[day-2]_ Swift_Introduction

* Closure ≈ Anonymous function (which has no name)

variable name

```
Let greetPerson = {
    Print("Hello there")
}
```

→ this function is not taking or returning any value.

{ () → () }

```
func Sample() {
    Print ("Hello there")
}
```

→ greenPerson()

[let greetPerson = {] ((()) → ())

empty braces

doesn't take any argument

doesn't return any argument (Return type)

```
func ~~filterValue~~()
func isValid (num: Int)→{
  Return n%2 == 0 {
}
```

→ {Bool}

```
func isValid (num: Int) → Bool { }
  func filterValues (num:[Int], filter: ((Int)→Bool)){
}
  }
  filtervalues (num: [1,2,3,..7])
```

[variable or closure or nameless function]

* func Sample(num: Int) → Bool {
    return num%2
}

```
var closureFunction = { (num: Int) → Bool in
    return num %2 == 0
}
```

Function
Pass to
Closure

```
func filtervalue (nums: [Int], filter:
        ((Int) -> BOOL)) -> [Int]{
    for num in nums{
            if {filter (num){
  }             filterValues. append (num)
            }
    }
     -> return filteredValues.
}
```

-> filter -> colours -> powerfull
-> unlimited arguments and always return a single values.
-> Clouser function will be untouch; you will change only anonymous function logic

return num %2 == 0]

-> Clobused are used for call back.
* Event Driven Programming

filteredValues = [Int] -> functions

*

```
func Prefix Saluation (names: [Int],      10, 20, 30
        clause: filtervalues)
/*       (String) -> (String
      ↓
  vipul    => Mr. vipul

func Prefix Salutations (names: [Int], completion:
        ([String] -> ()) {
     var processed Names = [String]()
  }  for name in names{
        }  processednames. append ("Mr..."
  }
```

// Swift Classes.

```
Class BankAccount {
    var accountBalance: Float? → default value
    var accountNum: Int ? → value
}
```

→ class with optional value →

[Constructure]

```
→ init( accountNum : Int, acounBa : Float) {
        Self. accountNumbe = accountNumber
        Self. account Balance = account Balance
    }
    func displayAcountInfo() {
        Print ( "... \(Self. acantNumber)")
    }
}
```

```
Let account1 = BankAccount (accu : 111, acca = 1000)
```

* Inheritance

```
Let SavingAccount : BankAccount = SavingAccount
    ( accountNumber : 123, accountBalance : 1000,
    interestRate : 5 )
if let account = savingAcount [as!] SavingAcount {
    Print ( SavingAccount. calculateInterest())
}
```

(as!) → as a type casting from parent to
(Optional Cast) Current type.

```
Let account = SavingAccount [as!] saving.
```

↑ Forcefull type casting

* `var interestRate : Float {`

Computed properties ←

```
    get {
        return (Self. access / 100) * time
    }
    set {
        —
    }
}
```

→ compute themselve

← assign time

// convenience constructure.
↳ method

```
Class BaseClass {
    var name: String
    init (name: String) {
        self.name = name
    }
}
Class ChildClass: BaseClass {
    var age: Int
    var location: String
    init (name: String
}
```

→ convenience → keyword for constructor
↳ overloaded constructor.

Convience ≒ Builder design pattern
[quite similar]

pass by value in structure →

* ~~Generic concepts~~. Pass by value

```
Struct Person {
    var age: Int
}
```

var person1 = Person (age: 10) → changed from 10 to 30

var person2 = person1

person2.age = 30

print (person1.age) → 10

this will pass the value to person2 in difference space.

Structure	=	Pass by value.
Class	=	Pass by Reference.

// Generics in Swift
↳ data type independent.

```
Class MagicClass <T> {
    var value: T
    init (value: T) {
        self.value = value
    }
}
```

* // Extensions

```
var sampleName = "Vipul Shah"
print (sampleName.characters.count)
```

everytime need to write this stuff.

```
Class MyString extends String {
    Public int sizeC) {..
}
```

.java

```
extension String {
    var length: Int {
        get {
            return self.characters.count
        }
    }
}
```

// Sorting

```
let names = ["Vipul", "Akshay", "Rangesh"]
print (names.sorted())
```

// Sorting

```
Class Friend: Comparable {
    var name: String
    init (name: String) {
        self.name = name
    }
    Public static func <(lhs: Friend, rhs: Friend) -> Bool{
        return lhs.name < rhs.name
    }
}
    public static func ==( lhs: Friend, rhs: Friend)->
                                                Bool {
        return lhs.name == rhs.name
    }
}
```

// Subscript

```
enum MealTime {
    case Breakfast
    case Lunch
    case Dinner
}
person.SetFoodItem(.Breacfast, "Pasta")
person.Sol GetFoodItem(.Breakfast) // Pasta
person[.Breacfast] = "Pasta"
```

```
class DailyMeal {
    var meals: [MealTime: String] = [:]

    subscript(mealTime: MealTime) -> String? {
        get {
            return meals[mealTime]
        }
        set {
            meals[mealTime] = newValue
        }
    }
}

var dailyMeal = DailyMeal()
dailyMeal.meals[.Breakfast] = "Pasta"
if let meal = dailyMeal.meals[.Breakfast] {
    print(meal)
}
```

Exercise

```
var people: [[String, String]] = [
    [
        "firstName": "Anil
```

convert (↓) → [ ]

↑ Fullname