Introduction to Swift & SwiftUI

Girish Lukka

Topics to be covered

- Classes
- Structs
- Enums
- Functions
- Memory Management
- Live coding – complete week1 example

Classes

- In Swift we can use classes to build custom and complex data types.
- Classes are similar to structs but are quite different in behaviour.
- No automatic memberwise initializer for your classes - need to write one.
- There is inheritance.
- A class is a blueprint for that object
- A copy of the object points to the same data by default – change one, and the copy changes too.
- An object is simply a collection of data (variables) and methods (functions).

Creating Classes – refer to playground file

```
1
2  import Foundation
3
4  class Person{
5      var clothes: String
6      var shoes: String
7  }
8
```

```
expression failed to parse:
error: ClassPH_1.xcplaygroundpage:4:7: error: class 'Person' has no initializers
class Person{
      ^
```

The properties clothes and shoes must have values.
Solution:

1. Make the values optional – messy propagation of optionals
2. Give them default values – may never be used
3. Write an initializer init() that manges the 2 properties – java class constructor

Classes Examples– Additional material P.Hudson

```swift
import Foundation
class Enemy{
    var health = 100
    var attackStrength = 10

    func move(){
        print("Walk forwards")
    }

    func attack(){

        print("Made a hit, \(attackStrength) damage")
    }
}

let enemy1 = Enemy()
print(enemy1.health)
enemy1.move()
enemy1.attack()
let enemy2  = Enemy()
```

```
100
Walk forwards
Made a hit, 10 damage
```

```swift
//inheritance

class BigEnemy : Enemy {
    var wingSpan = 2

    func talk(speech: String){
        print("Said: \(speech)")
    }
    override func move(){
        print("run fast")
    }

    override func attack(){
        super.attack()
        print("jumps, damages by 10")
    }
}
let bigEnemy = BigEnemy()
bigEnemy.wingSpan = 5
bigEnemy.attackStrength = 15
print(bigEnemy.health)
print(bigEnemy.attackStrength)
bigEnemy.talk(speech:"Hello")
bigEnemy.move()
bigEnemy.attack()
```

Classes Example demonstrate passing by reference

```
21  class Enemy1{
23      var health: Int
24      var attackStrength : Int
25
26      init(health: Int, attackStrength: Int){
27          self.health = health
28          self.attackStrength = attackStrength
29      }
30
31      func move(){
32          print("Walk forwards")
33      }
34
35      func attack(){
36
37          print("Made a hit, \(attackStrength) damage")
38      }
39
40      func takeDamage(amount: Int){
41          health = health - amount
42      }
43  }
44  let enemy1 = Enemy1(health: 10, attackStrength: 20)
45  print(enemy1.health)
46  enemy1.takeDamage(amount: 5)
47  print(enemy1.health)
48  let enemy2 = enemy1
49  print(enemy2.health)
50  // passing by reference updates for all objects.
```

```
10
5
5
```

Classes Example – effect of let?

```
2
3   class ClassHero{
4       var name: String
5       var location: String
6
7       init(name:String, location: String){
8           self.name = name
9           self.location = location
10      }
11  }
12  let classHero = ClassHero(name: "Superman", location: "Mars")
13  print(classHero.name)
14  classHero.name = "Cat Woman"
15  print(classHero.name)
16
```

```
Superman
Cat Woman
```

Structures aka Structs

Can use structs to build custom and complex data types.

Automatic generation of initializer, however, quite useful to define one.

- A struct is a blueprint for object creation
- A copy of the object is a new object – change anyone, the others unaffected, unlike class objects.
- Value type, copies the object whenever it's passed around.
- No inheritance, only encapsulation.
- Faster than classes as it doesn't require reference counting for memory management.

Struct Examples

```swift
import Foundation

struct Town{
    let name: String
    var people: [String]
    var items: [String:Int]

    func newRoad(){
        print("New road to be built")
    }
}

var myTown = Town(name: String, people: [String], items: [String : Int])

```

Struct Examples

Week2Lecture 〉 StructsDeemo_0

```swift
import Foundation

struct Town{
    let name: String
    var people: [String]
    var items: [String:Int]

    func newRoad(){
        print("New road to be built")
    }
}

var myTown = Town(name: "MyPlace", people: ["Girish","girish"], items: ["laptops": 2, "consoles": 3])

print(myTown) // shows all the properties and the values
print("\(myTown.name) has \(myTown.items["laptops"]!): laptops")

myTown.people.append("GIRISH")
print(myTown.people.count)
myTown.newRoad()
```

```
Town(name: "MyPlace", people: ["Girish", "girish"], items: ["consoles": 3, "laptops": 2])
MyPlace has 2: laptops
3
New road to be built
```

Struct examples

```swift
import Foundation

struct Town{
    let name = "NewLands"
    var people = ["Alex", "Ali"]
    var items = ["Houses": 5, "Cars": 2, "Shops" : 1]

    func newRoad(){
        print("New road to be built")
    }
}

var myTown = Town()

print(myTown) // shows all the properties and the values
print("\(myTown.name) has \(myTown.items["Houses"]!) houses")

myTown.people.append("Amar")
print(myTown.people.count)
myTown.newRoad()
```

```
Town(name: "NewLands", people: ["Alex", "Ali"], items: ["Cars": 2, "Houses": 5, "Shops": 1])
NewLands has 5 houses
3
New road to be built
```

Struct examples

Week2Lecture 〉 StructsDemo3

```swift
import Foundation

struct Town{
    let name : String
    var people : [String]
    var items : [String: Int]

    init(newName: String, newPeople: [String], newItems: [String : Int])

    {
        name = newName
        people = newPeople
        items = newItems
    }

    func newRoad(){
        print("New road to be built")
    }
}
var newTown = Town(newName: "NewLands", newPeople: ["Tom", "Mark"], newItems:
    ["Homes" : 5, "Shops" : 2])
print(newTown)
newTown.people.append("Alex")
print(newTown.people)
```

```
Town(name: "NewLands", people: ["Tom", "Mark"], items: ["Shops": 2, "Homes": 5])
["Tom", "Mark", "Alex"]
Town(name: "NewLands", people: ["Tom", "Mark", "Alex"], items: ["Shops": 2, "Homes": 5])
Town(name: "NewLands", people: [], items: ["Shops": 2, "Homes": 5])
["Tom", "Mark", "Alex"]
```

Struct examples

Week2Lecture › StructsDemo2

```swift
import Foundation

struct Town{
    let name : String
    var people : [String]
    var items : [String: Int]

    init(name: String, people: [String], items: [String : Int])
    {
        self.name = name
        self.people = people
        self.items  = items
    }
    func newRoad(){
        print("New road to be built")
    }
}

var newTown = Town(name: "NewLands", people: ["Tom", "Mark"], items: ["Homes" : 5,
    "Shops" : 2])
print(newTown)
newTown.people.append("Alex")
print(newTown.people)

var oldTown = newTown
print(oldTown)
oldTown.people = []
print(oldTown)
print(newTown.people)
```

```
Town(name: "NewLands", people: ["Tom", "Mark"], items: ["Shops": 2, "Homes": 5])
["Tom", "Mark", "Alex"]
Town(name: "NewLands", people: ["Tom", "Mark", "Alex"], items: ["Shops": 2, "Homes": 5])
Town(name: "NewLands", people: [], items: ["Shops": 2, "Homes": 5])
["Tom", "Mark", "Alex"]
```

Struct examples mutating method

```
import Foundation

struct Town{
    let name : String
    var people : [String]
    var items : [String: Int]

    init(name: String, people: [String], items: [String : Int])
    {
        self.name = name
        self.people = people
        self.items  = items
    }
    func newRoad(){
        print("New road to be built")
    }

     mutating func addItems(){
        items["Hotel"] = 1
    }
}

var newTown = Town(name: "NewLands", people: ["Tom", "Mark"], items: ["Homes" : 5,
    "Shops" : 2])
print(newTown)
newTown.people.append("Alex") // change from outside the struct
print(newTown.people)
newTown.addItems()
print(newTown)
```

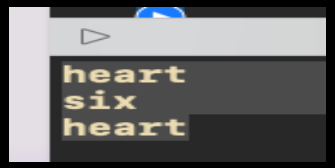
```
Town(name: "NewLands", people: ["Tom", "Mark"], items: ["Homes": 5, "Shops": 2])
["Tom", "Mark", "Alex"]
Town(name: "NewLands", people: ["Tom", "Mark", "Alex"], items: ["Hotel": 1, "Homes": 5, "Shops": 2])
```

Enums example

```swift
import Foundation

struct Card {

    let rank: Rank
    let suit: Suit

    enum Rank {
        case two, three, four, five, six, seven, eigth, nine, ten, jack,
            queen, king, ace
    }

    enum Suit {
        case heart, diamond, club, spades
    }
}

struct PokerHand {
    let cards: [Card]
}
var myCard = Card(rank: .five, suit: .heart)
var myHand = PokerHand(cards: [Card(rank: .six, suit: .heart), Card(rank:
    .seven, suit: .club)])

print(myCard.suit)
print(myHand.cards[0].rank)
print(myHand.cards[0].suit)
```

Card
PokerHand

"heart\n"
"six\n"
"heart\n"



How to choose between a class, struct and enum – Apple guidelines

1. Use a struct for value types: If you want to represent a simple data structure with a few properties, a struct is the way to go. Structs are value types, which means they are copied when they are passed around in your code. This makes them suitable for small data structures that don't need to be shared between multiple instances.
2. Use a class for reference types: If you want to represent an object that can be shared between multiple instances or passed around as a reference, use a class. Classes are reference types, which means that multiple instances can reference the same object in memory.
3. Use enums for simple cases: If you want to represent a fixed set of related values, use an enum. Enums are especially useful for representing a small, finite set of values, such as the different states of a button (e.g. pressed, released).

Functions

- There are five parts to a function:

1. The keyword func. This keyword signifies the start of a function definition.
2. The name of the function.
3. The parameters (named or un-named)
4. The body of the function.
5. The return type followed by a -> eg: -> String

Function Examples – refer to playground file

```swift
import Foundation

//1. Basic Function Call:
func greet(name: String) {

    print("Hi \(name)!")
}
//  call the function like this:
greet(name: "Girish")

//2. Function with Multiple Arguments:
func getFullName(firstName: String, lastName: String) ->  String {
    return "\(firstName) \(lastName)"
}
//call function like this:
print(getFullName(firstName: "girish", lastName: "lukka"))

//3. External Parameter Names:
func greeting(_ firstName: String, _ lastName: String) -> String{
    return ("Hello \(firstName) \(lastName)")
}
// call function like this:
print(greeting("girish", "lukka"))

//4. Variadic Parameters:
func average(numbers: Double...) -> Double {
    let sum = numbers.reduce(0, +)
    print(sum)
    return sum / Double(numbers.count)
}
```

Function Examples inout

```
3   import Foundation
4   func doubleInPlace(number: inout Int) {
5       number *= 2
6   }
7   var myNum = 10
8   print(myNum)
9   doubleInPlace(number: &myNum)
10  print(myNum)
```

```
10
20
```

Nested Functions

```
func calculateMonthlyPayments(carPrice: Double, downPayment: Double,
                              interestRate: Double, paymentTerm: Double) -> Double {
    func loanAmount() -> Double {
        return carPrice - downPayment
    }
    func totalInterest() -> Double {
        return interestRate * paymentTerm
    }
    func numberOfMonths() -> Double {
        return paymentTerm * 12
    }
    return ((loanAmount() + ( loanAmount() *
                              totalInterest() / 100 )) / numberOfMonths())
}


let monthlyPayment = calculateMonthlyPayments(carPrice: 50000, downPayment: 5000,
    interestRate: 3.5,paymentTerm: 7.0)

print(monthlyPayment)
```

(2 times)

24.5

84

666.9642857..

666.964285742857

"666.964285...

Mutating functions - BMI

```swift
import Foundation

struct BMIData {
    var height = 0.0
    var weight  = 0.0
    var BMI = 0.0

    mutating func calBMI(){

        let bmiValue = weight / (height * height)
        self = BMIData(height: height, weight: weight, BMI:bmiValue)
        print("self... \(self)")

    }
}
var bmi = BMIData(height:1.67, weight: 67, BMI: 0.0)
print(bmi)
bmi.calBMI()
print(bmi)
```

```
BMIData(height: 1.67, weight: 67.0, BMI: 0.0)
self... BMIData(height: 1.67, weight: 67.0, BMI: 24.023808670084982)
BMIData(height: 1.67, weight: 67.0, BMI: 24.023808670084982)
```

Mutating functions - BMI

```
22  struct BMI2{
23      var height : Double
24      var weight : Double
25      var BMI = 0.0
26
27      init(height: Double, weight: Double){
28
29          self.height = height
30          self.weight = weight
31      }
32
33      mutating func calBMI2(){
34
35          let bmiValue = weight / (height * height)
36
37          self.BMI = bmiValue
38      }
39  }
40
41  var bmi2 = BMI2(height: 1.67, weight: 67)
42
43  bmi2.calBMI2()
44  print(bmi2.BMI)
```

```
24.02380867
BMI2
BMI2
BMI2
"24.0238086
```

Automatic Reference Counting - ARC

Swift uses *Automatic Reference Counting* (ARC) to track and manage app's memory usage. No garbage collector, like in Java and Python.

See playground example..

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) is being deinitialized") }
}

class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") }
}
```