iOS Development with Swift

Classes & Structures

Week *4*

Outline

- Classes & Structures
- Properties
- Methods
- Inheritance
- Initialization
- Deinitialization
- Workshop

Classes and Structures

Classes and structures are general-purpose, flexible constructs that become the building blocks of your program's code.

You define properties and methods to add functionality to your classes and structures by using exactly the same syntax as for constants, variables, and functions.

Classes and Structures Cont'd

Classes and structures in Swift have many things in common:

- Define properties to store values
- Define methods to provide functionality
- Define subscripts to provide access to their values using subscript syntax
- Define initializers to set up their initial state
- Be extended to expand their functionality beyond a default implementation

Classes and Structures Cont'd

Classes have additional capabilities that structures do not:

- Inheritance enables one class to inherit the characteristics of another.
- Type casting enables you to check and interpret the type of a class instance at runtime.
- Deinitializers enable an instance of a class to free up any resources it has assigned.
- Reference counting allows more than one reference to a class instance.

Definitions

Classes and structures have a similar definition syntax. You introduce classes with the **class** keyword and structures with the **struct** keyword.

```
class SomeClass {
    // class definition goes here
}
struct SomeStructure {
    // structure definition goes here
}
```

Properties

Properties associate values with a particular class, structure, or enumeration. Any value can be stored in a particular class or structure.

```
class Point {
    var x : Int
    var y : Int
}
```

Properties

Properties can be assigned and retrieved by special keywords which are **get** and **set**

```
class Point {
    var x : Int{
        get{
            return self.x
        }

        set(newValue){
            self.x = newValue
        }
    }
    var y : Int{
        get{
            return self.y
        }

        set(newValue){
            self.y = newValue
        }
    }

}
```

Methods

Methods are functions that are associated with a particular type. Classes, structures, and enumerations can all define instance methods, which encapsulate specific tasks and functionality for working with an instance of a given type

```
class Shape{
    func draw(){
        print("draw a shape")
    }
}
```

Inheritance

A class can inherit methods, properties, and other characteristics from another class. When one class inherits from another, the inheriting class is known as a subclass, and the class it inherits from is known as its superclass.

```
class Shape{

}

class Triangle : Shape{

}

class Square : Shape{

}
```

Overriding

A subclass can provide its own custom implementation of an instance method, type method, instance property, type property, or subscript that it would otherwise inherit from a superclass. This is known as overriding.

```
class Shape{

    init() {
        print("Shape:init")
    }

    func draw(){
        print("draw a shape")
    }
}

class Triangle : Shape{
    override init() {
        super.init()
        print("Triangle:init")
    }

    override func draw() {
        print("draw a triangle")
    }
}
```

Initialisation

Initialization is the process of preparing an instance of a class, structure, or enumeration for use.

```
struct Fahrenheit {
    var temperature: Double
    init() {
        temperature = 32.0
    }
}
var f = Fahrenheit()
print("The default temperature is \(f.temperature)° Fahrenheit")
// Prints "The default temperature is 32.0° Fahrenheit"
```

Deinitialization

A deinitializer is called immediately before a class instance is deallocated. You write deinitializers with the **deinit** keyword.

```
deinit {
    // perform the deinitialization
}
```

Automatic Reference Counting

Swift uses Automatic Reference Counting (ARC) to track and manage your app's memory usage. In most cases, this means that memory management "just works" in Swift, and you do not need to think about memory management yourself. ARC automatically frees up the memory used by class instances when those instances are no longer needed.

Access Controls

- **Open** and **Public** enable entities to be used within any source file from their defining module, and also in a source file from another module that imports the defining module.
- **Internal** entities to be used within any source file from their defining module, but not in any source file outside of that module.
- **File-private** access restricts the use of an entity to its own defining source file.
- **Private** restricts the use of an entity to the enclosing declaration.

Workshop - 1

Question: Create a class named **Student**

- Student will have unique **id**, **firstname** and **lastname** attributes.
- Write an initialisation for **Student** class.
- Write a method named **info** that prints out student info.
- Create two different classes named **CollegeStudent** and **UniversityStudent** which inherit **Student** class.
- Create 5 different student objects and call **info** method on each object.

Workshop - 2

Question: Create a class named **Animal**

- Create new class named **Cats** inherited from **Animal** class.
- Write a function in cats named **roar** that prints a string.
- Create a **Tiger** class and override **roar** method and print **tiger**.
- Create a **Jaguar** class and override **roar** method and print **jaguar**.
- Create a **DomesticCat** class and override **roar** method that prints **meow**.

Workshop - 3

Question: Create a class named **List**

- Class will have a property named **elements** with array of integers.
- Write a method named **append** that adds new integer to to **elements**.
- Write a method named **remove** that removes an integer from **elements** at given index.
- Write a method named **first** that returns first element of **elements.**
- Write a method named **count** that returns number of elements in the array.