CS 329E
Elements of Mobile Computing

Spring 2018
University of Texas at Austin

Lecture 3

More Swift

- Optionals
- Nil coalescing operator
- guard keyword
- More on Classes
- Collection types

More Swift

Optionals:

- What are optionals?
  - A way to allow a variable to be ‘nil’ (nothing)
  - Similar in concept to pointers in Objective-C (and other languages) being nil - that is, not pointing to anything

More Swift

Optionals:

- Swift, by default, enforces the notion of “value required” for every type of variable
- This means that, unless specified otherwise, every variable *must* represent/point-to a value

So, this is **not valid**, by default, in Swift:

```
// error: type 'Person' does not conform to
// protocol 'NilLiteralConvertible'
var p:Person = nil
```

More Swift

Optionals:

- Forcing “value required” means you, by default, write more robust/protective code; because you can’t, by default, use a variable that doesn’t represent/point-to something!!
- A very common source of app crashes (all kinds of apps, not just iOS apps) is a pointer that is not pointing to anything and code uses the variable thinking it is - then Kaboom!!
  - The dreaded ‘null reference’/‘nil pointer’ exception

More Swift

Optionals:

- That said, IF you need a variable to be able to not point to anything, you can define the variable to be a data type with the *optional operator* - ‘**?**’
- Such a variable can be set to nil to represent the absence of a value
- Any data type can be used with the optional operator

```
var tot1:Int? = 55   // valid
var tot2:Int? = nil   // valid
```

More Swift

Optionals:

- A variable declared as an optional is a *wrapper* object for the underlying variable
  - Therefore, you cannot directly access the wrapped value

```
 var tot:Int? = 55

 // error: Value of optional type ‘Int?’ not unwrapped;
 // did you mean to use ‘!’ or ‘?’?
var sum = tot * 2
```

More Swift

Optionals:

- If an optional is not given an initial value, it is automatically set to nil

```
        var tot:Int?
```

- You can check for the presence of a value by comparing to nil

```
        if tot != nil {
        }
```

More Swift

Optionals - Force Unwrapping:

- You can obtain the value of an optional by using the *force unwrap* operator - ! (the exclamation point)

```
var tot:Int? = 55

var sum = tot! * 2   // Success!! sum = 110
```

More Swift

Optionals - Force Unwrapping:

```
var tot:Int? = nil

var sum = tot! * 2  // This causes a crash!
```

Unwrapping an optional variable when it is set to nil (either implicitly or explicitly) causes a crash!

More Swift

Optionals - Force Unwrapping:

```
var tot:Int? = 55
var sum:Int

if tot != nil {
    sum = tot! * 2
}
```

Ugh! Do we really need to do this every time we want to use an optional - to prevent a crash? Yes - something like it.

In-Class Exercise

In-Class Exercise

Write some code using optional variables.

- Define an integer optional variable named ‘age’; do not set it to anything; use it in a print statement

- Set the variable to 35; use it in a print statement

- Define an integer variable named ‘age2’ and assign it the value of the variable ‘age’.

More Swift

Optionals - Force Unwrapping:

- There are a few features in Swift that assist with force unwrapping:
  - Optional Binding
  - Optional Chaining
  - Implicitly Unwrapped Optionals

More Swift

Optionals - Optional Binding:

- You conditionally assign the unwrapped value to a temporary variable - typically in an if statement

```
var tot:Int? = 55
var sum:Int

if let t = tot {
    sum = t * 2
}
```

The only benefit is if ‘tot’ is not nil, ’t’ contains the value, with no need to explicitly unwrap. That said, you’ve still got an ‘if’ statement.

More Swift

Optionals - Optional Binding:

```
var tot:Int? = 55
var sum:Int

if let t = tot {
    sum = t * 2
}

 print(sum) <== compile error!!
            “Variable ‘sum’ used before being initialized”
```

More Swift

Optionals - Optional Chaining:

- Similar to optional binding, but can create a chain of optionals to check

We’ll start with these classes:

```
class Department {
    var title:String = “Accounting”
}

class Person {
    var name:String = “Joe”
    var dept:Department?
}

class Customer {
    var salesRep:Person?
}
```

More Swift

Optionals - Optional Chaining:

- Let’s assume a Customer object was constructed and assigned to the variable named ‘cust’

```
// Can’t do this, because salesRep is an optional
//
// error: Value of optional type ‘Person?’ not
// unwrapped; did you mean to use ‘!’ or ‘?’?
let cust = Customer()
let name = cust.salesRep.name
```

More Swift

Optionals - Optional Chaining:

```
// You must unwrap salesRep using the force
// unwrap operator
 let name = cust.salesRep!.name
```

But there’s a chance salesRep could be nil! So, you still need to check before doing this.

More Swift

Optionals - Optional Chaining:

```
// Instead of this
let name = cust.salesRep!.name

// You could do this
if let name = cust.salesRep?.name {
}
```

Benefit: you are checking the optional for nil and, if it’s not nil, force unwrapping occurs and the eventual value (name) is assigned to the local constant ‘name’.

More Swift

Optionals - Optional Chaining:

- You can chain optionals into one statement
- Unwrapping occurs until an optional element of the chain is nil
- If any optional element of the chain is nil the rest of the processing is skipped

```
// Will only execute the if-statement true path if both
// optionals are not nil.
if let deptTitle = cust.salesRep?.dept?.title {
}
```

Benefit: You can include force unwrapping of multiple optionals in one statement.

More Swift

Optionals - Implicitly Unwrapped Optionals:

- Automatically unwraps a variable when it is used
- Add unwrap operator to the end the data type

```
class AutoUnwrapped {
    var me: Person!
}
```

More Swift

Optionals - Implicitly Unwrapped Optionals:

```
class AutoUnwrapped {
    var me: Person!
}

var au = AutoUnwrapped()

// You have to be careful to make sure the - in
// this case - ‘me’ property is set before using.
println(au.me.firstName)  // Crash!!!! ‘me’ not set.
```

More Swift

Optionals - Implicitly Unwrapped Optionals:

```
class AutoUnwrapped {
    var me: Person!
    init() {
        me = Person(firstName:”Joe”, lastName:”Smith”)
    }
}

var au = AutoUnwrapped()
print(au.me.firstName)      // output: Joe
```

More Swift

Optionals - Implicitly Unwrapped Optionals:

- Used in the context of user interfaces, where you *know* the properties will have values before use
  - Because the framework sets them

```
class MyController {
    var myButton: UIButton!
    var myTextField: UITextField!
}
```

More Swift

Optionals:

- Be judicious in your use of optionals
- Don’t use them if a variable or property should never be nil - that is, always have a value when used

More Swift

Nil coalescing operator - ??:

A way to easily return an *unwrapped* optional or a default value

```
    var someOptional : Int?
    var aDefaultValue = 42

    var theAnswer = someOptional ?? aDefaultValue
```

More Swift

Nil coalescing operator - ??:

```
var theAnswer = someOptional ?? aDefaultValue
```

Same as:

```
if (someOptional != nil) {
    theAnswer = someOptional!   ←——  Unwrapped
} else {
    theAnswer = aDefaultValue
}
```

More Swift

guard keyword:

- To provide a better syntactic structure to verify data before use.

1. Pyramid of doom - indentation gets worse with each check:

```
if firstName != "" {
    if lastName != "" {
        if address != "" {
            // do something
        }
    }
}
```

More Swift

guard keyword:

2. Early return - a little better, but not much:

```
if firstName == "" { return }
if lastName == "" { return }
if address == "" { return }
// do something
```

More Swift

guard keyword:

- Any conditions you would have checked using ‘if’ before, you can now check using guard

Example 1:

```
        guard age > 18 else { return false }
```

More Swift

guard keyword:

Example 2:

```
func printName(name: String?) {
    guard let unwrappedName = name else {
        print("You need to provide a name.")
        return
    }

    print(unwrappedName)  ←  No additional unwrapping needed
}
```

More Swift

guard keyword:

Benefits:

- Makes your intent clearer
  - You tell guard what you want to be the case rather than the reverse
- Any optional variables unwrapped by guard remain in scope after the guard finishes
  - Means after you check an optional variable with guard and it passes, you can use it immediately
- Gives you shorter code
  - Means fewer bugs and happier developers

More Swift

Variables - rules to live by:

- When you need a variable, make it a constant
- If it can't be a constant, make it require a value (the default)
- If it needs to possibly be nil, make it an optional
- When using an optional, program defensively – what if it's nil?
- If you know if will never be nil when you're using it, maybe it should be an implicitly unwrapped optional (or maybe it shouldn't be an optional at all)

More on Classes

More Swift

More on classes:

- Computed Properties
- Inheritance
- Initializers
- Default initializer
- Designated initializer
- Convenience initializers

More Swift

Computed Properties:

- Properties in a class that provide a getter and optional setter to retrieve and set other properties
- They do not have a value of their own

```
class Person {
    private var _name: String

    // computed property
    var name : String {
        get { return _name }
        set(newValue) { _name = newValue }
    }
}
```

More Swift

Computed Properties:

- ‘get’ can be omitted if there is no ‘set’

```
class Person {
    private var _name: String

    // computed property
    var name : String {
        return _name
    }
}
```

More Swift

Computed Properties:

- You can add any code within the computer property

```
class Person {
    private var _name: String

    // computed property
    var name : String {
        return “Name is: ” + _name
    }
}
```

More Swift

Inheritance:

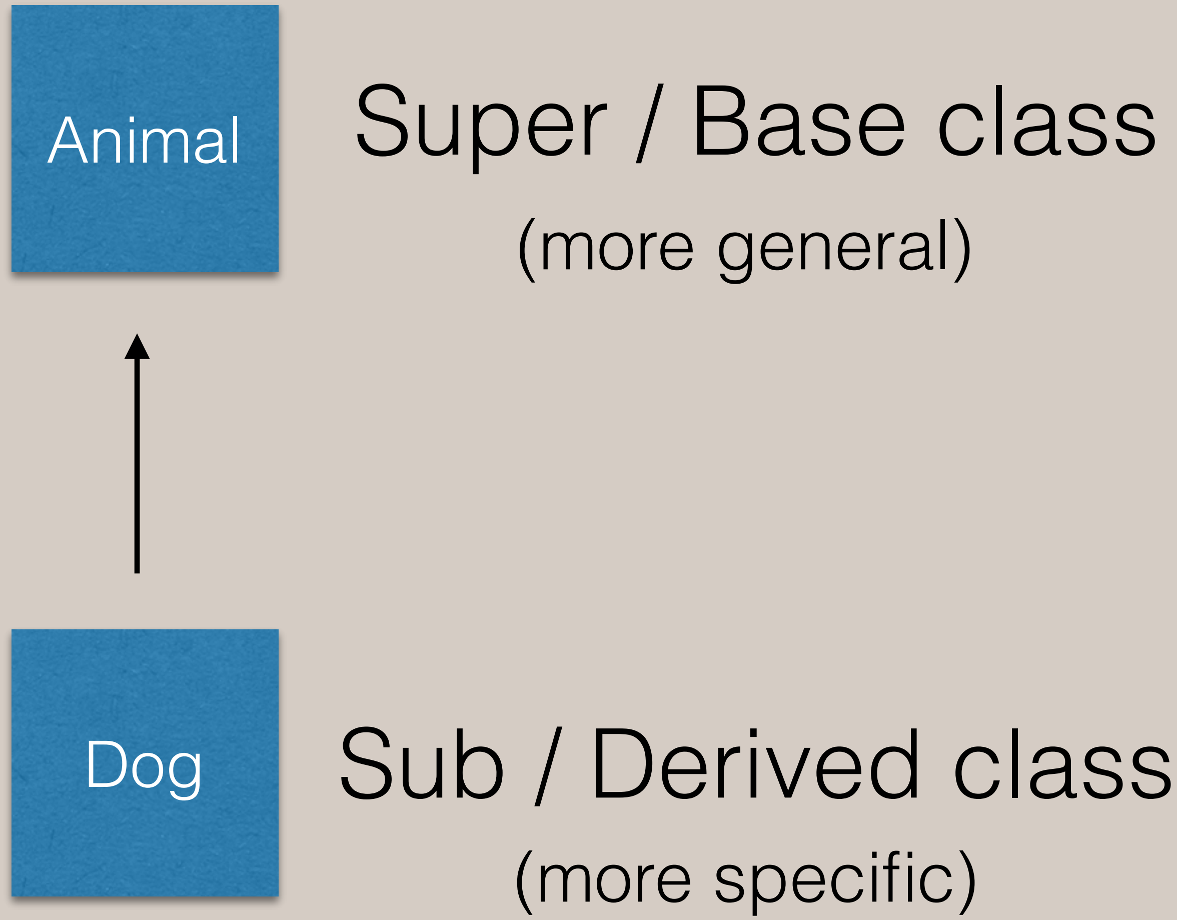
- Basics of inheritance:
  - A class can inherit properties, methods and other characteristics from another class
  - The inheriting class is known as the sub or derived class
  - The inherited class is known as the super or base class

More Swift

Inheritance:

- How to establish inheritance:

```
class Animal {
    var id:Int = 0
    init(id:Int) {
        self.id = id
    }
}
class Dog : Animal {
    var name:String = “”
    init(id: Int, name:String) {
       super.init(id: id)
       self.name = name
}
```



More Swift

Class initializers:

- Class initializers are automatically called after memory has been allocated for the object
- The purpose of an initializer method is to get an object into a *good known starting state*

More Swift

Default initializer:

- The ‘init’ method with no arguments
- If you don’t define one, one is auto-generated for you with an empty method body
- You *must* define an ‘init’ method if you don’t set property defaults for any non-optional properties

```
class Person
{
    var name: String
    init() {
       name = "Joe"
    }
}
```

Usage:

```
    var p = Person()
    print(p.name)   // output:  Joe
```

More Swift

Default initializer:

- You can avoid a default initializer implementation by setting property default values
  - But we will always write one for every class we create

```
class Person
{
    var name: String = “Joe”
~~    init() {~~
~~        name = "Joe"~~
~~    }~~
}
```

More Swift

Default initializer:

- You can have additional initializers by defining versions with arguments - method *overloading*

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

More Swift

Default initializer:

- ‘self’ is needed only when there is ambiguity
  - Case - when an instance method argument has the same name as a property

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

More Swift

Default initializer:

- If you don’t explicitly include ‘self’, Swift assumes you are referring to a property or method of the current instance

```
class Person
{
    var name: String
    init() {
        name = "Joe"
    }
}
```

More Swift

Default initializer:

- You can have overloaded default initializers
- The differentiator is the argument list

```
class Person
{
    var name: String
    init() {
       name = "Joe"
    }
    init(name:String) {
       self.name = name
    }
}
```

More Swift

Default initializer:

- Default argument values

```
class Person
{
    var name: String
    init(name:String = “Sam”) {
       self.name = name
    }
}
```

More Swift

Default initializer:

- Default argument values

```
class Person
{
   var firstName: String
   var lastName: String
   init(firstName:String = “Sam”, lastName:String =“Smith”) {
      self.firstName = firstName
      self.lastName = lastName
   }
}
```

Usage - 4 ways to instantiate an object of type Person, based on above definition:

```
p1 = Person()
p2 = Person(firstName: “Joe”)
p3 = Person(lastName: “Bailey”)
p4 = Person(firstName: “Joe”, lastName: “Bailey”)
```

More Swift

Designated initializer:

- The main initializer to be used for a class
- The initializer that all other initializers eventually (should) end up funneling through
- Funnel points through which initialization takes place, and through which the initialization process continues up the superclass chain
- Designated initializers tend to set all of the properties and let the user send in values for each

More Swift

Designated initializer:

```
class Person
{
    var firstName: String
    var lastName: String
    // This init is the designated initializer
    init(firstName:String = “Sam”, lastName:String =“Smith”) {
       self.firstName = firstName
       self.lastName = lastName
    }
    convenience init() {
       self.init(“joe”, “Johnson”)  // calls the designated initializer
    }
}
```

More Swift

Convenience initializers:

- Secondary, supporting initializers for a class
- You should define a convenience initializer to call a designated initializer from the same class
- The init method is prefixed with *convenience*

More Swift

Convenience initializer:

```
class Person
{
    var firstName: String
    var lastName: String
    init(firstName:String = “Sam”, lastName:String =“Smith”) {
       self.firstName = firstName
       self.lastName = lastName
    }
    convenience init() {
       // calls the designated initializer
       self.init(firstName:“joe”, lastName:“Johnson”)
   }
}
```

More Swift

Rules for Designated and Convenience Initializers:

- Swift has three rules as to how designated and convenience initializers relate to each other
    - A designated initializer must call a designated initializer from its immediate superclass, if it has one
    - A convenience initializer must call another initializer from the same class
    - A convenience initializer must ultimately call a designated initializer

In-Class Exercise

In-Class Exercise

Define a class and use it:

- Class name: Person
- Properties:
    - First name, last name, age
    - Make all optional
    - Make storage private
    - Use computed properties
- Designated and convenience initializers

More Swift

Collection Types

More Swift

Collection types:

- Two primary types:
  - Arrays
  - Dictionaries

More Swift

Collection types - Arrays:

- Contains zero or more elements of the same type

Defining an array:

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

Adding elements to an array:

```
totals.append(35)
```

Accessing elements in an array:

```
println(totals[0])
```

More Swift

Collection types - Dictionaries:

- Stores associations between keys of the same type and values of the same type
- No defined ordering

Defining a dictionary:

```
var totals = [Int: Double]()
```

Adding elements to a dictionary:

```
// 10 and 20 are department numbers
totals[10] = 350.0
totals[20] = 780.0
```

Wrap Up

Wrap Up

That concludes our *very* brief overview of Swift.

Moving forward I’ll be introducing additional aspects of Swift, either because they’d be interesting to learn about and/or they have a direct connection to the primary topic of the lecture.

Next week we’ll create our first iOS app; which will be an example to refer to for homework 2.