CSC 581: Mobile App Development

Spring 2019

Unit 2: More Swift & UIKit

- computed properties/fields
- classes & inheritance
- collections:
  - arrays, dictionaries
- UIKit
  - UIView subclasses: UILabel, UIImageView, …
  - UIControl subclasses: UIButton, UISwitch, …
- Appicon

RouletteWheel struct

consider a struct for simulating a roulette wheel

- American roulette wheels contain 00,0,1…34
- to differentiate between 0 and 00, spin method will return a String

- `numSpins` field seems redundant – the private die field already keeps track of the # of rolls

```
struct RouletteWheel {
    private var die = Die(withSides: 36)
    private(set) var numSpins = 0

    mutating func spin() -> String {
        let dieRoll = self.die.roll()
        self.numSpins += 1

        if dieRoll == 36 {
            return "00"
        }
        else if dieRoll == 35 {
            return "0"
        }
        else {
            return String(dieRoll)
        }
    }
}

var wheel = RouletteWheel()
for _ in 1...10 {
    print(wheel.spin())
}

print(wheel.numSpins)
```

Avoiding redundant counters

we could get rid of the `numSpins` field

- instead have a method for determining the number of spins
- this avoids the redundant counter, but many feel that a field is more natural

```
struct RouletteWheel {
    private var die = Die(withSides: 36)

    mutating func spin() -> String {
        let dieRoll = self.die.roll()
        if dieRoll == 36 {
            return "00"
        }
        else if dieRoll == 35 {
            return "0"
        }
        else {
            return String(dieRoll)
        }
    }

    func numSpins() -> Int {
        return die.numRolls
    }
}

var wheel = RouletteWheel()
for _ in 1...10 {
    print(wheel.spin())
}

print(wheel.numSpins())
```

3

Computed properties/fields

it is possible to make the method look like a field

a *computed property/field* is declared and accessed like a field, but its value is computed whenever it is accessed

- after the field declaration, specify code in { } that computes & returns the value
- don't specify protection, but implicitly `private(set)`
- essentially, it acts like an accessor method but looks like a `private(set)` field

```
struct RouletteWheel {
    private var die = Die(withSides: 36)
    var numSpins: Int {
        return die.numRolls
    }

    mutating func spin() -> String {
        let dieRoll = self.die.roll()
        if dieRoll == 36 {
            return "00"
        }
        else if dieRoll == 35 {
            return "0"
        }
        else {
            return String(dieRoll)
        }
    }
}

var wheel = RouletteWheel()
for _ in 1...10 {
    print(wheel.spin())
}

print(wheel.numSpins)
```

4

Classes

in addition to structs, Swift supports classes for defining new types

- can define just like in Java
- private fields
- init (constructor)
- accessor & mutator methods

note: unlike structs, you do not declare methods mutating

classes are typically more work & less natural than structs

- used in Swift mainly when you want to utilize inheritance

```
class Die {
    private var numRolls: Int
    private var numSides: Int

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

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

    func numberOfRolls() -> Int {
        return self.numRolls
    }

    func numberOfSides() -> Int {
        return self.numSides
    }
}

var d = Die()
for _ in 1...10 {
    print(d.roll())
}

print(d.numberOfRolls())
```

5

Inheritance

to inherit from an existing class:

- specify the parent class in the header
- all fields & methods of the parent class are inherited
- call `super.init` AT THE END of the `init`
- if overriding a method, must declare as `override`

- as in Java, private fields from parent class are not accessible (can allow access by declaring as `internal`)

```
class ColoredDie: Die {
    private var color: String

    init(withColor color: String,
              withSides sides: Int = 6) {
        self.color = color
        super.init(withSides: sides)
    }

    override func roll() -> Int {
        if self.color == "red" {
            return 1
        }
        else {
            return super.roll()
        }
    }

    func dieColor() -> String {
        return self.color
    }
}

var d = ColoredDie(withColor: "white")
print(d.roll())
print(d.dieColor())

var poly: Die = ColoredDie(withColor: "red")
print(poly.roll())
```

6

Values vs. references

structs are value types (as in Python)

- when you assign a *struct* object to a variable or pass it as a parameter, a *copy* of that object is assigned

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

me.first = "Dave"

me.first        →       "Dave"
copy.first      →       "David"
```

classes are reference types (as in Java)

- when you assign a *class* object to a variable or pass it as a parameter, a *reference* to that object is assigned

```
var die1 = ColoredDie(withColor: "green")
var die2 = die1

die1.roll()

die1.numberOfRolls()   → 1
die2.numberOfRolls()   → 1
```

7

Collections

Swift provides three different collection structs for storing values

- array: an indexed list
- dictionary: a list of key-value pairs
- set: a non-index list of unique items (*we will ignore for now*)

note: all three are homogeneous (items must all be of the same type)

arrays

```
var example: Array<String>        // declares an array of Strings
var example: [String]             // same thing, but cleaner

var empty: Array<String>()        // declares & inits to be empty
var empty: [String]()             // same thing
var empty: [String] = []          // same thing

var words = ["foo", "bar"]        // declares & initializes
```

8

Array properties & methods

```
var words = ["foo", "bar"]

words.count                    → 2

words.append("biz")
words                          → ["foo", "bar", "biz"]

words += ["baz", "boo"]
words                          → ["foo", "bar", "biz", "baz", "boo"]

words[0]                       → "foo"
words[words.count-1]           → "boo"

for str in words {             // displays all five words
    print(str)
}

for i in 0..<words.count {     // also displays all five words
    print(words[i])
}

words.remove(at: 2)
words                          → ["foo", "bar", "baz", "boo"]

words.insert("zoo", at: 1)
words                          → ["foo", "zoo", "bar", "baz", "boo"]
```

9

Decision Maker

for HW2, you separated the Model & Controller in your Decision Maker app

- could have taken the HW1 code as is and placed it in a struct with no fields

this is not ideal

- hardcoded to those three answers
- adding more answers is tedious
- impossible to have different Deciders with different options

better solution:

- when a Decider is created, pass in a list of the options
- store that list as a private field
- can then randomly choose an option from the list

```
struct Decider {
    func getAnswer() -> String {
        let randomNum = Int.random(in: 1...3)
        if randomNum == 1 {
            return "YES"
        }
        else if randomNum == 2 {
            return "NO"
        }
        else {
            return "MAYBE"
        }
    }
}


var chooser = Decider()


print(chooser.getAnswer())
```

10

Better Decider

```
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 chooser = Decider(between: ["yes", "no", "maybe"])

print(chooser.getAnswer())


var dinnerOptions = ["Pizza", "Sushi", "Burger",
                     "Tacos", "Pancakes", "Salad"]

var foodPicker = Decider(between: dinnerOptions)

print(foodPicker.getAnswer())
```

11

Dictionaries (a.k.a. Maps)

suppose you wanted to store people and their ages

- e.g., Pat is 18, Chris is 20, …
- once stored, could look up a person to get their age

a dictionary is a collection of key:value pairs

```
var ages = ["Pat": 18, "Chris": 20, "Kelly": 19]
```

- can add pairs simply by assigning a value to a key

```
ages["Jan"] = 21   // adds "Jan": 21 to the dictionary
```

- can access the value by indexing with the key
- note: indexing returns an Optional (`nil` if key is not stored in the dictionary)

```
ages["Chris"]      → Optional(20)

if let age = ages["Chris"] {
    print(age)
}
```

12

Dictionary properties & methods

```
var example = Dictionary<String, Int>  // declares Dictionary of
var example = [String: Int]            //   String:Int pairs

var example: Dictionary<String: Int>() // declares and initializes as
var example: [String: Int]()           //   empty
var example: [String: Int] = [:]       //


var ages = ["Pat": 18, "Chris": 20, "Kelly": 19]

for (name, age) in ages {
    print("\(name) is \(age) years old.")
}

ages["Pat"] = 19                       // updates "Pat"'s value

ages["Sean"] = 22                      // adds "Sean":22 to end

for name in ages.keys {                // adds 1 to each age
    if let age = ages[name] {
        ages[name] = age+1
    }
}

print(ages.values.max()!)              // prints 22
```

13

Copy care

arrays and dictionaries, like other built-in Swift types, are structs

- that means that assigning an array/dictionary or passing it as a parameter to a function makes a copy

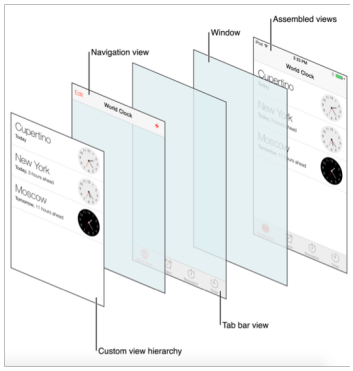
```
var nums = [3, 6, 1, 7]

var copy = nums

nums += [10, 20, 30]

nums.count                → 7

copy.count                → 4


func zeroOut(numList: [Int]) {
    for index in 0..<numList.count {
        numList[index] = 0
    }
}

zeroOut(numList: nums)

nums                      → [3, 6, 1, 7, 10, 20, 30]
```

14

UIKit

the UIKit is a code framework for building mobile apps

- the foundational class for all visual elements in the UIKit is the UIView (or just *view*)
- a view is a clear rectangular shape that is customized to display things on the screen
- UIView subclasses:
  - ✓ UILabel – for displaying text
  - ✓ UIImageView – for displaying images
  - ✓ UIScrollView – for scrollable content
- most app screens consist of many views



15

UIView subclasses (so far)

✓ UILabel

- for displaying text
- access/change label: `labelOutlet.text`
- access/change text color: `labelOutlet.textColor`
- access/change background color: `labelOutlet.backgroundColor`

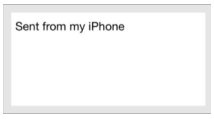
The volume of the ringer and alerts can be adjusted using the volume buttons.

✓ UIImageView

- for displaying images
- access/change image: `imgOutlet.image`

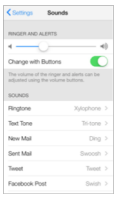
16

Other UIView subclasses

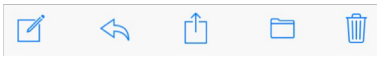


- ✓ UITextView
  - for entering and displaying multiline text

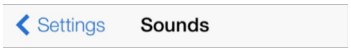
- ✓ UIScrollView
  - for scrollable content



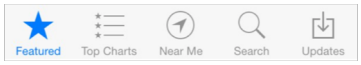
- ✓ UITableView
  - for displaying data in a scrollable column of rows and sections
- ✓ UIToolbar
  - for displaying a toolbar of buttons
  - usually at the bottom of the screen



- ✓ UINavigationBar
  - for displaying main navigational controls
  - usually at the top of the screen



- ✓ UITabBar
  - for displaying options for actions



17

UIControl subclasses (so far)

- ✓ UIButton
  - can be clicked

  - change label text: `buttonOutlet.setTitle("???", for: .normal)`
  - change label color: `buttonOutlet.setTitleColor(???, for: .normal)`
  - change background color: `buttonOutlet.backgroundColor`
  - change image: `buttonOutput.setImage(named: ???, for: .normal)`

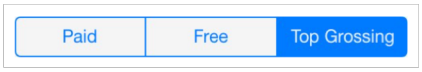
- ✓ UISwitch
  - can turn on or off

  - determine its position: `switchOutlet.isOn`

18

Other UIControl subclasses

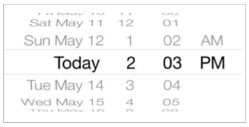
- ✓ UISegmentedControl
  - can select from visible options



- ✓ UITextField
  - can enter text

- ✓ UISlider
  - can select from a range

- ✓ UIDatePicker
  - can select a date



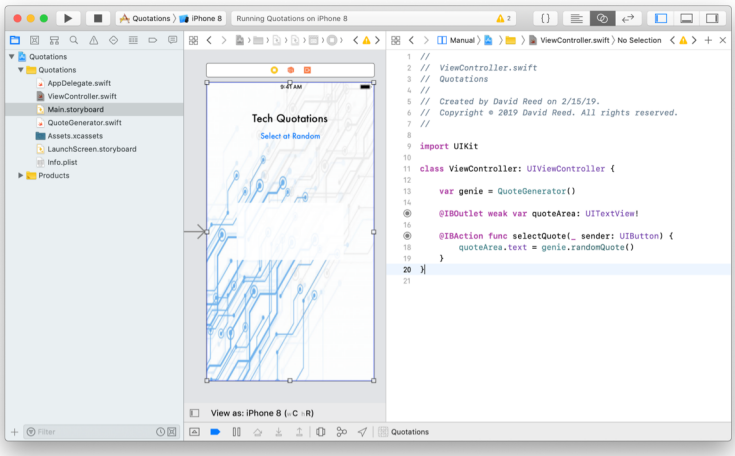
- ✓ UIStepper
  - can increment/decrement a counter

19

Example: text view

UITextView can display multiline text in an editable, scrollable rectangle

- here, a random quotation is selected and displayed in the text view
- a UIImageView provides the background
- setting the UITextView's alpha to 0.9 make it semi-transparent

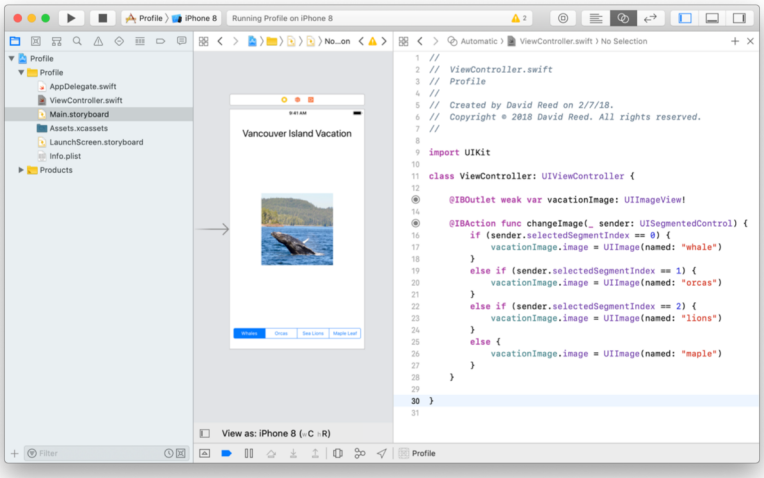


20

Example: segmented control

UISegmentedControl provides a series of clickable segments

- here, a different image is displayed when a segment is selected

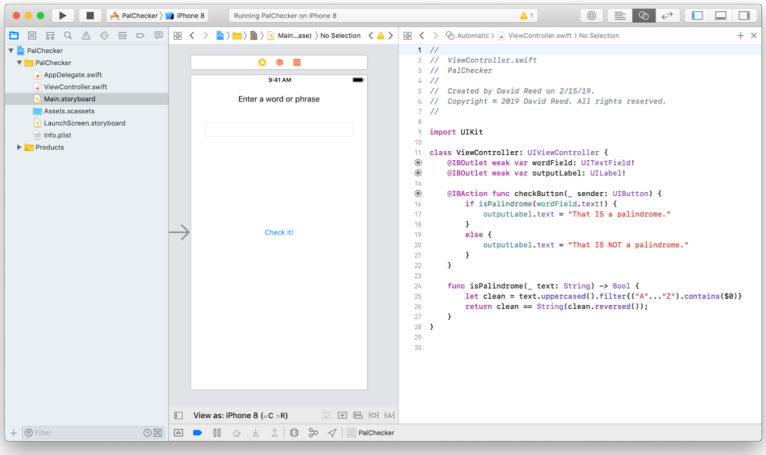


21

Example: text field

UITextField allows the user to enter text, which can be accessed

- here, the user enters a word or phrase in the text field
- a message identifying whether it is a palindrome or not is displayed in a label

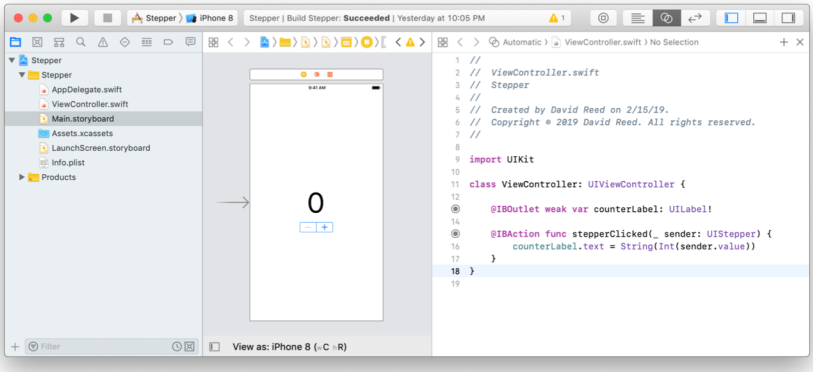


22

Example: stepper

UIStepper allows simple increment/decrement of a counter

- here, the user clicks +/– to increment/decrement

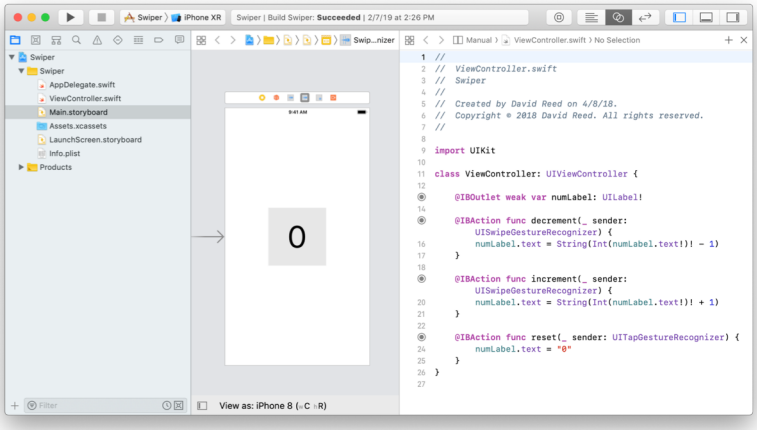


23

Example: swipe gesture recognizers

similar to the earlier Tap Gesture Recognizer example, we can add Swipe Gesture Recognizers to the main View

- here, swipe right increments, swipe left decrements, tap resets



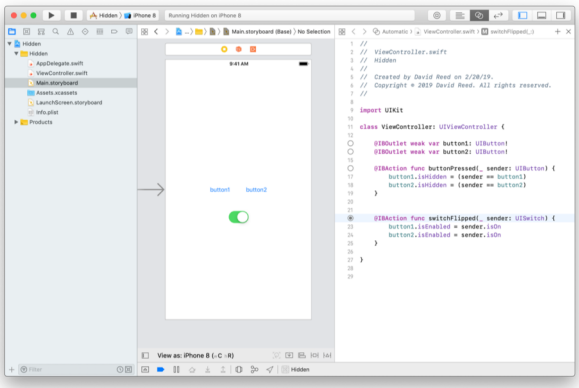
24

Example: isHidden & isEnabled

can hide views or even disable controls

- every UIView element has an isHidden field
- every UIControl (which inherits from UIView) also has an isEnabled field

- here, one button is visible, clicking on it hides it and makes the other visible
- if the switch is turned off, all the buttons are disabled

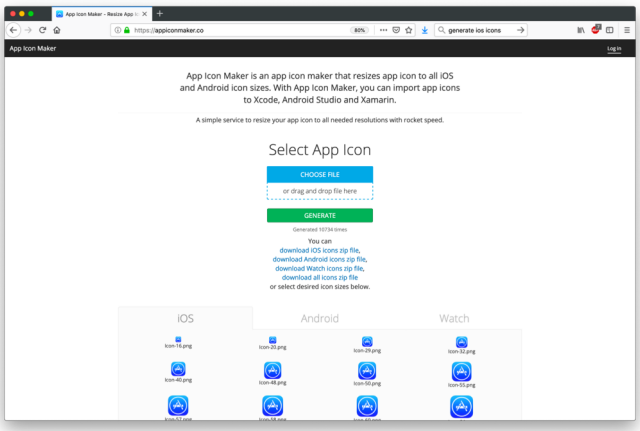


25

Adding an app icon

adding an icon to your app is simple but tedious

1. download or create an icon image (it should be square)
2. go to a site (many free ones exist) that will generate all of the required resolutions
3. drag your image there, then download a zip file of images



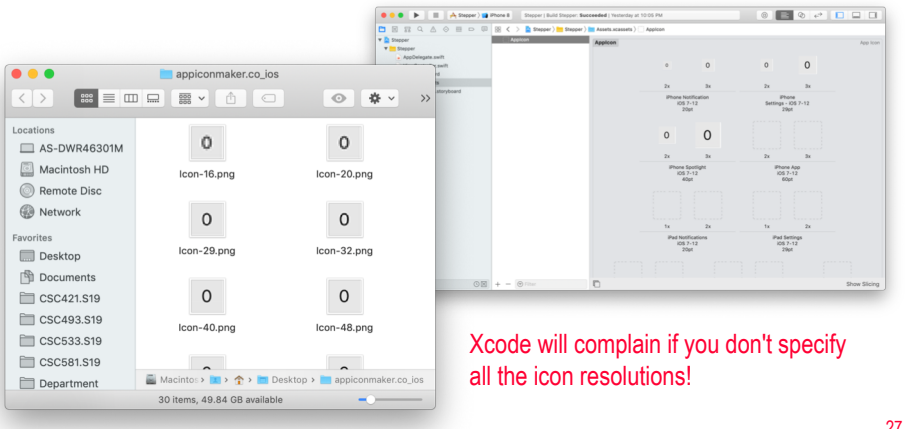
26

Adding an app icon

4. open the Assets folder in your project and select Appicon
5. drag the corresponding icon image to each slot

   2x specifies twice the resolution; 3x specifies three times

   e.g., 20 pt 2x → drag in Icon-40.png



Xcode will complain if you don't specify all the icon resolutions!

27