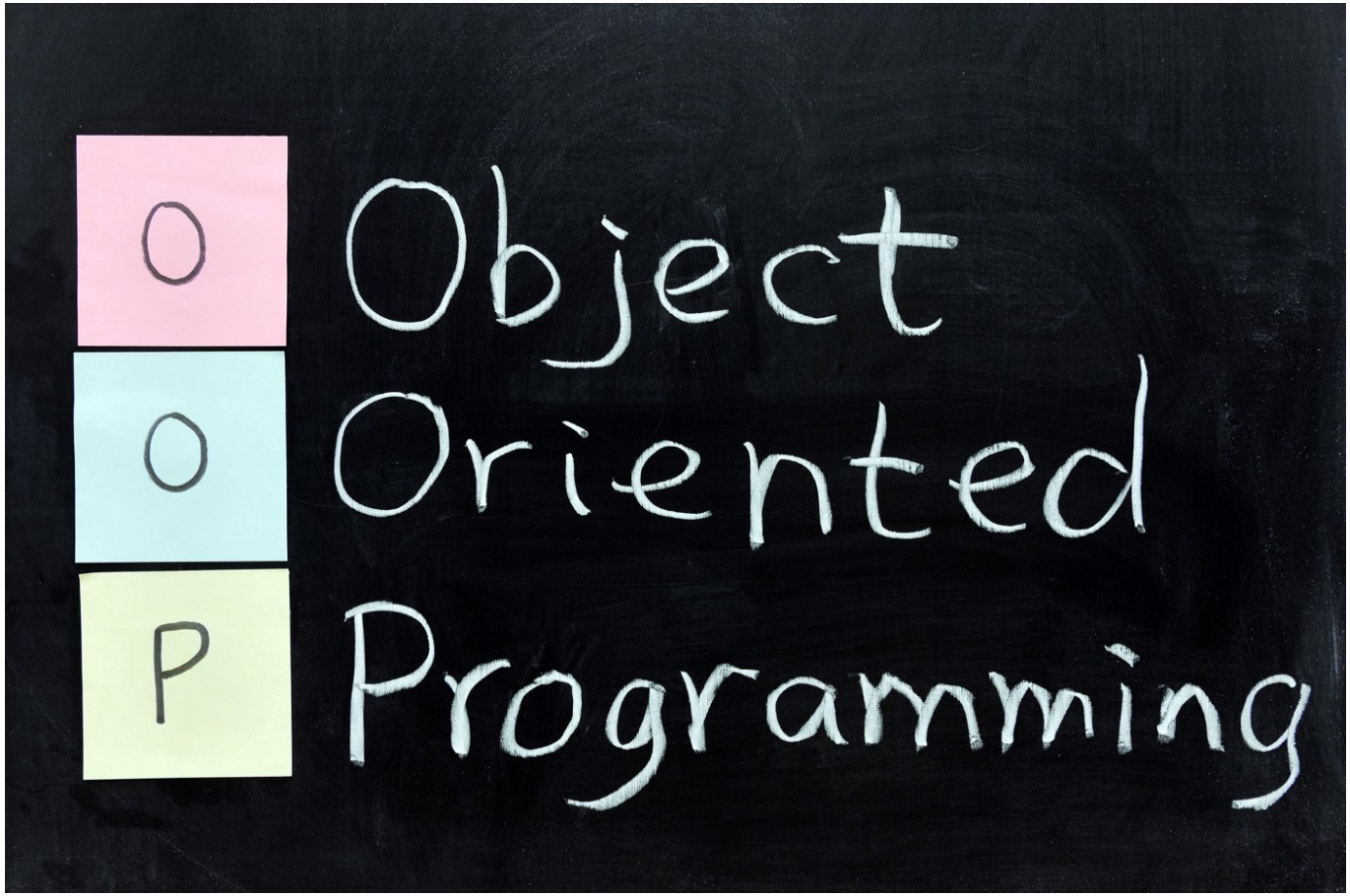
OOPs in SWIFT

G. Abhisek Follow

Jun 9, 2018 · 7 min read



OOP(Object Oriented Programming), a three lettered magical word on which almost all the modern programming language stand. SWIFT, APPLE's trending language is no different from this. OOP concepts are the backbone of SWIFT. What I have marked is many developers fail to correlate the concepts of OOP. It is not that we don't know about OOPs, its just we can't correlate the applications. So guys its a time to revisit basic OOP concepts in a ***SWIFTY*** way.

OOP, We are coming with SWIFT….. Hell Yeah ……!!!

Okay, we will cover up the following basic concepts of OOPs in SWIFTY way:

- Classes
- Objects
- Properties
- Methods
- Access Control
- Encapsulation

- Abstraction
- Inheritance
- Method Overriding
- Method Overloading
- Polymorphism

I don’t believe in technical definitions rather we will understand the concepts in a more realistic way with examples from iOS which implements the same. I will suggest you to try this out on your playground. :]

CLASSES:

Classes can be compared to a real-world group to which certain items or objects or living beings belong and each of these has similar kind of properties as present in the group. Eg — Think of Person as a group or class. Every Person be it, men or women have properties and attributes which are common to both.

Code:

```
class Person {


 // Your Personal attributes and abilities can be defined here.
// Don't write if u are a vampire ;]

}
```

What we did here:

We declared a class.

iOS Example:

UIView, the iOS's main UI class, can be considered as a class.

OBJECTS:

An object is anything that you see which comes from a particular class. Eg — Dog, Cat, Pen, Pencil, etc, everything is an object. From our Person example, men and women are examples of objects which belong to same class i.e Person.

Code:

```
let man = Person() // we created an object of Person
```

What we did here:

We created an object of the class by instantiating it.

iOS Example:

When we create an outlet of UITableView, then we create an object of UITableView class.

PROPERTIES:

Lets again go back to our earlier Person example. As I earlier said that each Person has some common attributes, properties, and functions that they perform are same irrespective across all groups like we belong to a particular gender, color, have age, etc. So these can be called as properties of the Person class.

Technically, properties of a class are common attributes of that class that can be shared across each object which is derived from it.

Code:

```
class Person {


//1
    var age: Int! // These are some of the properties of Person
class

    var gender: String!

    var color: String!

    var maritialStatus: String!


//2
    init(age: Int, gender: String, color: String, maritialStatus:
String) {

    }

}
```

What we did here:

1. We declared some properties of Person class.
2. We are calling an initializer method to initialize the variables(by using ‘init’) if someone instantiates it.

iOS Example:

UIView class has many properties of itself, eg. frame, backgroundColor, isHidden, etc.

METHODS

Methods or Functions are the behavior of the objects of a class. Let us say a person can walk, sing, play, etc irrespective of any object(Man/Woman). These all can be said to be methods/functions of the class.

Code:

Write down the following code in your Person class.

```
func play(sport: String) {

        //Write down how you will make your person instance play.

 }
```

What we did here:

1. We declared a property of Person class i.e play

iOS Example:

UIView class has many methods. Eg. “setNeedsLayout()”, which we call if we want to forcefully reset our UI layout.

Access Controls

Swift provides quite many handy access control levels which are really helpful while writing our code. Some of the access levels provided SWIFT:

- *Open access* and *public access* — Entities with this access level can be accessed within the module that they are defined as well as outside their

module.

- *Internal access* — Entities with this access level can be accessed by any files within the same module but not outside of it.
- *File-private access* — Entities with this access level can only be accessed within the defining source file.
- *Private access* — Entities with this access level can be accessed only within the defining enclosure.

ENCAPSULATION:

Encapsulation is a concept by which we hide data and methods from outside intervention and usage.

Code:

```
class Maths {//1


//2
    let a: Int!

    let b: Int!
```

```
    private var result: Int?


//3
    init(a: Int,b: Int) {

        self.a = a

        self.b = b
    }



//4
    func add() {
        result = a + b
    }



//5
    func displayResult() {
        print("Result - \(result)")
    }

}

let calculation = Maths(a: 2, b: 3)

calculation.add()

calculation.displayResult()
```

What we did here:

1. We declared a Maths class which does up some mathematical calculations.
2. We declared two variables required for inputting values.
3. Initialise the variables.
4. We declare a method to add the two variables
5. And then another method to display the result.

In the above example, we encapsulated the variable “result” by using the access specifier “private”. We hide the data of variable “result” from any outside intervention and usage.

ABSTRACTION:

Abstraction is an OOP concept by which we expose relevant data and methods of an object hiding their internal implementation.

Eg. When we go to a shop to buy a product, we just get the product that we want. The shopkeeper doesn’t tell us about how the product was bought.

We can think this as an example of abstraction.

In our example in encapsulation, we are exposing displayTotal() and add() method to the user to perform the calculations, but hiding the internal calculations.

INHERITANCE:

Inheritance is defined as a process by which you inherit the properties of your parent. Technically, Inheritance is a process by which a child class inherits the properties of its parent class.

Code:

```
class Men: Person {//1

}

//2
let andy = Men(age: 2, gender: "M", color: "White", maritialStatus:
"M")
print(andy.age) // prints 2
```

What we did here:

1. We declared a child class Men which inherits the Person class…..Ya of course men are persons. Wait did you think of something else!! ;]
2. And since Men class inherits from Person parent class, it can also access its properties as it does in the code.

iOS Example:

UIButton class inherits from UIView thus is able to access its properties like backgroundColor, frame, etc.

METHOD OVERLOADING:

Method overloading is the process by which a class has two or more methods with same name but different parameters.

Code:

Write down the following code in Person class. Now we see

```
func play(instrument: String) {

}
```

Now we have two play functions with different arguments. A person can play a music as well as an instrument. Can't he?

METHOD OVERRIDING:

Overriding is the process by which two methods have the same method name and parameters. One of the methods is in the parent class and the other is in the child class.

Code:

```
class Men {

 override init(age: Int, gender: String, color: String,
maritialStatus: String) {
        //initialise men
    }
}
```

What we did here:

Since Men inherit from Person, we can override the init method in Men class so that it would behave differently for the objects of Men along with how it does for objects of Person class.

iOS Example:

When we create a UIViewController subclass, we override the viewDidLoad() method of UIViewController.

POLYMORPHISM:

One of the important aspects of OOP which makes it a hero is the behavior of objects. Objects of the same class can behave independently within the same interface.

Code:

```
class Player {
  let name: String
```

```
  init(name: String) {
    self.name = name
  }

  func play() { }

}

class Batsman: Player {
  override func play() {
    bat()
  }

  private func bat() {
    print("\(name) is batting 🏏")
  }

}

class Bowler: Player {
  override func play() {
    bowl()
  }

  private func bowl() {
    print("\(name) is bowling 🏏")
  }
}

class CricketTeam {
  let name: String
  let team: [Player]
```

```
  init(team: [Player]) {
    self.team = team
  }


  func play() {
    team.forEach { $0.play() }
  }
}


let rohitSharma = Batsman(name: "Rohit Sharma")
let bumrah = Batsman(name: "Jasprit Bumrah")


let indianTeam = CricketTeam(name: "India", team: [rohitSharma,
bumrah])
indianTeam.play()
```

```
Rohit Sharma is batting 🏏
Jasprit Bumrah is bowling 🏏
```

What we did here:

We created a `Player` class and inherited `Batsman` and `Bowler` classes which overrides the `play()` function of their parent class. `CricketTeam` class has a name and consists of different players. We initialised a `bowler` and a `batsman` object and added them to the Indian cricket team.

When Indian team starts playing by invoking `play()`, you can see Rohit Sharma starts batting whereas Jasprit Bumrah starts bowling because the earlier is a batsman whereas the later is a bowler.

For `CricketTeam` class all the players belong to `Player` class. It does not matter which subclasses they do inherit or how they play in the field.

Our almost all day to day coding revolves around the concepts of OOP, and it really needs us to be strong and creative in it. Hope this revisit would have been enjoyable.

I would love to hear from you

You can reach me for any query, feedback or just want to have a discussion by the following channels:

Twitter — @G_ABHISEK

LinkedIn

Gmail: abhisekbunty94@gmail.com

Please feel free to share with your fellow developers.

Programming iOS iOS App Development Oop Swift 4