CSC 581: Mobile App Development

Spring 2019

Unit 2: Structs & layout

- Strings
- functions
  external & internal parameter names, return types
- structs
  public vs. private vs. private(set) fields, init, methods
- layout with Stack Views
- Model-View-Controller pattern
- Optionals
- UI toolbox (so far):
  - ✓ UILabel, UIImageView
  - ✓ UISwitch, UIButton, Tap Gesture Recognizer

Strings

- Swift Strings have many similarities to Python/Java strings

```
var str1 = "foobar"
var str2 = "State \"your name.\""
var cute = "👍😃🌈"                // can contain Unicode symbols

str2 = str1 + str1
str2 += str1

var single = """
    line 1 \
    line2
    """

var multi= """
    line 1
    line2
    """
```

- a single character is inferred to be a String, but can be declared to be a char

```
var ch1 = "q"                  // type(of: ch1) → String.Type

var ch2: Character = "q"       // type(of: ch2) → Character.Type
```

String interpolations & comparisons

- String interpolations make it easy to embed values in Strings

```
var a = 7
var b = 2
print("The sum of \(a) and \(b) is \(a+b)")
```

- comparison operators can be applied to Strings

```
if str1 == str2 {
    print("same")
}

if str1 != "foobar" {
    print("not foobar")
    if str1 < "bar" {
      print("before bar")
    }
    else if str1 >= "foo" {
        print("foo or after")
    }
    else {
        print("between bar and foo")
    }
}
```

3

String properties & methods

```
var str = "Creighton"

str.count             → 9

str.lowercased()      → "creighton"

str.uppercased()      → "CREIGHTON"

str.contains("igh")   → true

str.hasPrefix("Cr")   → true

str.hasSuffix("ton")  → true

for ch in str {                        // iterates over the characters
    print(ch)
}

for index in str.indices {             // iterates over indices, then
    print(str[index])                  // prints each character
}
```

4

Functions

- general form of a Swift function

```
func functionName(param1: Type1, param2: Type2, …) -> ReturnType {
    // BODY OF THE FUNCTION
}
```

e.g.

```
func increment(num: Int) -> Int {
    return num+1
}

func greet(name: String) -> String {
    return "Hello, " + name + "!"
}

func compare(num1: Int, num2: Int) {
    if (num1 == num2) {
        print("same")
    }
    else {
        print("different")
    }
}
```

5

External vs. internal names

- `low` and `high` are reasonable names within the function, but not so much in the call
- better:

```
var value = sum(from: 1, to: 10)
```

- but then the function looks strange

```
func sum(from: Int, to: Int) -> Int {
    var total = 0;
    for i in from...to {
        total += i
    }
    return total
}
```

- solution: specify two names – an external one for the call, an internal one for the function

```
func sum(from low: Int, to high : Int) -> Int {
    var total = 0;
    for i in low...high {
        total += i
    }
    return total
}
```

6

Swift parameter style

- if different external & internal names read better, specify both

```
display(message: String, in: UILabel)

func display(message msg: String, in label: UILabel) {
    ...
}
```

- if same name works externally and internally, specify a single name (internal & external)

```
func display(message: String, in label: UILabel) { ... }
```

- if you really don't want to specify an external name, can use an underscore (but not considered good style)

```
func increment(_ num: Int) -> Int {
    return num+1
}

x = increment(x)
```

7

Default parameters

- you can specify default values for parameters

```
func rollDie(sides: Int = 6) -> Int {
    return Int.random(in: 1...sides)
}

print(rollDie())              // rolls a 6-sided die,
                              // same as print(rollDie(sides: 6))

print(rollDie(sides: 8))      // rolls an 8-sided die
```

- if a function has multiple parameters, default parameters must be at the end

```
func drawSquare(size side: Int, at x: Int = 0, and y: Int = 0) {
    // draws a square side pixels big, centered at (x, y)
}

drawSquare(side: 50)    // draws a square with 50-pixel sides at (0,0)
                        // same as drawSquare(side: 50, at: 0, and: 0)

drawSquare(side: 50, at: 30)  // draws 50-pixel square at (30, 0)
                        // same as drawSquare(side: 50, at: 30, and: 0)

drawSquare(side: 50, at: 30, and: 40)
                        // draws a 50-pixel square at (30, 40)
```

8

In-class exercise

1. Create a playground, enter the `rollDie` function, and verify that it works

```
func rollDie(sides: Int = 6) -> Int {
    return Int.random(in: 1...sides)
}
```

2. define a function named `rollDice` that rolls a specified number of dice

```
rollDice(thisMany: 2, withSides: 6)  // returns the sum of 2 6-sided dice

rollDice(thisMany: 4, withSides: 8)  // returns the sum of 4 8-sided dice


rollDice()                 // specify defaults for dice (2) and sides (6)
```

3. define a function named `rollPercentage` that repeatedly rolls a pair of 6-sided dice and returns the percentage of a specified total

```
rollPercentage(of: 7, outOf: 10_000)    // returns the percentage of 7's
                                        // out of 10,000 rolls

rollPercentage(of: 7)                   // specify a default for the number
                                        // of rolls (10,000)
```

9

Defining types

so far, we have manipulated primitive data values

- `struct` and `class` are two ways to define a new type
- both can:
  - ✓ encapsulate fields/properties and methods/functions
  - ✓ hide fields/methods by making them private
  - ✓ fields/methods can be class-wide by declaring them `static`
  - ✓ utilize `init` to initialize fields (i.e., a constructor)
- significant differences:
  - ✓ structs are copied whenever you assign or pass as parameters; class objects are shared references (like in Java)
  - ✓ classes can utilize inheritance; structs cannot

common practice:

- structs are used most often in Swift, especially for simpler objects
- classes are used for more complex, reusable data types
  - ✓ classes are the only option if inheritance/polymorphism is desired

MORE ON CLASSES LATER

10

Name struct

a struct groups related data values into a single object

- specify the field values when creating a struct object

by default: no information hiding or protection

- e.g., can access and change the fields directly
- we can change this (LATER)

```
struct Name {
    var first: String
    var middle: Character
    var last: String
}


var me = Name(first: "David", middle: "W", last: "Reed")


print("\(me.last), \(me.first) \(me.last)")
                                    → "Reed, David Reed"

me.first = "Dave"

print("\(me.last), \(me.first) \(me.last)")
                                    → "Reed, Dave Reed"
```

11

Struct methods

structs can have methods

- e.g., add a comparison method for names (similar to Java `compareTo`)
  - returns -1 if <
  - returns 0 if ==
  - returns 1 if >

note: use of external and internal parameter names

also: the optional `self.` prefix for fields is similar to similar to `this.` in Java

```
struct Name {
    var first: String
    var middle: Character
    var last: String

    func compare(with other: Name) -> Int {
        if self.last == other.last &&
            self.first == other.first &&
            self.middle == other.middle {
            return 0
        } else if self.last < other.last ||
            (self.last == other.last &&
               self.first < other.first) ||
            (self.last == other.last &&
               self.first == other.first &&
               self.middle < other.middle) {
            return -1
        }
        else {
            return 1
        }
    }
}

var me = Name(first: "David", middle: "W", last: "Reed")
var him = Name(first: "Mark", middle: "V", last: "Reedy")

me.compare(with: him)      → -1
me.compare(with: me)       → 0
him.compare(with: me)      → 1
```

12

Private fields

in Java, we always avoided public fields

- they violate the spirit of classes, since they allow accessing/modifying fields without calling methods

we can similarly declare Swift fields to be private

- need to provide an `init` method (similar to Java constructor)
- might need to define accessor methods

```
struct Name {
    private var first: String
    private var middle: Character
    private var last: String

    init(first: String, middle: Character, last: String) {
        self.first = first
        self.middle = middle
        self.lat = last
    }

    func compare(with other: Name) -> Int {
        // SAME AS PREVIOUS SLIDE
    }
}

var me = Name(first: "David", middle: "W", last: "Reed")
var him= Name(first: "Mark", middle: "V", last: "Reedy")

me.compare(with: him)      → -1

me.first                   → NOT ALLOWED
```

13

Private(set) fields

a simpler alternative is available

- can declare a field `private(set)`
- this allows you to set the field when you create the struct
- subsequently, can view the field, but not change it
- no longer need the `init`
- also don't need accessor methods

```
struct Name {
    private(set) var first: String
    private(set) var middle: Character
    private(set) var last: String

    func compare(with other: Name) -> Int {
        // SAME AS PREVIOUS SLIDE
    }
}

var me = Name(first: "David", middle: "W", last: "Reed")
var him= Name(first: "Mark", middle: "V", last: "Reedy")

me.compare(with: him)      → -1

me.first                   → "David"

me.first = "Dave"          → NOT ALLOWED
```

14

Mutating methods

if a method is going to change any fields, must declare it as `mutating`

- methods are allowed to change `private(set)` fields
- note use of external and internal names for parameter

```
struct Name {
    private(set) var first: String
    private(set) var middle: Character
    private(set) var last: String

    func compare(with other: Name) -> Int {
        // SAME AS PREVIOUS SLIDE
    }

    mutating func changeLastName(to newName: String) {
        self.last = newName
    }
}

var me = Name(first: "David", middle: "W", last: "Reed")

me.last                    → "Reed"

me.changeLastName(to: "Pseudonym")

me.last                    → "Pseudonym"

me.last = "Reed"           → NOT ALLOWED
```

15

Die struct

to simulate a 6-sided die

- `private(set)` fields for # of sides & rolls
- mutating method for returning a random roll (+ update # rolls)
- note: `random` is a static method of the `Int` class
- takes a range as input, selects random number from that range

```
struct Die {
    private(set) var numSides: Int
    private(set) var numRolls: Int

    mutating func roll() -> Int {
        self.numRolls += 1
        return Int.random(in: 1...numSides)
    }
}

var d1 = Die(numSides: 6, numRolls: 0)

for i in 1...10 {
    print(d1.roll())       → DISPLAYS 10 ROLLS
}

d1.numRolls                → 10

d1.numSides                → 6

d1.numSides = 8            → NOT ALLOWED
```

16

Default fields

can provide default values for fields

- e.g., 6-sided die with 0 rolls
- can override the defaults by specifying all field values

would really like to have numSides changeable, but numRolls always start at 0

- unfortunately, default fields are all or nothing

```
struct Die {
    private(set) var numSides = 6
    private(set) var numRolls = 0

    mutating func roll() -> Int {
        self.numRolls += 1
        return Int.random(in: 1...numSides)
    }
}

var d1 = Die()

for i in 1...10 {
    print(d1.roll())        → DISPLAYS 10 ROLLS
}

var d2 = Die(numSides: 8, numRolls: 0)

d2.numSides                 → 8
```

17

Defaults via init

can have partial defaults using `init`

- have `init` with one parameter, with default value
- `numRolls` is automatically 0

can create a Die using the default # of sides or specify the desired #

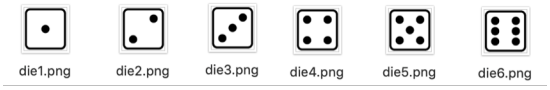
```
struct Die {
    private(set) var numSides: Int
    private(set) var numRolls: Int

    init(withSides sides: Int = 6) {
        self.numSides = sides
        self.numRolls = 0
    }

    mutating func roll() -> Int {
        self.numRolls += 1
        return Int.random(in: 1...numSides)
    }
}

var d1 = Die()

var d2 = Die(withSides: 8)
```

18

Dice app

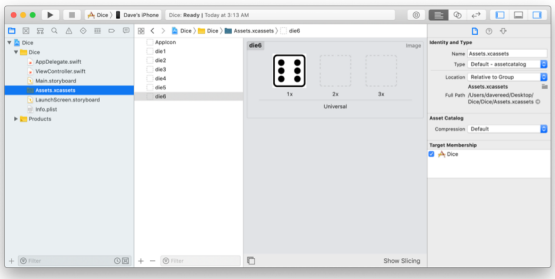
suppose we want to create an app that simulates rolling dice

- images of dice, change at random when clicked



in order to utilize images in an app,

- open `Assets.xcassets` in the Edit Area
- drag the desired images into the window

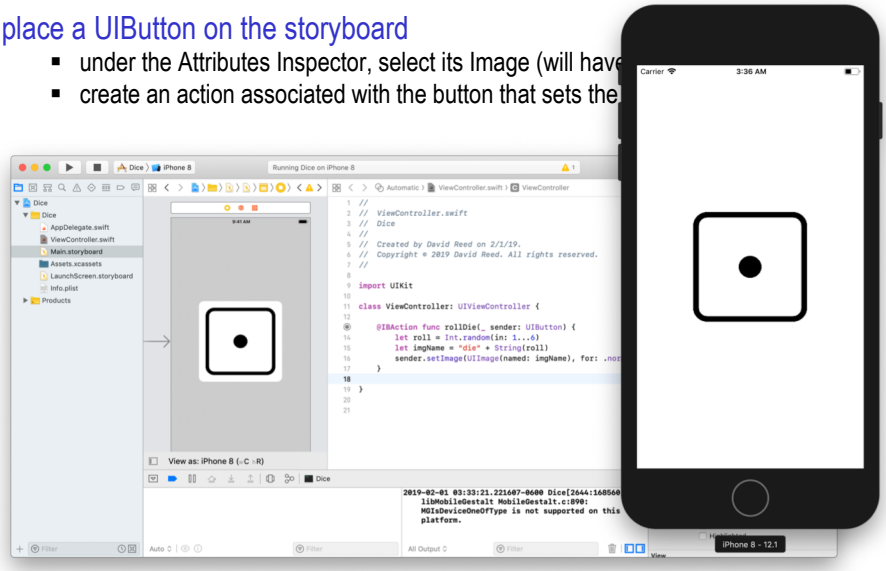


19

Single die

place a UIButton on the storyboard

- under the Attributes Inspector, select its Image (will have
- create an action associated with the button that sets the



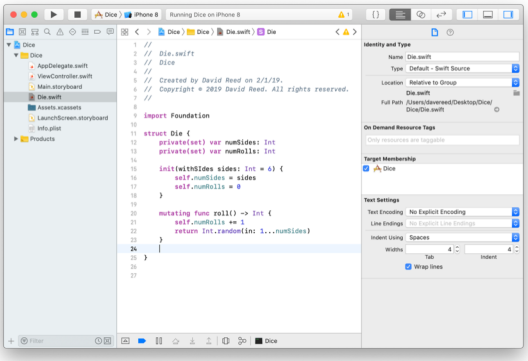
20

Adding a model

if things got more complicated (e.g., another die image, label with # rolls),

- the Controller would start getting messy
- better approach: place the logic for a die in a separate file (i.e., the Model)

- select File → New → File
- select Swift from from the options, specify name (e.g., Die.swift)
- enter the definition of the Die struct in this file

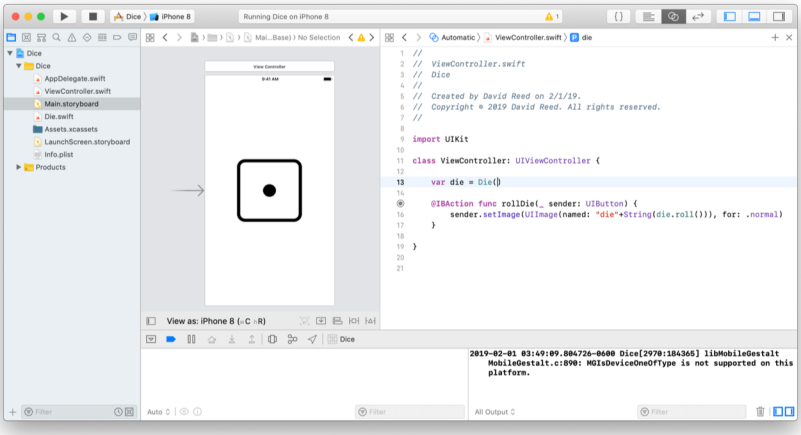


21

Using the model

can now have a field of type Die in the ViewController

- within the roll method can call methods of the Die field
- add a constraint so that the aspect ratio of the die is square



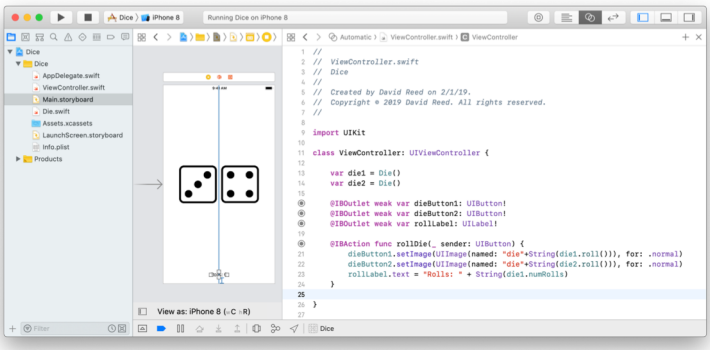
22

Two dice

can copy-paste the die button and arrange the two dice side-by-side

- want to make it so clicking either die rolls them both

- create outlets for each die button so that they can be referenced in the code
- update the rollDie method so that it changes both dice
  note: both buttons can be tied to the same action
- also create a label to display the # of rolls

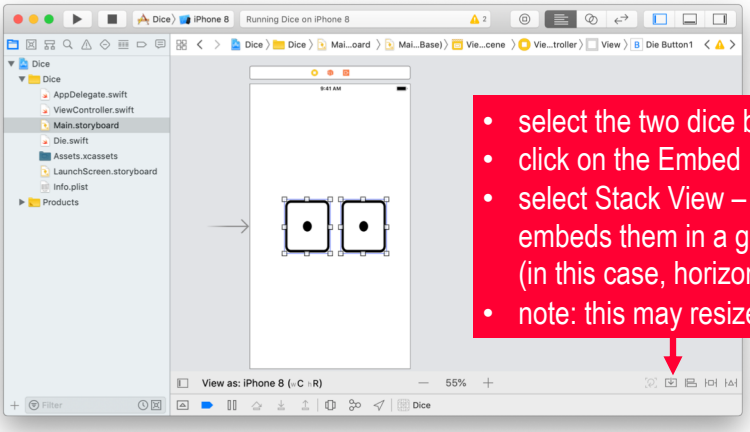


23

Dice layout

would like to ensure that the pair of dice are always centered

- could try to do it with alignment and layout constraints – TRICKY
- better solution, group the two dice together, then center the grouping

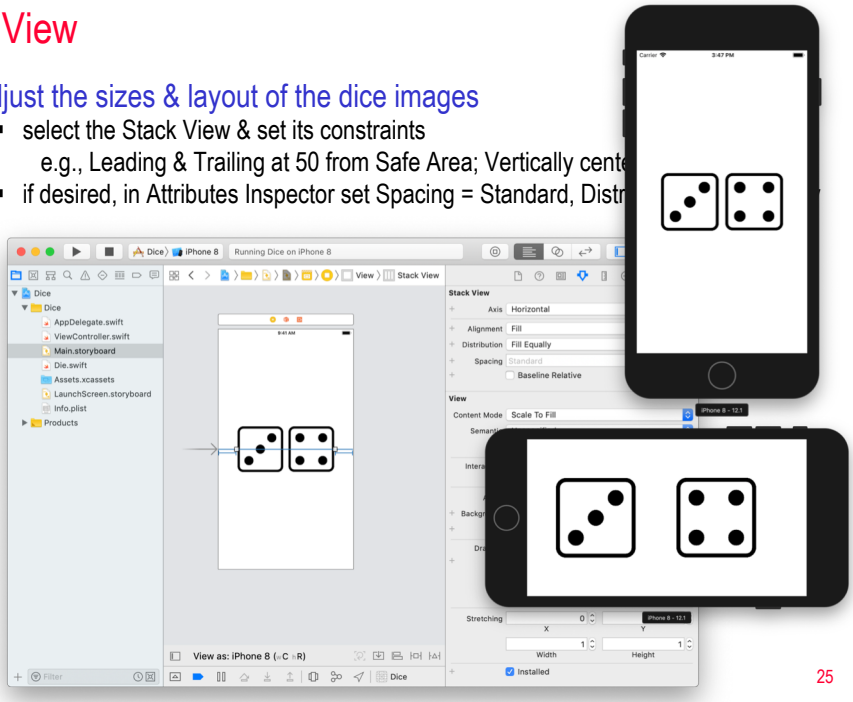


24

Stack View

to adjust the sizes & layout of the dice images

- select the Stack View & set its constraints
  e.g., Leading & Trailing at 50 from Safe Area; Vertically cente
- if desired, in Attributes Inspector set Spacing = Standard, Distr



25

Alternative design

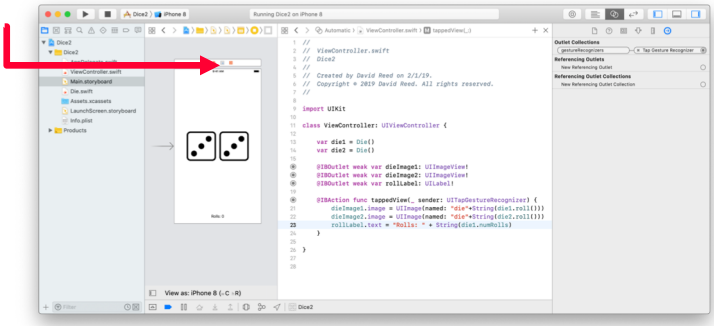
since the dice are buttons, you have to tap on one of the die images to roll

- what if we wanted to be able to tap anywhere on the screen?

could instead use a `UIImageView` for each die image

add a Tap Gesture Recognizer to the view

- drag Tap Gesture Recognizer from the Library to the main View
- this adds an icon above the storyboard
- control-drag from that icon to the ViewController to create an Action



26

Model-View-Controller pattern

in both versions of the dice app,

- the logic of the app is defined in the Die struct — MODEL
- the user interface is defined in the storyboard — VIEW
- the ViewController connects interface with the logic — CONTROLLER

we want to strive for this division of labor

- encourages modular design, supports code reuse, keeps the controller simple
- general rule of thumb -- the VewiController class can have:
  fields that are (1) structs/objects defined in separate files or (2) IBOutlets
  methods that are IBActions and primarily perform intermediary steps
  - ✓ access attributes of the IBOutlets (e.g., label text or background color)
  - ✓ call methods of struct/class objects
  - ✓ display the results of method calls in IBOutlets

27

HW 2.1

reimplement your Executive Decision Maker app using a struct for the model

- behavior should be the same, but the logic is defined in a separate struct
- should significantly simplify the ViewController

```
struct Decider {
    private let options: [String]

    init(between options: [String]) {
        self.options = options
    }

    func getAnswer() -> String {
        return self.options[Int.random(in: 0..<options.count)]
    }
}

var decisionMaker = Decider(between: ["YES", "NO", "MAYBE"])

decisionMaker.getAnswer()          → either "YES", "NO", or "MAYBE"

var dinnerPicker = Decider(between: ["pizza", "chinese", "tacos", "salad"])

dinnerPicker.getAnswer()           → one of the dinner options
```

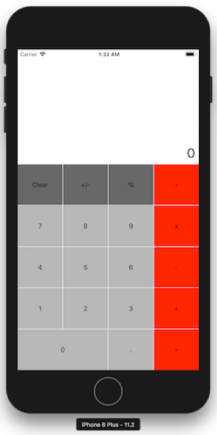
28

HW 2.2

follow the tutorial in Unit 2 to create a calculator interface

- define the `CalculatorModel` struct to serve as the model
- update the `ViewController` to connect the View & the Model

```
struct CalculatorModel {
    private(set) var currentValue = 0.0
    private var lastOp = "+"

    mutating func apply(op: String, with number: Double) {
        if self.lastOp == "=" {
            self.currentValue = number
        }
        else if self.lastOp == "+" {
            self.currentValue += number
        }
        else if self.lastOp == "-" {
            self.currentValue -= number
        }
        else if self.lastOp == "*" || self.lastOp == "x" {
            self.currentValue *= number
        }
        else if self.lastOp == "/" || self.lastOp == "÷" {
            self.currentValue /= number
        }
        self.lastOp = op
    }


    mutating func clear() {
        self.currentValue = 0.0;
        self.lastOp = "="
    }
}
```



29

Optionals

for HW 2.,2, you will need to know about Optionals

Swift is extremely picky about types

- recall that you can create an Int value out of a String

```
var x = Int("23")
```

- but what happens if the String is not just digits?

```
var y = Int("foo")
```

for values that could be undefined, Swift uses an Optional Type

- Optional is a wrapper (think Integer in Java), either wraps the value or `nil`

```
var x = Int("23")    → Optional(23)

var y = Int("foo")   → nil
```

30

Unwrapping Optionals with !

the simplest way to unwrap an Optional is using !

```
var x = Int("23")       → Optional(23)

var y = Int("23")!      → 23
```

this is dangerous in general

- unwrapping a nil value causes an error
- only over do it if you are sure you have a wrapped value

- or, can check before unwrapping

```
let str = ???

var value = Int(str)
if value != nil {
    print(value!)
}
```

31

! vs. if-let

Swift provides a control statement to do this more cleanly: if-let

- if the assigned value is not nil, it is unwrapped and the test succeeds

```
if let value = Int(str) {
    print(value)
}

or

if let value = Int(str) {
    print(value)
}
else {
    print("\(str) does not represent an integer")
}
```

Optionals are common in Swift, whenever an undefined value is possible

- in particular, many UI elements have fields/methods that use Optionals

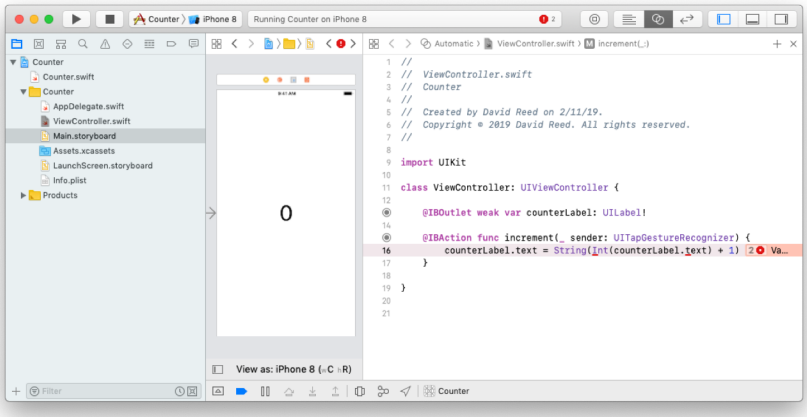
32

Optionals in UIKit

consider a simple counter app

- have a counter (initially 0) centered in a label
- use a Tap Gesture Recognizer to increment the counter when tapped

- can access `label.text` to get the text displayed in the label
- can then convert that to an Int, add 1, and display the incremented value



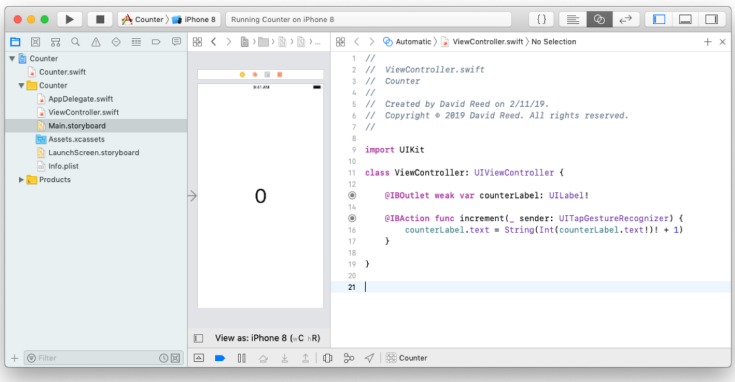
33

Unwrapping the Optionals

there are two problems:

1. `counterLabel.text` returns an Optional (since the text may not be defined)
2. `Int` also returns an Optional (as previously discussed)

- since we are confident that the label will always contain a number, using ! is OK
- it must be applied twice, one for `counterLabel.text` and one for `Int`

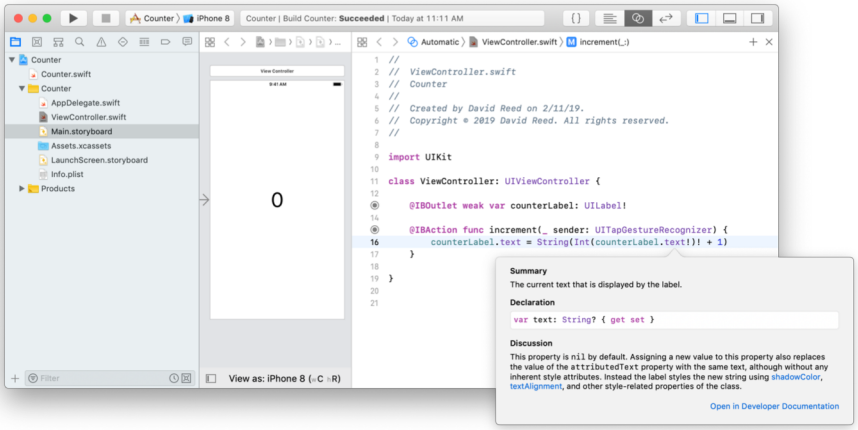


34

Inspecting variable info

recall: can get info on a variable in Xcode by Option-clicking

- Option-clicking on `counterLabel.text` shows that its type is `String?`
- the `?` at the end denotes Optional (e.g., `Int` returns a value of type `Int?`)

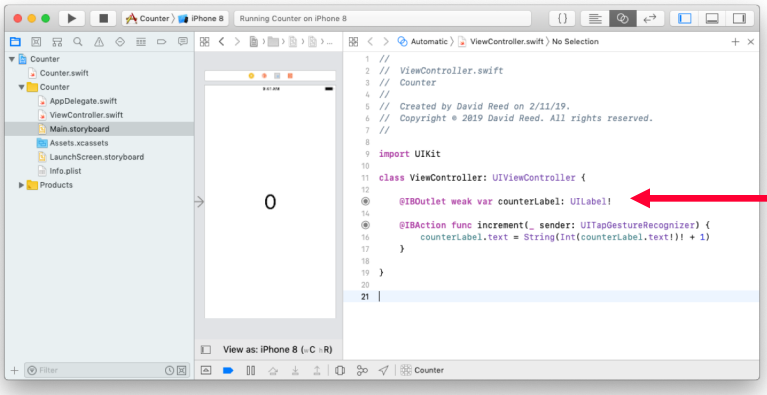


35

Unwrapping Outlets?

in case you are curious:

- the ! at the end of Outlet types is related but different
- it warns that the app will crash if the Outlet is not connected to a UI element



36