OOPs

-DNReddy

Reach me @iPhoneDev1990@gmail.com

Object Oriented Principles

Class ***(Logical structure to create an Object)***

Object ***(Physical implementation of Class)***

Encapsulation ***(Binding Instance Variables and Behaviours (Methods))***

Abstraction ***(Hiding unnecessary behaviours and exposing necessary functionality)***

Inheritance ***(Deriving functionality from an existing class)***

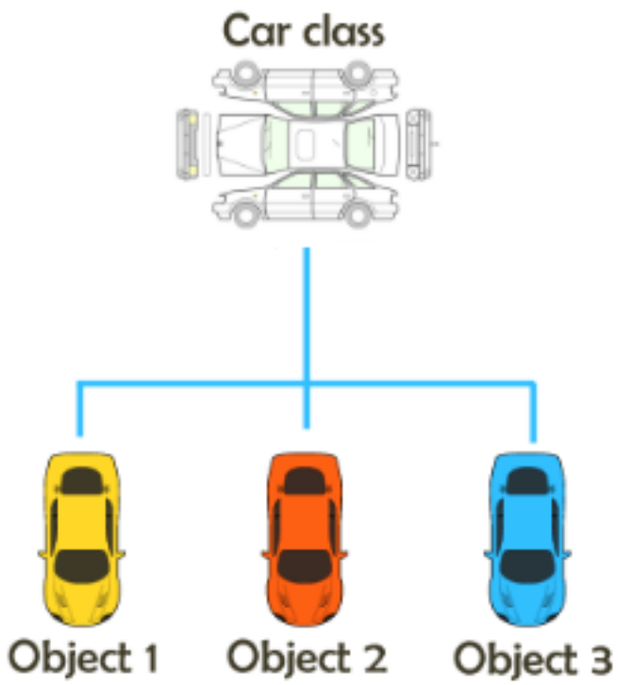
Polymorphism ***(Overloading and Overriding)***

Class

Definitions:
Logical structure / design to create physical object
Template / Blue print to create Objects
Class is collection of *properties(variables)* and *Behaviours* (Methods)

Note:
Once a class is created, you can create any number of objects

Examples: Building, Person, Car …



```
class ClassName
{
    // property declarations
    // Method Implementations
}
```

DNREDDY

Reach me @iPhoneDev1990@gmail.com

Object

Definition:

Object is an instance of a class

Object is a physical implementation of class

Note: Class doesn't contain any memory whereas Object contains memory

Syntax:

```
var  objectOne: ClassName = ClassName();
```

ClassName is optional

Ex:

```
var maruthi: Car = Car()
var audi = Car()
```

Methods

Definition: **Method is a block of statements which performs some specific task. Use func keyword to define Methods.**

Syntax:

```
func  methodName(inArgOne: Datatype, inArgTwo: Datatype, ..) —> Returntype;
{
    // Statements
}
```

Ex:

```
func start() —> Void
{
    // Statements
}
func moveWithSpeed(speed:Int) —> Void
{
    // Statements
}
```

DNR REDDY

Reach me @iPhoneDev1990@gmail.com

Variety of Methods

```
func functionWithNoArguments()
{
    // Statements
}

func functionWithArguments(argOne: Int, argTwo: Int)
-> Void
{
    // Statements
}

func functionWithObjects(audi: Car, maruthi: Car) ->
Car
{
    // Statements
}
Note: Return type is optional if it is Void.
```

DNREDDY

Reach me @iPhoneDev1990@gmail.com

type method & instance method

type methods:

- **class method starts with class / static keyword**
- **class methods can not access the properties in the method implementations**
- **Use class name to call class methods**

Instance methods:

- **properties can be accessed in instance method**
- **instance method must be called on object of that class**

DNREDDY

Reach me @iPhoneDev1990@gmail.com

Method Calling

object.methodName()
object.methodName(label: Value)
object.methodName(label: Value, label: Value ..,)

Example

```
class Car
{
    var aProperty = 10;
    func aInstanceMethod()
    {
        print(aProperty);
    }

    class func aClassMethod()
    {
        print(aProperty); // Error
    }
}

let car = Car()
car.aProperty = 10
car.aInstanceMethod()

Car.aClassMethod()
Car.aProperty = 10 // Error
```

DNREDDY

Reach me @iPhoneDev1990@gmail.com

```swift
class Person : NSObject
{
    var age: Int
    var name: String
    var numberOfLegs: Int

    func walk() -> Void
    {
        print("\(name) is Walking")
    }

    func sleep() -> Void
    {
        print("\(name) is Speaking")
    }

    func talk() -> Void
    {
        print("\(name) is Talking")
    }
}

let modi = Person()
modi.name = "Modi"
modi.numberOfLegs = 2
modi.age = 70
modi.walk()
modi.talk()

let advani = Person()
advani.name = "Advani"
modi.numberOfLegs = 2
modi.age = 78
advani.sleep()
advani.talk()
```

```swift
class Person : NSObject
{
    var age: Int
    var name: String
    var numberOfLegs: Int

    func walk() -> Void
    {
        print("\(name) is Walking")
    }

    func sleep() -> Void
    {
        print("\(name) is Speaking")
    }

    func talk() -> Void
    {
        print("\(name) is Talking")
    }
   override init()
     {
         numberOfLegs = 2
         age = 1
         name = "Baby"
     }
}
```

```swift
let modi = Person()
modi.name = "Modi"
modi.age = 70
modi.walk()
modi.talk()

let advani = Person()
advani.name = "Advani"
modi.age = 78
advani.sleep()
advani.talk()
```

Naming Conventions

Class names: UpperCamelCase

Ex: Car, Building, BigCar, BulletTrain

Variable/Object names: lowerCamelCase
Ex:
maruthiSuzki, audi, smallCar

Method names: lowerCamelCase
Ex: startWithSpeed, changeColor, changeGear …

Properties

Properties are classified into three categories

- **Stored Properties**
- **Lazy Stored Properties**
- **Computed Properties**
- **Static Properties**

Stored Properties:

- **Stored Properties stores a variable value or constant**
- **Stored Properties are used in Structures and Classes**

Lazy Stored Properties:

- **A lazy stored property is a property whose initial value is not calculated until the first time it is used.**
- **You indicate a lazy stored property by writing the lazy modifier before its declaration.**

Computed Properties:

- **Computed properties calculates the values rather than storing**
- **Computed properties are used in Structures, Enums and Classes.**
- **Computed properties provide a getter and an optional setter to retrieve and set other properties and values indirectly.**

Static Properties:

- **These properties are class level properties.**
- **These are accessed over Class Name**

Task

```
class Person
{
    // Stored Properties
    var clothsWeight: Float = 0.0
    var totalWeight: Float = 0.0

    // Lazy Stored Properties
    lazy var aLazyProperty = Calculator.performComplexOperation()

    // Computed Properties
    var actualWeight: Float{
        return totalWeight - clothsWeight;
    }
    // Static Properties
    static var sharedPropertyOne:String!
}


let modi = Person()
modi.clothsWeight = 2.5
modi.totalWeight = 70.0
Person.sharedPropertyOne = "This is static shared property"
print("Person cloths weight is : \(modi.clothsWeight)")
print("Person total weight is : \(modi.totalWeight)")
print("Person cloths weight is : \(modi.actualWeight)")

O/P:
Person cloths weight is : 2.5
Person total weight is : 70.0
Person cloths weight is : 67.5
```

Property Observers

Swift lets you add code to be run when a property is about to be changed or has been changed. This is frequently a good way to have a user interface update when a value changes, for example.

There are two kinds of property observer: **willSet** and **didSet**, and they are called before or after a property is changed. In **willSet** swift provides your code with a special value called **newValue** that contains what the new property value is going to be, and in didSet you are given **oldValue** to represent the previous value.

For Offline / Online Training, reach me@ iPhoneDev1990@gmail.com

```
class Person {
    var clothes: String {
        willSet {
            updateUI("I'm changing from \(clothes) to \(newValue)")
        }

        didSet {
            updateUI("I just changed from \(oldValue) to \(clothes)")
        }
    }
    func updateUI(msg: String)
    {
        print(msg)
    }
}

class Properties: NSObject
{
    func playWithProperties()
    {
        var taylor = Person(clothes: "T-shirts")
        taylor.clothes = "Jeans"
    }
}
// Output
I'm changing from T-shirts to Jeans
I just changed from T-shirts to Jeans
```

Initilizers and Deinitlizers

Initilizers: Initilizers are used to prepare an object with proper default values before it is created. Classes and Structures contains initializers.
init() is the default initializer and **deinit**() is default destructor in swift

```
class Test
{
    var a: Int = 10
    var b: Int
    var c: Int

    init()
    {
        b = 20
        c = 30
    }
}

let a = Test()

print(a.a)
print(a.b)
print(a.c)

Output:
10
20
30
```

For Offline / Online Training, reach me@ iPhoneDev1990@gmail.com

Custom Initilizers

```
class Test
{
    var a: Int = 10
    var b: Int!
    var c: Int!

    init(bValue:Int, cValue: Int)
    {
        b = bValue
        c = cValue
    }
}

let a = Test()

print(a.a)
print(a.b)
print(a.c)

let b = Test(bValue: 200, cValue: 300)

print(b.a)
print(b.b)
print(b.c)

Output:
10
20
30
10
200
300
```

For Offline / Online Training, reach me@ iPhoneDev1990@gmail.com

Designated & Convenience Initilizers

```
class Book {
    var bookName: String
    var authorName: String
    var pages: Int

    // Designated Initilizer
    init() {
        bookName = ""
        authorName = ""
        pages = 0
    }
    // Convenience Initilizer
    convenience init(customPages: Int) {
        self.init()
        pages = customPages
    }
}
```

convenience initializer is used to provide custom default values for some properties of a Class.

For Offline / Online Training, reach me@ iPhoneDev1990@gmail.com

Custom Deinitilizers

Deinitilizers disposed the waste resources.
Don’t call deinit() method. It will be called by the Runtime System.

```
class Test
{
    var a: Int!
    var b: Int!
    var c: Int!

    init()
    {
        a = 10
        b = 20
        c = 30
    }
    deinit {
        a = 0
        b = 0
        c = 0
    }
}

let a = Test()

print(a.a)
print(a.b)
print(a.c)
```

For Offline / Online Training, reach me@ iPhoneDev1990@gmail.com

Encapsulation

Encapsulation: Encapsulation is a process of binding instance variables and methods together into a single unit to keep safe from out side of that class.
Encapsulation is achieved through Class.

Data Encapsulation: Data encapsulation is the process of hiding instance variables' data to out side of the class/ Other classes. We can achieve data encapsulation using Accessor Specifiers

Q: How do you achieve the encapsulation in Swift?
Ans: Using Classes

Q: How do you achieve the data encapsulation?
Ans: Using Accessor Specifiers

DNR SREEDY

Reach me @iPhoneDev1990@gmail.com

Abstraction

Abstraction: Abstraction is a process of exposing necessary functionality and hiding unnecessary functionality.
We can achieve abstraction using Protocols file.

Inheritance

Inheritance is the process of extending the existing class. Deriving the existing class functionality and adding additional functionality to the newly creating class.

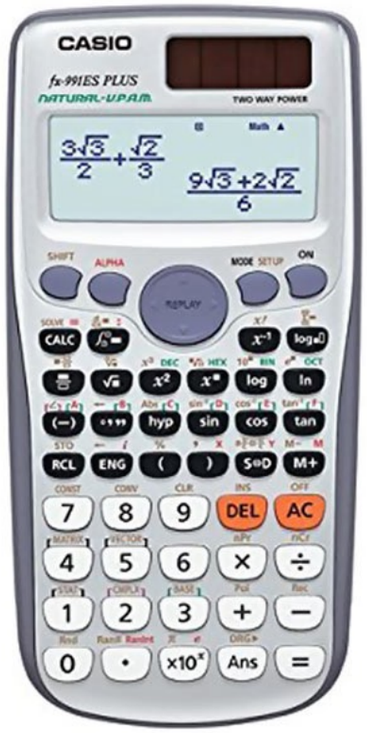
Syntax:

```
class NewClass : ExistingClass
{
    // Additional Properties
    // Additional Methods
}

@end
```

Here, NewClass is known as Derived class / sub class / child class.
Existing class is known as Super class / Parent Class.

: is the inheritance operator


AC
C
÷
7
8
9
X
4
5
6
−
1
2
3
=
+
0
,


CASIO
fx-991ES PLUS
NATURAL-V.P.A.M.
TWO WAY POWER
Math
SHIFT
ALPHA
MODE SETUP
ON
CALC
DEL
AC

```
class BasicCalculator
{
    var a: Int!
    var b: Int!

    var sum:Int{
        return a + b;
    }
    var sub:Int{
        return a - b;
    }
    var mul:Int{
        return a * b;
    }
    var div:Int{
        return a / b;
    }
    var mod:Int{
        return a % b;
    }

    func displayResults() -> Void
    {
        print("Sum :%i", sum) // 30
        print("Sub :%i", sub) // -10
        print("Mul :%i", mul) // 200
        print("Div :%i", div) // 2
        print("Mod :%i", mod) // 10
    }
}

let basicCalc = BasicCalculator()
basicCalc.a = 20;
basicCalc.b = 10;
basicCalc.displayResults()
```

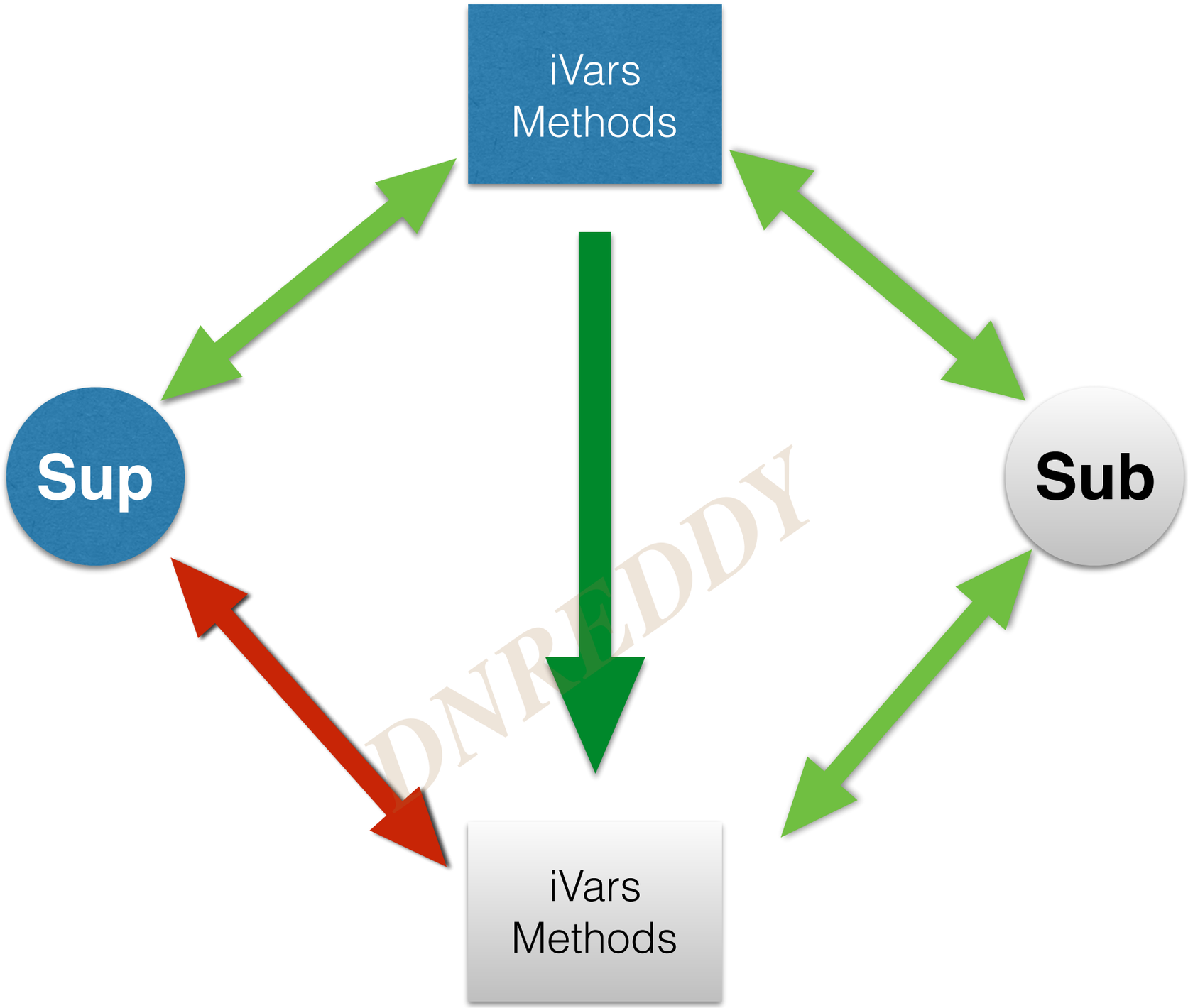
Output:

```
Sum :30
Sub :10
Mul :200
Div :2
Mod :0
```

```swift
class ScientificCalculator: BasicCalculator
{
    var Pi: Float!

    var power: Int{
        return a * a;
    }

    func factorial(aNumber: Int) -> Void
    {
        var fact = 1;
        for i in 1...aNumber
        {
            fact = fact + fact * i
        }
        print("Factorial is : \(fact)")
    }

    func displayOutputs()
    {
        print("Sum :\(sum)") // 300
        print("Sub :\(sub)") // 100
        print("Mul :\(mul)") // 2000
        print("Div :\(div)") // 2
        print("Mod :\(mod)") // 0
        print("Power :\(power)") // 40000
    }
}

var scientificCalc = ScientificCalculator()
scientificCalc.a = 200
scientificCalc.b = 100
scientificCalc.Pi = 3.145

scientificCalc.factorial(aNumber: 5)
scientificCalc.displayResults()
scientificCalc.displayOutputs()
```

Output:

```
Factorial is: 720
Sum :300
Sub :100
Mul :20000
Div :2
Mod :0
Power :40000
```



Sub class variables / methods are not accessible in Super class.

Super class variables / methods are accessible in Sub class.

Reach me @iPhoneDev1990@gmail.com

Polymorphism (Many Forms)

i) <u>Overloading</u>: In any class multiple methods has *same* name but difference in number of parameters / order of parameters / type of parameters

Ex:

```
func display() -> Void
{
    print("Nothing to print")
}

func display(aVar:Int) -> Void
{
    print("\(aVar)")
}

func display(aVar:Float) -> Void
{
    print("\(aVar)")
}

func display(aVar:Int, bVar: Float) -> Void
{
    print("\(aVar)")
    print("\(bVar)")
}

func display(aVar:Float, bVar: Int) -> Void
{
    print("\(aVar)")
    print("\(bVar)")
}
```

Here, display is method name. It is same in all 5 statements but there is a difference in number of parameters/type of parameters/order of parameters

NOTE:

Swift supports Overloading but not Objective-C

Overriding: In Inheritance relationship, having same methods in super class and sub class with difference in implementation. Use override keyword to override a function.

To achieve Overriding, you must have inheritance relationship between two classes.

BasicCalculator Class method:

```
func displayOutputs()
{
    print("Sum :\(sum)") // 300
    print("Sub :\(sub)") // 100
    print("Mul :\(mul)") // 2000
    print("Div :\(div)") // 2
    print("Mod :\(mod)") // 0
}
```

Scientific Calculator Class method:

```
override func displayOutputs()
{
    print("Sum :\(sum)") // 300
    print("Sub :\(sub)") // 100
    print("Mul :\(mul)") // 2000
    print("Div :\(div)") // 2
    print("Mod :\(mod)") // 0
    print("Power :\(power)") // 40000
}
```

DNREDDY

Reach me @iPhoneDev1990@gmail.com

Thank You

-DNReddy

DNREDDY

Reach me @iPhoneDev1990@gmail.com