



# Làm việc với collection type

Cài đặt và lập trình căn bản  
[cuong@techmaster.vn](mailto:cuong@techmaster.vn)

# Map



- Các operator chuyển đổi array trong swift gồm có : Map, Filter, Flatmap, Reduce, Compact Map
- **Map:** Lặp qua một collection và áp dụng một thao tác cho mỗi phần tử trong collection đó

*map nhận các closure là các argument và trả về kết quả như cách thực hiện vòng lặp thông thường*

# Ví dụ về Map

```
let numberArray = [1, 2, 3, 4, 5]
var squareArray: [Int] = []
// cách truyền thống
for number in numberArray{
    squareArray.append(number * number)
}

//Cach 1:
let squareArray1 = numberArray.map { (value: Int) -> Int in
    return value * value
}

//Cach 2
let squareArray2 = numberArray.map { (value: Int) in
    return value * value
}

//Cach 3
let squareArray3 = numberArray.map { value in
    value * value
}

//Cach 4
let squareArray4 = numberArray.map { $0 * $0 }

print(squareArray, squareArray1, squareArray2, squareArray3, squareArray4)
```

# Filter



- **Filter:** Duyệt qua collection và trả về một mảng chứa các phần tử đáp ứng điều kiện.

*// cách truyền thống*

```
var filterArray1: [Int] = []  
for number in numberArray {  
    if (number % 2 == 0) {  
        filterArray1.append(number)  
    }  
}
```

*// sử dụng filter*

```
let filterArray2 = numberArray.filter { $0 % 2 == 0 }
```

```
print(filterArray1, filterArray2)
```

# Reduce

- **Reduce:** Kết hợp tất cả các phần tử trong collection để tạo một giá trị mới

*// cách truyền thống*

```
var sum = 0
for number in numbers {
    sum += number
}
```

*//Reduce*

```
let sum1 = numbers.reduce(0, { $0 + $1 })
let sum2 = numbers.reduce(0, +)
print(sum, sum1, sum2)
```

*// có thể sử dụng với string*

```
let stringArray = ["one", "two"]
let oneTwo = stringArray.reduce("", +)
print(oneTwo)
```

# Flatmap

- ***Flatmap** giúp chúng ta đưa tất cả các dữ liệu trong các tập con về 1 tập duy nhất*

```
let arrayInArray = [[1, 2, 3], [4, 5, 6]]
```

*// cách truyền thống*

```
var resultArray: [Int] = [ ]  
for array in arrayInArray {  
    resultArray += array  
}  
print(resultArray)
```

*// sử dụng flatMap*

```
let flattenedArray = arrayInArray.flatMap{ $0 }  
print(flattenedArray)
```

*// hàm flatMap xóa nil khỏi collection*

```
let people: [String?] = ["A", nil, "B", nil, "C"]  
let validPeople = people.flatMap { $0 }
```

```
print(validPeople)
```

# Compact Map

- **Compact Map:** Tương tự với Map, Compact Map cũng lặp qua một collection và áp dụng thao tác cho mỗi phần tử trong collection đó nhưng kết quả trả về là một mảng không có nil, còn map trả về kết quả là một mảng có chứa nil

```
let possibleNumbers = ["1", "2", "three", "///4///", "5"]
```

```
let mapped: [Int] = possibleNumbers.map { str in Int(str) ?? 0 }  
print(mapped) // [Optional(1), Optional(2), nil, nil, Optional(5)]
```

```
let compactMapped: [Int] = possibleNumbers.compactMap { str in Int(str) }  
print(compactMapped) // [1, 2, 5]
```

```
let flattenedArray = possibleNumbers.flatMap{ $0 }  
print(flattenedArray)
```

# Chaning

- Chúng ta có thể kết hợp nhiều phương thức chuyển đổi mảng để ra được kết quả mong muốn khi thao tác với mảng

*// tính tổng bình phương của tất cả các số chẵn từ mảng của các mảng*

```
let arrayInArray = [[1, 2, 3], [4, 5, 6]]
```

```
let arrayValue = arrayInArray.flatMap{ $0 }.filter{$0 % 2 == 0}
```

```
let sumArray = arrayValue.map{ $0 * $0 }.reduce(0, +)
```

```
print(sumArray) // 56
```

*// Bước làm:*

*// đầu tiên hàm flatMap{ \$0 } gộp các mảng con: [1, 2, 3, 4, 5, 6]*

*// sau đó hàm filter{ \$0 % 2 == 0 } lấy ra các số chẵn: [2, 4, 6]*

*// tiếp theo map { \$0 \* \$0 } thực hiện bình phương: [4, 16, 36]*

*// cuối cùng hàm reduce(0, +) sẽ tính tổng:  $4 + 16 + 36 = 56$*





# Class và Struct

Cài đặt và lập trình căn bản  
[cuong@techmaster.vn](mailto:cuong@techmaster.vn)

# Class



- **Swift** là ngôn ngữ kế thừa ngôn ngữ **C** và **Objective-C**, nó vừa là một ngôn ngữ hướng thủ tục, vừa là một ngôn ngữ hướng đối tượng.
- **Class** chính là khái niệm của các ngôn ngữ lập trình hướng đối tượng
- **Class** có các thuộc tính và phương thức, về bản chất phương thức (method) được hiểu là một hàm của lớp.
- Từ một lớp, có thể tạo ra các đối tượng.

```
class <tên Class>: <Kế thừa Class cha, nếu có> {  
    // Khai báo thuộc tính  
    // Khởi tạo phương thức  
}
```

# Khai báo class với khởi tạo không tham số

- **init** là trình khởi tạo - một phương thức (**method**) mà **class** sẽ gọi đến khi chúng ta tạo một đối tượng **Person**.

```
class Rectangle1 {  
  
    // thuộc tính chiều rộng, chiều cao  
    var width: Int = 5  
    var height: Int = 10  
  
    // Constructor (khởi tạo) mặc định (Không tham số)  
    // (Được sử dụng để tạo ra đối tượng)  
    init() {  
    }  
  
    // Phương thức dùng để tính diện tích hình chữ nhật.  
    func getArea() -> Int {  
  
        var area = self.width * self.height  
        return area  
    }  
}
```

# Khởi tạo đối tượng Rectangle1

---

```
// Tạo một đối tượng Rectangle1
// thông qua Constructor mặc định: init()
var rec1 = Rectangle1()

// In ra width, height.
print("rec1.width = \$(rec1.width)")
print("rec1.height = \$(rec1.height)")

// Gọi phương thức để tính diện tích.
print("area = \$(rec1.getArea())")
```

# Khai báo class với khởi tạo có tham số

```
class Rectangle2 {  
  
    // thuộc tính chiều rộng, chiều cao  
    var width: Int  
    var height: Int  
  
    // Một Constructor có 2 tham số.  
    // (Được sử dụng để tạo ra đối tượng)  
    // self.width trỏ tới thuộc tính (property) width của class.  
    init (width: Int, height: Int) {  
        self.width = width  
        self.height = height  
    }  
  
    // Phương thức dùng để tính diện tích hình chữ nhật.  
    func getArea() -> Int {  
  
        var area = self.width * self.height  
        return area  
    }  
}
```

# Khởi tạo đối tượng Rectangle2

---

```
// Tạo đối tượng Rectangle2
// thông qua Constructor có 2 tham số: init(Int,Int)
var rec2 = Rectangle2(width: 10, height: 15)

// In ra width, height.
print("rec2.width = \$(rec2.width)")
print("rec2.height = \$(rec2.height)")

// Gọi phương thức để tính diện tích.
print("area = \$(rec2.getArea())")
```

# Deinitializer

- Deinitializer cho phép hàm khởi tạo của một class giải phóng bất cứ tài nguyên nào mà nó đã gán.
- Một hàm deinitializer được gọi ngay trước khi instance của một class được giải phóng
- Hàm deinit chỉ có sẵn trong class

```
class D {  
    deinit {  
        print("Deinit ")  
    }  
}
```

```
var d: D? = D()  
d = nil
```

# Struct là gì



- Trong **Swift**, **Struct** (cấu trúc) là một kiểu giá trị đặc biệt, nó tạo ra một biến để lưu trữ nhiều giá trị riêng lẻ, mà các giá trị này có liên quan tới nhau
- Ví dụ về một nhân viên bao gồm: mã nhân viên, tên nhân viên, chức vụ
- Chúng ta hoàn toàn có thể tạo ra 3 biến để lưu các trường giá trị của một nhân viên. Tuy nhiên chúng ta nên tạo ra một Struct để gộp tất cả thông tin vào một biến duy nhất.

```
struct <tên Struct>: <kế thừa 1 hoặc nhiều giao thức> {  
    // Khai báo thuộc tính  
    // Khai báo phương thức  
}
```



# Khai báo struct

```
struct Employee {  
  
    // khai báo thuộc tính  
    var empNumber: String  
    var empName: String  
    var position: String  
  
    // Constructor (khởi tạo)  
    init(empNumber:String, empName:String, position:String) {  
        self.empNumber = empNumber;  
        self.empName = empName;  
        self.position = position;  
    }  
}
```

# Khởi tạo một đối tượng Employee

---

```
func test_EmployeeStruct() {  
  
    // Tạo một biến kiểu struct Employee.  
    let john = Employee(empNumber:"E01",empName: "John", position:  
"CLERK")  
  
    print("Emp Number: " + john.empNumber)  
    print("Emp Name: " + john.empName)  
    print("Emp Position: " + john.position)  
  
}
```

# Phương thức khởi tạo của Struct

---

- Struct có thể có các Constructor (Phương thức khởi tạo), nhưng không có Destructor (Phương thức huỷ đối tượng)
- Bạn có thể không viết, viết một hoặc nhiều constructor cho Struct
- Trong khởi tạo, chúng ta phải gán tất cả các trường chưa có giá trị

// struct không có hàm khởi tạo

```
struct Employee {
```

```
    var empNumber:String
```

```
    var empName:String
```

```
    var position:String
```

```
}
```

# Phương thức và thuộc tính của Struct

- Struct có thể có các phương thức và thuộc tính:

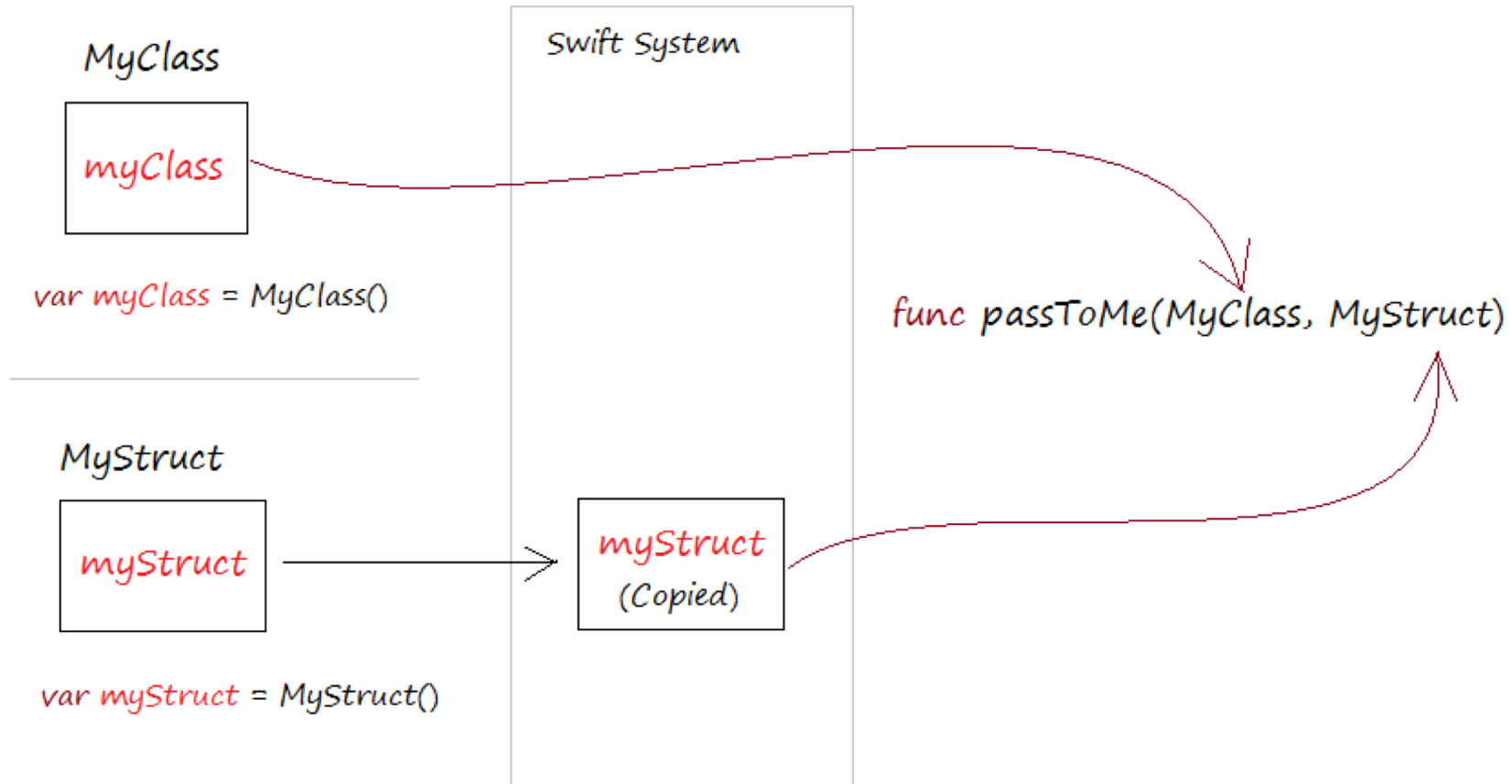
```
struct Book {  
  
    // thuộc tính  
    var title: String  
    var author: String  
  
    // Khởi tạo  
    init( title:String, author:String) {  
        self.title = title  
        self.author = author  
    }  
  
    // Phương thức  
    func getInfo() -> String {  
        return "Book Title: " + self.title + ", Author: " + self.author  
    }  
}
```

# Struct và Class



- Struct thường được sử dụng để tạo ra model (đối tượng) để lưu trữ giá trị, Class thì được sử dụng đa dạng hơn
- Struct không cho phép kế thừa từ một class hay một struct khác
- Nhưng struct cho phép kế thừa từ 1 hoặc nhiều Protocol
- Struct là kiểu tham số, Class là kiểu tham trị
- Nếu **struct** xuất hiện như một tham số trong một hàm (hoặc phương thức), nó được truyền (pass) dưới dạng giá trị. Trong khi đó nếu đối tượng của **class** xuất hiện như là một tham số trong một hàm (hoặc phương thức) nó được truyền (pass) như một tham chiếu (reference).

# Struct – Class: Tham chiếu và tham trị



# Ví dụ về tham chiếu và tham trị

```
class Person {  
    var name: String  
    var age: Int  
  
    init(name: String, age: Int) {  
        self.name = name  
        self.age = age  
    }  
}
```

```
let person1 = Person(name: "A", age: 13)  
var person2 = person1  
person2.age = 15
```

```
print(person1.age) // kq: 15  
print(person2.age) // kq: 15
```

```
struct Person {  
    var name: String  
    var age: Int  
  
    init(name: String, age: Int) {  
        self.name = name  
        self.age = age  
    }  
}
```

```
let person1 = Person(name: "A", age: 13)  
var person2 = person1  
person2.age = 15
```

```
print(person1.age) // kq: 13  
print(person2.age) // kq: 15
```

# Chọn sử dụng class hay struct



- Chúng ta đã biết class có instances được truyền bởi tham chiếu, còn struct được truyền bởi giá trị. Vậy nên tùy vào cấu trúc dữ liệu hay chức năng mà chúng ta quyết định sử dụng class hay struct:
- **Chúng ta xem xét việc tạo struct khi:**
  1. Cấu trúc dữ liệu đơn giản, có ít thuộc tính
  2. Những dữ liệu được đóng gói sẽ được copy hơn là tham chiếu khi bạn gán hay truyền instance của struct đó.
  3. Những thuộc tính được lưu trữ bởi struct thì bản thân nó là kiểu giá trị.
  4. Struct không cần thừa kế thuộc tính hay hành vi từ bất kì kiểu khác.





# Optional

Cài đặt và lập trình căn bản  
[cuong@techmaster.vn](mailto:cuong@techmaster.vn)

# Optional là gì?

- Optional là một khái niệm mới trong ngôn ngữ lập trình Swift
- Với việc sử dụng optional, ngôn ngữ Swift được xem là ngôn ngữ “an toàn” hơn so với ngôn ngữ Objective-C trước đó

*// khai báo optional*

**var** <tên biến>: <Kiểu dữ liệu>?


- Ví dụ:

*// Khai báo Integer Optional*

**var** perhapsInt: **Int**?

*// Khai báo String Optional*

**var** perhapsStr: **String**?

- 
- Trong Swift, khi khởi tạo các biến, các biến này mặc định sẽ được khởi tạo dưới dạng non-optional, tức là phải được gán giá trị mà không được để nil. Nếu chúng ta gán giá trị nil cho các biến non-optional này, trình biên dịch sẽ thông báo lỗi.

Ví dụ:

```
var str: String // khi sử dụng biến str thì compile error  
// biến str là kiểu String, kiểu non-optional nhưng không gán giá trị mặc định nên Xcode sẽ báo lỗi
```

```
var str: String = "Hello Swift" // OK  
str = nil // compile error  
// vì str là biến non-optional, nên không thể gán cho nó bằng nil
```

# Forced Unwrapping

Nếu có một biến được khai báo là optional và chúng ta muốn sử dụng giá trị của biến đó thì chúng ta phải unwrap biến đó

Ví dụ:

```
/* Nếu không unwrap*/  
  
// khai báo biến myString là một biến optional  
var myString:String?  
  
// khởi tạo giá trị cho biến myString  
myString = "Hello, Swift!"  
  
if myString != nil {  
    print(myString)  
} else {  
    print("myString has nil value")  
}
```

**Kết quả trả về**

```
Optional("Hello, Swift!")  
Program ended with exit code: 0
```

# Force Unwrapping

Force unwrap bằng cách thêm ! sau biến cần unwrap. Tuy nhiên việc force unwrap là việc nên tránh, vì trường hợp nếu biến force unwrap đó nil thì sẽ gây crash app

Ví dụ:

```
/* Nếu unwrap*/  
// khai báo biến myString là một biến optional  
var myString:String?  
  
// khởi tạo giá trị cho biến myString  
myString = "Hello, Swift!"  
  
if myString != nil {  
    print(myString!)  
} else {  
    print("myString has nil value")  
}
```

## Kết quả trả về

```
Hello, Swift!  
Program ended with exit code: 0
```

# Automatic Unwrapping

```
var myString:String!
```

```
myString = "Hello, Swift!"
```

```
if myString != nil {  
    print(myString)  
} else {  
    print("myString has nil value")  
}
```

- Ở đây biến myString được khai báo kiểu String!, khi ta gọi hàm print (myString) thì biến myString này đã tự động unwrap thành kiểu String

# Optional Binding

- Sử dụng optional binding để check xem biến optional có giá trị hay ko, để có những xử lý tương ứng tránh bị crash app
- Dùng **if let** hoặc **guard let** để binding optional

```
var myString:String?
```

```
myString = "Hello, Swift!"
```

```
if let yourString = myString {  
    print("\(yourString)")  
} else {  
    print("Your string does not have a value")  
}
```

## Kết quả:

```
Hello, Swift!
```

```
Program ended with exit code: 0
```

# if let và guard let

```
func optionalBinding() {  
    let name: String? = nil  
    let age: Int? = 3  
  
    if let ten = name{ // Khởi tạo một đối tượng ten = đối tượng optional name  
        print(ten) // sử dụng biến non-optional ten  
    }  
  
    // Khởi tạo một đối tượng tuoi = đối tượng optional age  
    guard let tuoi = age else {  
        print("age nil") // xử lý nếu age nil  
        return  
    }  
    print(tuoi) // sử dụng biến non-optional tuổi  
}  
  
optionalBinding()
```