**IOS**

Contents

[1. The Basics 3](#_Toc132703362)

[1.1 Type Aliases 3](#_Toc132703363)

[1.2 Tuples 3](#_Toc132703364)

[1.3 Optionals 4](#_Toc132703365)

[1.4 Error Handling 6](#_Toc132703366)

[2. Basic Opearators 7](#_Toc132703367)

[3. String and Character 10](#_Toc132703368)

[4. Collection types 14](#_Toc132703369)

[4.1 Array 15](#_Toc132703370)

[4.2 Set(Tạm bỏ) 16](#_Toc132703371)

[4.3 Dictionaries(tạm bỏ) 17](#_Toc132703372)

[5. Control Flow(làm chưa kĩ) 17](#_Toc132703373)

[5.1. Early Exit 17](#_Toc132703374)

[5.2 Checking API(chưa làm) 18](#_Toc132703375)

[6. Functions(Chưa làm) 18](#_Toc132703376)

[7. Closures 18](#_Toc132703377)

[7.1 Sorted Method(Sắp xếp mảng là chủ yếu) 18](#_Toc132703378)

[7.2 Closure Expression Syntax – Các thể loại Closure 18](#_Toc132703379)

[8. Enemerations 21](#_Toc132703380)

[8.1 Khởi tạo enum: 21](#_Toc132703381)

[8.2 Kết hợp Enumeration và Switch 22](#_Toc132703382)

[8.3 Raw value 23](#_Toc132703383)

[8.4 Recursive Enumeration 24](#_Toc132703384)

[9. Structures and Classes 25](#_Toc132703385)

[10. Computed Properties for Class, Struct, Emum 27](#_Toc132703386)

[10.1 Làm quen với Computed properties và Lỗi sinh ra vòng lặp vô hạn 27](#_Toc132703387)

[10.2 Các biến thể của Computed Properties 30](#_Toc132703388)

[10.3 Property Observe(willSet and didSet) 34](#_Toc132703389)

[11. Methods (Bỏ type Method) 36](#_Toc132703390)

[12. Subscripts 37](#_Toc132703391)

[13. Inheritance 39](#_Toc132703392)

[14. Initialization 43](#_Toc132703393)

[14.1 Struct 43](#_Toc132703394)

[14.2 Class Inheritance and Initialization 44](#_Toc132703395)

[14.2 Required Init 49](#_Toc132703396)

[15. Deinitalization 50](#_Toc132703397)

[16. Error Handling 50](#_Toc132703398)

[17. Type Casting 52](#_Toc132703399)

[18. Nested Type 55](#_Toc132703400)

[19. Extensions 56](#_Toc132703401)

[19.1. Computed Properties 57](#_Toc132703402)

[19.2. Initializers 58](#_Toc132703403)

[19.3. Methods 59](#_Toc132703404)

[19.4. Mutating Instances 60](#_Toc132703405)

[19.5 Adding Protocol with an Extension 61](#_Toc132703406)

[20. Protocol 62](#_Toc132703407)

[20.1 Property Requirements 62](#_Toc132703408)

[20.2 Method Requirement 63](#_Toc132703409)

[20.3 Mutating Protocol 64](#_Toc132703410)

[20.4 Initializer Requirements 64](#_Toc132703411)

[20.5 Protocol as Type 65](#_Toc132703412)

[20.6 Optional Protocol 66](#_Toc132703413)

[21. Generic 68](#_Toc132703414)

[22 Automatic reference counting 69](#_Toc132703415)

[22.1 Strong và weak 71](#_Toc132703416)

[23. Safety Memory 72](#_Toc132703417)

[24. Access Control 74](#_Toc132703418)

[24.1 Custom Types Access Control 75](#_Toc132703419)

[II. Các câu hỏi phỏng vấn 76](#_Toc132703420)

[2.1 So sánh class và struct 76](#_Toc132703421)

[2.2 Concurrency – Async Await 78](#_Toc132703422)

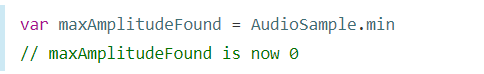
# 1. The Basics

## 1.1 Type Aliases

Type aliases define an alternative name for an existing type. You define type aliases with the typealias keyword



Lúc này ta có thể sử dụng AudioSample giống như UInt16

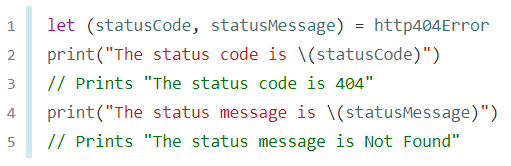


## 1.2 Tuples

Tuples group multiple values into a single compound value. The values within a tuple can be of any type and don’t have to be of the same type as each other.

VD:

Khi khai báo như trên, bên dưới ta có thể phân tách(decompose) nội dung thành các hằng số hoặc các biến để có thể truy cập:



Ta thấy http404Error được tupple thành (404, “Not Found”) sau đó lại được phân ra thành (statuscode, statusMessage). Ta cũng có thể đặt tên cho các thành phần phân ra luôn khi mới couple



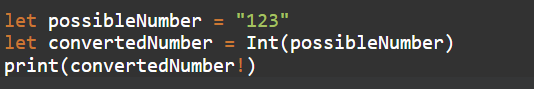
Kết quả in ra: 200

Từ đây ta cũng phát hiện ra để return nhiều giá trị, ta có thể return 1 Tupples

## 1.3 Optionals

Chúng ta sử dụng Optionals khi giá của variable có thể là nil. Một optional đại diện cho 2 khả năng: Nếu có value, ta có thể unwrap optional đó để truy cấp value, TH2 là không có value.

Ta có đoạn code:



Khi ta khai báo như VD trên, thì trình thông dịch sẽ gán ngầm định như sau:



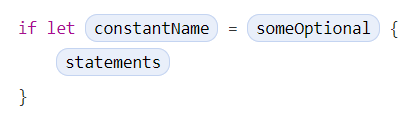
Lúc này biến convertedNumber sẽ là 1 optionals. Lý do là gì?

Lý do lúc này vì biến possibleNumber là 1 chuỗi String, nếu nó là “123” thì việc convert sang int sẽ hoàn tất và gán thành công, nhưng nếu possibleNumber là “aaa” thì việc convert sẽ failed và nó sẽ trả lại giá trị nil. Mà 1 biến thông thường không thể được gán bằng nil, nên lúc này biến convertedNumber sẽ là 1 optional.

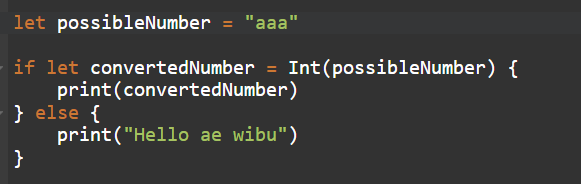
* Optional binding:

Ta có thể sử dụng Optional binding để kiểm tra xem optional có đang chứa value hay không. Optional binding có thể được sử dụng với if và while.

Chúng ta viết 1 optional binding với if như dưới



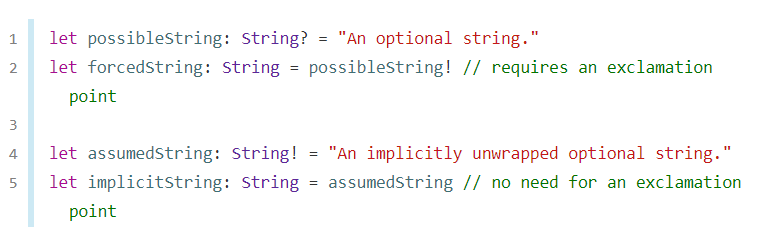
Ta có thể hiểu rằng, 1 biến thông thường không thể được gán trị nil, nên nếu someOptional lúc này có giá trị thì constanName sẽ được gán và chạy các lệnh statements, nhưng nếu someOptional là nil thì sẽ không được gán và sẽ không chạy statements



Kết quả in ra màn hình là “Hello ae wibu” và chương trình không bị crashed.

* Implicitly Unwrapped Optionals

Ta là 1 lập trình viên, và ta biết rõ hơn máy, trong trường hợp này, nó chắc chắn sẽ có value và không thể là nil nên đôi khi ta không cần check và unwrap optional. Lúc này ta sử dụng dấu ! sử dụng Implicit Unwrapp

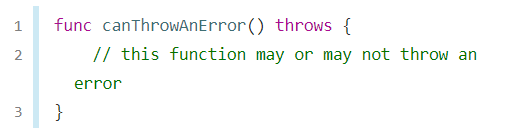


Ta thấy có 2 cách khác nhau, nhưng bên trên ta bắt buộc phải unwrap, bên dưới thì không. Ta hiểu đơn giản là “an implicitly unwrapped optional as giving permission for the optional to be force-unwrapped if needed”.

## 1.4 Error Handling

Chúng ta sử dụng Error handling để phản hồi(respond) mỗi khi chương trình gặp lỗi trong lúc thực thi.  Error handling cho phép ta xác định nguyên nhân cơ bản của lỗi và, nếu cần, ta sẽ thông báo lỗi sang một phần khác trong chương trình của mình.

“When a function encounters an error condition, it throws an error. That function’s caller can then catch the error and respond appropriately: ”



“ When you call a function that can throw an error, you prepend the try keyword to the expression. Swift automatically propagates errors out of their current scope until they’re handled by a catch clause.”



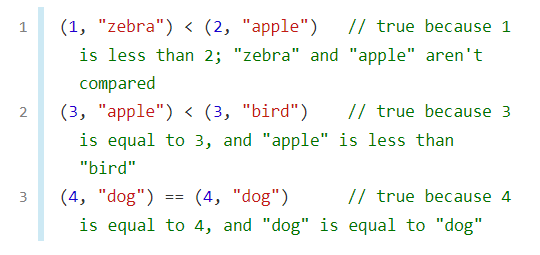
Ta hiểu như sau: “In this example, the makeASandwich() function will throw an error if **no clean dishes are available or if any ingredients(thành phần) are missing**. Because makeASandwich() can throw an error, the function call is wrapped in a try expression. By wrapping the function call in a do statement, any errors that are thrown will be propagated(tuyên truyền, thông báo) to the provided catch clauses.

If no error is thrown, the eatASandwich() function is called. If an error is thrown and it matches the SandwichError.outOfCleanDishes case, then the washDishes() function will be called. If an error is thrown and it matches the SandwichError.missingIngredients case, then the buyGroceries(\_:) function is called with the associated [String] value captured by the catch pattern”

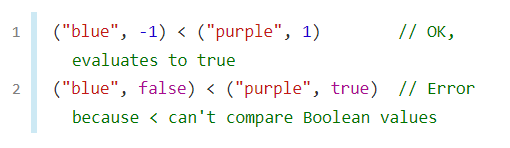
# 2. Basic Opearators

* Comparison Operators

“You can compare two tuples if they have the same type and the same number of values. Tuples are compared from left to right, one value at a time, until the comparison finds two values that aren’t equal. Those two values are compared, and the result of that comparison determines the overall result of the tuple comparison. If all the elements are equal, then the tuples themselves are equal. ”



“In the example above, you can see the left-to-right comparison behavior on the first line. Because 1 is less than 2, (1, "zebra") is considered less than (2, "apple"), regardless of any other values in the tuples. It doesn’t matter that "zebra" isn’t less than "apple", because the comparison is already determined by the tuples’ first elements. However, when the tuples’ first elements are the same, their second elements are compared—this is what happens on the second and third line.”

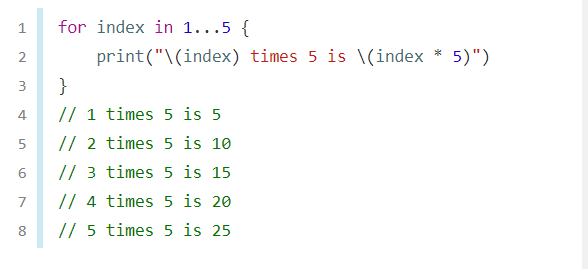


“ As demonstrated in the code above, you can compare two tuples of type (String, Int) because both String and Int values can be compared using the < operator. In contrast, two tuples of type (String, Bool) can’t be compared with the < operator because the < operator can’t be applied to Bool values.”

* Range Operators

“The closed range operator (a...b) defines a range that runs from a to b, and includes the values a and b. The value of a must not be greater than b.”

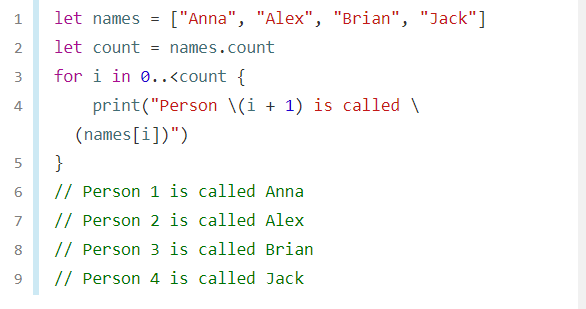
“The closed range operator is useful when iterating(lặp đi lặp lại) over a range in which you want all of the values to be used, such as with a for-in loop:”



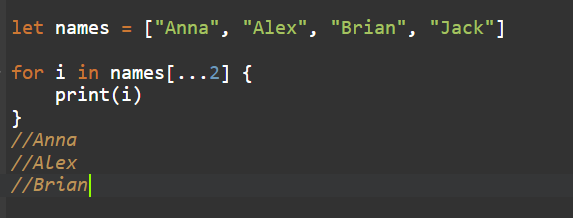
Ta cũng có các khái niệm như Half-Open Range Operator, One-Sided Ranges

* Half-Open range

Giá trị của i sẽ từ 0 đến 3 và sẽ quét hết mảng 4 phần tử

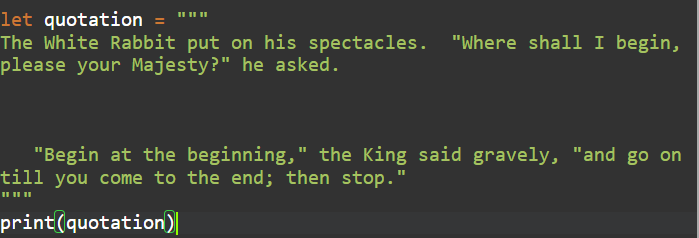


* One-Sided Ranges

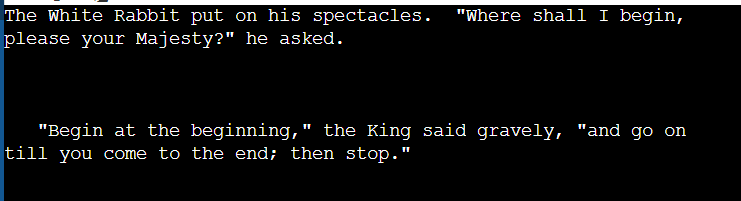


# 3. String and Character

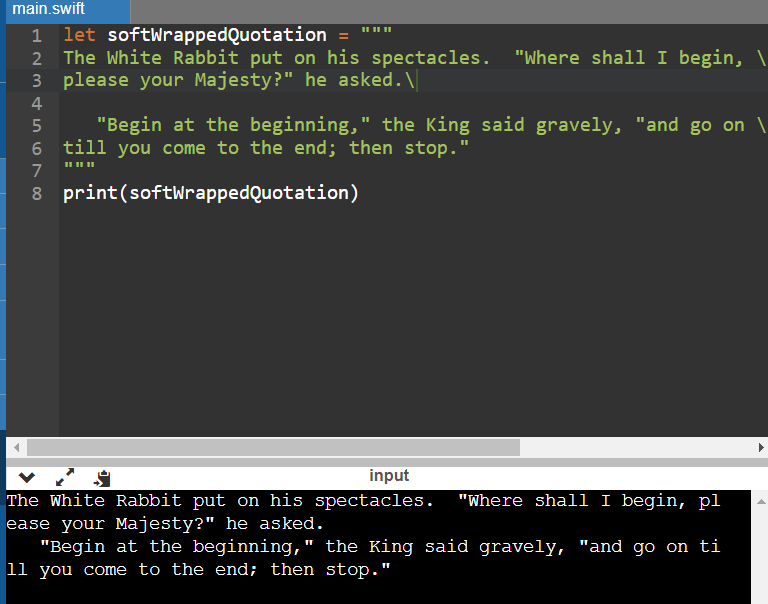
* Multiline String Literals

****

Kết quả:

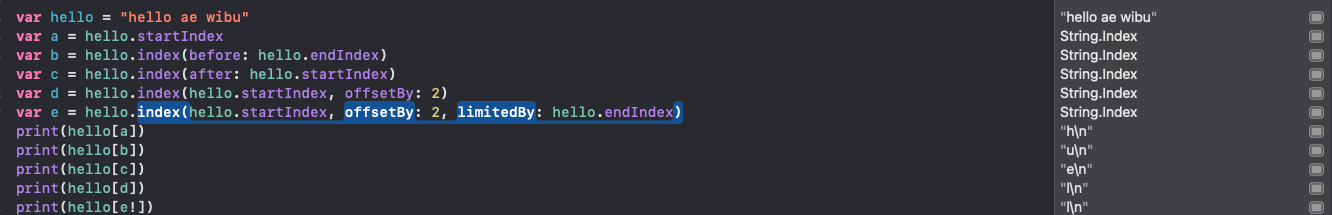


Ta sử dụng \ để nối liền dòng trên và dòng dưới



* Access and Modify a String
* String Indices(Chỉ số):

Để truy cập vào vị trí của phần tử trong String ta sử dụng startIndex để truy cập vị trí đầu tiên của String và endIndex để truy cập vị trí cuối cùng(ta phải lùi lại 1 đơn vị để lấy giá trị cuối cùng). Ta cũng có thể sử dụng them index(before:)và index(after:)và index(\_:offsetBy:)để lấy giá trị ở 1 vị trí cụ thể:

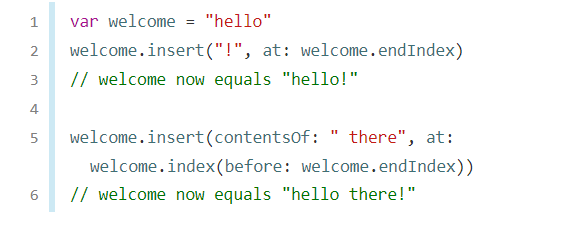


**Sử dụng startIndex, endIndex, index(before: ) và index(after: ) ta sẽ có được vị trí của các kí tự trong String, sau đó sử dụng subscription để lấy value của kí tự đó(VD như: hello[a]).**

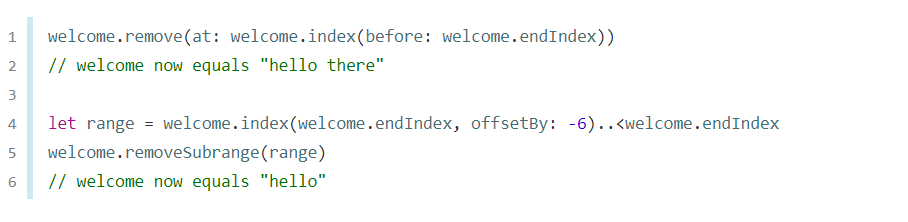
**Việc insert, remove đều phải thông qua index**

* Inserting and Removing:

- Insert



-Remove

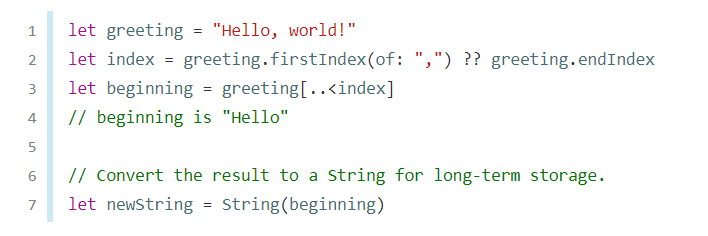


Chú ý: Ta thấy method là removeSubrange là xóa đi 1 khoảng kí tự trong String

Và let range kia là 1 biến có kiểu range 😊)) nghĩa là nó kiểu let range = 1…5

* Substrings

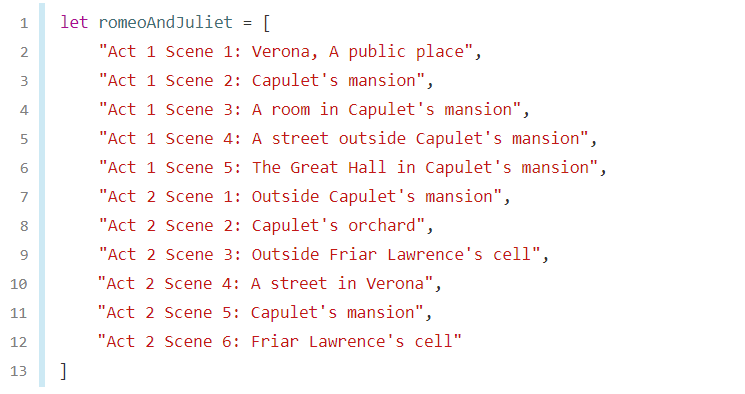
Là ta thu được 1 chuỗi nhỏ từ 1 chuỗi ta muốn

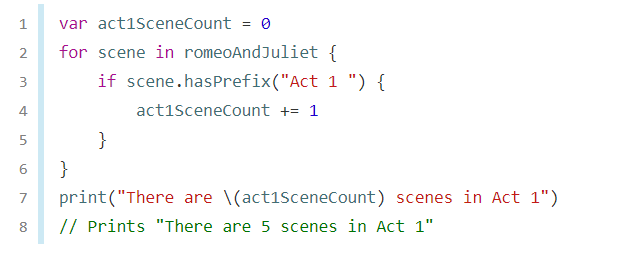


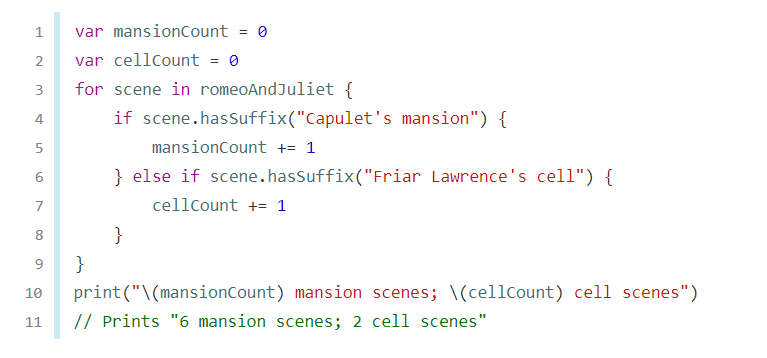
Ta có greeting.firstIndex(of: ",") là trả về vị trí đầu tiên gặp dấu , nếu không có dấu phẩy nào thì lấy giá trị nil.

* Prefix and Suffix Equality

Ta có 2 phương thức hasPrefix(\_:) và  hasSuffix(\_:), phương thức này sẽ kiểm tra xem có tồn tại chuỗi ta mong muốn trong String không, nếu có return true.

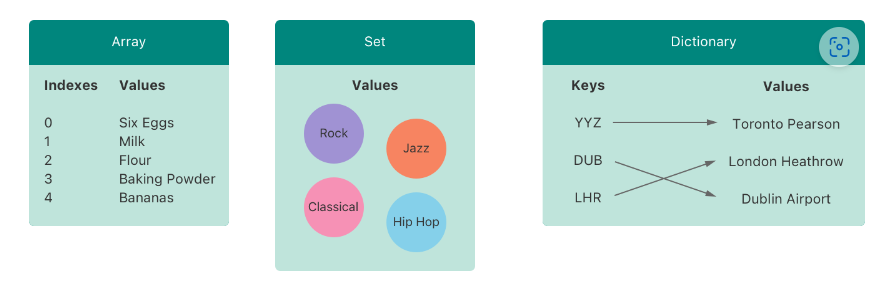






# 4. Collection types

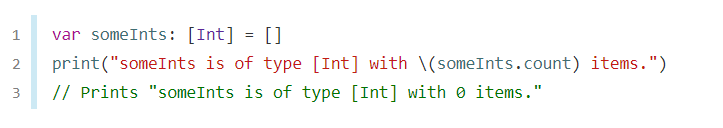
Swift cung cấp cho ta 3 collection types chính là: Arrays, Set và Dicionaries. Arrays tập hợp các giá trị được sắp xếp theo thứ tự. Set là tập giá trị không được sắp xếp theo thứ tự và Dictionaries cũng không theo thứ tự sử dụng cặp key-value



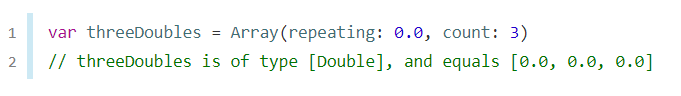
## 4.1 Array

Khởi tạo array 4 cách:

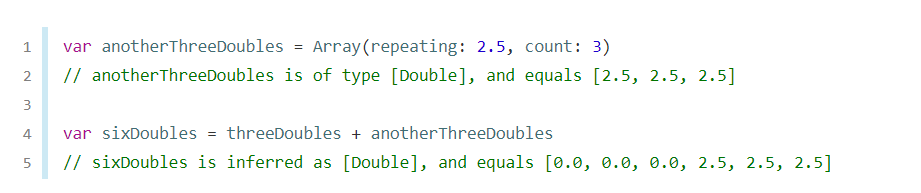
* Khởi tạo 1 empty array



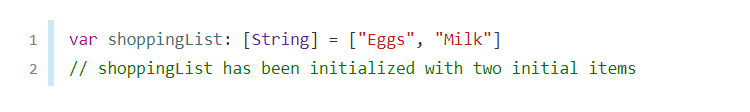
* Khởi tạo array với default value



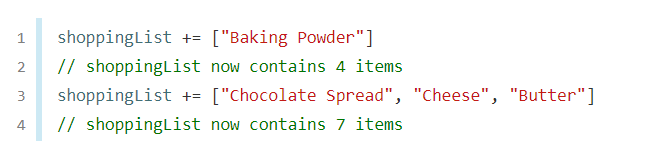
* Khởi tạo array với adding 2 arrays together



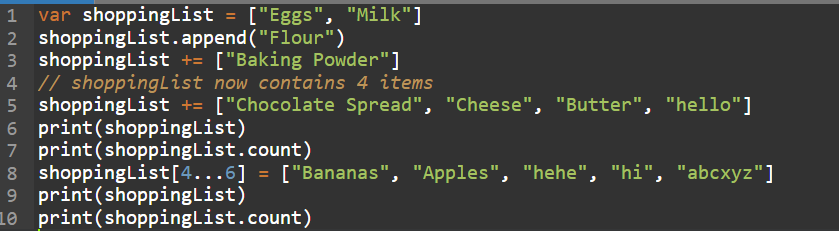
* Khởi tạo với việc gắn giá trị trước



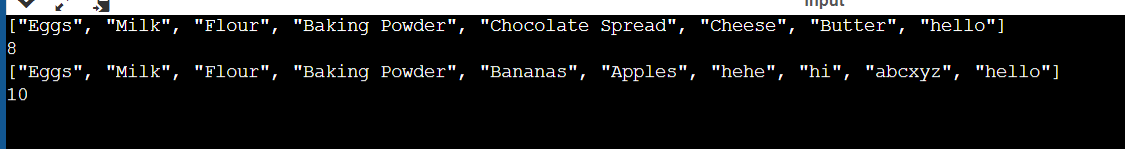
* Các phương thức chính của Array hay dung
  + Cộng Array



* + Sử dụng subscript để change range của array, nó kiểu là phải đọc ví dụ dưới



Kết quả:



Nó kiểu như sẽ thay thế từ vị trí 4 đến 6, sau đó nếu thừa thì sẽ ghi chèn vào đằng sau, chứ ko ghi đè lên phần tử tiếp theo.



Thì sẽ gặp lỗi, vì quá size

## 4.2 Set(Tạm bỏ)

“A set stores distinct values of the same type in a collection with no defined ordering. You can use a set instead of an array when the order of items isn’t important, or when you need to ensure that an item only appears once.”

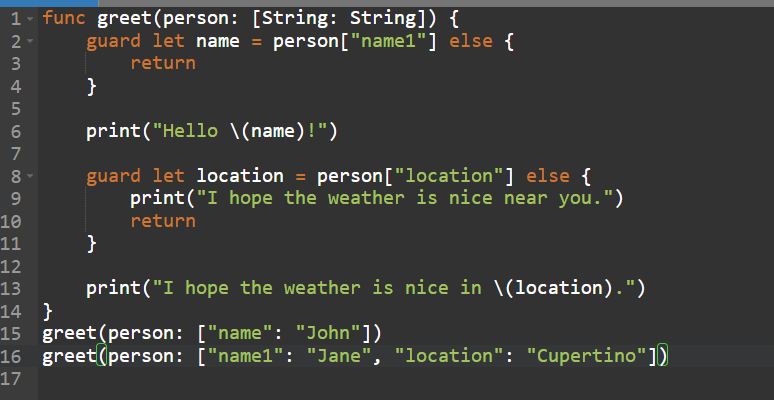
Tạm bỏ

## 4.3 Dictionaries(tạm bỏ)

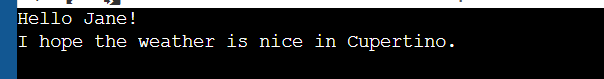
# 5. Control Flow(làm chưa kĩ)

## 5.1. Early Exit

A guard statement, like an if statement, executes statements depending on the Boolean value of an expression. You use a guard statement to require that a condition must be true in order for the code after the guard statement to be executed. Unlike an if statement, a guard statement always has an else clause—the code inside the else clause is executed if the condition isn’t true.



Kết quả:



Giải thích: Nếu không tồn tại key là “name1” thì name = nil và sẽ không được gán, dẫn đến hàm else sẽ được thực hiện nên với lời gọi hàm greet() đầu tiên sẽ không in ra gì. Nhưng với lời gọi hàm sau, ta có key là “location” nên việc gán thành công và else không được thực hiện.

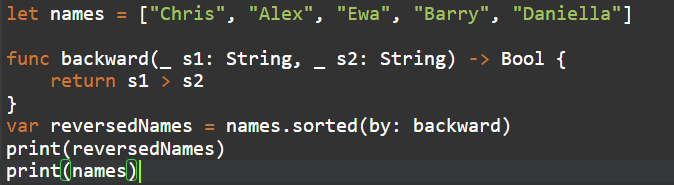
## 5.2 Checking API(chưa làm)

# 6. Functions(Chưa làm)

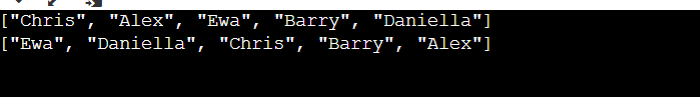
# 7. Closures

## 7.1 Sorted Method(Sắp xếp mảng là chủ yếu)

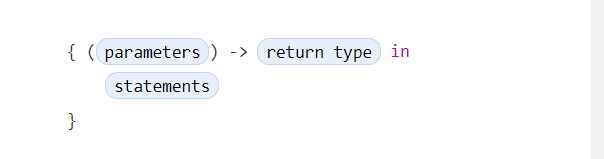
Swift’s standard library provides a method called sorted(by:),  which **sorts** an array of values of a known type, based on the output of a sorting closure that you provide. Once it completes the sorting process, the sorted(by:) method returns a new array of the same type and size as the old one, with its elements in the correct sorted order. **The original array isn’t modified by the sorted(by:) method.**

****

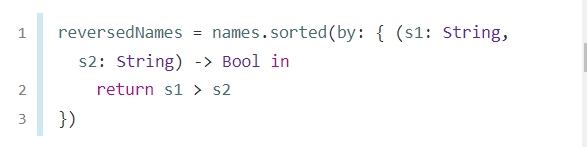
**Output:**

****

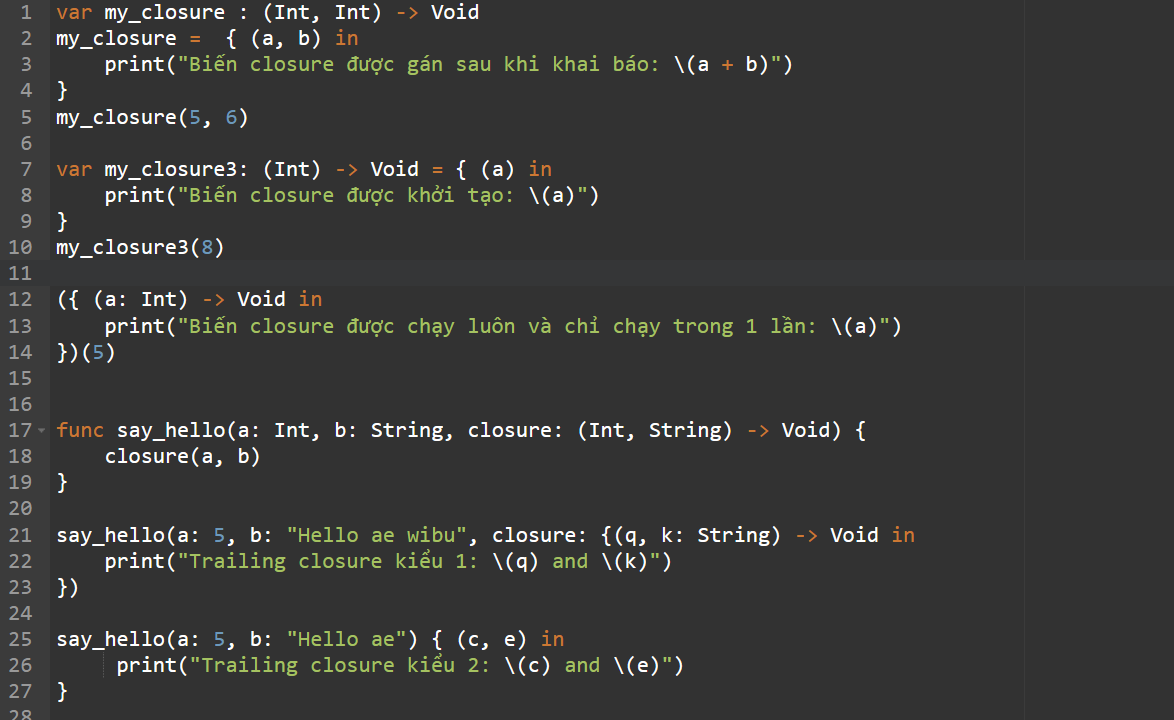
## 7.2 Closure Expression Syntax – Các thể loại Closure



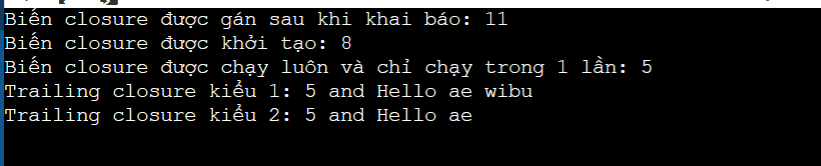
Ta sẽ kết hợp ví dụ trên và dưới ta được:



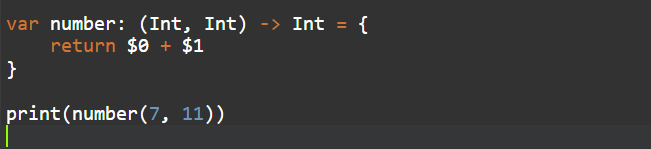
Các closure đặc biệt



Output:



* Closure kiểu chưa biết đặt tên



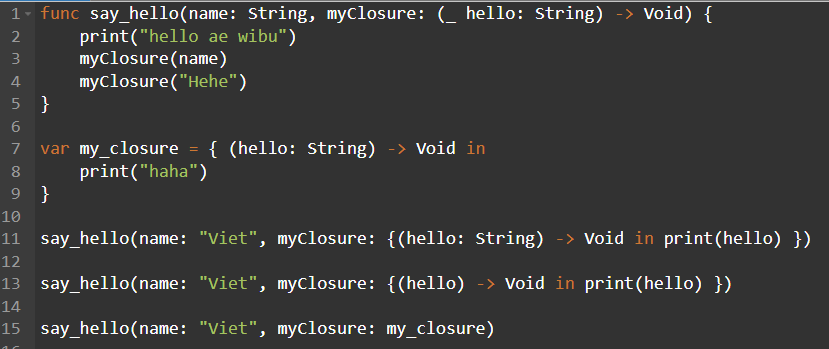
Output:



Ta có $0 sẽ tham chiếu đến số Int đầu, và S1 sẽ tham chiếu đến số Int thứ 2

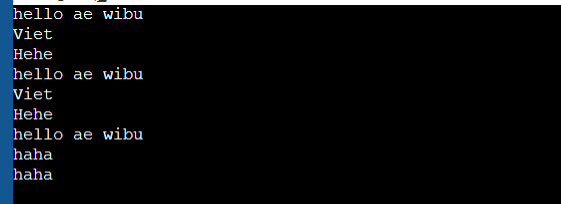
* Closure được sử dụng để làm tham số truyền vào cho function
* Chaining closure

Ta có ví dụ:



Ở khai báo func, ta khai báo 1 function có tham số truyền vào là 1 closure và closure đó có input là String và output là Void.

Ta có 3 hàm call function say\_hello() giống hệt nhau, cách đầu tiên và cách thứ 3 tường minh hơn cách thứ 2. **Ta thấy ở lúc triển khai func, ta có input là String và output là Void, nên lúc triển khai, ta cần viết 1 closure bất kì sao cho cứ có 1 đầu vào là String và đầu ra là Void là được** .Ta được Output

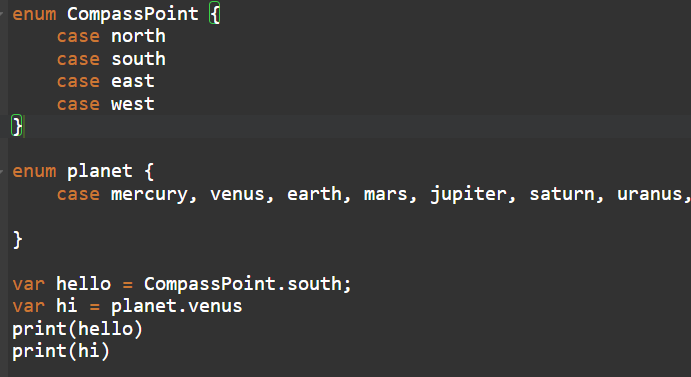


Ở cách viết 2, nó sẽ tự hiểu hello có kiểu String

# 8. Enemerations

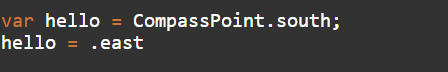
Enumerations in Swift are much more flexible, and don’t have to provide a value for each case of the enumeration. If a value (known as a raw value) is provided for each enumeration case, the value can be a string, a character, or a value of any integer or floating-point type

## 8.1 Khởi tạo enum:



Ta thấy nếu viết trên cùng 1 hàng, ta dung dấu , để tách các case

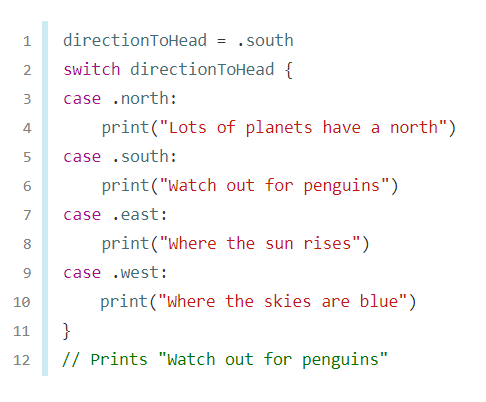
Khi biến hello được khai báo như trên, thì bên dưới ta có thể gán lại giá trị của hello qua lệnh sau:



* Để chỉ rõ kiểu dữ liệu nào gán với từng case nào ta làm như sau:

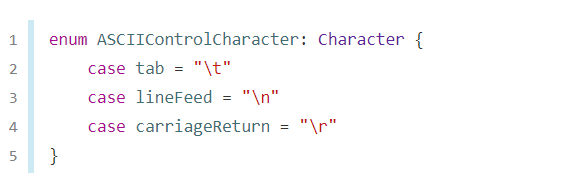


## Kết hợp Enumeration và Switch



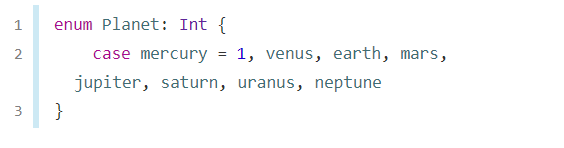
## 8.3 Raw value

Ta có thể gán các giá trị cho từng case thông qua việc gán kiểu dữ kiệu cho enum



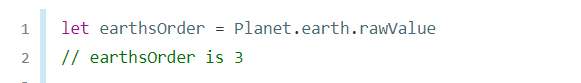
“Raw values can be strings, characters, or any of the integer or floating-point number types. Each raw value must be unique within its enumeration declaration.”

* Implicit Assigned Raw Value

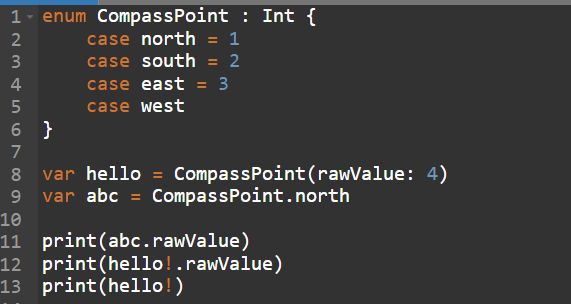


In the example above, Planet.mercury has an explicit raw value of 1, Planet.venus has an implicit raw value of 2, and so on. **When strings are used for raw values, the implicit value for each case is the text of that case’s name.**

You access the raw value of an enumeration case with its rawValue property:



* Gán biến khác cho rawvalue



Output:

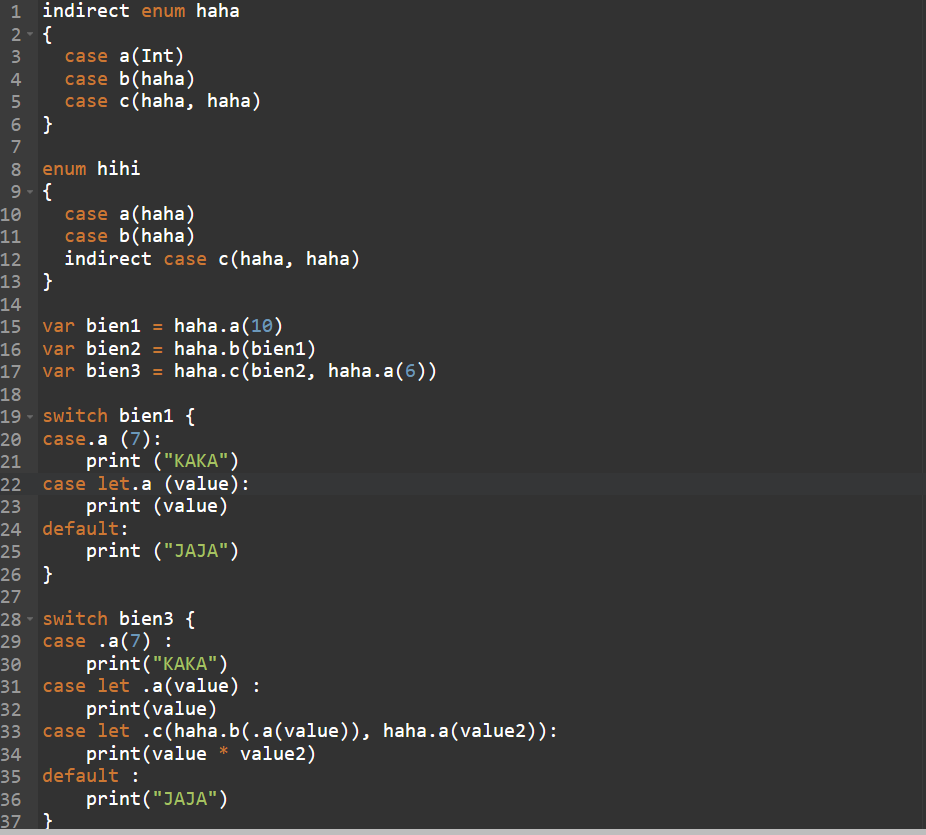


Giải thích:

Ta thấy là với việc khởi tạo: **var hello = CompassPoint(rawValue: 4)** thì sẽ tương đương với **var hello = CompassPoint.west**. Mà giá trị rawValue ta truyền vào có thể là 1 số bất kì như 10, 11, nó sẽ quá kích cỡ của enum, nên hoàn toàn biến hello có thể được gán giá trị là **nil**, nên hello sẽ được tự động gán là kiểu **optional**, vậy để lấy giá trị ta them dấu **!**

## 8.4 Recursive Enumeration

Recusive cho phép ta khai báo kiểu biến bên trung enum trùng kiểu dữ liệu chính nó 😊



Ta thấy case c khai báo kiểu dữ là haha, là tên của chính enum đó



# 9. Structures and Classes

“Structures and classes are general-purpose, flexible constructs that become the building blocks of your program’s code. You define properties and methods to add functionality to your structures and classes using the same syntax you use to define constants, variables, and functions.”

“Unlike other programming languages, Swift doesn’t require you to create separate interface and implementation files for custom structures and classes. In Swift, you define a structure or class in a single file, and the external interface to that class or structure is automatically made available for other code to use.”

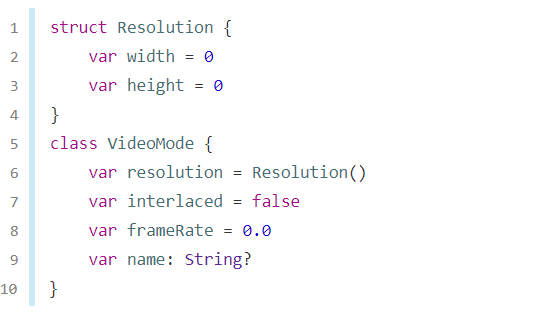
Comparing Structures and Classes

Structures and classes in Swift have many things in common. Both can:

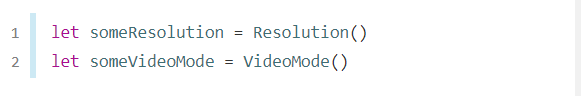
* Define properties to store values
* Define methods to provide functionality
* Define subscripts to provide access to their values using subscript syntax
* Define initializers to set up their initial state
* Be extended to expand their functionality beyond a default implementation
* Conform to protocols to provide standard functionality of a certain kind

Classes have additional capabilities that structures don’t have:

* Inheritance enables one class to inherit the characteristics of another.
* Type casting enables you to check and interpret(thông dịch) the type of a class instance at runtime.
* Deinitializers(Phân giải, giải phóng) enable an instance of a class to free up any resources it has assigned.
* Reference counting allows more than one reference to a class instance.



1. Structure and Class Instances



# 10. Computed Properties for Class, Struct, Emum

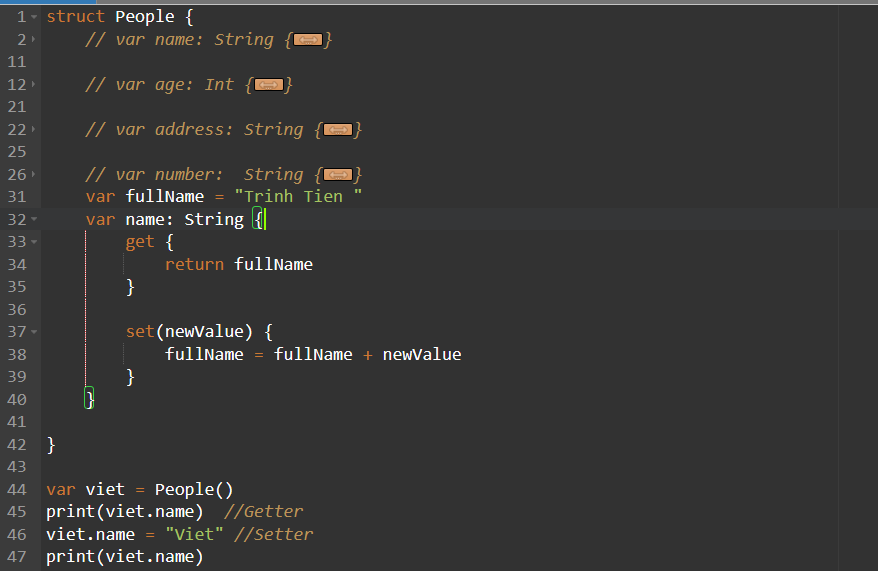
“In addition to stored properties, classes, structures, and enumerations can define **computed propertie**s, which don’t actually store a value. Instead, they provide a **getter** **and an optiona**l **setter** to retrieve and set other properties and values indirectly.”

Tác dụng chính của Computed Properties là có thể gán 1 biến thông qua 1 biến khác, hay có thể lấy giá trị của 1 biến thông qua 1 biến khác, như vd dưới

## 10.1 Làm quen với Computed properties và Lỗi sinh ra vòng lặp vô hạn

Ta sử dụng 2 Properties get và set để lấy giá trị của 1 biến:

VD:



Output:

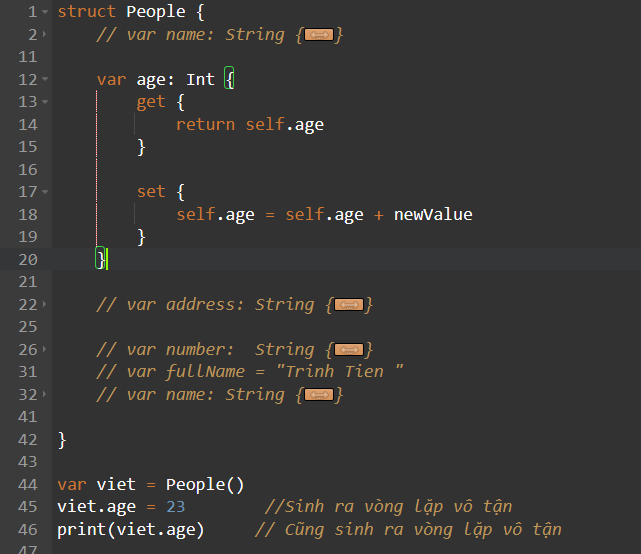


**Chú ý 1**: Để sử dụng Computed Properties ta phải khai báo sẵn kiểu dữ liệu cho biến đó như trên.

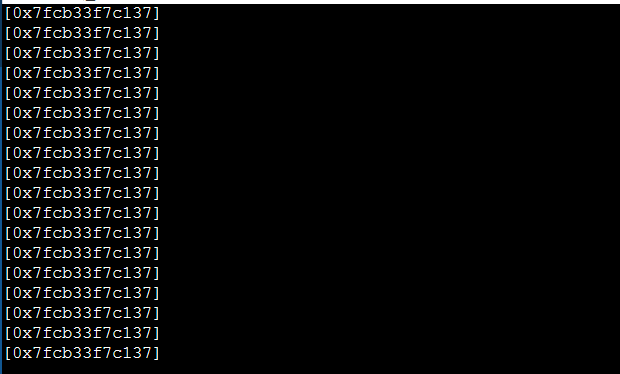
**Chú ý 2:** Ta thấy ta thay đổi giá trị biến fullName thông qua biến Name, và ta cũng có thể lấy giá trị biến fullName thông qua biến Name

**Chú ý 3:** Ta thấy biến name **không được phép** khởi tạo trong Constructor(Không được chứ không phải không cần khởi tạo). Điều đó càng chứng tỏ hơn biến Computed properties được sử dụng với mục đích chính là gán giá trị và lấy giá trị của 1 biến khác thông qua Computed properties đó.

**Chú ý 4:** Nếu ta code ngu như này, thì sẽ sinh ra vòng lặp vô tận:



Output:



**Giải thích:**

Ở dòng viet.age = 23, thì sẽ gọi tới computed set của biến age là self.age = self.age + newvalue. Lúc này ta lại thấy self.age lại đang được gán giá trị, nên nó lại gọi tiếp self.age = self.age + newvalue, cứ thế cứ thế nó sẽ rơi vào vòng lặp vô tận

Ở dòng print(viet.age), thì ta sẽ gọi tới computed get của biến age là return self.age. Lúc này ta lại thấy lúc này cũng là đang lấy giá trị của biến age, nên nó lại gọi tiếp self.age, cứ thế cứ thế vô hạn.

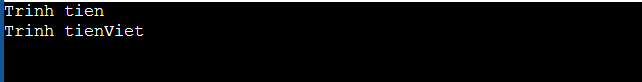
**Vì vậy, ta đel được phép return lại chính nó, hay gán giá trị cho chính nó. Mà phải return lại 1 biến khác và gán giá trị cho 1 biến khác**

## Các biến thể của Computed Properties

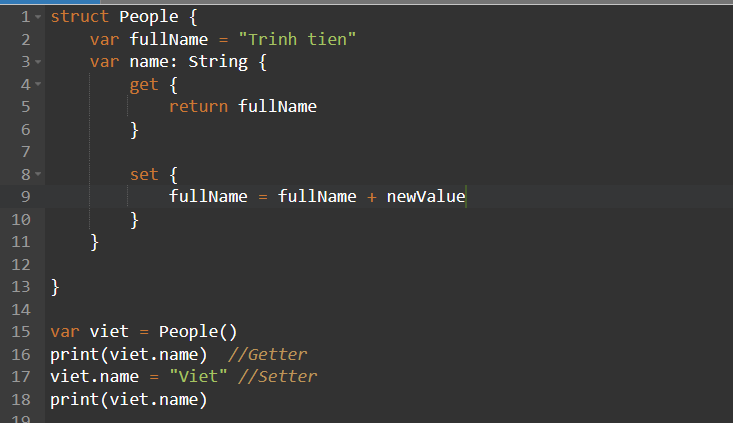
* Computed có cả get và set



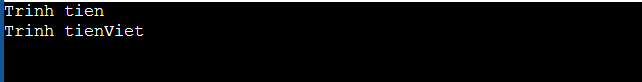
Output:



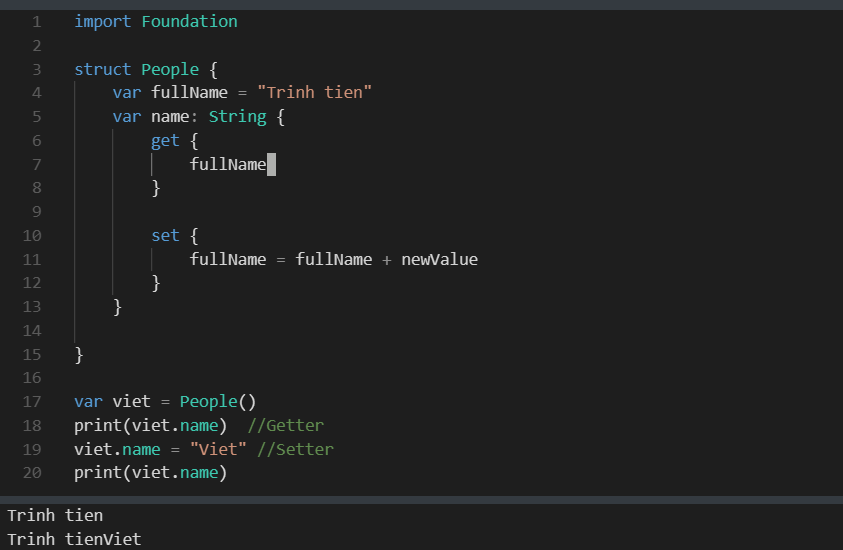
* Shorthand Setter: ta thấy ở ảnh bên trên, set ta truyền giá trị là my\_newvalue vào, nếu ta không truyền gì cả, thì default của biến đó sẽ là newValue, xem ví dụ:



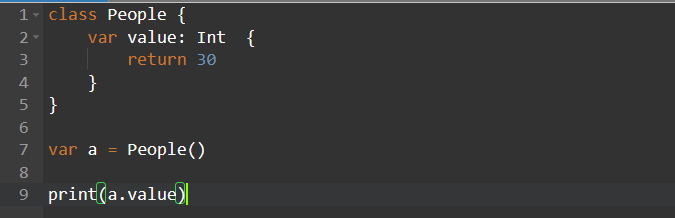
Output:



* Shorthand getter**: If the entire body of a getter is a single expression**, the getter implicitly returns that expression. Here’s an another version of the Rect structure that takes advantage of this shorthand notation and the shorthand notation for setters:



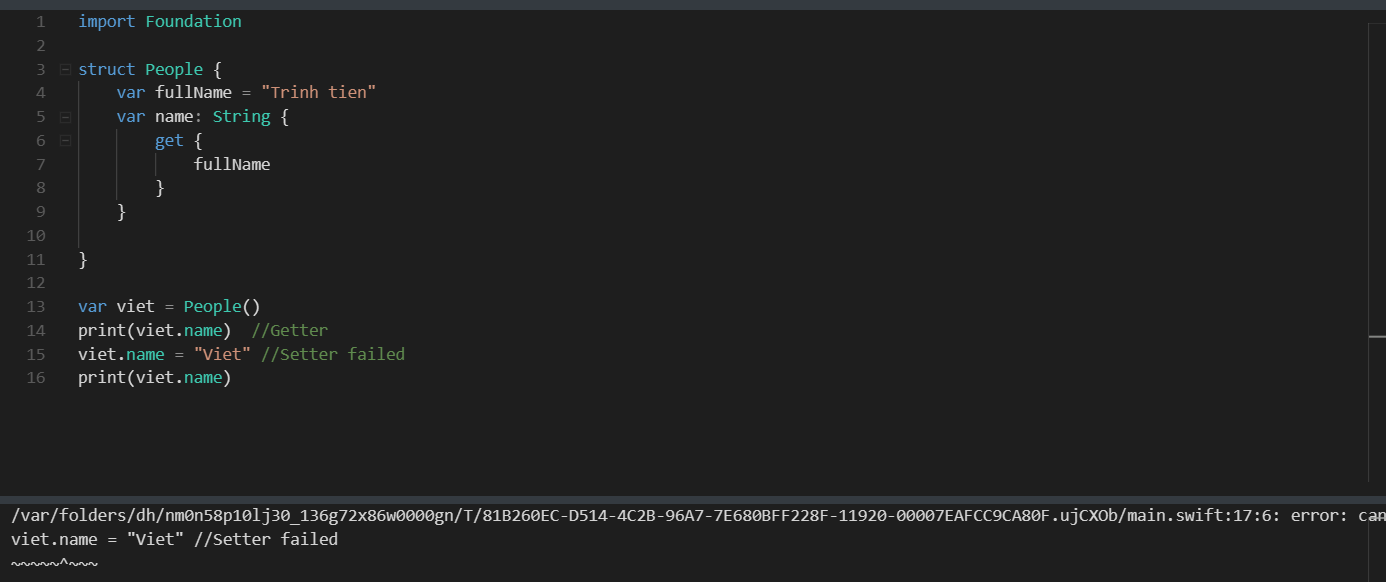
Cách 2:

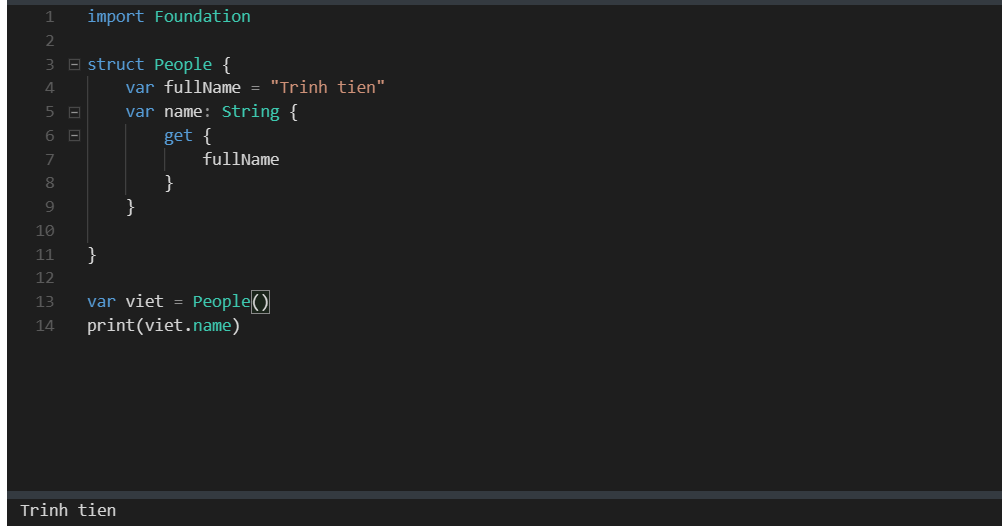


Output:



* Read Only Computed: Khi ta khai báo chỉ hàm get mà không có hàm set, thì biến đó sẽ read-only

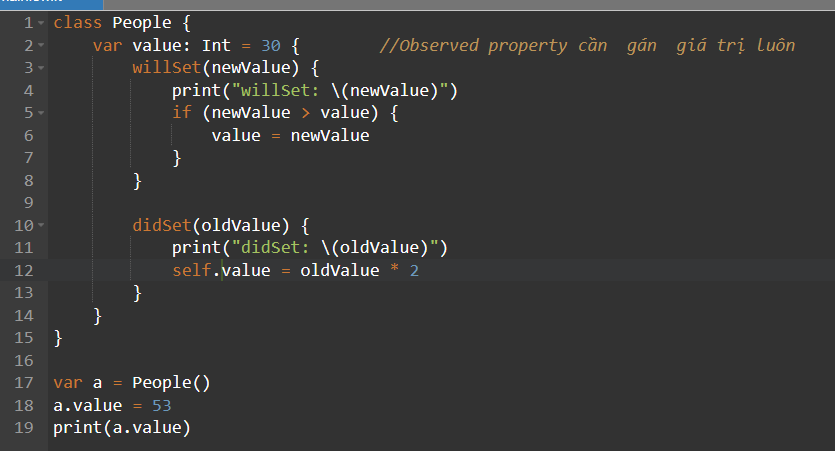




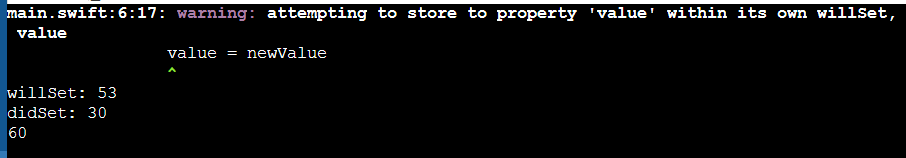
## 10.3 Property Observe(willSet and didSet)

ở willSet, ta có thể lấy giá trị nó sắp được gán thông qua para newValue

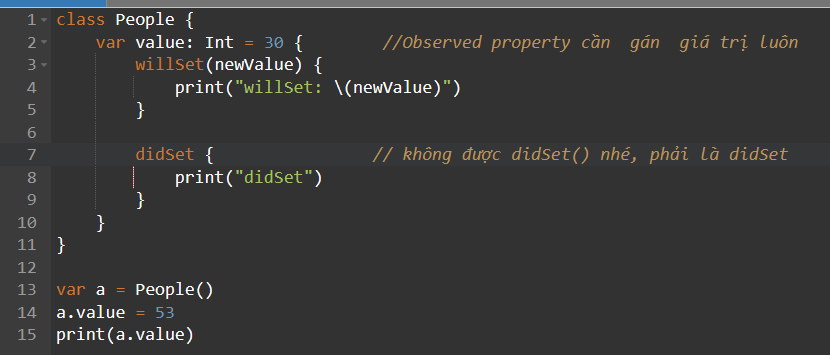
Ở didSet, ta có thể lấy giá trị của biến thông qua sử dụng para oldvalue



Output;

=>>>> Thế mà ko vào vòng lặp vô hạn nhỉ 😊)))

* Ta cũng có thể không truyền para vào các Observe Properties



Output:

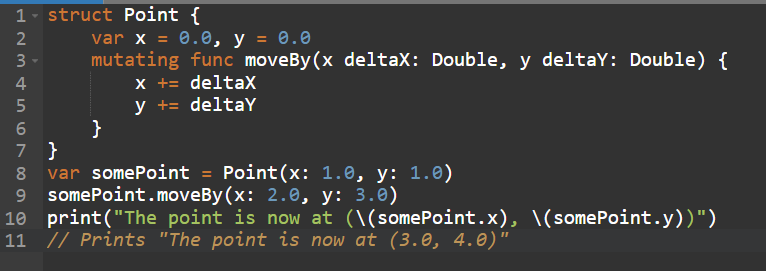


# 11. Methods (Bỏ type Method)

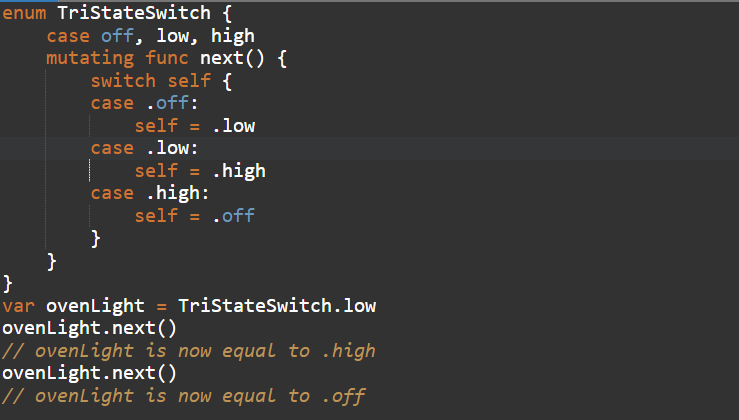
“Structures and enumerations are value types. By default, the properties of a value type can’t be modified from within its instance methods.”

Ngoài cách dung Computed Setter and getter như trên để thay đổi giá trị của của Properties, ta còn có thêm cách thứ 2 để thay đổi giá trị của properties với method, ta them từ khóa Structures and enumerations are value types. By default, the properties of a value type can’t be modified from within its instance methods.

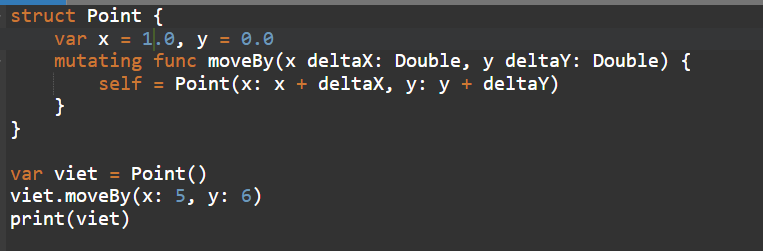
Với struct



Với Enum



Ta có thể dung biến self để lưu nhanh các properties trong struct



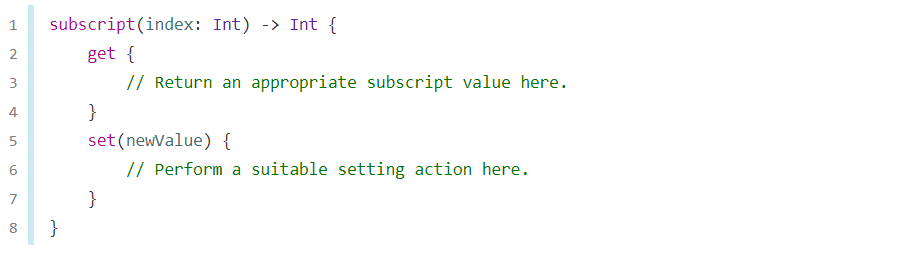
1. Type Method (tạm bỏ)

# 12. Subscripts

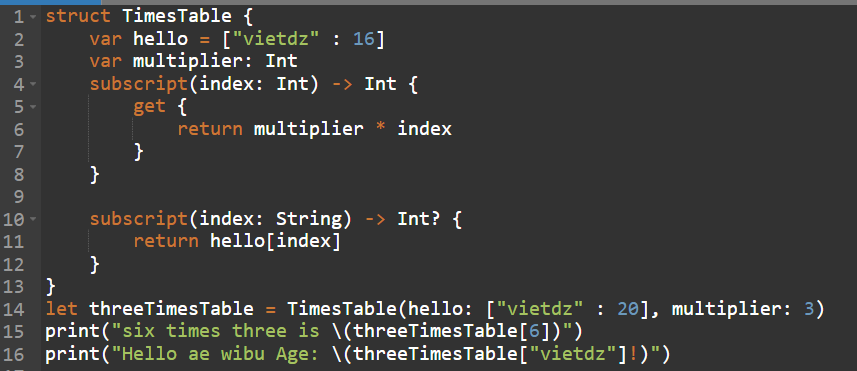
“Classes, structures, and enumerations can define subscripts, which are shortcuts for accessing the member elements of a collection, list, or sequence. You use subscripts to set and retrieve values by index without needing separate methods for setting and retrieval. For example, you access elements in an Array instance as someArray[index] and elements in a Dictionary instance as someDictionary[key].”

“You can define multiple subscripts for a single type, and the appropriate subscript overload to use is selected based on the type of index value you pass to the subscript. Subscripts aren’t limited to a single dimension, and you can define subscripts with multiple input parameters to suit your custom type’s needs.”

1. Subscript Syntax

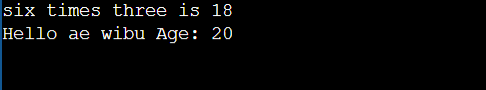


Ta có thể dung tên khác newValue. Ta có A default parameter called newValue is provided to your setter if you don’t provide one yourself. Ta thường ko dung set



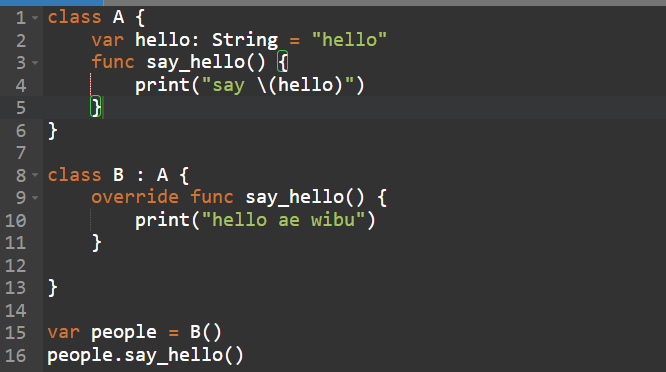
Ta thấy có nét tương đồng như mảng: arr[1] là lấy được giá trị tại vị trí thứ 2, giờ ta gọi threeTimesTable[6] thì ta cũng lấy được 1 giá trị nào đó.

Output:



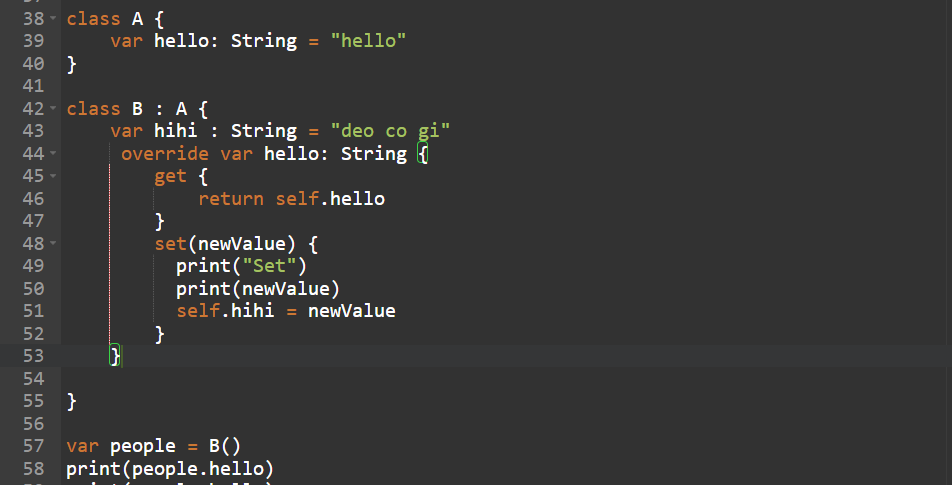
# 13. Inheritance

* Overiding: Để override một phương thức hoặc một property, ta them từ khóa override
  1. Override method:

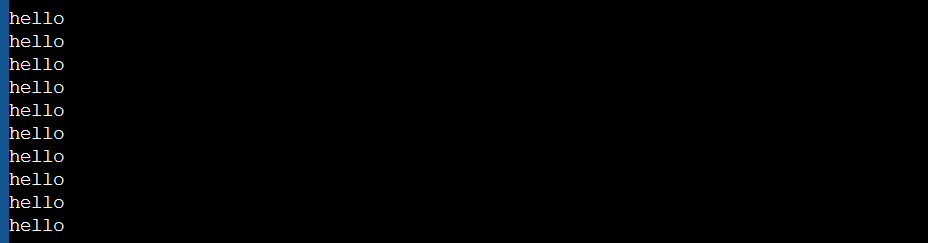


* 1. Override Property: Để override property, ta cũng thêm từ khóa override, việc override một propery giúp ta custom getter và setter instance cho property đó. Chú ý là khi ta overring 1 thuộc tính ta cần viết cả hàm get, set cho nó

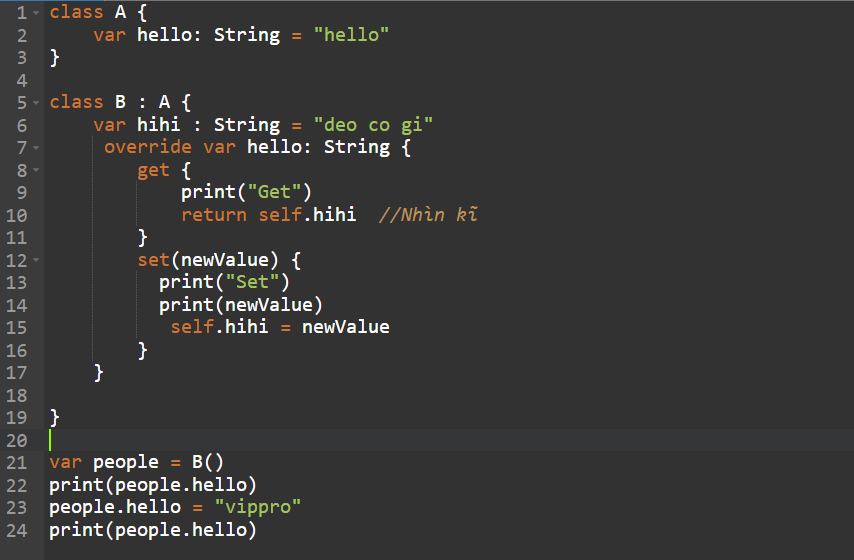
Chú ý 2:Khi ta viết gọi setter và getter mà return hoặc gán lại đúng với thuộc tính như thế thì sẽ vào vòng lặp vô hạn:



Output:



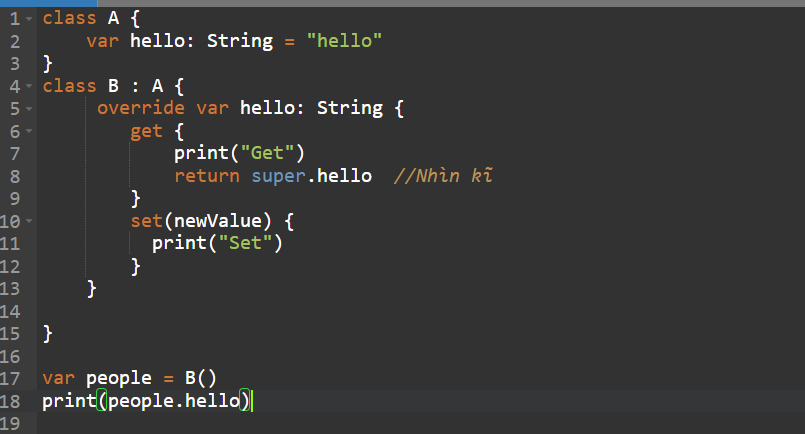
Nó sẽ in ra vô hạn chữ hello, vì nó sẽ call lại chính nó. Vì thế nếu để get hoặc set ta sẽ phải sử dụng 1 biến trung gian như sau:



Output:



* Ta dung them từ khóa super để lấy giá trị trước

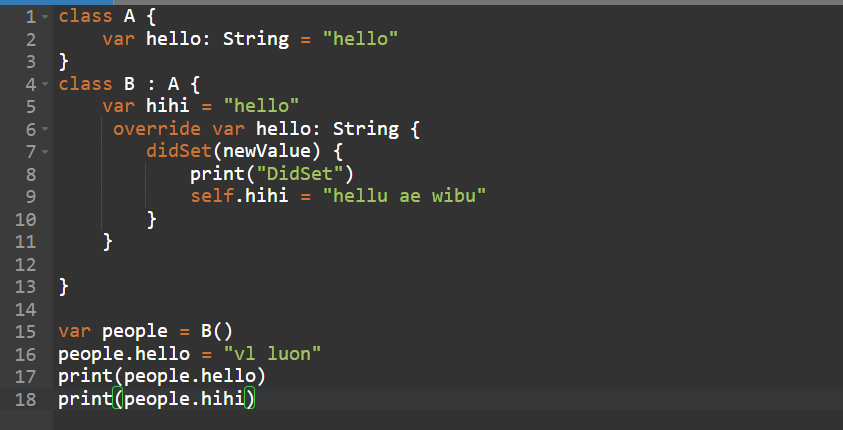


Kết quả:

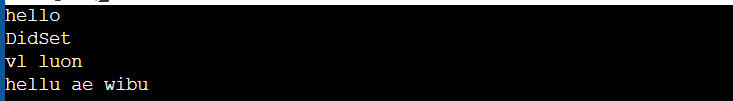


* “You can use property overriding to add property observers to an inherited property. This enables you to be notified when the value of an inherited property changes, regardless of how that property was originally implemented.”

“Note also that you can’t provide both an overriding setter and an overriding property observer for the same property. If you want to observe changes to a property’s value, and you are already providing a custom setter for that property, you can simply observe any value changes from within the custom setter.”



Output:

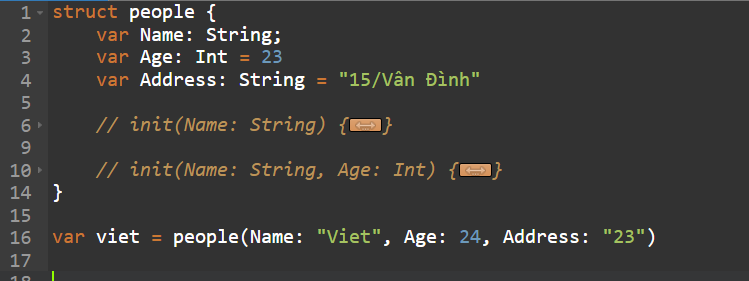


Ta thấy VD trên không có getter, vì khi dung didSet thì getter ko được xuất hiện và ngược lại.

# 14. Initialization

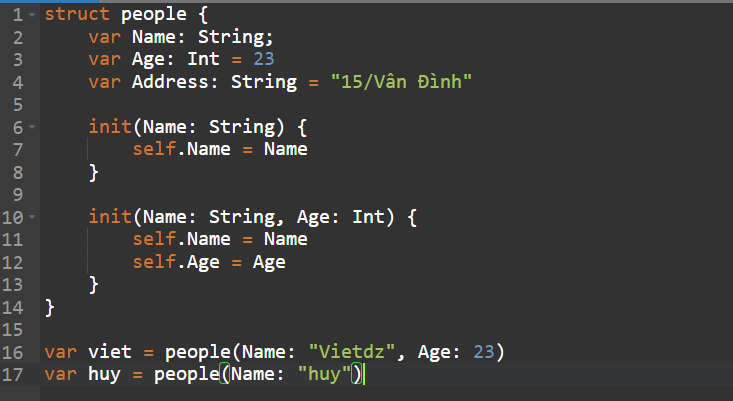
## 14.1 Struct

**Important 1 với Struct:** Khi ta không có hàm init(), ta có thể gọi hàm init() default ẩn với tất cả các thuộc tính được liệt kê

****

**Như này đúng.**

**Important 2 với Struct**: Khi ta Initialization 1 struct, làm thế nào thì làm nhưng bắt buộc các properties của nó phải được gắn giá trị. Các giá trị nào đã được gắn sẵn default thì có thể không cần gán giá trị nữa



Ta thấy sau việc khởi tạo object viet và huy thì tất cả các thuộc tính của nó đều được gán giá trị. Bên cạnh đó, khi ta khởi tạo init(), ta thấy là vì thuộc tính Age và Address đã được gán giá trị trước nên ta có thể them vào hàm init()

**Important 3 với Struct:** Khi ta tường minh hàm exit trong Struct, thì khi khởi tạo 1 object ta chỉ có thể khởi tạo bằng cách Init trong đó. Ta không thể gọi init() default với full giá trị



Như này sẽ bị **Error**

## 14.2 Class Inheritance and Initialization

***Importance 1 Class* : “**All of a class’s stored properties—including any properties the class inherits from its superclass—**must be assigned an initial value during initialization.**

**Swift defines two kinds of initializers for class types to help ensure all stored properties receive an initial value**. These are known as designated initializers and convenience initializers.**”**

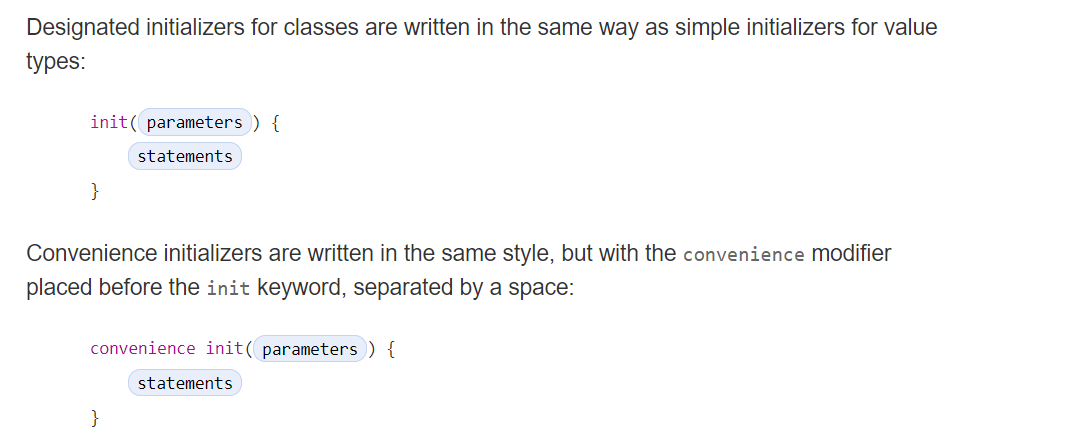
**Ta có: “**Swift defines two kinds of initializers for class types to help ensure all stored properties receive an initial value. These are known as **designated initializers and convenience initializers**.**”**

1. **Designated Initializers and Convenience Initializers**

**“Designated initializers** are the primary initializers for a class. A designated initializer **fully initializes all properties** introduced by that class and calls an appropriate superclass initializer to continue the initialization process up the superclass chain.”

**“Convenience initializers** are secondary, supporting initializers for a class. You can define a convenience initializer to call a designated initializer from the same class as the convenience initializer with some of the designated initializer’s parameters set to default values. You can also define a convenience initializer to create an instance of that class for a specific use case or input value type. You don’t have to provide convenience initializers if your class doesn’t require them. Create convenience initializers whenever a shortcut to a common initialization pattern will save time or make initialization of the class clearer in intent.”

* **Syntax for Designated and Convenience Initializers**

****

To simplify the relationships between **designated and convenience initializers**, Swift applies the following three rules for delegation calls between initializers:

**Rule 1**

A designated initializer must call a designated initializer from its immediate superclass.

**Rule 2**

A convenience initializer must call another initializer from the same class.

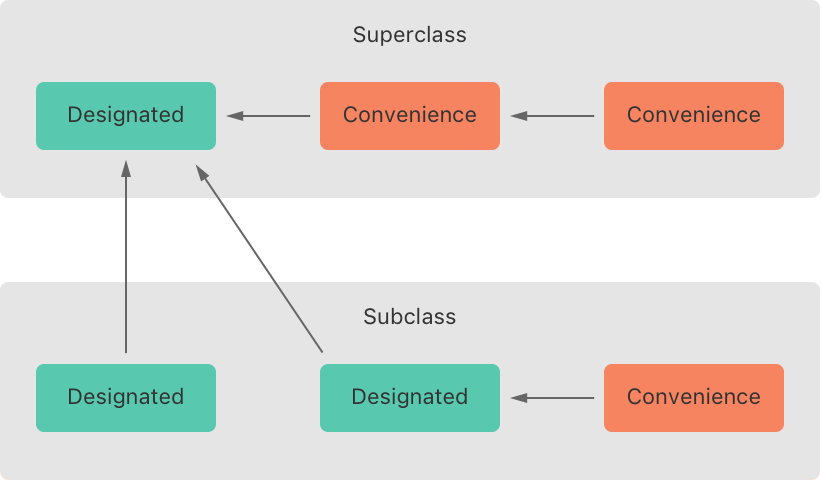
**Rule 3**

A convenience initializer must ultimately call a designated initializer.

A simple way to remember this is:

* Designated initializers must always delegate up.
* Convenience initializers must always delegate across.

These rules are illustrated in the figure below:

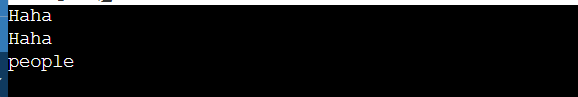


***Bốn Rule quan trọng:***

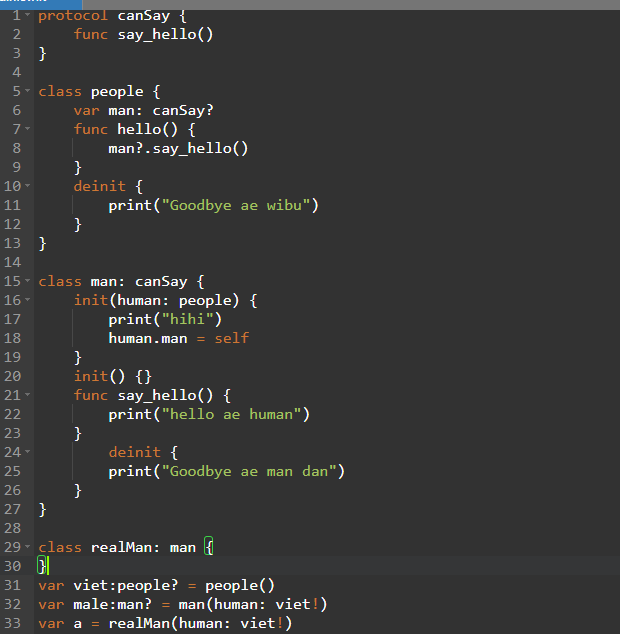
* ***Rule 1:*** Mọi designated init đều cần phải gán hết giá trị cho properties. Đel được để properties nào là nil hoặc chưa được gán giá trị
* ***Rule 2:*** Ở subclass, trong hàm init, **mọi properties của nó cũng đều phải được gán giá trị trước khi gọi tới hàm designated init của ở super class, lưu ý là các properties của subclass mà khai báo thêm thì mới cần được gán hết trước khi gọi super.init chứ ko phải là cả properties của Class Base**. Việc phải gọi tới designated Init ở super class, nó giống như C++, constructor kế thừa phải gới tới 1 constructor lớp cha
* ***Rule 3:*** Với convention init ở super class, ta cần gọi 1 hàm init() trong cùng class, sau đó ta mới được gán các properties của class.
* ***Rule 4:*** Trong lúc Initializer, ta không được phép read value hay sử dụng method nào của class, nếu sử dụng có thể dẫn đến crash chương trình(value của property có thể là nil)
* ***Rule 5: Cái này từ sau là đoán:***  *Khi class con không khởi tạo hàm init() nào thì có thể sử dụng init() của class cha. Một khi class con đã viết hàm init thì class con bắt buộc phải gọi tới 1 init của class cha thông qua super. Khi class con override class cha, thì vẫn phải gọi tới 1 init của class cha.*

******

***Output:***

******

***VD2:***

******

***Output:***

******

***Giải thích này:***

Rule 1: Ta thấy với bất kì hàm init nào, thì mọi properties của class man hay class people cũng đều được gán giá trị.

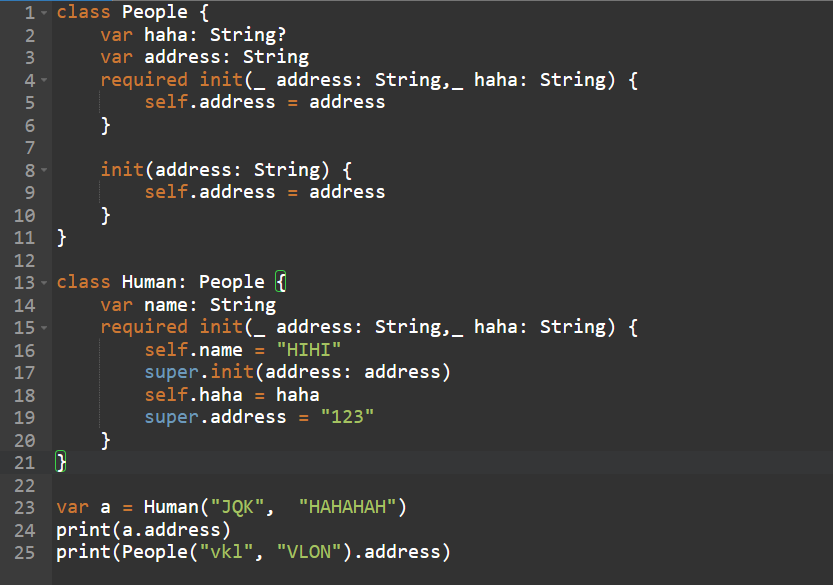
Rule 2: Ở subclass là class man, ta gán hết giá trị cho properties, ở đây là property male đã được gán là true. Sau khi gọi hàm init của superclass, bên dưới ta có thể tùy chỉnh giá trị properties theo ý mình(self.name = “HAHA”)

Rule 3: ở convenience init, ta gọi 1 hàm init() trong cùng class rồi mới tiến hành chỉnh sửa dữ liệu

Rule 4: Ta không hề gọi 1 method hay đọc value của 1 property nào cả

## 14.2 Required Init

Write the required modifier before the definition of a class initializer to indicate that every subclass of the class must implement that initializer:

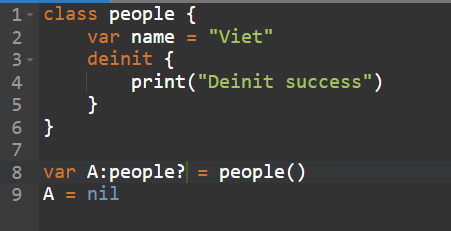


Output:

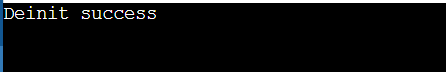


# 15. Deinitalization

Hmm cái này đọc bài tham chiếu strong và weak nhé 😊)

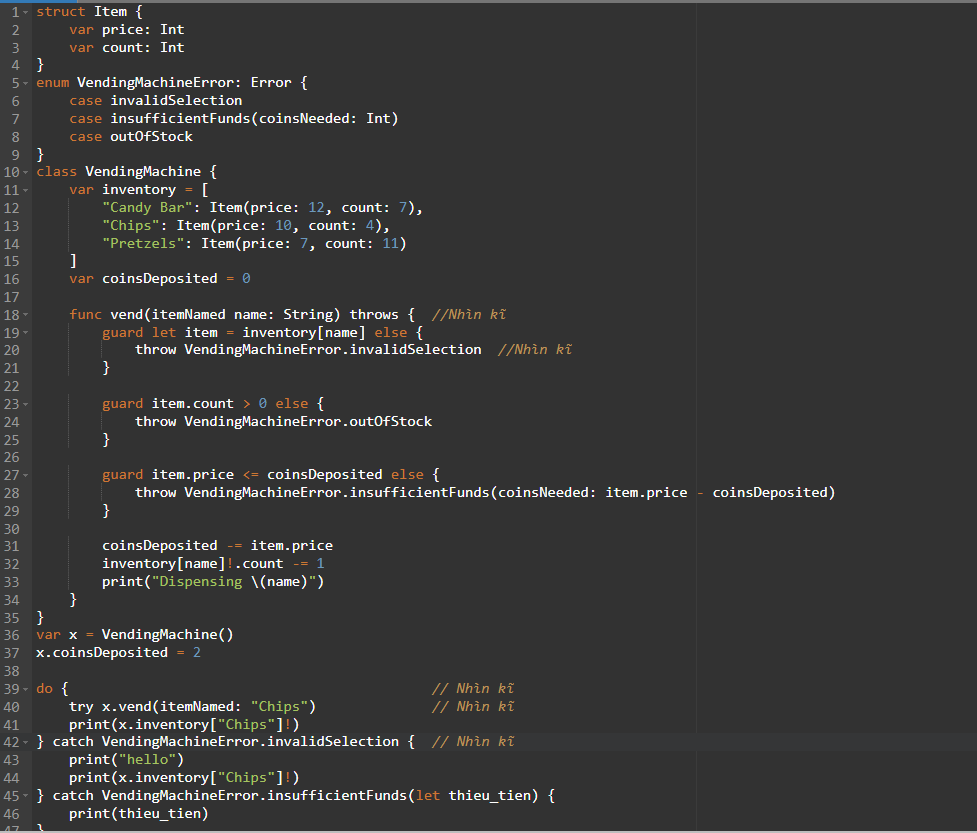


Output:



# 16. Error Handling

Ta sử dụng Error Handling để xử lý khi gặp tình huống nào đómà có thể dự đoán được, giả sử ta có hàm nau\_an(), nhưng ta đoán đôi khi là sẽ thiếu nguyên liệu, nên ta sẽ sử dụng try để thử nau\_an(), nếu thiếu nguyên liệu thì ta sẽ throw ra 1 lỗi để bên dưới có hàm sẽ catch được việc đó

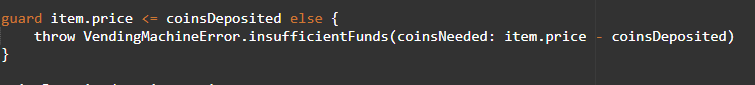


Ta có 1 chương trình bắt lỗi như sau:

Ở phần 1 :

Nếu name không trùng với key nào của list thì dẫn tới không tồn tại item và ta sẽ throw ra 1 lỗi VendingMachineError.invalidSelection, tương tự cho các trường hợp khác bên dưới.

Ở bên dưới khi ta tạo object x, và tiến hành chạy chương trình với do, try, catch, khi sinh ra lỗi nào thì catch sẽ hứng lỗi đó và sau đó chạy vào bên trong code.

Ta có **chú ý: catch VendingMachineError.insufficientFunds(let thieu\_tien)** thì lúc này variable thieu\_tien sẽ hứng giá trị trả về từ đây

Dẫn đến là ta được kết quả in ra là



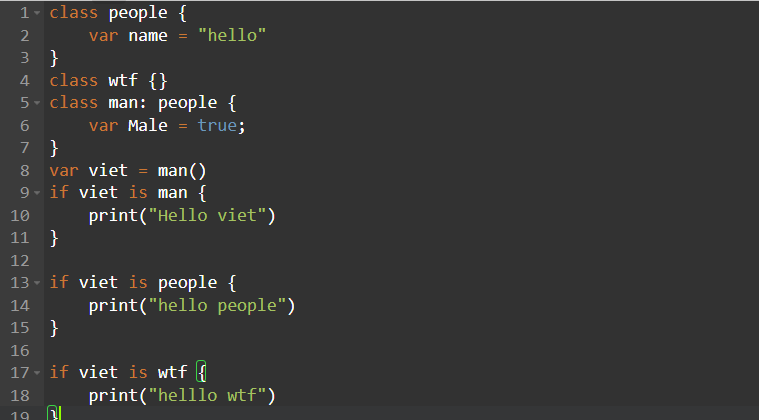
Chú ý 2: Ta thấy khi throw ra exception thì toàn bộ code bên dưới func vend() không được chạy nữa.

# 17. Type Casting

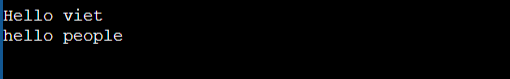
“Type casting is a way to check the type of an instance, or to treat that instance as a different superclass or subclass from somewhere else in its own class hierarchy.”

“Type casting in Swift is implemented with the is and as operators. These two operators provide a simple and expressive way to check the type of a value or cast a value to a different type.”

* Checking Type



Output:



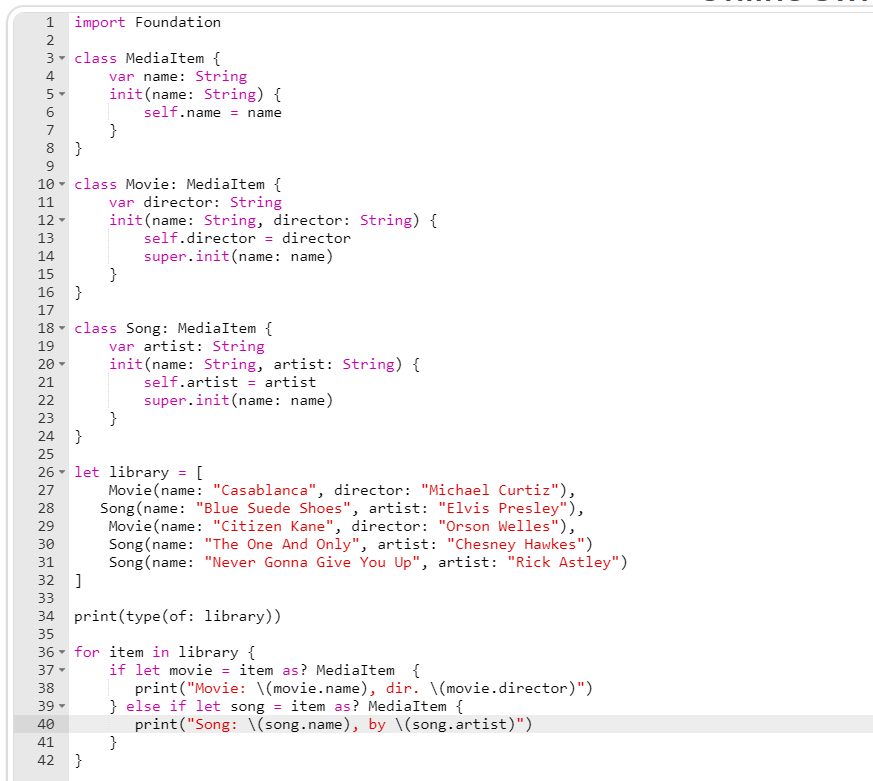
* DownCasting

“A constant or variable of a certain class type may actually refer to an instance of a subclass behind the scenes. Where you believe this is the case, you can try to downcast to the subclass type with a type cast operator (as? or as!).”

“**Because downcasting can fail, the type cast operator comes in two different forms.** The conditional form, as?, returns an optional value of the type you are trying to downcast to. The forced form, as!, attempts the downcast and force-unwraps the result as a single compound action.”

“Use the conditional form of the type cast operator (as?) when you aren’t sure if the downcast will succeed. This form of the operator will always return an optional value, and the value will be nil if the downcast was not possible. This enables you to check for a successful downcast.

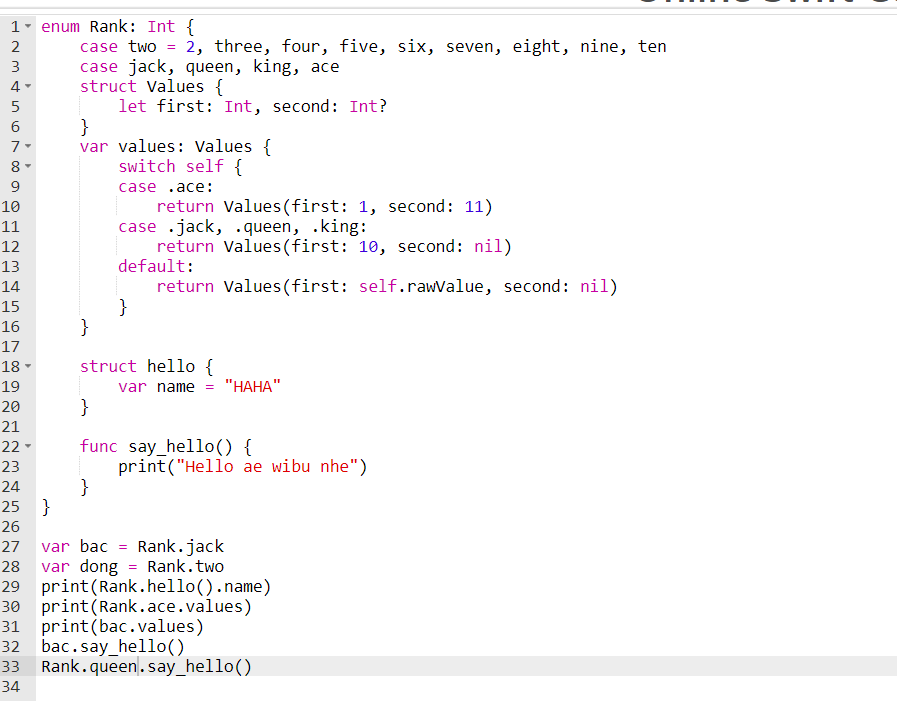
Use the forced form of the type cast operator (as!) only when you are sure that the downcast will always succeed. This form of the operator will trigger a runtime error if you try to downcast to an incorrect class type.”



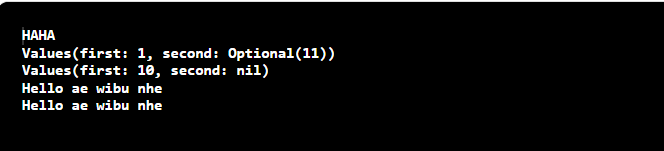
The example starts by trying to downcast the current item as a Movie. Because item is a MediaItem instance, it’s possible **that it might be a Movie; equally, it’s also possible that it might be a Song, or even just a base MediaItem. Because of this uncertainty,** the as? form of the type cast operator returns an optional value when attempting to downcast to a subclass type. The result of item as? Movie is of type Movie?, or “optional Movie”.

# 18. Nested Type

Swift cho phép ta lồng nhiều kiểu dữ liệu vào nhau, ta xem xét ví dụ dưới đây:

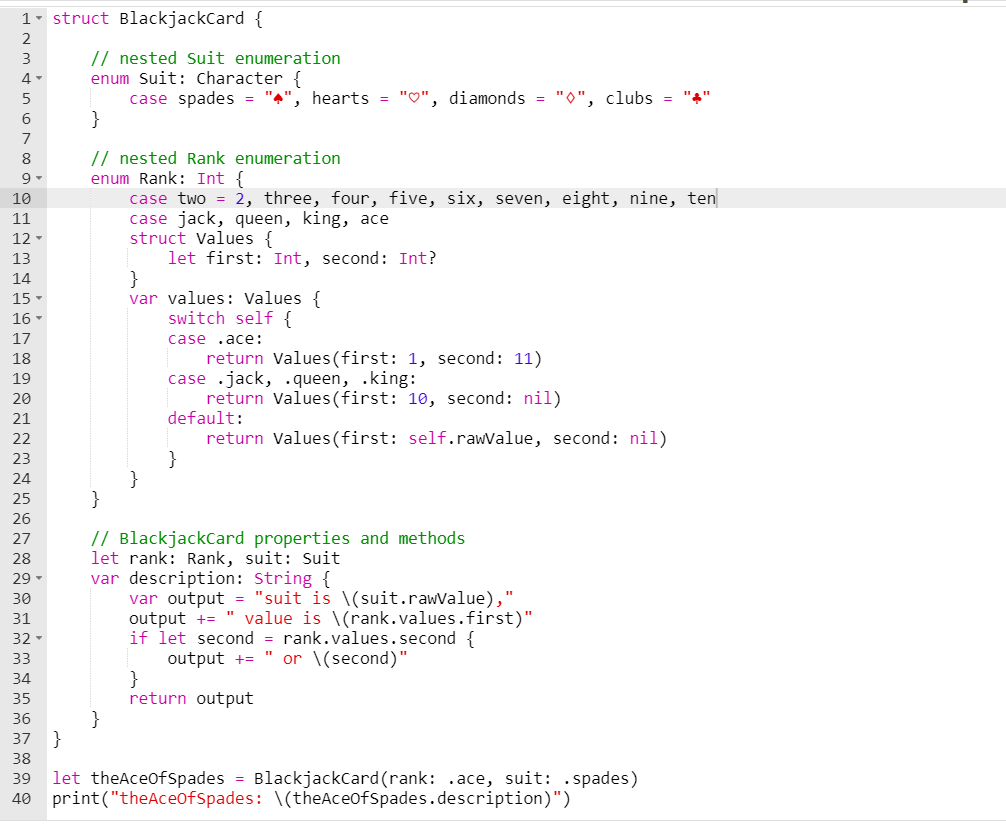


Output:



Ta thấy trong Enum, ta vừa có thể khai báo variable, vừa khai báo struct, vừa khai báo func(). Ta cũng có thể nhận thấy thêm, **chỉ các object mới có quyền gọi các variable và function trong enum**, vì enum không có hàm khởi tạo init(), nên sẽ không gọi được kiểu Rank().jack.values.

Ta xem xét ví dụ nâng cao hơn:



Trong struct ta khai báo 2 enum, 1 struct. Ta có biến object let rank thuộc kiểu enum Rank, nên nó có quyền .values

# 19. Extensions

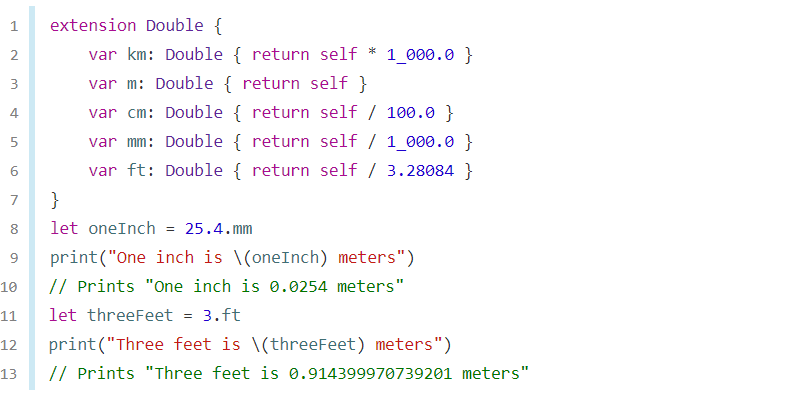
“Extensions **add new functionality** to an existing class, structure, enumeration, or protocol type. This includes the ability to extend types for which you don’t have access to the original source code (known as retroactive modeling).”

Extensions in Swift can:

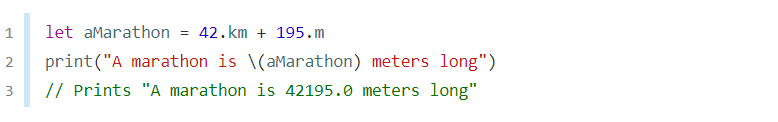
* Add computed instance properties and computed type properties
* Define instance methods and type methods
* Provide new initializers
* Define subscripts
* Define and use new nested types
* Make an existing type conform to a protocol

## 19.1. Computed Properties

“Extensions can add computed instance properties and computed type properties to existing types”



VD2:

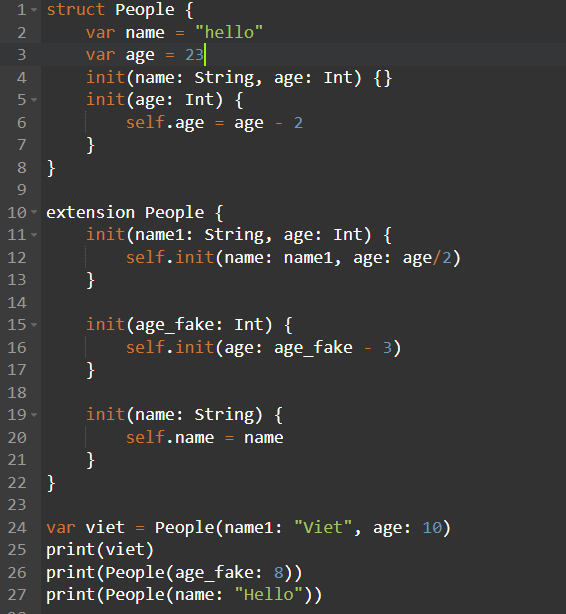


## 19.2. Initializers

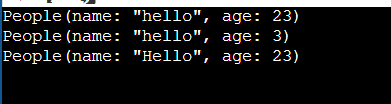
“Extensions can add new initializers to existing types. This enables you to extend other types to accept your own custom types as initializer parameters, or to provide additional initialization options that were not included as part of the type’s original implementation.”

“Extensions can add new convenience initializers to a class, **but they can’t add new designated initializers or deinitializers to a class**. Designated initializers and deinitializers must always be provided by the original class implementation.”

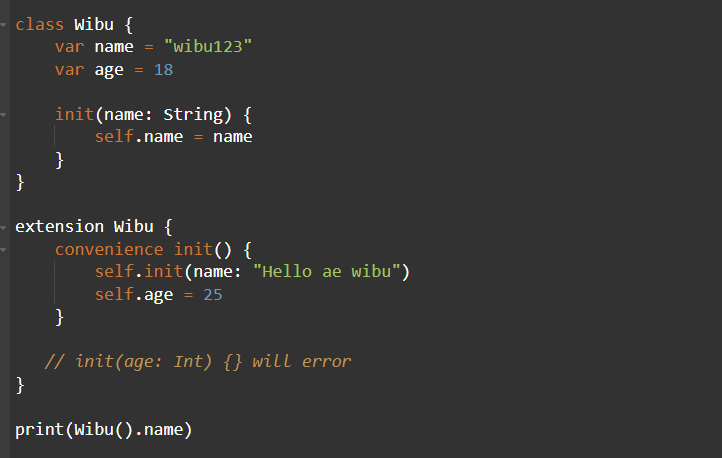
* Struct: Với Struct ta có thể thêm các hàm init() thông quá extension



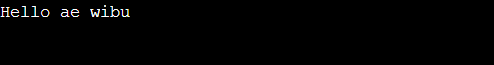
Output:



* Với class: ta chỉ có thể add thêm các convinence init() chứ không được add desinated init cho class

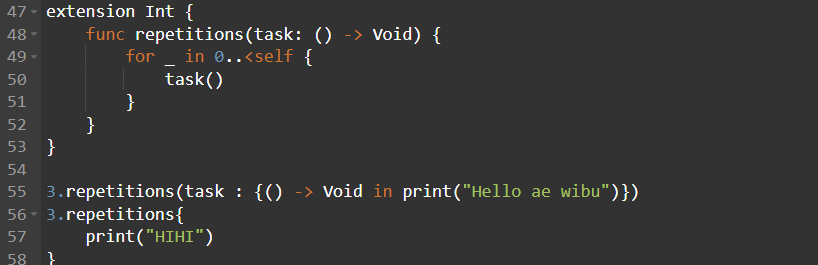


Output:

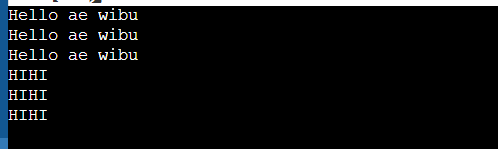


## 19.3. Methods

“Extensions can add new instance methods and type methods to existing types. The following example adds a new instance method called repetitions to the Int type:”



Output

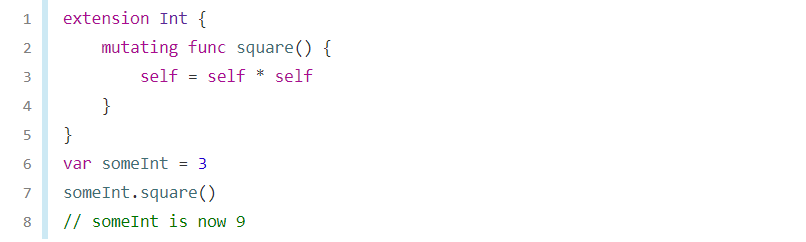


Ta đã add them method repetitions vào kiểu dữ liệu Int

## 19.4. Mutating Instances

“Instance methods added with an extension can also modify (or mutate) the instance itself. Structure and enumeration methods that modify self or its properties must mark the instance method as mutating, just like mutating methods from an original implementation.”

* Với struct



* Với class

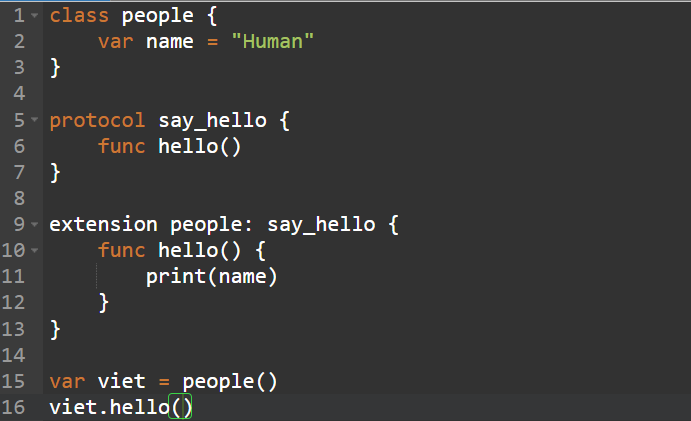


Output:

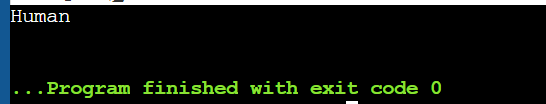


## 19.5 Adding Protocol with an Extension

Ta thậm chí là không cần triển khải các function của protocol trong source của class hay struct mà adopt protocol đó. Ta có thể triển khải bên ngoài class thông qua extension



Output:



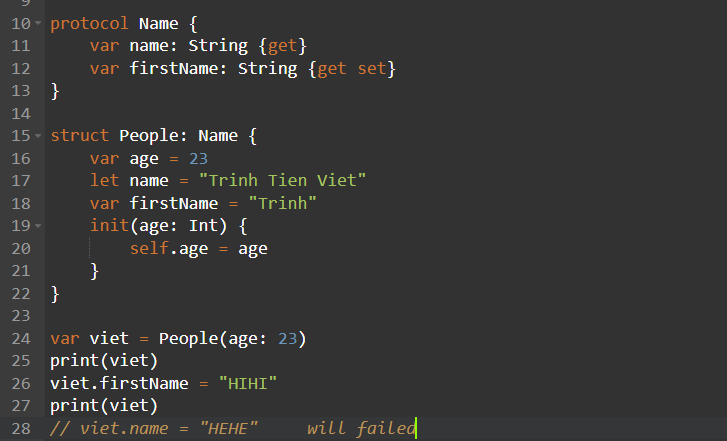
# 20. Protocol

“**A protocol defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality**. The protocol can then be adopted by a class, structure, or enumeration to provide an actual implementation of those requirements. Any type that satisfies the requirements of a protocol is said to conform to that protocol.”

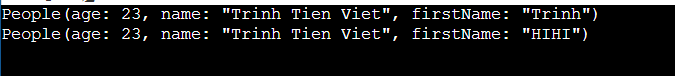
Giải thích khái niệm trên: Protocol nó chỉ define các **methods, properties, and other requirements** và nó hoàn toàn không triển khai các đắc điểm trên. Một class, struc hay enum adopted(Nghĩa là nhận nuôi: ý là đồng ý nhận nuôi các define của Protocol, sau đó triển khai các define đó).

## 20.1 Property Requirements

“Property requirements are always declared as variable properties**, prefixed with the var keyword.** Gettable and settable properties are indicated by writing { get set } after their type declaration, and gettable properties are indicated by writing { get }. “



Output:

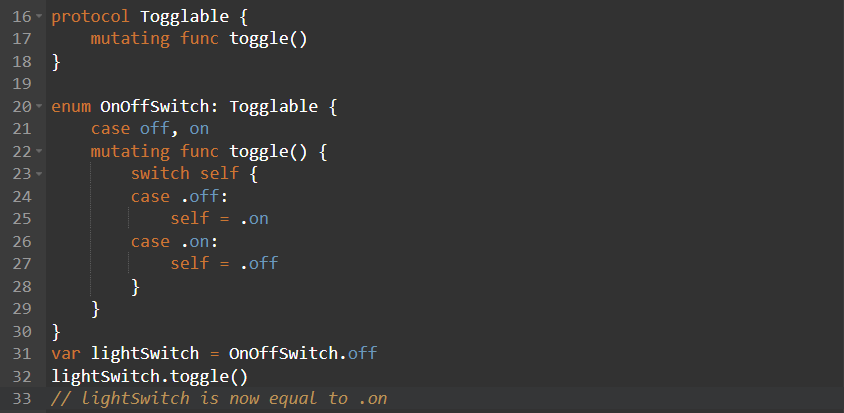


## 20.2 Method Requirement



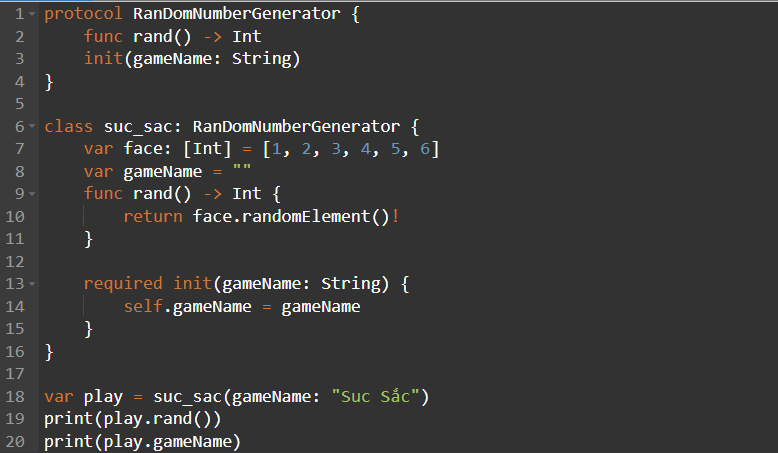
## 20.3 Mutating Protocol

Được sử dụng để thay đổi giá trị properties của enum và struct



## 20.4 Initializer Requirements

Trong hàm triển khai ta cần them từ khóa require



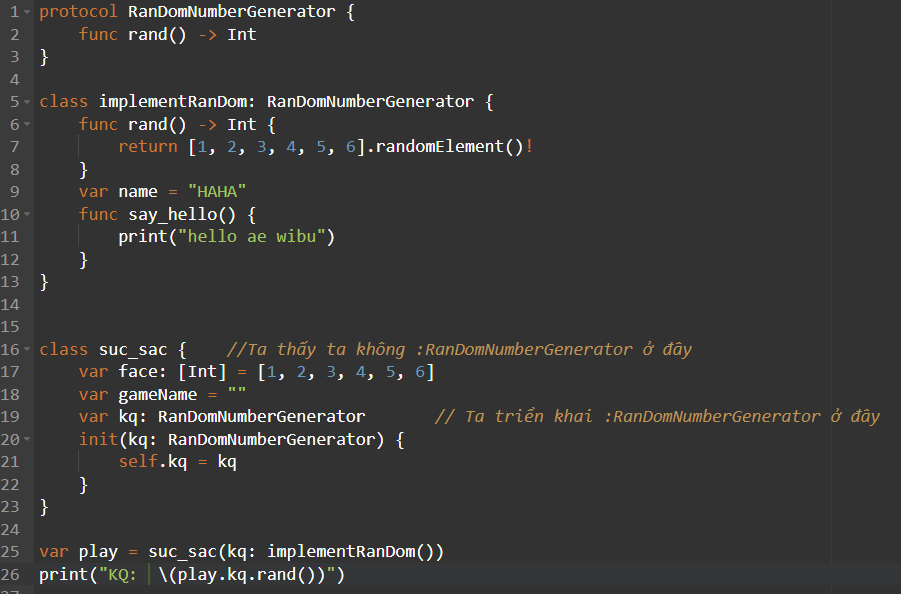
Output:



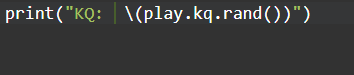
## 20.5 Protocol as Type

“Protocols don’t actually implement any functionality themselves. Nonetheless, you can use protocols as a fully fledged types in your code. Using a protocol as a type is sometimes called an existential type, which comes from the phrase “there exists a type T such that T conforms to the protocol”.”

Ta sử dụng protocol như 1 type:



Kết quả;



Giải thích: Khi ta sử dụng protocol như 1 type, ta có thể sử dụng các method, properties của protocol đó thông qua 1 class khác. Các thuộc tính còn lại của class mà không liên qua tới protocol thì biến type đó không được gọi.

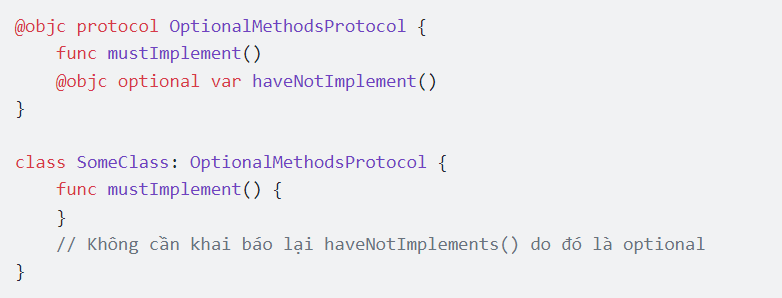
Ta thấy ở ví dụ trên, ta dung class implement để triển khai cho class suc\_sac không qua biến var kq. Tổng quát là ta sử dụng 1 class khác triển khai cho 1 class mong muốn. Như vậy gọi là Delegation. Hoạt động bằng cách chuyển chức năng từ một class cơ sở sang một delegate class khác. Class cơ sở này thường nằm ngoài sự kiểm soát của nhà phát triển, nhưng bằng cách sử dụng một delegate, bạn vẫn có thể ảnh hưởng đến chức năng của nó

## 20.6 Optional Protocol

*Có 1 câu hỏi: “Protocol có 100 function được khai báo. Và chúng ta khi implement thì phải định nghĩa lại 100 function đó hay sao? Trong khi nhiều function lại không cần dùng tới.”*

Cách 1: [@objc](https://viblo.asia/u/objc) optional

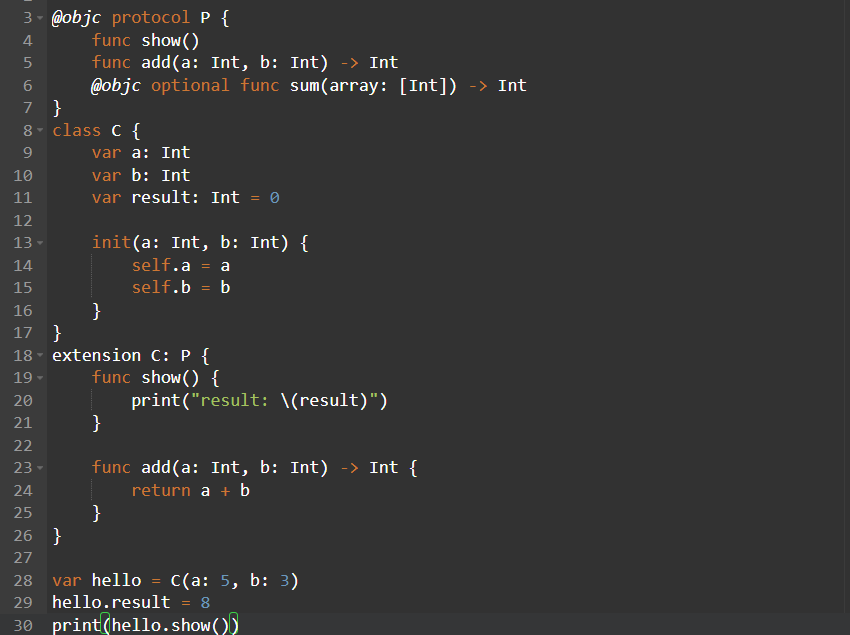
* **Bước 1:** thêm từ khoá @objc để khai báo protocol đó xài được với code Objective-C
* **Bước 2:** Thêm từ khoá optional trước function nào mà bạn mong muốn là không cần định nghĩa lại thì vẫn được.



Cách 2: Sử dụng Extension 🡪 Đel khác cách trên lắm nma ở đây dùng extension 😊))

Để sử dụng Protocol một cách chuyên nghiệp hơn thì lời khuyên của mình dành cho bạn là: “nên tạo extension cho class/struct/enum đó và implement protocol.”

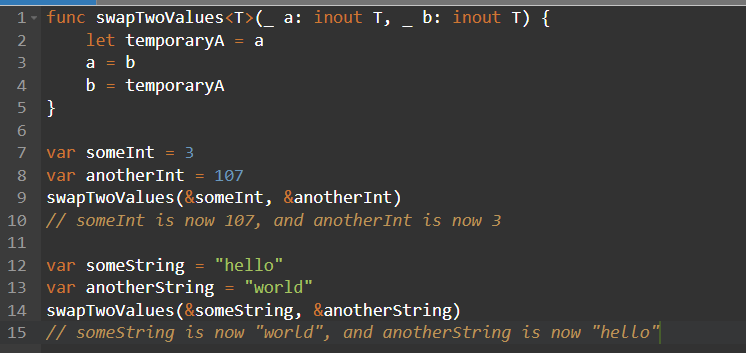
Vì 1 class có thể implement rất nhiều protocol và chúng nó có thể trùng tên với nhau hoặc bạn sẽ không phân biệt được function nào của protocol nào.



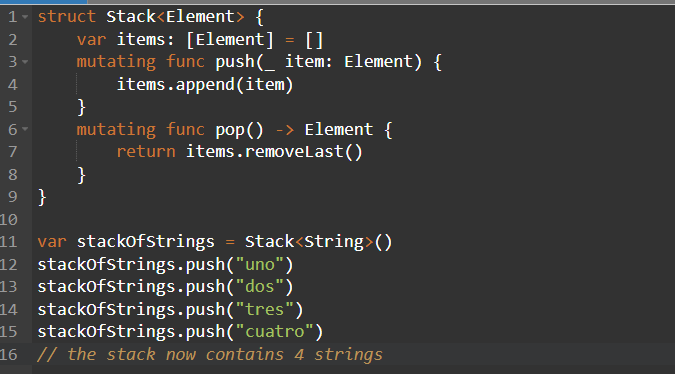
Output:



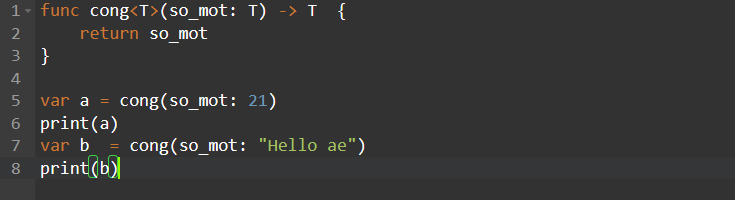
# 21. Generic



VD2:



VD3:



Output:



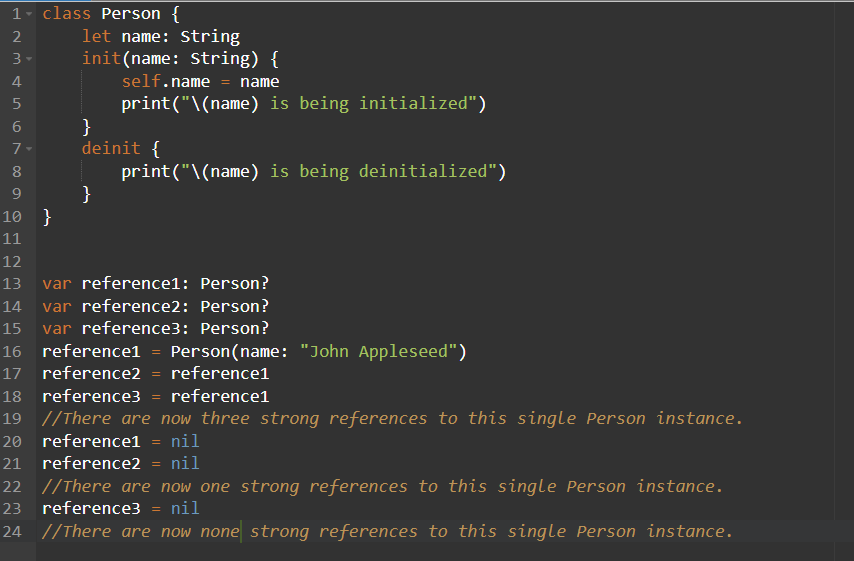
Ta thấy khi ta truyền dữ liệu gì vào func cong() thì T sẽ là kiểu dữ liệu đó.

# 22 Automatic reference counting

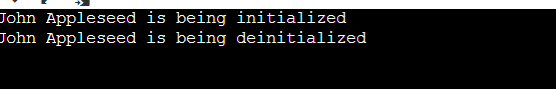
“Every time you create a new instance of a class, ARC allocates a chunk of memory to store information about that instance. This memory holds information about the type of the instance, together with the values of any stored properties associated with that instance.

Additionally, when an instance is no longer needed, ARC frees up the memory used by that instance so that the memory can be used for other purposes instead. This ensures that class instances don’t take up space in memory when they’re no longer needed.

However, if ARC were to deallocate an instance that was still in use, it would no longer be possible to access that instance’s properties, or call that instance’s methods. Indeed, if you tried to access the instance, your app would most likely crash.”

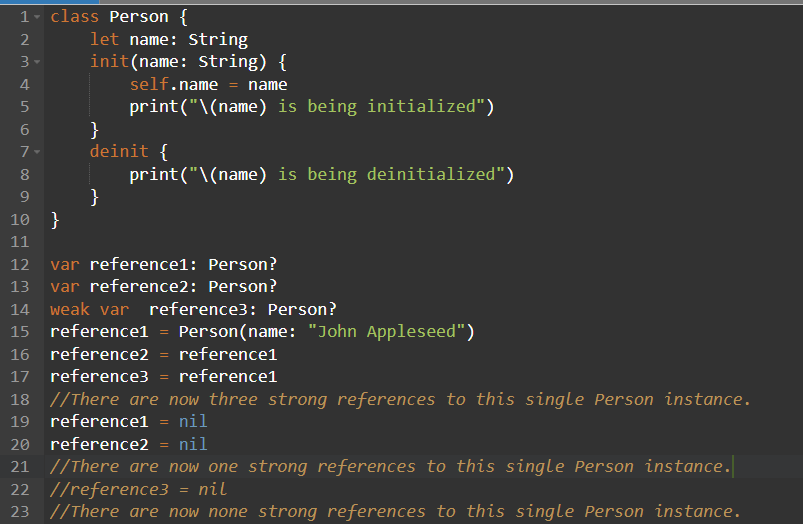


Output:

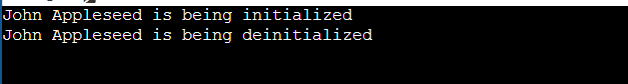


Ta thấy khi nào không có tham chiếu nào đến Person() nữa thì hàm deinit() mới được gọi, lúc đó vùng nhớ Person() chính thức bị ARC thu hồi.

## 22.1 Strong và weak

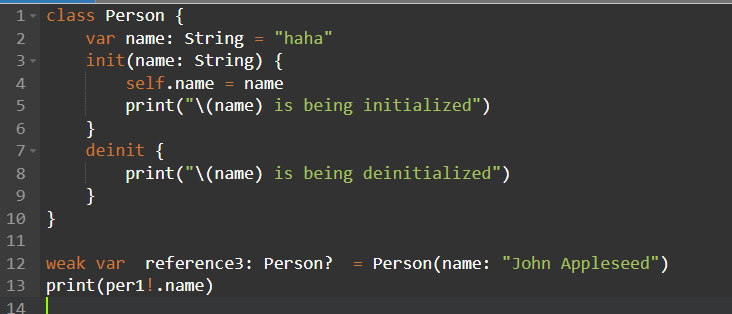


Output:

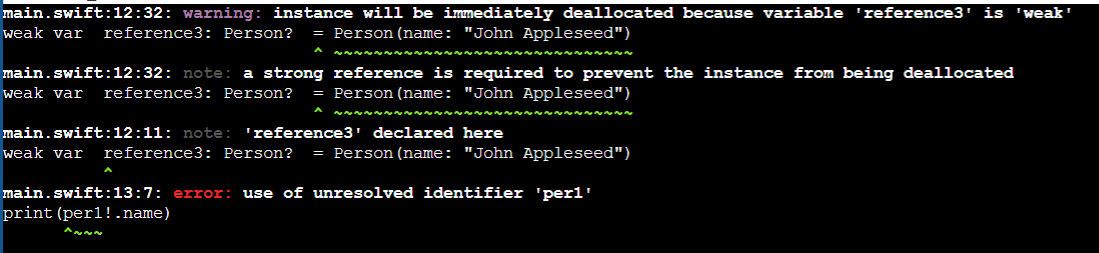


Ta thấy biến weak var referencen3 không cần gán = nil thì vùng nhớ Person() đã bị thu hồi rồi. Điều này muốn nói lên là khi khai báo với từ khóa weak thì kiểu refenrence này là kiểu tham chiếu yếu 😊))) Nghĩa là khi không còn thằng Strong nào tham chiếu đến vùng nhớ đó nữa thì vùng nhớ đó sẽ tự bị thu hồi luôn.

VD2:



Output:



Vùng Person() lập tức bị thu hồi luôn vì kiểu tham chiếu của reference3 là kiểu tham chiếu yếu.

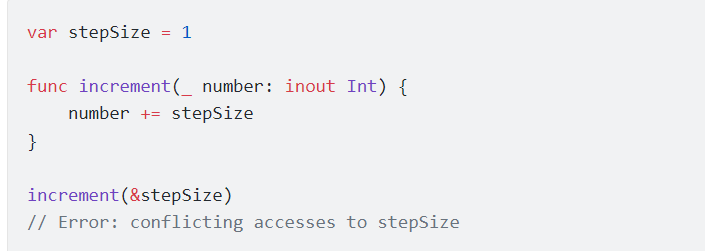
# 23. Safety Memory

Ta nhớ trong nhiều thread, nếu ta thay đổi 1 biến ở trong 1 thread trong khi có 1 thread khác đang đọc nó, dẫn tơi sai lệch dữ liệu. Nhưng với chỉ 1 thread, ta vẫn có thể code ngu dẫn tới xung đột bộ nhớ như vậy. Lỗi này thường xảy ra khi ta sử dụng hàm in out

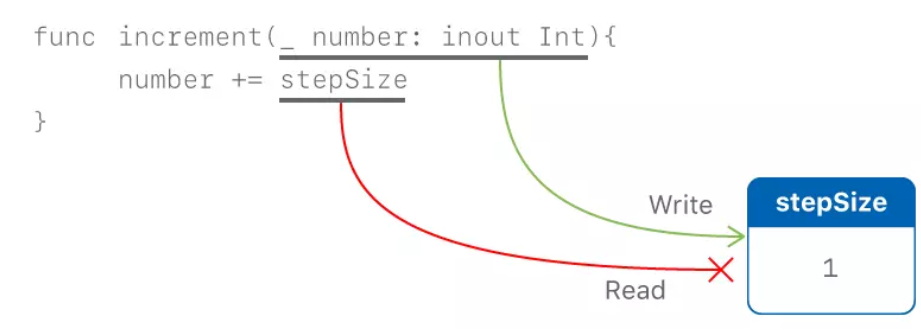
Việc xung đột khi truy cập bộ nhớ sẽ xảy ra khi bạn có hai luồng truy cập khác nhau đảm bảo toàn bộ các điều kiện sau:

* Ít nhất một trong hai luồng thực hiện ghi dữ liệu hoặc là nonatomic access
* Chúng truy cập tới cùng 1 vị trí trong bộ nhớ
* Chúng tiền hành truy cập tại cùng 1 thời điểm

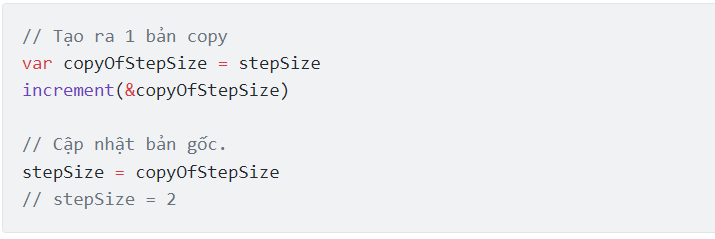
**Ta có ví dụ1:**



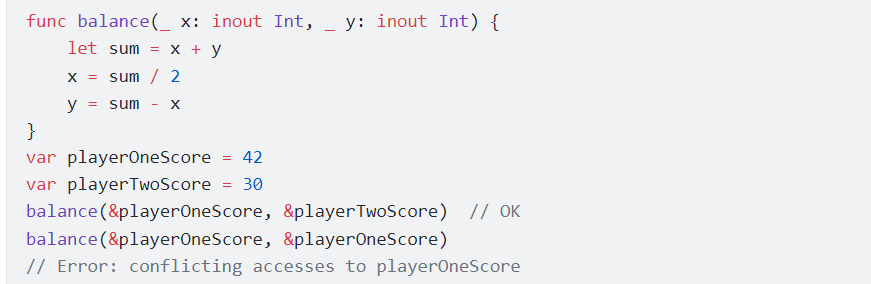
Ta thấy là tại vùng nhớ stepSize, ta đang vừa access read, và vừa access write trên nó nên dẫn tới xung đột.



Ta có thể sửa bằng cách sau:



**Ta có VD2:**

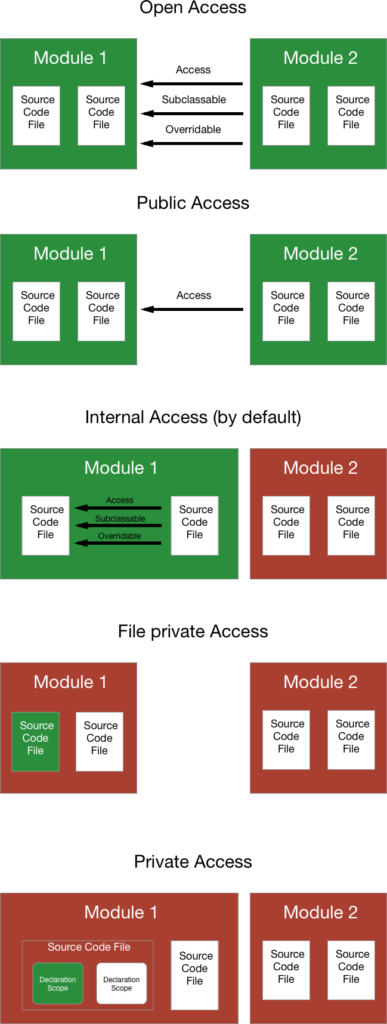


Dễ thấy ở trường hợp đầu tiên khi chúng ta truyền hai giá biến nhau vào hàm **balance()** thì không có lỗi gì xảy ra cả vì mặc dù hai biến này được truy cập đồng thời nhưng chúng lại nằm ở 2 địa chỉ khác nhau trong bộ nhớ. Còn ở trường hợp thứ hai, khi chúng ta truyền **playerOneScore** vào cả 2 tham số có của hàm thì xung đột đã xảy ra vì chúng ta đang thực hiện 2 **write access** vào cùng 1 địa chỉ.

# 24. Access Control

Trong Swift cung cấp cho ta 5 mức truy cập(Access level). Mức truy cập này liên quan tới source file và module.

* **Open Acess và Public Access:** cho phép mọi thực thể được sử dụng ở mọi source file trong module, và trong mọi source file ở module khác mà import module đó. Open Acess thì cho phép module khác có thể kế thừa, hoặc override mọi thuộc tính ở module khác, trong khi Public chỉ có quyền truy cập.
* **Internal Access:** Cho phép mọi thực thể được truy cập ở mọi source file trong module của chính nó, các module bên ngoài ko thể truy cập được(kể cả việc import module đó).
* **File Private:** Sẽ giới hạn thực thể chỉ có thể được truy cập ở bên trong Source file đó.
* **Private Access:** Sẽ giới hạn quyền truy cập thực thể ở trong nơi nó khai báo, và các **extension mà khai báo ở trong source file nó khai báo.**



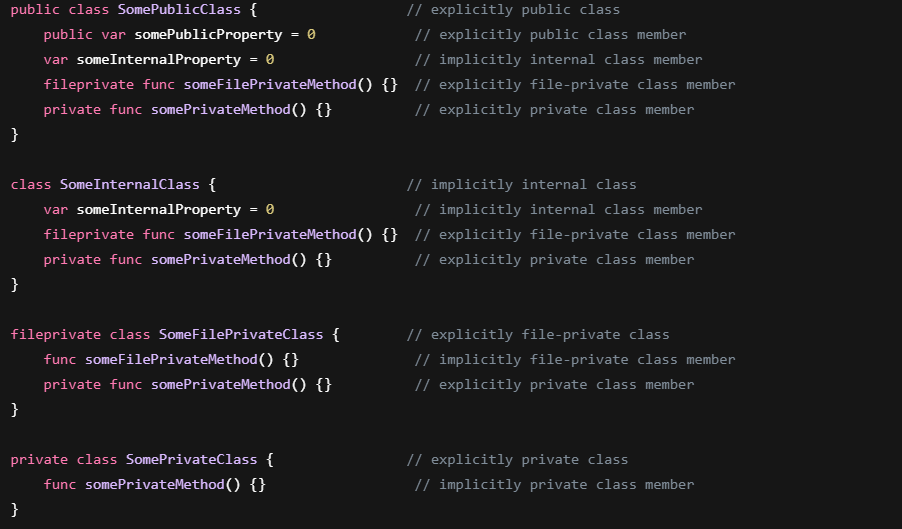
Swift đưa ta 1 nguyên tắc như sau: **“No entity can be defined in terms of another entity that has a lower (more restrictive) access level.”** Có nghĩa là trong 1 class nếu ta khai **báo 1 func public thì các kiểu dữ liệu parameter truyền vào hoặc kiểu giá trị return sẽ không thể có quyền truy cập thấp hơn như là: Internal, Private**,… Bởi vì **private và internal chỉ cho phép truy cập trong module đó, mà public cho phép các module bên ngoài có quyền truy cập vào, nên sẽ gây xung đột.**

* Default Access Control là Internal

## 24.1 Custom Types Access Control

The access control level of a type also affects the default access level of that type’s members (its properties, methods, initializers, and subscripts). If you define a type’s access level as private or file private, the default access level of its members will also be private or file private. If you define a type’s access level as internal or public (or use the default access level of internal without specifying an access level explicitly), the default access level of the type’s members will be internal**. (Nghĩa là khi ta định nghĩa Access Control cho Class thế nào, thì default Access Control cho các properties or method sẽ giống như thế, trừ khi ta explicit Access Control như dưới).**

Tuy nhiên **Public Type** sẽ có **internal member, chứ không phải Public members.**

****

# 25. Array

**Chú ý:** Các Method của 1 mảng mà duyệt qua các phần tử bên trong nó, sau đó return lại 1 Array mới thì đều có tham số là 1 closure gồm các parameters truyền vào và return lại giá trị (true/false hoặc value tùy thuộc hàm). Dưới đây là dạng đầy đủ nhất của 1 closure.



Thông thường trong thực tế, có nhiều cách viết tắt như đôi khi ta không thấy từ khóa **return hay parameter nào.**

* Khi không khai báo tham số truyền vào, thì parameter ẩn lần lượt được sử dụng là **$0 và $1,….** Lúc này ta ko cần phải khai báo **a như ví dụ trên mà sử dụng $0 thay thế như dưới đây.**



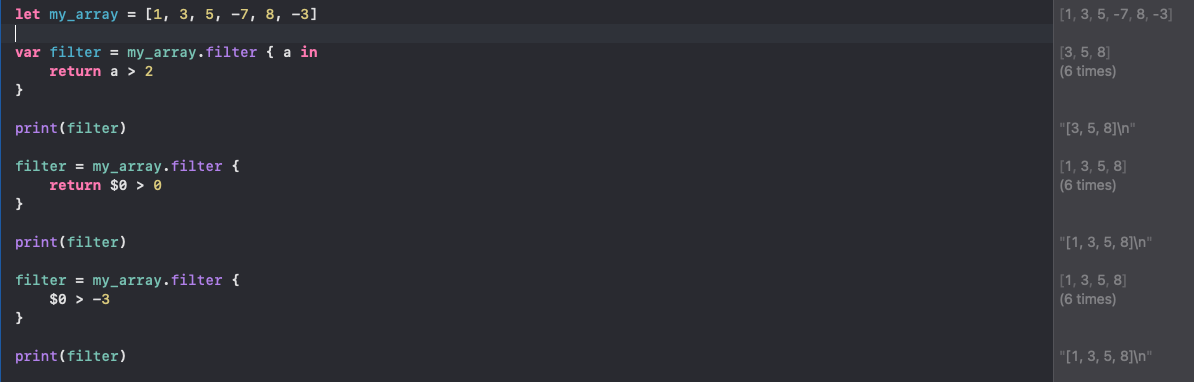
* Với trường hợp **không có keyword return thì closure đó chỉ có 1 câu lệnh, và câu lệnh đó sẽ chính là giá trị return lại:**



Với ví dụ trên thì closure có đúng 1 lệnh nên **$0 > -3 sẽ tương đương với return $0 > -3.**

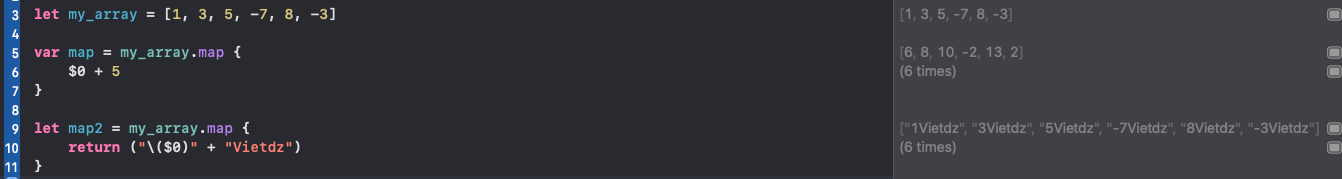
## 25.1 Array.filter()

Instead of creating a loop, you can use the array’s built-in filter() method to filter elements **to a new array**.



## 25.2 Array.map()

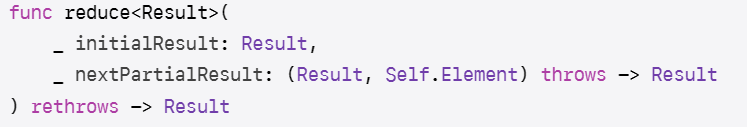
The map() method takes each value of an array, **performs some action on it, and finally places the new value into a new array**. This is handy(tiện dụng) and can be used instead of a for or while loop.



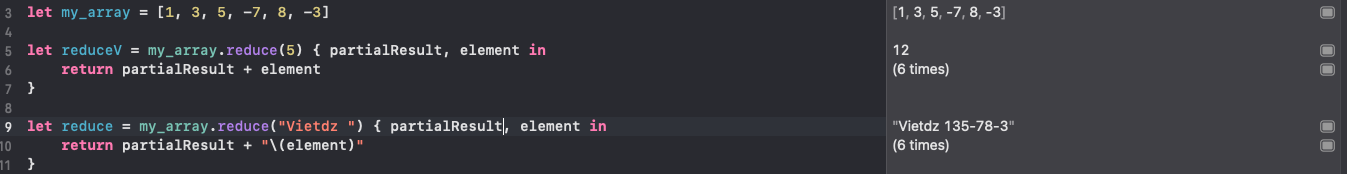
## 25.3 Array.reduce()

“Returns the result of combining the elements of the sequence using the given closure.”

Nghĩa là hàm này sẽ kết hợp các phần tử trong 1 mảng lại với nhau, và trả lại 1 giá trị duy nhất.

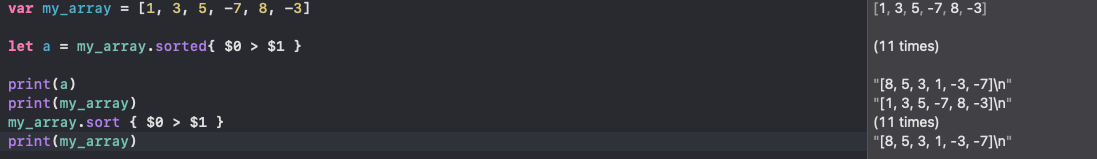


* **initialResult: Là giá trị khơỉ tạo ban đầu của kết quả**
* **nextPartialResult: Là một closure với Result là giá trị kết quả trước và self.element là các phần tử trong mảng**



## 25.4 Array.sort()

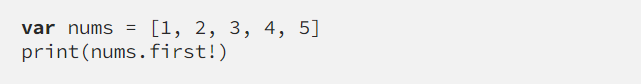
Ta có hàm **sort() sẽ thay đổi giá trực tiếp trên chính array đó**. Còn với hàm **sorted() thì sẽ không thay đổi giá trị trực tiếp trên array đó mà sẽ return lại 1 array mà đã được sort.**



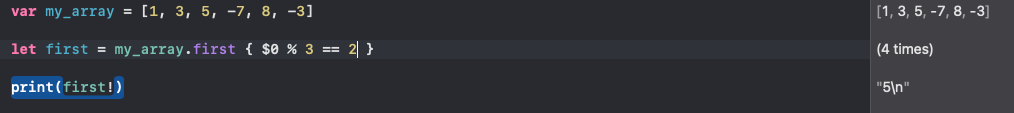
Ta thấy với hàm sorted thì my\_array ko bị thay đổi, còn với hàm sort thì array đó đã bị thay đổi,

## 25.5 Array.first

To get the first element of the array, you can access the array’s first property (nope, it is not a method):



Chúng ta có thể sử dụng method first() để lấy được giá trị đầu tiên thỏa mãn điều kiện truyền vào:



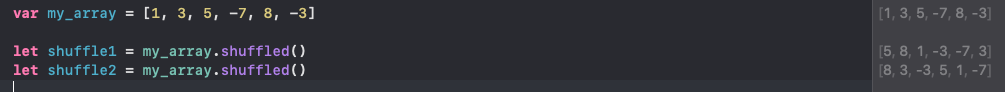
Value thu được sẽ là kiểu optional

## 25.6 Array.firstIndexOf()

Similarly to the first() method, firstIndexOf() can be used to get the first index of a possible element that satisfies a condition.

## 25.7 Array.shuffle()

To randomly mix up an array, use the shuffle() method.



## 25.8 Array.reverse()

You can reverse the order of an array by using the reverse() method.



## 25.9 Array.allSatisfy()

The allSatisfy() method is handy, yet you’d probably think it wouldn’t exist as a built-in method. You can use allSatisfy() to easily check if all the elements of an array satisfy a condition.

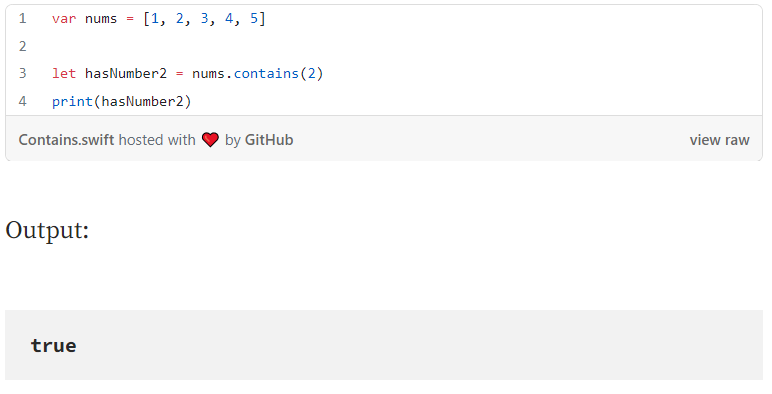
For example, let’s see if all the numbers are lower than 10:



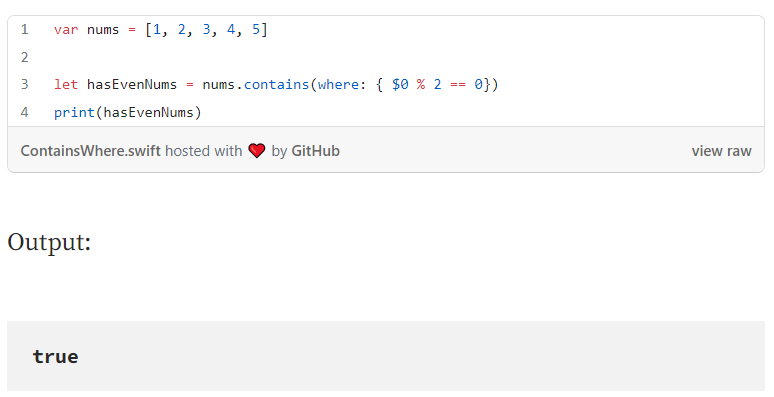
## 25.10 Array.contains()

Use the contains() method to check if:

* A specific element exists in an array.
* There’s an element that satisfies a condition.

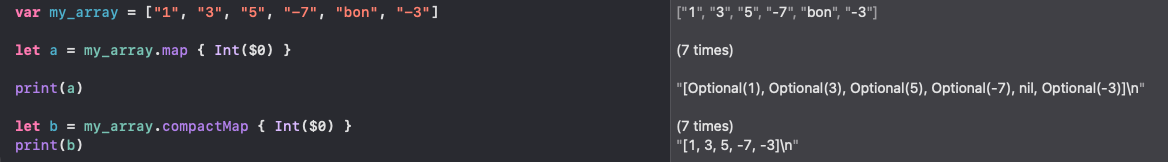


VD2:



## 25.11 Array.compactMap()

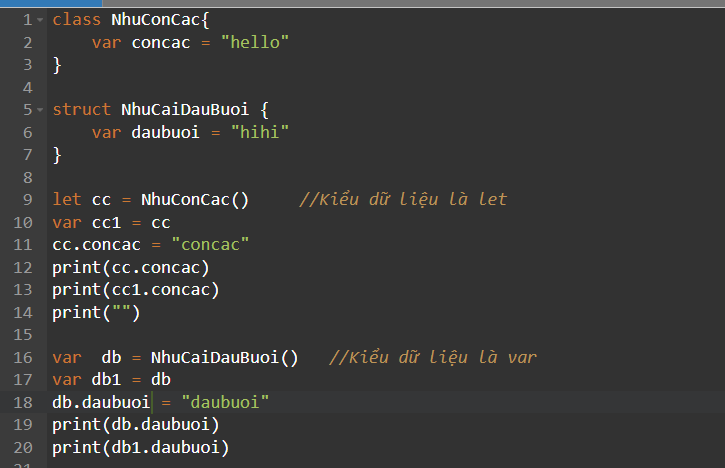
Như map, nó cũng return 1 mảng đã được chuyển đổi, nhưng với map, giá trị nhận được có thể là nil, nhưng compactMap sẽ loại bỏ các phần tử nil đó đi.



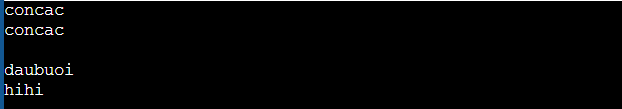
# II. Các câu hỏi phỏng vấn

## 2.1 So sánh class và struct

* Giống nhau:
* Đề chứa properties và method,
* Đều sử dụng được private, public
* Khác nhau:
* Để sửa property thì struct phải sử dụng mutating
* Class có kế thừa nhưng struct thì không
* Struct là kiểu Pass by value và Class là Pass by reference
* So sánh Pass by value và pass by reference:



Kết quả:



**Giải thích 1**: Ta có cc và cc1 cùng tham chiếu(reference) đến 1 địa chỉ vùng nhớ, nên khi vùng nhớ đó thay đổi thì các biến còn lại tham chiếu đến đều thay đổi theo.

Ta có db và db1 là pass by value, nghĩa là db và db1 tham chiếu đến 2 vùng nhớ khác nhau. Nên 1 thằng thay đổi thì thằng còn lại ko thay đổi

**Giải thích 2:** Ta có khi khai báo let cc = NhuConCac(), nghĩa là biến cc chỉ được phép reference đến vùng nhớ này, nếu gán tiếp cc = NhuConCac() thì sẽ failed. Và keyword let đó chỉ có nghĩa ko cho biến đó reference đến vùng nhớ khác, còn các property trong class vẫn thay đổi được(Như trong VD trên ta thay đổi property concac)

Nếu ta khai báo **let db = NhuCaiDauBuoi()** thì ta sẽ làm biến đó vừa không được reference sang vùng nhớ khác và sẽ làm cho tất cả các property trong class trở thành let, nghĩa là ta sẽ không được phép thay đổi giá trị của property

* Lời khuyên của Apple:

Khi nào cần kế thừa thì mới dùng class 😊))))-

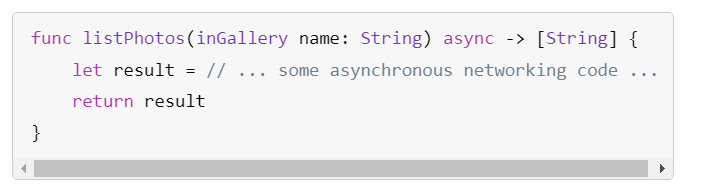
## 2.2 Concurrency – Async Await

Swift has built-in support for writing asynchronous and parallel code in a structured way. **Asynchronous code can be suspended and resumed later.** Việc dừng và tiếp tục vận hành code cho phép ta updateUI ngay lập tức sau đó tiếp tục làm các việc tốn thời gian như fetch data(như kiểu đang fetch data, ta suspend lại để update UI, sau đó lại resume để fetch data**).  Parallel code means multiple pieces of code run simultaneously** — for example, a computer with a four-core processor can run four pieces of code at the same time, with each core carrying out one of the tasks. A program that uses parallel and asynchronous code carries out multiple operations at a time; it suspends operations that are waiting for an external system, and makes it easier to write this code in a memory-safe way.

* **Defining and Calling Asynchronous Functions**

**An asynchronous function or asynchronous method is a special kind of function or method that can be suspended.** Một hàm synchronous là 1 hàm chạy đến khi hoàn thành, throw error hoặc return value,… Nhưng với một hàm **Asynchronous** thì nó có thể làm được những điều đấy, nhưng nó có thêm 1 tính năng đó là có thể được dừng lại ở giữa quá trình khi đợi (**waiting**) 1 điều gì đó. Trong thân hàm hoặc method, ta đánh dấu vị trị có thể sẽ được dừng lại.

Để chỉ ra rằng phương thức đó là **asynchronous,** ta viết keyword **async** đằng sau parameter.



For a function or method that’s both asynchronous and throwing, you write async before throws.

Khi thực thi 1 **func or method asynchronous,** việc thực thi sẽ bị suspend cho đến khi phương thức đó return. Ta viết **await** trước lời gọi hàm để đánh dấu rằng hàm này có thể sẽ bị dừng lại. Việc ghi **await** trước **func or method** cũng giống như việc viết **try** trước 1 **func** mà có thể **throw error.** Khi chạy 1 **asynchronous func**, việc thực thi func chỉ có thể bị **suspended** khi ta gọi 1 **asynchronous func khác. Ví dụ:**

****

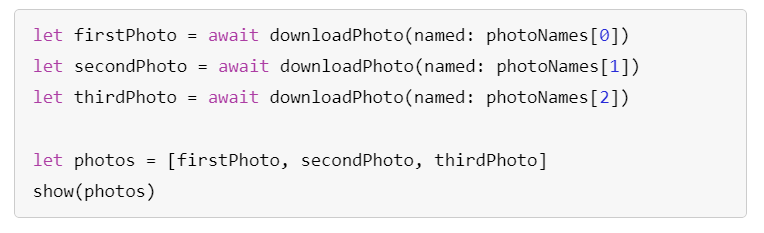
Bởi vì 2 func listPhotos(inGallery:) and downloadPhoto(named:) đều cần request netword , nó có thể tốn rất nhiều thời gian thực thi. Đánh dấu cả 2 func bằng từ khóa **async** sẽ cho phép app không bị block trong khi chờ ảnh sẵn sàng. Để hiểu bản chất đồng thời của ví dụ trên, đây là một thứ tự thực hiện có thể:

* The code starts running from the first line and runs up to the first await. It calls the listPhotos(inGallery:) function and **suspends execution while it waits for that function to return.**
* Trong khi main thread bị block, thì việc requet Photo sẽ được thực thi trong background thread.
* After listPhotos(inGallery:) returns, this code continues execution starting at that point. It assigns the value that was returned to photoNames.
* The lines that define sortedNames and name are regular, synchronous code. Because nothing is marked await on these lines, there aren’t any possible suspension points.(Ý là ko có từ **await,** vậy nó là synchronous code nền app ko bị suspend).
* Tương tự cho phần bên dưới

Ta thấy rằng việc đánh dấu **await** chỉ định rằng phần code ở đây có thể tạm dừng việc thực thi code cho đến khi func kết thúc hoặc return. Swift suspends the execution of your code on the **current thread** and runs some other code on that thread instead**. Tóm lại hiện tại phần ta học được là việc đánh dấu async và await sẽ tạm dừng main thread(block thread) và đẩy phần code trong func mà được đánh dấu vào trong 1 thread khác để nó thực thi, sau khi thực thi xong sẽ trả về cho main thread.**

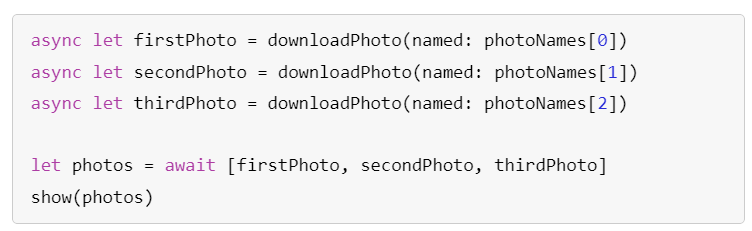
* **Calling Asynchronous Functions in Parallel**

Calling an asynchronous function with await runs only one piece of code at a time. While the asynchronous code is running**, the caller waits for that code to finish before moving on to run the next line of code.**



Với ví dụ này thì secondPhoto phải đợi xong việc downloadPhoto() ở firstPhoto xong,… dẫn đến nó như shit. Ta có một cách để gọi 3 hàm này 1 cách đồng thời tại 1 thời điểm(each photo can download independently, or even at the same time.)

To call an asynchronous function and let **it run in parallel with code around it**, write **async** in front of **let** when you define a constant, and then write await each time you use the constant.



In this example, all three calls to downloadPhoto(named:) start without waiting for the previous one to complete. If there are enough system resources available, they can run at the same time. **None of these function calls are marked with await because the code doesn’t suspend to wait for the function’s result. Tuy nhiên chương trình cần phải đợi kết quả trả về để gán vào photos, nên ta viết await để pause việc thực thi cho đến khi cả 3 ảnh được download xong.**

* Tóm lại việc xử dụng await thì sẽ block ở chỗ đấy để thực thi xong
* Sử dụng async let sẽ chạy đồng thời nhiều đoạn code await
* **Task và Task Group**

A task is a unit of work that can be run asynchronously as part of your program. All asynchronous code runs as part of some task. The async-let syntax described in the previous section creates a child task for you. You can also create a task group and add child tasks to that group, which gives you more control over priority and cancellation, and lets you create a dynamic number of tasks.

Tasks are arranged in a hierarchy. Each task in a task group has the same parent task, and each task can have child tasks. Because of the explicit relationship between tasks and task groups, this approach is called structured concurrency



Một vài chân lý:

* Mọi Process đều có 1 Thread Pool, hệ điều hành sẽ quản lý Thread Pool cho ta. Khi ta tạo 1 Task, hệ thống sẽ đính Task đó cho 1 Thread thích hợp trong Thread Pool dưa trện các yếu tố như tính sẵn sàng của resourse và độ ưu tiên của Task.
* Khi 1 thread được tạo ra, thread đó sẽ được đính với Actor Context. Ta có **Backroud Thread** được đính với s**Global Context**. Vậy nếu trong class A được đánh dấu là @MainActor, mà ta sử dụng DispathQueue.Global.async {Task {funcA()}} thì funcA sẽ được thực thi trong **Backgroud Thread chứ ko phải MainThread.s**
* Khi ta đáng dấu function hoặc class là @MainActor, điều đó có nghĩa là **mọi Tasks mà được tạo từ function hoặc class đó sẽ tự động được gán với main actor context, nghĩa là main thread.**
* Nếu ta thực thi 1 function hoặc class ở 1 actor context, thì các Task được tạo ra từ class hoặc function đó sẽ chạy ở actor’context đó. Có nghĩa là các Task sẽ kế thừa context từ context cha. VD Nếu class B có 1 function C tạo Task và là non-actor context, Task đó sẽ được đính với global-actor context . Sau đó ta gọi function đó trong Class A được đính là MainActor. Nếu ta gọi B().C thì function C sẽ được chạy trong global context, nghĩa là chạy trong Thread khác. Nếu ta gọi Task {B().C} thì lúc này C sẽ kế thừa context của Task to, và context đó chính là MainContext-Actor, dẫn đến function sẽ chạy trong Thread Main. Để có thể gọi B().C ko trong MainActor, ta sử dụng Task.Detached {}. Ví dụ: Task.Detached {B().C}.
* Ta biết rằng Task chia code thành các phần cô lập mà chạy song song với nhau. Ta có thể sử dụng TaskDetach để chạy multiple Task tại 1 thời điểm (Vì nếu với class được đính là MainActor thì khi dùng Task thì Task đó sẽ kế thừa context của MainActor dẫn đến tất cả Task sẽ phải chạy trong Main Thread, dẫn đến chúng phải đợi Task này kết thúc rồi đến Task kia chạy), dẫn đến có thể có nhiều Task thay đổi giá trị 1 properties tại 1 thời điểm. Dẫn đến hiện tượng Race Conditions. Để tránh hiện tượng này Swift cung cấp cho ta **Actors. Actors** cũng là kiểu reference types, không giống Class, **Actor chỉ cho phép 1 Task được phép truy cập tới các thuộc tính có thể thay đổi(mutable state) tại tại 1 thời điểm**. Dẫn đến an toàn hơn cho chương trình khi thực thi multiple Tasks trong cùng 1 instance của Actor. Swift đảm bảo cơ chế chỉ cho phép 1 Task tương tác với trạng thái thuộc tính của Actor tại 1 thời điểm. Các Task khác muốn thực thi thì phải đợi đến khi Task đó kết thúc việc truy cập thì mới có thể truy cập.