ATLS 4120: Mobile Application Development
Week 4: OOP, Model-View-Controller, Swift

Optionals

One of the unique aspects of Swift is the concept of optionals(book page 724, Swift programming language book end of basics section)

- Defining a variable as an optional says it might have a value or it might not
- If it does not have a value it has the value nil
  - Nil is the absence of a value
- Optionals of any type can have the value nil
- A '?' after the type indicates it's an optional

Swift

Go into Xcode
File | New | Playground
Name: swift2
Platform: iOS
Save
Delete what's there so you start with an empty file

```
var score : Int?
print("Score is \(score)")
score=80
print(score)

//force unwrapping without checking if it's nil, don't do this
print("score is \(score!)")
```

To access the value of an optional you must add an '!'. This is called forced unwrapping.
If you force unwrap an optional that is nil your program will crash so you should always check first to find out if an optional has a value before unwrapping it.

```
if score != nil {
    print("The score is \(score!)")
}
```

This is so common in Swift that there's a shorthand for it called optional binding. You can conditionally unwrap an optional and if it contains a value, assigns it to a temporary variable or constant.

```
if let currentScore = score {
    print("My current score is \(currentScore)")
}
```

Make score nil and show that the above binding is false.

Sometimes it's clear that an optional will always have a value and never be nil
We can unwrap these optionals without the need to check it each time
These are called implicitly unwrapped optionals
'!' after the type indicates it's an implicitly unwrapped optional

```
let newScore : Int! = 95
print("My new score is \(newScore)")
```

- No “!” is needed to access the optional because it’s an implicitly unwrapped optional
- Implicitly unwrapped optionals should not be used when there is a possibility of a variable becoming nil at a later point.
- You will see that the variables created as outlet connections are implicitly unwrapped. That’s because once the view is loaded we can be sure that object exists because it’s part of the UI.

If a variable is not defined as an optional, it cannot have the value nil, it must have a value.

```
var finalScore : Int
finalScore = nil //error
```

OOP

- Object-Oriented Programming models the natural way humans think about things in terms of attributes and behavior.
- Object-Oriented Programming (OOP) makes building complex, modular and reusable applications much easier.
- OOP combines data and operations on the data (behavior) into one unit, a class.
- Swift and Java are both object oriented languages

Class

- Models a concept or thing in the real world
- Provides a template, or blueprint for its objects.
- Defines the characteristics (data properties) and behavior (methods) of its objects.
  - The data properties provide the class attributes, or characteristics.
  - Similar to functions, methods are the actions or behaviors of the class
- Defines which parts can be seen outside of the class therefore hiding private information
- Analogy: cookie cutter
- Example:

  Dog class (slide)
  - Breed
  - Age
  - Owner
  - Favorite activity

Objects

- Something that can be acted on
- An instance, or occurrence, of a given class
  - An object of a given class has the structure and behavior defined by the class
  - Many different objects can be defined for a given class
  - All objects of the same class have the same structure
- Properties store data/information
- Methods are actions the object can perform
  - A method should focus on one specific task
  - Just like functions but part of a class
  - Often change the values of properties
  - Instance methods are functions that operate on an object
  - Type methods are functions that operate on the class itself

  In real life an object like a chair has properties such as color, material, seating capacity. And it would have functions like reclining, rocking, or maybe vibrating.

Initialization

- Initialization is the process of preparing an object of a class for use.
- Initializers/Constructors are special methods that instantiate a new object of a class.

Encapsulation

- Modular
  - data properties and behavior are packaged into a single well-defined programming unit
- Information Hiding
  - Methods provide the public interface to objects of a class
  - Implementation is kept private

Inheritance

- Inheritance enables classes to form a hierarchy like a family tree.
- Allows subclasses to share the structure and behavior of its superclass.
  - Superclass is the parent class
  - A subclass extends a class
    - Inherits from the superclass
    - Can add properties and methods
    - Can override a method with a new one
- A super class should be generalized and its subclasses become a more specialized definition
- Inheritance allows you to easily reuse code

Example:

Animal class

- Sex
- Type

- Might make sense to move age to the Animal superclass if all animals have an age

The ViewController class created for us to control our view was a subclass of the UIViewController class provided in the SDK.

Why use OOP?

- iOS and Android are based on a OOP architecture
- Encapsulation: data and methods in a cohesive unit
- Defines a public interface through the methods while the rest of the information is private within the object
- eliminates redundancy because code is reusable
- easier to maintain
- easier to debug

CSCI 5828 Foundations of Software Engineering goes more in depth in OOP

<u>Classes and Structures</u>

Classes and structures are the building blocks in OOP.

Swift classes and structures are much closer in functionality than in other languages, and a lot of the functionality can apply to instances of either a class or a structure type.

- A class/struct provides a template, or blueprint for its objects.
- A class/struct defines the characteristics (data properties) and behavior (methods) of its objects.
  - Properties define the values associated with a class/struct
  - Methods are functions that are associated with a class/struct
- Classes/structs should have UpperCamelCase names to match the capitalization of standard Swift types.

Classes

- Classes are reference types so they are passed by reference, not copied when they are assigned to a variable/constant, or passed to a function.

    - Use automatic reference counting
- Classes support inheritance, structs do not
- Classes support type casting so you can check and interpret the type of an instance

Structs

- Structs and enumerations are value types, so they are copied when assigned or passed
    - Passed by value, not passed by reference
- Structs do not support inheritance

```
class Vehicle {
    var wheelNum = 4
    var speed = 25
    var mpg = 20
    let tankCapacity = 20
    var name : String?
    func changeSpeed(amount: Int){
        speed = speed + amount
    }
    func changeEfficiency(newSpeed: Int, newmpg: Int){
        speed = newSpeed
        mpg = newmpg
    }
}
```

<u>Instances</u>

An instance of a class is traditionally known as an object. Because Swift classes and structures are much closer in functionality than in other languages, the more general term instance is used.

- An instance is an occurrence of a given class or struct that is ready to use
    - An instance of a given class/struct has the structure and behavior defined by the class/struct
    - Many different instances can be defined for a given class or struct
    - All instances of the same class or struct have the same structure
- Create an instance of a class or struct by calling an initializer method.
- The simplest initializer syntax is () which calls the default init()
- Initializers make sure that every stored property has a value when initialization completes
- Properties are accessed using dot notation

```
let myJeep = Vehicle()
myJeep.mpg
myJeep.speed
myJeep.name
```

Name is allowed to be nil because it's an optional

<u>Initialization</u>

Initialization is the process of preparing an instance of a class or struct for use.

- A class/struct includes methods used to create and initialize a new instance of a class called initializers.
    - They ensure that the new instance is correctly initialized before they are used the first time
- During initialization an initial value must be set for each stored property on that instance
    - Default value
    - Initial value

- Swift provides a default initializer called init() that has default values for all its properties
- You can also create your own **init()** methods to provide initial values that don't have defaults
- As with functions and methods, Swift initializers require an external and internal parameter name.
  - If only one name is given it's used for both
  - Use an _ if you explicitly don't want an external parameter name

[Update Vehicle class]
```
    init(vehicleName vname: String){
        name = vname
    }
```

But now you get errors where you defined myJeep because there's no empty init() method. So you must add one to the class.
```
    init(){
    }
```

```
let myHybrid = Vehicle(vehicleName: "Prius")
myHybrid.name
```

Check that name isn't nil and then force unwrap it
```
if myHybrid.name != nil {
   println(myHybrid.name!)
}
```

<u>Methods</u>
- Instance methods are functions that belong to a class/struct
  - Same syntax as functions
  - Call methods with the same dot notation as properties
- Like functions, method parameters must have a both a external and internal name
  - If only one name is given it's used for both
  - Use an _ if you explicitly don't want an external parameter name
- Method headers in the documentation will show the first parameter as _: if it doesn't have an external name

```
myJeep.changeSpeed(amount: 10)
myJeep.speed
myJeep.changeEfficiency(newSpeed: 35, newmpg: 25)
myJeep.speed
myJeep.mpg
```

Swift also has type methods
- functions that are called on the class itself
- Use the keyword 'static' before the function

<u>Inheritance</u>
- Inheritance enables classes to form a class hierarchy like a family tree.
- Allows subclasses to share the structure and behavior of its superclass.
  - Superclass is the parent class
  - A subclass extends a class

    - Inherits from the superclass
    - Can add properties and methods
    - Can modify inherited properties
- A subclass can provide its own custom implementation of methods or properties through overriding
  - Prefix your overriding definition with the keyword ‘override’
  - Overriding says that you intend to provide an override, not that you’re providing a matching definition by accident
- When creating an initializer in a subclass, set your own properties and then you must call the superclass’s initializer **super.init()**

```
class Bicycle : Vehicle {
    var reflectors = true
}

var bike=Bicycle()
bike.wheelNum
bike.wheelNum = 2
bike.wheelNum
```

`bike.reflectors` is true
`myJeep.reflectors` error – class Vehicle doesn’t have a reflectors property

change Bicycle
`var reflectors : Bool`
Get errors because reflectors doesn’t have a value.

```
    init(_ ref : Bool){
        reflectors=ref
        super.init()
    }
```

super.init() calls the superclass,Vehicle, init()

`var bike=Bicycle(false)`
note there’s no named parameter because we used _ in the header of the init method.

<u>Class or struct</u>
Classes and structures have a lot of similar functionality but the main difference is that structure instances are always passed by value, and class instances are always passed by reference. This means that they are suited to different kinds of tasks.
As a general guideline, consider creating a structure when one or more of these conditions apply:

- The structure’s primary purpose is to encapsulate a few relatively simple data values.
- It is reasonable to expect that the encapsulated values will be copied rather than referenced when you assign or pass around an instance of that structure.
- Any properties stored by the structure are themselves value types, which would also be expected to be copied rather than referenced.
- The structure does not need to inherit properties or behavior from another existing type.

In all other cases, define a class, and create instances of that class. In practice, this means that most custom data constructs should be classes, not structures.

```
//classes are reference types
var new_bike = bike
new_bike.reflectors
new_bike.reflectors = true
new_bike.reflectors
bike.reflectors

//structs are value types
struct Skateboard {
    var color : String
    var type : String
}

var board = Skateboard(color: "black", type: "longboard")
board.color
var new_board = board
new_board.color
new_board.color = "purple"
new_board.color
board.color
```

<u>Collection Types</u>
Swift has three types of collections

- Arrays
  - ordered collections of values
- Sets
  - unordered collections of distinct values
- Dictionaries
  - unordered collections of key/value pairs

The collection will be mutable if it's assigned to a variable, immutable if it's assigned to a constant
Properties

- .count returns the number of items in an array
- .isEmpty is a boolean to see if count is 0

In Swift, many basic data types such as String, Array, and Dictionary are implemented as structures. Since structs are value types, strings, arrays, and dictionaries are passed by value and copied when they are assigned to a new constant or variable, or when they are passed to a function or method.

<u>Arrays</u>
Arrays store an ordered collection of values
Arrays start with an index of 0 just as in other languages

- insert(_:at:) inserts an item into the array at a specified index
- remove(at:) and removeLast() return the removed item

```
var myList=[String]()
var shoppingList=["cereal", "milk"]
print(shoppingList[0])
shoppingList.append("bread")
```

```
if shoppingList.isEmpty{
    print("there's nothing you need")
} else {
    print("You need \(shoppingList.count)" + " items")
}

let item = shoppingList.removeLast()
print("\(shoppingList.count)")

shoppingList.insert("coffee", at:0)
let olditem=shoppingList.remove(at: 1)
```

<u>Dictionaries</u>

Dictionaries store unordered key/value data pairs

- **.keys** returns all the keys
- **.values** returns all the values
- updateValue(_:forKey:) returns the old value for that key
- removeValue(forKey:) returns the removed value or nil if no value existed

```
var newList=[String:String]()
var classes:[String: String]=["4120":"MAD", "2200":"Web"]
```

As with arrays, you don't have to write the type of the dictionary if you're initializing it with a dictionary literal whose keys and values have consistent types.

```
classes["4120"]
classes["2000"]="MIT"
classes.count

classes.updateValue("Mobile App Dev", forKey: "4120")
classes["4120"]
classes.removeValue(forKey: "2200")
classes.count
```

<u>Memory Management</u>

Swift uses Automatic Reference Counting (ARC) to manage memory usage
ARC automatically frees up the memory used by class instances when those instances are no longer needed
Automatic Reference Counting (ARC) was introduced in iOS5 so you don't have to worry about it. Yay!

Model-View-Controller(MVC)

The MVC architecture is a common design pattern and is used in both iOS and Android (slide)

- Model: holds the data and classes
  - Should be UI independent
- View: all items for the user interface (objects in IB)
- Controller: links the model and the view together. The backbone or brain of the app.
  - usually subclasses from UI frameworks that will allow the view and the model to interact
  - Controllers are usually paired with a single view, that's where ViewController comes from

- The goal of MVC is to have any object be in only one of these categories.
  - These categories should never overlap
  - You should be able to change the model and not have to change the UI, and vice versa
- Ensures reusability