Chapter X

Swift Programming Part II

(Advance Swift Programming Topic)

9. Swift Programming Part II

1. Class

I. Introduction

In object-oriented programming, a class is an extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behaviour (member functions or methods).Classes are composed from structural and behavioural constituents.
The structure defined by the class determines the layout of the memory used by its instances.The behaviour of class or its instances is defined using methods. Methods are subroutines with the ability to operate on objects or classes. Some types of methods are created and called by programmer code, while other special methods —such as constructors, destructors, and conversion operators—are created and called by compiler-generated code. A language may also allow the programmer to define and call these special methods.

eg.
class <classname> {
Definition 1
Definition 2

Definition N
}

eg.
class student{
var studname: String = "Alex"
var mark: Int = 80
var mark2: Int = 76
}

eg.
```
class Shape {
   var numberOfSides = 0

   func simpleDescription() -> String {
      return "A shape with \(numberOfSides) sides."
   }

}
```

The syntax for creating instances

let stuRecord = student()
var triangle = Shape( )

And access the properties and methods as of objects as follow:

stuRecord.studname

triangle.numberofSides = 3
triangle.simpleDescription( )

II.Class with Initialiser

First version of the Shape class is missing something important: an initializer to set up the class when an instance is created. Use init to create one.

```
class NamedShape {
    var numberOfSides: Int = 0
    var name: String

    init(name: String) {
        self.name = name //****
    }

    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

To Create Instance:

```
var shape = NamedShape("square")
print(shape.name)
```

Note:
self is used to distinguish the name property from the name argument to the initializer. The arguments to the initializer are passed like a function call when you create an instance of the class. Every property needs a value assigned—either in its declaration (as with numberOfSides) or in the initializer (as with name).And also Use deinit to create a deinitializer if you need to perform some cleanup before the object is deallocated.

III.Sub Classing and Inheritance:

The act of basing a new class on an existing class is defined as 'Subclass'. The subclass inherits the properties, methods and functions of its base class. To define a subclass ':' is used before the base class name.

eg.

```
class Square: NamedShape { // Now Inheritance from class NamedShape
    var sideLength: Double          //New variable, Not in Super Class

    init(sideLength: Double, name: String) {
        self.sideLength = sideLength
        super.init(name: name)
        numberOfSides = 4
    }

    func area() ->  Double {
        return sideLength * sideLength
    }

    override func simpleDescription() -> String {
        return "A square with sides of length \(sideLength)."
```

```
    }
}
let test = Square(sideLength: 5.2, name: "my test square")
test.area()
test.simpleDescription()”
```

Note:
'super' keyword is used as a prefix to access the methods, properties and subscripts declared in the super class. Subclass provides the concept of overriding. 'override' keyword is used to override the methods declared in the superclass. And 'final' keyword is used with properties and method in the parent class to prevent overriding throughout the child hierarchy

Classes in Swift refers multiple constants and variables pointing to a single instance. To know about the constants and variables pointing to a particular class instance identity operators are used. Class instances are always passed by reference. In Classes NSString, NSArray, and NSDictionary instances are always assigned and passed around as a reference to an existing instance, rather than as a copy.

eg.

Identical to Operators	Not Identical to Operators
Operator used is (===)	Operator used is (!==)

2. Struct & Enumeration

Use struct to create a structure. Structures support many of the same behaviors as classes, including methods and initializers. One of the most important differences between structures and classes is that structures are always copied when they are passed around in your code, but classes are passed by reference.

Use enum to create an enumeration. Like classes and all other named types, enumerations can have methods associated with them.

e.g

```
enum Rank: Int {
    case Ace = 1
    case Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten
    case Jack, Queen, King
    func simpleDescription() -> String {
        switch self {
        case .Ace:
            return "ace"
        case .Jack:
            return "jack"
        case .Queen:
            return "queen"
        case .King:
            return "king"
        default:
            return String(self.rawValue)
        }
    }
```

```
}
let ace = Rank.Ace
let aceRawValue = ace.rawValue
```

1. Swift Programming Check List

Variable

Types

(Int, Float, Double, Bool, String, Tuples,

UInt8, UInt16, UInt32, UInt64)

Characters, printing out characters from String

Array Operation

count, isEmpty, += [value1, value2] ,

array[1…3] =[value1,value2,value3]

insert(value , at: IndexNumber)

remove(at: IndexNumber), removeLast()

popLast(),

compoundArray with +

reversed()

Dictionary Operation

Iterating over the whole dictionary or keys or values

removeValue(forKey: **)**

Conversion

scope of variable

private

fileprivate , internal

public

Basic Operators

+, - , / , % , ! , == , != ,> , <, <= , >= , << , >>, | , &

&& , || , ??

Conditional

- if , if then -else if -else , guard
- switch

Looping

- while, repeat ,
- for
  - for in loop
  - reverse for in loop
  - stride

String

- characters
- comparing

Array

- Getting value
- Adding value to array
- Copy behaviour
- Mutability

Function

- with no parameter with no return value
- with parameter with no return value
- with no parameter with return value
- with multiple parameters with multiple return value
- external parameter name and internal parameter name
- default parameter value
- In-Out parameter
- Recursion

2. Try to think following condition and write down the solution in

Swift Playground

1. Able to identify maximum number of two numbers?
2. Able to identify maximum number of numbers more than two? and Average ?
3. Abe to identify a given number as odd or even?
4. Able to Decide a point (x , y) is within a given rectangle (x, y , w , h)?
5. Try to print rectangle of * , composed by 4 stars of 6 rows using for loop by printing only ONE star

3. Extension

Extensions add new functionality to an existing class, structure, enumeration, or protocol type. This includes the ability to extend types for which you do not have access to the original source code

Extensions in Swift can:

- Add computed instance properties and computed type properties
- Define instance methods and type methods
- Provide new initializers
- Define subscripts
- Define and use new nested types
- Make an existing type conform to a protocol

Extension Syntax:
Declare extensions with the extension keyword:

```
extension SomeType {
   // new functionality to add to SomeType
}
```

An extension can extend an existing type to make it adopt one or more protocols. To add protocol conformance, you write the protocol names the same way as you write them for a class or structure:

```
extension SomeType: SomeProtocol, AnotherProtocol {
   // implementation of protocol requirements
}
```

4. Protocol

A *protocol* defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality. The protocol can then be *adopted* by a class, structure, or enumeration to provide an actual implementation of those requirements. Any type that satisfies the requirements of a protocol is said to *conform* to that protocol.(protocol are extensible using extension).

Protocol Syntax

```
protocol SomeProtocol {
        . . .
}

struct SomeStructure: FirstProtocol, AnotherProtocol {
      . . .
}
class SomeClass: SomeSuperclass, FirstProtocol,
                    AnotherProtocol {
      . . .
}
```

1. Property Requirements

A protocol can require any conforming type to provide an instance property or type property with a particular name and type. The protocol doesn't specify whether the property should be a stored property or a computed property—it only specifies the required property name and type. The protocol also specifies whether each property must be gettable or gettable and settable.

```
protocol SomeProtocol {
   var mustBeSettable: Int { get set }
```

```
    var doesNotNeedToBeSettable: Int { get }
}
protocol AnotherProtocol {
    static var someTypeProperty: Int { get set }
}
//For single instance variable
protocol FullyNamed {
    var fullName: String { get }
}
```

2. Method Requirements

Protocols can require specific instance methods and type methods to be implemented by conforming types. These methods are written as part of the protocol's definition in exactly the same way as for normal instance and type methods, but without curly braces or a method body.

```
protocol SomeProtocol {
    static func someTypeMethod()
}
protocol RandomNumberGenerator {
    func random() -> Double
}
```

3. Mutating method requirement

The toggle() method is marked with the mutating keyword as part of the Togglable protocol definition, to indicate that the method is expected to mutate the state of a conforming instance when it's called:

For mutating methods, put the mutating keyword before a method's func keyword to indicate that the method is allowed to modify the instance it belongs to and any properties of that instance.

4. Initializer requirement

Likewise, we can set for initializer, init function for initialization with certain parameters.

eg.

```
protocol SomeProtocol {
   init(someParameter: Int)
}
//Note. Implement that init as Designated or Convenience initializer

//Usage
class SomeClass: SomeProtocol {
   required init(someParameter: Int) {.
            //MUST Be mark with required but not for final modifier
            // initializer implementation goes here
   }
}
```

If a subclass overrides a designated initializer from a superclass, and also implements a matching initializer requirement from a protocol, mark the initializer implementation with both the required and override modifiers:

```
protocol SomeProtocol {
    init()
}
class SomeSuperClass {
    init() {
        // initializer implementation goes here
    }
}
class SomeSubClass: SomeSuperClass, SomeProtocol {
    // "required" from SomeProtocol conformance; "override" from
SomeSuperClass
    required override init() {.
```

```
            . . .
    }
}
```

Note: Failable Initializer Requirements: Protocols can define failable initializer requirements for conforming types, as defined in Failable Initializers.

A failable initializer requirement can be satisfied by a failable or nonfailable initializer on a conforming type. A nonfailable initializer requirement can be satisfied by a nonfailable initializer or an implicitly unwrapped failable initializer.

5. Protocols as Types

any protocol you create will become a fully-fledged type for use in your code. As follow:

- As a parameter type or return type in a function, method, or initializer
- As the type of a constant, variable, or property
- As the type of items in an array, dictionary, or other container

Eg

6. Delegation

Delegation is a design pattern that enables a class or structure to hand off (or *delegate*) some of its responsibilities to an instance of another type. This design pattern is implemented by defining a protocol that encapsulates the delegated responsibilities, such that a conforming type (known as a delegate) is guaranteed to provide the functionality that has been delegated. Delegation can be used to respond to a particular action, or to retrieve data from an external source without needing to know the underlying type of that source.

The example below defines two protocols for use with dice-based board games:

```
protocol DiceGame {
    var dice: Dice { get }
    func play()
}
protocol DiceGameDelegate: AnyObject {
```

- func gameDidStart(_ game: DiceGame)
- func game(_ game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int)
- func gameDidEnd(_ game: DiceGame)
- }

The DiceGame protocol is a protocol that can be adopted by any game that involves dice. Then here is implementation adapting to DiceGame Protocol

```
class SnakesAndLadders: DiceGame {
    let finalSquare = 25
    let dice = Dice(sides: 6, generator: LinearCongruentialGenerator())
    var square = 0
    var board: [Int]
    init() {
        board = Array(repeating: 0, count: finalSquare + 1)
        board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
        board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
    }
    weak var delegate: DiceGameDelegate?
    func play() {
        square = 0
        delegate?.gameDidStart(self)
        gameLoop: while square != finalSquare {
            let diceRoll = dice.roll()
            delegate?.game(self, didStartNewTurnWithDiceRoll: diceRoll)
            switch square + diceRoll {
            case finalSquare:
                break gameLoop
            case let newSquare where newSquare > finalSquare:
                continue gameLoop
            default:
                square += diceRoll
                square += board[square]
            }
```

```
        }
        delegate?.gameDidEnd(self)
    }
}
```

Note: we can adapt to protocol with extension and protocol can be treat in collections too.

eg.

```
struct Hamster {
    var name: String
    var textualDescription: String {
        return "A hamster named \(name)"
    }
}
extension Hamster: TextRepresentable {}
let things: [TextRepresentable] = [game, d12, simonTheHamster]
```

7. Protocol Inheritance

A protocol can inherit one or more other protocols and can add further requirements on top of the requirements it inherits. The syntax for protocol inheritance is similar to the syntax for class inheritance, but with the option to list multiple inherited protocols, separated by commas:

```
protocol InheritingProtocol: SomeProtocol, AnotherProtocol {
    // protocol definition goes here
}
```

8. Class-Only Protocols

You can limit protocol adoption to class types (and not structures or enumerations) by adding the AnyObject protocol to a protocol's inheritance list.

```
protocol SomeClassOnlyProtocol: AnyObject, SomeInheritedProtocol {
    // class-only protocol definition goes here
}
```

Note:You can use the is and as operators described in Type Casting to check for protocol conformance

9. Optional Protocol Requirements

Both the protocol and the optional requirement must be marked with the @objc attribute. Note that @objc protocols can be adopted only by classes that inherit from Objective-C classes or other @objc classes. They can't be adopted by structures or enumerations.

```
@objc protocol CounterDataSource {
    @objc optional func increment(forCount count: Int) -> Int
    @objc optional var fixedIncrement: Int { get }
}
```

5. Subscripts

Classes, structures, and enumerations can define *subscripts*, which are shortcuts for accessing the member elements of a collection, list, or sequence. You use subscripts to set and retrieve values by index without needing separate methods for setting and retrieval. Subscripts can take any number of input parameters, and these input parameters can be of any type. Subscripts can also return any type. Subscripts can use variadic parameters, but they can't use in-out parameters or provide default parameter values.

Subscript Syntax

```
  subscript(index: Int) -> Int {
      get {
          // return an appropriate subscript value here
      }
      set(newValue) {
          // perform a suitable setting action here
      }
}
subscript(index: Int) -> Int {
   // return an appropriate subscript value here
}
```

Eg.

6. Generics

Generic code enables you to write flexible, reusable functions and types that can work with **any type**, subject to requirements that you define. You can write code that **avoids duplication** and expresses its intent in a clear, abstracted manner.

1. Generic Function

Eg: for specific type:

Eg.2

2. Generic function with multiple parameters

By adding additional place holder type , says: 'T' between < >, we can pass more parameters to generic function.

Such :

```
func anotherMethod<T, U> (first: T, second: U) {
        print("\(T) & \(U)")
}
```

3. Generic Type:

Swift enables you to define your own generic types. These are custom classes, structures, and enumerations that can work with any type, in a similar way

to Array and Dictionary

Eg.

Usage :

Note:They type parameter 'Element' is written within a pair of angle brackets immediately after the structure's name.

Generic can be extend and the type parameter list will be available, from the original type definition is available, within the body.

Eg.

```
extension Stack {
    var topItem: Element? {
        return items.isEmpty ? nil : items[items.count - 1]
    }
```

```
}
```

4. Type Constrain

it's sometimes useful to enforce certain type constraints on the types that can be used with generic functions and generic types. Type constraints specify that a type parameter must inherit from a specific class, or conform to a particular protocol or protocol composition.

Type Constrain Syntax

```
func someFunction<T: SomeClass, U: SomeProtocol>(someT: T, someU: U) {
    // function body goes here
}
```

Eg. Non Generic function

```
func findIndex(ofString valueToFind: String, in array: [String]) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

Usage

```
let strings = ["cat", "dog", "llama", "parakeet", "terrapin"]
if let foundIndex = findIndex(ofString: "llama", in: strings) {
    print("The index of llama is \(foundIndex)")
}
```

Eg. Error trying to be generic

```
func findIndex<T>(of valueToFind: T, in array:[T]) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
```

```
    return nil
}
```

Eg.Fix error by adding equatable

```
func findIndex<T: Equatable>(of valueToFind: T, in array:[T]) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

Note: There is more topic for generic in docs.swift.org, you are recommend to study

- Associated Types
- Adding Constraints to an Associated Type
- Generic Where Clauses
- Associated Types with a Generic Where Clause
- Generic Subscripts

7. ARC (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.

1. How ARC Works

Every time you create a new instance of a class, ARC allocates a chunk of memory to store information about that instance. This memory holds information about the type of the instance, together with the values of any stored properties associated with that instance.

Additionally, when an instance is no longer needed, ARC frees up the memory used by that instance so that the memory can be used for other purposes instead. This ensures that class instances do not take up space in memory when they are no longer needed.

However, if ARC were to deallocate an instance that was still in use, it would no longer be possible to access that instance's properties, or call that instance's methods. Indeed, if you tried to access the instance, your app would most likely crash.

To make sure that instances don't disappear while they are still needed, ARC tracks how many properties, constants, and variables are currently referring to each class instance. ARC will not deallocate an instance as long as at least one active reference to that instance still exists.

To make this possible, whenever you assign a class instance to a property, constant, or variable, that property, constant, or variable makes a *strong reference* to the instance. The reference is called a "strong" reference because it keeps a firm hold on that instance, and does not allow it to be deallocated for as long as that strong reference remains.

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

var reference1: Person?
var reference2: Person?
var reference3: Person?

reference1 = Person(name: "John Appleseed”)

reference2 = reference1
reference3 = reference1

reference1 = nil //Check what happen
reference2 = nil //Check what happen
reference3 = nil. //Check what happen
```

2. Strong Reference Cycles Between Class Instances

```
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") }
}
var john: Person?
var unit4A: Apartment?
```

- john = Person(name: "John Appleseed")
- unit4A = Apartment(unit: "4A")

3. Resolving Strong Reference Cycles Between Class Instances (weak)

Swift provides two ways to resolve strong reference cycles when you work with properties of class type: **weak references** and **unowned references**.

Weak and unowned references enable one instance in a reference cycle to refer to the other instance without keeping a strong hold on it. The instances can then refer to each other without creating a strong reference cycle.Use a weak reference when the other instance has a shorter lifetime—that is, when the other instance can be deallocated first.
Modified Class to Apartment:

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

4. Resolving Strong Reference Cycles Between Class Instances (unowned)

Like a weak reference, an unowned reference does not keep a strong hold on the instance it refers to. Unlike a weak reference, however, an unowned reference is used when the other instance has the same lifetime or a longer lifetime.An unowned reference is

expected to always have a value. As a result, ARC never sets an unowned reference's value to `nil`, which means that unowned references are defined using nonoptional types.

```
class Customer {
    let name: String
    var card: CreditCard?
    init(name: String) {
        self.name = name
    }
    deinit { print("\(name) is being deinitialized") }
}
class CreditCard {
    let number: UInt64
    unowned let customer: Customer
    init(number: UInt64, customer: Customer) {
        self.number = number
        self.customer = customer
    }
    deinit { print("Card #\(number) is being deinitialized") }
}

var john: Customer?

john = Customer(name: "John Appleseed")
john!.card = CreditCard(number: 1234_5678_9012_3456, customer: john!)

John = nil
```

5. Strong Reference Cycles for Closures

```
class HTMLElement {

        let name: String
        let text: String?

        lazy var asHTML: () -> String = {
        if let text = self.text {
                return "<\(self.name)>\(text)</\(self.name)>"
        } else {
                return "<\(self.name) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
```

```
        }

        deinit {
                print("\(name) is being deinitialized")
                }

        }

        let heading = HTMLElement(name: "h1")
        let defaultText = "some default text"
        heading.asHTML = {
                return "<\(heading.name)>\(heading.text ?? defaultText)</\
                (heading.name)>"
        }
        print(heading.asHTML())

        var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
        print(paragraph!.asHTML())
```

6. Resolving Strong Reference Cycles for Closures

```
class HTMLElement {

    let name: String
    let text: String?

    lazy var asHTML: () -> String = {
        [unowned self] in
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }

    deinit {
        print("\(name) is being deinitialized")
    }

}

var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
```

```
print(paragraph!.asHTML())
```

Paragraph = **nil**

8. Final Tips for Swift Developer

A. Use Optional Binding when engage with Optional Type Value *

B. Switch with comma separated, range and tuple matching, used with underscore and assigning to let

```
let someChar = "e"
switch someChar {
	case "a", "e", "i", "o", "u":
		print("\(someChar) is a vowel")
	default:
		print("\(someChar) is a consonant")
}

//————————————————[Range Matching]——————————
let count = 3_000
let countedThings = "stars"

switch count {
	case 0...9:
		print("a few")
	case 10...10_000:
		print("many")
	default:
		print("a lot of")
	}
//————————————————[Tuple Matching]——————————
let coord = (1,1)
switch coord {
	case (0,0):
		print("Origin")
	case (_, 0):
		print("x axis")
	case (0, _):
		print("y axis")
	case (-2...2, -3...3):
		print("within boundries")
	default:
		print("out of bounds")
}

//————————[Tuple Matching with assignment]——————————

let anotherPoint = (10, 10)
switch anotherPoint {
	case (let x, 0):
		print("on the x-axis with an x value of \(x)")
```

```
        case (0, let y):
                print("on the y-axis with a y value of \(y)")
        case let (z, w): //This acts as the default case. Since it is only
        assigning a tuple, any value matches.
        print("somewhere else at (\(z), \(w))")
}

//————————[Tuple Matching with conditional]——————————
switch anotherPoint {
        case let (x, y) where x == y:
                print("x = y")
        default:
                break
}
```

C. Use **map** to map all elements to transform other type in closure

```
var data = [1,2,3,4,5]

var otherData = data.map {
        (number)-> String in

        return String(number*3)
}
print(otherData)

//————————————-[Apple Example ]
let digitNames = [
                0: "Zero", 1: "One", 2: "Two",   3: "Three", 4: "Four",
                5: "Five", 6: "Six", 7: "Seven", 8: "Eight", 9: "Nine"
                ]
let numbers = [16, 58, 510]

let stringsArray = numbers.map {
        (number) -> String in
                var number = number
                var output = ""
                while number > 0 {
                        output = digitNames[number % 10]! + output
                number /= 10
                        }
                return output
}
print(stringsArray)
```

D. Use **Closure to perform cascade functioning**

```
func primaryFunction(_ theNextFunc:()->())

{
        //Note theNextFunction is a parameter
        //also a function taking no parameter with no return value


         //Do your primary job first and let use this function
        //do something after your work by calling

        theNextFunc()
}


//Then Call the primaryFunction with cascaded function in somwhere

primaryFunction()
{
        print("I am cascade function")
        //this is cascaded function running
}


let array = ["John", "Tim", "Steve"]
```

E. Use **Closure to perform sort with various facts**

```
var reversed = array.sorted(by: {
        (s1: String, s2: String) -> Bool in

        return s1 > s2
})
```

```
//Using type inference, we can omit the params and return types. This is true when passing closures as params to a function.
reversed = array.sorted(by: {s1, s2 in return s1 > s2})

//In case of single-expression closures, the return value is implicit, thus the return expression can be omitted.
reversed = array.sorted(by: {s1, s2 in s1 > s2})

//In the previous examples, the names of the closure's params were explicit. You can use the $X variables to refer to params for the closure.
//This eliminates the need for the first params list, which makes the body the only relevant part.
reversed = array.sorted(by: {$0 > $1})
```

```
//We can even take this to an extreme. String defines its own
implementation for the ">" operator, which is really all the closure does.
reversed = array.sorted(by: >)
```

F. **Use Enumeration for type consistency and readable constant**

G. **Use Computed Properties for validating the value of the properties using get , set , willSet and didSet**

```
struct MyClass {

        var firstValue: Int
        let length: Int

        var computedTypeProperty: Int {
                // return an Int value here
                get {
                        return self.computedTypeProperty
                }
                set {
                        self.computedTypeProperty = newValue
                }

        }

    //This property has observers for when the the value will & did set

    var propWithObserver: Int = 0 {
            willSet {
                        print(newValue)
        }


        didSet {
                        print(oldValue)
        }

    }
```

Note:

Stored Properties of Constant Structure Instances

If you create an instance of a structure and assign that instance to a constant, you cannot modify the instance's properties, even if they were declared as variable properties.

This behavior is due to structures being value types. When an instance of a value type is marked as a constant, so are all of its properties.

The same is not true for classes, which are reference types. If you assign an instance of a reference type to a constant, you can still change that instance's variable properties.

For the struct, a memberwise initializer is provided. This means that stored properties don't necessarily need to have an initial value. A default initializer is created for all of them.

H. **Use subscript for easier access for a properties, which is mostly array, of an object by using indexing to the object like a regular array**

```
class Daysofaweek {
        private var days = ["Sunday", "Monday", "Tuesday",
        "Wednesday", "Thursday", "Friday", "saturday"]
subscript(index: Int) -> String {
    get {
        return days[index]
    }
    set(newValue) {
        self.days[index] = newValue
        }
        }
```

```
}

var p = Daysofaweek()

println(p[0])

println(p[1])
```

I. **Failable initializers allow us to return nil during initialization in case there was a problem.**

The object being initalized is treated as an optional.

You can also define a failable initializer that returns an implicitly unwrapped optional instance by writing init!

e.g

```
// For enums, nil can be returned at any point of initalizations

enum TemperatureUnit {
    case Kelvin, Celsius, Fahrenheit
    init?(symbol: Character) {
        switch symbol {
        case "K":
            self = .Kelvin
        case "C":
            self = .Celsius
        case "F":
            self = .Fahrenheit
        default:
            return nil
        }
    }
}
```

e.g..

```
// For class instances, nil can only be returned after initalizing all properties.

class Product {

    let name: String!

    init?(name: String) {

        self.name = name

        if name.isEmpty { return nil }
```

```
    }
}
```

J. **Use Deinitializer to make final execution before the object (Class Object) is destroyed. Deinitializer is only for class**

```
class Bank {
    static var coinsInBank = 10_000
    static func vendCoins(numberOfCoinsToVend: Int) -> Int {
        let numberOfCoinsAllowedToVend = min(numberOfCoinsToVend,
coinsInBank)
        coinsInBank -= numberOfCoinsAllowedToVend
        return numberOfCoinsAllowedToVend
    }
    static func receiveCoins(coins: Int) {
        coinsInBank += coins
    }
}

class Player {
    var coinsInPurse: Int
    init(coins: Int) {
        coinsInPurse = Bank.vendCoins(numberOfCoinsToVend: coins)
    }
    func winCoins(coins: Int) {
        coinsInPurse += Bank.vendCoins(numberOfCoinsToVend: coins)
    }
    deinit {
        Bank.receiveCoins(coins: coinsInPurse)
    }
}


var playerOne: Player? = Player(coins: 100)

print(playerOne!.coinsInPurse)

print(Bank.coinsInBank)

playerOne!.winCoins(coins: 2_000)

print(playerOne!.coinsInPurse )

print(Bank.coinsInBank)
```

```
playerOne = nil //Just before this happens, its deinitializer is called
automatically.
print(Bank.coinsInBank)
```