

Lập trình iOS - Swift

Swift

- Swift là một ngôn ngữ lập trình hướng đối tượng dành cho việc phát triển iOS và macOS, watchOS, tvOS, Linux, và z/OS
- Được giới thiệu bởi Apple tại hội nghị WWDC 2014 (02/06/2014)
- Swift có thể sử dụng chung với mã C, C++ và Objective-C trong cùng một ứng dụng
- Swift còn khắc phục một số điểm yếu của Objective-C: nhẹ nhàng, nhanh, đơn giản hơn và hiệu năng xử lý tốt hơn rất nhiều
- Swift là ngôn ngữ mã nguồn mở
- Swift Playground: tính năng giúp chúng ta có thể xem nhanh kết quả thực theo thời gian thực mà không cần phải build hoặc run project



Các phiên bản

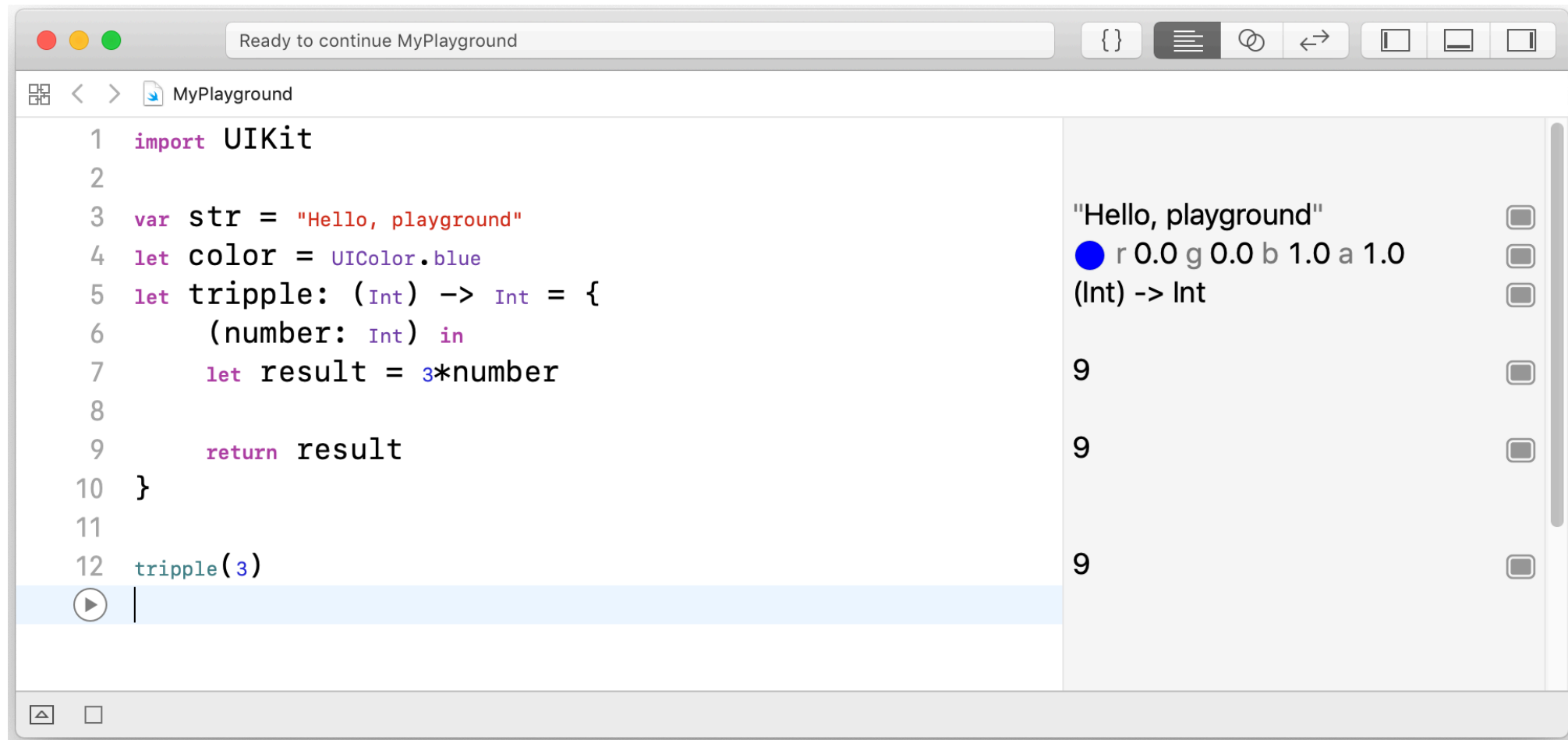
- Được giới thiệu lần đầu tại hội nghị WWDC 2014 (2/6/2014)
- Swift 1.0 9/9/2014
- Swift 1.1 22/10/2014
- Swift 1.2 8/4/2015
- Swift 2.0 21/9/2015
- Swift 3.0 13/9/2016
- Swift 4.0 19/9/2017
- Swift 4.1 29/3/2017
- Swift 4.2 17/9/2018
- Swift 5.0 25/3/2019

Swift vs Objective C

- Một số ưu điểm vượt trội của ngôn ngữ lập trình Swift so với Objective C
 - Apple và IBM đang hướng đến Swift
 - Ít code hơn, ít legacy hơn
 - Swift ít lỗi hơn
 - Swift nhanh hơn
 - Swift là mã nguồn mở
 - Swift có khả năng tương tác cao hơn
 - Swift gần hơn với các platforms khác

Playground

- Playground là một môi trường tương tác với code Swift, nó có thể hiển thị kết quả trong mỗi câu lệnh mà không cần compile hay run một project



Basic

- Constant and Variable

```
var age:Int = 42
```

```
var x = 0.0, y = 0.0, z = 0.0
```

```
var red, green, blue: Double
```

```
      variable name  
      ┌───┐  
var address:String = "1 Infinite Loop, Cupertino, CA 95014"  
└──┘ └──┘ └──────────────────────────────────────────┘  
keyword  variable type                                value
```

- Comment

```
// This is a comment."
```

```
/* This is also a comment  
   but is written over multiple lines. */
```

Basic

- Semicolon
 - Không giống các ngôn ngữ khác, Swift không yêu cầu “;” sau mỗi dòng code
 - Vẫn có thể thêm “;” sau mỗi dòng code nếu muốn
 - Tuy nhiên, nếu viết nhiều dòng code trên cùng một dòng thì cần phân cách nhau bằng “;”

```
var someNumber = 0;  
var otherNumber:Int = 5  
var age = 42;var name = "Swift"
```
- Boolean: Swift cung cấp 2 giá trị boolean là true và false

```
let orangesAreOrange: Bool = true  
let turnipsAreDelicious = false
```
- In giá trị
 - `print(friendlyWelcome)`
 - `// Prints "Bonjour!"`

Các kiểu dữ liệu trong Swift

- Kiểu dữ liệu cơ bản

- Int(Int32, Int64): `let meaningOfLife = 42`
- Float `let pi: Float = 3.14159`
- Double `let abc = 103.14159`
- Character `let key: Character = "8"`
- String `let value = "Swift iOS"`

- Kiểu dữ liệu tập hợp

- Array: Tập hợp các trị cùng dữ liệu và được sắp xếp có thứ tự
- Set: Tập hợp các trị khác kiểu dữ liệu và không có thứ tự
- Dictionary: Tập hợp các giá trị khác nhau, không có thứ tự, được truy xuất giá trị theo key nên key phải duy nhất
- Tuple: Là một nhóm các dữ liệu được gom lại trong cặp dấu (). Một Tuple có thể chứa nhiều kiểu dữ liệu khác nhau.

Basic

- Type Alias
 - Xác định một tùy chọn cho một kiểu đã có
 - Sử dụng từ khoá **typealias**

```
 typealias AudioSample = UInt16  
var maxAmplitudeFound = AudioSample.min  
// maxAmplitudeFound is now 0
```

Tuples

- Là một nhóm các dữ liệu được gom lại trong cặp dấu ().
- Một Tuples có thể chứa nhiều kiểu dữ liệu khác nhau

```
let http404Error = (404, "Not Found")  
let (statusCode, statusMessage) = http404Error
```

```
print("The status code is \(statusCode)")  
print("The status code is \(http404Error.0)")  
// Prints "The status code is 404"
```

```
print("The status message is \(statusMessage)")  
print("The status message is \(http404Error.1)")  
// Prints "The status message is Not Found"
```

Optional

- Sử dụng trong trường hợp biến có thể có hoặc không có giá trị
- Có thể thiết lập giá trị **nil** cho biến optional

```
var perhapsStr: String?  
perhapsStr = nil
```

```
var str: String //compile error  
var strValue: String = "Hello Swift" // OK  
strValue = nil // compile error  
perhapsStr = strValue // compile error
```

```
var otherStr: String? = "Hello Swift" // OK  
otherStr = nil // OK
```

Forced Unwrapping

- Nếu định nghĩa một biến kiểu optional, sau đó muốn lấy giá trị từ biến này, ta phải unwrap nó.
- Ở đây có 2 khái niệm là wrap và unwrap

```
var myString:String?
```

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

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

```
//Kết quả  
Optional("Hello, Swift!")
```

```
var myString:String?
```

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

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

```
//Kết quả  
"Hello, Swift!"
```

Optional Binding

- Cũng như forced unwrapping, Optional Binding cũng là một cách để unwrap
- Sử dụng câu lệnh **if** và **while** để kiểm tra giá trị bên trong optional và trích xuất giá trị đó thành hằng hoặc biến (sử dụng từ khoá **let** hoặc **var**)

```
var yearOfBirth: Int?
```

```
if let yearOfBirth = yearOfBirth {  
    var age = 2019 - yearOfBirth  
    print("Tuoi cua ban \ (age)")  
} else {  
    print("yearOfBirth is nil")  
}
```

Error Handling

- Xử lý lỗi để xử lý các điều kiện lỗi mà chương trình có thể gặp phải trong quá trình thực thi
- Để văng ra lỗi sử dụng từ khoá **throws**
- Sử dụng **try** và **do** catch để quản lý lỗi từ các function văng ra

```
func makeASandwich() throws { // ... }  
do {  
    try makeASandwich()  
    eatASandwich()  
} catch SandwichError.outOfCleanDishes {  
    washDishes()  
} catch SandwichError.missingIngredients(let ingredients) {  
    buyGroceries(ingredients)  
}
```

Basic Operator

- Thuật ngữ
 - Toán tử một ngôi – Unary Operators : hoạt động trên mục tiêu duy nhất (như là $-a$). Tiền tố toán tử một ngôi xuất hiện ngay trước mục tiêu của chúng (như là $!b$) và các hậu tố toán tử một ngôi xuất hiện ngay sau mục tiêu của chúng (như $i++$).
 - Toán tử hai ngôi – Binary Operators : hoạt động trên hai mục tiêu (như là $2 + 3$) và là trong tố (infix) vì chúng xuất hiện giữa hai mục tiêu
 - Toán tử ba ngôi – Ternary Operators : hoạt động trên ba mục tiêu . Như C, Swift chỉ có một toán tử ba ngôi , toán tử điều kiện ba ngôi ($a ? b : c$)
- Các giá trị mà các toán tử ảnh hưởng lên là các toán hạng (operands). Trong biểu thức $1 + 2$, ký hiệu $+$ là một toán tử hai ngôi và hai toán hạng của nó là các giá trị 1 và 2.

Basic Operator

- Assignment Operator – Toán tử gán

Toán tử gán ($a = b$) khởi tạo hoặc cập nhật giá trị của a với giá trị của b:

```
let b = 10
var a = 5
a = b
//a is now equal to 10
```

```
if x = y {
    //this is not a valid, because x =
    y does not return a type
}
```

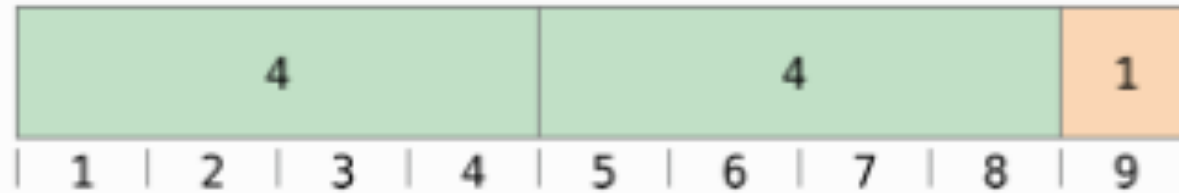
```
let (x,y) = (1,2)
//x is equal to 1, y is equal
to 2
```


Basic Operator

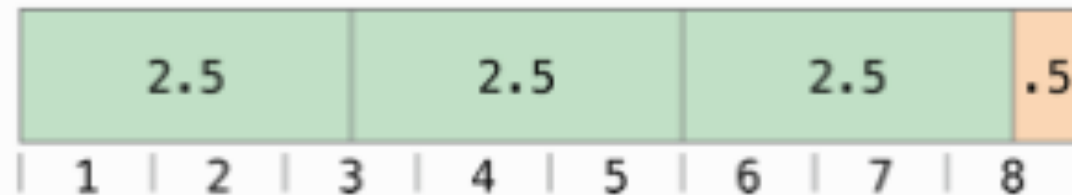
- Arithmetic Operators – Toán tử số học
 - Swift hỗ trợ 4 toán tử số học chuẩn cho tất cả các kiểu số : + , − , * , /
`1 + 2 //equals 3`
`5 − 3 //equals 2`
`2 * 3 //equals 6`
`10.0 / 2.5 //equals 4.0`
 - Toán tử Cộng cũng được hỗ trợ cho việc nối chuỗi
`"hello, " + "world" //equals "hello, world"`

Basic Operator

- Remainder Operator – Toán tử lấy số dư
 - Ví dụ: $9 \% 4$ // bằng 1



- Floating-Point Remainder Calculations – Tính toán số dư Dấu phẩy động (số thực)
 - Ví dụ: $8 \% 2,5$ // bằng 0,5



Basic Operator

- Compound Assignment Operators – Toán tử gán phức hợp

```
var a = 1
```

```
a += 2
```

//a is now equal to 3

- Comparison Operators – Toán tử so sánh
 - Equal to (a == b)
 - Not equal to (a != b)
 - Greater than (a > b)
 - Less than (a < b)
 - Greater than or equal to (a >= b)
 - Less than or equal to (a <= b)

Basic Operator

- Ternary Conditional Operator – Toán tử điều kiện ba ngôi

Mẫu: **question ? answer 1 : answer 2**

Tương đương

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

Basic Operator

- Range Operators – Toán tử phạm vi: Swift có hai toán tử phạm vi, đó là các lệnh tắt để diễn tả một phạm vi giá trị
 - Closed Range Operator – Toán tử phạm vi khép kín
 - Toán tử phạm vi khép kín (**a...b**) định nghĩa một phạm vi mà chạy từ a tới b, và bao gồm giá trị a và b. Giá trị của a cần phải không lớn hơn b.
 - Toán tử phạm vi khép kín hữu ích khi lặp qua một phạm vi trong đó bạn muốn tất cả các giá trị được sử dụng , nhưng là với một vòng lặp for-in :

```
for index in 1...5 {  
    print("\(index) times 5 is \(index * 5)")  
}  
  
//1 times 5 is 5  
//2 times 5 is 10  
//3 times 5 is 15  
//4 times 5 is 20  
//5 times 5 is 25
```

Basic Operator

- Half-Open Range Operator – Toán tử nửa Phạm vi
 - Toán tử nửa Phạm vi (**a..**b****) định nghĩa một phạm vi mà chạy từ a đến b , nhưng không bao gồm b.
 - Nó được nói là nửa phạm vi bởi vì nó chưa giá trị đầu tiên của nó, nhưng không có giá trị cuối cùng. Nếu giá trị của a bằng b, sau đó phạm vi kết quả sẽ là rỗng.

```
let names = ["Anna", "Alex", "Brian", "Jack"]
let count = names.count
for i in 0..
```

Basic Operator

- Logical Operators – Toán tử Logic
 - Toán tử logic chỉnh sửa hoặc kết hợp Giá trị logic Boolean true và false.
 - Swift hỗ trợ ba toán tử logic chuẩn được tìm thấy trong ngôn ngữ cơ bản C:
 - Logical NOT (!a)
 - Logical AND (a && b)
 - Logical OR (a || b)

```
let allowEntry = false
if !allowEntry {
    print("ACCESS DENIED")
}
//prints ACCESS DENIED
```

Basic Operator

```
let enteredDoorCode = true
let passedRetinaScan = false
```

```
if enteredDoorCode && passedRetinaScan
{
    print("Welcome!")
} else {
    print("ACCESS DENIED")
}
//prints ACCESS DENIED
```

```
let hasDoorKey = false
let knowsOverridePassword = true
```

```
if hasDoorKey || knowsOverridePassword
{
    print("Welcome!")
} else {
    print("ACCESS DENIED")
}
//prints Welcome!
```


Basic Operator

- Combining Logical Operators – Kết hợp các toán tử logic

```
if enteredDoorCode && passedRetinaScan || hasDoorKey ||  
   knowsOverridePassword {  
    print("Welcome!")  
} else {  
    print("ACCESS DENIED")  
}  
  
//prints Welcome!
```

String

- Là một chuỗi liên tiếp các ký tự (character) bao quanh bởi dấu nháy kép “

- String thuộc value type

```
let someString = "Some string literal value"
```

- Để định nghĩa multiline string: bao quanh bởi 3 dấu nháy kép liên tiếp

```
let quotation = """
```

```
The White Rabbit put on his spectacles. "Where shall I begin,  
please your Majesty?" he asked.
```

```
"Begin at the beginning," the King said gravely, "and go on  
till you come to the end; then stop.
```

```
"""
```

String

- Sử dụng backslash (\) ở cuối mỗi dòng để thay thế line break trong string

```
let softWrappedQuotation = ""
```

```
The White Rabbit put on his spectacles. "Where shall I  
begin, \n  
please your Majesty?" he asked.
```

```
"Begin at the beginning," the King said gravely, "and go on \n  
till you come to the end; then stop.  
""
```

String

- Đối với multiline string, khoảng trống trước closing quotation marks (""") sẽ cho biết khoảng trắng nào cần bỏ qua trước tất cả dòng khác

```
let linesWithIndentation = """
    This line doesn't begin with whitespace.
    This line begins with four spaces.
    This line doesn't begin with whitespace.
    """
```

Space ignored ———

Appears in string ———

Ký tự đặc biệt

- String có thể chứa các ký tự đặc biệt:
 - `\0` null character
 - `\\` backslash
 - `\t` horizontal tab
 - `\n` new line
 - `\r` di chuyển con trỏ về đầu dòng
 - `\"` double quotation mark
 - `'` single quotation mark
- String cũng có thể chứa các mã Unicode được viết với cú pháp sau: `\u{n}`
`let` wiseWords = `"\"Imagination is more important than knowledge\" – Einstein"`
`// "Imagination is more important than knowledge" – Einstein`
`let` dollarSign = `"\u{24}"` `// $, Unicode scalar U+0024`
`let` blackHeart = `"\u{2665}"` `// , Unicode scalar U+2665`
`let` sparklingHeart = `"\u{1F496}"` `// 💖, Unicode scalar U+1F496"`

String

- Khởi tạo empty string

```
var emptyString = ""           // empty string literal
var anotherEmptyString = String() // initializer syntax
// these two strings are both empty, and are equivalent to each other"
```

- Biến đổi string

```
var variableString = "Horse"
variableString += " and carriage"
// variableString is now "Horse and carriage"
```

```
let constantString = "Highlander"
constantString += " and another Highlander"
// this reports a compile-time error – a constant string cannot be modified"
```

String

- Sử dụng thuộc tính `isEmpty` để kiểm tra string có rỗng hay không

```
if emptyString.isEmpty {  
    print("Nothing to see here")  
}  
// Prints "Nothing to see here"
```

- Sử dụng thuộc tính `count` để biết độ dài của string

```
let unusualMenagerie = "Koala 🐨, Snail 🐌, Penguin 🐧, Dromedary 🐪"  
print("unusualMenagerie has \(unusualMenagerie.count) characters")  
// Prints "unusualMenagerie has 40 characters"
```

Character

- Khởi tạo character

```
let exclamationMark: Character = "!"
```

- Duyệt character trong string

```
for character in "Dog!🐶" {  
    print(character)  
}
```

- Tạo string từ mảng character

```
let catCharacters: [Character] = ["C", "a", "t", "!", "🐱"]  
let catString = String(catCharacters)  
print(catString)  
// Prints "Cat!🐱"
```


Character

- Nối 2 string

```
let string1 = "hello"  
let string2 = " there"  
var welcome = string1 + string2  
// welcome now equals "hello there"
```

```
var instruction = "look over"  
instruction += string2  
// instruction now equals "look over  
there"
```

- Append character vào string: sử dụng phương thức `append()`

```
let exclamationMark: Character = "!"  
welcome.append(exclamationMark)  
// welcome now equals "hello there!"
```

String Interpolation

- Là cách xây dựng một string mới từ hỗn hợp các constants, variables, string và biểu thức bằng cách bao gồm các giá trị của chúng bên trong một chuỗi ký tự
- Mỗi item chèn vào chuỗi ký tự được gói trong một cặp dấu ngoặc đơn, có tiền tố là dấu gạch chéo ngược (\)

```
let multiplier = 3
```

```
let message = "\(multiplier) times 2.5 is \((Double(multiplier) *  
2.5)"
```

```
// message is "3 times 2.5 is 7.5"
```

Accessing and Modifying a String

- Một số thuộc tính và phương thức hay sử dụng
 - `index`
 - Ý nghĩa: vị trí của mỗi character trong string
 - `startIndex`
 - Ý nghĩa: vị trí character đầu tiên của string
 - `endIndex`
 - Ý nghĩa: vị trí character cuối cùng của string
 - `index(before:)`
 - Ý nghĩa: vị trí character liền trước index cho trước
 - `index(after:)`
 - Ý nghĩa: vị trí character liền sau index cho trước
 - `index(_ :offsetBy: 7)`
 - Ý nghĩa: vị trí character cách vị trí index cho trước 1 khoảng offset

Accessing and Modifying a String

- Ví dụ:

```
let greeting = "Guten Tag!"
greeting[greeting.startIndex]
// G
greeting[greeting.index(before: greeting endIndex)]
// !
greeting[greeting.index(after: greeting.startIndex)]
// u
let index = greeting.index(greeting.startIndex, offsetBy: 7)
greeting[index]
// a"
```

Accessing and Modifying a String

- Truy cập một index bên ngoài phạm vi của string hoặc character tại một index bên ngoài phạm vi của chuỗi sẽ gây ra lỗi

```
greeting[greeting endIndex] // Error  
greeting.index(after: greeting endIndex) // Error"
```

- Sử dụng thuộc tính `indices` để truy cập toàn bộ index của character trong string

```
for index in greeting.indices {  
    print("\(greeting[index]) ", terminator: " ")  
}  
// Prints "G u t e n    T a g ! "
```

Accessing and Modifying a String

- Inserting and Removing

- `insert(_:at:)`: chèn một ký tự đơn vào chuỗi tại index
- `insert(contentsOf:at:)`: chèn nội dung chuỗi khác vào chuỗi tại index

```
var welcome = "hello"  
welcome.insert("!", at: welcome endIndex)  
// welcome now equals "hello!"
```

```
welcome.insert(contentsOf: " there", at: welcome.index(before:  
welcome endIndex))  
// welcome now equals "hello there!"
```

Accessing and Modifying a String

- `remove(at:)`: loại bỏ ký tự khỏi chuỗi tại index xác định
- `removeSubrange(_:)`: loại bỏ chuỗi con khỏi chuỗi tại phạm vi xác định

```
welcome.remove(at: welcome.index(before: welcome endIndex))  
// welcome now equals "hello there"
```

```
let range = welcome.index(welcome endIndex, offsetBy: -  
6)..  
welcome.removeSubrange(range)  
// welcome now equals "hello"
```

Accessing and Modifying a String

- Substrings

```
let greeting = "Hello, world!"
```

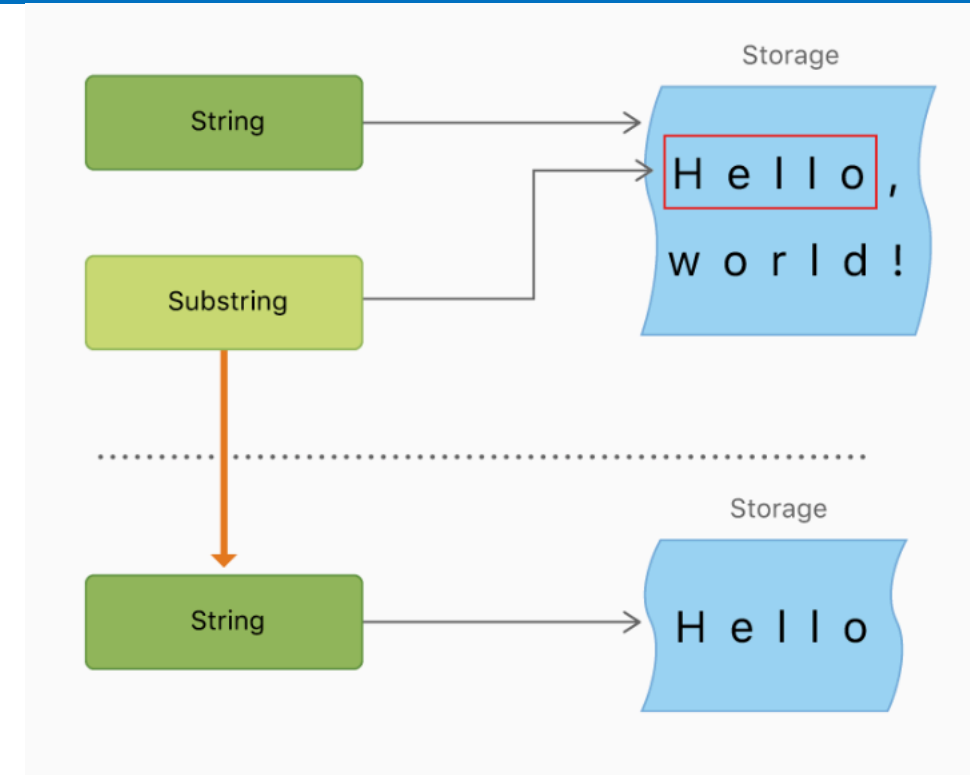
```
let index = greeting.index(of: ",") ??  
greeting.endIndex
```

```
let beginning = greeting[..<index]
```

```
// beginning is "Hello"
```

```
// Convert the result to a String for long-  
term storage.
```

```
let newString = String(beginning)
```



Accessing and Modifying a String

- Compare String: String cung cấp 3 phương thức để so sánh giá trị:
 - String and character equality
 - Prefix equality
 - Suffix equality
- String and character equality: sử dụng toán tử `==` hoặc `!=`

```
let quotation = "We're a lot alike, you and I."
let sameQuotation = "We're a lot alike, you and I."
if quotation == sameQuotation {
    print("These two strings are considered equal")
}
// Prints "These two strings are considered equal"
```

Accessing and Modifying a String

- Prefix and suffix equality: sử dụng hai phương thức `hasPrefix(_:)`, `hasSuffix(_:)`

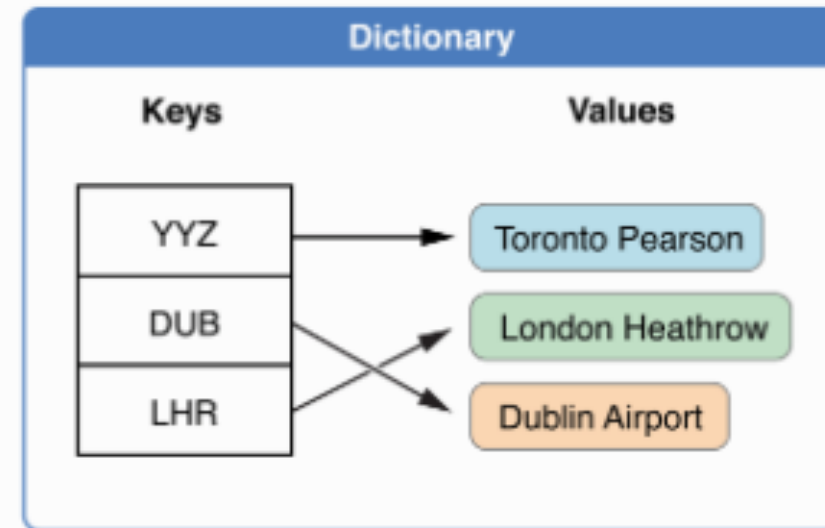
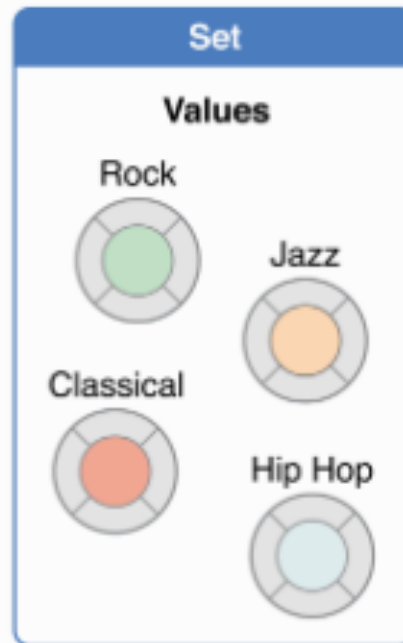
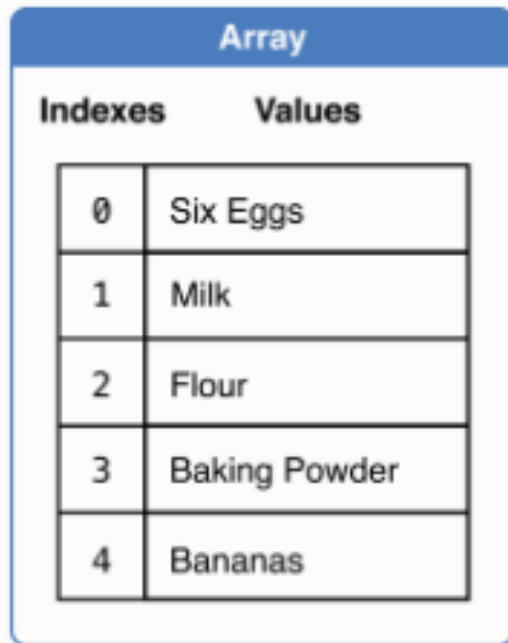
```
let str = "Hello, playground"

if str.hasPrefix("Hello") { // true
    print("Prefix exists")
}

if str.hasSuffix("ground") { // true
    print("Suffix exists")
}
```

Collection Types

- Swift cung cấp 3 collection types cơ bản: Array, Set, Dictionary



Array

- Tập hợp các trị cùng dữ liệu và được sắp xếp có thứ tự
- Các giá trị giống nhau có thể xuất hiện nhiều lần trong array ở các vị trí khác nhau
- Cú pháp array:
 - Full: `Array<Element>`
 - Shorthand: `[Element]`

```
var someInts: [Int]
```

Array

- Khởi tạo Array

```
var someInts = [Int]()  
print("someInts is of type [Int] with \(${someInts.count}) items.")  
// Prints "someInts is of type [Int] with 0 items."
```

```
var threeDoubles = Array(repeating: 0.0, count: 3)  
// threeDoubles is of type [Double], and equals [0.0, 0.0, 0.0]"
```

```
var shoppingList: [String] = ["Eggs", "Milk"]  
// shoppingList has been initialized with two initial items"
```

Array

- Adding 2 array

```
var anotherThreeDoubles = Array(repeating: 2.5, count: 3)  
// anotherThreeDoubles is of type [Double], and equals [2.5, 2.5, 2.5]
```

```
var sixDoubles = threeDoubles + anotherThreeDoubles  
// sixDoubles is inferred as [Double], and equals [0.0, 0.0, 0.0, 2.5, 2.5, 2.5]"
```

Truy cập và sửa đổi mảng

- Sử dụng thuộc tính `count` để biết được số lượng phần tử trong một mảng

```
print("The shopping list contains \(shoppingList.count) items.")"
```
- Sử dụng thuộc tính `isEmpty` để kiểm tra mảng có rỗng hay không

```
if shoppingList.isEmpty {  
    print("The shopping list is empty.")  
} else {  
    print("The shopping list is not empty.")  
}
```
- Sử dụng phương thức `append(_:)` để thêm một phần tử vào cuối mảng

```
shoppingList.append("Flour")
```
- Sử dụng toán tử `+=` để nối mảng có một hoặc nhiều phần tử

```
shoppingList += ["Chocolate Spread", "Cheese", "Butter"]
```

Truy cập và sửa đổi mảng

- Để truy cập một phần tử trong mảng, ta sử dụng index. Phần tử đầu tiên trong mảng có index = 0

```
let firstItem = shoppingList[0]  
shoppingList[0] = "Six eggs"  
let firstItem = shoppingList.first
```

```
let lastItem = shoppingList[shoppingList.count - 1]  
let lastItem = shoppingList.last
```

- Sử dụng subscript syntax để thay đổi 1 khoảng giá trị trong mảng

```
var shoppingList = ["Eggs", "Milk", "Flour", "Baking Powder",  
"Chocolate Spread", "Cheese", "Butter"]  
shoppingList[4...6] = ["Bananas", "Apples"]
```


Truy cập và sửa đổi mảng

- Thêm hoặc xoá phần tử trong mảng tại một vị trí

```
shoppingList.insert("Maple Syrup", at: 0)  
let mapleSyrup = shoppingList.remove(at: 0)
```

```
let firstItem = shoppingList.removeFirst()  
let lastItem = shoppingList.removeLast()
```

- Duyệt mảng

```
for item in shoppingList {  
    print(item)  
}
```

```
for (index, value) in shoppingList.enumerated() {  
    print("Item \(index + 1): \(value)")  
}
```

Set

- Tập hợp các trị khác kiểu dữ liệu và không có thứ tự
- Sử dụng set khi thứ tự của các phần tử không quan trọng và khi cần đảm bảo một giá trị chỉ xuất hiện một lần duy nhất
- Cú pháp: `Set<Element>`

```
var letters = Set<Character>()  
print("letters is of type Set<Character> with \((letters.count)  
items.")  
// Prints "letters is of type Set<Character> with 0 items."
```

Set

- Khởi tạo một Set

```
var favoriteGenres: Set<String> = ["Rock", "Classical", "Hip hop"]  
var favoriteGenres: Set = ["Rock", "Classical", "Hip hop"]
```

- Truy cập và sửa đổi Set

- Sử dụng thuộc tính `count` để biết được số lượng phần tử trong set

```
print("I have \(favoriteGenres.count) favorite music genres.")
```

- Sử dụng thuộc tính `isEmpty` để kiểm tra set có rỗng hay không

```
if favoriteGenres.isEmpty {  
    print("As far as music goes, I'm not picky.")  
} else {  
    print("I have particular music preferences.")  
}
```

Truy cập và sửa đổi Set

- Sử dụng phương thức `insert(_:)` để thêm một phần tử vào trong set
`favoriteGenres.insert("Jazz")`
- Sử dụng phương thức `remove(_:)` hoặc `removeAll()` để loại bỏ một hoặc toàn bộ phần tử trong set

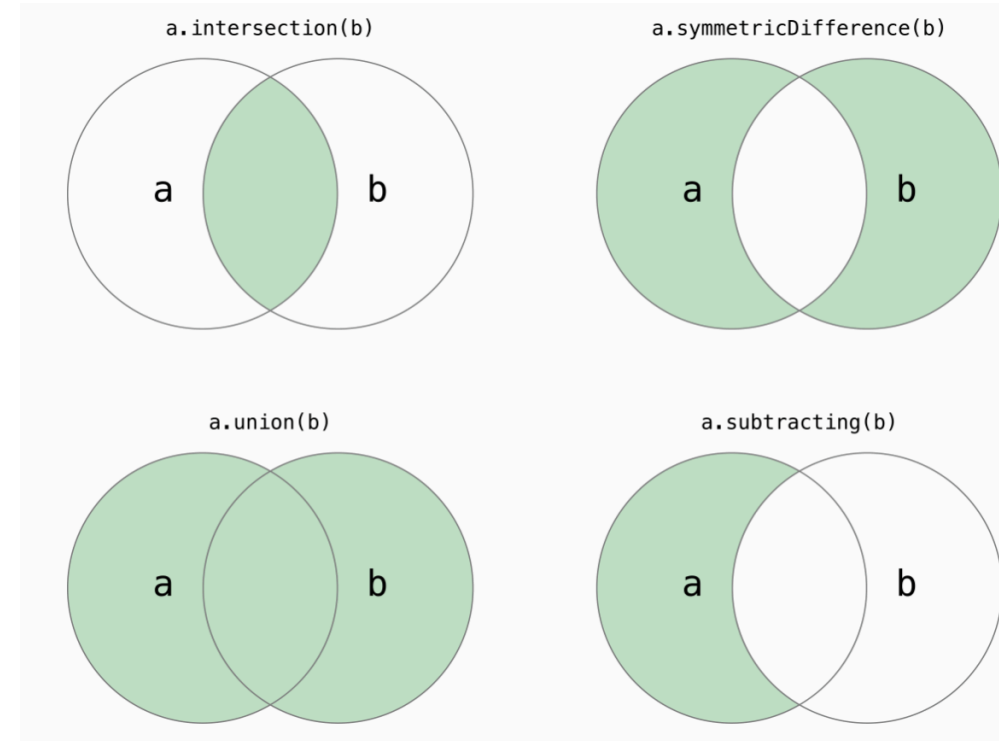
```
if let removedGenre = favoriteGenres.remove("Rock") {  
    print("\(removedGenre)? I'm over it.")  
} else {  
    print("I never much cared for that.")  
}
```

- Duyệt phần tử trong set

```
for genre in favoriteGenres.sorted() {  
    print("\(genre)")  
}
```

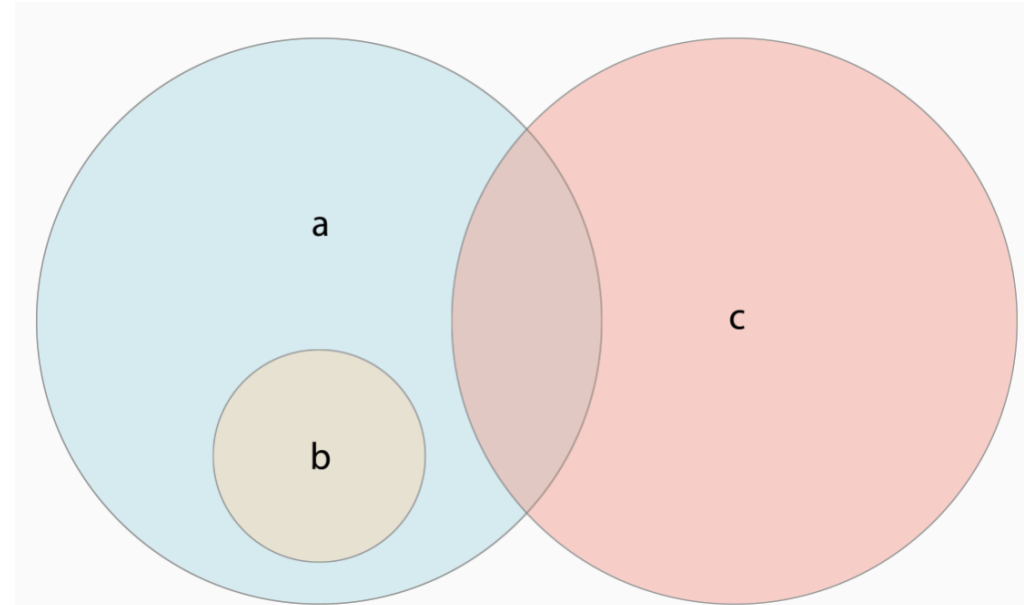
Một số phương thức khác của Set

```
let oddDigits: Set = [1, 3, 5, 7, 9]
let evenDigits: Set = [0, 2, 4, 6, 8]
let singleDigitPrimeNumbers: Set = [2, 3, 5, 7]
oddDigits.union(evenDigits).sorted()
// [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
oddDigits.intersection(evenDigits).sorted()
// []
oddDigits.subtracting(singleDigitPrimeNumbers).sorted()
// [1, 9]
oddDigits.symmetricDifference(singleDigitPrimeNumbers).sorted()
// [1, 2, 9]
```



Một số phương thức khác của Set

- Một số phương thức
 - `==`
 - Ý nghĩa: so sánh bằng 2 set
 - `isSubset(of:)`
 - Ý nghĩa: mọi phần tử set A chứa trong set B
 - `isSuperset(of:)`
 - Ý nghĩa: set A có chứa mọi phần tử set B
 - `isStrictSubset(of:)`
 - Ý nghĩa: mọi phần tử set A chứa trong set B nhưng 2 set không bằng nhau
 - `isStrictSuperset(of:)`
 - Ý nghĩa: set A có chứa mọi phần tử set B nhưng 2 set không bằng nhau
 - `isDisjoint(with:)`
 - Ý nghĩa: 2 set A, B không có phần tử chung



Một số phương thức khác của Set

```
let houseAnimals: Set = ["🐶", "🐱"]  
let farmAnimals: Set = ["🐮", "🐔", "🐑", "🐶", "🐱"]  
let cityAnimals: Set = ["🐼", "🐭"]
```

```
houseAnimals.isSubset(of: farmAnimals)  
// true  
farmAnimals.isSuperset(of: houseAnimals)  
// true  
farmAnimals.isDisjoint(with: cityAnimals)  
// true"
```

Dictionary

- Tập hợp các giá trị khác nhau, không có thứ tự, được truy xuất giá trị theo key nên key phải duy nhất
- Các key phải có cùng kiểu, các value cũng phải có cùng kiểu
- Cú pháp: `Dictionary<Key, Value>` hoặc `[Key: Value]`
`var namesOfIntegers: [Int: String]`

Dictionary

- Khai báo

```
var namesOfIntegers = [Int: String]()  
// namesOfIntegers is an empty [Int: String] dictionary"  
namesOfIntegers[16] = "sixteen"  
// namesOfIntegers now contains 1 key-value pair  
  
namesOfIntegers = [:]  
// namesOfIntegers is once again an empty dictionary of type [Int:  
String]"  
  
var airports: [String: String] = ["YYZ": "Toronto Pearson", "DUB":  
"Dublin"]
```

Truy cập và sửa đổi Dictionary

- Sử dụng thuộc tính `count` để biết được số lượng phần tử trong dictionary
`print("The airports dictionary contains \(airports.count) items.")`
- Sử dụng thuộc tính `isEmpty` để kiểm tra dictionary có rỗng hay không
`if airports.isEmpty {
 print("The airports dictionary is empty.")
} else {
 print("The airports dictionary is not empty.")
}`
- Thêm hoặc thay đổi giá trị cho 1 key trong dictionary
`airports["LHR"] = "London"
airports["LHR"] = "London Heathrow"
// the value for "LHR" has been changed to "London Heathrow"`

Dictionary

- Lấy giá trị của một key trong dictionary

```
if let airportName = airports["DUB"] {  
    print("The name of the airport is \(airportName).")  
} else {  
    print("That airport is not in the airports dictionary.")  
}
```

- Remove key-value trong dictionary

```
airports["APL"] = nil  
// APL has now been removed from the dictionary"
```

Duyệt phần tử trong Dictionary

```
for (airportCode, airportName) in airports {  
    print("\(airportCode): \(airportName)")  
}
```

```
for airportCode in airports.keys {  
    print("Airport code: \(airportCode)")  
}
```

```
for airportName in airports.values {  
    print("Airport name: \(airportName)")  
}
```

```
let airportCodes = [String](airports.keys)  
let airportNames = [String](airports.values)
```

Control Flow

- For-in

```
let names = ["Anna", "Alex",  
"Brian", "Jack"]  
for name in names {  
    print("Hello, \(name)!")  
}  
// Hello, Anna!  
// Hello, Alex!  
// Hello, Brian!  
// Hello, Jack!"
```

```
let numberOfLegs = ["spider": 8, "ant": 6,  
"cat": 4]  
for (animalName, legCount) in numberOfLegs {  
    print("\(animalName)s have \(legCount)  
legs")  
}  
// ants have 6 legs  
// spiders have 8 legs  
// cats have 4 legs"
```

Control Flow

- For-in

```
for index in 1...5 {  
    print("\(index) times 5 is  
    \(index * 5)")  
}  
  
// 1 times 5 is 5  
// 2 times 5 is 10  
// 3 times 5 is 15  
// 4 times 5 is 20  
// 5 times 5 is 25"
```

```
let minutes = 60  
for tickMark in 0..    // render the tick mark each  
    minute (60 times)  
}
```

```
let base = 3  
let power = 10  
var answer = 1  
for _ in 1...power {  
    answer *= base  
}  
print("\(base) to the power of \(power) is  
    \(answer)")
```

```
let minuteInterval = 5  
for tickMark in stride(from: 0, to:  
    minutes, by: minuteInterval) {  
    // render the tick mark every 5  
    minutes (0, 5, 10, 15 ... 45, 50, 55)  
}
```

Control Flow

- While

```
var index = 10
while index < 20 {
    print( "Value of index is \ (index)")
    index = index + 1
}
```

- repeat-while

```
var sum = 2
repeat {
    print(sum)
    sum = sum + 2
} while sum < 10
```

Control Flow

- If

```
let temperatureInFahrenheit = 90
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
} else if temperatureInFahrenheit >= 86 {
    print("It's really warm. Don't forget to wear sunscreen.")
} else {
    print("It's not that cold. Wear a t-shirt.")
}
// Prints "It's really warm. Don't forget to wear sunscreen."
```


Control Flow

- switch

```
let anotherCharacter: Character =  
"a"  
switch anotherCharacter {  
case "a", "A":  
    print("The letter A")  
default:  
    print("Not the letter A")  
}  
// Prints "The letter A"
```

```
let value = 10  
  
switch value {  
case 4...20:  
    print("10 is in range 4-20")  
default:  
    print("10 is not in range 4-20")  
}
```

Control Flow

- Switch

```
let somePoint = (1, 1)
switch somePoint {
case (0, 0):
    print("\(somePoint) is at the origin")
case (_, 0):
    print("\(somePoint) is on the x-axis")
case (0, _):
    print("\(somePoint) is on the y-axis")
case (-2...2, -2...2):
    print("\(somePoint) is inside the
box")
default:
    print("\(somePoint) is outside of the
box")
}
// Prints "(1, 1) is inside the box"
```

```
let anotherPoint = (2, 0)
switch anotherPoint {
case (let x, 0):
    print("on the x-axis with an x
value of \(x)")
case (0, let y):
    print("on the y-axis with a y
value of \(y)")
case let (x, y):
    print("somewhere else at (\(x),
\(\y))")
}
// Prints "on the x-axis with an x
value of 2"
```

Control Flow

- Switch

```
let yetAnotherPoint = (1, -1)
switch yetAnotherPoint {
case let (x, y) where x == y:
    print("\(x), \(y)) is on the line x == y")
case let (x, y) where x == -y:
    print("\(x), \(y)) is on the line x == -y")
case let (x, y):
    print("\(x), \(y)) is just some arbitrary point")
}
// Prints "(1, -1) is on the line x == -y"
```

Control Flow

- Control Transfer Statement: Swift cung cấp 5 câu lệnh chuyển điều khiển
 - **continue:** trong vòng lặp nếu gặp continue thì nó sẽ bỏ qua trường hợp đó (ở đây trong câu lệnh if) rồi tiếp tục duyệt tiếp
 - **break:** kết thúc chương trình trong loop, if hoặc switch
 - **fallthrough:** xét trên xuống, nếu khớp 1 case nào đó vẫn tiếp tục xét tiếp 1 case liền kề
 - **return:** thoát khỏi hàm, và có thể trả về giá trị nào đó tùy chúng ta khai báo
 - **throw:** ném ra 1 error từ 1 hàm.

```
for idx in 0...3 {  
    if idx % 2 == 0 {  
        continue  
    }  
    print("This code never fires on even numbers")  
}
```

```
for idx in 0...3 {  
    if idx % 2 == 0 {  
        break  
    }  
}
```

Control Flow

- Control Transfer Statement

```
let box = 0
switch box
{
case 0:
    print(0) // In ra 0
    fallthrough
case 1:
    print(1) // In ra 1
case 2:
    print(2) // Không có được in ra
default:
    print("default")
}
```

```
func doNothing() {
    return //Immediately leaves the
context
    let anInt = 0
    print("This never prints \((anInt)")
}

//=====
enum WeekendError: Error {
    case Overtime
    case WorkAllWeekend
}

func workOvertime () throws {
    throw WeekendError.Overtime
}
```

Control Flow

- Labeled Statements: cho phép dán nhãn các câu lệnh điều khiển

```
fancyLabel: for each in array {  
    for eachSubItem in subArray {  
        switch statement {  
            case one:  
                continue fancyLabel  
            ...  
        }  
    }  
    ...  
}
```

Control Flow

- Early exit
 - Sử dụng guard statement để kiểm tra điều kiện. Nếu điều kiện đúng thì đoạn mã sau câu lệnh guard sẽ được thực thi
 - Không giống như mệnh đề if, một guard statement luôn có else clause

```
private func printRecordFromLastName(userLastName: String?) {  
    guard let name = userLastName, userLastName != "Null" else {  
        //Sorry Bill Null, find a new job  
        return  
    }  
    //Party on  
    print(dataStore.findByName(name))  
}
```

Functions

- Hàm bao gồm tham số (parameter) và kiểu trả về (return types)
- Để khai báo hàm sử dụng từ khoá theo sau là tên hàm và tham số. Cuối cùng là kiểu trả về
- Một hàm có hoặc không có tham số và kiểu trả về

```
func sayHello(personName: String)
-> String {
    let greeting = "Hello, " +
    personName + "!"
    return greeting
}
print(sayHello("Henry"))
```

```
func greet(person: String) {
    print("Hello, \(person)!")
}
greet(person: "Dave")
// Prints "Hello, Dave!"
```


Functions

- Trong Swift, một hàm có thể trả về một lúc nhiều giá trị

```
func minMax(array: [Int]) -> (min: Int, max: Int) {  
    var currentMinValue: Int = array[0];  
    var currentMaxValue: Int = array[0];  
  
    for number in numberArray {  
        if currentMinValue > number {  
            currentMinValue = number  
        } else if currentMaxValue < number {  
            currentMaxValue = number  
        }  
    }  
    return (currentMinValue, currentMaxValue)  
}
```

Functions

- Parameter Names
 - Một tham số có cả tên toàn cục (external parameter name) và tên cục bộ (local parameter name).
 - Tên toàn cục được sử dụng lúc gọi hàm trong khi đó tên cục bộ được sử dụng trong hàm
 - Tên toàn cục bắt buộc phải có khi gọi hàm

```
func sayHello(to person: String, and anotherPerson: String) -> String
{
    return "Hello \ (person) and \ (anotherPerson)"
}
```

```
sayHello(to: "Bill", and: "Ted")
```

Functions

- Mặc định, nếu không định nghĩa tên toàn cục cho tham số. thì tham số đầu tiên sẽ không có tên toàn cục, tham số thứ 2 sẽ dùng tên cục bộ của nó làm tên toàn cục

```
func someFunction(firstParameter:  
Int, secondParameter: Int) {  
  
}  
someFunction(1, secondParameter: 2)
```

```
func someFunction(firstParameter:  
Int, _ secondParameter: Int) {  
  
}  
someFunction(1, 2)
```

Functions

- Default Parameter: Ta có thể định nghĩa một giá trị mặc định cho tham số
- Variadic Parameters: Tham số variadic có thể nhận 0 hoặc nhiều giá trị của một loại biết xác định

```
func sumInt(firstNumber: Int = 0, _  
secondNumber: Int = 1) -> Int {  
    let sum = firstNumber + secondNumber;  
    return sum;  
}  
print(sumInt(2, 4)) // Print: 6  
print(sumInt(7))    // Print: 8  
print(sumInt())     // Print: 1
```

```
func sumDouble(numbers:  
Double...) -> Double {  
    var total: Double = 0;  
    for number in numbers {  
        total += number  
    }  
    return total  
}  
print(sumDouble(5,8,2.1))
```

Functions

- Constant and Variable Parameters
 - Tham số của hàm được mặc định là những hằng số. Việc cố gắng thay đổi giá trị của tham số sẽ gây ra lỗi compile-time. Tuy nhiên Swift hỗ trợ một cách để sử dụng tham số như là một biến trong hàm. Bằng cách thêm từ khoá var phía trước tên biến.

```
func increase(var number: Int) -> Int {  
    number++  
    return number  
}
```

```
print(increase(4))
```

Functions

- In-Out Parameters
 - Tham số chỉ tồn tại trong phạm vi của hàm. Nếu muốn thay đổi giá trị của một tham số của hàm và sự thay đổi đó vẫn còn khi hàm kết thúc, hãy định nghĩa đó là một tham số in-out.
 - Khi gọi hàm phải đặt dấu (&) ngay trước tên biến để cho trình biên dịch biết biến đó có thể thay đổi trong hàm.

```
func swapInts( a: inout Int, _ b:inout Int) {  
    let temp = a  
    a = b  
    b = temp  
}  
  
var a: Int = 5  
var b: Int = 24  
swapInts(a: &a, &b);  
print("\(a)")      // Print 24  
print("\(b)")      // Print 5
```

Closure

- Closure là một block code, có thể tách ra để tái sử dụng.
- Đơn giản hơn thì Closure là function, nhưng khuyết danh.
- Closure có thể được gán vào biến và sử dụng như các kiểu value khác
- Closures có thể là 1 trong 3 loại sau:
 - Global functions: là closures có tên và không “capture” các giá trị.
 - Nested functions: là closures có tên và có thể “capture” các giá trị từ function chứa nó.
 - Closure expressions: là closures không có tên được viết dưới dạng giản lược syntax và có thể “capture” các giá trị từ các bối cảnh xung quanh.

Closure

- Cú pháp:

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

- Ví dụ

// Declare a variable to hold a closure

```
var add: (Int, Int) -> Int
```

// Assign a closure to a variable

```
add = { (a: Int, b: Int) -> Int in  
  return a + b  
}
```

// Or combine like this

```
var sub = { (a: Int, b: Int) -> Int in  
  return a - b  
}
```

```
add(1, 2)  
sub(1, 2)
```


Closure

- Note: parameters trong Closure có thể là kiểu “in-out”, variadic, tuples, nhưng không thể có default value
- So với Function thì Closure đã được viết ra với mục đích ngắn gọn nhất có thể

```
// Declare a variable to hold a closure
var add: (Int, Int) -> Int

/** SHORTHAND SYNTAX */
// Not need return keyword when only have single return statement
add = {(a: Int, b: Int) -> Int in
    a + b
}
add(1, 1)
```

```
// Remove return type and parameters type
// Because we already declare: var
add: (Int, Int) -> Int
add = {(a, b) in
    a + b
}
add(9, 2)
```

```
// Remove parameters, Swift will refer parameters by number, start from 0:
add = {
    $0 + $1
}
add(99, 1)
```

Closure

- So sánh closure và function

Closure	Function
<ul style="list-style-type: none">• Có tên• Có func• Không có in	<ul style="list-style-type: none">• Không có tên• Không có func• Có in

Closure

- Các bước để chuyển function về closure
- Ví dụ: chuyển function *sayHello* về closure

```
func sayHello(name: String) -> String {  
    return "Hello \(name)"  
}
```

- Bước 1: Loại bỏ dấu ngoặc

```
func sayHello(name: String) -> String  
    return "Hello \(name)"
```

- Bước 2: Thêm từ khoá **in** vào giữa các đối số và thân hàm

```
func sayHello(name: String) -> String in  
    return "Hello \(name)"
```

Closure

- Bước 3: Loại bỏ từ khoá **func** và tên hàm
 (name: String) -> String in
 return "Hello \ \(name)"
- Bước 4: Bao đóng nó bởi cặp ngoặc nhọn

```
{ (name: String) -> String in  
    return "Hello \ \(name)"  
}
```

Closure

- Closure như là parameter cho function, Trailing closure syntax
 - Với function thì ta hoàn toàn có thể truyền vào cho function khác dưới dạng arguments, tuy nhiên trước khi truyền thì phải define function sẽ dùng làm argument:

//Declare a function

```
func add(_ a: Int, _ b: Int) -> Int {  
    return a + b  
}
```

//Declare other function has parameter is a function/closure

```
func dynamicCalculate(_ a: Int, _ b: Int, _ c: Int, _ paramFunction:  
(Int, Int) -> Int) -> Int {  
    let result = paramFunction(a,b)  
    return result - c  
}  
  
dynamicCalculate(1, 2, 3, add)
```

Closure

- Đối với Closure thì đơn giản hơn, ta có thể define closure inline:

//Declare other function has parameter is a function/closure

```
func dynamicCalculate(_ a: Int, _ b: Int, _ c: Int, _ paramFunction:  
(Int, Int) -> Int) -> Int {  
    let result = paramFunction(a,b)  
    return result - c  
}
```

//Do it Closure way, define Closure inline

```
let a = dynamicCalculate(4, 5, 6, { (a: Int, b: Int) -> Int in  
    return a + b  
})
```

//Or shorter

```
dynamicCalculate(7, 8, 9, { $0 + $1})
```

Closure

- Đối với những function có parameter cuối cùng là Closure, có thể viết lại function call dưới dạng Trailing Closure như sau:

```
//From  
dynamicCalculate(7, 8, 9, { $0 + $1})
```

```
//To Trailing Closure  
dynamicCalculate(7, 8, 9) {  
    $0 + $1  
}
```

Closure

- Các function sử dụng closure
 - *sorted()*: dùng để thay đổi điều kiện sort cho array/collection...

//SORTED

```
let ages: [Int] = [1, 6, 99, 2, 5, 3, 57, 28, 19]  
var sortedAges = ages.sorted()
```

//Using custom closure to change sort order

```
sortedAges = ages.sorted {  
    $0 > $1  
}
```


Closure

- *filter()*: dùng để lọc các phần tử của collection/array với điều kiện nhất định

```
//FILTER
```

```
let memberAges: [Int] = [1, 6, 12, 17, 19, 22, 35, 43, 57]
let adults = memberAges.filter {
    $0 > 17
}
```

- *map()*: dùng để áp dụng điều kiện nào đó cho tất cả các item trong array/collection

```
//MAP
```

```
let carSalePrices = [10_000, 34_000, 45_000, 99_000, 103_000,
999_696]
let carPriceInVietNam = carSalePrices.map {
    $0 * 200/100
}
```

Closure

- `reduce()`: dùng để tính tổng của array...

```
//REDUCE
```

```
//Note 39_000 = 39000, this is Swift way to make easy to read  
source code
```

```
//Array of tuple (tuple format: (Book name, book quantity, book  
price)
```

```
let books: [(String, Int, Double)] = [("A", 12, 39_000),  
                                       ("B", 9, 22_000),  
                                       ("F", 22, 13_000),  
                                       ("T", 4, 9_000)]
```

```
//Using reduce to calculate total prices of all book
```

```
let sumPrice = books.reduce(0) {  
    //Init value (0) or current value + (number if books * price  
of each book)  
    $0 + Double($1.1) * $1.2  
}
```

Enumerations

- Trong Swift, sử dụng từ khóa **enum** để định nghĩa một tập hợp có số phần tử cố định và liệt kê sẵn, không thể thêm hoặc bớt số phần tử
- Ví dụ định nghĩa ra một tập hợp các ngày trong tuần (Thứ 2, thứ 3,... , chủ nhật).

```
enum WeekDay {  
    // Các phần tử  
    case MONDAY  
    case TUESDAY  
    case WEDNESDAY  
    case THURSDAY  
    case FRIDAY  
    case SATURDAY  
    case SUNDAY  
}  
  
func getJob(weekDay: WeekDay) -> String {  
    if (weekDay == WeekDay.SATURDAY || weekDay ==  
        WeekDay.SUNDAY) {  
        return "Nothing"  
    }  
    return "Coding"  
}
```

Enumerations

- Duyệt trên các phần tử của enum
 - Không có phương thức sẵn có nào cho phép lấy ra danh sách các phần tử của một enum bất kì
 - Để giải quyết vấn đề này, có thể tự định nghĩa biến chứa tất cả các phần tử của enum

```
enum Season {  
    case Spring  
    case Summer  
    case Autumn  
    case Winter  
    // Một hằng số tĩnh, chứa tất cả  
    các phần tử (element) của Enum.  
    static let allValues = [Spring,  
    Summer, Autumn, Winter]  
}
```

```
func test_getAllSeasons() {  
    for season in Season.allValues {  
        print(season)  
    }  
}
```

Enumerations

- Enum trong câu lệnh switch: cũng giống với các kiểu dữ liệu nguyên thủy (Int, Float,..) Enum có thể sử dụng như một tham số trong câu lệnh switch

```
func test_switchEnum() {  
  
    var day = WeekDay.THURSDAY;  
    switch (day) {  
    case .MONDAY:  
        print("Working day");  
    case .SATURDAY, .SUNDAY :  
        print("Holiday");  
    default:  
        print(day);  
    }  
}
```

Enumerations

- Enum với dữ liệu thô (raw value)
 - Có thể định nghĩa một Enum với các giá trị thô (raw values)
 - Các giá trị thô ở đây có thể là kiểu String, character, Int, Number,...
 - Một Enum như vậy mở rộng một trong các kiểu String, Character, Int,...

```
enum Month : Int {  
    // Gán giá trị thô (raw value) cho phần tử đầu tiên.  
    // (Nếu một phần tử không được gán giá trị thô,  
    // Theo mặc định giá trị của nó là giá trị của phần tử đứng trước  
    cộng thêm 1).  
    case January = 1, February, March, April, May, June,  
    July, August, September, October, November, December  
  
    static let allValues = [January, February, March, April, May, June,  
    July, August, September, October, November, December]  
}
```

Enumerations

- In ra các phần tử của enum Month, và các giá trị thô (raw value) tương ứng.

```
func test_MonthEnum() {  
  
    print("All element/raw value of Month enum");  
    for e in Month.allValues {  
        let rawValue = e.rawValue  
        print("Element \(e), raw value: \(rawValue)" );  
    }  
}
```

Enumerations

- Enum với các giá trị liên hợp: có thể tạo ra một **Enum** với các phần tử có thể chứa các giá trị liên hợp với nó

```
enum Promotion {  
    // Tặng quà (Tên quà tặng và số lượng)  
    case GIFT (name: String, quantity: Int)  
    // Giảm giá (phần trăm: 0 - 100)  
    case DISCOUNT(percent: Int)  
    // Tặng tiền (Số tiền).  
    case MONEY(amount : Int)  
    // Miễn phí giao hàng.  
    case SHIPPING  
}
```


Enumerations

```
func getRandomPromotion() -> Promotion {
    // Một giá trị ngẫu nhiên
    var random: UInt32 = arc4random()
    // Phép chia cho 4 và lấy số dư (0, 1,
    2, 3).
    random = random % 4
    switch random {
    case 0:
        return Promotion.DISCOUNT(percent:
10)
    case 1:
        return Promotion.GIFT(name: "Doll",
quantity: 1)
    case 2:
        return Promotion.MONEY(amount: 100)
    case 3:
        return Promotion.SHIPPING
    default:
        return Promotion.SHIPPING
    }
}
```

```
func test_Promotion() {

    var myProm = getRandomPromotion()
    print("Get Promotion: \(myProm)")

    switch myProm {
    case .DISCOUNT(let percent):
        print("Percent is: \(percent)")
    case .GIFT(let name, let quantity ):
        print("Gift name: \(name), quantity:
\(quantity)")
    case .MONEY(let amount) :
        print("Amount: \(amount)")
    default:
        print(myProm)
    }
}
```

Enumerations

- Phương thức trong enum: Bạn có thể định nghĩa các phương thức trong enum

```
enum Currency {  
    case USD  
    case YEN  
    case VND  
  
    func description() -> String {  
        switch self {  
            case .USD:  
                return "America's currency"  
            case .YEN:  
                return "Japan's currency"  
            case .VND :  
                return "Vietnam's currency"  
        }  
    }  
}
```

Class và Struct

- *Class* và *Struct* là những thành phần code chính trong hầu hết mọi ứng dụng iOS
- Giúp tổ chức và quản lý code thành những khối một cách trực giác và dễ dàng sử dụng

```
struct Resolution {  
    var width = 0  
    var height = 0  
}
```

```
class VideoMode {  
    var resolution = Resolution()  
    var interlaced = false  
    var frameRate = 0.0  
    var name: String?  
}
```

So sánh Class và Struct

- Điểm tương đồng
 1. Cho phép khai báo những thuộc tính (property) để lưu trữ giá trị (//1), khai báo những phương thức (methods) để cung cấp chức năng (//2), khai báo subscripts để cung cấp khả năng truy cập giá trị của chúng sử dụng cú pháp subscript
 2. Cho phép định nghĩa constructor (init) để khởi tạo giá trị ban đầu (//2)
 3. Cho phép mở rộng chức năng xa so với hiện thực mặc định ban đầu (//4)
 4. Cho phép hiện thực protocols để cung cấp những chức năng tiêu chuẩn của một loại hình nào đó (//4). Protocol trong Swift tương tự như interface trong ngôn ngữ lập trình Java. Một protocol bao gồm tên những hàm chưa được hiện thực chi tiết. Khi một class hay struct adopt protocol đó, nó phải hiện thực tất cả các hàm đó.

So sánh Class và Struct

- Ví dụ

```
struct Bird {  
    var code: Int //1  
    var name: String //1  
  
    init(code: Int, name: String) { //3  
        self.code = code  
        self.name = name  
    }  
  
    func introduce(){ //2  
        print("I am \(name).")  
    }  
}
```

```
protocol Flyable{  
    func fly()  
}  
  
extension Bird: Flyable { //4  
    func fly() {  
        print("I can fly.")  
    }  
}
```

So sánh Class và Struct


- Điểm khác nhau: Class có những khả năng mà struct không có như sau:
 - Tính kế thừa (inheritance)
 - Classes hỗ trợ thừa kế trong khi Structs thì không.
 - Thừa kế là một đặc tính không thể thiếu được trong lập trình hướng đối tượng. Nó cho phép một lớp thừa hưởng những thuộc tính và hành vi của lớp cha.

```
class Vehicle{  
    var manufacturer: String?  
    let passengerCapacity: Int  
  
    init(passengerCapacity: Int) {  
        self.passengerCapacity =  
passengerCapacity  
    }  
}
```

```
class Car: Vehicle {  
    var fuelType: String?  
}
```

```
let car = Car(passengerCapacity: 4)
```

```
struct Driver: People {  
  
}
```

 Inheritance from non-protocol type 'People'

So sánh Class và Struct

- Kiểu tham chiếu và kiểu giá trị (Reference Types vs. Value Types)
 - Trong Swift, Struct là kiểu giá trị trong khi Class là kiểu tham chiếu
- Ví dụ: Với Struct

```
struct Location{  
    var longitude: Double  
    var latitude: Double  
  
    init(longitude: Double,  
latitude: Double) {  
        self.longitude = longitude  
        self.latitude = latitude  
    }  
}
```

```
var location1 = Location(longitude:  
1.23, latitude: 1.23)  
var location2 = location1  
  
location1.longitude = 4.56  
  
print(location1.longitude) //4.56  
print(location2.longitude) //1.23
```

So sánh Class và Struct

- Ví dụ với Class

```
class Location{  
    var longitude: Double  
    var latitude: Double  
  
    init(longitude: Double, latitude:  
Double) {  
        self.longitude = longitude  
        self.latitude = latitude  
    }  
}
```

```
var location1 = Location(longitude:  
1.23, latitude: 1.23)  
var location2 = location1
```

```
location1.longitude = 4.56
```

```
print(location1.longitude) //4.56  
print(location2.longitude) //4.56
```


So sánh Class và Struct

- Toán tử đồng nhất thức (Identity Operators): Có hai khái niệm cần quan tâm
 - “identical to” được biểu diễn bởi **ba** dấu `===`
 - Ý nghĩa: Hai biến hoặc hằng của kiểu class tham chiếu tới chính xác cùng một instance của class
 - “equal to” được biểu diễn bởi **hai** dấu `==`
 - Ý nghĩa: Hai instance được xem như bằng nhau (equal) hoặc tương đương nhau (equivalent) trong giá trị (value)

```
class Car{}  
let toyota = Car()  
let lexus = toyota
```

```
10 == 10 // true  
"same string" == "same string" // true  
"one string" == "different string" // false
```

```
toyota === lexus //true
```

```
let honda = Car()  
honda === toyota //false
```

So sánh Class và Struct

- Deinitializer
 - Cho phép instance của một class phải phóng bất cứ tài nguyên nào mà nó đã gán.
 - Hàm deinitializer được gọi ngay trước khi instance của một class được giải phóng (trả lại bộ nhớ đã được cấp phát tới ram)
 - Viết hàm deinitializer với từ khoá **deinit**

```
class D {  
    deinit {  
        //Deallocated from the heap, tear down things here  
        print("Deallocated from the heap")  
    }  
}
```

```
var d:D? = D()  
d = nil //Deallocated from the heap
```

Chọn giữa Classes và Structs

- Những instances của struct luôn luôn được truyền bởi giá trị và instance của class luôn luôn được truyền bởi tham chiếu
- Chúng phù hợp cho mỗi loại hình công việc khác nhau. Tùy vào cấu trúc dữ liệu và chức năng cần cho project mà quyết định nên sử dụng class hay struct
- Những lý do xem xét tạo struct:
 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 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

Properties

- Properties liên kết các giá trị với một class, structure hay enumeration cụ thể.
- Properties bao gồm hai loại: stored properties và computed properties
 - Stored properties
 - Lưu trữ các giá trị constant và variable như là một phần của một instance
 - Được cung cấp bởi chỉ class và structure
 - Computed properties
 - Có nhiệm vụ tính toán
 - Được cung cấp bởi class, structure và enums.
- Stored và computed properties thường được liên kết với instance của một loại (type) cụ thể.
- Ngoài ra, ta còn có thể định nghĩa property observers để theo dõi sự thay đổi giá trị của property (Property Observers)

Properties

- Stored properties
 - Một stored property là một constant hoặc variable được lưu trữ như là một phần của instance thuộc về một class hoặc structure cụ thể.
 - Stored properties có thể là variable stored properties (với từ khoá var) hoặc constant stored properties (với từ khoá let)

```
struct FixedLengthRange {  
    var firstValue: Int  
    let length: Int  
}
```

```
var rangeOfThreeItems = FixedLengthRange(firstValue: 0, length: 3)  
// the range represents integer values 0, 1, and 2  
rangeOfThreeItems.firstValue = 6  
// the range now represents integer values 6, 7, and 8"
```

Properties

- Lazy Stored Properties
 - Là một property có giá trị khởi tạo không được tính toán cho đến lần sử dụng đầu tiên của nó.
 - Sử dụng từ khoá **lazy** để khai báo một lazy stored property.
 - Lazy property luôn là một variable (với từ khoá var), vì giá trị khởi tạo của nó sẽ không được truy cập cho đến khi quá trình khởi tạo instance được hoàn thành. Trong khi đó, constant properties luôn phải có giá trị trước khi sự khởi tạo được hoàn tất, vì thế, nó không thể là một lazy.

Properties

- Ví dụ lazy Stored Properties

```
class LoadImage {  
    /*  
        LoadImage is a class to load image  
        from an external resource.  
        The class is assumed to take a  
        nontrivial amount of time to initialize.  
    */  
    var imageData = "data.txt"  
}
```

```
class ImageManager {  
    lazy var loader = LoadImage()  
    var data = [String]()  
}  
let manager = ImageManager()  
manager.data.append("Image data")
```

Properties

- Computed properties
 - Là loại property không thật sự lưu trữ value
 - Cung cấp getter và optional setter để nhận và set những properties và giá trị một cách gián tiếp
- Ví dụ

```
struct Point {  
    var x = 0.0, y = 0.0  
}  
struct Size {  
    var width = 0.0, height = 0.0  
}
```


Properties

```
struct Rect {  
    var origin = Point()  
    var size = Size()  
    var center: Point {  
        get {  
            let centerX = origin.x + (size.width / 2)  
            let centerY = origin.y + (size.height / 2)  
            return Point(x: centerX, y: centerY)  
        }  
        set(newCenter) {  
            origin.x = newCenter.x - (size.width / 2)  
            origin.y = newCenter.y - (size.height / 2)  
        }  
    }  
}
```

Properties

```
var square = Rect(origin: Point(x: 0.0, y: 0.0),  
                  size: Size(width: 10.0, height: 10.0))  
let initialSquareCenter = square.center  
square.center = Point(x: 15.0, y: 15.0)  
print("square.origin is now at \(square.origin.x), \(square.origin.y)")  
// Prints "square.origin is now at (10.0, 10.0)"
```

Properties

- Property Observers
 - Quan sát và phản hồi sự thay đổi trong giá trị của property
 - Được gọi mỗi khi một giá trị của property được thiết lập, thậm chí trong trường hợp giá trị mới giống với giá trị hiện tại.
 - Property observers có thể được thêm vào bất cứ stored properties nào mà ta định nghĩa, ngoại trừ lazy
 - Định nghĩa observer cho một properties bằng các lựa chọn sau:
 - Sử dụng willSet, được gọi trước khi giá trị được lưu trữ
 - Sử dụng didSet, được gọi ngay khi một giá trị mới được lưu trữ

Properties

- Ví dụ Property Observers

```
class StepCounter {  
    var totalSteps: Int = 0 {  
        willSet(newTotalSteps) {  
            print("About to set  
totalSteps to \$(newTotalSteps)")  
        }  
        didSet {  
            if totalSteps >  
oldValue {  
                print("Added  
\$(totalSteps - oldValue) steps")  
            }  
        }  
    }  
}
```

```
let stepCounter = StepCounter()  
stepCounter.totalSteps = 200  
// About to set totalSteps to 200  
// Added 200 steps  
stepCounter.totalSteps = 360  
// About to set totalSteps to 360  
// Added 160 steps  
stepCounter.totalSteps = 896  
// About to set totalSteps to 896  
// Added 536 steps
```

Properties

- Type Properties
 - Instance properties là những properties thuộc về một instance của một type cụ thể. Mỗi lần ta tạo mới một instance của type đó, nó có tập hợp những giá trị của property riêng, độc lập với các instance khác.
 - Ngược lại, type properties là những properties chỉ có duy nhất một bản sao của nó, mặc cho bao nhiêu instances được tạo ra

Properties

- Ví dụ: Khai báo stored và computed type properties với từ khoá **static**

```
struct SomeStructure {  
    static var storedTypeProperty =  
    "Some value."  
    static var computedTypeProperty:  
    Int {  
        return 1  
    }  
}  
enum SomeEnumeration {  
    static var storedTypeProperty =  
    "Some value."  
    static var computedTypeProperty:  
    Int {  
        return 6  
    }  
}
```

```
class SomeClass {  
    static var storedTypeProperty =  
    "Some value."  
    static var computedTypeProperty:  
    Int {  
        return 27  
    }  
    class var  
    overrideableComputedTypeProperty:  
    Int {  
        return 107  
    }  
}
```

Properties

```
print(SomeStructure.storedTypeProperty)
// Prints "Some value."
SomeStructure.storedTypeProperty = "Another value."
print(SomeStructure.storedTypeProperty)
// Prints "Another value."
print(SomeEnumeration.computedTypeProperty)
// Prints "6"
print(SomeClass.computedTypeProperty)
// Prints "27"
```

Methods

- Phương thức là một chức năng để liên kết với một đối tượng cụ thể.
- Một class, struct và enum đều có thể định nghĩa một instance method
- Một class, struct và enum cũng có thể có thể định nghĩa một type method.
- Instance Methods
 - Là những hàm chức năng đại diện cho class, struct hay enum cụ thể.
 - Hỗ trợ tính năng của những thể hiện, cung cấp những cách để truy xuất và thay đổi thuộc tính thể hiện hoặc cung cấp những tính năng có liên quan đến mục đích của instance
 - Một instance method có thể ngầm truy xuất đến tất cả các thuộc tính cũng như instance method khác trong cùng một class (struct, enum).

Methods

- Ví dụ:

```
class Counter {  
    var count = 0  
    func increment() {  
        count += 1  
    }  
    func increment(by amount: Int) {  
        count += amount  
    }  
    func reset() {  
        count = 0  
    }  
}
```

```
let counter = Counter()  
// the initial counter value is 0  
counter.increment()  
// the counter's value is now 1  
counter.increment(by: 5)  
// the counter's value is now 6  
counter.reset()  
// the counter's value is now 0"
```

Methods

- Self Property
 - Là từ khóa của Swift tương tự như “*this*” trong java.
 - Trong một phương thức của class, nếu muốn gọi một thuộc tính bên ngoài phương thức thì nên sử dụng từ khóa self để gọi thuộc tính đó, nếu không thì Swift sẽ lấy giá trị của biến ở bên trong phương thức nếu như thuộc tính và biến có tên giống nhau.
 - Sử dụng từ khóa self để phân định rõ ràng giữa thuộc tính của class và biến (hoặc tham số) bên trong hàm.

Methods

- Ví dụ:

```
struct Point {  
    var x = 0.0, y = 0.0  
    func isToTheRightOf(x: Double) -> Bool {  
        return self.x > x  
    }  
}  
  
let somePoint = Point(x: 4.0, y: 5.0)  
if somePoint.isToTheRightOf(x: 1.0) {  
    print("This point is to the right of the line where x == 1.0")  
}  
  
// Prints "This point is to the right of the line where x == 1.0"
```

Methods

- Mutating method: Để những thuộc tính của class, struct hoặc enum có thể thay đổi bên trong một phương thức, thêm từ **mutating** vào trước từ khóa **func** khi khai báo hàm

```
struct Point {  
    var x = 0.0, y = 0.0  
    mutating func moveBy(x deltaX: Double, y deltaY: Double) {  
        x += deltaX  
        y += deltaY  
    }  
}  
  
var somePoint = Point(x: 1.0, y: 1.0)  
somePoint.moveBy(x: 2.0, y: 3.0)  
print("The point is now at \(somePoint.x), \(somePoint.y)")  
// Prints "The point is now at (3.0, 4.0)"  
// Prints "The point is now at (3.0, 4.0)"
```

Methods

- Type Methods
 - Là phương thức được gọi trên một thể hiện của một kiểu cụ thể.
 - Khai báo: thêm từ khoá **class** vào trước từ khoá **func**

```
class SomeClass {  
    class func someTypeMethod() {  
        // type method implementation goes here  
    }  
}  
SomeClass.someTypeMethod()
```

Kế thừa (Inheritance)

- Một lớp có thể kế thừa các phương thức, thuộc tính và các đặc điểm khác từ một lớp khác.
- Khi một lớp kế thừa từ lớp khác, lớp kế thừa được gọi là lớp con và lớp mà nó kế thừa được gọi là lớp cha của nó

```
class Vehicle {  
    var currentSpeed = 0.0  
    var description: String {  
        return "traveling at  
\(currentSpeed) miles per hour"  
    }  
    func makeNoise() {  
        // do nothing – an arbitrary  
        vehicle doesn't necessarily make a noise  
    }  
}
```

```
class Bicycle: Vehicle {  
    var hasBasket = false  
}  
  
let bicycle = Bicycle()  
bicycle.hasBasket = true  
  
bicycle.currentSpeed = 15.0  
print("Bicycle:  
\(bicycle.description)")  
// Bicycle: traveling at 15.0  
miles per hour"
```

Kế thừa (Inheritance)

- Overriding Methods

```
class Train: Vehicle {  
    override func makeNoise() {  
        print("Choo Choo")  
    }  
}
```

- Overriding Property Getters and Setters

```
class Car: Vehicle {  
    var gear = 1  
    override var description: String {  
        return super.description + " in gear \ (gear)"  
    }  
}
```

Kế thừa (Inheritance)

- Overriding Property Observers

```
class AutomaticCar: Car {  
    override var currentSpeed:  
Double {  
        didSet {  
            gear =  
Int(currentSpeed / 10.0) + 1  
        }  
    }  
}
```

```
let automatic = AutomaticCar()  
automatic.currentSpeed = 35.0  
print("AutomaticCar:  
\(automatic.description)")  
// AutomaticCar: traveling at 35.0  
miles per hour in gear 4"
```


Kế thừa (Inheritance)

- Preventing Overrides
 - Sử dụng từ khoá **final** để ngăn chặn kế thừa
 - Ví dụ: `final var`, `final func`, `final class func`
 - Ngoài ra, để ngăn chặn một lớp được kế thừa, đặt từ khoá **final** trước từ khoá **class**
 - Ví dụ: `final class`

Optional Chaining

- Optional chaining là một tiến trình truy vấn và gọi thuộc tính, phương thức và subscript của một optional mà hiện tại là nil
- Nếu optional chứa một giá trị, thuộc tính, phương thức hoặc subscript gọi thành công, còn nếu optional là nil thì thuộc tính, phương thức hoặc subscript được gọi trả về nil
- Nhiều truy vấn có thể bị xích cùng nhau và toàn bộ những chain thất bại nếu bất kỳ liên kết trong chain là nil

Optional Chaining

- Xác định optional chaining bởi dấu chấm hỏi (?) sau giá trị optional khi muốn gọi một thuộc tính, phương thức hoặc subscript nếu optional là non-nil.
- Điều này thì tương tự để thay thế dấu cảm thán (!) sau một giá trị optional để buộc unwrapping giá trị của nó.
- Những khác biệt chính là optional chaining thất bại khi optional là nil
- Kết quả trả về khi gọi optional chaining luôn luôn có giá trị optional

Optional Chaining

- Ví dụ 1:

```
class Person {  
    var residence: Residence?  
}  
class Residence {  
    var numberOfRooms = 1  
}  
  
let john = Person()  
let roomCount =  
john.residence!.numberOfRooms  
// this triggers a runtime error
```

```
if let roomCount =  
john.residence?.numberOfRooms {  
    print("John's residence has  
    \ (roomCount) room(s).")  
} else {  
    print("Unable to retrieve the number  
of rooms.")  
}  
// Prints "Unable to retrieve the number  
of rooms."
```

Optional Chaining

```
let john = Person()
john.residence = Residence()

if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}

// Prints "John's residence has 1 room(s)."
```

Optional Chaining

- Ví dụ 2:

```
struct Person {  
    let firstName: String  
    let middleName: String?  
    let lastName: String  
    let pet: Pet?  
  
    func eat(){  
        print("\(firstName) nom nom")  
    }  
}
```

```
struct Pet {  
    func makeNoise(){  
        print("woof")  
    }  
}
```

```
let vulpes : Pet? = Pet()  
let axel : Person? = Person(firstName:  
    "Axel", middleName: nil, lastName:  
    "Kee", pet: vulpes)
```

```
// makeNoise will only execute if axel  
is not nil and axel's pet is not nil  
axel?.pet?.makeNoise()
```

```
//output 'woof'
```

Type Casting

- Là một cách để kiểm tra kiểu của một thể hiện, hoặc để xử lý thể hiện đó như một superclass khác, hoặc subclass từ một vài nơi khác trong hệ thống class của nó.
- Type casting trong Swift được thực thi với toán tử **is** hoặc **as**. Hai toán tử này cung cấp một cách đơn giản và ấn tượng để kiểm tra kiểu của giá trị hoặc chuyển một giá trị sang một kiểu khác.
- Type casting cũng có thể sử dụng để kiểm tra một kiểu tuân thủ sang một protocol

Type Casting

- Ví dụ

```
class Movie: MediaItem {  
    var director: String  
    init(name: String, director: String) {  
        self.director = director  
        super.init(name: name)  
    }  
}
```

```
class Movie: MediaItem {  
    var director: String  
    init(name: String, director: String) {  
        self.director = director  
        super.init(name: name)  
    }  
}
```

```
class Song: MediaItem {  
    var artist: String  
    init(name: String, artist: String) {  
        self.artist = artist  
        super.init(name: name)  
    }  
}
```


Type Casting

```
let library = [  
    Movie(name: "Casablanca", director: "Michael Curtiz"),  
    Song(name: "Blue Suede Shoes", artist: "Elvis Presley"),  
    Movie(name: "Citizen Kane", director: "Orson Welles"),  
    Song(name: "The One And Only", artist: "Chesney Hawkes"),  
    Song(name: "Never Gonna Give You Up", artist: "Rick Astley")  
]
```

Type Casting

- Checking Type: sử dụng toán tử **is** để kiểm tra một instance có thuộc một subclass hay không
- Ví dụ:

```
var movieCount = 0
var songCount = 0
for item in library {
    if item is Movie {
        movieCount += 1
    } else if item is Song {
        songCount += 1
    }
}
print("Media library contains \(movieCount) movies and \(songCount) songs")
// Prints "Media library contains 2 movies and 3 songs"
```

Type Casting

- Downcasting: Sử dụng toán tử **as?** hoặc **as!** để ép kiểu
 - **as?** : Trả về một giá trị optional
 - **as!** : Trả về một giá trị force-unwrap

- Ví dụ:

```
for item in library {
    if let movie = item as? Movie {
        print("Movie: \(movie.name), dir. \(movie.director)")
    } else if let song = item as? Song {
        print("Song: \(song.name), by \(song.artist)")
    }
}
```

Extension

- Tương tự như categories trong Objective C, extension nói nôm na là giúp bạn mở rộng một class có sẵn nào đó
- Hay nói cách khác, extensions trong Swift cho phép thêm các phương thức mới vào class mà không làm thay đổi mã nguồn của 1 thư viện hay chính class
- Khi nào sử dụng extension:
 1. Khi muốn thêm 1 phương thức mới vào 1 class có sẵn.
 2. Cung cấp thêm 1 số cách để khởi tạo đối tượng
 3. Đáp ứng thêm 1 hoặc nhiều giao thức
 4. Thêm các thuộc tính tính toán
 5. Mở rộng giao thức (Protocol)
 6. Giúp cho cấu trúc code rõ ràng hơn, dễ đọc hiểu hơn

Extension

- Cú pháp:
extension <Tên class muốn extension>

- Ví dụ 1:

```
extension NSDate {  
    func toString(withFormat: String) -> String {  
        let dateFormatter = DateFormatter()  
        dateFormatter.dateFormat = withFormat  
        return dateFormatter.string(from: self)  
    }  
}
```

```
let now = Date() // 2018-09-06 13:00:12 Đây là dạng Date  
print(now.toString("yyyy-MM-dd HH:mm:ss")) // 2018-09-06 13:00:12  
Sau format sẽ ra string.
```

Extension

- Ví dụ 2:

```
class TestClass {  
    var i = 3  
}  
  
extension TestClass {  
    func demoFunction() {  
        print("i is equal to \(i)")  
    }  
    var demoComputedValue: Int {  
        get {  
            return 10  
        }  
        set {  
            print("\(newValue)")  
        }  
    }  
}
```

```
let test = TestClass()  
test.demoFunction() //i is equal to 3  
print(test.demoComputedValue) //10  
test.demoComputedValue = 20 //20
```

Extension

- Ví dụ 3:

```
class TestClass {  
    var i = 3  
}
```

```
protocol TestProtocol {  
    func protocolFunction()  
}
```

```
extension TestClass: TestProtocol {  
  
    func protocolFunction() {  
        print("protocol function")  
    }  
}
```

Protocol

- Protocol là thành phần trừu tượng cho phép bạn khai báo danh sách các phương thức và thuộc tính nhưng không cài đặt các phương thức này
- Protocol được sử dụng làm lớp cơ sở cho bất kỳ class, struct, enum nào cũng có thể áp dụng thực thi. Class, struct, enum áp dụng protocol sẽ cài đặt các phương thức được khai báo trong protocol
- Cú pháp:

```
protocol SomeProtocol {  
    // protocol definition goes here  
}
```


Protocol

- Để thực thi một protocol, sử dụng toán tử “:” tương tự việc kế thừa

```
struct SomeStructure: FirstProtocol, AnotherProtocol {  
    // structure definition goes here  
}
```

```
struct SomeStructure: FirstProtocol, AnotherProtocol {  
    // structure definition goes here  
}
```

Protocol

- Khai báo thuộc tính
 - Khai báo phương thức trong protocol tương tự khai báo hàm thông thường, nhưng không cài đặt. Việc cài đặt các phương thức này sẽ được class/struct/enum nào áp dụng thực hiện
 - Ngoài ra, protocol cũng cho phép khai báo phương thức mutating để hỗ trợ người dùng thực hiện việc thay đổi giá trị nội tại của class/struct/enum thực thi protocol

```
protocol StreetLegal {  
    var chapter: String {get}  
    var isActive: Bool {get, set}  
}
```

- Khi định nghĩa thuộc tính chỉ có {get} , chúng ta có thể khai báo là **var** hoặc **let** đều được nhưng nên khai báo là **let** do không thay đổi được giá trị.
- Khi định nghĩa thuộc tính có cả {get, set} , ta phải khai báo là **var** do được phép thay đổi giá trị.

Protocol

- Khai báo phương thức
 - Khai báo phương thức trong protocol tương tự khai báo hàm thông thường, nhưng không cài đặt. Việc cài đặt các phương thức này sẽ được class/struct/enum nào áp dụng thực hiện.
 - Ngoài ra, protocol cũng cho phép khai báo phương thức **mutating** để hỗ trợ người dùng thực hiện việc thay đổi giá trị nội tại của class/struct/enum thực thi protocol

```
protocol StreetLegal {  
    var chapter: String {get}  
    var isActive: Bool {get, set}  
  
    func signalStop()  
    func signalTurnLeft()  
    func signalTurnRight()  
}
```

Protocol

- Áp dụng thực thi protocol

```
class Bicycle: StreetLegal {  
    // chapter được chỉ định {get}  
    trong protocol nên ta phải khai báo  
    let để người dùng không thay đổi giá  
    trị.  
    let chapter: String = "Chapter1:  
Bicycle Legal."  
    // isActive được chỉ định {get,  
    set} nên ta khai báo var để người dùng  
    có thể gán giá trị.  
    var isActive: Bool = true
```

```
func signalStop() {  
    print("Bending left arm  
downwards")  
}  
func signalTurnLeft() {}  
    func signalTurnRight() {}  
    // Method của Bicycle.  
    func startpedaling() {  
        print("Here we go.")  
    }  
}
```

Protocol

- Protocol for class-only: protocol chỉ sử dụng được cho class.

```
protocol ForClassProtocol: class {  
    func test()  
}
```

```
class TestClass: ForClassProtocol {  
    // OK  
    func test() {}  
}
```

```
struct TestStruct: ForClassProtocol  
{ // Error  
    func test() {}  
}
```

```
enum TestEnum: ForClassProtocol { //  
    Error  
    func test() {}  
}
```

Protocol

- Protocol còn được sử dụng vào mục đích tạo delegate
- delegate là một design pattern dùng để truyền dữ liệu giữa các class hoặc struct
- Ví dụ:

```
protocol FirstVCDelegate {  
    func passData(data: String)  
}  
  
class FirstVC {  
    var delegate: FirstVCDelegate?  
}
```

```
class SecondVC: FirstVCDelegate {  
    func passData(data: String) {  
        print("Something  
happened")  
    }  
}
```

```
let firstVC = FirstVC()  
let secondVC = SecondVC()
```

Protocol

```
firstVC.delegate = secondVC  // secondVC = delegate
```

```
firstVC.delegate?.passData(data: "a bunch of contracts")  
// "Something happened"
```

Generic

- Là một trong những tính năng mạnh mẽ nhất của Swift
- Phần lớn các thư viện chuẩn Swift được xây dựng với mã generic
- Trong thực tế ta ta đã sử dụng thường xuyên. Ví dụ như Array và Dictionary là hay tập hợp generic
- Ví dụ: non-generic function hàm trao đổi hai giá trị int

```
func oneSwapTwoInts( a: inout Int, b: inout Int) {  
    let temporaryA = a  
    a = b  
    b = temporaryA  
}
```


Generic

```
var someInt = 3  
var anotherInt = 107
```

```
oneSwapTwoInts(&someInt, &anotherInt)
```

```
println("someInt is now \someInt), and anotherInt is now  
\anotherInt)")
```

```
// prints "someInt is now 107, and anotherInt is now 3"
```

Generic

- Phương thức `oneSwapTwoInts` chỉ có thể được sử dụng với giá trị `Int`. Nếu muốn trao đổi 2 giá trị `String`, hoặc 2 giá trị `Double`, ta phải viết nhiều phương thức hơn nữa, như là phương thức `oneSwapTwoStrings` và hàm `swapTwoDoubles`

```
func oneSwapTwoStrings( a: inout String, b: inout String) {  
    let temporaryA = a  
    a = b  
    b = temporaryA  
}
```

```
func swapTwoDoubles( a: inout Double, b: inout Double) {  
    let temporaryA = a  
    a = b  
    b = temporaryA  
}
```

Generic

- Phương thức generic: Generic functions có thể làm việc với bất kỳ kiểu nào
- Ví dụ:

```
func oneSwapTwoValues<T>( a: inout T, b: inout T) {  
  
    let temporaryA = a  
    a = b  
    b = temporaryA  
}
```

Generic

- Sử dụng:

```
var someInt = 3
var anotherInt = 107
oneSwapTwoValues(&someInt, &anotherInt)
// someInt is now 107, and anotherInt is now 3
```

```
var someString = "hello"
var anotherString = "world"
oneSwapTwoValues(&someString, &anotherString)
// someString is now "world", and anotherString is now "hello"
```

Q&A