KHOA PHAM .VN
KHOA PHAM TRAINING

TRUNG TÂM ĐÀO TẠO TIN HỌC KHOA PHẠM

Website: www.KhoaPham.vn
Facebook: facebook.com/khoapham.vn
Email: khoaphp@yahoo.com
Hotline: 9066 908 907 - 094 276 4080

LẬP TRÌNH SWIFT 2.X

LƯU HÀNH NỘI BỘ | KHOAPHAM.VN

KhoaPham.Vn

Mục Lục

KhoaPham.Vn

KhoaPham.vn

KhoaPham.Vn

I.Cơ bản về lập trinh Swift

Swift là một ngôn ngữ lập trình mới cho phát triển ứng dụng iOS, OS X, Watch OS, Swift có khá nhiều điểm giống với Objective C.

Swift cung cấp các kiểu cơ bản như Int cho các số nguyên, Double và Float cho dồ thực , Bool cho giá trị True hoặc False và String cho chuỗi kí tự. Swift cũng cung cấp 3 kiểu Collection như Array, Set và Dictionary để quản lý dạng danh sách mảng.

Swift sử dụng các biến để lưu trữ và tham chiếu giá trị bởi một tên xác định. Swift sử dụng nhiều những giá trị không thay đổi được gọi là hằng số và mạnh hơn nhiều so với hằng số trong C. Hằng số được sử dụng giúp cho code rõ ràng và an toàn hơn trong lúc bạn làm việc với các giá trị mà không cần thay đổi.

Ngoài các kiểu quen thuộc , Swift còn giới thiệu một kiểu mới hoàn toàn không có trong Objective C, đó là kiểu Tuple. Tuple cho phép bạn tạo và gom nhóm các giá trị không cùng kiểu dữ liệu. Tuple có thể trả ra nhiều giá trị từ một hàm như là một giá trị duy nhất.

Swift cũng giới thiệu các kiểu Optional để xử lý các trường hợp không có giá trị. Optional có thể có một giá trị hoặc không có giá trị. Optional tương tự như sử dụng nil với pointer trong objective C. Optional an toàn và là một trong những tính năng mạnh mẽ nhất của Swift.

Swift muốn bạn phải rõ ràng về các kiểu giá trị trong quá trình code. Điều này cho phép bạn nắm bắt và sửa lỗi càng sớm càng tốt trong quá trình phát triển.

A.Hằng số và biến

Hằng là giá trị không được thay đổi sau khi nó được khai báo.
Biến là giá trị có thể được thay đổi bằng một giá trị khác khi cần.

1.Khai báo hằng và biến

Hằng và biến phải được khai báo trước khi sử dụng. Bạn phải khai báo các hằng với từ khoá là let và biến với từ khoá là var.

```
//Hằng số
let hangso = 10

//Biến
var bien = 20
```

Bạn có thể khai báo nhiều hằng hoặc nhiều biến như sau:

```
//Khai báo nhiều hằng hoặc nhiều biến trên 1 dòng.
var a = 1, b = 2, c = 3
```

Bạn có thể cung cấp các kiểu khi khai báo biến hoặc hằng, để rõ ràng hơn cho kiểu giá trị. Được viết bằng dấu hai chấm và phía sau tên biến hoặc tên hằng.

```
var xinchao:String
```

Ở đây có thể được đọc như sau: “khai báo biến gọi là xinchao có kiểu String”. Như vậy ta chỉ có thể gán chuỗi kí tự vào cho biến xinchao.
Bây giờ ta có thể gán chuỗi kí tự bất kì vào cho biến xinchao:

```
xinchao = "HelloWorld"
```

Bạn có thể khai báo nhiều biến có cùng kiểu như sau:

```
//Khai báo nhiều biến có cùng kiểu trên 1 dòng.
var x, y, z:Double
```

2.Tên Hằng và Tên Biến.

Tên hằng và tên biến có thể chứa hầu hết bất kỳ character(kí tự), bao gồm cả kí tự Unicode.

```
//Tên Hằng và Biến.
let π = 3.14159
let 你好 = "你好世界"
let □□ = "dogcow"
```

Tên hằng và tên biến không thể chứa các kí tự khoảng trắng, ký hiệu toán học, các mũi tên... và không thể bắt đầu bằng một con số.
Khi bạn đã khai báo một hằng hoặc một biến có kiểu nhất định, bạn không thể khai báo lại nữa với cùng tên, hoặc thay đổi kiểu khác.
Bạn có thể thay đổi giá trị của biến có cùng kiểu.

```
//Thay đổi giá trị của biến xinchao từ HelloWorld thành KhoaPham.Vn
xinchao = "KhoaPham.Vn"
```

Với giá trị của hằng thì không thể thay đổi khi đã được khai báo.

3.Print hằng và biến

Bạn có thể print trước kết quả của hằng hoặc biến để xem hoặc kiểm tra.

```
//Print Hằng và Biến.
print(xinchao)
//kết quả: "KhoaPham.Vn"
```

Ở Swift 1 ta có 2 hàm in là print, println nhưng với Swift 2.0 chỉ còn hàm print. Ta có thể dùng print để in ra những thông điệp phức tạp hơn. Bao gồm những các giá trị hiện tại của hằng và biến.

```
//Print thông điệp phức tạp hơn
print("Khoá Học iOS của Trung Tâm \(xinchao)")
//Kết quả: “Khoá Học iOS của Trung Tam KhoaPham.Vn”
```

B.Integer

Là số nguyên. Swift cung cấp Sign và Unsign integer trong mẫu 8, 16, 32 và 64 bit.

1.Giá trị cao nhất trong Interger

Bạn có thể kiểm tra giá trị cao nhất của từng kiểu integer với thuộc tính Max

```
var maxValue:UInt8 = UInt8.max //kết quả : 255
```

2.Int

Bạn không cần phải chọn một kích thước cụ thể nào bởi Int sẽ thay thế hết tất cả. Trừ khi bạn muốn chọn một kích thước cụ thể của integer. Nên sử dụng Int cho các giá trị số nguyên để hỗ trợ đoạn mã nhất quán và khả năng tương thích hơn.

```
var max:Int = Int.max
//Kết quả: 9223372036854775807
```

3.UInt

Bạn không cần phải chọn một kích thước cụ thể nào bởi UInt sẽ thay thế hết tất cả. Trừ khi bạn muốn chọn một kích thước cụ thể của integer. Nên sử dụng UInt cho các giá trị số nguyên dương để hỗ trợ đoạn mã nhất quán và khả năng tương thích hơn.

```
var max2:UInt = UInt.max
//Kết quả: 18446744073709551615
```

C.Kiểu Float và kiểu Double

Là một dạng phân số. Có thể lưu trữ những số thập phân mà kiểu integer không lưu trữ được.Swift cung cấp 2 kiểu là:

- Float(32-bit)
- Double(64-bit)

```
//Giá trị lưu trữ lớn nhất của kiểu float
var float:Float = FLT_MAX   //Kết quả: 3.402823e+38

//Giá trị lưu trữ lớn nhất của kiểu double
var double:Double = DBL_MAX //Kết quả: 1.797693134862316e+308
```

D.Kiểu Safety và Kiểu Inference

Swift là một ngôn ngữ lập trình an toàn. Vì thế Swift muốn bạn phải khai báo rõ ràng về các kiểu giá trị. Nếu có bạn khai báo biến kiểu String thì bạn không thể truyền giá trị kiểu Int hoặc ngược lại. Bởi vì Swift thực hiện kiểm tra kiểu (type check) khi biên dịch và báo hiệu những kiểu không phù hợp là lỗi. Điều này cho phép bạn nắm bắt và sửa lỗi càng sớm càng tốt.

Type check giúp bạn tránh được lỗi khi bạn đang làm việc với các kiểu giá trị khác nhau. Tuy nhiên, không có nghĩa là bạn phải xác định kiểu của mỗi hằng và biến mà bạn khai báo. Nếu bạn không chỉ định các kiểu giá trị, Swift sẽ sử dụng kiểu Inference để tìm ra kiểu thích hợp dựa trên giá trị mà bạn đã cung cấp.
Kiểu Inference đặt biệt hữu ích khi bạn khai báo một hằng hoặc một biến có giá trị ban đầu.

```
//Kiểu Inference sẽ hiểu là kiểu String
var chuoi = "KhoaPham.vn"

//Kiểu Inference sẽ hiểu là kiểu Int
var soInt = 23
```

E.Chuyển đổi kiểu số(Numeric Type Conversion)

Chuyển các kiểu Integer về một kiểu đồng nhất để phù hợp các tình huống sử dụng. Tránh tính trạng tràn dữ liệu, tối ưu hoá bộ nhớ.

1.Chuyển kiểu số nguyên(Integer Conversion)

Ở ví dụ dưới hằng so1 và hằng so2 có 2 kiểu số nguyên khác nhau, vì thế ta phải ép kiểu của hằng so1 sang cùng kiểu với hằng so2 là kiểu Int16

```
let so1: Int8 = 2
let so2: Int16 = 1500
let tong = so2 + Int16(so1)
```

2.Chuyển kiểu Int với Float/Double

Ở ví dụ dưới hằng so3 có kiểu Int và hằng so4 có kiểu Double, vì thế ta phải ưu tiên ép kiểu Int sang Double

```
//Int and Float/Double
let so3 = 2
let so4 = 0.2356
let tong2so = Double(so3) + so4
//Kết quả: 2.2356
```

F.Kiểu Aliases

Thay thế một kiểu hiện có. Được khai báo với từ khoá typealias.

```
//Kiểu Aliases
typealias alias = UInt32
var so5:alias = 5
```

Ta có thể thấy kiểu Aliases có tên alias được nhận kiểu UInt32 và biến so5 có kiểu alias. Thực chất là biến so5 nhận kiểu UInt32 thông qua alias
Từ ví dụ trên ta thấy kiểu aliases hữu ích khi muốn để cập đến một kiểu đã tồn tại với tên khác.

G.Kiểu Boolean

Đây là kiểu trả về hai giá trị True và False.

```
//Kiểu Boolean
let dung = true
let sai = false
```

Thường được sử dụng nhiều trong trường hợp so sánh, câu lệnh if

```
if sai{
    print("Giá trị true")
}else{
    print("Giá trị false")
}
//Kết quả: "Giá trị false"
```

Trong trường hợp bạn dùng câu lệnh if để so sánh giá trị kiểu số:

```
let kieuso = 1
//Trường hợp sai sẽ xuất hiện lỗi
if kieuso{

}

//Trường hợp đúng
if kieuso == 1{

}
```

H.Kiểu Tuple

Là một kiểu mới trong Swift. Kiểu Tuple có thể nhóm các giá trị và các giá trị đó không nhất thiết phải cùng kiểu.

```
//Kiểu Tuple
let tuple = ("khoapham",123456)
```

Bạn có thể nhóm nhiều giá trị khác kiểu lại với nhau.Và ta có thể lấy từng giá trị ra như sau:

```
print("username: \(tuple.0) và password: \(tuple.1)")
//Kết quả: "username: khoapham và password: 123456"
```

Bạn có thể khai báo và lấy giá trị bằng cách sau:

```
let tuple2 = (username: "khoapham", password: 123456)
print("username: \(tuple2.username) và password: \(tuple2.password)")
//Kết quả: "username: khoapham và password: 123456"
```

I.Optional

Bạn có thể sử dụng Optional trong tình huống và giá trị không có.Một Optional có thể có 1 giá trị và nó bằng x hoặc không có một giá trị nào.

```
//Optional
let optional: String? = nil
if optional == nil{
    print("nil")
}
//Kết quả: "nil"
```

J.Error Handling

Error Handling mới được apple nâng cấp vào Swift 2.0. Bạn có thể sử dụng Error Handling để xử lý những lỗi mà bạn có thể gặp phải. Khi một func gặp phải một tính trạng lỗi nó sẽ ném ra lỗi và bạn có thể bắt lỗi và xử lý thích hợp.

```
//Error Handling
func XuatLoi() throws{

}

do{
    try XuatLoi()
}catch{

}
```

II.Toán tử cơ bản

Toán tử là một ký hiệu đặc biệt hoặc cụm từ mà bạn sử dụng để kiểm tra, thay đổi, hoặc kết hợp các giá trị. Ví dụ, các toán tử cộng (+) thêm hai số với nhau (như let i = 1 + 2). Ví dụ phức tạp hơn bao gồm toán tử logic AND && (như *if enteredDoorCode && passedRetinaScan*) và toán tử tăng dần ++ i, đó là một lệnh tắt để tăng giá trị của i lên 1.

Swift hỗ trợ chuẩn toán tử C và cải thiện một số tính năng để loại bỏ các lỗi mã hóa phổ biến. Toán tử gán (=) không trả về một giá trị, để ngăn chặn nó được sử dụng lẫn lộn với toán tử so sánh (==). Toán tử số học (+, -, *, /,% và vv) phát hiện và không cho phép tràn giá trị, để tránh những kết quả bất ngờ khi làm việc với các con số lớn hơn hay nhỏ hơn so với phạm vi cho phép giá trị của các kiểu lưu trphẩyữ chúng. Bạn có thể quyết định tham gia vào hành vi tràn giá trị bằng cách sử dụng các toán tràn Swift, như được mô tả trong Overflow Operators.

Không giống như C, Swift cho phép bạn thực hiện tìm số dư (%) tính trên số phẩy động. Swift cũng cung cấp hai toán tử phạm vi (a ..<b và a …b) không tìm thấy trong C, như một lệnh tắt để thể hiện một loạt các giá trị.
Phần này mô tả các toán tử phổ biến trong Swift. Advanced Operators bao gồm các toán tử nâng cao của Swift, và mô tả làm thế nào để xác định các tuỳ chỉnh toán tử của riêng bạn và thực hiện các tiêu chuẩn vận hành với nhiều tùy chỉnh của riêng bạn.

1.Terminology(Thuật ngữ)

Hầu hết các toán tử một ngôi, hai ngôi, hoặc ba ngôi:

- Toán tử một ngôi hoạt động trên một phần tử dữ liệu duy nhất (như *-a*). Các tiền tố toán tử một ngôi xuất hiện ngay lập tức trước biến của chúng (như *!b*), và các hậu tố toán tử một ngôi xuất hiện ngay sau biến của chúng (như i ++).

- Toán tử hai ngôi hoạt động trên 2 phần tử dữ liệu (như *2 + 3*) và là trung tố vì chúng xuất hiện ở giữa hai phần tử dữ liệu của chúng.

- Toán tử ba ngôi hoạt động trên ba phần tử dữ liệu. Giống như C, Swift chỉ có duy nhất một toán tử ba ngôi, các điều kiện của toán tử ba ngôi (*a? b : c*).
Các giá trị mà các toán tử ảnh hưởng là các toán hạng. Trong biểu thức *1 + 2*, biểu tượng + là một toán tử nhị phân và hai toán hạng của nó là các giá trị 1 và 2.

2.Assignment Operator(Toán tử gán)

Toán tử gán (a = b) khởi tạo hay cập nhật giá trị của a với giá trị của b.

```
let b = 20
var a = 16
a = b
// Kết quả: a = 20 và b = 20
```

Nếu phía bên phải của việc gán là một bộ với nhiều giá trị, các yếu tố của nó có thể được phân tách ra thành nhiều hằng hoặc biến cùng một lúc:
Không giống như các toán tử gán trong C và Objective-C, các toán tử gán trong Swift không tự trả về một giá trị. Tuyên bố sau đây là không hợp lệ:

3.Arithmetic Operators(Toán tử toán học)

Swift hỗ trợ bốn toán tử toán học tiêu chuẩn cho tất cả các kiểu số:

- Phép cộng (+)

```
let (x,y) = (5, 10)
// Kết quả: x = 5, y = 10
```

- Phép trừ (-)
- Phép nhân (*)
- Phân chia (/)

```
//Arithmetic Operators
1 + 2   //kêt quả: 3
3 - 1   //kêt quả: 2
5*5     //kêt quả: 25
8/2     //kêt quả: 4
```

Không giống như các toán tử trong C và Objective-C, các toán tử số học Swift không cho phép các giá trị để tràn theo mặc định. Bạn có thể chọn tham gia vào hành vi tràn giá trị bằng cách sử dụng các toán tràn Swift (chẳng hạn như a & + b). Xem Overflow Operators.
Tử tử cộng còn hỗ trợ nối chuỗi String:

```
"khoa" + "pham" + ".vn"
//Kết quả: khoapham.vn
```

4.Remainder Operator(Toán tử số dư)

Toán tử số dư *(a% b)* tính toán ra bao nhiêu bội số của b sẽ phù hợp với bên a và trả về giá trị được để lại (gọi là phần còn lại).
Đây là cách toán tử số dư hoạt động. Để tính toán 10% 8

```
//Remainder Operator(Toán tử số dư)
10 % 8 //Kết quả: 2

-10 % 8 //Kết quả: -2
```

5.Floating-Point Remainder Calculations(Toán tử số dư thập phân)

Không giống như toán tử số dư trong C và Objective-C, toán tử số dư của Swift cũng có thể hoạt động trên các số dấu chấm động:

```
10%3.6 //Kết quả: 2.8
```

6.Increment and Decrement Operators(Toán tử tăng và giảm dần)

Toán tử tăng dần (++) có nghĩa là i = i + 1
Toán tử giảm dần(--) có nghĩa là i = i -1

```
var a1 = 0

let b1 = ++a1   //b1=1 và a1=1

let c1 = a1++   //c1=1 và a1=2
```

Trong ví dụ trên, let b1 = ++a1 tăng giá trị của a1 trước khi trả về giá trị của nó. Đây là lý do tại sao cả a1 và b1 đều có giá trị mới là 1.
Tuy nhiên, let c1 = a1++ tăng giá trị của a1 sau khi trả về giá trị của c1. Điều này có nghĩa rằng lấy giá trị cũ là 1 và sau đó a1 được cập nhật giá trị bằng 2.
Trừ khi bạn cần các hành vi cụ thể của i ++, khuyên bạn nên sử dụng ++ i và –i trong mọi trường hợp, bởi vì họ có những hành vi mong đợi điển hình của việc sửa đổi i và trả về kết quả.

7.Compound Assignment Operators(Toán tử gán phức hợp)

Giống như C, Swift cung cấp các toán tử gán phức hợp kết hợp phép gán (=) với các toán tử khác. Một ví dụ là toán tử gán thêm (+ =):

```
var a2 = 5
a2 += 3     //Kết quả a2 = 8
```

Biểu thức a2 + = 2 là viết tắt cho a2 = a2 + 2. Hiệu quả cho, việc bổ sung và gán được kết hợp thành một toán tử mà thực hiện cả hai tác vụ cùng một lúc.
Một danh sách đầy đủ của các toán tử gán phức hợp có thể được tìm thấy trong Expressions.

8.Comparison Operators(Toán tử so sánh)

Swift hỗ trợ tất cả chuẩn toán tử so sánh trong C:

- Bằng (a == b)
- Không bằng (a! = B)
- Lớn hơn (a> b)
- Nhỏ hơn (a <b)
- Lớn hơn hoặc bằng (a> = b)
- Nhỏ hơn hoặc bằng (a <= b)

Mỗi toán tử so sánh trả về một giá trị Bool để cho biết hay không câu lệnh là *true*:

```
//Comparison Operators
1 == 1      //Kết quả: true
3 != 5      //Kết quả: true
2 > 1       //Kết quả: true
6 < 10      //Kết quả: true
10 <= 20    //Kết quả: true
10 >= 20    //Kết quả:false
```

Toán tử so sánh thường được sử dụng trong câu lệnh if

9.Ternary Conditional Operator(Toán tử điều kiện tam phân)

Toán tử điều kiện tam phân là một toán tử đặc biệt với ba phần, trong đó có dạng *question ? answer1: answer2*. Nó là một phím tắt để đánh giá một trong hai biểu thức dựa trên một trong hai giá trị *question* là *true* hay *false*. Nếu *question* là *true*, nó lấy *answer1* và trả về giá trị của nó; nếu *false*, nó lấy *answer2* và trả về giá trị của nó.

```
if question {
   answer1
} else {
   answer2
}
```

Ở ví dụ này đoạn code "(Bool ? 40 : 10)" có nghĩa là nếu Bool là true thì lấy 40 ngược lại Bool là false lấy 10. ở đây Bool là false nên kết quả soluong bằng 30.

```
//Ternary Conditional Operator
let tao = 20
let Bool = false
let soluong = tao + (Bool ? 40 : 10)
//Kết quả: soluong = 30
```

10.Nil Coalescing Operator(toán tử kết hợp nil)

Toán tử hợp nhất Nil – Nil coalescing operator *(a ?? b)* tháo bỏ một optional a nếu nó bao gồm một giá trị, hoặc trả về một giá trị mặc định b nếu a là nil. Biểu thức b phải phù hợp với kiểu đang được lưu trữ trong a.

nil coalescing operator là một phím tắt cho mã dưới đây:

```
a != nil ? a! : b
```

Đoạn mã trên sử dụng toán tử ba ngôi và tháo buộc(a!) Để truy cập các giá trị bên trong một gói khi mà không phải là nil, và để trả về một giá trị b. Toán tử hợp nhất Nil cung cấp một cách thanh lịch hơn để đóng gói kiểm tra điều kiện này và unwrapping trong một hình thức ngắn gọn và dễ đọc.

```
let red = "red"
var chuoi:String? //mặc định là nil
var Mau = chuoi ?? red //nếu chuoi bằng nil thì xuất ra red
//Kết quả: "red"
```

11.Range Operators(Toán tử phạm vi)

Swift gồm 2 loạt toán tử là: Closed Range Operator và Half-Open Range Operator

a)Closed Range Operator

Phạm vi toán tử kép kín *(a … b)* định nghĩa một phạm vi đó chạy từ *a* đến *b*, và bao gồm các giá trị *a* và *b*. Giá trị của *a* không lớn hơn *b*.

Toán tử phạm vi khép kín hữu ích khi lặp qua một dãy mà bạn muốn tất cả các giá trị được sử dụng, chẳng hạn như với một vòng lặp for-in:

```
//Toán tử phạm vi
for index in 1...10{
    print("\(index)")   //print từ 1 đến 10
}
```

b)Half-Open Range Operator

Toán tử bán phạm vi *(a .. <b)* định nghĩa một phạm vi chạy từ a đến b, nhưng không bao gồm b. Nó được cho là mở một nửa *(half-open)* bởi vì nó có chứa các giá trị đầu tiên, nhưng không có giá trị cuối cùng của nó. Giống như với toán tử phạm vi khép kín, giá trị của *a* không được lớn hơn *b*.

Toán tử bán phạm vi đặc biệt hữu ích khi bạn làm việc với danh sách dạng zero-based như mảng, nơi mà nó hữu ích để đếm (nhưng không bao gồm) độ dài của danh sách:

```
for index in 1..<3{
    print("\(index)")   //print từ 1 đến 2
}
```

12.Logical Operators(Toán tử logic)

Toán tử logic -Logical operators – chỉnh sửa hoặc kết hợp các giá trị logic Boolean *true* và *false*. Swift hỗ trợ ba chuẩn toán tử logic được tìm thấy dựa trên ngôn ngữ C:

- Logical NOT (!a)
- Logical AND (a && b)
- Logical OR (a || b)

a)Logical NOT Operator(Toán tử Logic NOT)

Toán tử logic NOT – logical NOT operator – (! a) đảo ngược một giá trị Boolean để *true* trở thành *false*, và *false* trở thành *true*.

Toán tử logic NOT là một toán tử tiền tố, và sẽ xuất hiện ngay trước khi giá trị nó hoạt động, mà không cần bất kỳ khoảng trắng nào. Nó có thể được đọc là “not a”, như đã thấy trong các ví dụ sau đây:

```
//Toán tử Logic NOT
let khongdung = false
if !khongdung{
        print("KhoaPham.Vn")
}
//Kết quả: "KhoaPham.Vn"
```

13.Logical AND Operator(Toán tử Logic AND)

Toán tử logic AND *(a && b)* tạo ra các biểu thức logic mà cả hai giá trị phải *true* cho kết quả cộng chung cũng là *true*.

Nếu một trong hai giá trị là *false*, kết quả cộng chung cũng sẽ là *false*. Trong thực tế, nếu giá trị đầu tiên là false, giá trị thứ hai thậm chí sẽ không được đánh giá, bởi vì nó không thể làm thay đổi sự kết quả cộng chung tương đương với *true*. Điều này được gọi là đánh giá ngắn mạch – short-circuit evaluation.
Ví dụ này xem xét hai giá trị *Bool* và chỉ cho phép truy cập nếu cả hai giá trị là true:

```
//Toán tử Logic AND
let checkUsername = true
let checkPassword = true
if checkUsername && checkPassword{
    print("Đăng nhập thành công")
}else{
    print("Đăng nhập thất bại")
}
//Kết quả: "Đăng nhập thành công"
```

14.Logical OR Operator(Toán tử Logic OR)

Toán tử logic OR (a || b) là một toán tử được viết bởi hai dấu gạch liền kề. Bạn sử dụng nó để tạo ra các biểu thức logic trong đó chỉ có một trong hai giá trị là true thì những kết quả cộng chung là true.
Giống như toán tử logic AND, toán tử logic OR sử dụng đánh giá ngắn mạch để xem xét biểu thức của nó. Nếu phía bên trái của một biểu thức logic OR là *true*, bên phải không được đánh giá, bởi vì nó không thể thay đổi kết quả của biểu thức tổng thể.

```
//Toán tử Logic OR
let checkUsername = true
let checkPassword = false
if checkUsername || checkPassword{
    print("Đăng nhập thành công")
}else{
    print("Đăng nhập thất bại")
}
//Kết quả: "Đăng nhập thành công"
```

III.Kiểu String and Kiểu Character

Một chuỗi là một tập hợp có sắp xếp của những kí tự, như là “hello world” hay “albatross”.Chuỗi trong Swift biểu diễn bằng kiểu *String*, chúng lần lượt biểu diễn một tập hợp các giá trị của kiểu ký tự (Character).

Kiểu *String* và *Character* của Swift cung cấp nhanh, cách phù hợp với mã thống nhất (Unicode) để làm việc với văn bản trong mã code của bạn.Cú pháp để tạo ra các chuỗi và thao tác đỡ nặng nề hơn và dễ đọc hơn, với cú pháp chuỗi ký tự cũng tương tự như C.Nối chuỗi cũng đơn giản như cách cộng hai chuỗi với nhau với toán tử +. và tính biến đổi chuỗi được quản lý bằng cách chọn giữa một hằng hoặc một biến, giống như bất kỳ giá trị nào khác trong Swift.

Với cú pháp đơn giản, kiểu String của Swift rất nhanh trong việc thực hiện các thao tác với chuỗi.Mỗi chuỗi bao gồm các ký tự Unicode mã hóa độc lập, và cung cấp hỗ trợ cho việc truy cập các ký tự đại diện trong mã Unicode khác nhau.
Bạn cũng có thể sử dụng chuỗi để chèn hằng, biến, chữ, và các biểu thức thành chuỗi dài, trong một quá trình được gọi là suy chuỗi. Điều này làm cho nó dễ dàng để tạo ra các giá trị chuỗi tùy chỉnh để hiển thị, lưu trữ và in ấn.

A.String Literal(Chuỗi kí tự)
Chuỗi kí tự là một chuỗi cố định của văn bản bao quanh bởi một cặp dấu ngoặc kép(“”)

```
//Chuỗi kí tự
let chuoiString = "KhoaPham.Vn"
```

B.Khởi tạo chuỗi
Cách kiểm tra chuỗi rỗng:

```
//Chuỗi rỗng
let chuoirong = ""          //kết quả: ""
let chuoirong2 = String()   //kết quả: ""
```

```
//Kiểm tra chuỗi rỗng
if chuoirong.isEmpty{
    print("Đây là chuỗi rỗng")
}
//Kết quả: "Đây là chuỗi rỗng"
```

C.String Mutability(Biến đổi chuỗi)
Bạn có thể thay đổi chuỗi cho một biến có kiểu String bằng cách gán vào cho biến đó một giá trị kiểu String.

```
//Thay đổi chuỗi
var bienchuoi = "Khoa"
bienchuoi += "Pham.Vn"

var bienchuoi2 = "Trung Tâm Tin Học"
bienchuoi2 = "KhoaPham.Vn"
```

D.Character
Là một kí tự trong một chuỗi.

```
//Character
for character in "KhoaPham".characters{
    print(character)
}
//Kết quả: K h o a P h a m
```

E.Interpolation(nội suy chuỗi)
Là một cách xây dựng chuỗi String mới từ hỗn hợp các hằng, biến,..

```
//Nội suy chuỗi
let chuoi = "KhoaPham"
let tuoi = 28
let thongtin = "Tên: \(chuoi), tuổi: \(tuoi)"
//Kết quả: "Tên: KhoaPham, tuổi: 28"
```

F.Counting Character
Để lấy số lượng giá trị trong một chuỗi hàm sau:

```
//Counting Character
var ten = "KhoaPham.Vn và WePro.Vn"
print("Số Lượng: \(ten.characters.count)")
```

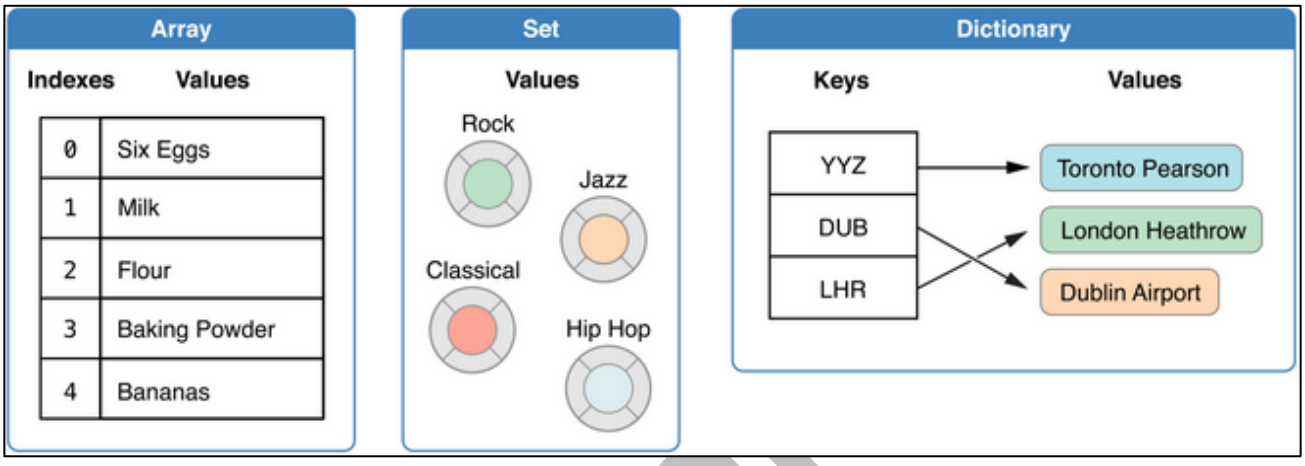
Để so sánh 2 chuỗi với nhau, ta dùng toán tử bằng(==), toán tử không bằng(!=)

```
//So Sánh
let chuoi1 = "KhoaPham"
let chuoi2 = "KhoaPham"
if chuoi1 == chuoi2{
    print("Hai chuỗi giống nhau")
}
//Kết quả: "Hai chuỗi giống nhau"
```

IV.Collection Type

Swift cung cấp cho chúng ta 3 kiểu Collection là : Array, Set, Dictionary

- Array lưu trữ số thứ tự của các giá trị có cùng kiểu.
- Set thứ tự các giá trị duy nhất.
- Dictionary lưu trữ các giá trị cùng kiểu có thứ tự, truy xuất thông qua một định danh duy nhất.



A.Array(mảng)

Lưu trữ các giá trị cùng kiểu theo một danh sách có thứ tự. Các giá trị có thể truy xuất theo vị trí trong Array.

```
//Array
//Cách 1:
var mang:[Int] = [1, 2, 3]

//Cách 2:
var mang2 = [Int]()
mang2.append(1)
mang2.append(2)
mang2.append(3)
```

Các thao tác cơ bản trong Array:

```
//Cách thêm phần tử cho mang
mang.append(4)
//Kết quả: [1,2,3,4]

//Lấy giá trị
print("Số: \(mang[1])")
//Kết quả: "Số: 2"

//Xoá vị trí phần tử trong mang
mang.removeAtIndex(2)   //Vị trí 2 là số 3
print(mang)
//Kết quả: [1,2,4]

//Thêm một phần tử vào mang tại vị trí nhất định
mang.insert(5, atIndex: 2)
//Kết quả: [1,2,5,4]
```

1.Iterating Over an Array(Duyệt qua một mảng)

```
//Duyệt qua một mang
for phantu in mang{
    print(phantu)
}
//Kết quả: 1
//Kết quả: 2
//Kết quả: 5
//Kết quả: 4
```

Nếu bạn muốn lấy giá trị và vị trí của phần tử thì bạn có thể như sau:

```
for (vitri, giatri) in mang.enumerate(){
    print("Phần tử \(vitri + 1): \(giatri)")
}
// Phần tử 1: 1
// Phần tử 2: 2
// Phần tử 3: 5
// Phần tử 4: 4
```

B.Set

Set là một kiểu Collection mới có trong Swift 2.0.
Set lưu trữ những giá trị khác nhau có cùng kiểu nhưng không có thứ tự rõ ràng.Bạn có thể sử dụng để thay thế Array khi thứ tự của các giá trị là không quan trọng, hoặc khi bạn muốn các giá trị chỉ xuất hiện một lần.

1.Thao tác cơ bản

```
//khai báo và thêm phần tử
//Cách 1:
var set = Set<String>()
set.insert("Chó")
set.insert("Mèo")
//Cách 2:
var set2:Set<String> = ["Cúc", "Lan", "Hồng", "Huệ"]

//Đếm số phần tử trong Set
print("Số Phần Tử: \(set2.count)")
//Kết quả: "Số Phần Tử: 4"
```

ở trường hợp xoá phần tử ta nên thêm hàm if để kiểm tra xem giá trị đó có trong Set mà ta xoá không:

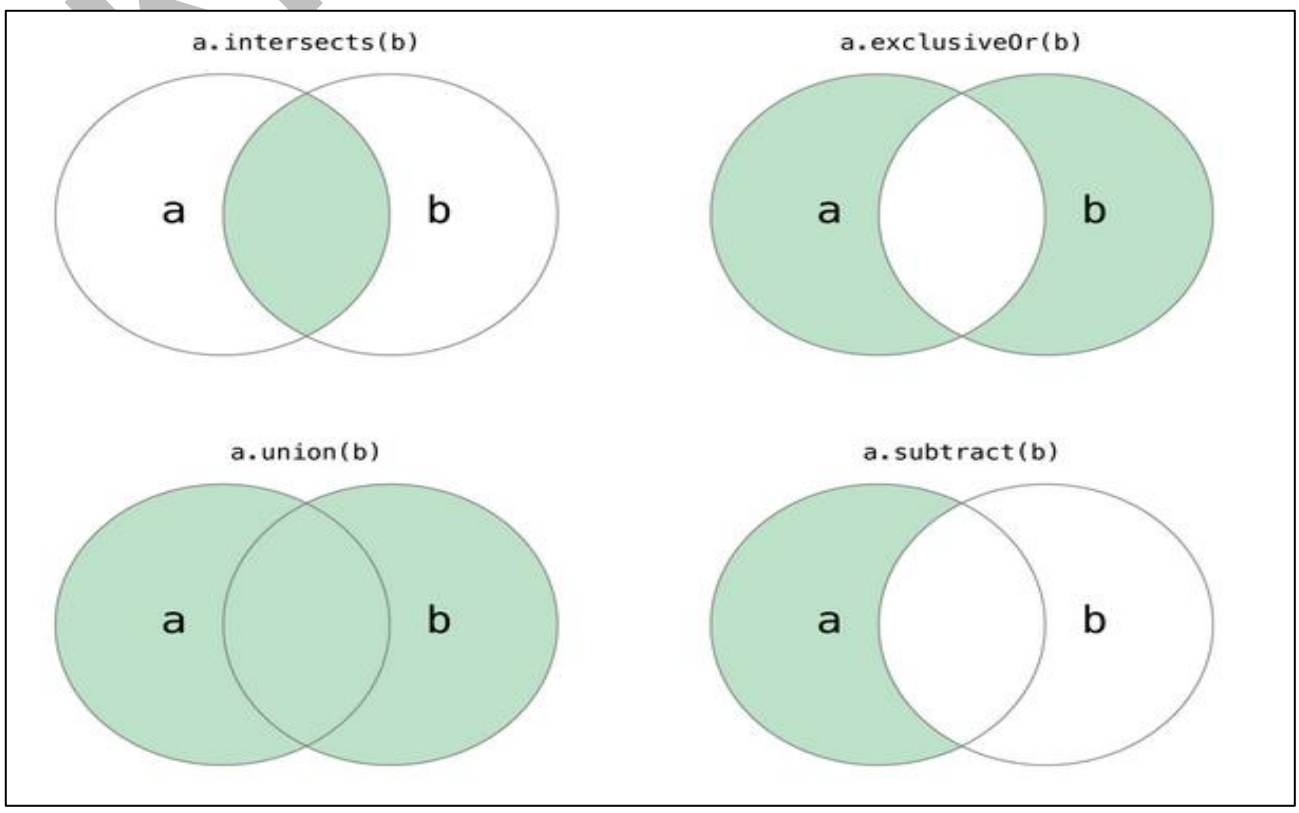
```
//Remove phần tử
if let xoa = set2.remove("Lan"){
    print("Đã Xoá: \(xoa)")
}else{
    print("Không Có Phần Tử")
}
//Kết quả:"Đã Xoá: Lan"
```

Set có một hàm để ta có thể kiểm tra xem phần tử có trong Set hay không:

```
//Kiểm tra có phần tử trong Set
if set2.contains("Cúc"){
    print("Đã Có")
}else{
    print("Chưa Có")
}
//Kết quả: "Đã Có"
```

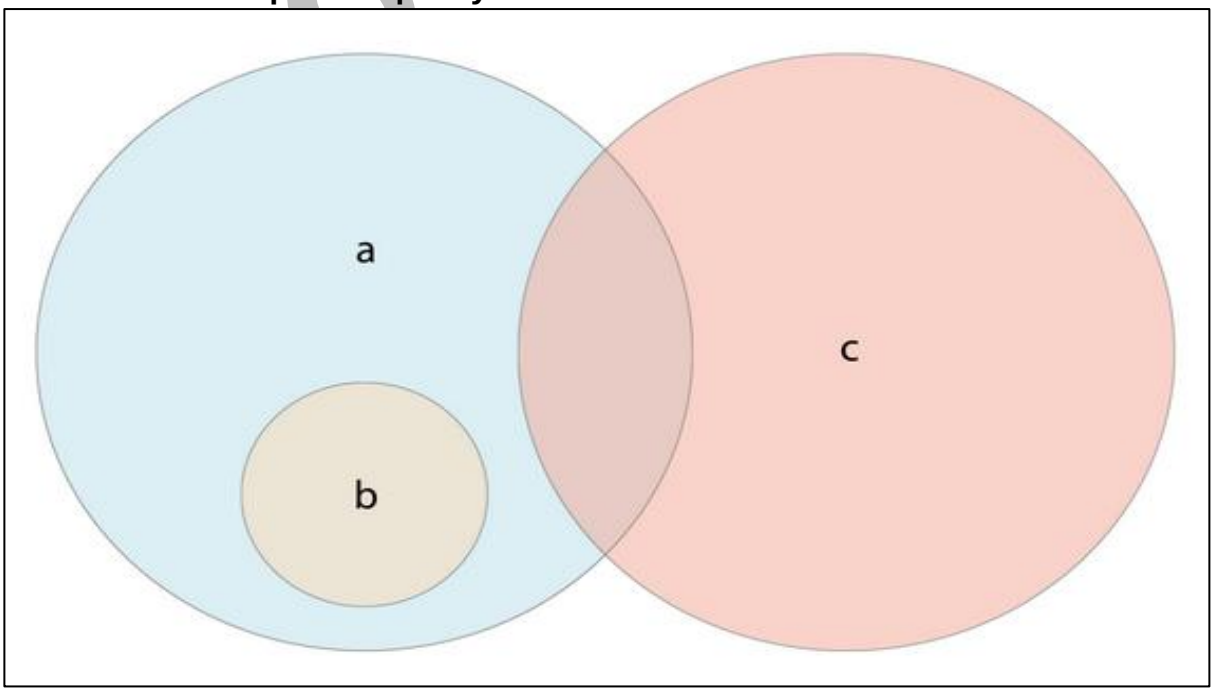
2.Performing Set Operations

Bạn có thể sử dụng Set theo các toán tử cơ bản như kết hợp hai Set , xác định những giá trị giống trong 2 Set...



```
let SoLe:Set = [1,3,5,7,9]
let SoChan:Set = [0,2,4,6,8]
let giatri:Set = [0,4,9]

//Kết Hợp 2 Set lại với nhau
SoLe.union(SoChan).sort()
//Kết quả: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

//Hợp những phần tử giống nhau
SoLe.intersect(SoChan).sort()
//Kết quả: []

//Những phần tử riêng biệt của Set SoLe so với Set giatri
SoLe.subtract(giatri).sort()
//Kết quả: [1,3,5,7]

//Hợp 2 Set lại và bỏ những phần tử giống nhau
SoLe.exclusiveOr(giatri).sort()
//Kết quả: [0,1,3,4,5,7]
```

3.Set Membership and Equality



Hình trên cho ta thấy có 3 Set a,b,c chồng chéo lên nhau. Set b nằm trong set a và Set a giao nhau với Set b ta sẽ có những quan hệ sau:

```
let a:Set = ["Hồng", "Cúc", "Huệ", "Lan", "Hướng Dương"]
let b:Set = ["Cúc", "Huệ", "Lan"]
let c:Set = ["Hồng", "Hướng Dương", "Sen", "Súng"]

//Kiểm tra b nằm trong a
b.isSubsetOf(a)
//Kết quả: true

//Kiểm tra a bao b
a.isSupersetOf(b)
//Kết quả: true

//Kiểm tra b có giao với c
b.isDisjointWith(c)
//Kết quả: true
```

C.Dictionary

Lưu trữ nhiều giá trị cùng kiểu, giá trị của các kiểu giống như Array nhưng không có thứ tự nhất định. Mỗi giá trị được liên kết thông qua một key duy nhất. Bạn sử dụng một từ điển khi bạn cần tìm ra giá trị dựa trên nhận dạng của chúng, theo cách giống như một từ điển thực tế được sử dụng để tìm định nghĩa cho một từ cụ thể. Những thao tác cơ bản trong Dictionary:

```
//Khai báo Dictionary
//Cách 1:
var dictionary = [Int: String]()
dictionary[1] = "KhoaPham"
//Cách 2:
var dictionary2: [String: String] = ["True":"Đúng","False":"Sai","Do":"Làm","Why":"Tại sao"]

//Lấy phần tử
print(dictionary2["True"]!)
//Kết quả: "Đúng"

//Đếm phần tử trong Dictionary
print("Số phần tử: \(dictionary2.count)")
//Kết quả: "Số phần tử: 4"

//Thêm phần tử
dictionary2["Go"] = "Đi"
print(dictionary2)
//Kết quả: "[Go: Đi, False: Sai, True: Đúng, Do: Làm, Why: Tại sao]"

//Xoá phần tử
```

V.Control Flow

A.For-In

Bạn có thể sử dụng vòng lặp để duyệt qua các phần tử, các phần tử trong mảng, các kí tự trong chuỗi...
Vòng lặp duyệt từ 0 đến 3:

```
for index in 0...3{
    print("KhoaPham\(index)")
}
//Kết quả: KhoaPham0
//Kết quả: KhoaPham1
//Kết quả: KhoaPham2
//Kết quả: KhoaPham3
```

KhoaPham.Vn

Vòng lặp duyệt qua các phần tử trong mảng:

```
//Duyệt qua các phần tử trong mảng
let danhsachten = ["Ronaldo","Messi","Ronaldinho"]
for ten in danhsachten{
    print(ten)
}
//Kết quả: Ronaldo
//Kết quả: Messi
//Kết quả: Ronaldinho
```

Vòng lặp duyệt qua các ký tự trong chuỗi:

```
for kytu in "KhoaPham".characters{
    print(kytu)
}
//Kết quả: K
//Kết quả: h
//Kết quả: o
//Kết quả: a
//Kết quả: P
//Kết quả: h
//Kết quả: a
//Kết quả: m
```

B.While Loops

1.While

```
//While
var so = 0
var ketqua = 6
while so < ketqua{
    so = so + 1
    print(so)
}
//Kết quả: 1
//Kết quả: 2
//Kết quả: 3
//Kết quả: 4
//Kết quả: 5
//Kết quả: 6
```

2.Repeat-While

Repeat-While là vòng lặp sẽ hoạt động khi điều kiện While vẫn còn thoả.

```
//Repeat-While
var a = 0
repeat{
    a = a + 1
    print(a)
} while a < 4
//Kết quả: 1
//Kết quả: 2
//Kết quả: 3
//Kết quả: 4
```

3.If

Câu lệnh điều kiện đơn giản. Khi điều kiện đúng nó sẽ thực hiện một tập lệnh này, khi điều kiện sai sẽ thực hiện một tập lệnh khác.

```
//IF
var x = 2
if x == 2{
    print("x = 2")
}else{
    print("x != 2")
}
//Kết quả: "x = 2"
```

```
var y = 3
if y == 1{
    print("y == 1")
}else if y == 2{
    print("y == 2")
}else if y == 3{
    print("y == 3")
}else{
    print("y > 3")
}
//Kết quả: "y == 3"
```

4.Swich

Một lệnh switch xem xét một giá trị và so sánh nó với một số mô hình phù hợp có thể. Sau đó nó thực hiện một khối mã code thích hợp, dựa trên mô hình đầu tiên mà sự phù hợp thành công. Một lệnh switch cung cấp một thay thế cho lệnh if để hồi đáp cho nhiều trạng thái tiềm năng. Mỗi Swich gồm nhiều Case.

```
var z = 3
switch z{
case 0:
    print("z = 0")
case 1:
    print("z = 1")
case 2:
    print("z = 2")
case 3:
    print("z = 3")
default:
    print("z > 3")
}
//Kết quả: "z = 3"
```

5.Where

Một switch case có thể sử dụng where để kiểm tra điều kiện bổ sung.

```
let wh = (1,4)
switch wh{
case let(x,y) where x == y:
    print("\(x) = \(y)")
case let(x,y) where x > y:
    print("\(x) > \(y)")
case let(x,y) where x < y:
    print("\(x) < \(y)")
}
//Kết quả: "1 < 4"
```

C.Control Transfer Statements

Gồm : continue, break, fallthrough, return, throw.

1.Continue

Câu lệnh continue nói với một vòng lặp dừng hành động nó đang thực hiện và bắt đầu lại tại điểm bắt đầu của việc duyệt qua vòng lặp tiếp theo.

```
let chuoi = "say you do"
var catchuoi = ""
for kitu in chuoi.characters{
    switch kitu{
    case "a","o"," ":
        continue
    default:
    catchuoi.append(kitu)
    }
}
print(catchuoi)
//Kết quả: "syyud"
```

2.Break

Lệnh break kết thúc việc thực hiện của toàn bộ lệnh chuyển điều khiển ngay lập tức. Lệnh break có thể được sử dụng bên trong lệnh switch, hoặc lệnh lặp khi bạn muốn chấm dứt việc thực hiện của lệnh switch hoặc lệnh lặp sớm hơn nếu không phải là trường hợp.

```
//Break
let bre = 1
switch bre{
case 1:
    print("bre = 1")
default:
    break
}
```

3.Fallthrough

```
let soF = 5
var chuoiF = ""
switch soF{
case 2, 5, 6, 7:
    chuoiF += "Ket Qua \(soF)"
    fallthrough
default:
    chuoiF += " Chay"
}
print(chuoiF)
//Kết quả: "Ket Qua 5 Chay"
```

VI.Function

Hàm(function) là khối mã khép kín dùng để thực hiện một tác vụ cụ thể. Bạn cho một tên hàm để xác định những gì nó làm, và tên này được sử dụng để "gọi" hàm để thực hiện tác vụ khi cần thiết.

Cú pháp hàm thống nhất của Swift là đủ linh hoạt để thể hiện bất cứ điều gì từ một hàm C-style đơn giản không có tên tham số đến một phương thức Objective-C-style phức tạp với các tên tham số hàm địa phương và bên ngoài cho mỗi tham số. Các thông số có thể cung cấp các giá trị để mặc định để đơn giản hóa các cuộc gọi chức năng và có thể được thông qua như tham số in-out, mà sửa đổi một biến được khi hàm đã hoàn tất việc thực hiện.

Mỗi hàm trong Swift đều có một kiểu, bao gồm các kiểu tham số và kiểu trả về của hàm. Bạn có thể sử dụng kiểu này giống như bất kỳ kiểu nào khác trong Swift, việc này dễ dàng để truyền vào hàm cũng như tham số cho các hàm khác, và để trả về hàm này từ hàm khác. Hàm này cũng có thể được viết trong các hàm khác để đóng gói các hàm hữu ích trong phạm vi hàm lồng nhau.

A.Defining and Calling Functions

```
//Khai Báo
func say()->String{
    let name = "KhoaPham"
    return name
}

//Gọi hàm
print(say())
//Kết quả: "KhoaPham"
```

B.Function Parameters and Return Values

C.Func không có kiểu trả về

```
//func không có kiểu trả về
func hello(){
    let hel = "Hello, KhoaPham"
}
//Kết quả: "Hello, KhoaPham"
```

D.Func có kiểu trả về

```
//func có kiểu trả về
func hello2()->String{
    let hel = "Hello, KhoaPham"
    return hel
}
//nhận kiểu trả về
var chuoi = hello2()
print(chuoi)
//Kết quả: "Hello, KhoaPham"
```

E.Func nhận tham số và có kiểu trả về

F.Func nhận nhiều tham số và có kiểu trả về

```
//func nhận tham số kiểu String và trả về 1 biến có kiểu String
func sayHello(chuoi:String) -> String{
    let chuoi2 = "Hello, \(chuoi)"
    return chuoi2
}
var string = sayHello("Lam")
print(string)
//Kết quả: "Hello, Lam"
```

```
//func nhận nhiều tham số và trả về biến có kiểu String
func sayThongTin(chuoi:String, tuoi:Int) -> String{
    let chuoi1 = "Hello, \(chuoi) Tuổi: \(tuoi)"
    return chuoi1
}
var string2 = sayThongTin("Lam", tuoi: 22)
print(string2)
//Kết quả: "Hello, Lam Tuổi 22"
```

G.Func đếm kí tự trong chuỗi

```
//func đếm kí tự trong chuỗi
func demkitu(chuoi:String) ->Int{
    return chuoi.characters.count
}
var so = demkitu("KhoaPham")
//Kết quả: 8
```

H.Func trả về nhiều giá trị

```
//func trả về nhiều giá trị
func timMinMax(mang:[Int])->(min:Int,max:Int){
    var valueMin = mang[0]
    var valueMax = mang[0]
    for value in mang[1..<mang.count]{
    if value < valueMin{
    valueMin = value
}else if value > valueMax{
    valueMax = value
    }
    }
    return (valueMin,valueMax)
}
var MinMax = timMinMax([2,3,4,5,6,7,8,10])
print("Min:\(MinMax.min), Max:\(MinMax.max)")
//Kết quả: "Min: 2, Max: 10"
```

I.Func nhiều tham số Variadic

```
//func nhiều tham sô Variadic
func tinhTrungBinh(number:Double...) -> Double{
    var tong:Double = 0
    for so in number{
    tong += so
    }
    return tong / Double(number.count)
}
var soDouble:Double = tinhTrungBinh(2,3,6,7,8)
//Kết quả: 5.2
```

J.Func InOut

```
//func InOut
func doiViTri(inout so1:Int, inout _ so2:Int){
    let temporary = so1
    so1 = so2
    so2 = temporary
}
var a = 200
var b = 150
doiViTri(&a, &b)
print("so a = \(a), b = \(b)")
//Kết quả: a = 150, b = 200
```

K.Biến bằng Func

```
//Biến = func
func tong2So(a:Int, b:Int) ->Int{
    return a+b
}
func tich2So(a:Int, b:Int) ->Int{

    return a*b
}
var PhepTong = tong2So
print("Tong: \(PhepTong(3,b: 5))")  //kết quả: 8
PhepTong  = tich2So
print(PhepTong(3,b: 5))            //kết quả: 15
```

VII.Closure

Closures là những khối độc lập chứa các chức năng có thể được truyền qua và sử dụng trong mã code của bạn. Closures trong Swift tương tự như blocks trong C và Object-C, và như lambas trong một số ngôn ngữ lập trình khác.

Closures có thể thu nạp và lưu trữ giá trị tham chiếu đến bất kỳ hằng số và biến nào đó từ những bố cảnh mà chúng được định nghĩa. Điều này được gọi là đóng trên những hằng số và biến, do đó tên “closures”. Swift xử lý tất cả công việc quản lý bộ nhớ của việc thu nạp cho bạn.

Hàm toàn cục và hàm lồng nhau, như đã giới thiệu trong Function, thực sự là trường hợp đặc biệt của closures. Closures lấy một trong ba hình thức:

- Hàm toàn cục là closures có một tên và không nắm bắt được bất kỳ giá trị nào cả.
- Hàm lồng nhau là closures có một tên và có thể nắm bắt các giá trị từ hàm kèm theo của chúng.
- Biểu thức closure là closures không rõ tên được viết theo cú pháp nhẹ mà có thể nắm bắt các giá trị từ bối cảnh xung quanh.

Biểu thức closure của Swift có một phong cách trôi chảy và rõ ràng, với việc tối ưu hóa để khuyến khích ngắn gọn, cú pháp lộn xộn-miễn phí trong các kịch bản phổ biến. Những tối ưu hóa bao gồm:

- Suy luận các tham số và giá trị trả về các loại từ bối cảnh
- Implicit returns from single-expression closures
- Khai báo tên đối số ngắn gọn
- Trailing cú pháp closure

1.Closure Expressions

Hàm lồng nhau, như đã giới thiệu ở Nested Functions, là thuận tiện trong việc đặt tên và định nghĩa các khối độc lập của mã nguồn như là một phần của một hàm lớn hơn. Tuy nhiên, đôi khi nó là hữu ích để viết các phiên bản ngắn hơn của cấu trúc giống như hàm không có khai báo và tên đầy đủ. Điều này đặc biệt đúng khi bạn làm việc với các hàm mà phải đặt vào các hàm khác như một hoặc nhiều đối số của chúng.

Biểu thức Closure – Closure expressions – là một cách để viết ngắn gọn trực tiếp, tập trung vào cú pháp. Biểu thức Closure cung cấp một vài tối ưu hóa cú pháp cho việc viết closures dưới dạng rút gọn mà không mất sự rõ ràng hay mục đích. Các ví dụ biểu closure dưới đây minh hoạ những tối ưu hóa bằng cách chỉnh lại một ví dụ duy nhất của hàm được sắp xếp trên một số lần lặp, ví dụ thể hiện chức năng tương tự trong một cách gọn gàng hơn.

2.Closure Expression Syntax

```
{ (parameters) -> return type in
    statements
}
```

Cú pháp biểu thức closure có thể sử dụng các tham số hằng, các tham số biến, và tham số inout. Các giá trị mặc định không được cung cấp. Các tham số lệnh biến thiên có thể không được sử dụng nếu bạn đặt tên tham số lệnh biến thiên và đặt nó ở cuối cùng trong danh sách tham số. Tuples có thể được sử dụng như là các kiểu

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
func backwards(s1: String, s2: String) -> Bool {
    return s1 > s2
}
var reversed = names.sort(backwards)
//Closure
reversed = names.sort({ (s1: String, s2: String) -> Bool in
    return s1 > s2
})
//Kết quả: false
//Kết quả: true
//Kết quả: true
//Kết quả: true
//Kết quả: false
//Kết quả: true
//Kết quả: true
//Kết quả: true
//Kết quả: false
```

tham số và các kiểu trả về.

VIII.Enumeration(Liệt Kê)

Một liệt kê – *enumeration* – định nghĩa một kiểu phổ biến cho một nhóm các giá trị liên quan và cho phép bạn làm việc với các giá trị trong một cách an toàn kiểu trong mã code của bạn.
Nếu bạn đã quen thuộc với C, bạn sẽ biết rằng C enumerations gán những tên có liên quan cho một tập hợp các giá trị số nguyên. Liệt kê trong Swift linh hoạt hơn nhiều, và không cần phải cung cấp một giá trị cho từng phần tử thuộc liệt kê. Nếu một giá trị (được biết đến như một giá trị "thô" – "raw") được cung cấp cho mỗi phần tử, giá trị có thể là một chuỗi, một ký tự hoặc một giá trị của bất kỳ kiểu số nguyên hoặc kiểu số phẩy động.
Ngoài ra, các phần tử liệt kê có thể chỉ định các giá trị liên quan của bất kỳ kiểu nào để được lưu giữ cùng với mỗi giá trị phần tử khác, giống như hợp thức hoặc biến thức trong các ngôn ngữ khác. Bạn có thể định nghĩa một tập hợp chung của các phần tử liên quan như là một phần của một *enumeration*, mỗi phần tử trong số đó có một bộ các giá trị khác nhau của các kiểu thích hợp liên kết với nó.
Liệt kê trong Swift là lớp đầu tiên theo đúng nghĩa của nó. Chúng áp dụng nhiều tính năng truyền thống được hỗ trợ chỉ bởi các lớp, chẳng hạn như các thuộc tính tính toán để cung cấp thêm thông tin về giá trị hiện tại của liệt kê – *enumeration*, và phương thức *thể hiện – instance* - để cung cấp các chức năng liên quan đến các giá trị liệt kê biểu diễn. Enumerations cũng có thể định nghĩa trình khởi tạo để cung cấp một giá trị phần tử ban đầu; có thể được mở rộng để mở rộng chức năng của chúng vượt ra ngoài thực hiện ban đầu của chúng; và có thể phù hợp với các giao thức để cung cấp chức năng tiêu chuẩn chức năng.

1.Enumeration Syntax

```
//khai báo enum
//cách 1
enum testEnum{
    case Trai
    case Phai
    case Tren
    case Duoi
}
//cách 2
enum testEnum2{
    case Dong,Tay,Nam,Bac
}
```

2.Matching Enumeration Values with a Switch Statement

KhoaPham.vn

Ta có thể kết hợp kiểu enumeration với Switch như sau:

```
var timduong = testEnum.Phai
switch timduong{
case .Trai:
   print("Bên Trái")
case .Phai:
   print("Bên Phải")
case .Tren:
   print("Ở Trên")
case .Duoi:
   print("Ở Dưới")
//Kết quả: "Bên Phải"
```

IX.Classes and Structures

Lớp (class) và cấu trúc (structure) được tạo ra với cùng mục đích chung, những cấu trúc linh hoạt đó trở thành các khối xây dựng của mã chương trình của bạn. Bạn định nghĩa các thuộc tính và phương thức để thêm chức năng cho các lớp và cấu trúc của bạn bằng cách sử dụng chính xác cú pháp tương tự như đối với các hằng, biến, và các hàm.

Không giống như các ngôn ngữ lập trình khác, Swift không yêu cầu bạn phải tạo ra giao diện độc lập và các tập tin thực hiện cho các lớp và cấu trúc tùy chỉnh. Trong Swift, bạn định nghĩa một lớp hoặc một cấu trúc trong một tập tin duy nhất, và các giao diện mở rộng để lớp hoặc cấu trúc đó tự động làm sẵn cho các mã code khác để sử dụng.

1.Comparing Classes and Structures

Class và Structure trong Swift có nhiều điểm chung. Cả hai có thể:
- Định nghĩa các thuộc tính(Property) để lưu trữ các giá trị.
- Định nghĩa các phương thức(Method) để cung cấp chức năng.
- Định nghĩa subscripts để cung cấp quyền truy cập vào các gía trị của chúng sử dụng cú pháp subscript.
- Định nghĩa bộ khởi tạo để thiết lập trạng thái ban đầu của chúng.
- Để mở rộng triển khai chức năng của chúng ra bên ngoài thực thi mặc định.
- Phù hợp với các giao thức để cung cấp những chuẩn chức năng của một kiểu nhất định.

Lớp – class – có khả năng mở rộng, cấu trúc -structure – thì không:
- Tính kế thừa cho phép một lớp thừa hưởng các đặc tính của lớp khác.
- Type casting cho phép bạn kiểm tra và giải thích kiểu của một thể hiện lớp trong thời gian chạy.
- Deinitializers cho phép thể hiện của một lớp giải phóng bất kỳ mã nguồn nào đó mà nó được gán.

- Reference counting cho phép nhiều hơn một tham chiếu đến một thể hiện lớp.

```
//Khai Báo
class AClass{

}

struct AStruct {

}
```

Ví dụ: Struct và Class:

```
struct AStruct {
    var a = 0
    var b = 3
}
class AClass{
    var BStruct = AStruct()
    var kieuBool = true
    var a = 0.3
    var chuoi:String?
}
```

B.Class and Structure Instances

Bạn có thể nhận các giá trị trong class và struct:

```
//Lấy giá trị trong Class và Struct
var BStruct = AStruct()
var BClass = AClass()
print("Lấy giá trị b: \(BStruct.b)")
print("Lấy giá trị a: \(BClass.a)")
//Kết quả: "Lấy giá trị b: 3"
//Kết quả: "Lấy giá trị a: 0.3"
```

Thay đổi giá trị trong Class và Struct:

```
//Thay đổi giá trị
BStruct.a = 5
BStruct.b = 10
print("Giá trị a: \(BStruct.a) và b: \(BStruct.b)")
//Kết quả: "Giá trị a: 5 và b: 10"
```

X.Subscript

Lớp (class) , cấu trúc (structure) , và kiểu liệt kê (enumaration) có thể định nghĩa *subscript*, đó là các phím tắt để truy cập vào các phần tử của một tập hợp, danh sách, hoặc chuỗi. Bạn sử dụng subscripts để thiết lập và lấy giá trị của *index* mà không cần phương thức riêng biệt cho thiết lập và phục hồi.

A.Subscript Syntax

Subscripts cho phép bạn truy vấn các thể hiện của một kiểu bằng cách viết một hoặc nhiều giá trị trong dấu ngoặc vuông sau tên thể hiện. Cú pháp của chúng tương tự cho cả cú pháp phương thức thể hiện và cú pháp thuộc tính tính toán. Bạn viết định nghĩa subscript với từ khóa subscript, và chỉ định một hoặc nhiều tham số đầu vào và một kiểu trả về, trong cùng một cách như các phương thức thể hiện. Không giống như các phương thức thể hiện, kí hiệu có thể được đọc-ghi hay chỉ-đọc. Hành vi này được truyền đạt bởi một getter và setter trong cùng một cách như đối với thuộc tính tính toán:

```
//Subscript đọc và ghi
subscript(index: Int) -> Int{
    get{
        //Return giá trị thích hợp
    }
    set(giatri){
        //Thực hiện những
    }
}
//subscript chỉ đọc
subscript(index: Int)->Int{
    //Return giá trị thích hợp
}
```

Ví dụ Subcript chỉ đọc:

```
//subscript chỉ đọc
struct AStruct {
    let a:Int
    subscript(index: Int)-> Int{
        return a * index
    }
}
let b = AStruct(a: 3)
print("\(b[3])")
//Kết quả: "9"
```

XI.Inheritance(Kế Thừa)

Một Class có thể kế thừa (inherit) method, property, và các đặc tính khác từ các Class khác. Khi một Class kế thừa từ một lớp khác, Class kế thừa được gọi là một Class con (*subclass*), và Class được nó kế thừa gọi là Class cha (superclass) của nó. Kế thừa là một hành vi cơ bản để phân biệt các lớp từ các kiểu khác nhau trong Swift.

Class trong Swift có thể gọi và truy cập các method, property, và subscript thuộc class cha của chúng và có thể cung cấp các phiên bản ghi đè của bản thân chúng với những method, property, và subsript để tinh chỉnh hay thay đổi hành vi của chúng. Swift giúp đảm bảo việc ghi đè của bạn là chính xác bằng cách kiểm tra các định nghĩa ghi đè có một định nghĩa phù hợp với class cha.

Class cũng có thể thêm các quan sát thuộc tính với các thuộc tính kế thừa để được thông báo khi giá trị của một thuộc tính thay đổi. Quan sát thuộc tính có thể được thêm vào bất kỳ thuộc tính nào, cho dù ban đầu nó được định nghĩa như là một thuộc tính lưu trữ hoặc thuộc tính tính toán.

A.Defining a Base Class

```
//Kế thừa
class AClass{
    var a = 5
    var name: String{
        return "KhoaPham.Vn, số: \(a)"
    }
    func tinh(){

    }
}
let BClass = AClass()
print("\(BClass.name)")
//Kết quả: "KhoaPham.Vn, số: 5"
```

XII.Property

Property là thuộc tính của một đối tượng.

```
class Nguoi {
    var ten:String = ""
    var tuoi:Int = 0
}
```

XIII.Method

Method là những function liên quan tới đối tượng đó.

```
class Nguoi {
    var ten:String = ""
    var tuoi:Int = 0
    func di(){
        ...
    }
    func an(){
        ...
    }
    func ngu(){
        ...
    }
}
```

A.Subclass(lớp con)

Lớp con (Subclassing) là hành động của một lớp mới dựa trên một lớp hiện có. Các lớp con thừa hưởng những đặc tính từ lớp hiện có, sau đó bạn có thể tinh chỉnh. Bạn cũng có thể thêm các đặc tính mới cho các lớp con.

Để chỉ ra rằng một lớp con có một lớp cha, viết tên lớp con trước tên lớp cha, cách nhau bằng dấu hai chấm:

```
class SomeSubclass: SomeSuperclass {
    // subclass definition goes here
}
```

Ví dụ:

```
//SubClass
class CClass: AClass{
    var b = 10
    func tinh()-> Int {
        return a + b
    }
}
let getClass = CClass()
print("\(getClass.tinh())")
//Kết quả: "15"
```

B.Overriding(ghi đè)

Bạn có thể ghi đè lên một thể hiện hay thuộc tính lớp thừa kế để cung cấp getter và

```
//Ghi đè
class classOveride:CClass{
    override var name: String{
        return super.name + ", WePro.Vn, số: \(b)"
    }
}
let over = classOveride()
print("\(over.name)")
//Kết quả: "KhoaPham.Vn, số: 5, WePro.Vn, số: 10"
```

setter tùy chỉnh của riêng bạn cho thuộc tính đó, hoặc để thêm quan sát thuộc tính để cho phép thuộc tính ghi đè để quan sát khi giá trị thuộc tính ngầm thay đổi.

C.Preventing Overrides(Ngăn ghi đè)

Bạn có thể ngăn chặn một phương thức, thuộc tính, hoặc subscript bị ghi đè bằng cách đánh dấu nó như là final – cuối cùng. Làm điều này bằng cách viết các sửa đổi final trước từ khoá giới thiệu của phương thức, thuộc tính, hoặc subscript (như final var, final func, final class func, và final subscript).
Bất kỳ nỗ lực để ghi đè lên một thức phương thức, thuộc tính, hoặc subscript trong một lớp con được báo cáo như là một lỗi biên dịch thời gian. Phương thức, thuộc tính, hay subscript mà bạn thêm vào một lớp trong một phần mở rộng cũng có thể được đánh dấu như là cuối cùng trong định nghĩa của phần mở rộng.
Bạn có thể đánh dấu toàn bộ một lớp như là cuối cùng bằng cách viết chỉnh sửa final trước từ khoá class trong lớp nó định nghĩa (final class). Bất kỳ thực thi trên lớp con, một lớp cuối cùng, được báo cáo như là một lỗi thời gian biên dịch.

XIV.Initialization(Khởi tạo)

Init là bắt buộc khởi tạo những thuộc tính của đối tượng, khi đối tượng đó được khởi tạo.

```
class Nguoi {
    var ten:String = ""
    var tuoi:Int = 0
    init(Ten:String, Tuoi:Int){
        ten = Ten
        tuoi = Tuoi
    }
}
var nguoi2:Nguoi = Nguoi()  //Missing argument for parameter 'Ten' in call

var nguoi1:Nguoi = Nguoi(Ten: "Le Lam", Tuoi: 26)
```

Ở ví dụ trên khi khởi tạo đối tượng nguoi1 thì nó bắt buộc ta phải truyền 2 tham số Ten và Tuoi vì trong class Nguoi ta khởi tạo hàm init. Còn nguoi2 sẽ báo lỗi do ta chưa truyền vào 2 tham số Ten và Tuoi.

XV.Deinitialization(Hàm Huỷ)

Một *deinitializer* được gọi là ngay trước khi một thể hiện lớp được giải phóng. Bạn viết deinitializers với từ khóa *deinit*, tương tự như cách bộ khởi tạo – intializers – được viết với từ khóa *init*. Deinitializers chỉ có sẵn trên các kiểu lớp.

```
class Nguoi {
    var ten:String = ""
    var tuoi:Int = 0
    init(Ten:String, Tuoi:Int){
        ten = Ten
        tuoi = Tuoi
    }
    deinit{
        print("Giá trị nil")
    }
}
var nguoi1:Nguoi = Nguoi(Ten: "Le Lam", Tuoi: 26)
var nguoi2:Nguoi? = Nguoi(Ten: "KhoaPham", Tuoi: 28)    //Nguoi2 có giá trị

nguoi2 = nil    //đã được giải phóng và thực thi các câu lệnh trong deinit
```

XVI. Optional Chaining

Optional Chaining là một quá trình để truy vấn và gọi các property, method, và subscript vào một *optional* mà hiện tại có thể là *nil*. Nếu *optional* có chứa một giá trị,

```
class Nguoi{
    var Ten:String = "Lam"
    var Tuoi:Int = 26
}
class Nha{
    var nguoi:Nguoi?
}

//Gọi gián tiếp
let nha = Nha()
if let nguoi = nha.nguoi?.Ten{
    print("Tên: \(nguoi)")
}else{
    print("nguoi có giá trị nil")
}
//kết quả: "nguoi có giá trị nil"

//Gọi trực tiếp
nha.nguoi = Nguoi()
if let nguoi = nha.nguoi?.Ten{
    print("Tên: \(nguoi)")
}else{
    print("nguoi không có giá trị")
}
//kết quả: "Tên: Lam"
```

thuộc tính, phương thức, hoặc kí hiệu gọi thành công; nếu optional là nil, thuộc tính, phương thức, hoặc kí hiệu gọi trả về nil. Nhiều truy vấn có thể được nối lại với nhau, và toàn bộ chuỗi thất bại nếu bất kỳ liên kết trong chuỗi là nil.

Khi gọi gián tiếp do nha.nguoi?.Ten ta khai báo có dấu “?” nên giá trị là nil
Còn khi gọi trực tiếp nha.nguoi = Nguoi() lúc này đã có giá trị

KhoaPham.Vn

XVII.Handling Error

Là quá trình xử lý lỗi và khắc phục từ các lỗi trong chương trình. Swift đã cung cấp first-class hỗ trợ việc ta xuất lỗi, bắt lỗi, tuyên truyền và khắc phục lỗi khi chạy.

A.Representing Errors

Enumeration đặc biệt thích hợp để mô hình hoá một nhóm các điều kiện lỗi, với các giá trị liên quan để biết thêm về bản chất của một lỗi được truyền vào.

```
enum loiMaHoa:ErrorType{
    case Empty
    case Short
}
```

B.Throwing Errors

Chỉ ra functionn hoặc method có thể ném ra lỗi.

```
func mahoaPassword(str:String, password:String) throws ->String{
    guard password.characters.count > 0 else{throw loiMaHoa.Empty}

    guard password.characters.count >= 6 else{throw loiMaHoa.Short}

    let mahoa = password + str + password
    return String(mahoa.characters.sort())
}
```

C.Catching and Handling Errors

```
do {
    try function that throws
    statements
} catch pattern {
    statements
}
```

```
do{
        let chuoimahoa = try mahoaPassword("Lam", password: "123456")
        print(chuoimahoa)
    }catch loiMaHoa.Empty{
        print("Bo Trong")
    }catch{
        print("Loi Phat Sinh")
    }
```

ở đây nếu str = “” thì sẽ xuất ra lỗi Empty.

KhoaPham

XVIII.Tổng hợp những điểm khác trong Swift 2.0

A.Defer

```
//defer
func printABC(a:String, b:String, c:String){
    print(a)
    defer{
    print(b)
    }
    print(c)
}
printABC("A", b: "B", c: "C")
//Kết Ouả : “A” “C” “B”
```

B.Repeat-While

```
//Repeat-While
var a = 0
repeat{
        a = a + 1
        print(a)
    } while a < 4
//Kết quả: 1
//Kết quả: 2
//Kết quả: 3
//Kết quả: 4
```

C.Continue

```
let chuoi = "say you do"
var catchuoi = ""
for kitu in chuoi.characters{
        switch kitu{
    case "a","o"," ":
        continue
    default:
        catchuoi.append(kitu)
        }
}
print(catchuoi)
//Kết quả: "syyud"
```

D.Where

```
//where
let wh = (1,4)
switch wh{
case let(x,y) where x == y:
    print("\(x) = \(y)")
case let(x,y) where x > y:
    print("\(x) > \(y)")
case let(x,y) where x < y:
    print("\(x) < \(y)")
}
//Kết quả: "1 < 4"
```

E.Fallthrough

```
//Fallthrough
let soF = 5
var chuoiF = ""
switch soF{
case 2, 5, 6, 7:
    chuoiF += "Ket Qua \(soF)"
    fallthrough
default:
    chuoiF += " Chay"
}
print(chuoiF)
//Kết quả: "Ket Qua 5 Chay"
```

F.Guard

```
guard a == b else{
print("Khác Nhau")
return
}
```

G.Set

1.Khai Báo

```
//Cách 1:
var set = Set<String>()
set.insert("Chó")
set.insert("Chó")
//Cách 2:
var set2:Set<String> = ["Cúc", "Lan", "Hồng", "Huệ"]
```

2.Đếm Phần tử

```
//Đếm số phần tử trong Set
print("Số Phần Tử: \(set2.count)")
```

3.Xoá Phần tử

```
if let xoa = set2.remove("Lan"){
    print("Đã Xoá: \(xoa)")
}else{
    print("Không Có Phần Tử")
}
//Kết quả:"Đã Xoá: Lan"
```

4.Kiểm tra phần tử

```
if set2.contains("Cúc"){
    print("Đã Có")
}else{
    print("Chưa Có")
}
//Kết quả: "Đã Có"
```

5.Toán tử cơ bản

```
let SoChan:Set = [0,2,4,6,8]
let giatri:Set = [0,4,9]

//Kết Hợp 2 Set lại với nhau
SoLe.union(SoChan).sort()
//Kết quả: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

//Hợp những phần tử giống nhau
SoLe.intersect(SoChan).sort()
//Kết quả: []

//Những phần tử riêng biệt của Set SoLe so vơi Set giatri
SoLe.subtract(giatri).sort()
//Kết quả: [1,3,5,7]

//Hợp 2 Set lại và bỏ những phần tử giống nhau
SoLe.exclusiveOr(giatri).sort()
//Kết quả: [0,1,3,4,5,7]
```

6. Set Membership and Equality

```
let b:Set = ["Cúc", "Huệ", "Lan"]
let c:Set = ["Hồng", "Hướng Dương", "Sen", "Súng"]

//Kiểm tra b nằm trong a
b.isSubsetOf(a)
b.isSubsetOf(a)

//Kiểm tra a bao b
a.isSupersetOf(b)
//Kết quả: true

//Kiểm tra b có giao với c
b.isDisjointWith(c)
//Kết quả: true
```

H.Handling Error

1.Enum

```
//Enum
enum loiMaHoa:ErrorType{
    case Empty
    case Short
}
```

2.Throws

```
//Func có thể phát sinh lỗi
func mahoaPassword(str:String, password:String) throws ->String{
    guard password.characters.count > 0 else{throw loiMaHoa.Empty}

    guard password.characters.count >= 6 else{throw loiMaHoa.Short}

    let mahoa = password + str + password
    return String(mahoa.characters.sort())
}
```

3.Do-Catch

```
//thực hiện bắt lỗi
do{
    let chuoimahoa = try mahoaPassword("Lam", password: "123456")
        print(chuoimahoa)
    }catch loiMaHoa.Empty{
        print("Bo Trong")
    }catch{
        print("Loi Phat Sinh")
}
```

Để học khoá đầy đủ và chuyên sâu hơn bạn có thể tham gia khoá học lập trình iOS tại trung tâm đào tạo tin học KhoaPham.
Khoá học iOS cơ bản:
http://khoapham.vn/khoa-hoc-laptrinhios.html
Khoá học iOS nâng cao:
http://khoapham.vn/lap-trinh-ios-nang-cao.html

KhoaPham.Vn


KHOA PHAM .VN
KHOA PHAM TRAINING
TRUNG TÂM ĐÀO TẠO TIN HỌC KHOA PHẠM
CHUYÊN ĐÀO TẠO LẬP TRÌNH THIẾT KẾ WEBSITE ĐỒ HỌA VÀ ỨNG DỤNG DI ĐỘNG
90 Lê Thị Riêng P.Bến Thành, Q.1, TP.HCM - www.KhoaPham.vn - 0966 908 907
CÁC KHÓA ĐÀO TẠO
Khóa học Lập trình PHP & MySQL
Khóa học Lập trình iOS
Khóa học Lập trình Android
Thiết kế Đồ họa đa truyền thông
Khóa học Lập trình Facebook
Khóa học Lập trình Wordpress
Khóa học Lập trình Joomla
Khóa học Lập trình ASP.NET & SQL Server
Khóa học Lập trình Zend Framework 1.X
Khóa học Lập trình Zend Framework 2.X
Khóa học Lập trình Windows Phone
Khóa học Lập Trình Thiết kế Game
CÁC MÔN CHUYÊN ĐỀ
Lập trình jQuery (jQuery Master)
HTM- CSS JAVASCRIPT
jQuery- Bootstrap NodeJs
Thiết kế website với Dreamweaver
Thiết kế website với JOOMLA
Tạo Shop với VIRTUEMART
HỌC TRONG 3 THÁNG ĐỂ CÓ MỘT NGHỀ VỮNG CHẮC
Có Tuyển Sinh Viên Thực Tập
CÁC DỊCH VỤ TẠI KHOAPHAM.VN
Thiết kế website.
Thiết kế các ứng dụng di động.
Đào tạo kỹ năng CNTT cho cá nhân/doanh nghiệp
Website: www.KhoaPham.vn
Hotline: 0966 908 907
Facebook: facebook.com/khoapham.vn
Support miễn phí: http://KhoaPham.vn/forum/