**IOS**

Contents

[1. The Basics 2](#_Toc121142881)

[1.1 Type Aliases 2](#_Toc121142882)

[1.2 Tuples 3](#_Toc121142883)

[1.3 Optionals 4](#_Toc121142884)

[1.4 Error Handling 5](#_Toc121142885)

[2. Basic Opearators 7](#_Toc121142886)

[3. String and Character 9](#_Toc121142887)

[4. Collection types 13](#_Toc121142888)

[4.1 Array 14](#_Toc121142889)

[4.2 Set(Tạm bỏ) 15](#_Toc121142890)

[4.3 Dictionaries(tạm bỏ) 16](#_Toc121142891)

[5. Control Flow(làm chưa kĩ) 16](#_Toc121142892)

[5.1. Early Exit 16](#_Toc121142893)

[5.2 Checking API(chưa làm) 17](#_Toc121142894)

[6. Functions(Chưa làm) 17](#_Toc121142895)

[7. Closures 17](#_Toc121142896)

[7.1 Sorted Method(Sắp xếp mảng là chủ yếu) 17](#_Toc121142897)

[8. Enemerations(Recursive chưa làm) 21](#_Toc121142898)

[9. Structures and Classes 24](#_Toc121142899)

[10. Properties For Structs (Làm sơ sài vlon) 26](#_Toc121142900)

[11. Methods (Bỏ type Method) 28](#_Toc121142901)

[12. Subscripts 29](#_Toc121142902)

[13. Inheritance 30](#_Toc121142903)

[14. Initialization 34](#_Toc121142904)

[14.1 Struct 34](#_Toc121142905)

[14.2 Class Inheritance and Initialization 35](#_Toc121142906)

[15. Deinitalization 39](#_Toc121142907)

[16. Error Handling 39](#_Toc121142908)

[17. Type Casting 41](#_Toc121142909)

[18. Nested Type 43](#_Toc121142910)

[19. Extensions 45](#_Toc121142911)

[19.1. Computed Properties 46](#_Toc121142912)

[19.2. Initializers 46](#_Toc121142913)

[19.3. Methods 48](#_Toc121142914)

[19.4. Mutating Instances 49](#_Toc121142915)

[20. Protocol 50](#_Toc121142916)

[20.1 Property Requirements 50](#_Toc121142917)

[20.2 Method Requirement 52](#_Toc121142918)

[20.3 Mutating Protocol 52](#_Toc121142919)

[20.4 Initializer Requirements 52](#_Toc121142920)

[20.5 Protocol as Type 53](#_Toc121142921)

[20.6 Optional Protocol 54](#_Toc121142922)

[21. Generic 57](#_Toc121142923)

[22 Automatic reference counting 57](#_Toc121142924)

[22.1 Strong và weak 59](#_Toc121142925)

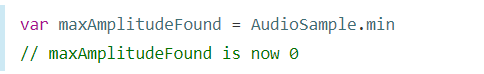
# 1. The Basics

## 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

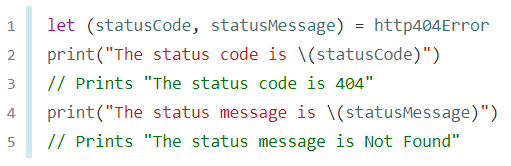


## 1.2 Tuples

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



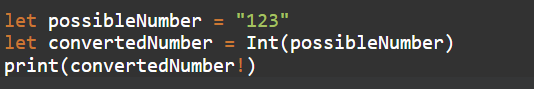
Kết quả in ra: 200

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:



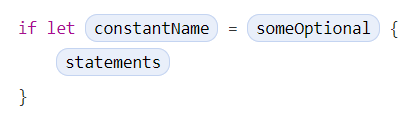
Lúc này biến convertedNumber sẽ là 1 optionals. Lý do là gì?

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.

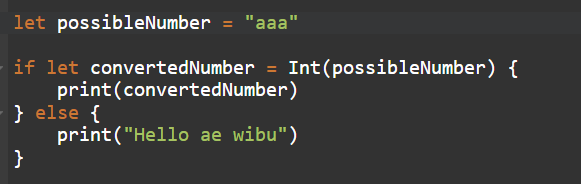
* Optional binding:

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



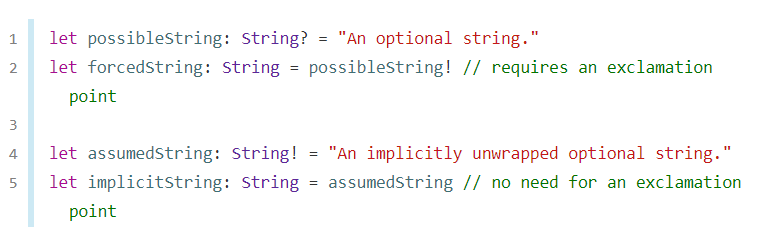
Ta có thể hiểu rằng, 1 biến thông thường không thể được gán trị nil, nên nếu someOptional lúc này có giá trị thì constanName sẽ được gán và chạy các lệnh statements, nhưng nếu someOptional là nil thì sẽ không được gán và sẽ không chạy statements



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

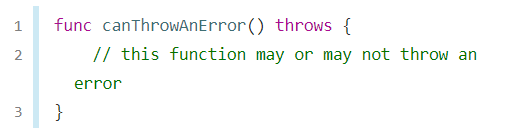


Ta thấy có 2 cách khác nhau, nhưng bên trên ta bắt buộc phải unwrap, bên dưới thì không. Ta hiểu đơn giản là “an implicitly unwrapped optional as giving permission for the optional to be force-unwrapped if needed”.

## 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.”



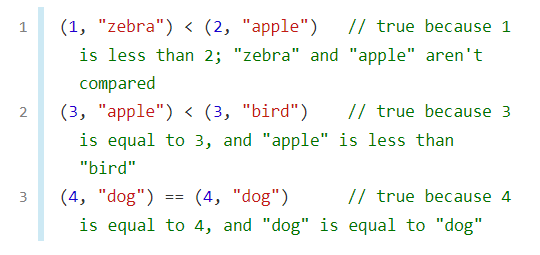
Ta hiểu như sau: “In this example, the makeASandwich() function will throw an error if **no clean dishes are available or if any ingredients(thành phần) are missing**. Because makeASandwich() can throw an error, the function call is wrapped in a try expression. By wrapping the function call in a do statement, 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”

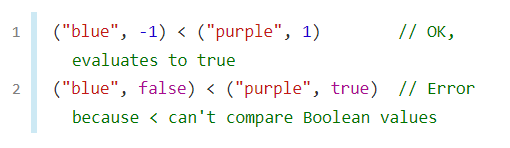
# 2. Basic Opearators

* 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.”

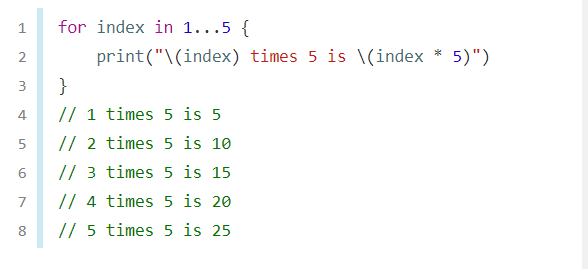


“ As demonstrated in the code above, you can compare two tuples of type (String, Int) because both String and Int values can be compared using the < operator. In contrast, two tuples of type (String, Bool) can’t be compared with the < operator because the < operator can’t be applied to Bool values.”

* 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.”

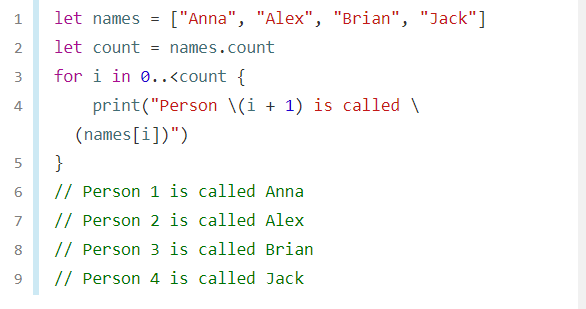
“The closed range operator is useful when iterating(lặp đi lặp lại) over a range in which you want all of the values to be used, such as with a for-in loop:”



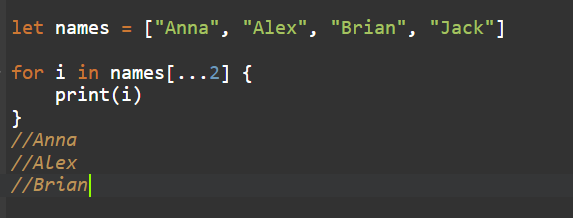
Ta cũng có các khái niệm như Half-Open Range Operator, One-Sided Ranges

* Half-Open range

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

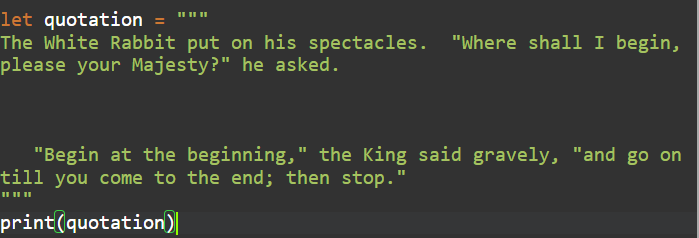


* One-Sided Ranges

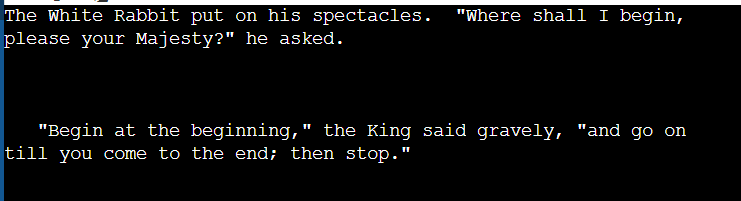


# 3. String and Character

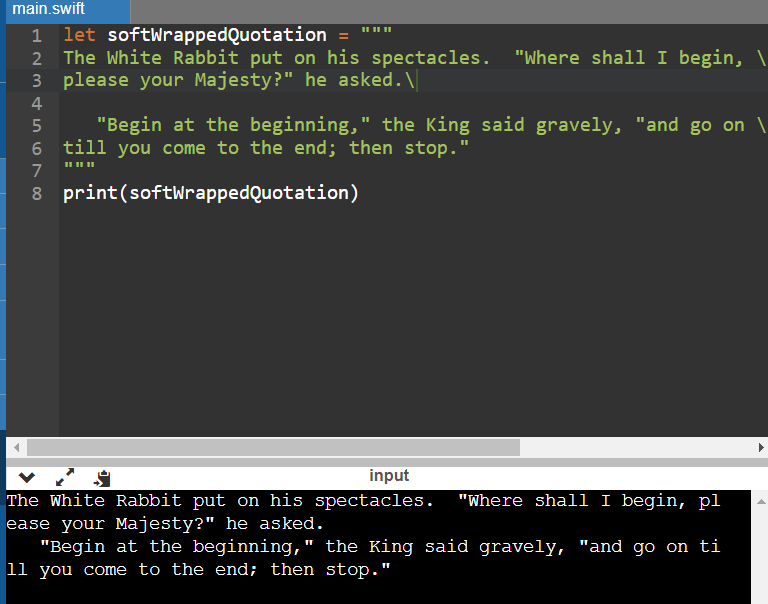
* Multiline String Literals

****

Kết quả:

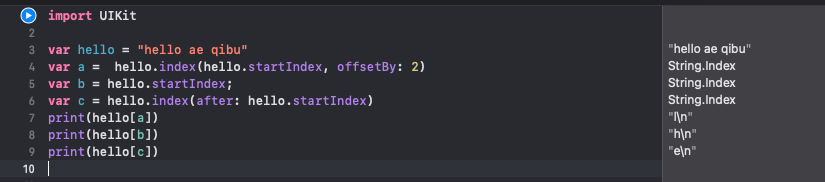


Ta sử dụng \ để nối liền dòng trên và dòng dưới



* 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 đằng sau String(ý là 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:



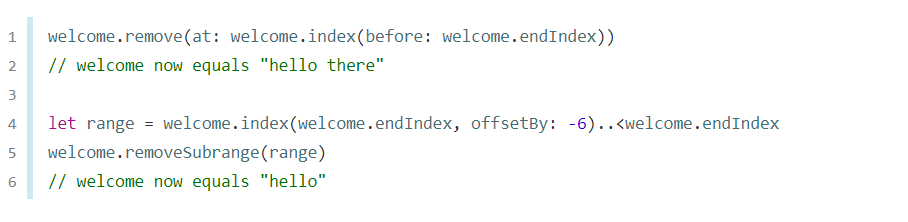
**Từ nay việc truy cập, thay đổi, thêm, sửa xóa String ta đều thông qua các phương thức như endIndex, startIndex, index(after: ), index(before: ), index(\_ offset: )**

* Inserting and Removing:

- Insert



-Remove

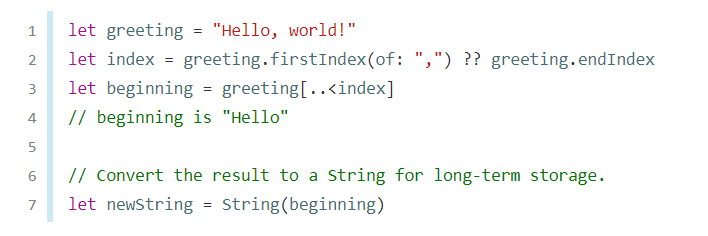


Chú ý: Ta thấy method là removeSubrange là xóa đi 1 khoảng kí tự trong String

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

* Substrings

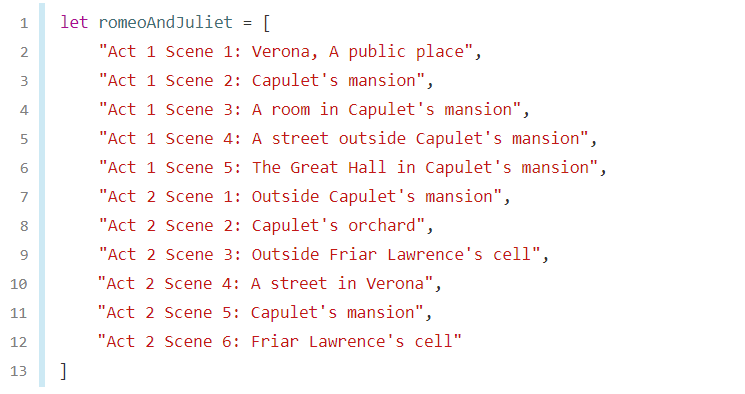
Là ta thu được 1 chuỗi nhỏ từ 1 chuỗi ta muốn

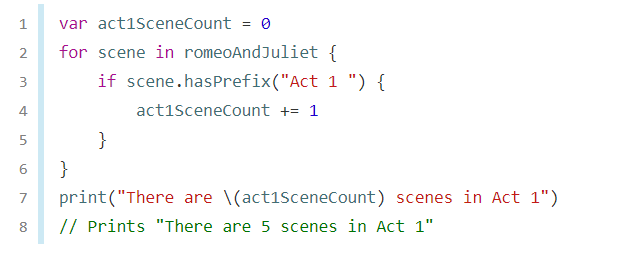


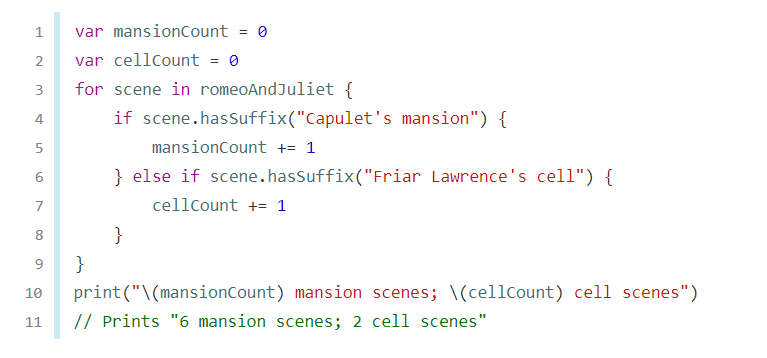
Ta có greeting.firstIndex(of: ",") là trả về vị trí đầu tiên gặp dấu , nếu không có dấu phẩy nào thì lấy giá trị nil.

* 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.

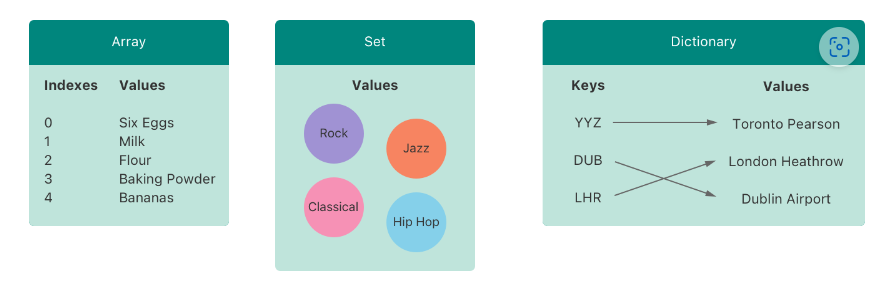






# 4. Collection types

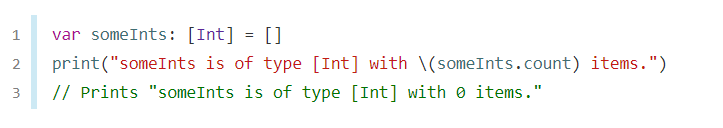
Swift cung cấp cho ta 3 collection types chính là: Arrays, Set và Dicionaries. Arrays tập hợp các giá trị được sắp xếp theo thứ tự. Set là tập giá trị không được sắp xếp theo thứ tự và Dictionaries cũng không theo thứ tự sử dụng cặp key-value



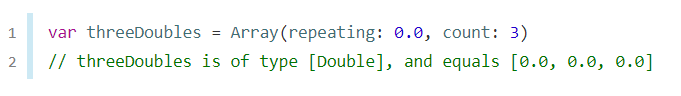
## 4.1 Array

Khởi tạo array 4 cách:

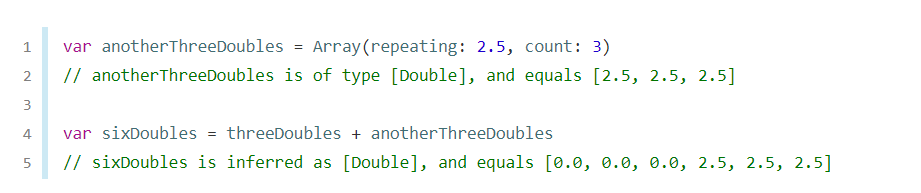
* Khởi tạo 1 empty array



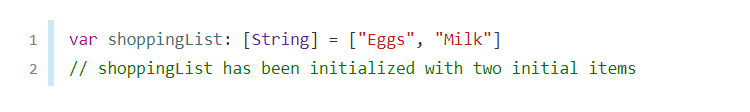
* Khởi tạo array với default value



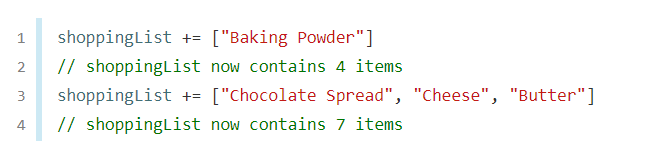
* Khởi tạo array với adding 2 arrays together



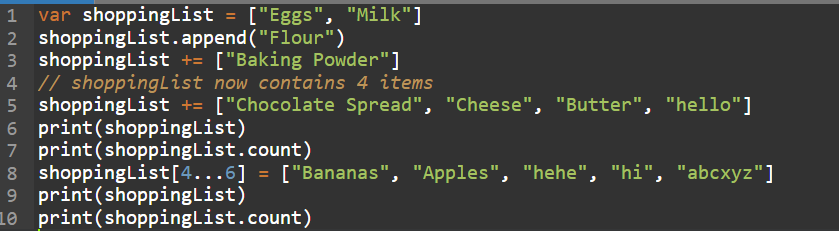
* Khởi tạo với việc gắn giá trị trước



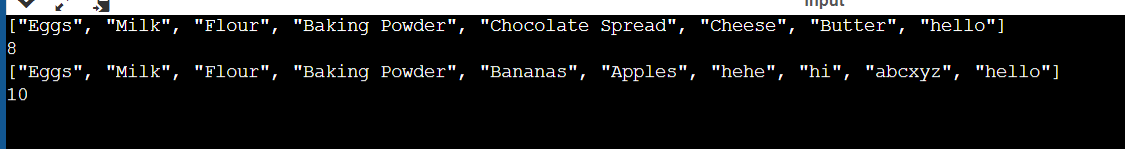
* Các phương thức chính của Array hay dung
  + Cộng Array



* + 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.”

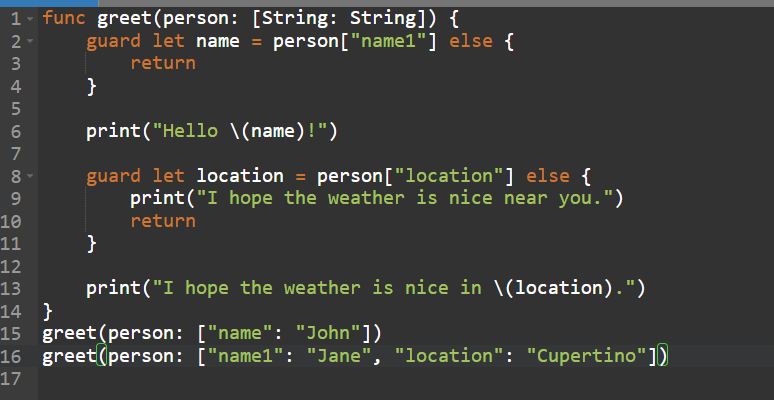
Tạm bỏ

## 4.3 Dictionaries(tạm bỏ)

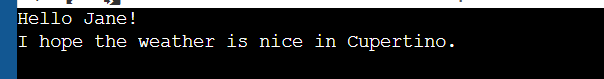
# 5. Control Flow(làm chưa kĩ)

## 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.

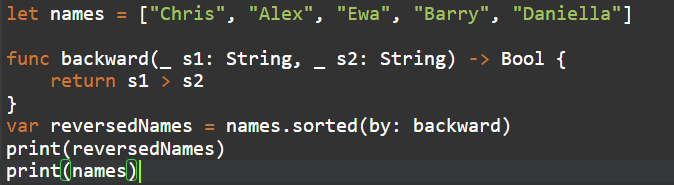
## 5.2 Checking API(chưa làm)

# 6. Functions(Chưa làm)

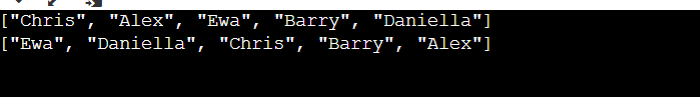
# 7. Closures

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

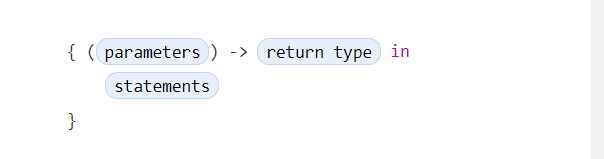
Swift’s standard library provides a method called sorted(by:),  which **sorts** an array of values of a known type, based on the output of a sorting closure that you provide. Once it completes the sorting process, the sorted(by:) method returns a new array of the same type and size as the old one, with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

****

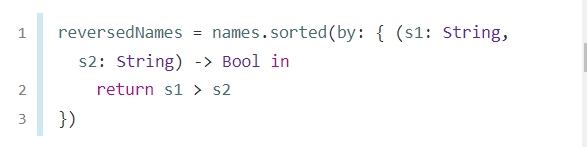
**Output:**

****

#### 7.2 Closure Expression Syntax

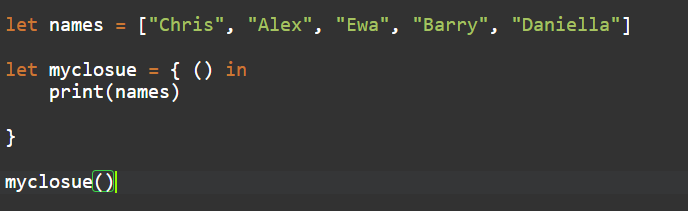


Ta sẽ kết hợp ví dụ trên và dưới ta được:

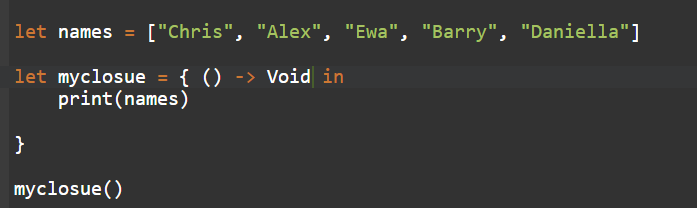


Các closure đặc biệt

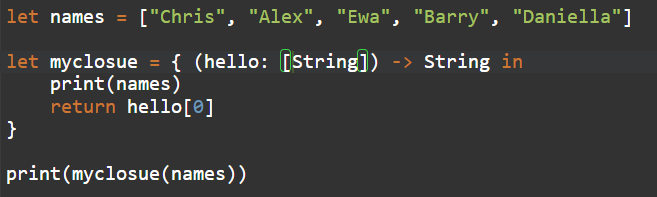
* Closure không có kiểu trả về:



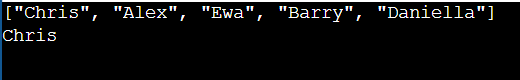
Ta bỏ cả -> (), hoặc



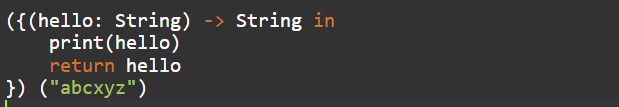
* Closure có tham số truyền vào và có kiểu trả về:



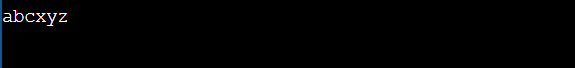
Output:



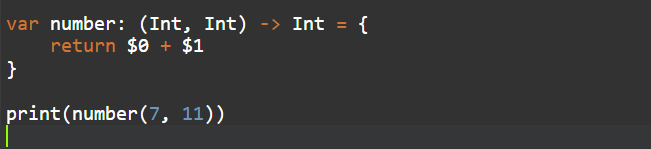
* Sủ dụng closure không sử dụng biến để hứng



Output:



* Closure kiểu chưa biết đặt tên



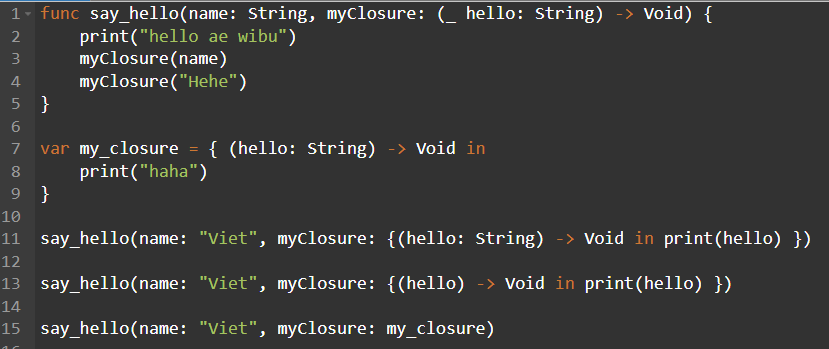
Output:



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

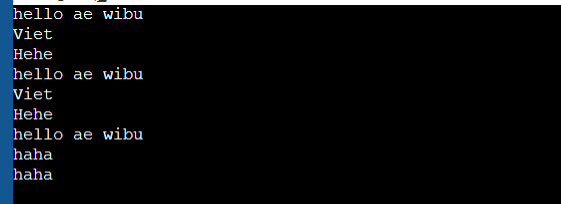
* Closure được sử dụng để làm tham số truyền vào cho function
* Chaining closure

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

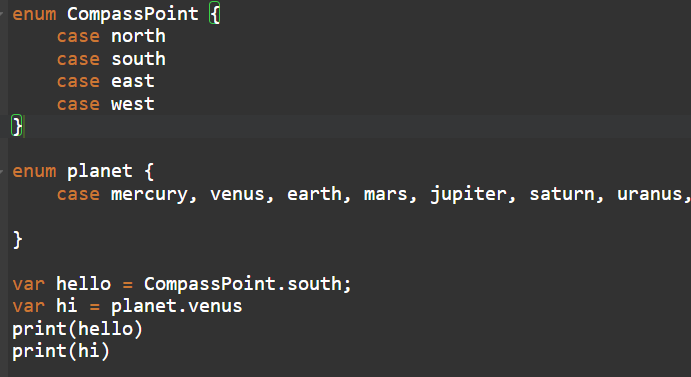


Ở cách viết 2, nó sẽ tự hiểu hello có kiểu String

# 8. Enemerations(Recursive chưa làm)

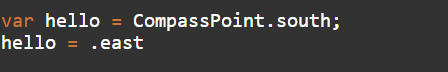
Enumerations in Swift are much more flexible, and don’t have to provide a value for each case of the enumeration. If a value (known as a raw value) is provided for each enumeration case, the value can be a string, a character, or a value of any integer or floating-point type

* Khởi tạo enum:

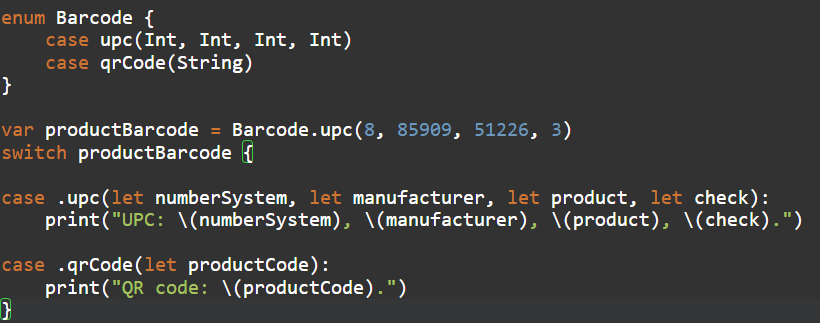


Ta thấy nếu viết trên cùng 1 hàng, ta dung dấu , để tách các case

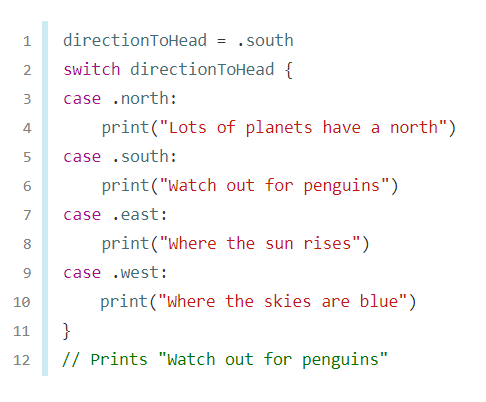
Khi biến hello được khai báo như trên, thì bên dưới ta có thể gán lại giá trị của hello qua lệnh sau:



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

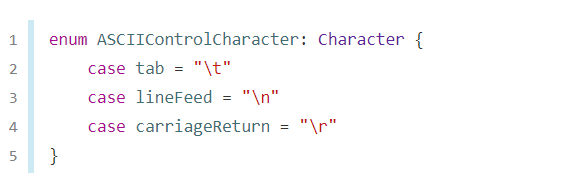


* Kết hợp Enumeration và Switch



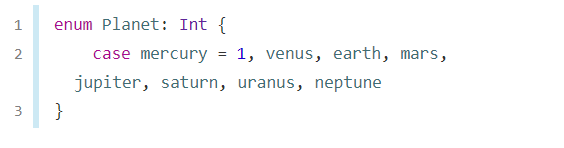
* Raw value

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



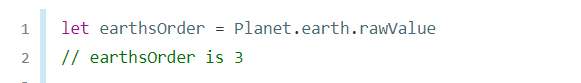
“Raw values can be strings, characters, or any of the integer or floating-point number types. Each raw value must be unique within its enumeration declaration.”

* Implicit Assigned Raw Value

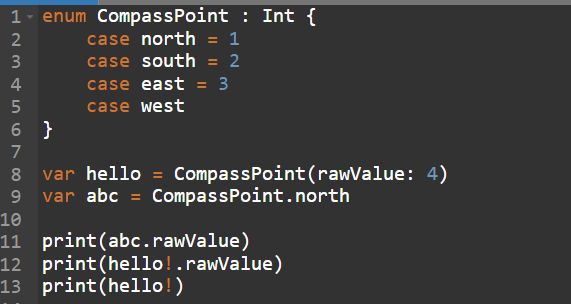


In the example above, Planet.mercury has an explicit raw value of 1, Planet.venus has an implicit raw value of 2, and so on. **When strings are used for raw values, the implicit value for each case is the text of that case’s name.**

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 **!**

* Recursive Enumeration(chưa làm)

# 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.”

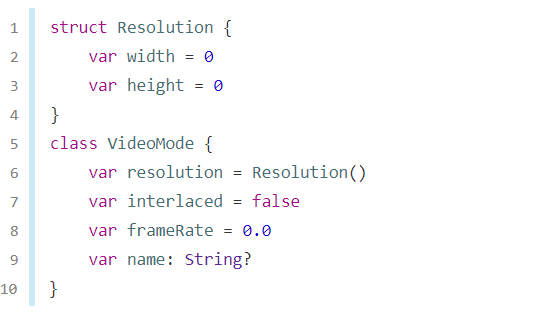
* Comparing Structures and Classes

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

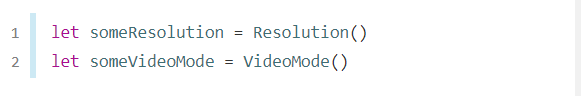
* Define properties to store values
* Define methods to provide functionality
* Define subscripts to provide access to their values using subscript syntax
* Define initializers to set up their initial state
* Be extended to expand their functionality beyond a default implementation
* Conform to protocols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

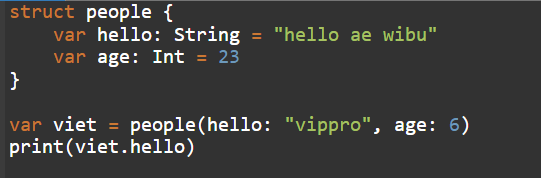


* Structure and Class Instances



* Memberwise Initializers for Structures Types

“All structures have an automatically generated memberwise initializer, which you can use to initialize the member properties of new structure instances. Initial values for the properties of the new instance can be passed to the memberwise initializer by name:”

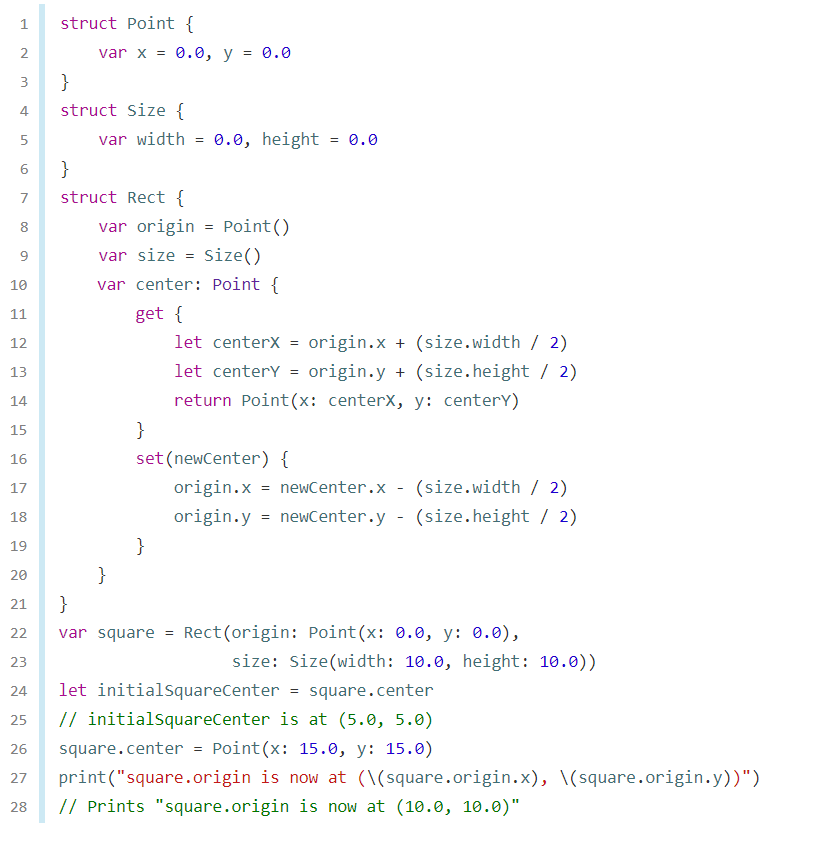


“Unlike structures, class instances don’t receive a default memberwise initializer. ”

# 10. Properties For Structs (Làm sơ sài vlon)

* Computed Properties(Chưa kĩ)

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



Tạm hiểu là là khi gán giá trị cho 1 biến khác thì gọi get



Còn khi được gán giá trị thì gọi set



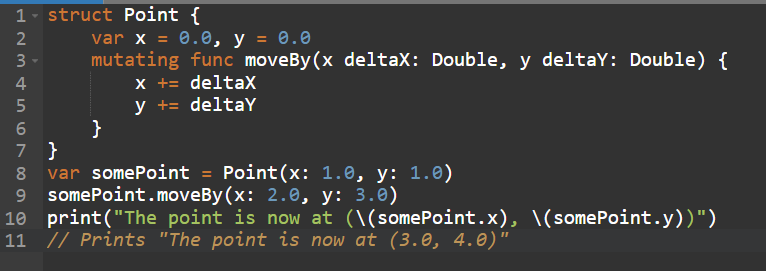
* Property Wrapper

# 11. Methods (Bỏ type Method)

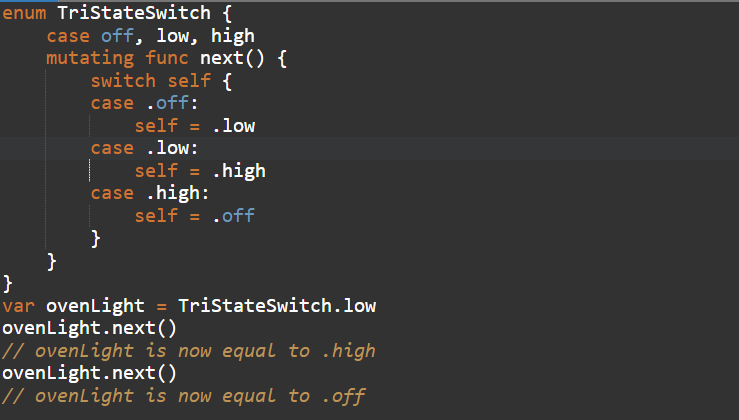
“Structures and enumerations are value types. By default, the properties of a value type can’t be modified from within its instance methods.”

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.

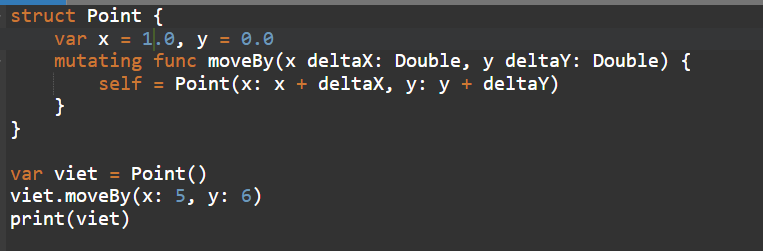
Với struct



Với Enum



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



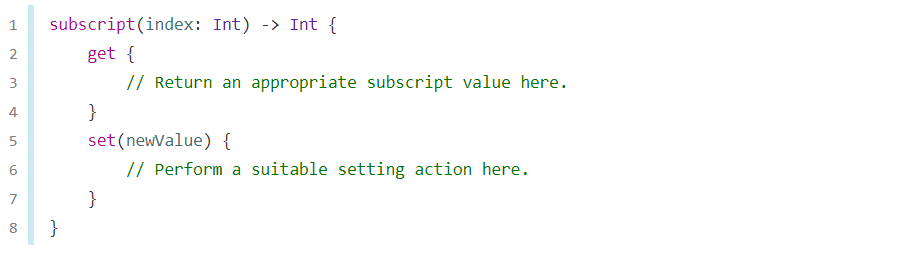
* Type Method (tạm bỏ)

# 12. Subscripts

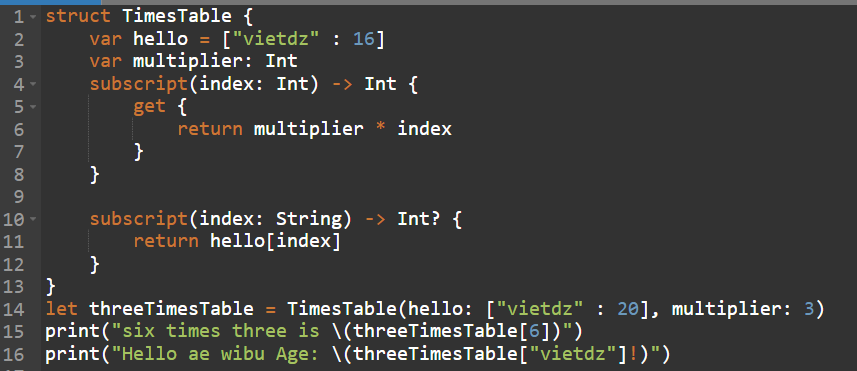
“Classes, structures, and enumerations can define subscripts, which are shortcuts for accessing the member elements of a collection, list, or sequence. You use subscripts to set and retrieve values by index without needing separate methods for setting and retrieval. For example, you access elements in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

* Subscript Syntax

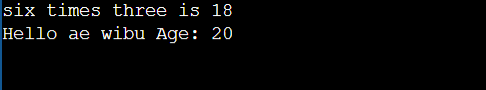


Ta có thể dung tên khác newValue. Ta có A default parameter called newValue is provided to your setter if you don’t provide one yourself. Ta thường ko dung set



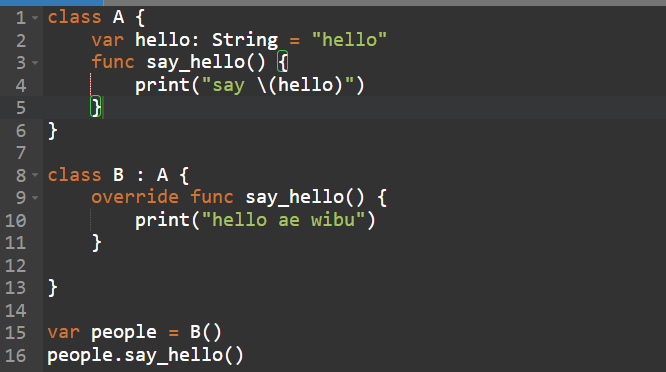
Ta thấy có nét tương đồng như mảng: arr[1] là lấy được giá trị tại vị trí thứ 2, giờ ta gọi threeTimesTable[6] thì ta cũng lấy được 1 giá trị nào đó.

Output:



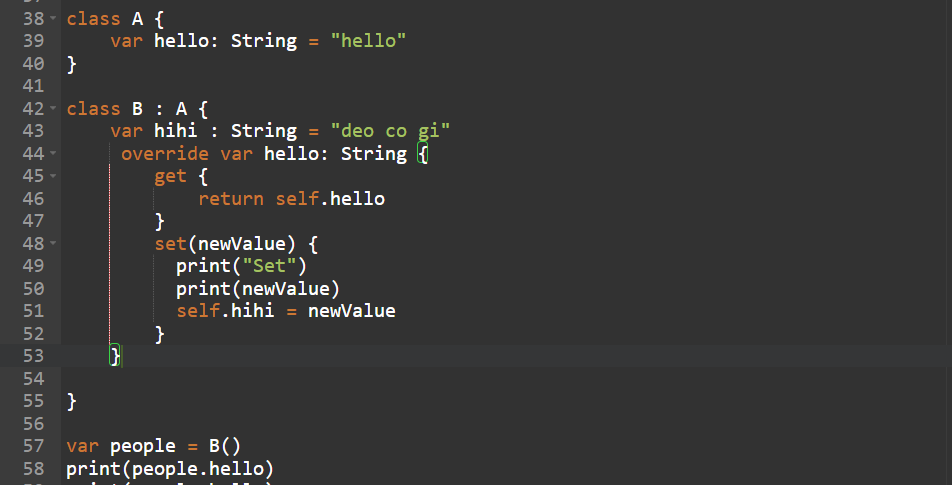
# 13. Inheritance

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

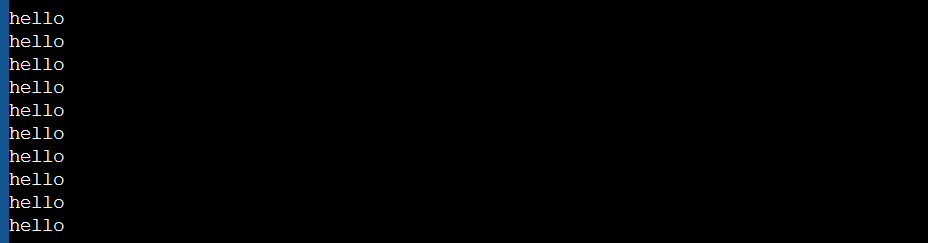


* 1. Override Property: Để override property, ta cũng thêm từ khóa override, việc override một propery giúp ta custom getter và setter instance cho property đó. Chú ý là khi ta overring 1 thuộc tính ta cần viết cả hàm get, set cho nó

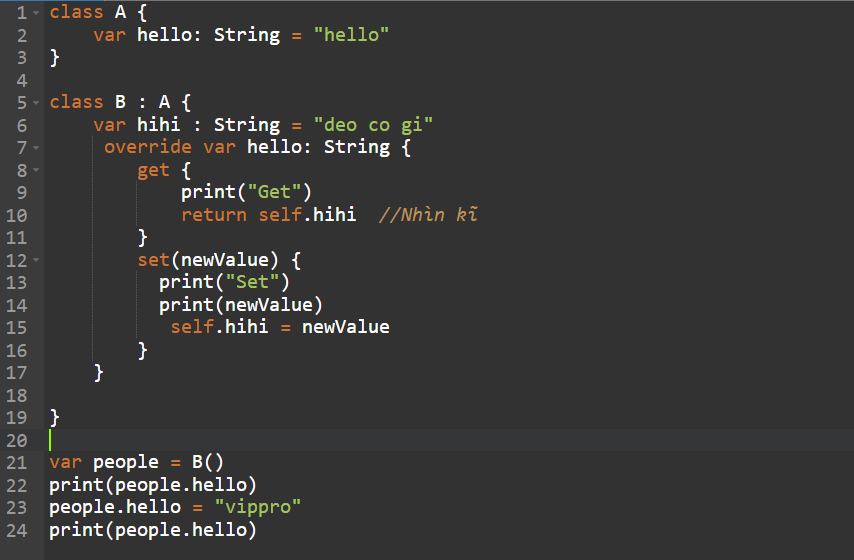
Chú ý 2:Khi ta viết gọi setter và getter mà return hoặc gán lại đúng với thuộc tính như thế thì sẽ vào vòng lặp vô hạn:



Output:



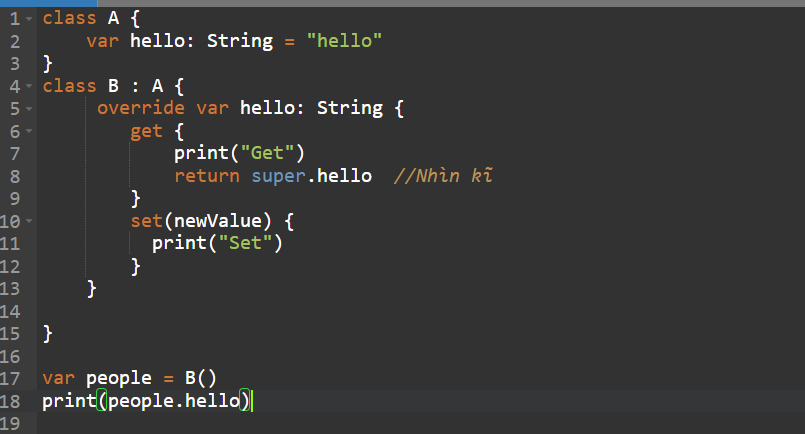
Nó sẽ in ra vô hạn chữ hello, vì nó sẽ call lại chính nó. Vì thế nếu để get hoặc set ta sẽ phải sử dụng 1 biến trung gian như sau:



Output:



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

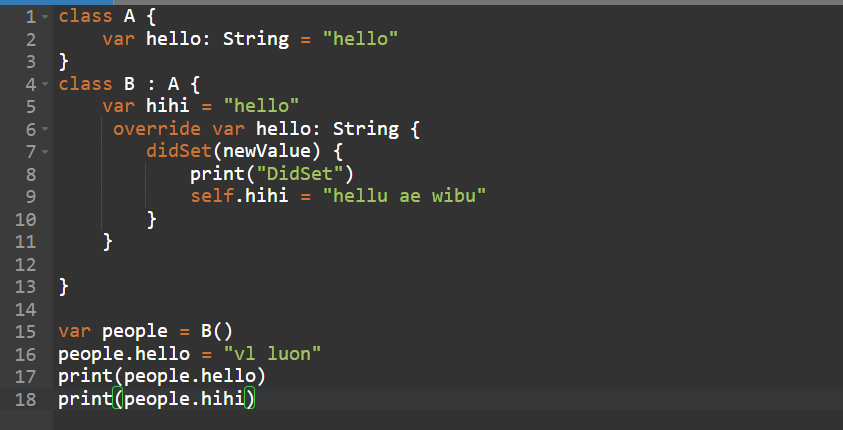


Kết quả:

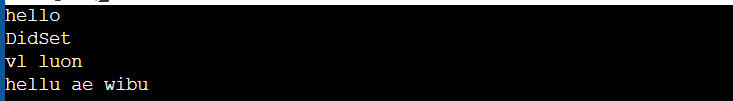


* “You can use property overriding to add property observers to an inherited property. This enables you to be notified when the value of an inherited property changes, regardless of how that property was originally implemented.”

“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:

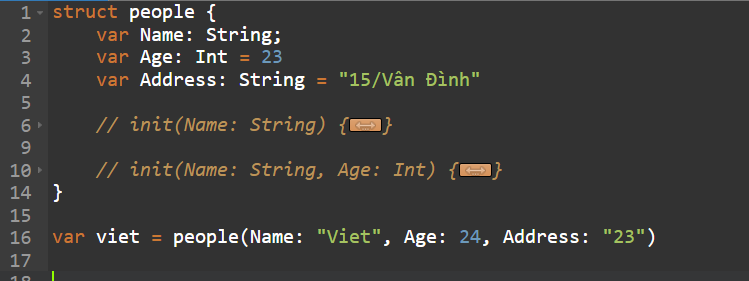


Ta thấy VD trên không có getter, vì khi dung didSet thì getter ko được xuất hiện và ngược lại.

# 14. Initialization

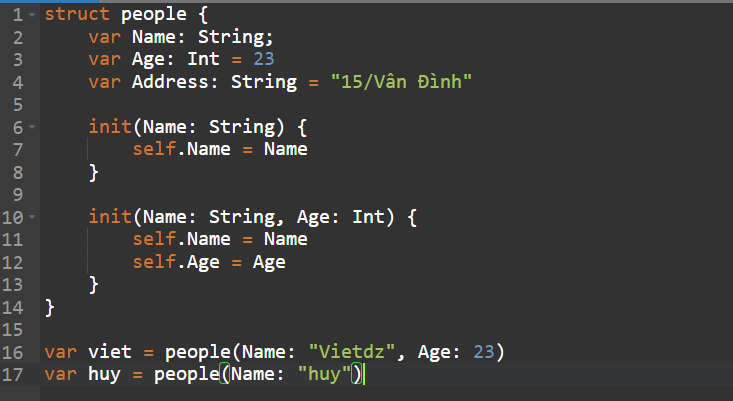
## 14.1 Struct

**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.**”**

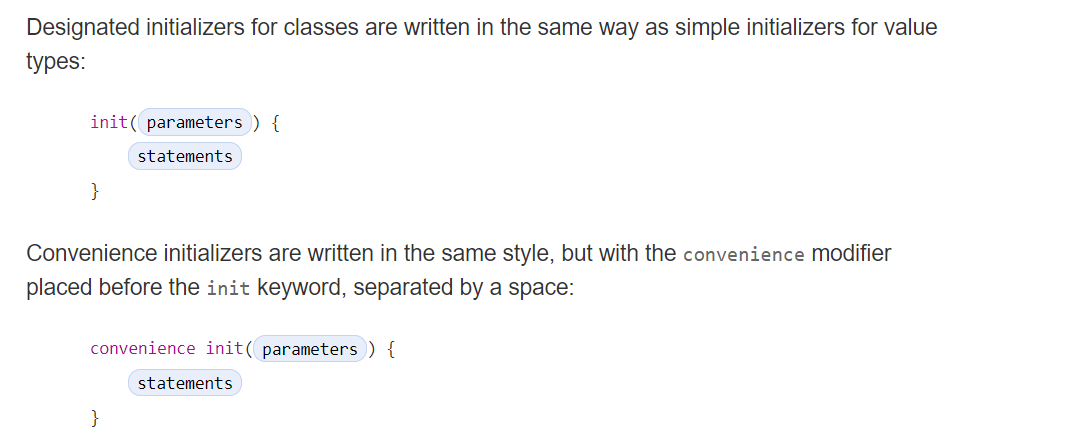
**Ta có: “**Swift defines two kinds of initializers for class types to help ensure all stored properties receive an initial value. These are known as **designated initializers and convenience initializers**.**”**

* **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.

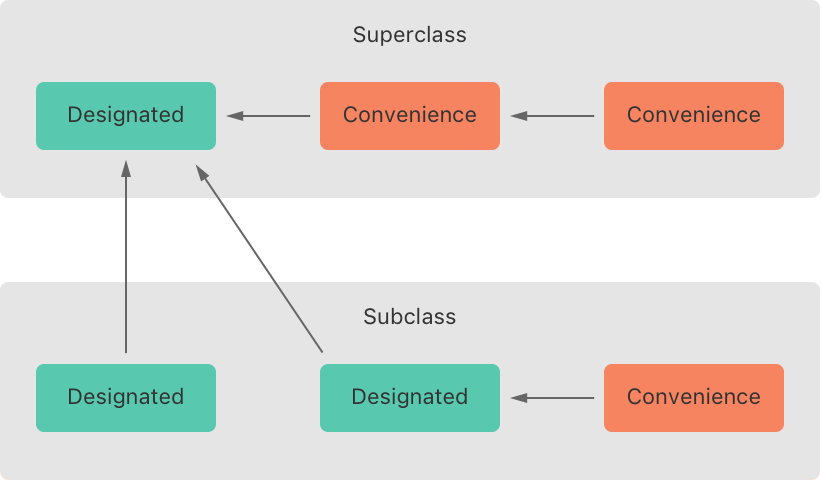
**Rule 3**

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:

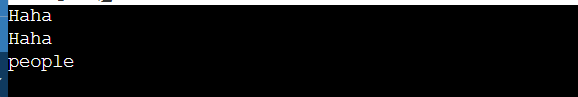


***Bốn Rule quan trọng:***

* ***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. 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)

******

***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.

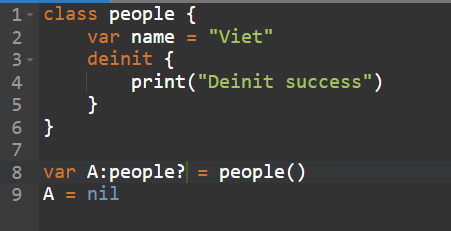
Rule 2: Ở subclass là class man, ta gán hết giá trị cho properties, ở đây là property male đã được gán là true. Sau khi gọi hàm init của superclass, bên dưới ta có thể tùy chỉnh giá trị properties theo ý mình(self.name = “HAHA”)

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

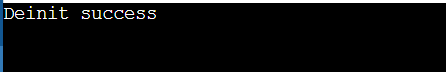
Rule 4: Ta không hề gọi 1 method hay đọc value của 1 property nào cả

# 15. Deinitalization

Ta sử dụng hàm deinit(), khi ta gán 1 object = nil thì hàm deinit() này sẽ được gọi

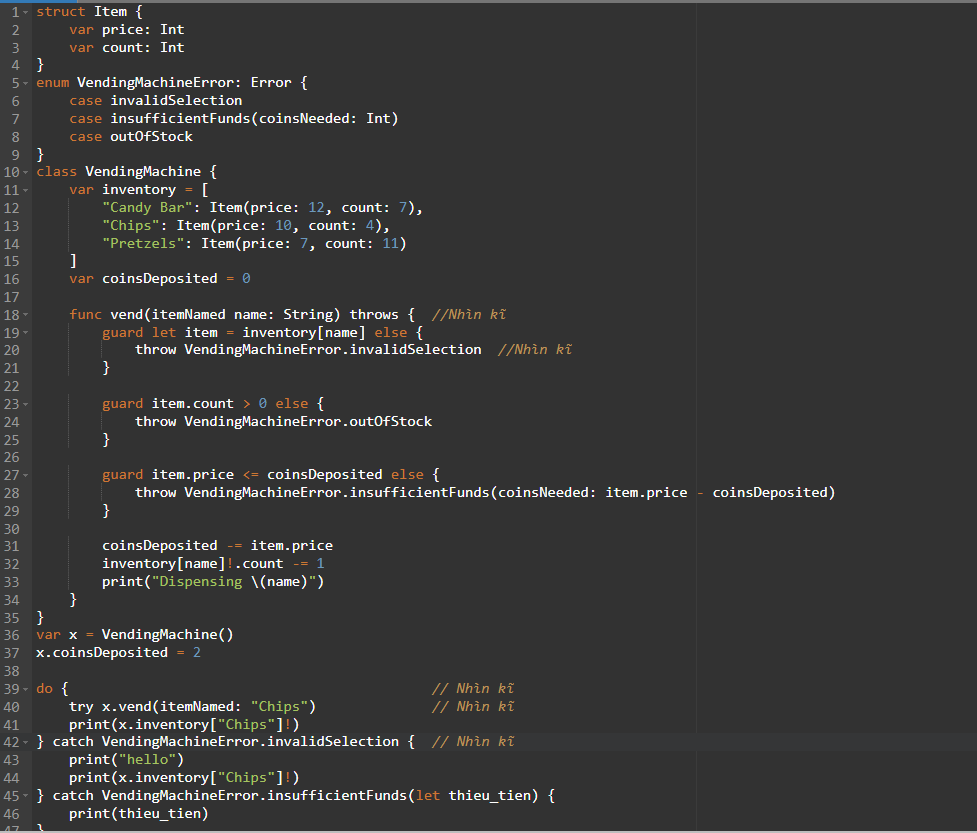


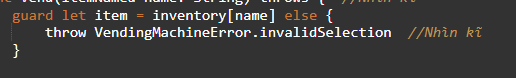
Output:



# 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

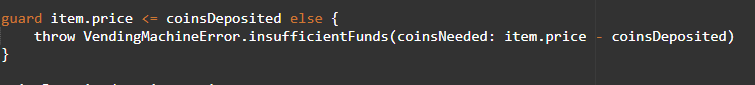


Ta có 1 chương trình bắt lỗi như sau:

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



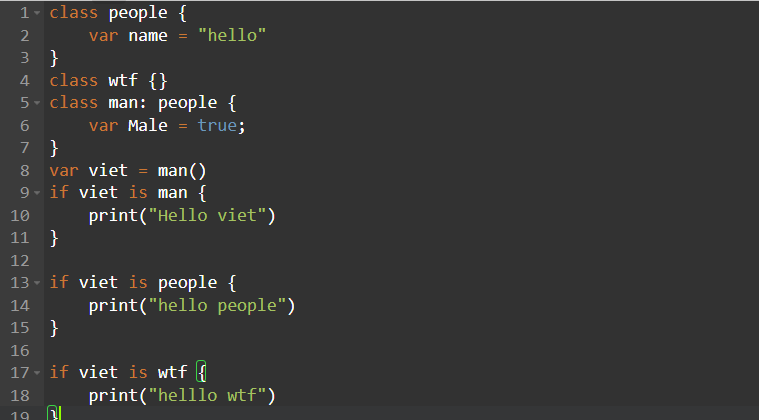
Chú ý 2: Ta thấy khi throw ra exception thì toàn bộ code bên dưới func vend() không được chạy nữa.

# 17. Type Casting

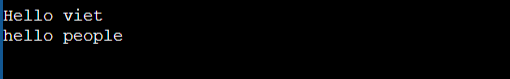
“Type casting is a way to check the type of an instance, or to treat that instance as a different superclass or subclass from somewhere else in its own class hierarchy.”

“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:



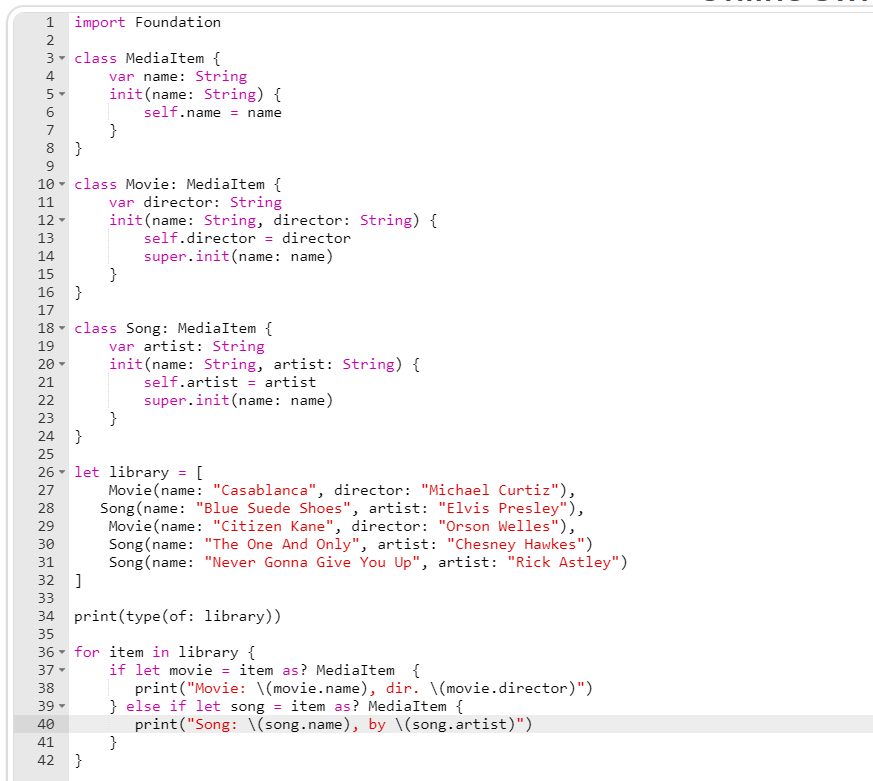
* DownCasting

“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.

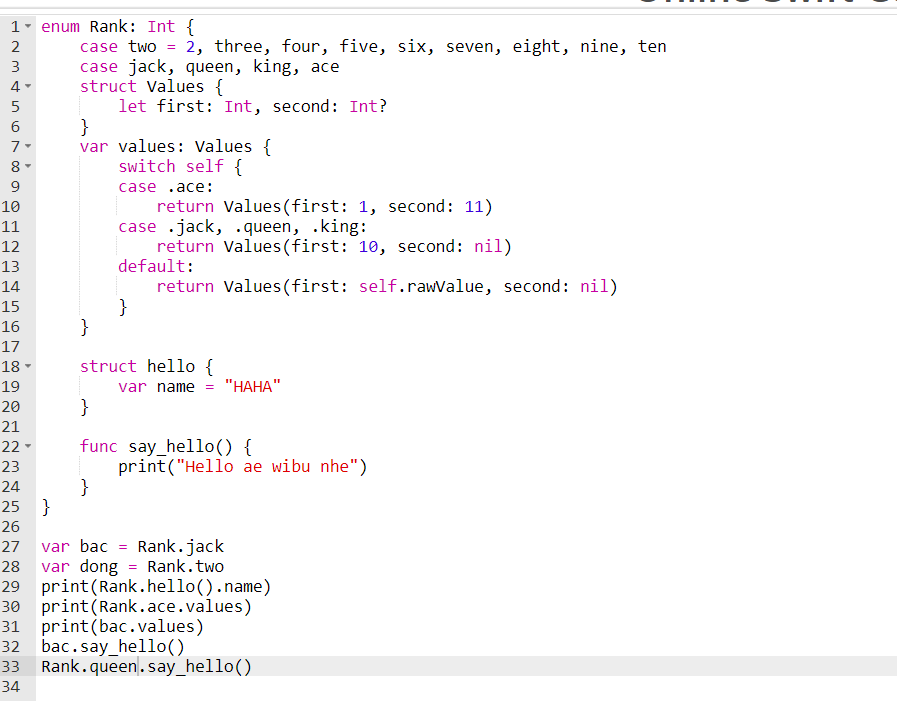
Use the forced form of the type cast operator (as!) only when you are sure that the downcast will always succeed. This form of the operator will trigger a runtime error if you try to downcast to an incorrect class type.”



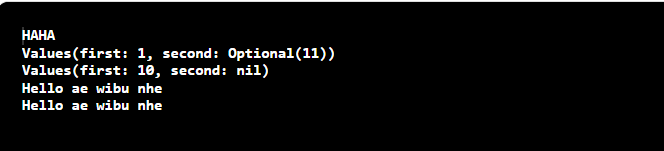
The example starts by trying to downcast the current item as a Movie. Because item is a MediaItem instance, it’s possible **that it might be a Movie; equally, it’s also possible that it might be a Song, or even just a base MediaItem. Because of this uncertainty,** the as? form of the type cast operator 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:

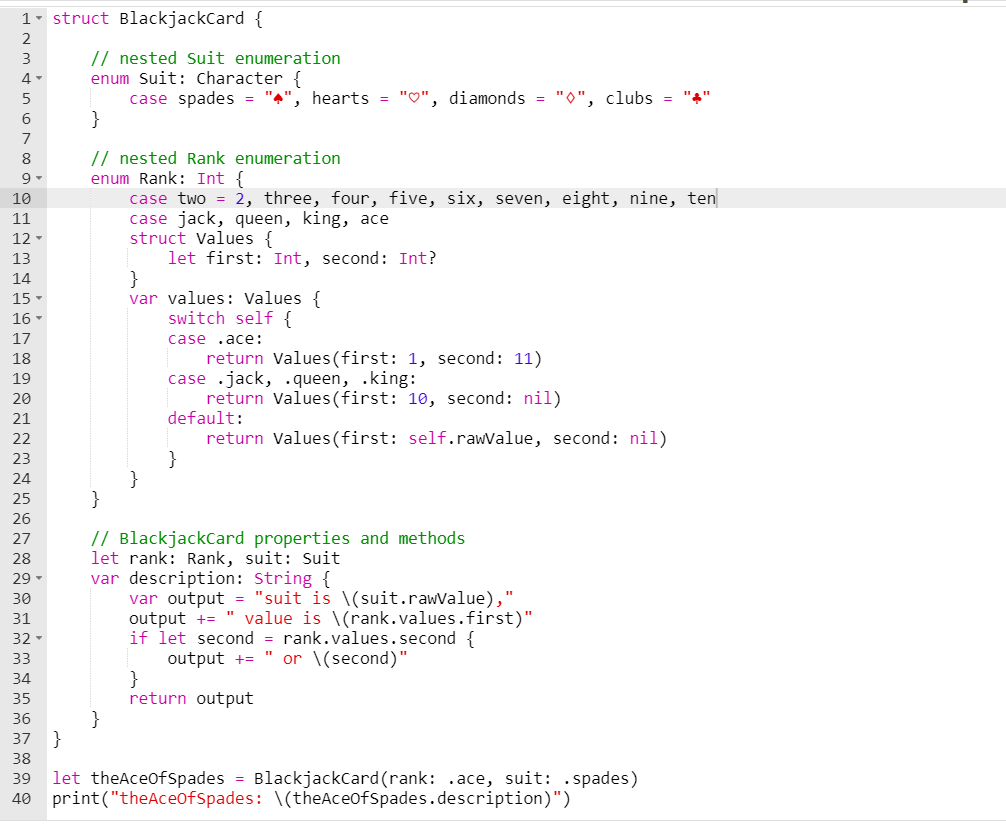


Output:



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

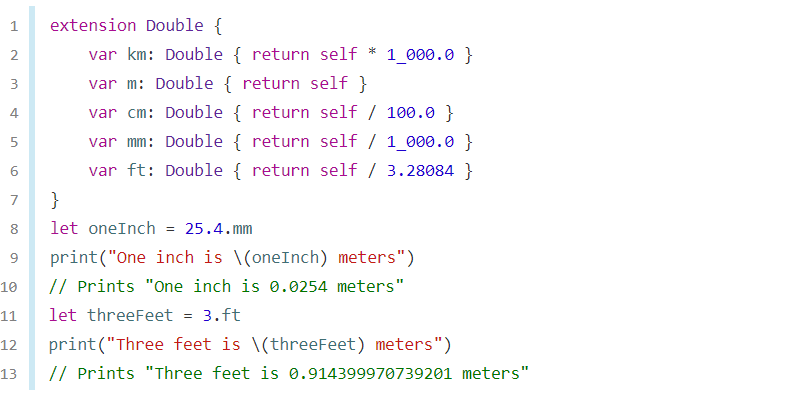
“Extensions **add new functionality** to an existing class, structure, enumeration, or protocol type. This includes the ability to extend types for which you don’t have access to the original source code (known as retroactive modeling).”

Extensions in Swift can:

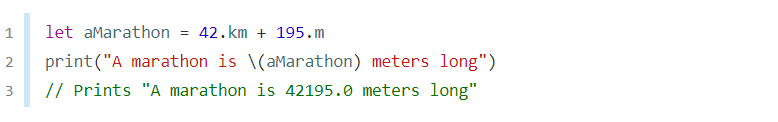
* Add computed instance properties and computed type properties
* Define instance methods and type methods
* Provide new initializers
* Define subscripts
* Define and use new nested types
* Make an existing type conform to a protocol

## 19.1. Computed Properties

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



VD2:



## 19.2. Initializers

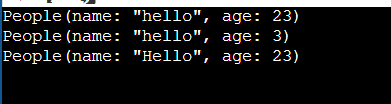
“Extensions can add new initializers to existing types. This enables you to extend other types to accept your own custom types as initializer parameters, or to provide additional initialization options that were not included as part of the type’s original implementation.”

“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.”

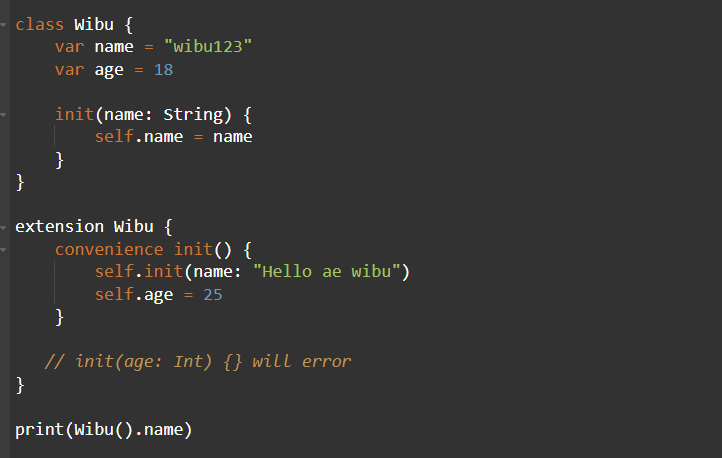
* Struct: Với Struct ta có thể thêm các hàm init() thông quá extension



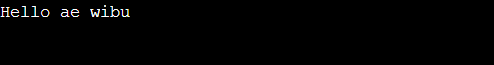
Output:



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

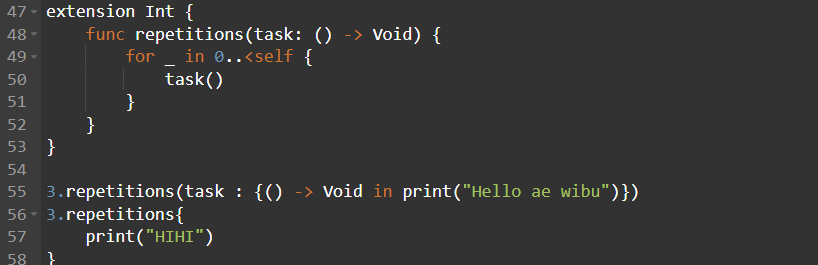


Output:

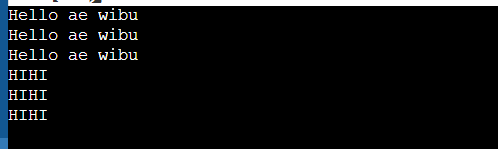


## 19.3. Methods

“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

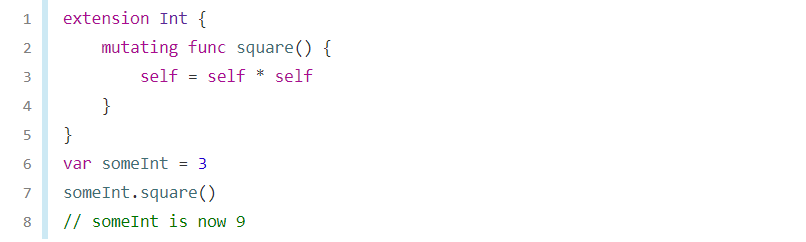


Ta đã add them method repetitions vào kiểu dữ liệu Int

## 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:



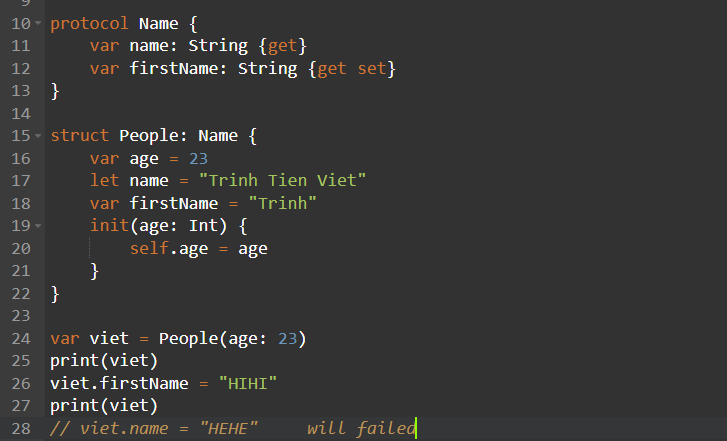
# 20. Protocol

“**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.”

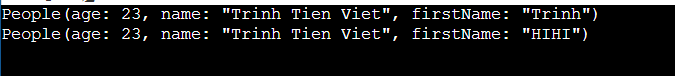
Giải thích khái niệm trên: Protocol nó chỉ define các **methods, properties, and other requirements** và nó hoàn toàn không triển khai các đắc điểm trên. Một class, struc hay enum adopted(Nghĩa là nhận nuôi: ý là đồng ý nhận nuôi các define của Protocol, sau đó triển khai các define đó).

## 20.1 Property Requirements

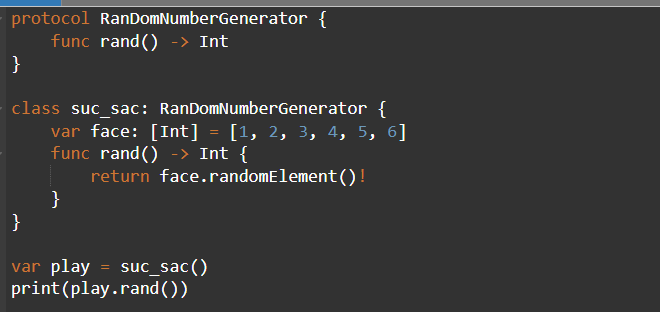
“Property requirements are always declared as variable properties**, prefixed with the var keyword.** Gettable and settable properties are indicated by writing { get set } after their type declaration, and gettable properties are indicated by writing { get }. “



Output:

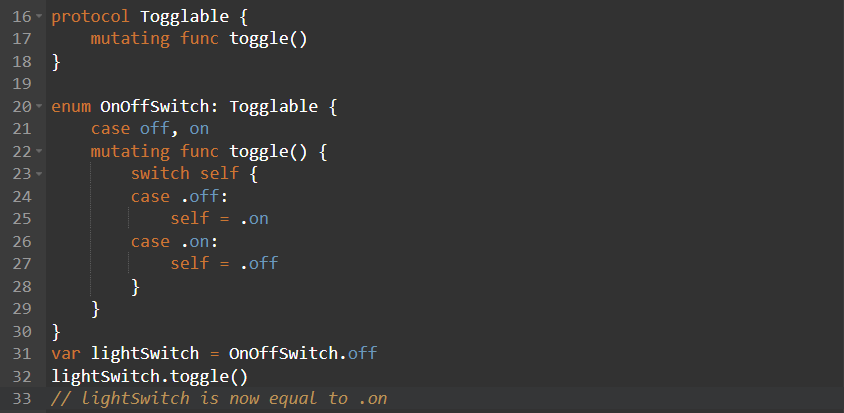


## 20.2 Method Requirement



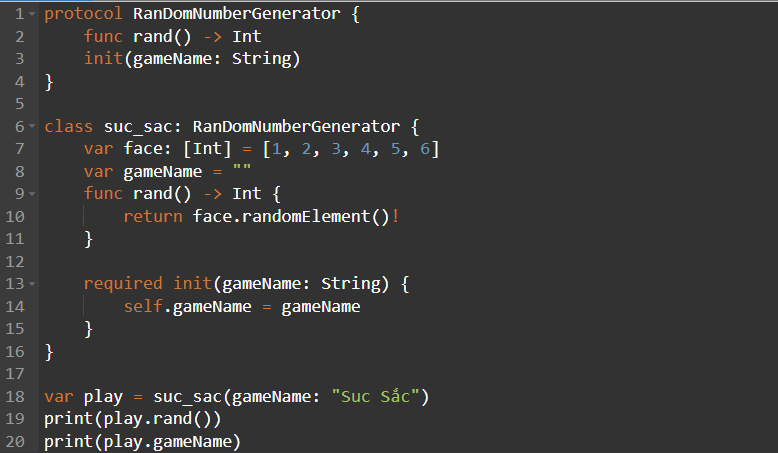
## 20.3 Mutating Protocol

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



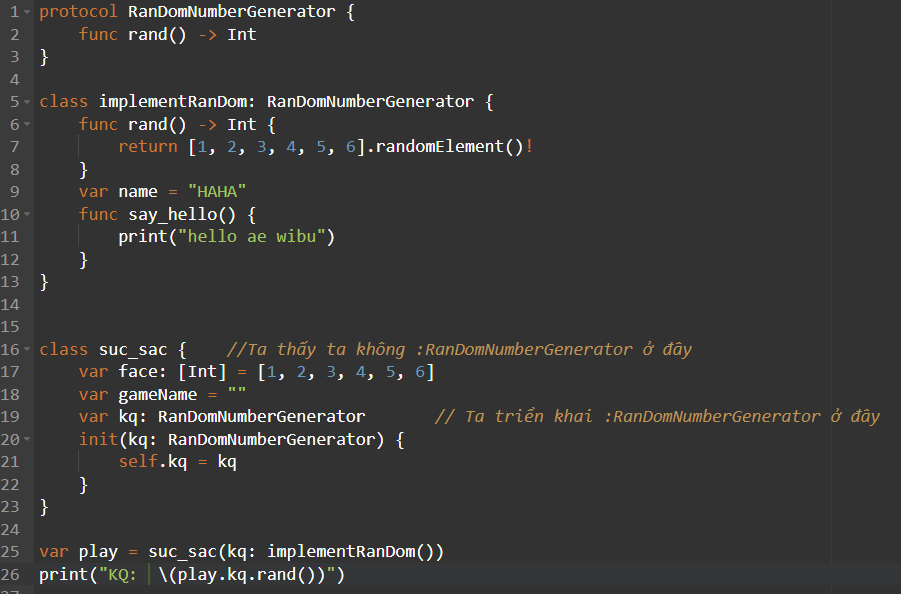
Output:



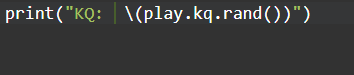
## 20.5 Protocol as Type

“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.

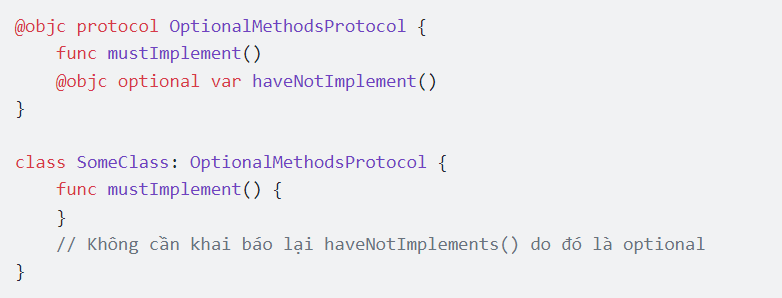
Ta thấy ở ví dụ trên, ta dung class implement để triển khai cho class suc\_sac không qua biến var kq. Tổng quát là ta sử dụng 1 class khác triển khai cho 1 class mong muốn. Như vậy gọi là Delegation. Delegate hoạt động bằng cách chuyển chức năng từ một class cơ sở sang một delegate class khác. Class 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

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.”

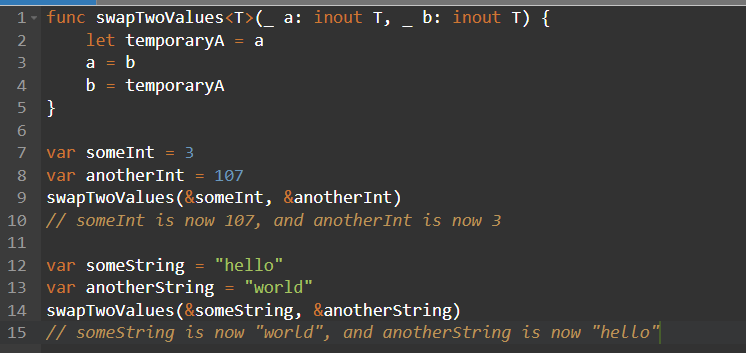
Vì 1 class có thể implement rất nhiều protocol và chúng nó có thể trùng tên với nhau hoặc bạn sẽ không phân biệt được function nào của protocol nào.



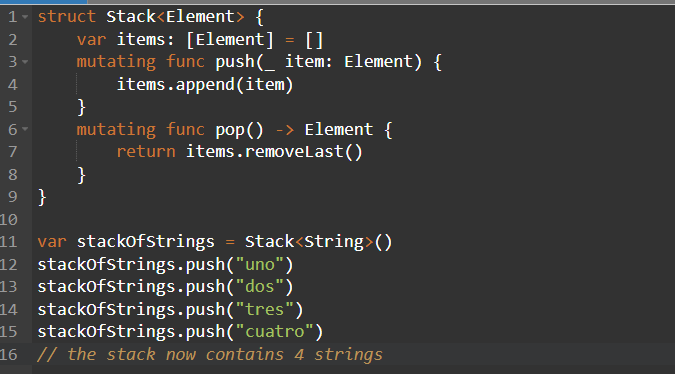
Output:



# 21. Generic



VD2:

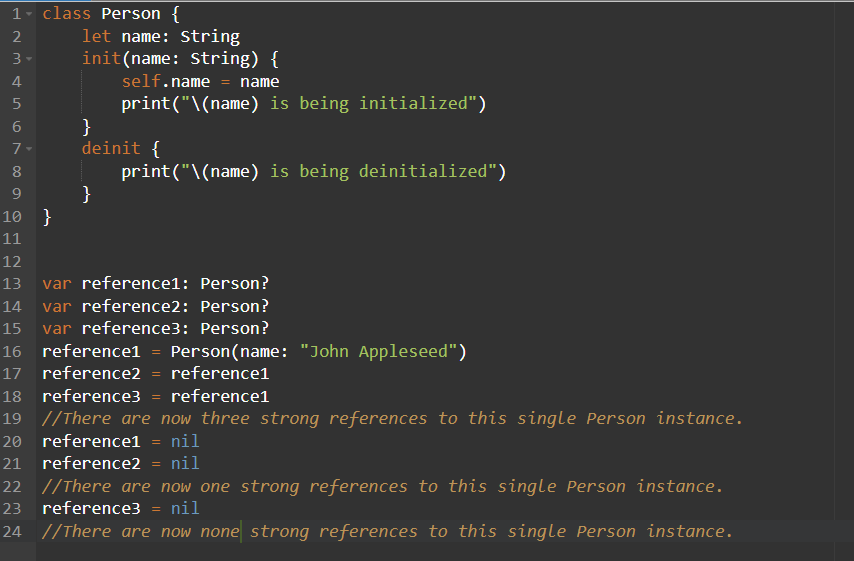


# 22 Automatic reference counting

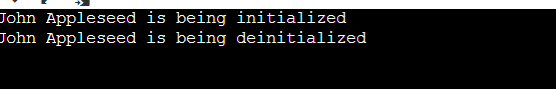
“Every time you create a new instance of a class, ARC allocates a chunk of memory to store information about that instance. This memory holds information about the type of the instance, together with the values of any stored properties associated with that instance.

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.”

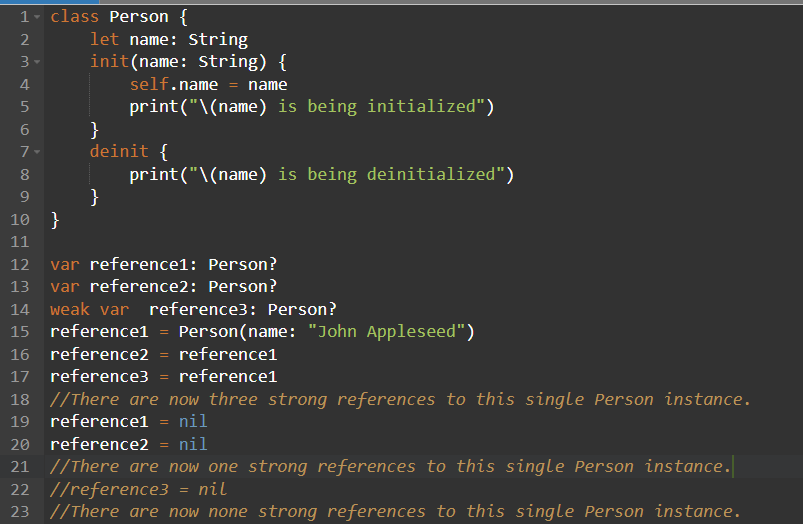


Output:

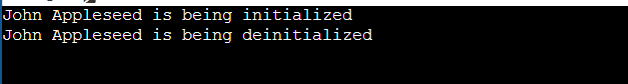


Ta thấy khi nào không có tham chiếu nào đến Person() nữa thì hàm deinit() mới được gọi, lúc đó vùng nhớ Person() chính thức bị ARC thu hồi.

## 22.1 Strong và weak

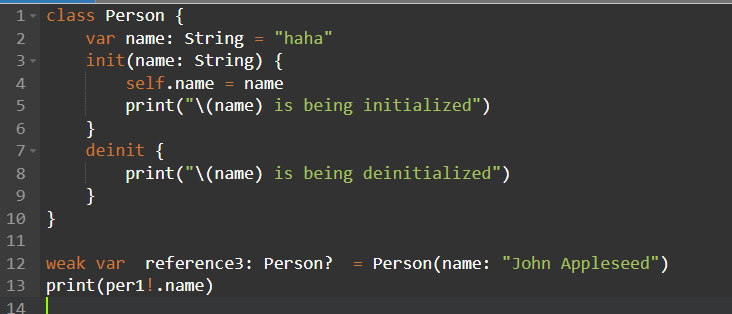


Output:

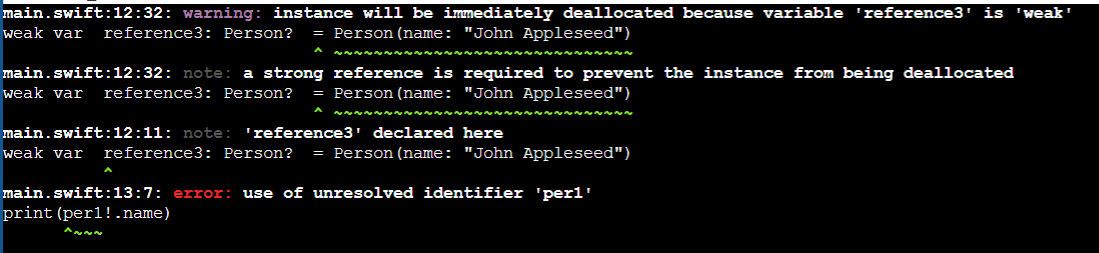


Ta thấy biến weak var referencen3 không cần gán = nil thì vùng nhớ Person() đã bị thu hồi rồi. Điều này muốn nói lên là khi khai báo với từ khóa weak thì kiểu refenrence này là kiểu tham chiếu yếu 😊))) Nghĩa là khi không còn thằng Strong nào tham chiếu đến vùng nhớ đó nữa thì vùng nhớ đó sẽ tự bị thu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

VD2:



Output:



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