Swift - Classes and Structs

Mobile Computing - iOS

Objectives

- Students will be able to:
  - Contrast classes and structs; methods and functions
  - Create classes and structs
  - Write type properties and methods and explain their purpose

Classes & Structs

You should be familiar with classes from Java, but possibly not structs. Structs, short for structure, were the original way that C bundled together named fields and only contained values. Their usage has evolved to where in Swift there is very little difference between a struct and a class. *Both* classes and structs ...

- Define **properties** to store values (similar to attributes)
- Define **methods** to provide functionality
- Define **initializers** (similar to constructors)
- Can conform to **protocols** (similar to interfaces)
- Define **subscripts** so we can use [ ] to access individual elements

This document will just cover the basics of initializers ... slightly more detail will follow once we've studied inheritance

A Classy Class Example

```
class Planet {

    var name:String      // stored properties
    var mass:Double

    init(name:String, mass:Double){  // initializer
        self.name = name
        self.mass = mass
    }

    // method
    func isSmallerThan(anotherPlanet:Planet) -> Bool {
        return mass < anotherPlanet.mass
    }
}

let pl = Planet(name: "Earth", mass: 5.972e24)
let pl2 = Planet(name: "Jupiter", mass:1.898e27 )

if pl.isSmallerThan(anotherPlanet:pl2) {
    print("Yes, \(pl.name) is smaller than \(pl2.name)")
}
```

A class in Swift consists of properties, initializers and methods

Instantiate a class by invoking an initializer. Notice, no **new** -- this isn't Java!

Use dot syntax to invoke a method or access a property, just like in Java

A Super Struct Example

```
struct Planet {

    var name:String      // stored properties
    var mass:Double

    func isSmallerThan(anotherPlanet:Planet) -> Bool {
        return mass < anotherPlanet.mass
    }
}

let plan = Planet(name: "Earth", mass: 5.972e24)
let plan2 = Planet(name: "Jupiter", mass:1.898e27 )

if plan.isSmallerThan(anotherPlanet:plan2) {
    print("Yes, \(plan.name) is smaller than \(plan2.name)")
}
```

What's being defined for us?

Classes vs. Structs

1. Classes have an inheritance mechanism, structs do not

2. Structs automatically get an initializer to initialize all of its properties: with classes, you have to write your own.

3. Both classes and structs can support polymorphism (using either protocols, or in the case of classes, protocols or superclasses)

4. Instances of a class, aka objects, are passed by reference (*reference* type) -- structs are *copied* (*value* type)

    1. Copying can be a Good Thing, preventing various bugs: if you have a struct, nobody else (no other function in your program, or no other thread) can change it

5. A Struct variable is allocated on the stack. (Lifetimes are limited and garbage collection is avoided.) Objects of a class are allocated in heap.

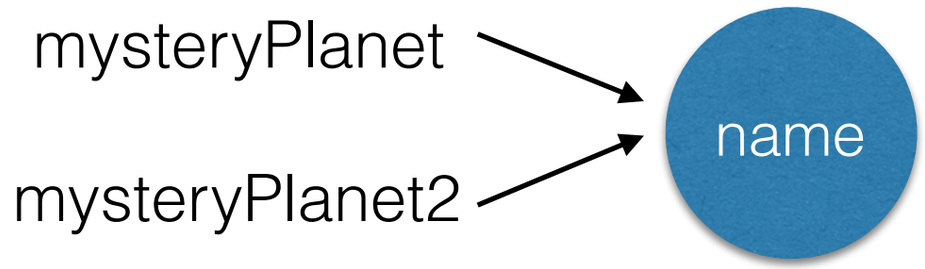
6. Generally speaking, structs are faster.

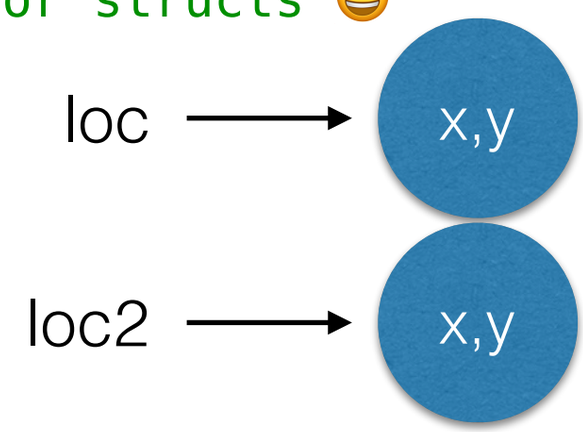
7. Apple is fond of structs: Strings, Arrays, Sets, Dictionaries, Ints, Floats, Doubles, Bools, etc., etc., are all structs (value types)

8. Most of the people reading these notes will be more comfortable with classes -- but that is *not* a reason for using them instead of structs!! You should use structs when it is appropriate. I.E. You have relatively few properties that are value types. You can treat instances as values and copying is appropriate. Otherwise, Apple recommends that you use a class.

References versus Copying

```
class Planet {
    var name:String      // a property
    init(name:String){   // an init
        self.name = name
    }
}

var mysteryPlanet:Planet = Planet(name:"Mars") // Swift does not use new keyword
var mysteryPlanet2:Planet = mysteryPlanet // mysteryPlanet2 also points to "Mars"

mysteryPlanet.name = "Jupiter"
print(mysteryPlanet2.name) // output: Jupiter
/* ================================================================*/
struct Location {
    var x:Int
    var y:Int    // we don't have to define an initializer for structs 😀
}

var loc:Location = Location(x: 10, y: 10)
var loc2:Location = loc   // makes a separate copy of loc

loc.x = 100
print(loc2.x) // output: 10
```





A Simple Class Example

```
class Student {
    var name:String
    var gpa:Double

    init(name:String, gpa:Double){
        self.name = name
        self.gpa = gpa
    }

    func isPassing() -> Bool {
        return gpa >= 2.0
    }

    func asString() -> String {
        return String(format:"Name: %@, gpa: %.2f",name, gpa)
    }
}
```

name & gpa are properties

Use **self** when an object needs to refer to itself in a method or init

Properties must be assigned a value *when* they are declared, or *before* the init() method finishes -- unless they are lazy (see the docs for details)

A Simple Class Example, Completed

To instantiate an object, write the type name, and in ( ), the arguments for its initializer. Swift does not use **new**.

```
var advisee:Student = Student(name: "George Lucas", gpa: 3.5)
```

Use standard dot notation to access properties and invoke methods

```
if advisee.isPassing() {
    print("That's a start!")
}

if advisee.gpa == 4.0 {
    print("That is a really great gpa, \(advisee.name)")
} else {
    print("Well, \(advisee.name), \(advisee.gpa) is not too shabby")
}

print("The student is \(advisee.asString())”)
advisee.gpa *= 0.3
```

Class to Struct

```
struct Student {

    var name:String
    var gpa:Double

    func isPassing() -> Bool {
        return gpa >= 2.0
    }

    func asString() -> String {
        return String(format:"Name: %@, gpa: %.2f",name, gpa)
    }
}

var advisee:Student = Student(name: "George Lucas", gpa: 3.5)

if(advisee.isPassing()){
    print("That's a start!")
}
```

To make this into a struct, all we did was
a) change **class** to **struct**,
b) delete the initializer.

Properties

- **Stored properties** hold a value that you can access directly. They look a *lot* like variables (but behind the scenes they are different)

- **Computed properties** have a getter and setter -- code that gets executed when you attempt to access (get) or mutate (set) a property

An Example

```
struct Square  {
    var x:Double
    var y:Double
    var side:Double // a backing variable

    var area:Double {
        get {
            return side * side
        }

        set {
            side = sqrt(newValue) //  area = side*side, so side= √area
        }
    }

}
```

0. Omit set for a read-only property
1. To simplify a read-only property, omit get { } -- only include the code in between, e.g.,
var area:Double {return side*side }

newValue is the assigned value.

```
var square:Square = Square(x: 0, y: 0, side: 9.0)
print("Area is \(square.area)") // Output: Area is 81.0

square.area = 100
print("Side is \(square.side)") // Output: Side is 10.0
```

More on Properties

- Properties can be private. Only methods inside the class are allowed to access the property.

- Read only computed properties have a getter and no setter.

- Sometimes a computed property will be backed by a private stored property. In that case, the store property usually has the same name as the computed property with an underscore prefix.

An Example

```
class Counter  {
    private var _capacity:Int
    var capacity:Int {
        get { return _capacity }
        set {
            if newValue < 0 {_capacity = 0}
            else  {_capacity = newValue}
            if _count > capacity {_count = capacity}
        }
    }

    private var _count:Int
    var count:Int { return _count}

    init(capacity:Int){
        _capacity = capacity
        _count = 0
    }

    func reset(){
        _count = 0
    }

    func add(persons:Int){
        if(persons < 0) { return}

        _count += persons
        if(_count > capacity){
            _count = _capacity
        }
    }
}
```

capacity is a computed property backed by a private stored property

count is a read only computed property.

Could/should this be a struct? Why?

Read-Only Computed Properties

- If a computed property is read-only, you can eliminate the get { }. You can have multiple statements, but you must return a value

```
struct Temperature {

    var kelvin:Double

    var celsius:Double {
        get {
            return kelvin  - 273.13
        }
    }
}
```

```
struct Temperature {

    var kelvin:Double

    var celsius:Double {
            return kelvin  - 273.13
    }
}
```

These two structs are equivalent

Invoking Methods

- Historical fun fact: In previous versions of Swift:

    1. When invoking an init(), all the arguments were labeled with the corresponding parameters

    2. When invoking a method all the arguments *except the first* were labeled with the corresponding parameters

        - Why not the first? The idea was that the method name would serve as a label for that first argument

- Starting in Swift 3, labeling of arguments depends on how the parameters of the methods are declared and follows the rules we established for functions.

- Apple is making a push to remove unneeded words from function names and towards an encoding for every argument. Even though this is the case, there are still legacy classes that will follow the older style.

The Mechanics

```
class Drawing {
    var x:Double
    var y:Double

    init(x:Double, y:Double){
        self.x = x
        self.y = y
    }

    func distance(_ a:Double, b:Double, c:String){
            return sqrt((x-a)*(x-a) + (y-b)*(y-b))
    }
}

var pt:Point = Point(x:15.0, y:25.0)

pt.distance(25.0, b:15, c:"What is this??")
```

In this init(), all the parameters serve as external labels for the arguments

In this method all the parameter names except the first serve as argument labels. The first argument is unlabeled.

If you leave out the _, then you would need to specify a: for the first argument.

Mutating functions

- A struct is a value type, so by default *its properties cannot be changed* within instance methods

- If you do need to change the values of a type, you can, however: just declare the function that does so as mutating

```
struct Location {
    var latitude:Double
    var longitude:Double

    mutating func move(deltaLatitude:Double, deltaLongitude:Double){
        latitude += deltaLatitude
        longitude += deltaLongitude
    }
}

// omitting mutating produces a cryptic error and a helpful note:
//   left side of mutating operator isn't mutable: 'self' is immutable
//   mark method 'mutating' to make 'self' mutable
```

Type Properties and Methods

- Heretofore we have worked exclusively with **instance** properties and methods -- those associated with individual instances, and that vary in value/behavior from one instance to another

- **Type** properties and methods belong to a type (class or struct), not to an instance.

- Use type properties:
    - To define a variable/constant that applies to all instances
    - When you need a collection of constants (a struct would work particularly well)
- Use type methods:
    - When the method does not require any instance properties/methods, i.e., it is "freestanding"

Working with Type Properties and Methods

- Type properties and methods are declared by prefixing them with the word **static**.

- To use a type property or method, generally precede it with the type's name. [Exception: in a type method, we can invoke other type methods, or use type properties, without using the type's name, although it is OK to do so.]

Type Properties & Methods - Example

```
struct Math {
    static let pi:Double = 3.141592654
    static let e:Double = 2.71828

    static func pow(x:Double, n:Int) -> Double {
        var product:Double = 1.0
        for _ in 0 ..< n {
            product *= x
        }
        return product
    }
}

var radius:Double = 10.0
var area:Double = Math.pi * Math.pow(x:radius, n:2)

print(Math.pow(x:3.0, n:4)) // Output: 81.0 (3**4)

var euler:Math = Math()
// print(euler.pi) // this would generate an error
//-- static member 'pi' cannot be used on instance of type 'Math'
```

ICE [26]

- Create a class called Point2D, with properties for:
    - x, y (Doubles), realName (String), name (String) [2]
    - name will be a computed property — it must have a length > 0 [4]
    - use realName as the stored property that name will be based on [1]
    - A default initializer that sets x, y and name to 0.0, 0.0 and “generic” [3 - 1 point off if it doesn’t invoke the other initializer]
    - A 3-parameter initializer that does the usual thing [3]
    - distanceFrom() — it will be passed in another Point2D and return the distance between this point and the other using the standard distance formula [4]
    - closeTo() — it will be passed in another Point2D and a tolerance, return true if the distance between this Point2D and that is within tolerance [4 - 2 points off if it doesn’t call distanceFrom()]
- Demonstrate that your code works by instantiating 2 Point2D objects,
    - (0.0,0.0, “Charlie”), [2]
    - (3.0, 4.0, “Brown”). Print out the distance between them, [1] and verify that they are within a tolerance of 6.0 of each other, but *not* within a tolerance of 2.0 of each other. [You will need to invoke closeTo() twice.] [2]

Resources

- You can read about functions in Swift 2.2 vs Swift 3 here…https://www.hackingwithswift.com/swift3

- https://developer.apple.com/videos/play/wwdc2015/414

- WWDC Video on structs vs classes

- https://lists.swift.org/pipermail/swift-users/Week-of-Mon-20170626/005731.html

- Apple documentation: Stacks vs Classes