



# Ngôn ngữ Swift căn bản

Cài đặt và lập trình căn bản

# Giới thiệu



- Swift là ngôn ngữ được Apple ra mắt vào năm 2014 và được cải tiến hằng năm
- Phiên bản hiện tại của Swift là 5.x
- Yêu cầu phiên bản trong khoá học này:
  - o macOS 11.x trở lên
  - o Xcode 12.x trở lên

# Cài đặt môi trường

App Store -> Xcode -> Downloads -> Installs



# Viết Swift ở đâu?

- Để biên dịch ngôn ngữ Swift, chúng ta có thể viết trên các nền tảng: Xcode, Terminal, Swift Online
- **Đối với terminal (hoặc iTerm2)**, chúng ta phải khởi động môi trường Swift

A screenshot of a macOS terminal window. The title bar shows the window name 'taof' and the command 'lldb --repl=-enable-objc-interop -sdk /Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDK...'. The terminal output shows the last login time, the user typing 'swift', and the Swift environment being initialized. A red arrow points to the 'swift' command, and a red text overlay reads 'Gõ lệnh Swift khởi động môi trường'. The terminal then shows a successful execution of a 'print' statement.

```
taof — lldb --repl=-enable-objc-interop -sdk /Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDK...
Last login: Wed Jul 10 10:40:17 on console
TaoQuynh:~ taof$ swift
Welcome to Apple Swift version 4.2.1 (swiftlang-1000.11.42 clang-1000.11.45.1).
Type :help for assistance.
1> print("Hello Swift")
Hello Swift
2> |
```

# Viết Swift ở đâu?

---

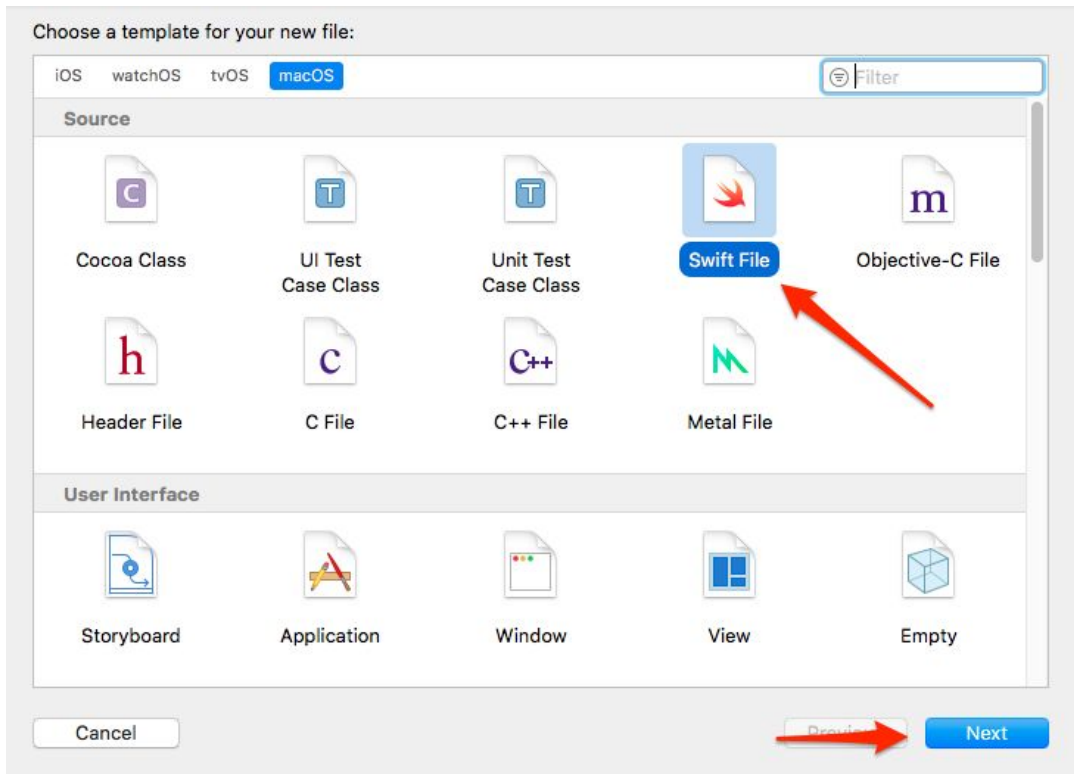
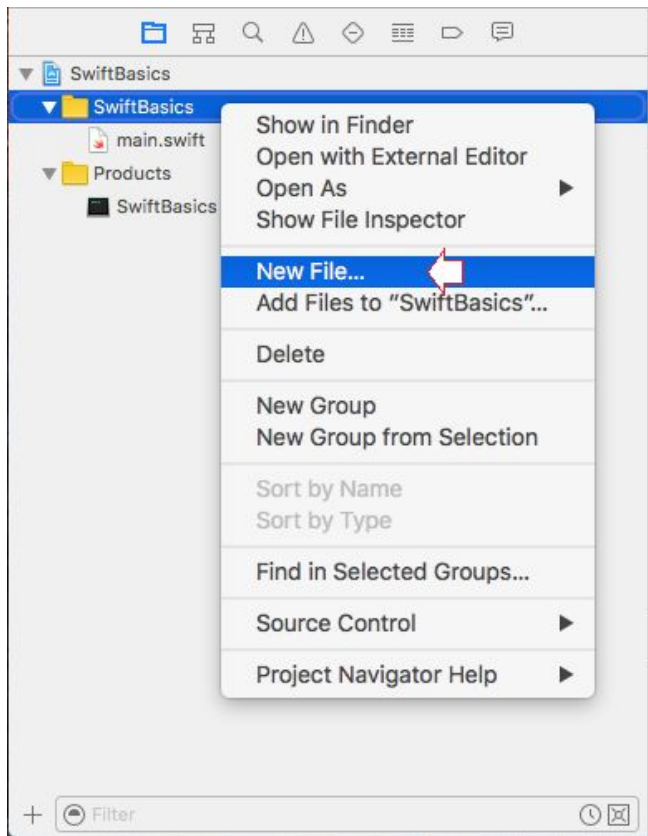
- Đối với Xcode, chúng ta có thể sử dụng **Playground** hoặc **Command Line Tools**:
  - + Playground: Xcode -> File -> New -> Playground -> Blank
  - + Command Line Tools: Xcode -> New Projects -> tab MacOS -> Command Line Tool

# Đặc điểm cơ bản của Swift

---

- **Top Level:** Một dòng lệnh hoặc một biểu thức không nằm trong một hàm, khối lệnh hoặc một class nghĩa là nó nằm ở Top-Level, là nơi khai báo sử dụng các thư viện, biến, hằng số, hàm, lớp.
- Điểm bắt đầu của chương trình Swift: trong một chương trình Swift file **main.swift** là một file đặc biệt, vì nó là điểm bắt đầu để chạy chương trình. Có thể gọi hàm hoặc viết biểu thức ở Top-Level trên file nguồn **main.swift**, đó là một ngoại lệ dành riêng cho file này.

# Thêm mới một file nguồn



# Các kiểu dữ liệu trong Swift

Trong Swift có các kiểu dữ liệu cơ bản:

- Kiểu chuỗi: **String**
- Kiểu logic: **Bool**, nhận giá trị **true** hoặc **false**
- Kiểu tập hợp: **Array**, **Set**, **Dictionary**, **Enum**
- Kiểu số:

Kiểu dữ liệu	Độ rộng	Phạm vi giá trị
Int8	1byte	-127 tới 127
UInt8	1byte	0 tới 255
Int32	4bytes	-2147483648 tới 2147483647
UInt32	4bytes	0 tới 4294967295
Int64	8bytes	-9223372036854775808 tới 9223372036854775807
UInt64	8bytes	0 tới 18446744073709551615
Float	4bytes	1.2E-38 tới 3.4E+38 (~6 digits)
Double	8bytes	2.3E-308 tới 1.7E+308 (~15 digits)



# Khai báo biến

- Chúng ta sử dụng từ khoá **var** để khai báo biến, cú pháp:

```
// Khai báo một biến.  
var <tên biến>: <kiểu dữ liệu>  
  
// Khai báo một biến đồng thời gán luôn giá trị.  
var <tên biến>: <kiểu dữ liệu> = <giá trị>  
  
// Khai báo một biến không xác định kiểu dữ liệu.  
var <tên biến> = <giá trị>
```

Ví dụ:

```
var strName: String // Khai báo biến kiểu chuỗi  
strName = "Swift"
```

```
var year: Int = 1985 // Khai báo và gán giá trị cho biến
```

```
var a, b, c: Double // Khai báo đồng thời nhiều biến  
var x = 0.0, y = 0.0, z = 0.0
```

# Khai báo hằng số

- Chúng ta sử dụng từ khoá **let** để khai báo hằng số, cú pháp:

```
// Khai báo một hằng số xác định kiểu dữ liệu.  
let <tên hằng số>: <kiểu dữ liệu>  
  
// Khai báo một hằng số đồng thời gán luôn giá trị.  
let <tên hằng số>: <kiểu dữ liệu> = <giá trị>  
  
// Khai báo một hằng số không xác định kiểu dữ liệu.  
let <tên hằng số> = <giá trị>
```

Ví dụ:

```
let pi = 3.14159
```

```
let languageName = "Swift"
```

```
let 🐵 = "Monkey" // Có thể sử dụng unicode để khai báo hằng số
```

# Xuất ra màn hình Console

- Sử dụng lệnh `print` in chuỗi hoặc biến ra màn hình

```
print("My name is Swift") // Kết quả: My name is Swift
```

```
var str = "Hello, Swift"  
print(str) // Kết quả: Hello, Swift
```

- Chúng ta có thể chèn giá trị biến/hằng vào một chuỗi trong câu lệnh `print`:

```
var name = "Swift"  
var age = 5  
print("I am \(name), I am \(age) years old.") // kết quả: I am Swift,  
I am 5 years old.
```

- Bản thân lệnh `print` đã có `\n` xuống dòng, để ngắt việc xuống dòng, hãy sử dụng terminator:

```
print("I am ", terminator: "")  
print("Swift")
```

# Toán tử số học

Toán tử	Ý nghĩa	Ví dụ
-(số âm)	Giá trị âm	<code>var x = -10</code>
*	Phép nhân	<code>var x: Int</code>
/	Phép chia	<code>var y: Int = 10</code> <code>var z: Int = 5</code>
+	Phép cộng	
-	Phép trừ	<code>x = y * 10 + z - 5 / 4</code>
%	Chia lấy dư	<code>var x: Int = 9 % 4 // Kết quả: x = 1</code>

# Toán tử so sánh

Toán tử	Ý nghĩa	Ví dụ
>	Lớn hơn	$5 > 4$ là đúng (true)
<	Nhỏ hơn	$4 < 5$ là đúng (true)
>=	Lớn hơn hoặc bằng	$4 >= 4$ là đúng (true)
<=	Nhỏ hơn hoặc bằng	$3 <= 4$ là đúng (true)
==	Bằng nhau	$1 == 1$ là đúng (true)
!=	Không bằng nhau	$1 != 2$ là đúng (true)
&&	Và	$a > 4 \ \&\& \ a < 10$
	Hoặc	$a == 1 \    \ a == 4$

# Toán tử gán

Toán tử	Ý nghĩa	Ví dụ
<code>+=</code>	Tính tổng	<code>a += b</code> tương đương <code>a = a + b</code>
<code>-=</code>	Tính hiệu	<code>a -= b</code> tương đương <code>a = a - b</code>
<code>*=</code>	Tính tích	<code>a *= b</code> tương đương <code>a = a * b</code>
<code>/=</code>	Tính thương	<code>a /= b</code> tương đương <code>a = a / b</code>
<code>%=</code>	Chia lấy dư	<code>a %= b</code> tương đương <code>a = a % b</code>

# Toán tử Ternary



<biểu thức điều kiện> ? <kết quả 1> : <kết quả 2>

```
var a: Int = 19
```

```
var b: Int = 2
```

```
let min a > b ? b : a // Kết quả: min = 2
```

# Chuỗi và kí tự

Chuỗi là một tập hợp các kí tự. Một chuỗi có thể tách thành một mảng các kí tự và ngược lại, một mảng các kí tự có thể ghép lại thành một chuỗi.

Kết quả:

```
let dogCharacter: [Character] = ["D", "o", "g", "🐶"]
let dogString: String(dogCharacter)

print(dogString)

for character in dogString {
    print(character)
}
```



```
Dog🐶
D
o
g
🐶
Program ended with exit code: 0
```



# Chuỗi và kí tự

---

```
let currentDay = "Tuesday"  
let prefix = "Today is "
```

```
let today = prefix + currentDay  
print(today)
```

```
// .isEmpty để kiểm tra xem chuỗi có rỗng hay không, isEmpty = true là rỗng  
print(currentDay.isEmpty)
```

```
// nối chuỗi = appending  
let anotherToday = prefix.appending(currentDay)  
print(anotherToday)
```

```
// viết hoa  
print(today.uppercased())
```

```
// viết thường  
print(today.lowercased())
```

# Chuỗi và kí tự

```
// Kiểm tra đầu chuỗi, cuối chuỗi
print(today.prefix(5)); print(today.suffix(6))

// Kiểm tra xem trong chuỗi có chứa chuỗi mình muốn tìm
print(today.contains("Monday"))

// đảo chuỗi
today.reversed()

// khai báo một mảng string
let myArrayString = ["This", "is", "Techmaster", "iOS", "class"]

// nối chuỗi từ một mảng
print(myArrayString.joined())

// nối chuỗi
print(myArrayString.joined(separator: " "))

// Cắt chuỗi thành một mảng
let joinedMyString = myArrayString.joined()
print(joinedMyString.components(separatedBy: "h"))
```



# Array

# Mảng



Nhiều phần tử cùng kiểu, tập hợp lại với nhau thành một thứ tự tạo thành mảng

- Khởi tạo mảng:

```
// Một mảng các số nguyên 1, 2, 3  
let myNumbers = [1, 2, 3]
```

```
// Mảng String rỗng  
var emptyStrings: [String] = [] // hoặc var emptyStrings = [String]()
```

```
// Khởi tạo mảng có 10 phần tử số nguyên, các phần tử có giá trị giống nhau  
var digits = [Int](repeating: 0, count: 10)
```

# Thao tác với mảng

```
var numberArray = [1, 43, 23, 0]

// Kiểm tra mảng rỗng
numberArray.isEmpty

// Kiểm tra mảng có bao nhiêu phần tử
numberArray.count

// Truy cập phần tử trong mảng bằng index
print(numberArray[2])

// Truy cập nhanh đến phần tử đầu / cuối mảng
numberArray.first; numberArray.last

// Duyệt mảng
for i in numberArray {
    print(i)
}

// Duyệt mảng lấy index
for (index, value) in numberArray.enumerated() {
    print("Chỉ số \(index) có giá trị \(value)")
}
```

# Thao tác với mảng



```
// Thêm một phần tử  
numberArray.append(19)
```

```
// Thêm một mảng phần tử  
numberArray += [12, 0, 5] // hoặc: numberArray.append(contentOf: [12, 0, 5])
```

```
// Chèn 1 phần tử vào vị trí index  
numberArray.insert(28, at: 2)
```

```
// Xoá khỏi mảng một phần tử theo vị trí index  
numberArray.remove(at: 3)
```

```
// Xoá phần tử đầu, cuối của mảng  
numberArray.removeFirst(); numberArray.removeLast()
```

```
// Xoá tất cả phần tử của mảng  
numberArray.removeAll()
```

# Dictionary

- Dictionary giống mảng đều là một **collection type**, là tập hợp gồm nhiều phần tử
- Mảng quản lý phần tử theo index, Dictionary quản lý phần tử theo định danh

```
var <Tên dictionary> = [<Kiểu dữ liệu Key>: < Kiểu dữ liệu Value>]()
```

Khởi tạo dictionary:

```
// tạo dictionary rỗng
var dictionaryOne = [String: String]()
var dictionaryTwo = [Int: String]()

// tạo dictionary 2 phần tử
var airports: [String: String] = ["NoiBai": "Hà Nội", "SaoVang": "Thanh Hoá"]
```

# Thao tác với Dictionary

---

```
// Thêm phần tử
airports["TanSonNhat"] = "TP. Hồ Chí Minh"

// Sửa giá trị phần tử
airports.updateValue("Thành phố Hồ Chí Minh", forKey: "TanSonNhat")

// Xoá phần tử
airports.removeValue(forKey: "TanSonNhat")

// Duyệt dictionary
for (airportCode, airportName) in airports {
    print("\(airportCode) - \(airportName)")
}

// Duyệt theo key hoặc value
for airportCode in airports.keys {
    print(airportCode)
}

for airportName in airports.values {
    print(airportName)
}
```





# If else – For loop

# Câu lệnh rẽ nhánh **if ... else**

- Cấu trúc **if**: kiểm tra một biểu thức nào đó có hợp lệ hay không

```
let x: Int = 10
if x > 9 {
    print("Giá trị x lớn hơn 9")
}
```

- Cấu trúc **if ... else**: mở rộng cấu trúc của if, quan tâm đến việc nếu biểu thức không hợp lệ thì làm gì

```
let a: Int = 20
if a % 2 == 0 {
    print("\(a) là số chẵn")
} else {
    print("\(a) là số lẻ")
}
```

# Câu lệnh rẽ nhánh **if ... else**

- Cấu trúc **if ... else if ... else**: kiểm tra nhiều trường hợp

```
let n: Int = -20
if n < 0 {
    print("n âm")
} else if n > 0 {
    print("n dương")
} else {
    print("n = 0")
}
```

- Cấu trúc **guard ... else**: chỉ thực hiện khối lệnh bên trong nếu biểu thức điều kiện sai

```
let y: Int? = nil
guard let x = y else {
    print("y nil")
    return
}
print(x)
```

# Câu lệnh rẽ nhánh switch ... case

---

- **switch**: xét và so sánh đối tượng xem đúng giá trị với case nào
- **case**: là một trường hợp để so sánh giá trị. Có thể là 1 hoặc nhiều giá trị
- **default**: nếu không đúng với case nào thì sẽ thực hiện khối lệnh trong default
- **fallthrough**: cho phép thực hiện case kế tiếp

# Ví dụ switch ... case

```
// khai báo biến option
let option = 15

switch option {
case 0...10:
    print("Case 0...10")
    // fallthrough: Thực thi trường hợp tiếp theo
    fallthrough
case 11...20:
    print("Case 11...20")
    // fallthrough: Thực thi trường hợp tiếp theo
    fallthrough
case 21...30:
    print("Case 21...30")
default:
    print("Default case")
}
```

# Vòng lặp for ... in

---

- Khi cần thực hiện một khối lệnh nhiều lần thì chúng ta sử dụng vòng lặp
- Vòng lặp **for ... in**: xử lý lặp trong các trường hợp như dãy số, tập hợp

```
// Khai báo một mảng các string với 5 phần tử
var languages:[String] = ["Java", "C", "Go", "Swift", "Ruby"]

for lang in languages {
    print("Language " + lang)
}
```

# Vòng lặp while

---

- Vòng lặp **while**: Khối lệnh bên trong vòng lặp sẽ được thực hiện lặp lại cho đến khi điều kiện lặp là sai

```
// Khai báo 1 biến và gán giá trị 2 cho nó
var x = 2

// Điều kiện là x < 10
// Nếu x < 10 là đúng (true) thì thực hiện khối lệnh
while (x < 10) {
    print("Value of X: \(x)")
    x = x+3
}
```

# Vòng lặp repeat ... while

- Vòng lặp **repeat ... while**: khối lệnh bên trong vòng lặp repeat ... while được thực hiện ít nhất 1 lần, điều kiện được kiểm tra ở cuối vòng lặp

```
// Khai báo 1 biến và gán giá trị 2 cho nó
```

```
var x = 2
```

```
// Thực thi khối lệnh ít nhất 1 lần
```

```
// Sau mỗi lần thực hiện xong khối lệnh nó sẽ kiểm tra điều kiện
```

```
// nếu điều kiện vẫn đúng, khối lệnh sẽ được thực thi tiếp
```

```
repeat {  
    print("Value of X: \(x)")
```

```
        x = x + 2  
} while (x < 10)
```



# Câu lệnh điều khiển trong vòng lặp

- **continue**: bỏ qua đoạn lệnh phía sau, thực hiện một vòng lặp mới
- **break**: kết thúc vòng lặp

```
var x = 2

while (x < 15) {
    print("X là: \x")

    // x = 5 thì thoát khỏi vòng lặp
    if (x==5) {
        break
    }

    // Tăng giá trị của x thêm 1
    x = x + 1
}
```

```
var x = 2

while (x < 7) {
    // Nếu x chẵn thì bỏ qua dòng lệnh phía dưới
    continue
    if (x%2 == 0) {
        x = x + 1
        continue
    } else {
        x = x + 1
    }
    print("After + 1, x = \x")
}
```



# Function

# Hàm (Function)



- Trong Swift, một hàm được định nghĩa bởi từ khoá **func**, hàm có tên cụ thể, hàm có thể có **không** hoặc **nhiều** tham số, và **có** hoặc **không có** kiểu trả về
- Hàm gồm 2 phần là **khai báo hàm** và **định nghĩa hàm**
- **Khai báo hàm** là thông báo với trình biên dịch về tên hàm, tham số truyền vào, kiểu trả về
- **Định nghĩa hàm** là phần thân hàm (xử lý của hàm)

# Ví dụ về định nghĩa hàm

```
// Định nghĩa một hàm
// Tên hàm: sayHello
// Tham số: name, kiểu String
// Trả về (return): String

func sayHello(name: String) -> String {

    // Nếu name rỗng
    if name.isEmpty {
        return "Hello every body!"
    }

    // Nếu name có giá trị
    return "Hello" + name
}
```

## Ví dụ 2 về định nghĩa hàm

```
// Định nghĩa một hàm, không có tham số, không có kiểu trả về
func testSayHello(){

    // Gọi hàm sayHello(), truyền vào một string rỗng
    let greeting1 = sayHello(name: "")
    print("greeting1: " + greeting1)

    // Gọi hàm sayHello(), truyền vào một string rỗng
    let greeting2 = sayHello(name: "Swift")
    print("greeting2: " + greeting2)
}

// Gọi hàm testSayHello()
testSayHello()
```

# Hàm trả về 1 giá trị

```
// Định nghĩa một hàm tính tổng 3 số Int, trả về kiểu Int.  
func sum(a: Int, b: Int, c: Int) -> Int {  
    return a + b + c  
}
```

```
// Định nghĩa một hàm để tìm số lớn nhất trong 3 số  
func max3So(a: Int, b: Int, c: Int) -> Int {  
  
    var m = a  
    if m < b {  
        m = b  
    }  
    if m > c {  
        return m  
    }  
  
    return c  
}
```

# Hàm trả về nhiều giá trị (Tuples)

```
func getMinMax(arrs: [Int]) -> (min: Int, max: Int) {  
    // Nếu mảng không có phần tử thì trả về (0, 0)  
    if arrs.count == 0 {  
        return (0, 0)  
    }  
  
    var min = arrs[0]  
    var max = arrs[0]  
  
    for a in arrs {  
        if min > a {  
            min = a  
        }  
  
        if max < a {  
            max = a  
        }  
    }  
  
    return (min, max)  
}
```

# Hàm với tham số Variadic

- Swift sử dụng `variableName: DataType...` để đánh dấu một tham số là **Variadic**

```
// Một hàm với các tham số variadic: nums
// Tham số nums: giống như một mảng các số Int
func sum(nums: Int...) -> Int {
    var tong = 0
    for i in nums {
        tong += i
    }
    return tong
}

// in hàm truyền vào 3 số
print(sum(nums: 1, 2, 4))

// in hàm truyền vào 7 số
print(sum(nums: 3, 23, 1, 0, 58, 5, 9))
```



# Hàm với tham số inout

- Tham số của hàm mặc định là hằng số, do đó nếu muốn thay đổi giá trị của các tham số và muốn nó tồn tại sau lời gọi hàm kết thúc thì chúng ta định nghĩa hàm với tham số **inout**

```
// Hàm hoán vị 2 số nguyên
func swap( a: inout Int, b: inout Int) {
    let t = a
    a = b
    b = t
}
```

```
var a = 10
var b = 17
```

```
// Gọi hàm
swap(&a, &b)
```

```
// Sau khi chạy hàm: a là 17, b là 10
print("a = \((a), b = \((b)")
```

# Hàm lồng nhau

- Swift cho phép viết một hàm bên trong một hàm khác, hàm này được sử dụng trong nội bộ của hàm cha

```
// Hàm trả về tiền thuế, dựa trên mã quốc gia và lương
func getTaxAmount(countryCode: String, salaryAmount: Int) -> Int {
    func getUSATaxAmount(salaryAmount: Int) -> Int {
        return 15 * salaryAmount / 100
    }

    func getVietNamTaxAmount(salaryAmount: Int) -> Int {
        return 10 * salaryAmount / 100
    }

    if countryCode == "$" {
        // USA
        return getUSATaxAmount(salaryAmount: salaryAmount)
    } else if countryCode == "VND" {
        // VietNam
        return getVietNamTaxAmount(salaryAmount: salaryAmount)
    }

    // Các quốc gia khác
    return 5 * salaryAmount / 100
}
```



# Optional

# 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 //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()
```



# Class và Struct

# 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

---

```
// Ví dụ 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:

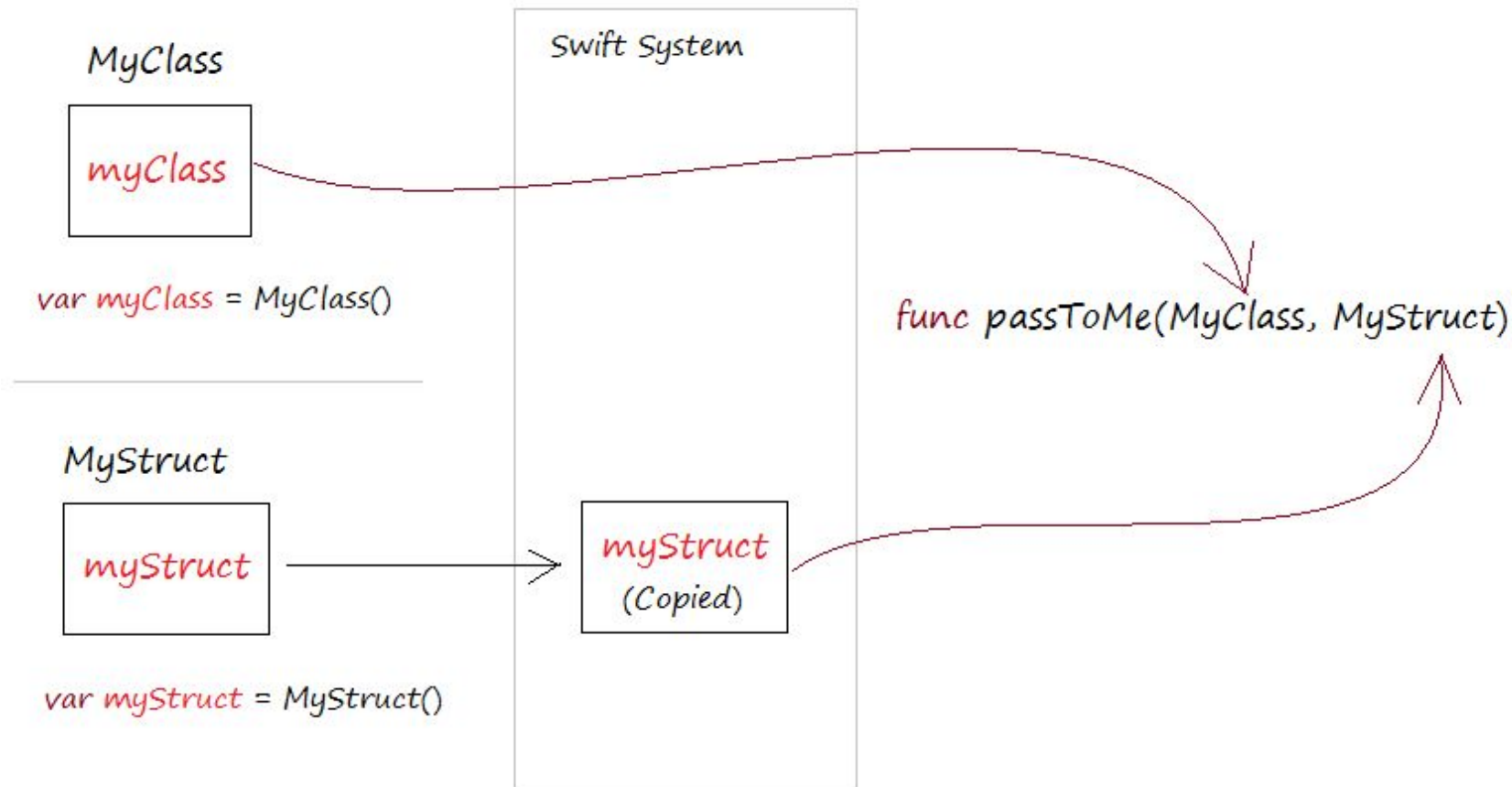
```
struct Book {  
  
    // thuộc tính  
    var title: String  
    var author: String  
  
    // Khởi tạo  
    init( title:String, author:String) {  
        self.title = title  
        self.author = author  
    }  
  
    // Phương thức  
    func getInfo() -> String {  
        return "Book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



# Làm việc với collection type

# 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
```

# Tài liệu tham khảo



Tài liệu tham khảo:

- Ebook [The Swift Programming Language](#)
- Swift [blog](#) của Apple
- [Hacking Swift](#)
- Q & A: [Stack Overflow](#)