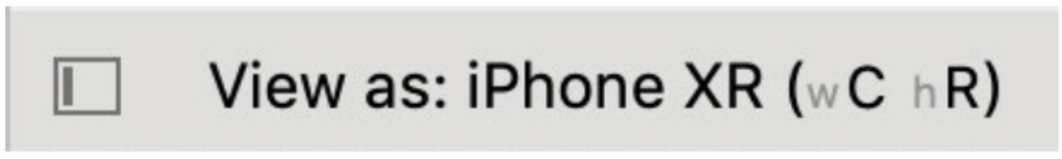
Now that you've defined some attributes, try adding new labels to the stack. Or rearrange the existing labels and adjust the stack's spacing and alignment. Imagine if you hadn't used a stack view and you wanted to adjust the interface. How many constraints would you have needed to add, remove, or adjust?

Whenever possible, it's a good practice to use stack views before trying to manage constraints individually. Stack views allow you to create nice-looking interfaces quickly —and also make it easy to modify or customize them in the future.

SIZE CLASSES

With so many different combinations of screen sizes and orientations, it's important to build an interface that works well on all iOS devices. In the case of iPads, the split-screen feature adds an additional layer of complexity, because an app may run in a smaller amount of space than you originally expected. To help simplify user interface development for so many different situations, iOS includes a concept called size classes.

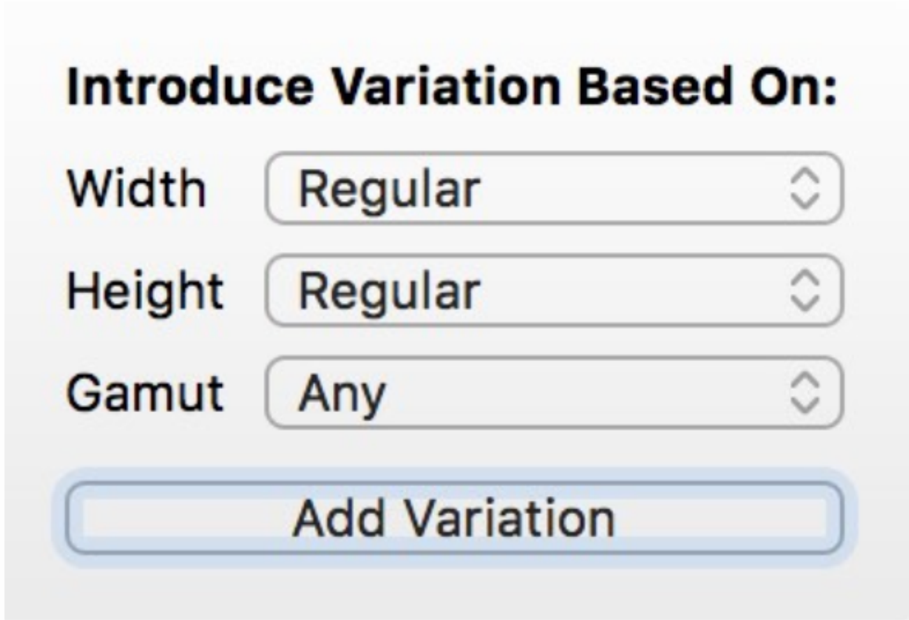
Each iOS device has a default set of size classes that you can use as a guide when designing your interface. Each dimension, the width and the height, is either the Compact or Regular size class, and the two dimensions form a trait collection. The "View as" button indicates the trait collection of the current device and orientation you have selected. For example, if you use the "View as" button to select an iPhone XR in Portrait mode, you will see "(wC hR)" as part of the button title. This indicates that the trait collection for this device is Compact Width, Regular Height.



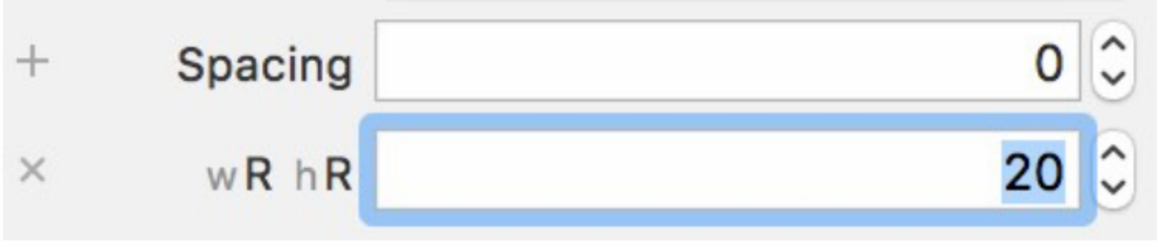
Though an iPad has a Regular Width, Regular Height size class in both portrait and landscape, an app does not always run full-screen. Using the split-screen feature, an app can consume one third, one half, or two thirds of the screen real estate, and different-sized iPads will have unique trait collections when multi-tasking. It sounds like a lot to handle, but don't think about the wide array of devices you need to support. Instead, focus on supporting the four different width/height trait collections, and your interface will be correct.

Vary Traits

Constraints and properties can be set to different values depending on the trait collection. Suppose you want a stack view's Spacing property to be 0 most of the time, but when run on a Regular Width, Regular Height device, it should be set to 20. Interface Builder makes it easy to add varying traits for a particular property or constraint. Using the stack view that you previously created with colored labels, select the stack and open the Attributes inspector. Next to Spacing, click the '+' button, which will open a popover for introducing variations.



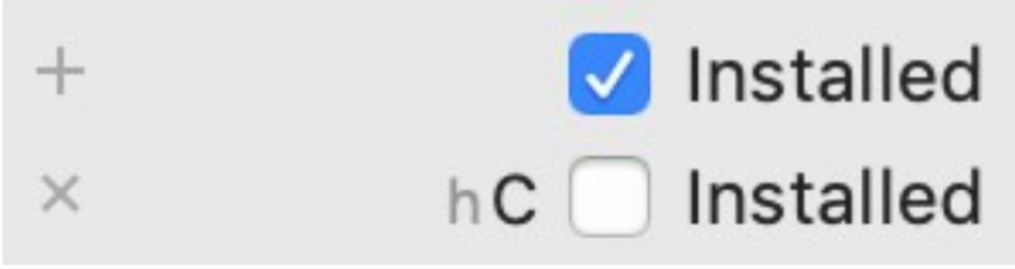
Since you want to add a new variation for the Regular Width, Regular Height trait collection, set the appropriate boxes to these values, then select "Add Variation". This will add a new Spacing property that is unique for this particular trait collection. Change this new Spacing to 20.



To verify that this trait variation is working properly, click the "View as" button and change the device to an iPad (which is Regular Width, Regular Height). You should see additional spacing between each label. When you return to an iPhone XR, the label spacing diminishes.

Installed

Sometimes you'll want to disable particular views or constraints depending on the trait collection. For example, maybe in order to save space, your app's interface should not include the red label when the trait collection's height is set to Compact. To begin implementing this design decision, select the red label and open the Attributes inspector. Near the bottom, you'll find an "Installed" checkbox. This determines whether the particular view or constraint exists within the view hierarchy. Click the '+' button next to the checkbox, and you'll be presented with the same popover to introduce a variation. Set the Width to Any and the Height to Compact, then click "Add Variation". This will add a new Installed property that spans any trait collection that includes a Compact Height.



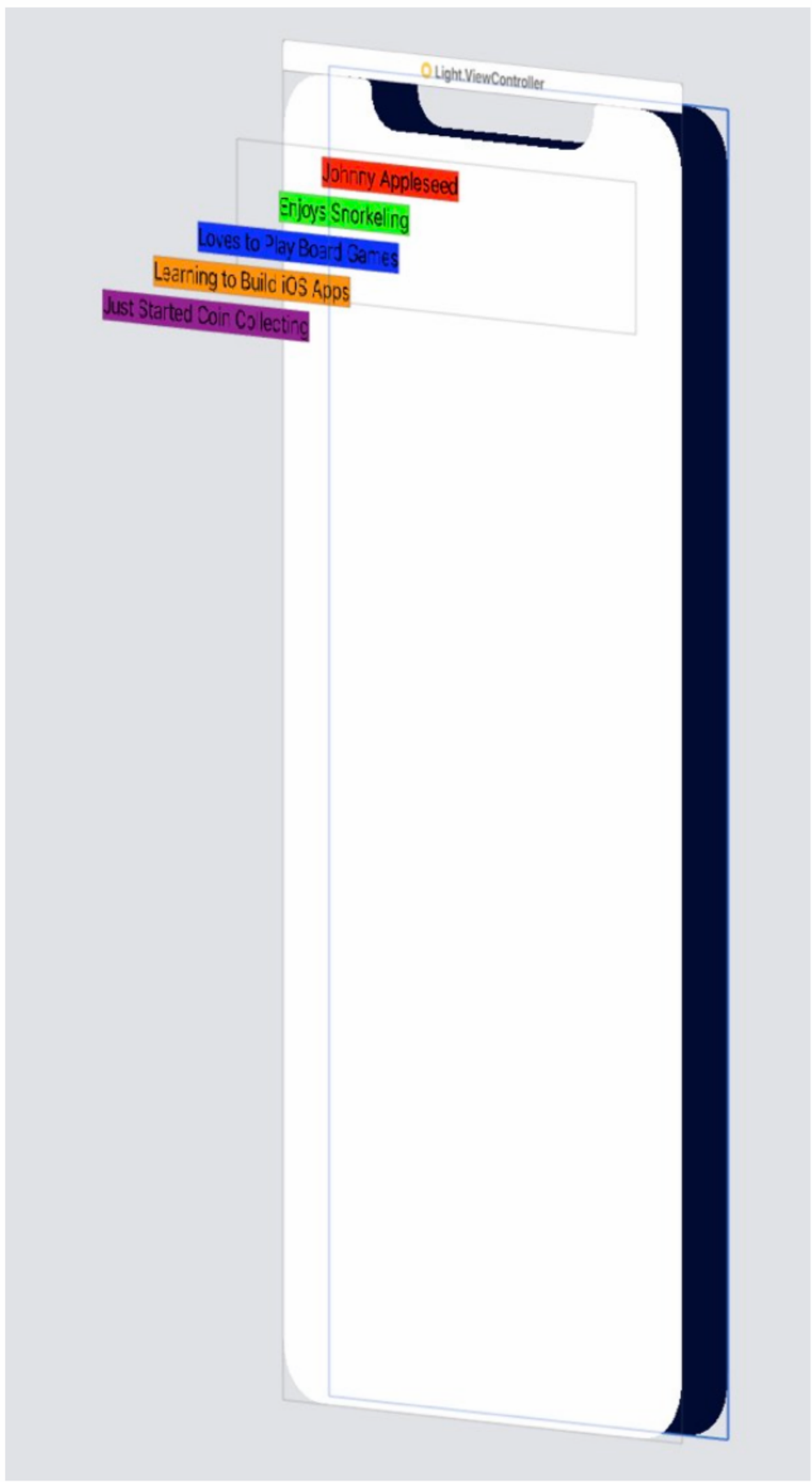
Now deselect the checkbox for this new property, and build and run your application using the iPhone XR Simulator. When the device is in Portrait mode (Compact Width, Regular Height), the red label will be displayed within the stack view. When you rotate the device to landscape, the trait collection updates to Regular Width, Compact Height and the red label disappears to save space.

LAB—CALCULATOR

DEBUG VIEW HIERARCHY

The view hierarchy can become *very* complex, with numerous stack views and an assortment of buttons, labels, and images. If you have issues at runtime where one view is covering another, or constraints are not working as you'd expect, Xcode includes a tool that can help you solve the problem.

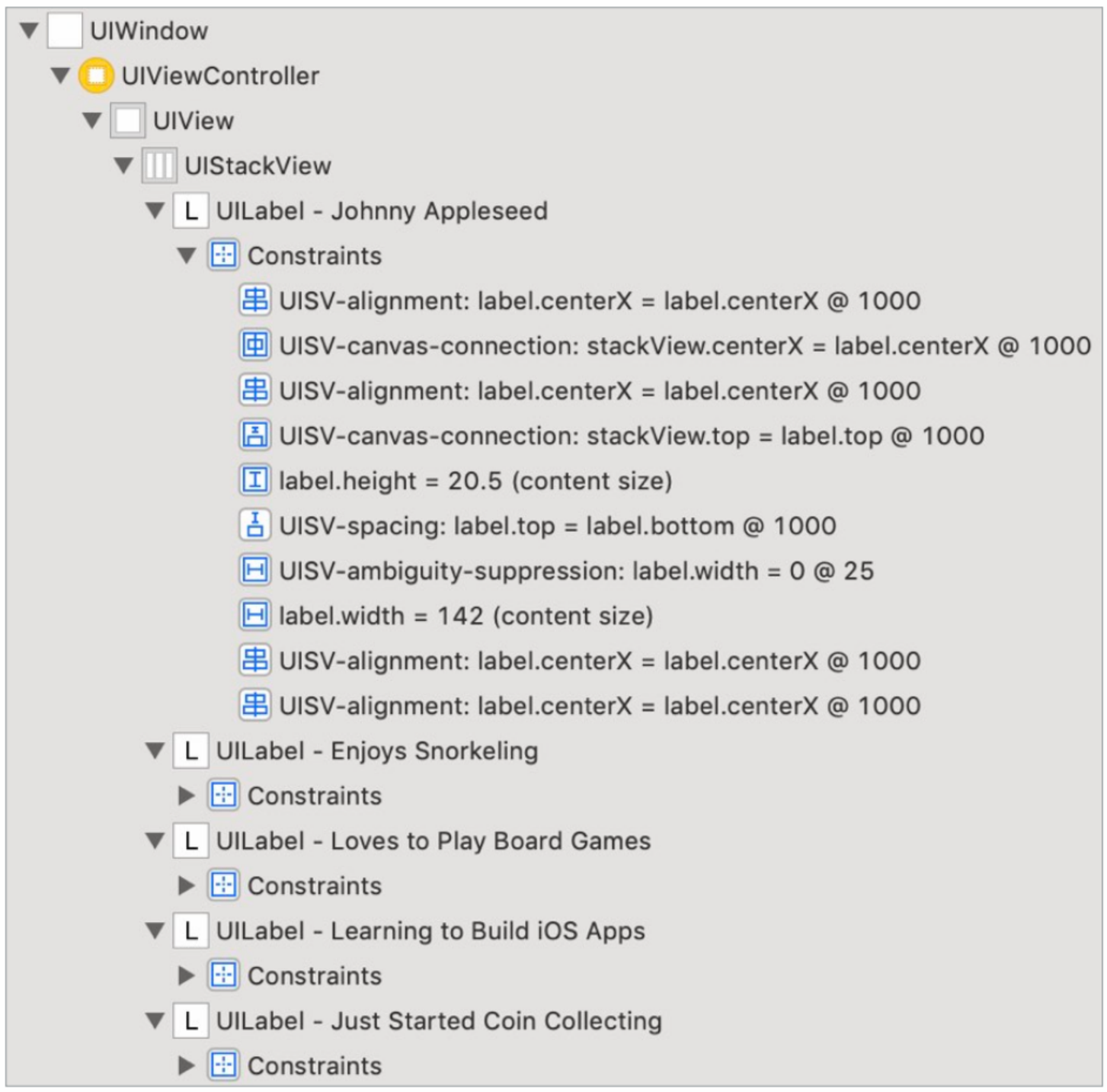
Build and run the app you just built that includes the colored labels. After the view has loaded, press the Debug View Hierarchy button at the top of the Debug area that extends from the bottom of Xcode. The view will appear in a 3D environment that you can swivel around to see how the view hierarchy was constructed.



Here is a breakdown of the screenshot. As you move back to front, the proceeding view is layered on top of the previous.

- The black view is your application's `UIWindow`
- The white view is your view controller's view
- The small clear rectangle is the `UIStackView` that contains the colored labels
- The red, green, blue, orange, and purple labels are added subsequently

Using the Debug navigator on the left, you can see a tree view of your hierarchy, along with the values of each of the constraints. This is a much quicker way to get the value of each constraint instead of printing them to the console.

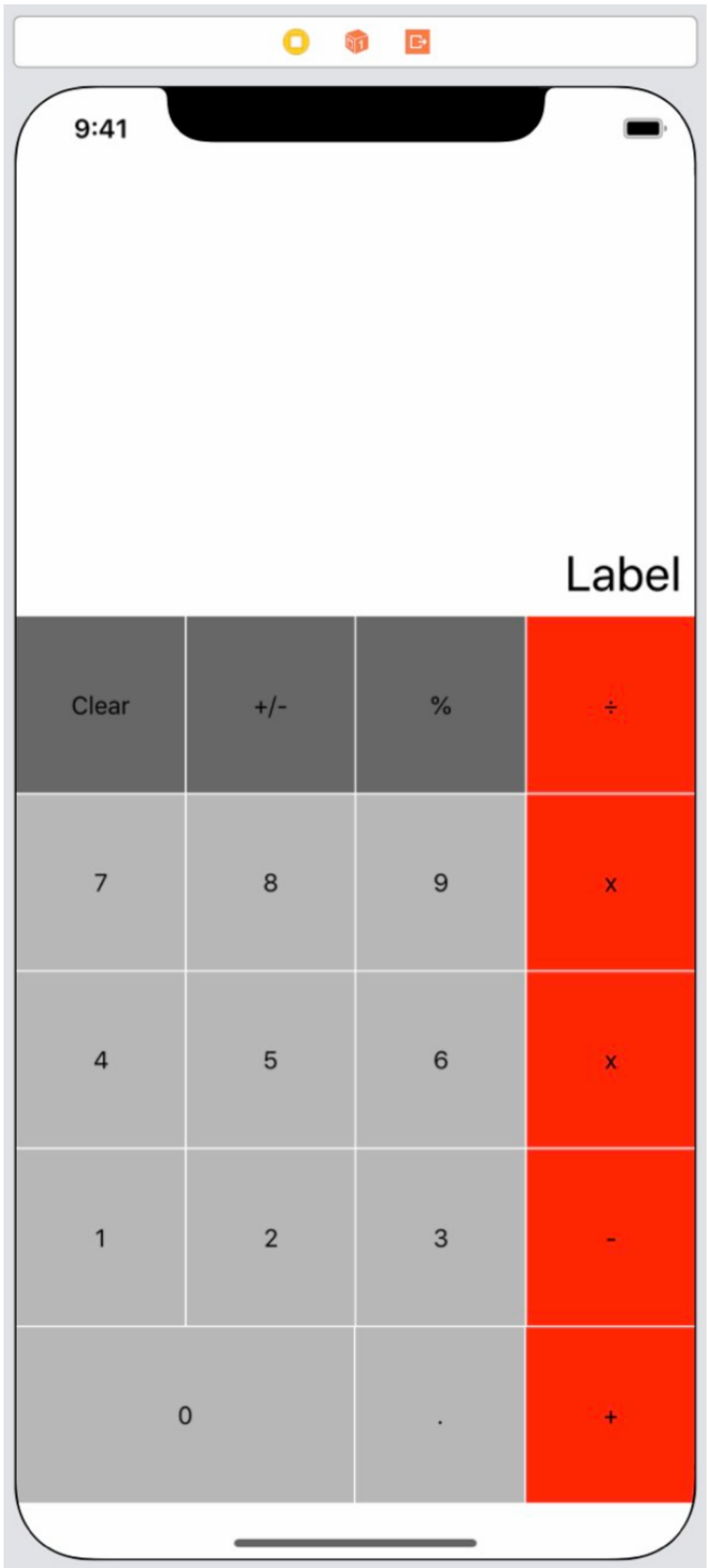


The Debug View Hierarchy tool also includes an assortment of buttons at the top of the Debug area that you can use to enable/disable the content of each view, hide/show constraints, and more. If you find yourself struggling to resolve an issue with a view, you should consider using this tool.

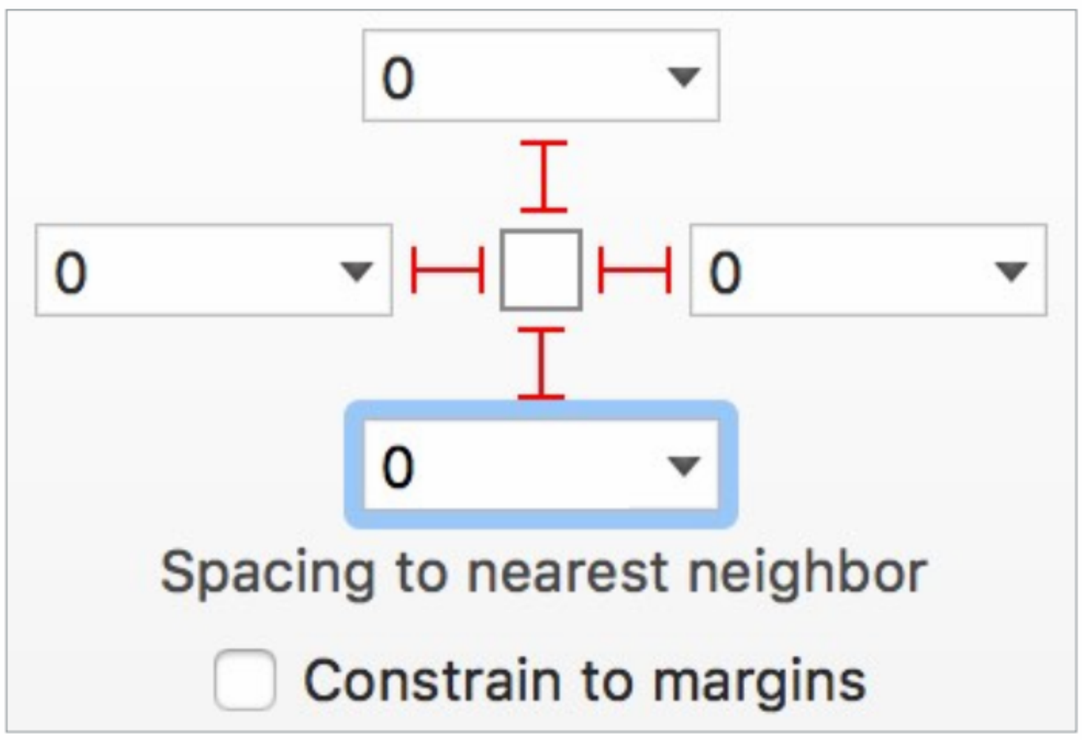
The objective of this lab is to use Auto Layout to create a view that will scale with the size and layout of any screen. You'll use view objects, constraints, and stack views to create a simple calculator that maintains its layout on all device sizes.

Create a new project called "Calculator" using the iOS template for a Single View App.

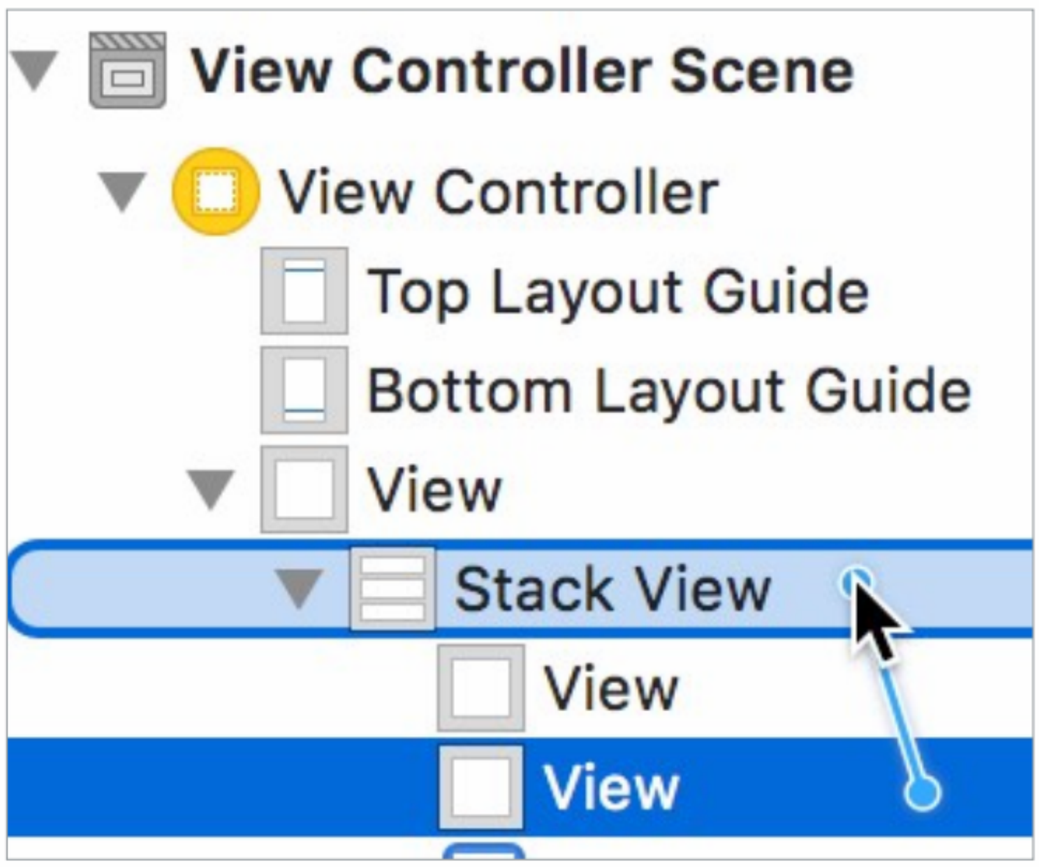
STEP 1: CREATE THE OUTER CONTAINERS



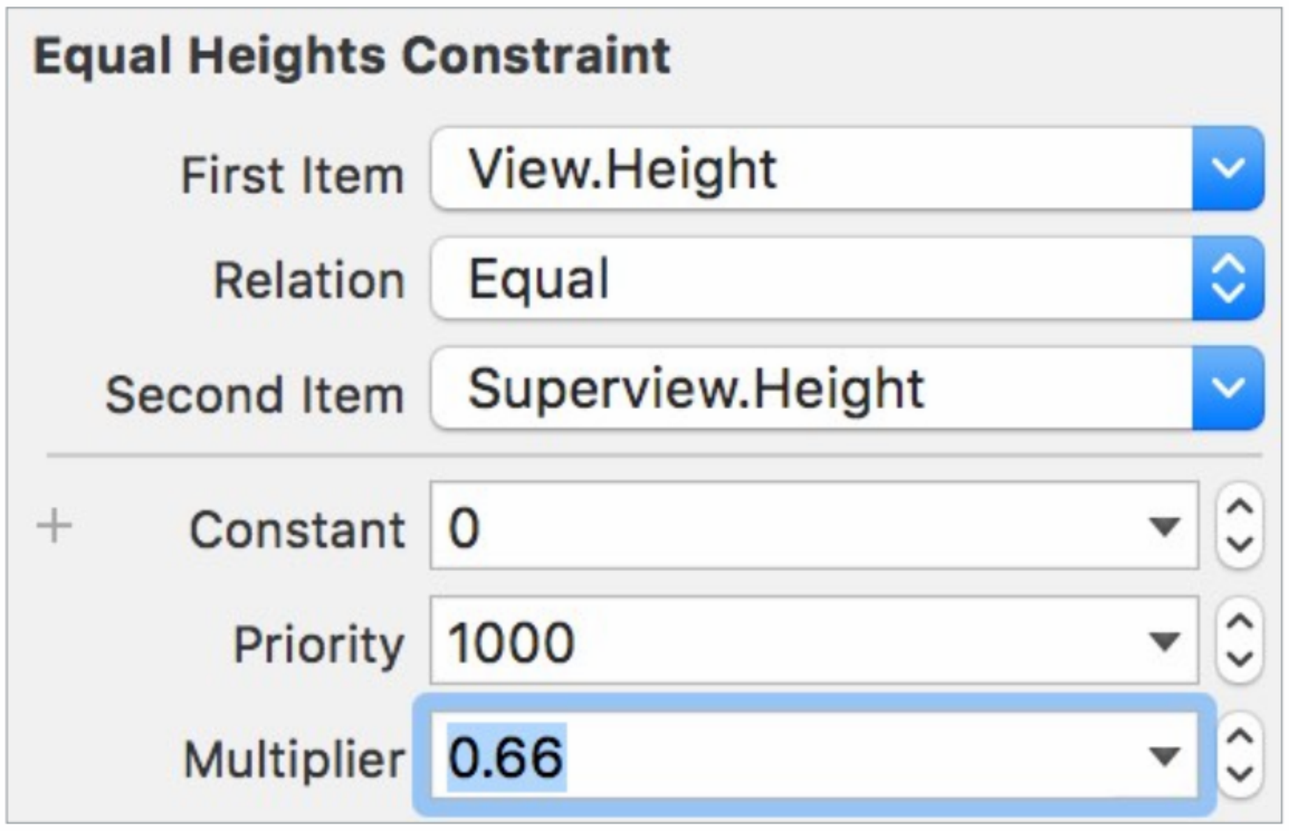
- In Interface Builder, set the "View as" device to iPhone XR.
- Ignoring the complexity of the buttons for now, it's easiest to break the calculator down into two distinct sections: the top third displays the digits that are entered, and the bottom two-thirds are used for input.
- Drag a vertical stack view from the Object library onto the scene. Use the Add New Constraints tool to add four constraints that align the top, leading, bottom, and trailing edges of the stack view to the the outer view's respective edges with 0 spacing. Click the Update Frames button, and the stack view should cover the entire screen.



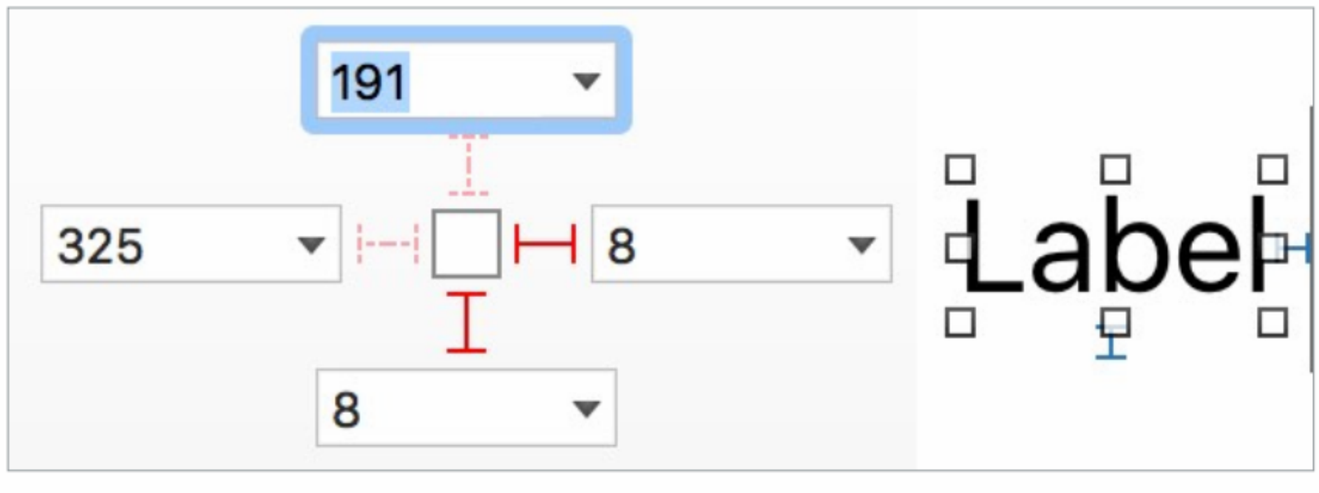
- Drag two `UIView` objects from the Object library into the stack view. Control-drag from the bottom view to the outer view, and select Equal Heights in the popup menu. This will set the inner view's height equal to the outer, which you will fix in the next step.



- Select the bottom view and open the Size inspector. Locate the height constraint and double-click it. Change the Multiplier value from 1 to 0.66, which will set the inner view's height equal to the outer view's height, multiplied by 0.66. The height of the bottom view will always be two-thirds the size of the view controller's view.

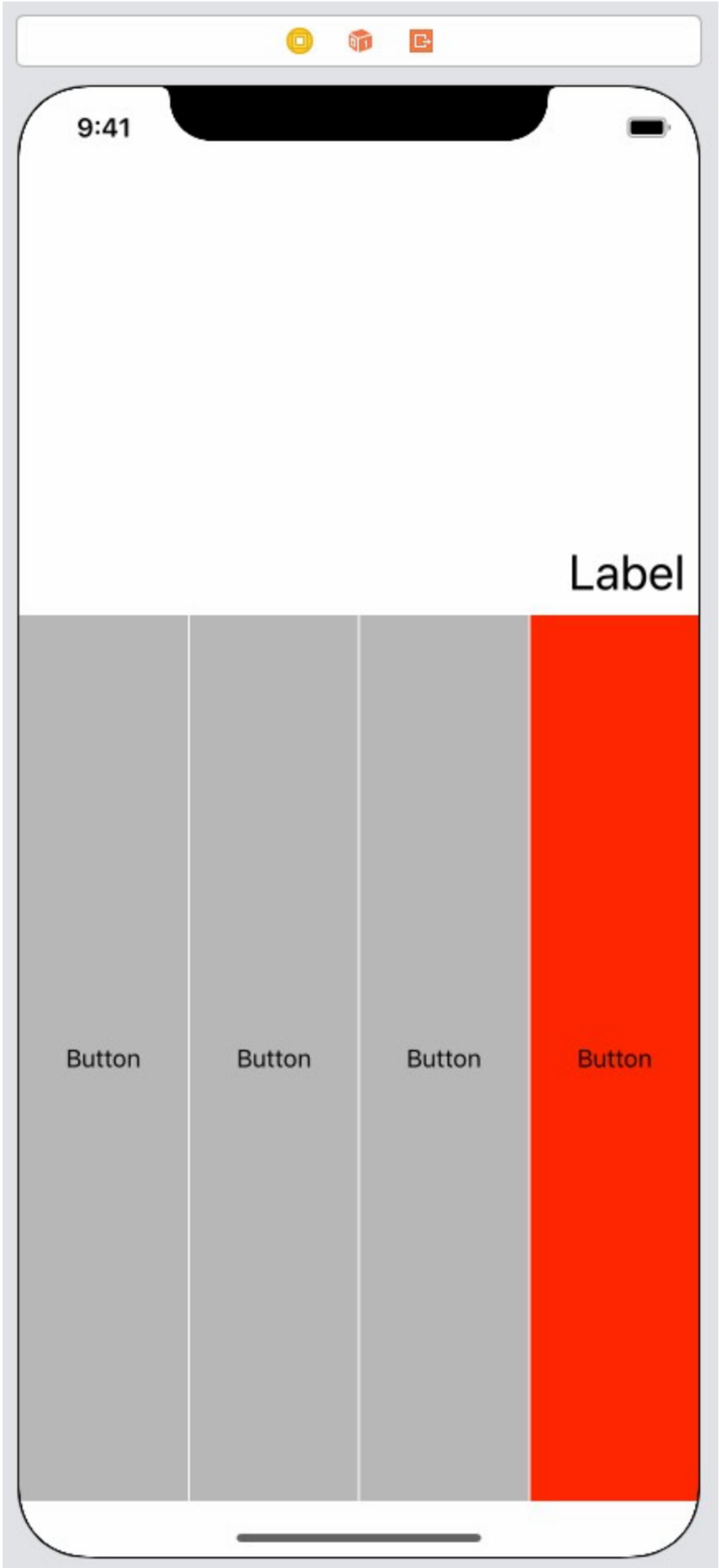


Add a `UILabel` from the Object library into the top container view. Set the font size to 30.0, then open the Add New Constraints tool. Add bottom and trailing constraints, with 8 pixels of spacing between them.



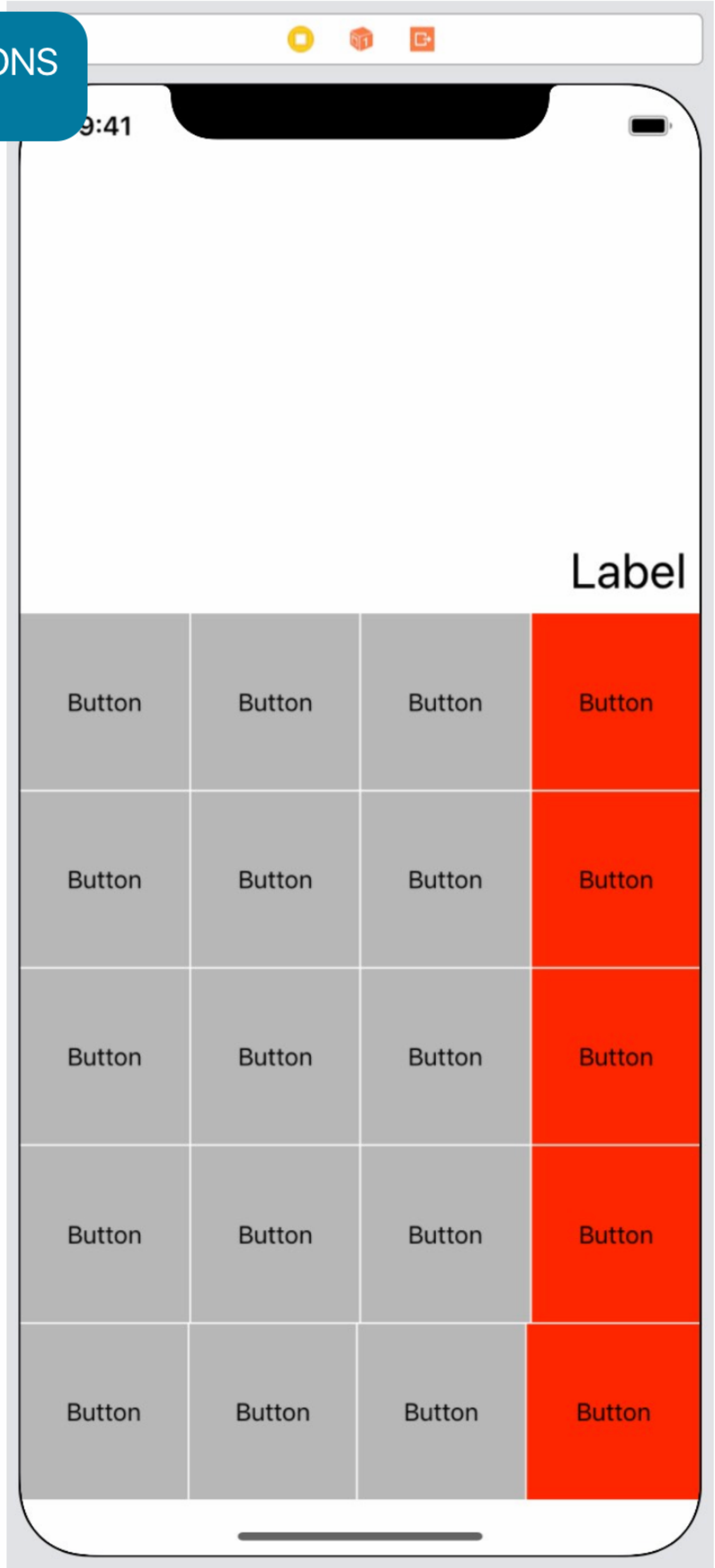
STEP 2: ADD AND SIZE BUTTONS

- The finished calculator has five rows of buttons, where each row has an equal height. You can use another vertical stack view to accomplish this. Drag a vertical stack view into the bottom container view, then use the Add New Constraints tool to constrain the top, leading, bottom, and trailing edges of the stack view to the the bottom container view's respective edges with 0 spacing. Click the Update Frames button, and the stack view should cover the entire bottom container.
- Select the newly-added vertical stack view, and open the Attributes inspector. Set the "Distribution" property to "Fill Equally" so that all subviews within the stack view will have the same height. Set the "Spacing" to 1, which will create the horizontal separation lines between each row.
- You can use horizontal stack views for each row of buttons. Add a horizontal stack view into the vertical stack view, and set its "Distribution" property to "Fill Equally" so that all subviews within the stack view have the same width. Set the "Spacing" to 1, which will create the horizontal separation lines between each subview.
- Add a `UIButton` into the horizontal stack view. Set the tint color to black, and the background color to a light gray color.
- Copy the button to your clipboard (Command-C), then paste (Command-V) three additional buttons into the horizontal stack view. Change the background color of the last button to red.

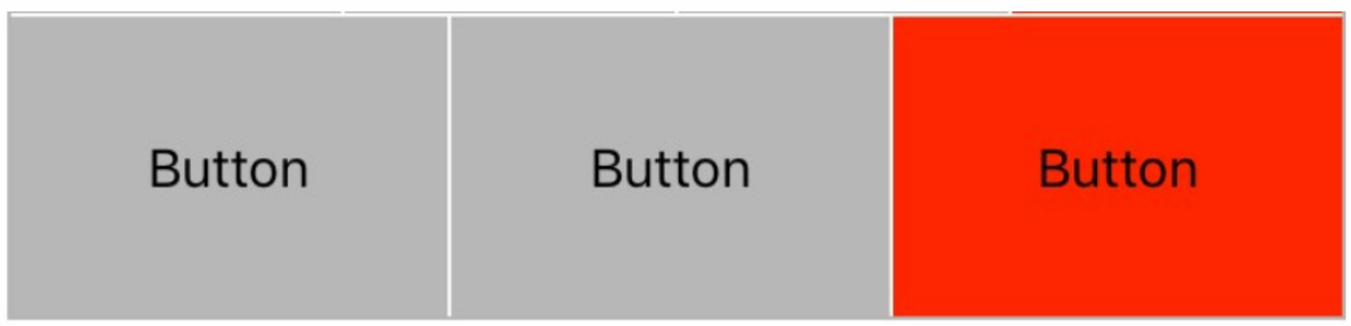


- Select the horizontal stack view in the Document Outline, and copy it to your clipboard (Command-C). Then paste (Command-V) four additional horizontal stack views into the vertical stack view.

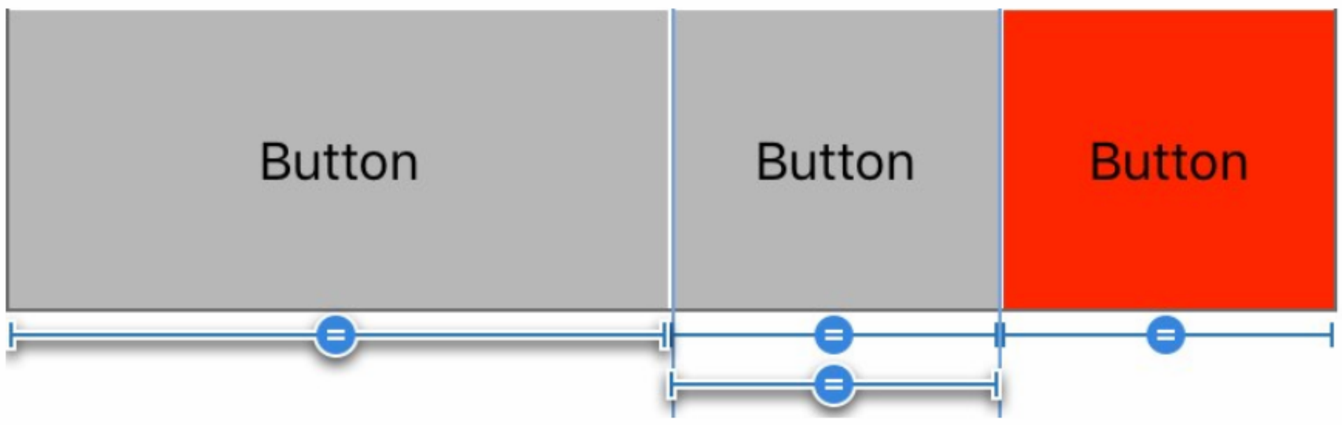
REVIEW QUESTIONS



- The bottom horizontal stack view uses only three buttons instead of four. Since some buttons are sized differently than others, you need to update the "Distribution" of this stack view to "Fill Proportionally". Then, delete one of the light gray buttons from the bottom so the stack view contains the proper amount of controls.



- Control-drag from the bottom red button to the light gray button next to it, and select "Equal Widths". This will ensure these two buttons are always the same size. Repeat this step between the two light gray buttons. Then select the left-most gray button, and open the Size inspector. Double-click its width constraint and change the multiplier to 2. This will ensure that the left gray button has twice the width of the other gray button.



- Finally, update the top three buttons to use a dark gray color, and update the button titles to match those on a traditional calculator.

Congratulations! You've used both constraints and stack views to create a simple calculator. Be sure to build and run your application on multiple iOS devices to verify that the constraints you've defined make sense on all screen sizes. Save your work to your project folder.

문제 1(총 5개)

If a button is centered vertically and horizontally within a view, which of the following constraints were most likely defined? (Check all that apply.)

- **A.** Center X
- **B.** Center Y
- **C.** Width
- **D.** Top

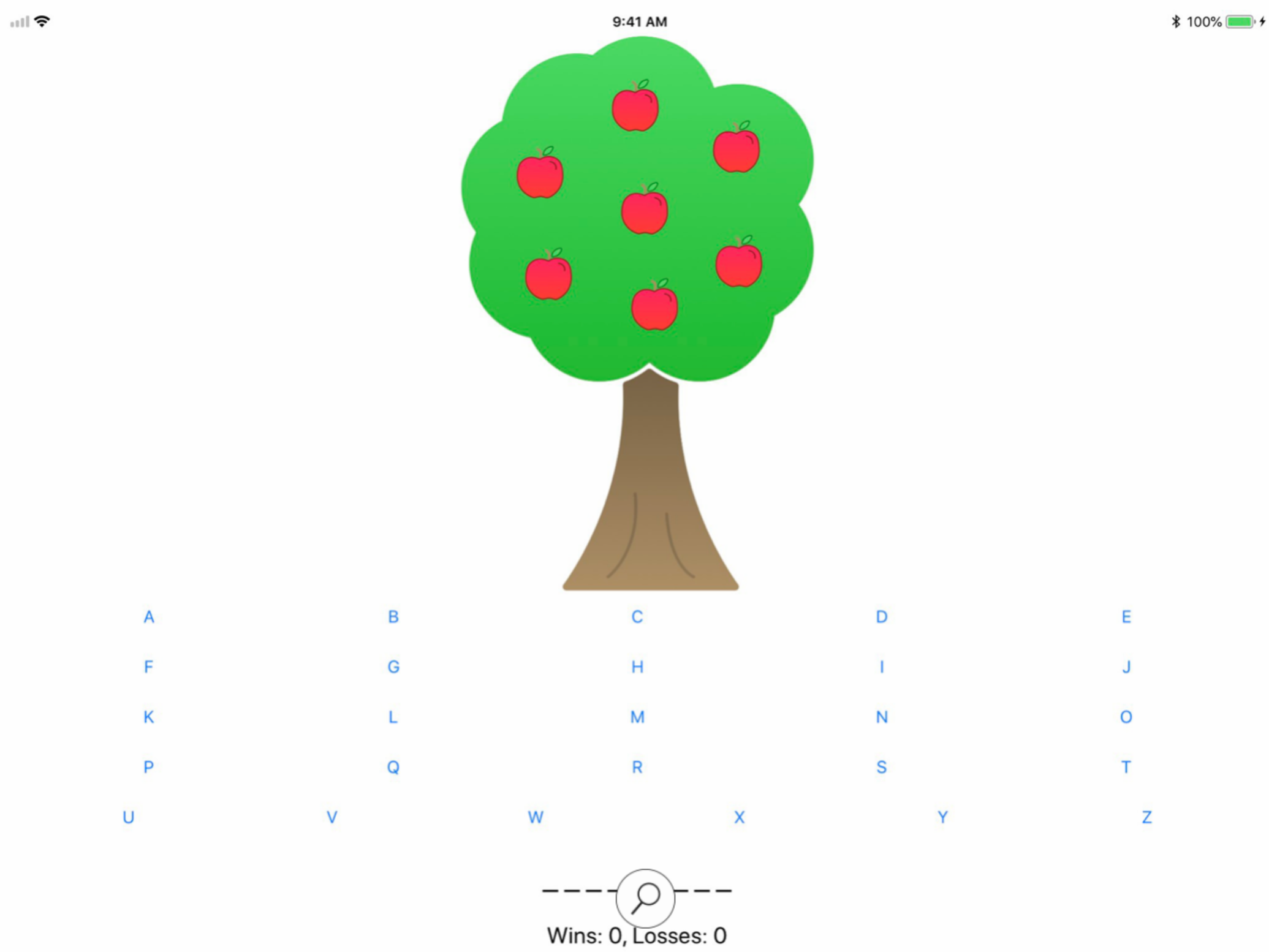
정답 확인

Guided Project: Apple Pie

So far in this unit, you've learned a lot about the fundamentals of Swift. Now it's time to put your knowledge to work.

Your project is to write a game called Apple Pie. In this simple word-guessing game, each player has a limited number of turns to guess the letters in a word. Each incorrect guess results in an apple falling off the tree. The player wins by guessing the word correctly before all the apples are gone.

As a new programmer, you may find this larger project a bit intimidating. Remember, building a working app happens one step at a time. If you get stuck on a particular step, go back and review that lesson. You can do it!



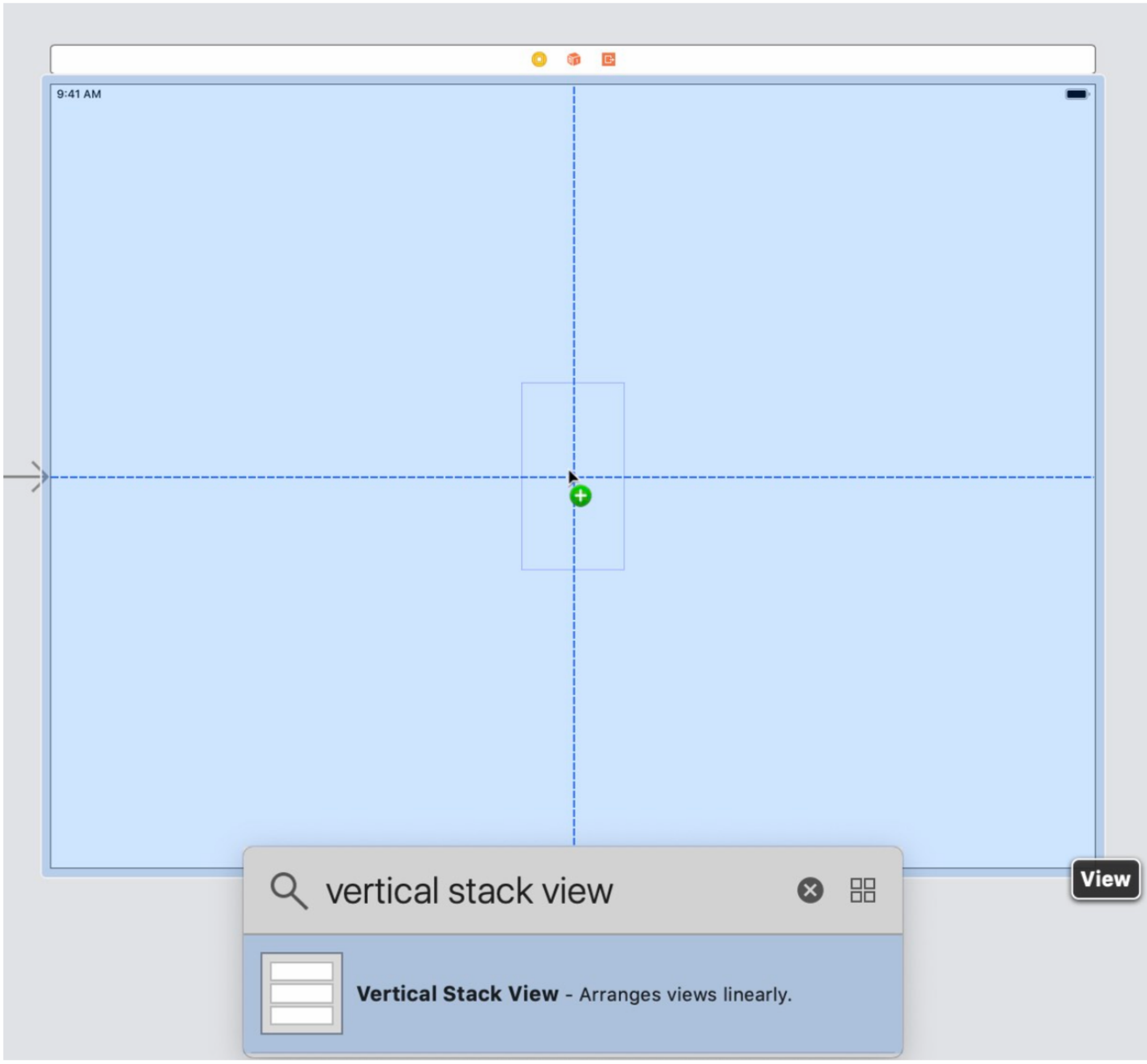
In this particular version of Apple Pie, whenever a round is won or lost, a new game is immediately started. If there are no more words left to guess, all buttons are disabled, and the app needs to be restarted.

PART ONE - BUILD THE INTERFACE

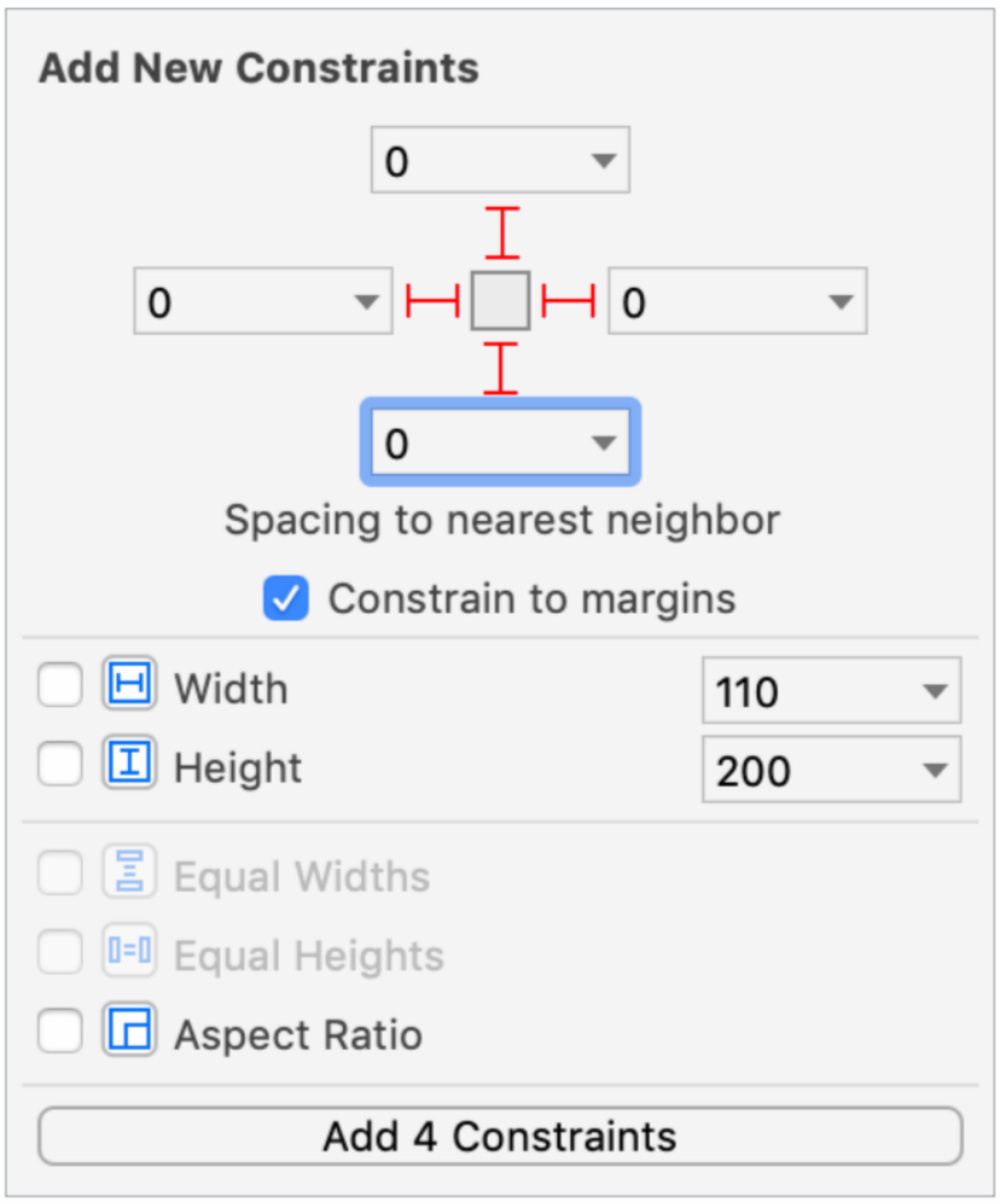
Create a new project using the Single View App template. Name the project "Apple Pie". This game is meant to be played on the iPad, and the interface that you'll be building doesn't accommodate a small iOS device. To make the app iPad only, select the project file in the Project Navigator, then under Deployment Info, change Devices from Universal to iPad.

Layout in Storyboard

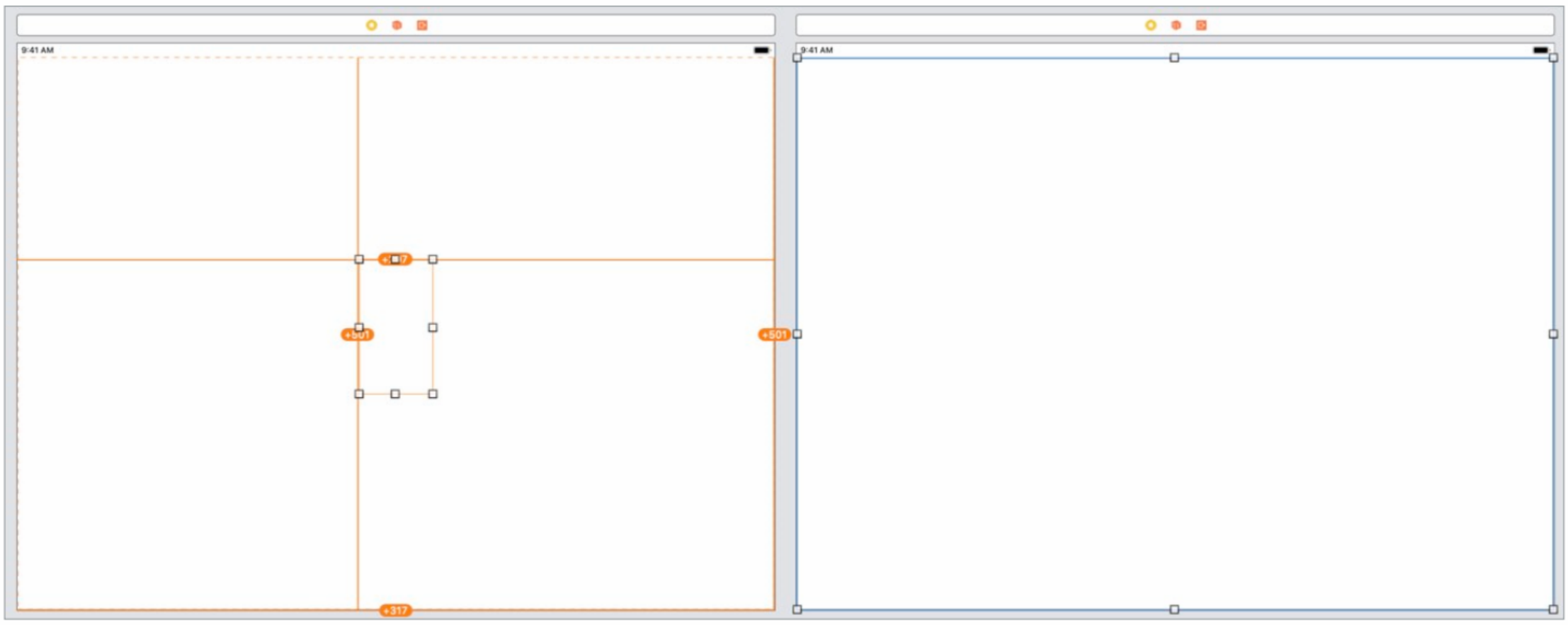
Open `Main.storyboard` and use the "View as" button to change the device and orientation to an iPad in Landscape mode. Since you'll be using Auto Layout to build the interface, it will work in Portrait mode as well. Look back at the image of the finished app. Using the Interface Builder tools that you've learned so far, how might you construct this interface? There is an image at the top, followed by a grid of buttons, and then two rows of labels containing text. Perhaps the simplest way to build this interface is by using a vertical stack view. Find the vertical stack view in the Object library, and drag it onto the iPad canvas.



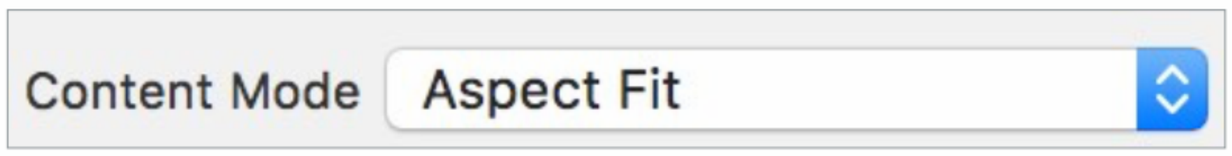
As you've learned in earlier lessons, you can determine the position, width, and height of a stack view by adding constraints between the stack and its superview. Where does the stack start, and where does it end? The image view is near the very top, and the last label goes along the bottom of the screen. The grid of buttons begins near the left edge, and ends near the right edge. Therefore, the stack view could be constrained to cover the entire screen. Select the stack view, then use the Add New Constraints tool to add four constraints. Set all four fields at the top of the popover to 0. The red indicators will illuminate to show the edges that are being constrained. When you're done, click "Add 4 Constraints".



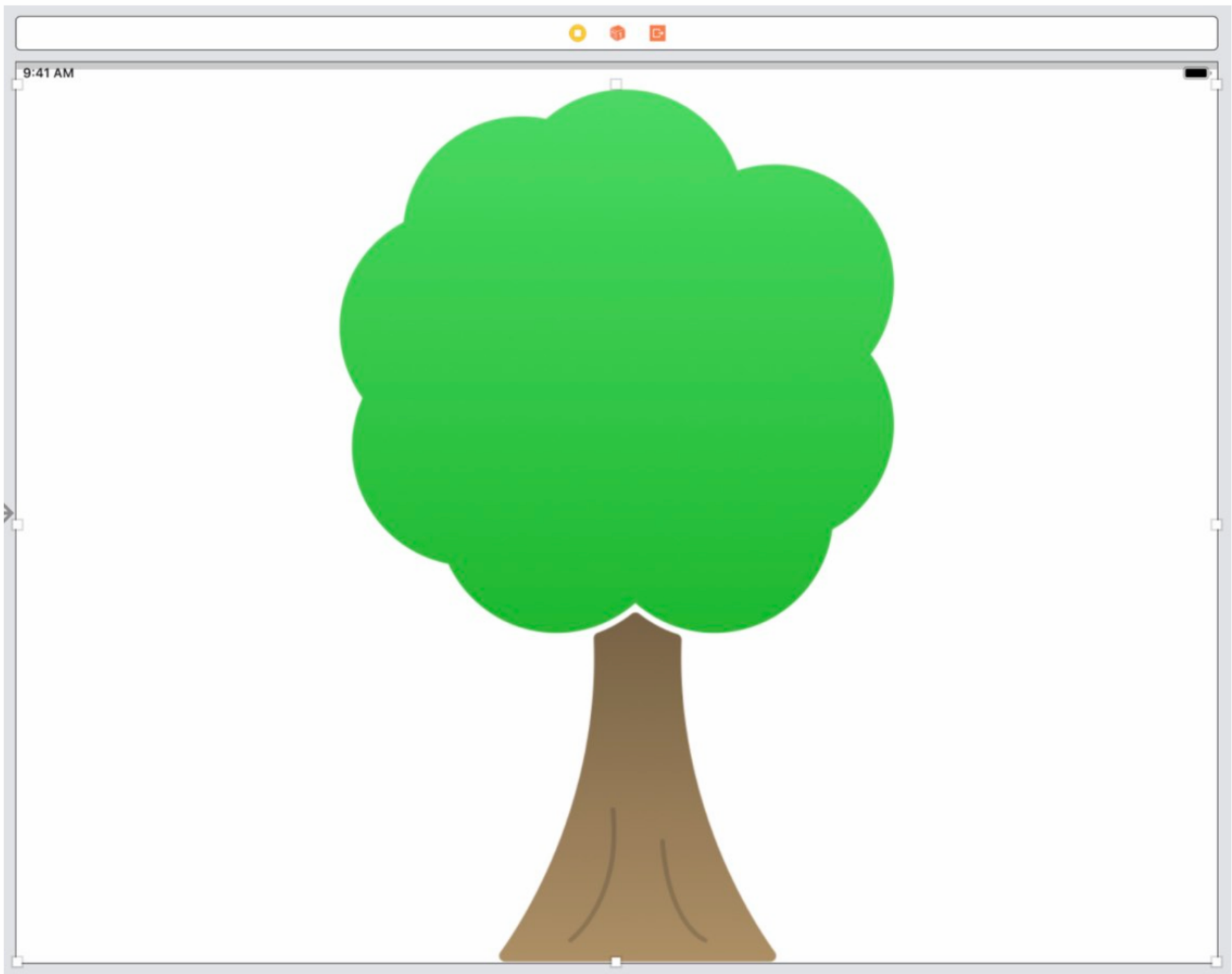
If you add the constraints and yellow warning lines appear around the stack view, press the Update Frames button so that the stack view frame resizes to match the constraints, or rules, you've applied.



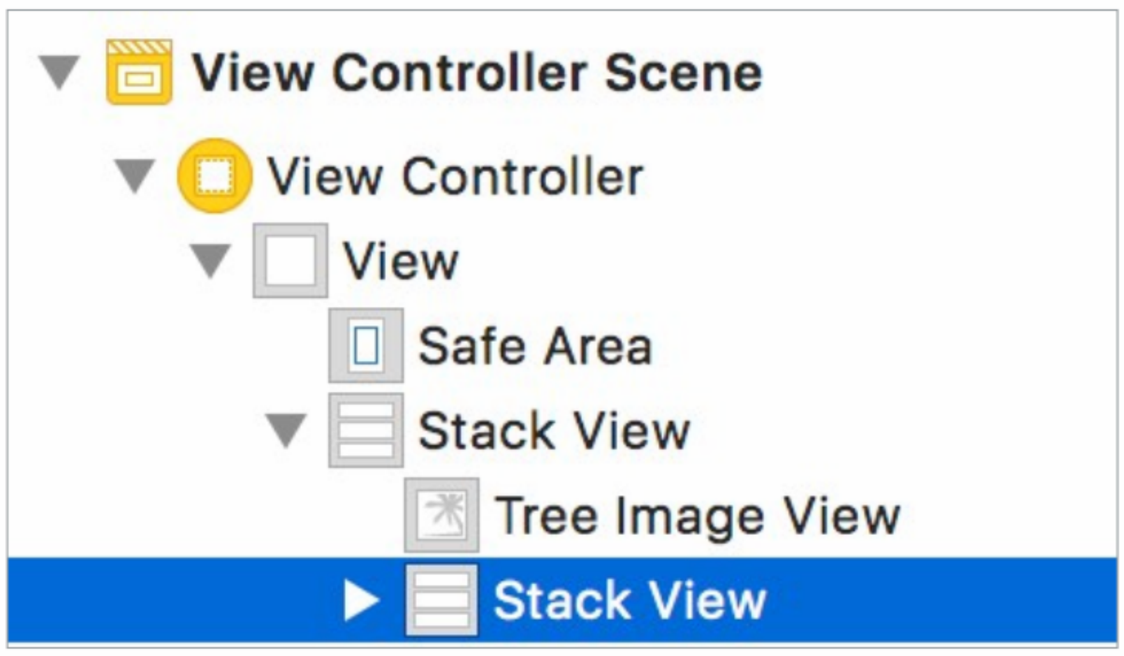
Now you're ready to add views and controls to the stack. Search the Object library for an image view, and drag one into the stack view. Since it's the only item within the stack, it will take up the entire space of the stack. Change the "Content Mode" of the image view to "Aspect Fit" in the Attributes inspector. This will ensure the width and height of the image is not distorted by the proportions of the image view.



Add the apple tree images from the student resources folder into your `Assets.xcassets` folder. Once they've been added, you can go back to the image view in the storyboard and update the image property to use one of the tree images. Even though you will be updating the image view's image in code, this can help you visualize your interface from within the storyboard.

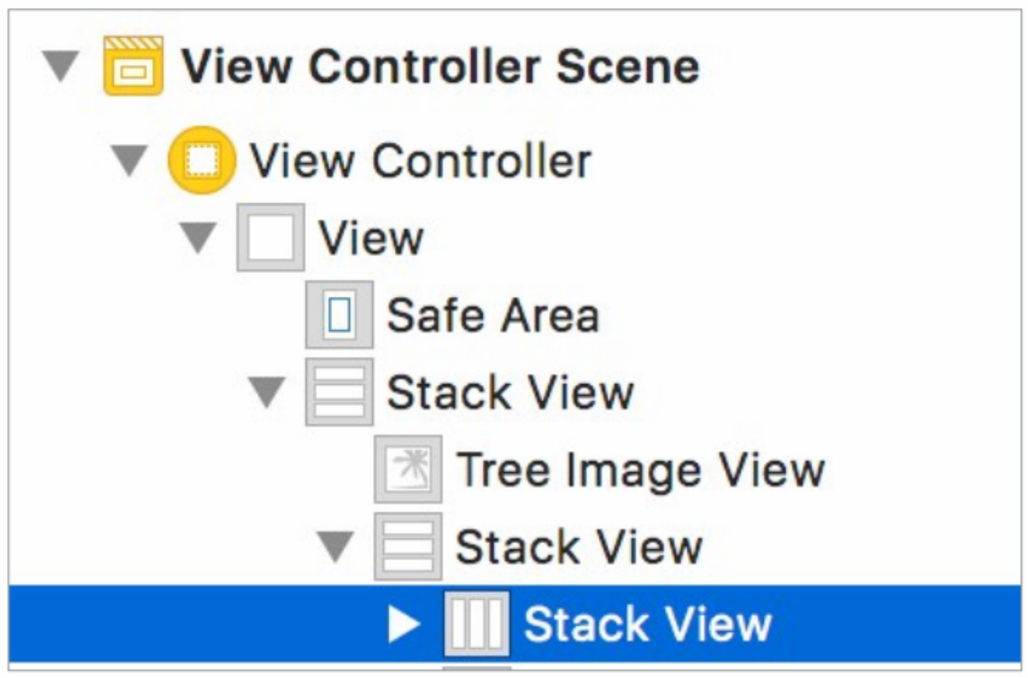


The button grid is a bit more complex. Like a checkerboard, the buttons are arranged in a set of rows and columns. You can imagine each row as its own horizontal stack view, and you can contain these rows in a vertical stack view. It's five horizontal stack views within one vertical stack view. Use the Object library to add a vertical stack view below the image view. You can reorder the items within the stack by dragging them around in the Document Outline. Give the newly-added stack view a height of 220.

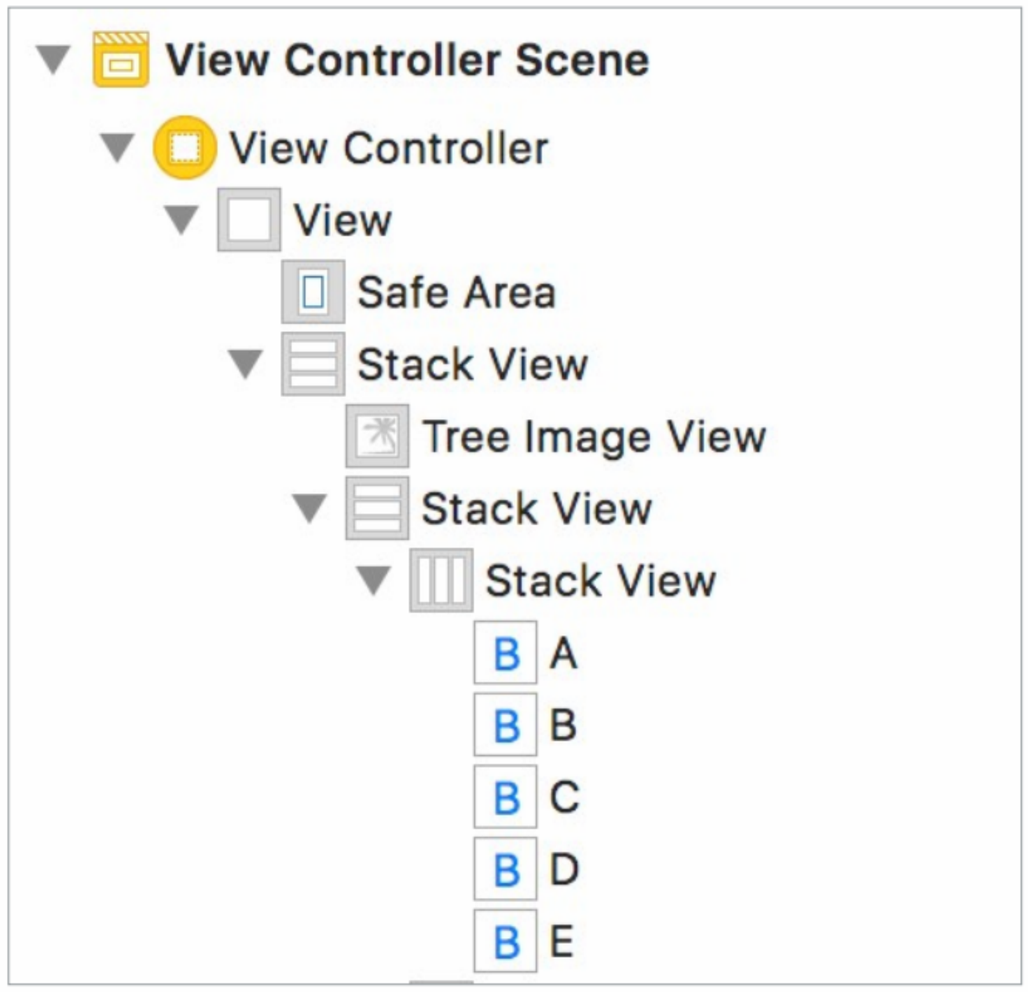


Each of the horizontal stack views (or rows) that you'll add will be equal in size. Select the newly-added vertical stack view, then open the Attributes inspector and change the Distribution property to Fill Equally.

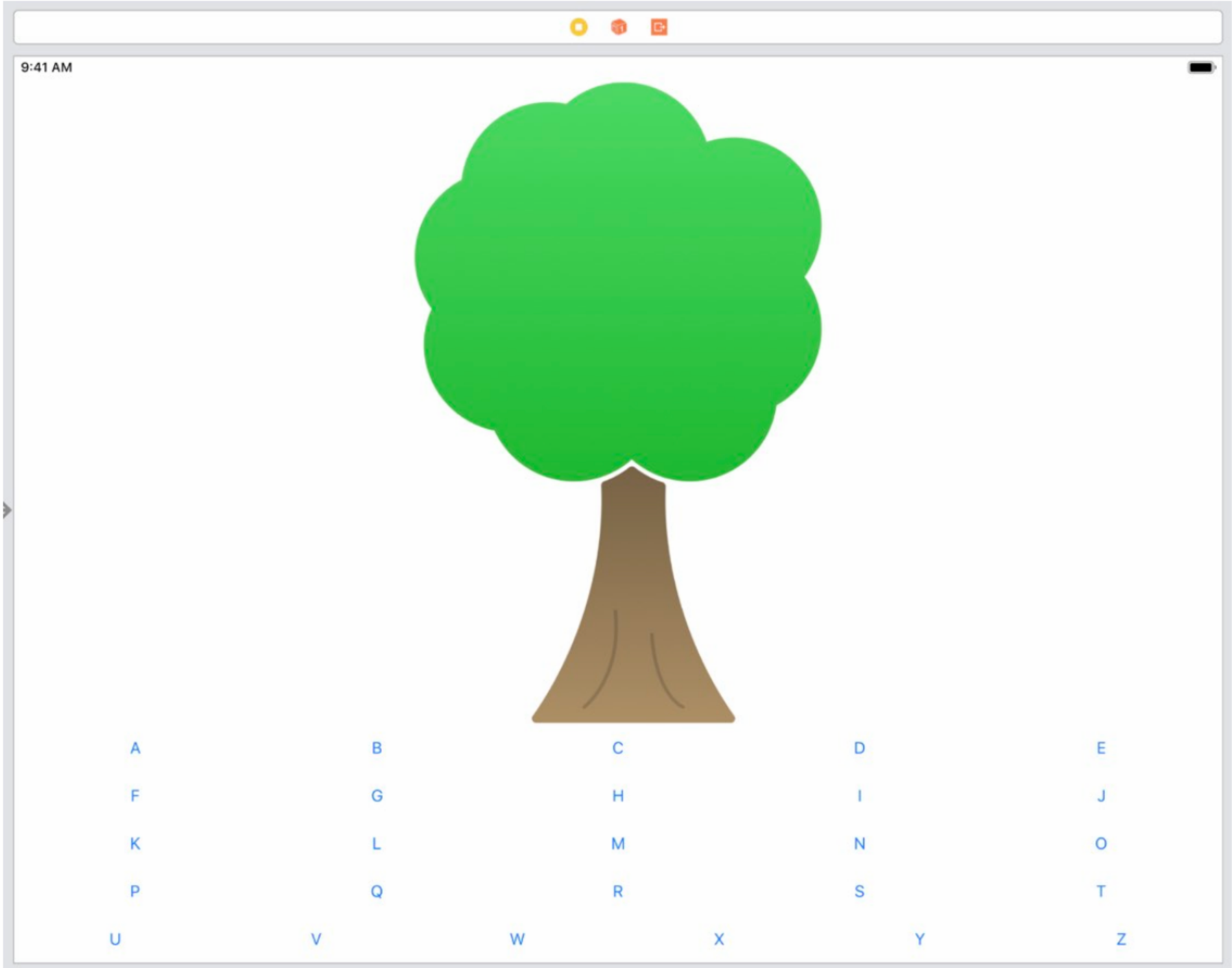
Now add a horizontal stack view within the vertical stack. Each of the buttons (or columns) are equal in size, so select the horizontal stack view, then open the Attributes inspector and change the Distribution property to Fill Equally.



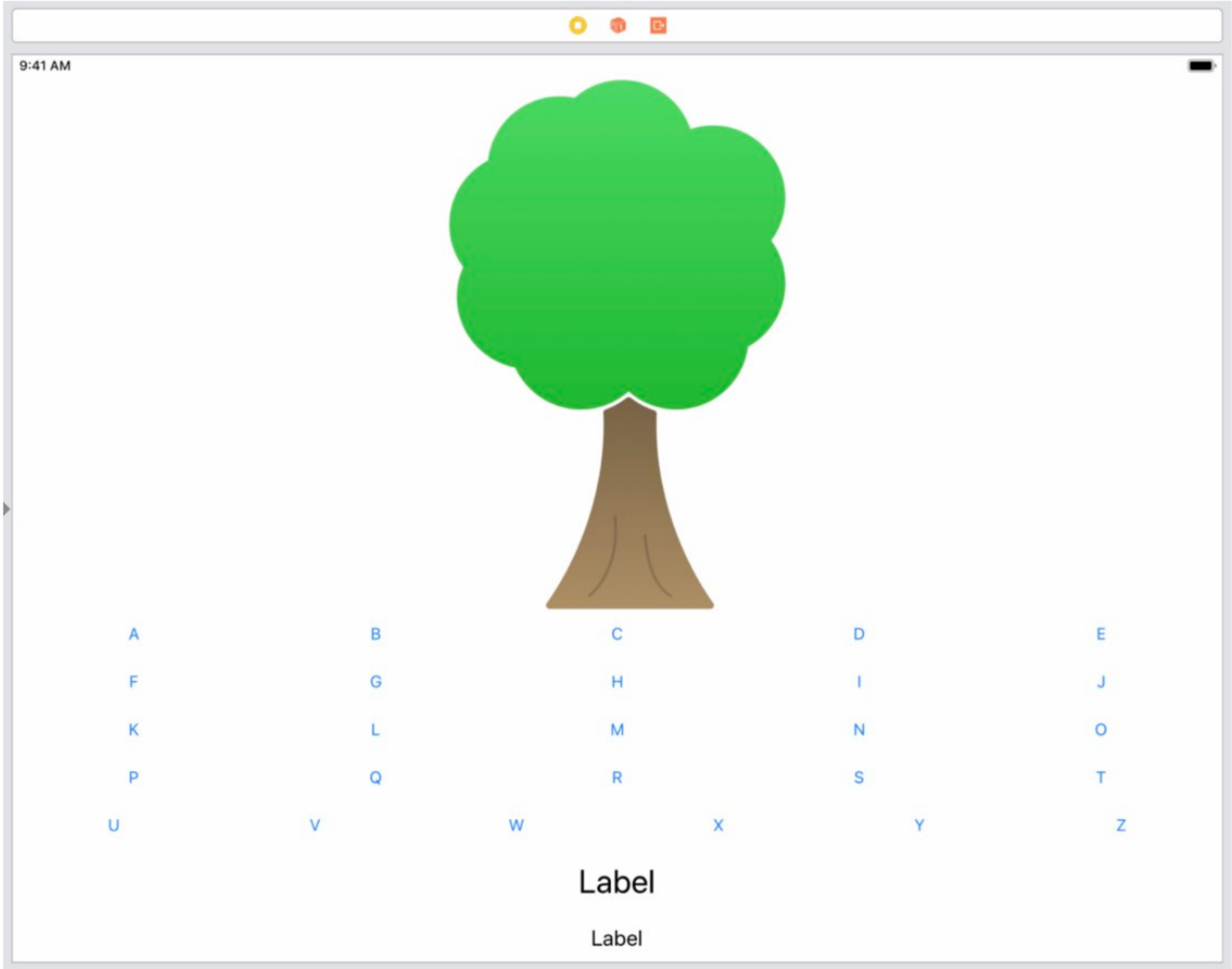
In the interface, there are five buttons per row, except for the last row which has six. Use the Object library to add a button to the horizontal stack view. Now select the button, and copy it to your clipboard (Command-C). Now paste a new copy of the button into the stack view (Command-V), and repeat this step until the stack view contains five buttons. Use the Attributes inspector to update the text of each button to a single capital letter.



Now select the horizontal stack view, and copy it to your clipboard. Paste a new copy of the stack view, and repeat this step until the vertical stack view contains five horizontal stacks. Finally, you need to add one additional button into the last horizontal stack view so that you have twenty-six buttons total.



Below the grid of buttons, add two labels from the Object library. Use the Attributes inspector to change the alignment of the text to Centered, then set the Font Size of the first label to 30.0, and the second to 20.0. Add a height constraint of 60 to the first label, and a height constraint of 44 to the second label. Since every view besides the image view has a height constraint, the image view will adjust its size accordingly, while the other views stay consistent.



Before you move on, verify that you don't have any warnings or errors in Interface Builder. You can build and run your app to ensure the layout looks correct on different iPad simulators.

Create Outlets and Actions

During gameplay, the text of the labels will change whenever a letter is guessed correctly, or whenever a new round begins. The image will need to change whenever an incorrect letter is guessed, and the letter buttons need to be disabled whenever they're pressed, and re-enabled before each round. In order to update these views, you need to create outlets so that you can reference the views in code.

Open the Assistant editor so that the `ViewController` class definition appears to the right of the storyboard. Next, select the image view on the left, and Control-drag to an area within the class definition, and release the mouse button. In the popup that appears, name the outlet `treeImageView`. When you press the Connect button, the code for the outlet will be created.

```
@IBOutlet var treeImageView: UIImageView!
```

Repeat this process for the two labels. Name the top label `correctWordLabel`, and the bottom label `scoreLabel`.

```
@IBOutlet var correctWordLabel: UILabel!
@IBOutlet var scoreLabel: UILabel!
```

It would be very tedious to create a separate outlet for each `UIButton`. Instead, you can create one outlet that holds the collection of buttons. Begin by selecting the first button with the letter "A" as the title. Control-drag to the class definition to create an outlet, then release the mouse button. In the popup that appears, change the Connection type from Outlet to Outlet Collection. Then set the name of the outlet collection to `letterButtons`. When you press Connect, an outlet will be created that references a collection of buttons.

```
@IBOutlet var letterButtons: [UIButton]!
```

Now click and drag from the circle next to the outlet collection to another button. A blue rectangle will appear, indicating a valid connection. Release the mouse button, then repeat this step for each button in the scene.

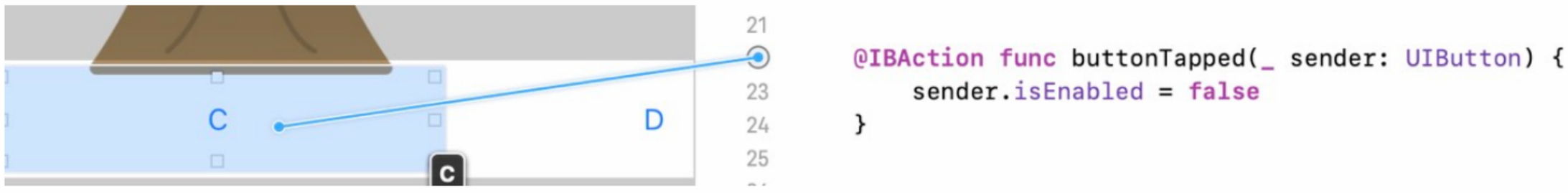
Each of the buttons also needs to be tied to an action. Again, it would be tedious to create a separate outlet for each button. Instead, you can create one action that all of the buttons call when pressed. Begin by selecting the first button with the letter "A" as the title. Control-drag to the class definition, and release the mouse button. In the popup that

appears, change the Connection type to Action. Set the name of the action to `buttonPressed`, and change the type of the argument to `UIButton`. When you press Connect, the method will be created. Whenever a letter button is tapped, it should be disabled (a player can't select a letter more than once in the same round).

```
@IBAction func buttonTapped(_ sender: UIButton) {
  sender.isEnabled = false
}
```

Why change the argument type from `Any` to `UIButton` in the popup? Since there are twenty-six buttons connected to the same action, you'll need to use the `sender` to determine *which* button, specifically, triggered the method. If `sender` has the `Any` type, you can't access the `title` property.

Control-drag the rest of the buttons to the existing action. You'll know the connection is valid because a blue rectangle will appear as you hover the mouse near the method.



Build and run your application to verify that tapping each button disables it. It'll become more apparent what else to do inside this method after you build other portions of the app.

PART TWO–BEGINNING A GAME

Now you're set to work on the Apple Pie game logic.

Define Words and Turns

Your first task is to supply a list of words for players to guess. At the top of `ViewController`, define a variable called `listOfWords`. Fill this array with words: food names, hobbies, animals, household objects, or whatever else. To keep things simple, use only lowercase letters.

```
var listOfWords = ["buccaneer", "swift", "glorious",
"incandescent", "bug", "program"]
```

Below `listOfWords`, define a constant called `incorrectMovesAllowed` which establishes how many incorrect guesses are allowed per round. The lower the number, the harder it will be for the player to win. There are seven different images of apple trees provided, so you'll want this value to be between 1 and 7.

```
let incorrectMovesAllowed = 7
```

Define Number of Wins and Losses

After each round, the bottom label will display an updated count of the number of wins and losses. Create two variables to hold each of these values, and set the initial values to 0.

```
var totalWins = 0
var totalLosses = 0
```

Begin First Round

When the application launches, the `viewDidLoad()` method of `ViewController` is called. This is a great place to start a new round. Define a method called `newRound`.

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

func newRound() {

}
```

What does it mean to start a new round? Each round begins with the selection of a new word, and resetting the number of moves the player can make to `incorrectMovesAllowed`. It would be helpful to hold the state of the game inside of a `Game` struct. Create a new file in your project by selecting File -> New -> File (Command-N) from the Xcode menubar. Select "Swift File" as your template, then select Next. Name the file `Game.swift`.

Inside of `Game.swift`, define a struct called `Game`. For now, you know that an instance of a `Game` has two properties: the word, and the number of turns you have left to properly guess the word.

```
import Foundation

struct Game {
    var word: String
    var incorrectMovesRemaining: Int
}
```

Back in the `newRound()` method, you can create a new instance of a `Game`. You should create a property that holds the current game's value so that it can be updated throughout the view controller code. You can give the `Game` a new word in the initializer by removing the first value from the `listOfWords` collection, and set `incorrectMovesRemaining` to the number of moves you allow, stored in `incorrectMovesAllowed`.

```
var currentGame: Game!

func newRound() {
  let newWord = listOfWords.removeFirst()
  currentGame = Game(word: newWord, incorrectMovesRemaining:
incorrectMovesAllowed)
}
```

Why does the `currentGame` variable have an explanation mark at the end? For a brief moment between the app launch and the beginning of the first round, `currentGame` does not have a value. This is a concept you will cover in the next unit, but for now, just know that the exclamation mark means that it's ok for this property not to have a value for a short period.

Now that you've started a new round, you need to update the interface to reflect the new game. Create a separate method called `updateUI()` that will handle the interface updates, then call it at the end of `newRound`. Inside of this method, there are two pieces of the interface that you can update with the code you've written thus far: the score label, and the image view. The score label uses simple string interpolation to combine `totalWins` and `totalLosses` into a single string. Since each of the tree images are named "Tree X", where X is the number of moves remaining, you can use string interpolation once more to construct the image name.

```
func newRound() {
    let newWord = listOfWords.removeFirst()
    currentGame = Game(word: newWord, incorrectMovesRemaining:
    incorrectMovesAllowed)
    updateUI()
}

func updateUI() {
    scoreLabel.text = "Wins: \(totalWins), Losses: \
(totalLosses)"
    treeImageView.image = UIImage(named: "Tree \
    (currentGame.incorrectMovesRemaining)")
}
```

Build and run your application. The score label and the image view should update to reflect the beginning of a new round. Great job!

PART THREE-UPDATE GAME STATE

So far, you've built a working interface, and you've successfully started a new round. Now you need to add some code that progresses the game further towards a win or a loss.

Extract Button Title

Currently, the `buttonTapped(_:)` method disables whichever button the player used to guess, but it does not update the game state. Whenever a button is clicked, you should read the button's title, and determine if that letter is in the word the player is trying to guess. Begin by reading the title that's used when the button is in its "normal" state. Not all buttons have titles, so you'll need to add an exclamation mark to tell the Swift compiler that your button *does* have a title. (You will learn more about the exclamation mark in the next unit) You should lowercase the letter, because it'll be much easier to compare everything in lowercase, and then convert it from a `String` to a `Character`.

```
@IBAction func buttonTapped(_ sender: UIButton) {
    sender.isEnabled = false
    let letterString = sender.title(for: .normal)!
    let letter = Character(letterString.lowercased())
}
```

Guess Letter

Now that you have the letter that the player guessed, what should you do with it? A `Game` manages how many more moves are remaining, but it doesn't know which letters have been selected during the round. You should add a collection of characters to `Game` that keeps track of the selected letters, called `guessedLetters`. Then add a method in `Game` that receives a `Character`, adds it to the collection, and updated `incorrectMovesRemaining`, if necessary. (By adding the `guessedLetters` property, you'll need to update the initialization for `currentGame`.)

```
struct Game {
    var word: String
    var incorrectMovesRemaining: Int
    var guessedLetters: [Character]
```

```
    mutating func playerGuessed(letter: Character) {
      guessedLetters.append(letter)
      if !word.contains(letter) {
        incorrectMovesRemaining -= 1
      }
    }
}

func newRound() {
    let newWord = listOfWords.removeFirst()
    currentGame = Game(word: newWord, incorrectMovesRemaining:
    incorrectMovesAllowed, guessedLetters: [])
    updateUI()
}

@IBAction func buttonTapped(_ sender: UIButton) {
    sender.isEnabled = false
    let letterString = sender.title(for: .normal)!
    let letter = Character(letterString.lowercased())
    currentGame.playerGuessed(letter: letter)
    updateUI()
}
```

Build and run your application. As you press each button, the character gets added to the list of letters the player has guessed, and `incorrectMovesRemaining` decreases by 1 for each incorrect letter.

PART FOUR-CREATE REVEALED WORD

Using `word` and `guessedLetters`, you can now compute a version of the word that hides the missing letters. For example, if the word for the round is "buccaneer" and the user has guessed the letters "c", "e," "b," and "j," the player should see "b_cc__ee_" at the bottom of the screen.

To begin, create a computed property called `formattedWord` within the definition of `Game`. Here's one way to compute `formattedWord`:

- Begin with an empty string variable.
- Loop through each character of `word`.
- If the character is in `guessedLetters`, convert it to a string, then append the letter onto the variable.
- Otherwise, append _ onto the variable.

```
var formattedWord: String {
    var guessedWord = ""
    for letter in word {
        if guessedLetters.contains(letter) {
            guessedWord += "\(letter)"
        } else {
            guessedWord += "_"
        }
    }
    return guessedWord
}
```

Now that `formattedWord` is a property that your UI can display, try using it for the text of `currentWordLabel` inside of `updateUI()`.

```
func updateUI() {
    correctWordLabel.text = currentGame.formattedWord
    scoreLabel.text = "Wins: \(totalWins), Losses: \
    (totalLosses)"
    treeImageView.image = UIImage(named: "Tree \
    (currentGame.incorrectMovesRemaining)")
}
```

Build and run your application. Letters will be added to the `guessedLetters` collection as they're selected, and `formattedWord` will be re-calculated whenever it's accessed in `updateUI()`.

You may notice a new issue: Because multiple underscores appear as a solid line in the interface, it can be difficult to tell how many letters are in the word. One solution would be to add spaces during your computation of `formattedWord`, but this issue is purely about improving the interface, not meddling with the computed data. A better solution would be to add the spaces when you update the text of `currentWordLabel`.

To properly set the text of `currentWordLabel`, you can use a Swift method called `joined(separator:)` that operates on an array of strings. This function concatenates the collection of strings into one string, separated by a given value. Here's an example:

```
let cast = ["Vivien", "Marlon", "Kim", "Karl"]
let list = cast.joined(separator: ", ")
print(list) // "Vivien, Marlon, Kim, Karl"
```

How can you imagine using the `joined(separator:)` method to set `currentWordLabel`? Start by converting the array of characters in `formattedWord` into an array of strings. Use a for loop to store each of the newly-created strings into a `[String]` array. Then you can call the `joined(separator:)` method to join the new collection together, separated by blank spaces.

```
func updateUI() {
    var letters = [String]()
    for letter in currentGame.formattedWord {
        letters.append(String(letter))
    }
    let wordWithSpacing = letters.joined(separator: " ")
    correctWordLabel.text = wordWithSpacing
    scoreLabel.text = "Wins: \(totalWins), Losses:
    \(totalLosses)"
    treeImageView.image = UIImage(named: "Tree
```

```
        \(currentGame.incorrectMovesRemaining)")
    }
```

Build and run your application to see a clear break between the letters and the underscores.

PART FIVE-HANDLE A WIN OR LOSS

Apple Pie is starting to feel like a real game. The round is progressing along with each button pressed, but there are two obvious issues. The first is that `incorrectMovesRemaining` can go below 0, and the second is that the player can't win a game, even if they guess all letters correctly.

When `incorrectMovesRemaining` reaches 0, `totalLosses` should be incremented, and a new round should begin. It would be nice to have a single method that checks the game state to see if a win or loss has occurred, and if so, update `totalWins` and `totalLosses`. Create a method called `updateGameState` that will perform this work, and call it after each button press instead of calling `updateUI()`.

```
@IBAction func buttonTapped(_ sender: UIButton) {
    sender.isEnabled = false
    let letterString = sender.title(for: .normal)!
    let letter = Character(letterString.lowercased())
    currentGame.playerGuessed(letter: letter)
    updateGameState()
}

func updateGameState() {

    }
```

How do you determine if a game is won, lost, or if the player should continue playing? A game is lost if `incorrectMovesRemaining` reaches 0. When it does, increment `totalLosses`. You can determine that a game has been won if the player has not yet

lost, and if the current game's `word` property is equal to the `formattedWord` (formattedWord won't have any underscore if every letter has been successfully guessed). When that happens, increment `totalWins`. If a game has not been won or lost yet, then the player should be allowed to continue guessing, and the interface should be updated.

```
func updateGameState() {
  if currentGame.incorrectMovesRemaining == 0 {
    totalLosses += 1
  } else if currentGame.word == currentGame.formattedWord {
    totalWins += 1
  } else {
    updateUI()
  }
}
```

Build and run your application. Is the game functioning properly? It's really close, but a new round does not begin after a win or loss. Whenever `totalWins` or `totalLosses` changes, a new round can be started, so this is a great time to add `didSet` property observers to `totalWins` and `totalLosses`.

```
var totalWins = 0 {
    didSet {
        newRound()
    }
}
var totalLosses = 0 {
    didSet {
        newRound()
    }
}
```

Now try running your application, verifying that both wins and losses are tallied up accordingly, and that a new round begins.

Re-enable Buttons and Fix Crash

It looks like you're approaching the finish line! You're successfully able to complete a round of Apple Pie, but a new round doesn't re-enable the letter buttons. Also, if you try to start a new round and there's no more words to choose from, the game will crash.

The `newRound` logic needs to be a little smarter. If `listOfWords` isn't empty, then you should perform the same work that you did previously, but also re-enable all of the buttons. If there are no more words to play with, disable all of the buttons so that the player cannot continue playing the game.

```
func newRound() {
    if !listOfWords.isEmpty {
        let newWord = listOfWords.removeFirst()
        currentGame = Game(word: newWord,
        incorrectMovesRemaining: incorrectMovesAllowed,
        guessedLetters: [])
        enableLetterButtons(true)
        updateUI()
    } else {
        enableLetterButtons(false)
    }
}
```

The `enableLetterButtons(_:)` method is fairly straightforward. It takes a `Bool` as an argument, and it uses the parameter to enable or disable the collection of buttons by looping through them.

```
func enableLetterButtons(_ enable: Bool) {
  for button in letterButtons {
    button.isEnabled = enable
  }
}
```

If you build and run the application, you should be able to get through all of the Apple Pie rounds until the end is reached and the buttons are disabled.

Wrapup

Congratulations on building your first mobile game with Swift!

By successfully completing Apple Pie, you've demonstrated an understanding of the building blocks of the Swift language. This project isn't easy, and you should be proud of what you've accomplished. If you struggled to follow along with any of the steps, set aside some time to rebuild Apple Pie on your own, without this guide. A second run-through will point out areas that you may not have understood—and you can look to the guide and earlier lessons to reinforce your knowledge.

Stretch Goals

If you'd like to continue working on Apple Pie, go ahead and try adding some features to the game. Here are a few ideas to play with. You can build most of these features using your existing knowledge of Swift, but a few may require you to use the Xcode documentation. Good luck!

Challenge yourself by adding these features to Apple Pie:

- Learn about the `map` method, and use it in place of the loop that converts the array of characters to an array of strings in `updateUI()`.
- Add a scoring feature that awards points for each correct guess and additional points for each successful word completion.
- Allow multiple players to play, switching turns after each incorrect guess.
- Allow the player to guess the full word using the keyboard instead of guessing one letter at a time using the interface buttons.
- Support letters with special characters. For example, the E button could check for "e" and "é" within a word.

Summary

You made it! In this unit, you learned about many new programming concepts and how the Swift language puts them to use. You've learned the basics of `UIKit`, the foundation of iOS development. You've learned how to display simple information with views, and respond to user input with a variety of controls. You also learned about the powerful tools in Xcode that enable you to build slick-looking interfaces that fit within the iOS environment.

After completing this unit, you should be ready to build much more interesting apps. Find out what's next.