Welcome Back

It’s yet another tuesday. But this one feels like monday.

New due dates

- Project 1: 9/18, 10pm CST
- Project 2: 10/13, 11pm CST
- Project 3: 11/10, 11pm CST

Classes in Swift

- In a general sense, a class is an extensible program-code template for creating objects
- provides initial values for state and implementations for behaviors

Classes and Structs

- Classes and Structs are virtually identical in most ways in Swift
- properties
- methods
- subscripts
- initializers
- extensible
- conform to protocols

Unique to Classes

- Classes can inherit from other classes
- can be type cast at runtime to determine class type
- have deinitializers
- use reference counting

Unique to Structs

- Structs are a value type

Declare a Struct or Class

- Keywords: struct / class

```
class ClassName {
// implementation
}
struct StructName {
// implementation
}
```

Television class

```swift
struct Resolution {
    var width: Int
    var height: Int
}
class Television {
    let resolution = Resolution(width:1920, height:1280)
    var visibleArea = Resolution(width:800, height:600)
    var channel = 3
}
```

Accessing properties

- Dot notation to access properties, just like in most languages

```
let tv = Television()
let tvResolution = tv.resolution
let fullWidth = tv.resolution.width
tv.visibleArea.width = 960
// THIS IS VALID!
```

Memberwise Initializers

- Structs get automatic memberwise initializers

```
struct PixelValue {
    var hue = 0.0
    var saturation = 0.0
    var intensity = 1.0
}
let pixel = PixelValue(hue: 2.3, saturation: 0.1, intensity: 0.5)
// This is automatically created for you!
```

Structs are Value Types

- This is essential to truly understanding class/structure delineation
- Entire WWDC 2015 session on using value types
- simple answer: value types allow safety

Structs are Value Types

```
class Speed {
    var mph = 0
}
class Automobile {
    var speed = Speed()
}
class Bicycle {
    var speed = Speed()
}
```

Structs are Value Types

```
var speed = Speed()
speed.mph = 60
var auto = Automobile()
auto.speed = speed
var bike = Bicycle()
speed.mph = 10
bike.speed = speed
// Now the automobile is following the limits of the bike!
```

Structs are Value Types

- Instead, define Speed as a struct
- This makes it a value type
- The speed is copied when necessary, that is, to make sure that the auto’s speed is not limited by the bicycle’s speed
- If the initial speed set was 10, and unchanged, then no copy would actually take place
- This is copy-on-write
- This is safe and efficient

Struct as value type

- I’ve covered this in 4 slides, it’s a 40 minute presentation in WWDC 2015
- Check out the video
- great puns
- haskell references
- digs into the internals

Classes are reference types

- This should seem familiar, congruent to pointers in Objective C (or more accurately, more like references in C++)
- Classes can optionally have equality operators (==) through a custom-defined function
- We'll cover this later
- To check if two objects are both references to the exact same object, use triple-equals (=== / !==)

Type bridging

- We’ve mentioned type bridging for Array/NSArray, Dictionary/NSDictionary, String/NSString
- One important difference here:
- Array, Dictionary, String are value types
- NSArray, NSDictionary, NSString are reference types

More about properties

- Properties may be stored or computed
- Can be associated with the type itself, known as type properties: this can be compared to class variables instead of member variables
- Properties can be monitored for changes

Stored Properties

- Both constants and variables may be stored properties
- Stored properties can have a default value
- Constants can be set during initialization, not just a default value
- more on this when we discuss initializers

Computed properties

- Sometimes we don't want to store properties
- Perhaps this is because the computation is expensive and rare
- Sometimes we know there are multiple factors that contribute to a value
- Computed properties allow us to obtain a value at the last possible moment

Computed properties

```
struct ComputingPropertyStruct {
    var value: Int
    var computed: Int {
        get {
            return value * 5
        }
    }
}

let cps = ComputingPropertyStruct(value:7)
print(cps.computed)
```

Constant structures

```
struct Orchestra {
    var numberOfViolins: Int
}
let myOrchestra = Orchestra(numberOfViolins:100)
myOrchestra.numberOfViolins = 20
// THIS WILL FAIL
```

Constant structures

- If a struct is defined as a constant, then even variable properties may not be changed at runtime
- This is in stark contrast to classes
- Exercise: experiment by changing Orchestra to a class, and see how it behaves differently

Lazy properties

- properties which are allocated / computed as needed, instead of at initialization time
- use the lazy keyword
- these must be variable (and often must be optional), as they will change from nil to a value as the lazy property is initialized

Struct Initializers

- With every struct you create, you get an initializer for free.
- If you manually create an initializer, you must initialize all non-optional variables & constants within your initializer

Struct Initializers

```
struct TemporaryObject {
    var one: Int
    let two: Int
    var three: Int?
    init(one: Int) {
        self.one = one
        self.two = 16
    }
    init(one: Int, two: Int) {
        self.one = one
        self.two = two
    }
}
let to = TemporaryObject(one: 3, two: 5)
let t2 = TemporaryObject(one: 6)
```

Two-phase initialization

- Initialization in swift takes place in two phases
- First, any default values are set
- Then all values are set in the initializer
- At the end of the second phase, all non-optional values must be set.

Two-phase initialization

```
// amend the previous example
var one: Int = 1
...
init(one: Int) {
    print(self.one)
    self.one = one
    print(self.two)
    self.two = 16
}
```

Class Initializers

- Due to inheritance, initializer rules for classes are slightly more complex
- All classes have a designated initializer
- By the end of the designated initializer, all elements must have a value
- Classes may have one or more convenience initializers
- Convenience initializers must ultimately call a designated initializer.

Class Initializers

```
class InterestingClass {
    var valueOne: Int
    var valueTwo: String
    init(one: Int, two: String) {
        valueOne = one
        valueTwo = two
    }
    convenience init(value: Int) {
        self.init(one:value, two:"\(value)")
    }
}
```

Closures in Swift

- Closures can be thought of, most simply as functions
- Closures capture variables from their surrounding environment
- Closures allow us to build software with callbacks, with a very simple syntax
- We most often use closures to achieve asynchronicity: the block is called when a very long process is complete, such as downloading information from the network

Adding a closure as an argument

```
func functionWithAClosure(value: Int, completion:(replacementValue:
Int)->()) {
    let v = value * 5
    completion(replacementValue: v)
}

functionWithAClosure(5, completion:  { (replacementValue) in
    print(replacementValue)
})
```

Adding a closure as an argument

- Please note the general syntax of ()->()
- In each case, the parenthesis can be understood as a tuple
- Note that you don't have to provide type for arguments in most cases; because Swift is type safe, the compiler knows what type it's passing, and assumes that you do as well.

Trailing closures

- One of the coolest parts of Swift's syntax is that a closure, when the final argument of a function, can be placed after the closing parenthesis.

```
func functionWithAClosure(value: Int, completion:
(replacementValue: Int)->()) {
    let v = value * 5
    completion(replacementValue: v)
}

functionWithAClosure(5)  { (replacementValue) in
    print(replacementValue)
  }
}
```

sort

- Using closures provide a very simple syntax for sort:

```
let array = [3,7,4,9,3,1,6,0,9,6,5]

let newArray = array.sort { (a:Int, b:Int) -> Bool in
    return a > b
}
```

simpler sort!

- Default arguments for closures are available as $0, $1, and so on

```
let newArray2 = array.sort() {
    return $0 > $1
}
```

even simpler!

```
let newArray3 = array.sort(){$0>$1}
```

what if it's not simple integers?

```
struct Pet {
    let name: String
    let weight: Int
    let age: Int
}
extension Pet: Comparable {
}
func ==(a:Pet, b:Pet) -> Bool{
    return a.age == b.age
}
func <(a:Pet, b:Pet) -> Bool{
    return a.age < b.age
}
```

Creating and sorting objects

```
let arrayOfPets = [
    Pet(name: "Pickles", weight: 18, age: 16),
    Pet(name: "Banjee", weight: 27, age: 5),
    Pet(name: "Rowlf", weight: 20, age: 8)
]

let sortedByAge = arrayOfPets.sort() {$0 > $1}
print(sortedByAge)
```

map

- Map solves the age-old question:
- I have a list of pets, how do I get a list of pet names?
- Simplifies (and safety-ifies) a larger batch of code

```
var arrayOfPetNames: [String] = []

for pet in arrayOfPets {
    arrayOfPetNames.append(pet.name)
}

print(arrayOfPetNames)
```

using a map

```
print(arrayOfPets.map(){$0.name})
```

filter

- Where map returns an equally-sized array of elements given an existing array of elements, filter reduces a potentially smaller array of elements which match a given condition
- Combined with map & sort, this becomes very powerful.

filter

```
var arrayOfOlderPets: [Pet] = []

for pet in arrayOfPets {
    if pet.age > 6 {
        arrayOfOlderPets.append(pet)
    }
}

print(arrayOfOlderPets)
```

using filter

```
print(arrayOfPets.filter(){$0.age>6})
```

Combine them all!

- This one line of code shows the magic in these functional programming elements
- It displays a list of pet names for pets over the age of 6, from youngest to oldest

```
print(arrayOfPets.filter(){$0.age>6}.sort().map{$0.name})
```