iOS Programming

Lecture 6

Recap

Enumeration

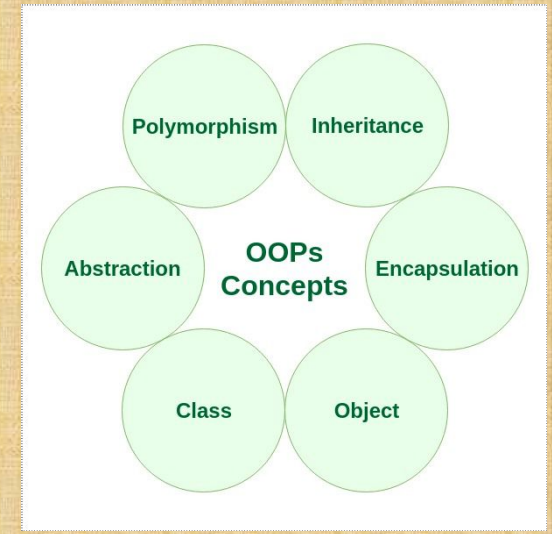
Structs

Tuples

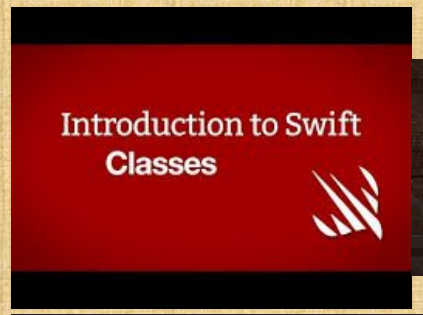
Closures

Today

Object Oriented Programming



Classes



Protocols

OOP - Class

Class Definition as on brilliant.org -

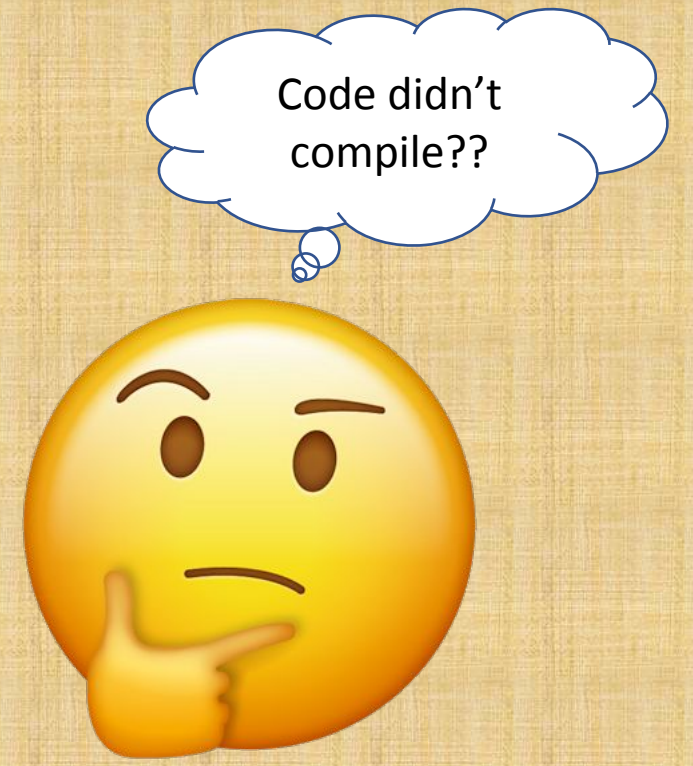
In object-oriented programming, a class is a blueprint for creating objects (a particular data structure), providing initial values for state (member variables or attributes), and implementations of behavior (member functions or methods).

OOP -Class

```
class ComputerLanguage{
    //Member to Store Language Name
    var languageName: String

    //Member to identify if it is a compiled language
    var isCompiled: Bool

    //Member storing the market adoption %ages
    var adoptionRate: Double

    //Member identifying if languages has OOP support
    var supportsOOP: Bool

    //and so on…
}
```

OOP -Class

Remember our discussion on variables needing initiation
You have two choices



OOP -Class

```
class ComputerLanguage{
    var languageName: String
    var isCompiled: Bool?
    var adoptionRate: Double
    var supportsOOP: Bool?

    init(languageName: String, adoptionRate: Double){
        self.languageName = languageName
        self.adoptionRate = adoptionRate
        self.isCompiled = true
        self.supportsOOP = true
    }
}


ComputerLanguage(languageName: "Swift",
adoptionRate: 5.9)
```

Either add initialization or make the member optional

OOP - Initializer

What parameters should I add to initializers?

Let's dissect the initializer a bit

No func keyword

```
init(languageName: String, adoptionRate: Double){
    self.languageName = languageName
    self.adoptionRate = adoptionRate
    self.isCompiled = true
    self.supportsOOP = true
}
```

No return type

OOP -Objects

An instance is a specific object created from a particular class. Classes are used to create and manage new objects and support inheritance.

OOP -Objects


```
var java = ComputerLanguage(languageName: "Java", isCompiled: true, adoptionRate: 40.2, supportsOOP: true)
```

```
var swift = ComputerLanguage(languageName: "Swift", isCompiled: true, adoptionRate: 5.9, supportsOOP: true)
```

```
var kotlin = ComputerLanguage(languageName: "Kotlin", isCompiled: true, adoptionRate: 7.8, supportsOOP: true)
```

OOP – Comment on classes and files

We are playing with playgrounds, so you will see me defining classes (sometimes multiple classes) and objects in the same file. For all professional implementations you will include each class in its individual file.

OOP – deinit

```
deinit{
    //clear any resources you want to release
    //file/db handle probably
}
```

OOP – Computed Properties

```
class Student{
    var fullName : String
    var mathScore: Int
    var scienceScore: Int
    var grade: String {
        get{
            if (mathScore > 90 && scienceScore > 90){
                return "A"
            }else if (mathScore > 80 && scienceScore > 80){
                return "B"
            }else{
                return "C"
            }
        }
        set{
            print("You have chosen to force override grade to: \(newValue)")
        }
    }
    init(fullName: String) {
        self.fullName = fullName
        self.mathScore = 0
        self.scienceScore = 0;
    }
}
var carlos = Student(fullName: "Carlos")
carlos.mathScore = 95
carlos.scienceScore = 85
print ("\(carlos.fullName) earned grade \(carlos.grade)")
```

Computed properties can accomplish tasks that require functions in other languages

OOP – Computed Properties

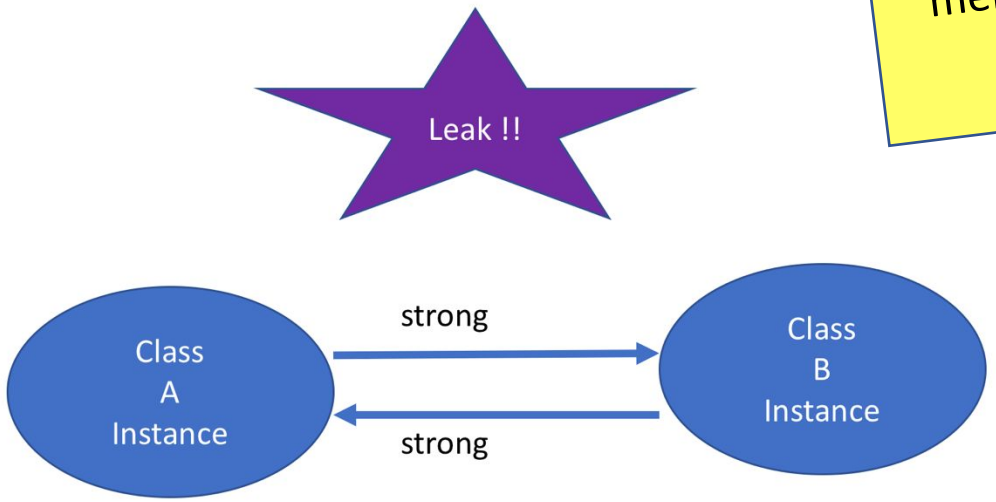
1
- You can drop set to make the property read only.

2
- For read only properties you don’t need to explicitly write get.

3
- Computed properties will always be a var and never a let.

ARC – Memory Management

Object Retain Cycles



Objects are deallocated when there are NO strong references left.

If deinit is not heavily used then how is memory being freed?

OOP -Encapsulation

Encapsulation refers to the bundling of data with the methods that operate on that data, or the restricting of direct access to some of an object's components. Encapsulation is used to hide the values or state of a structured data object inside a class, preventing unauthorized parties' direct access to them.

Think Access Specifiers

OOP -Encapsulation

```
class AmortizationCalculator{
    private var myInterestRate: Double = 3.4 //I may be pulling this from
network call or from DB

    public func getInterestRateApplied() -> Double{
        return self.myInterestRate
    }
}


var calculator = AmortizationCalculator()
print (calculator.getInterestRateApplied())
```

OOP -Inheritance

As the name says –

Pass on the wealth

OOP -Inheritance

```
class ComputerLanguage{
    var languageName: String
    init() {
        languageName = ""
    }
}
class OOPLanguage: ComputerLanguage{
    public func teachClassMeaning(){}
    public func teachObjects(){}
    public func teachInheritence(){}
}
class Swift: OOPLanguage{
    public func coverOOPBasics(){
        self.teachClassMeaning();
        self.teachObjects();
        self.teachInheritence();
    }
}
var swift = Swift()
swift.languageName = "Swift"
```

OOP –Inheritance vs Extensions

<u>Inheritance:</u>
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. Inheritance is a fundamental behavior that differentiates classes from other types in Swift.

<u>Extension:</u>

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 (known as retroactive modeling).

OOP - Extensions

```
class ComputerLanguage{
    var languageName: String
    init() {
        languageName = ""
    }
}
class OOPLanguage: ComputerLanguage{
    public func teachClassMeaning(){}
    public func teachObjects(){}
    public func teachInheritence(){}
}
extension OOPLanguage{
    public func coverOOPBasics(){
        self.teachClassMeaning();
        self.teachObjects();
        self.teachInheritence();
    }
}
var swift = OOPLanguage()
swift.languageName = "Swift"
swift.coverOOPBasics() //We called the function on OOPLanguage itself
```

OOP - Extensions

```
import UIKit
var pi = 3.1415926535897932 38

var piValue = "The value of PI is \(String(format: "%.2f",
pi))"
print(piValue)
```

Swift using extensions all the time. Eg: String is a struct so there is no other way to add functionality

Try running without the import and see.

OOP –Inheritance & initializer

```
class ComputerLanguage{
    var languageName: String
    init() {
        languageName = ""
        print ("Computer Language Initializer Done!!!")
    }
}


class Swift: ComputerLanguage{
    var isCompiled: Bool
    override init() {
        isCompiled = true
        print ("Swift Initializer Done!!!")
    }
}
var swift = Swift()
swift.languageName = "Swift"
```

Spot the Differences

OOP –Inheritance – final keyword

```
class ComputerLanguage{
    final func printMessage() {
        print ("Computer Language printing!!!")
    }
}


class Swift: ComputerLanguage{
    override func printMessage() {
        print ("Swift Language printing!!!")
    }
}
var swift = Swift()
swift.printMessage()
```

Will this code work?
Why?

OOP –Inheritance – final keyword

```
final class ComputerLanguage{
    func printMessage() {
        print ("Computer Language printing!!!")
    }
}


class Swift: ComputerLanguage{
    override func printMessage() {
        print ("Swift Language printing!!!")
    }
}
var swift = Swift()
swift.printMessage()
```

Will this code work?
Why?

Swift – Protocol Oriented Programming

Inheritance is not that common in Swift

1. the official procedure or system of rules governing affairs of state or diplomatic occasions.
"protocol forbids the prince from making any public statement in his defense"

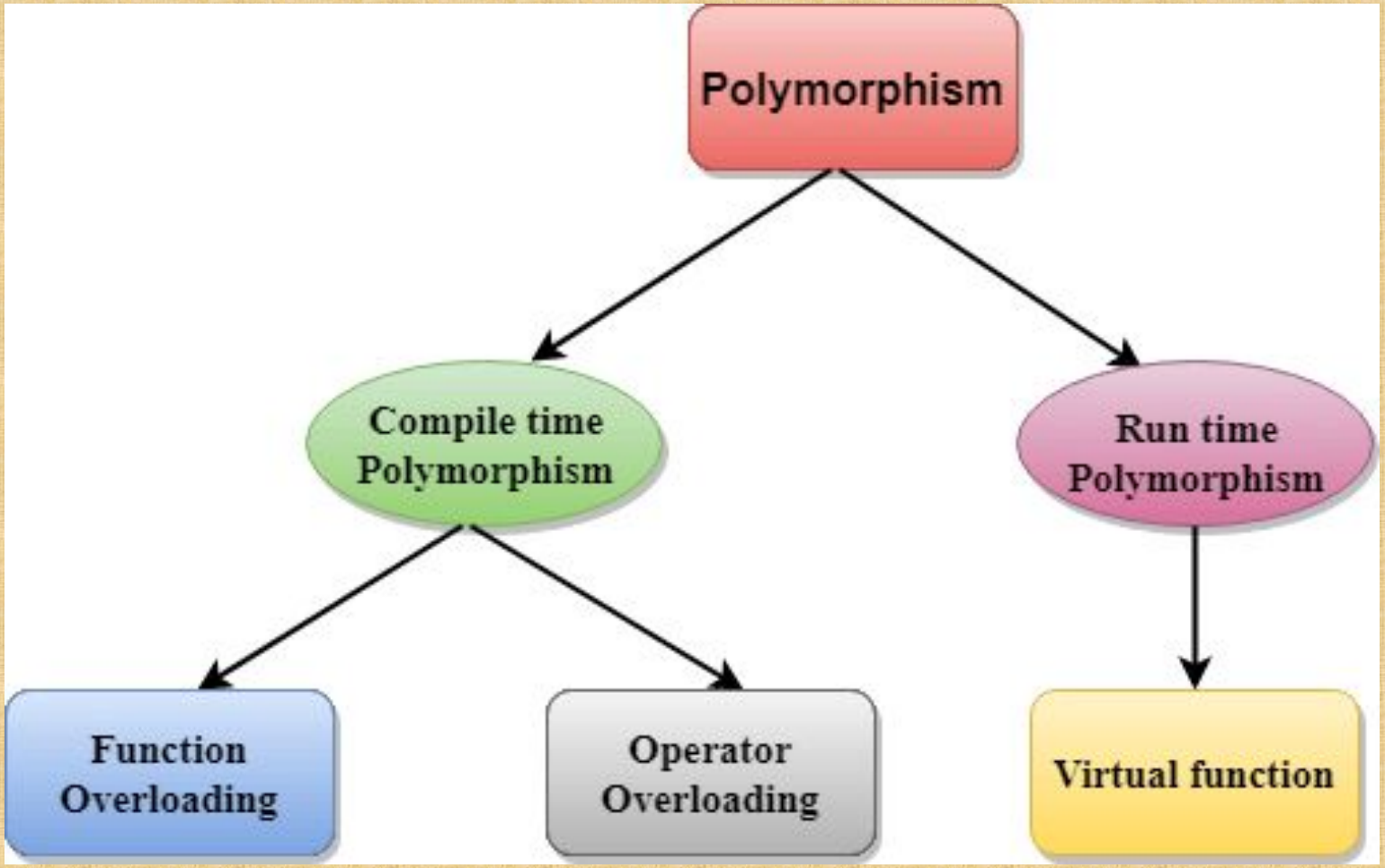
2. the original draft of a diplomatic document, especially of the terms of a treaty agreed to in conference and signed by the parties.

OOP –Protocol Example

```
class ComputerLanguage: CustomStringConvertible{
    var languageName: String
    var description: String {
        return "ComputerLanguage is initialized for: \(languageName)"
    }

    init(languageName: String) {
        self.languageName = languageName
    }
}


let swift = ComputerLanguage(languageName: "Swift")
print (swift)
```

OOP -Polymorphism

OOP -Polymorphism



OOP –Polymorphism - Compile time

Method overloading

```
class ComputerLanguage{
    var languageName: String
    init(languageName: String){
        self.languageName = languageName
    }
    public func doSyntaxAnalysis(){
        print ("My default syntax analysis implementation for: \(languageName) ")
    }

    public func doSyntaxAnalysis(ignoringWarnings: Bool){
        print ("My default syntax analysis implementation for: \(languageName) while ignoring
warnings: \(ignoringWarnings)")
    }
}


var swift = ComputerLanguage(languageName: "Swift")
swift.doSyntaxAnalysis()
swift.doSyntaxAnalysis(ignoring Warnings: true)
```

OOP –Polymorphism - Compile time

```
class ComputerLanguage{
    var languageName: String
    init(languageName: String){
        self.languageName = languageName
    }
    public func doSyntaxAnalysis(){
        print ("My default syntax analysis implementation for: \(language
    }
}
class Swift: ComputerLanguage{
    init(){
        super.init(languageName: "Swift")
    }
    override public func doSyntaxAnalysis(){
        print ("Doing my custom implementation for: \(languageName)")
    }
}
var swift = Swift()
swift.doSyntaxAnalysis()
```

Method overriding

OOP –Polymorphism - Run time

```
protocol ComputerLanguageProtocol{
    var languageName: String{ get set }
    func doSyntaxAnalysis()
}


class ComputerLanguage: ComputerLanguageProtocol{
    var languageName: String
    init(languageName: String){
        self.languageName = languageName
    }
    public func doSyntaxAnalysis(){
        print ("My default syntax analysis implementation for: \(languageName)")
    }
}
```

OOP –Polymorphism - Run time

```
class Swift: ComputerLanguage{
    init(){
        super.init(languageName: "Swift")
    }
    override public func doSyntaxAnalysis(){
        print ("Doing my custom implementation for: \(languageName)")
    }
}

class Java: ComputerLanguage{
    init(){
        super.init(languageName: "Java")
    }
}
var swift = Swift()
swift.doSyntaxAnalysis()
var java = Java()
java.doSyntaxAnalysis()
```

OOP -Abstraction

ABSTRACTION is the concept of object-oriented programming that "shows" only essential attributes and "hides" unnecessary information. The main purpose of abstraction is hiding the unnecessary details from the users. Abstraction is selecting data from a larger pool to show only relevant details of the object to the user. It helps in reducing programming complexity and efforts. It is one of the most important concepts of OOPs.

OOP -Abstraction

You don’t need to worry about what Xbox does on hitting X

OOP -Abstraction

```
protocol ComputerLanguageProtocol{
    var languageName: String{ get set }
    func doSyntaxAnalysis()
}
extension ComputerLanguageProtocol{
    public func doSyntaxAnalysis(){
        print ("My default syntax analysis implementation for: \(languageName)")
    }
}
class Swift: ComputerLanguageProtocol{
    var languageName = "Swift"
    public func doSyntaxAnalysis(){
        print ("Doing my custom implementation for: \(languageName)")
    }
}
class Java: ComputerLanguageProtocol{
    var languageName = "Java"
}
Swift().doSyntaxAnalysis()
Java().doSyntaxAnalysis()
```

Struct vs Classes

Classes don't get default member wise variables as structs

```
struct ComputerLanguageStruct{
    var languageName: String
}

class ComputerLanguage{
    var languageName: String
    init(languageName: String) {
        self.languageName = languageName
    }
}

ComputerLanguageStruct(languageName: "Swift")
ComputerLanguage(languageName: "Swift")
```

Struct vs Classes


```
struct ComputerLanguage{
    var languageName: String
}
struct Swift: ComputerLanguage{
    var languageName: String
    init(languageName: String) {
        self.languageName = languageName
    }
}
ComputerLanguage(languageName: "Swift")
```

No inheritance

Inheritance from non-protocol type 'ComputerLanguage'

Struct vs Classes

Structs are value types

Objects are reference types

Struct vs Classes


```
struct LangStruct{
    var languageName: String
}
class LangClass{
    var languageName: String
    init(languageName: String) {
        self.languageName = languageName
    }
}
var swift = LangStruct(languageName: "Swift")
var java = LangClass(languageName: "Java")
var objc = swift
objc.languageName = "Objective C"
var kotlin = java
kotlin.languageName = "Kotlin"
print("Final Eval, Swift:\(swift.languageName) Java:\(java.languageName) Objc:
\(objc.languageName) Kotlin:\(kotlin.languageName)")
```

Struct vs Classes

let for classes means references can't change but individual members can, while let for structures means everything is a constant

deinit is not available for structures

Parting Notes

Practice:

- Classes
- Protocols

Simple Exercise:

Write code to create a Simple Calculator Class

You are free to choose the Class and function names

Operations:

- Add
- Subtract
- Multiple
- Divide