Contents

[I. Các phương thức của Array 1](#_Toc125147651)

[1.1 Filter 1](#_Toc125147652)

[1.2 Enumrated 2](#_Toc125147653)

[1.3 Reduce 2](#_Toc125147654)

[1.4 Zip 3](#_Toc125147655)

[I. Các vấn đề của Class và Struct 4](#_Toc125147656)

[2.1 Nested type Function: 4](#_Toc125147657)

[1.2 Lazy 5](#_Toc125147658)

[2.3 Extension Init for struct 7](#_Toc125147659)

[1.3 Convenience Init 9](#_Toc125147660)

[II. Throw Error and Handle Error 10](#_Toc125147661)

[3.1 Basic Throw Error and Handle Error 10](#_Toc125147662)

[2.2 Advance Handle Error with Result<Success, Failure: Error> 13](#_Toc125147663)

[IV. Protocol 15](#_Toc125147664)

[4.1 Variable Type Protocol 15](#_Toc125147665)

[4.2 Generic cho Protocol 16](#_Toc125147666)

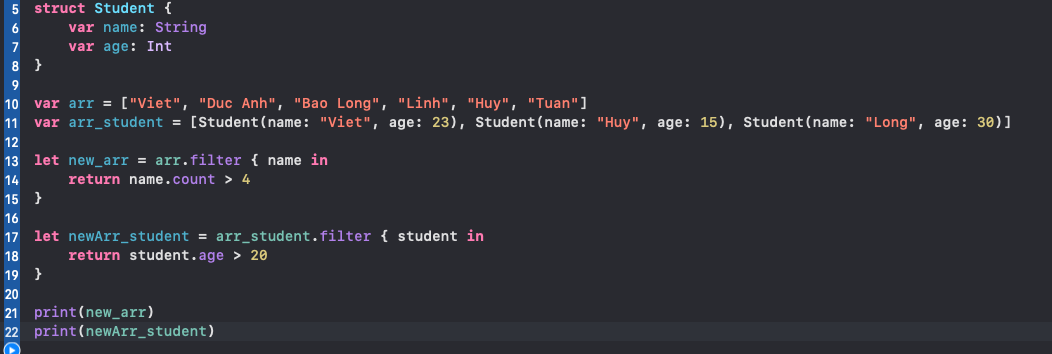
[4.3 Extension for Default Protocol 18](#_Toc125147667)

[V Generic 20](#_Toc125147668)

# I. Các phương thức của Array

## 1.1 Filter

Được sử dụng để lọc phần tử

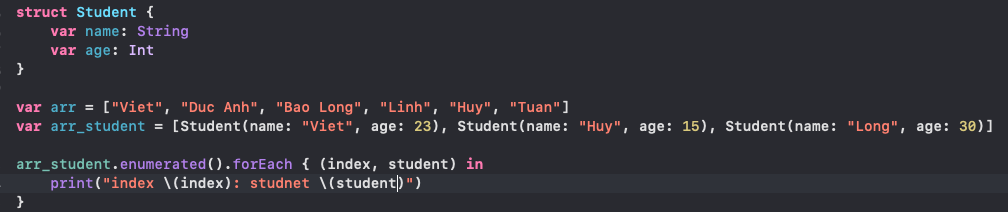


Output:

Không có mô tả.

## 1.2 Enumrated

Được sử dụng để truy suất vào phần tử và vị trí của chúng trong mảng:

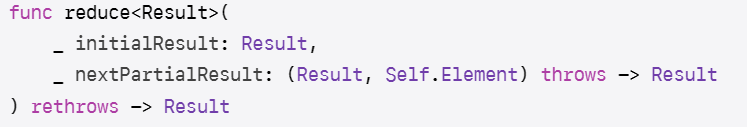


Output:

Không có mô tả.

## 1.3 Reduce

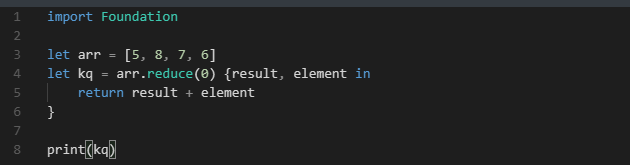
Phép xử lý **reduce** được sử dụng để kết hợp tất cả các phần tử trong một collection thành một kết quả đầu ra.



Với **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**

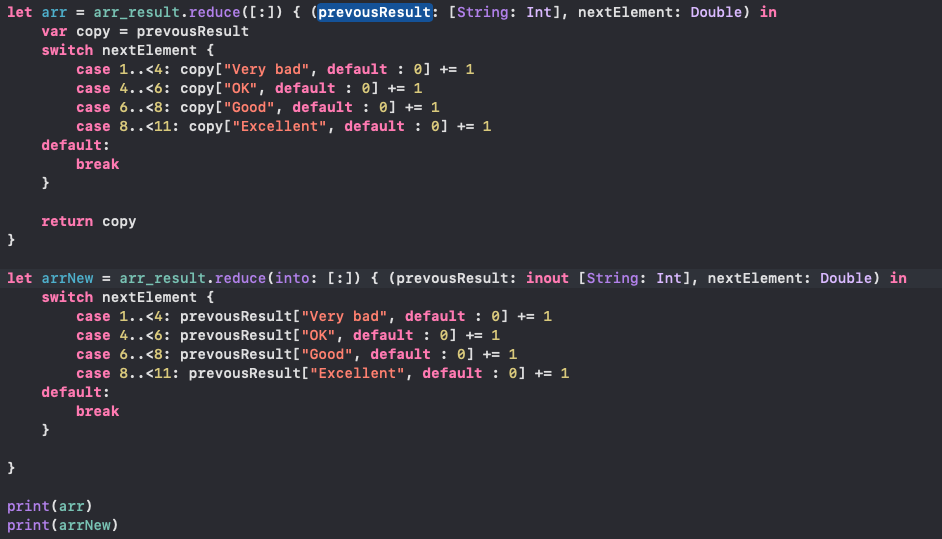
Xem ví dụ đơn giản :



Output:



Ví dụ phức tạp



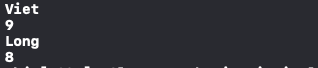
Ta có [:] là khởi tạo giá trị đầu tiên cho kết quả, ý ở đây là dictionary rỗng

## 1.4 Zip

Được sử dụng để ghép 2 mảng thành 1 tuples



Output:

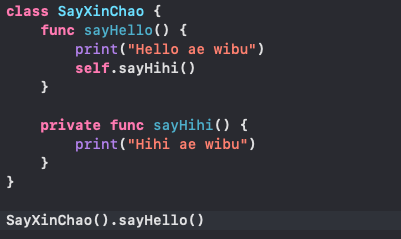


Ta thấy bên dưới chỉ có 2 phần tử nên chỉ ghép được 2 phần tử thoi

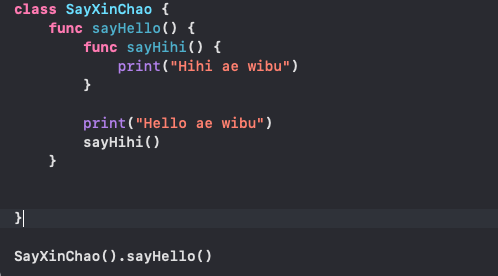
# Các vấn đề của Class và Struct

## 2.1 Nested type Function:

Trong 1 class hay struct, khi ta muốn 1 method chỉ được truy cập ở bên trong và không được phép truy cập từ bên ngoài class, ta có thể sử dụng từ khóa private:



Với 1 trường hợp khác, ta chỉ **muốn func sayHihi() chỉ được gọi bên trong fun sayHello(),** và bên ngoài Class cũng không thấy được, ta có thể sử dụng Nested Function:

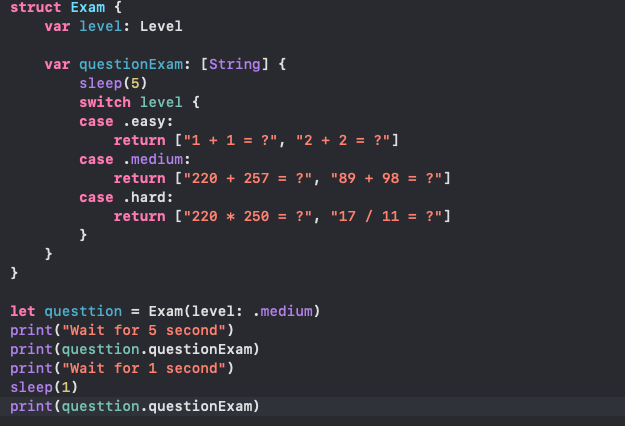


Output cả 2 đề giống nhau:

Không có mô tả.

## Lazy

Ta có một đoạn code như sau:



Ở đoạn code print(“Wait for 1 second”), khi chạy thực tế nó bị delay khoảng 6s, lý dó là vì sau khi sleep 1s ở lệnh dưới, thì chương trình tiếp tục bị sleep 5s khi truy cập tiếp vào properties questionExam.

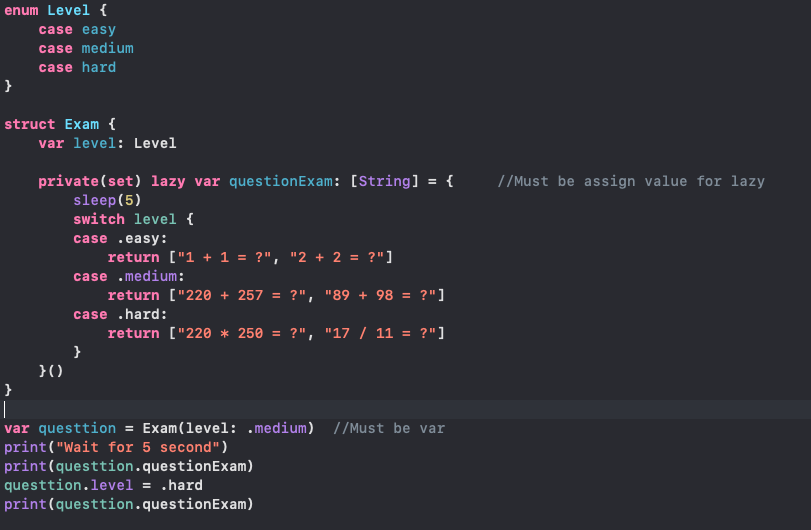
Ta thấy rằng việc này tương tự như việc call API**, nếu giá trị của 1 biến không đổi**, mà cứ khi nào gọi ta lại phải đợi fetch API hay sleep thì thật phiền phức để lấy lại giá trị ko đổi. Giải quyết vấn đề này ta sẽ sử dụng từ khóa lazy

Từ khóa lazy cho phép khởi tạo chỉ khởi tạo biến đó chỉ 1 lần và chỉ khi được gọi:



Lúc này chương trình phần print(“Wait for 1 second”) mới thực sự sleep 1s, bởi vì ở phần trên biến questionExam đã được gọi, và nó đã được khởi tạo lúc đó rồi, sau đó khi ta gọi lại lần nữa nó sẽ không thực thi phần code trong closure nền ko sleep 5s nữa.

Ta có thể sử dụng thêm 1 vài từ khóa nữa, giả sử ta không muốn user truy cập thay đổi trực tiếp giá trị questionExam đó nữa, ta sử dụng private(set) như sau:



Ta cũng đoán được thêm, cả khi question thay đổi thuộc tính level thì thì giá trị biến lazy cũng không thay đổi, vì nó có vào body closure nữa đâu:

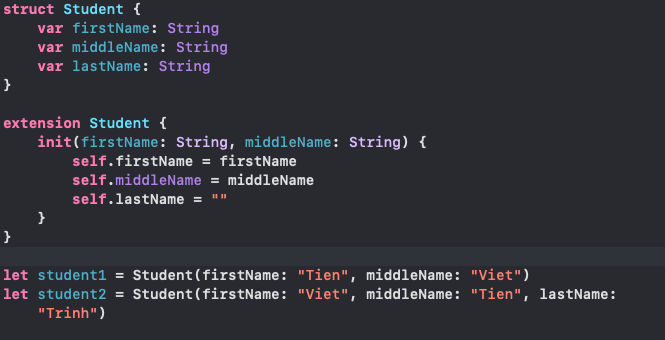
Không có mô tả.

## 2.3 Extension Init for struct

Ta có đoạn code lỗi sau:



Ta thấy khi khai báo 1 hàm init thì hàm init default cũng bị xóa, vậy làm thế nào để them 1 hàm init() mà vẫn giữ được default init(). Câu trả lời là sử dụng extension

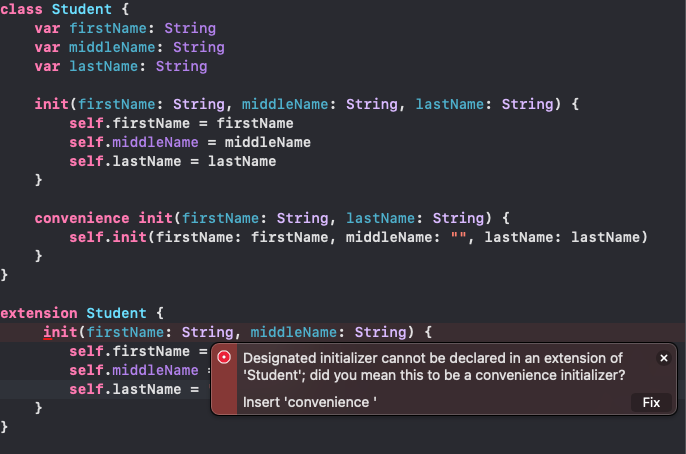


Ta thấy khi sử dụng extension để khai báo init thì default init() cũ của struct không bị xóa.

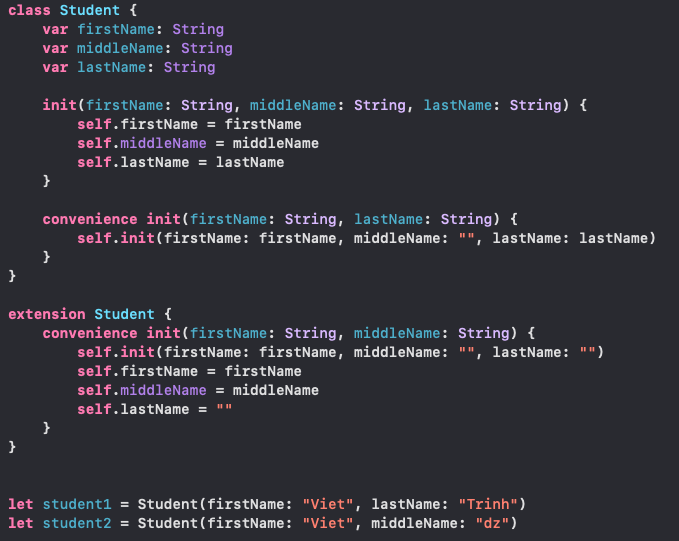
Lưu ý cái này chỉ áp dụng cho struct, bởi vì class không có default init.

## Convenience Init

Đối với Class, khi khai báo init trong extension, thì đó phải là Convenience Init()



Ta thấy chương trinhg đã báo lỗi, ta phải them convenience cho đúng:

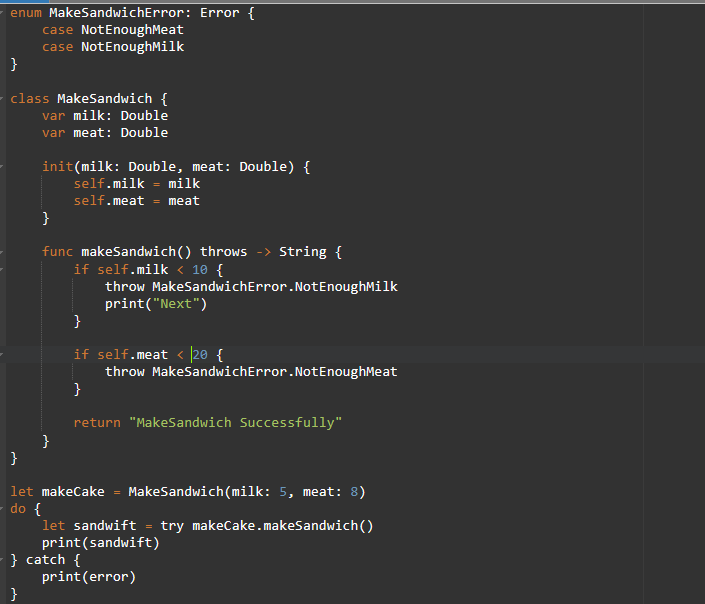


Ta có 1 Convenience Init luôn phải gọi hàm designated init() từ ban đầu

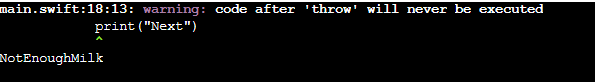
# Throw Error and Handle Error

## 3.1 Basic Throw Error and Handle Error

* Để có thể throw Error, ta sẽ khai báo 1 enum mà comform với protocol Error. This empty protocol indicates that a type can be used for error handling.

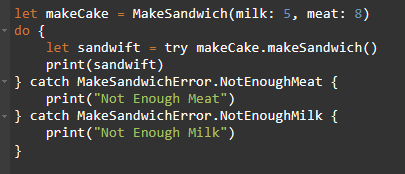


Output:



Ta cũng thấy thêm warning rằng code sau lệnh throw sẽ không được thực thi. Ta sử dụng do try cactch để cactch Error khi xảy ra.

* Ta cũng có thể catch cụ thể từng trường hợp error theo cách này

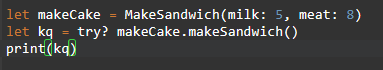


Output:

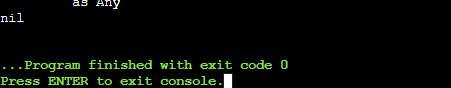


Với việc khai báo như trên, ta sẽ không sử dụng được biến chung chung là error nữa.

* Sometime chúng ta không quan tâm tới Error, ta không muốn catch chúng, ta có thể sử dụng try? Như sau



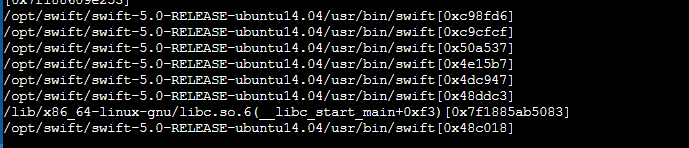
Output thu được:



* Đôi khi ta có thể đảm bảo rằng giá trị trả về không bao giờ là nil, ta dùng try!



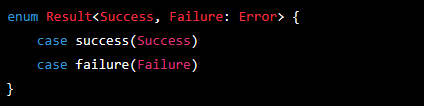
Lúc này kết quả thu được là dead code:



Bởi vì ta đã cố unwrap luôn kết quả trả về, mà kết quả lúc này lại là nil nên dead chương trình

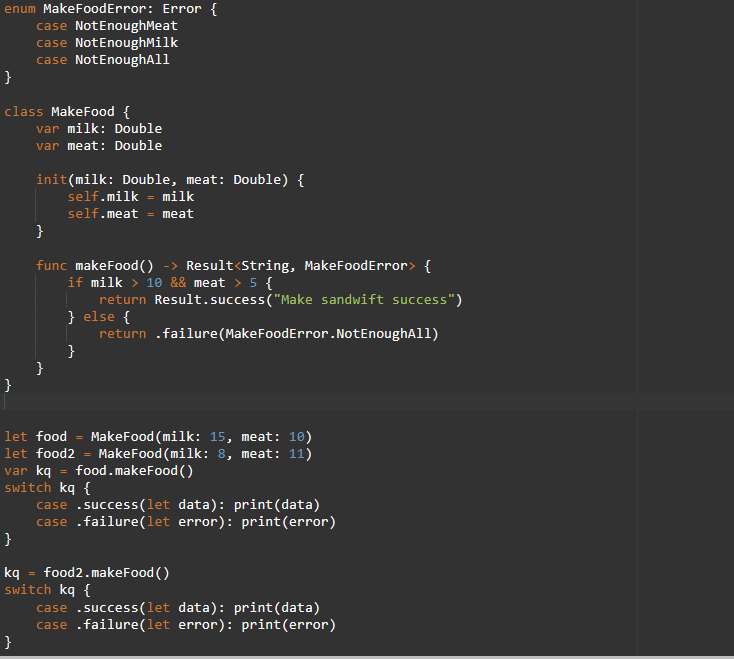
## Advance Handle Error with Result<Success, Failure: Error>

Result là 1 generic enum với 2 giá trị là success và failure . Nó được định nghĩa trong thư viện như sau



Result hoạt động cơ chế như throws nhưng có vài điểm quan trọng.

* Success là kiểu dữ liệu mong muốn nhận được khi success
* Failure là kiểu lỗi mà comform với type Error, nó như MakeSandwichError bên trên là comform với Error



Output:

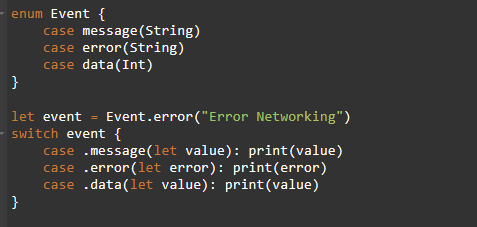


Phân tích:

Ta có func makeFood() return lại 1 Enum có kiểu Result<String, MakeFoodError>. Ta thấy MakeFoodError comform với protocol Error, nên truyền vào Result hợp lệ. Vì function makeFood() return lại kiểu Result, nên ta phải return lại 1 trong 2 case của nó là success hoặc failure. Ta lại chú ý case .success(Success), nó truyền vào kiểu dữ liệu Success, trong trường hợp này Success là String, nên ta truyền vào 1 String “Make Sandwift Success” vào success() là hoàn toàn hợp lý, tương tự cho error, ta truyền vào failure là hoàn toàn hợp lý.

Lúc này khi ta có: kq = food.makefood(), thì kq sẽ bằng Result.success(“Make Sandwift Success). Ta sử dụng switch để lấy dữ liệu truyền vào success và được như kết quả.

* Ôn lại:



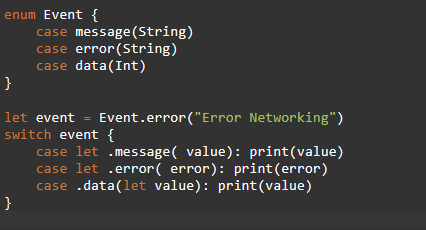
Ta thấy các case đều mang value là kiểu dữ liệu truyền vào, vậy làm thế nào để ta lấy được value từ các kiểu dữ liệu đó. Câu trả lời là sử dụng case let như trên để bóc tách dữ liệu

Output:



Ta cũng có thể sử dụng case let liền nhau để bóc tách multiple value

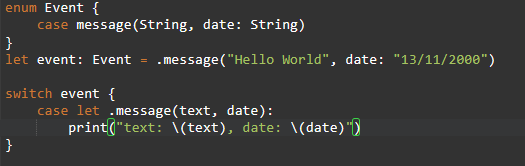
VD1:



Output:



VD2:



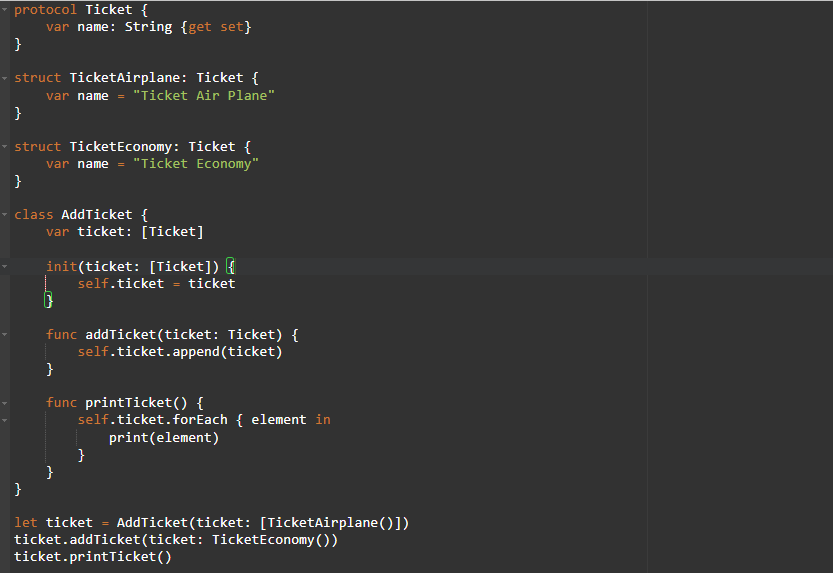
Output:



# IV. Protocol

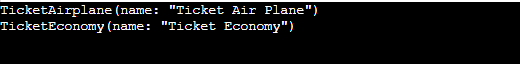
## 4.1 Variable Type Protocol

Ta có đoạn code sau:



Trong class AddTicket() ta có 1 biến ticket là type protocol Ticket. Ta có bất kì dữ liệu nào comform với protocol Ticket đều có thể append vào biến ticket.

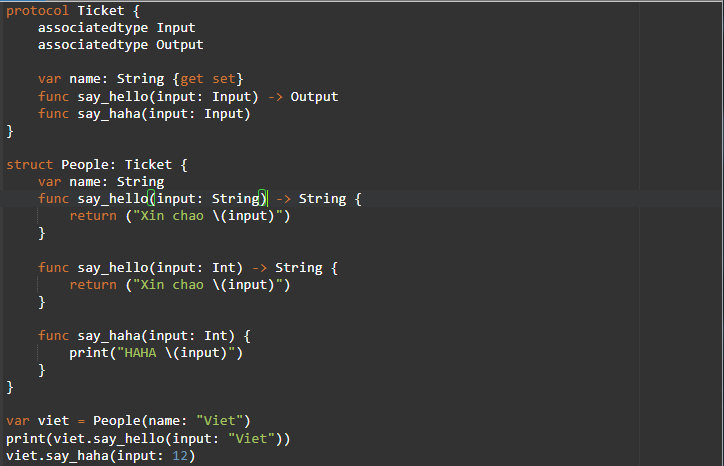
Output:



Ta thấy không cần biến ticket là kiểu any vẫn chứa được 2 instance của 2 class khác nhau.

## 4.2 Generic cho Protocol

Trong protocol ta có từ khóa **associatedtype** được sử dụng để định nghĩa 1 placeholder cho 1 kiểu dữ liệu đặc biệt mà sẽ được quyết định sau.



Output:

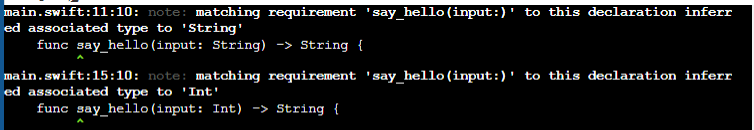


Ta sử dụng **associatedtype** để định nghĩa 2 placeholder Input và Output và sử dụng chúng trong func say\_hello(). Trong struc People, lúc này khi struct muốn dùng kiểu dữ liệu nào thì tự thay kiểu dữ liệu đó vào Input và Output.

Lưu ý ta không được làm như sau:



Output:

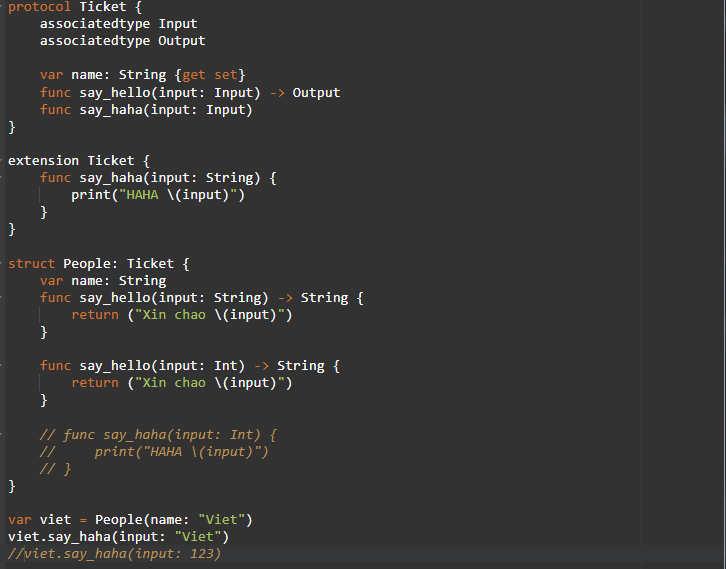


Sẽ xảy ra lỗi vì lúc này Input ở dunc say\_hello() sẽ không biết phải là kiểu dữ liệu là String hay Int.

## 4.3 Extension for Default Protocol

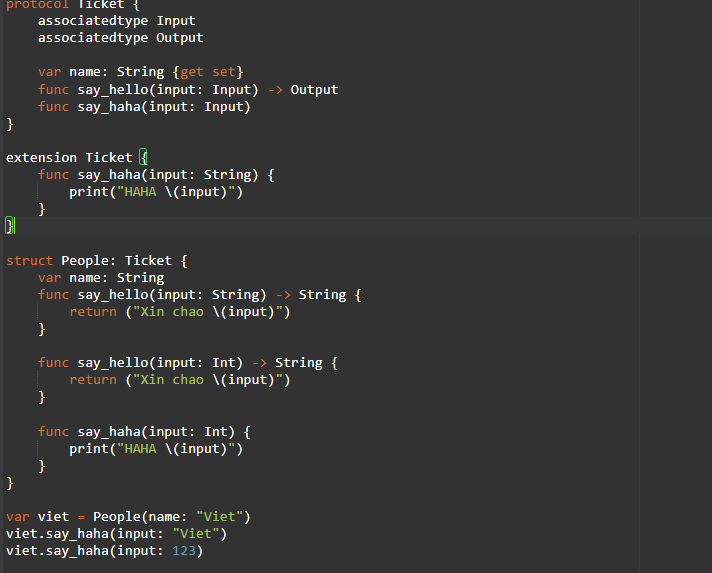
Ta thấy ở bài trước, bất kì class nào muốn conform 1 protocol đều phải triển khai các method, properties của Protocol. Như vậy với 1 protocol có 10 method và có 5 class muốn comform protocol đó thì ta triển khai 50 lần à.

Câu trả lời là sử dụng extension để khai báo default cho các thành phần bên trong protocol đó



Ta thấy People không cần triển khai method say\_haha() của Protocol Ticket. Bởi vì extension đã triển khai 1 default cho method say\_haha() protocol đó.

* Trong struct People, ta cũng có thể tự triển khai method say\_hello() và lúc này ta có thể sử dụng được cả 2 :

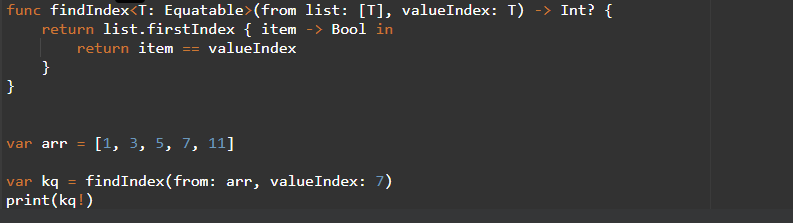


Output:



# V Generic

Ta có ví dụ sau:



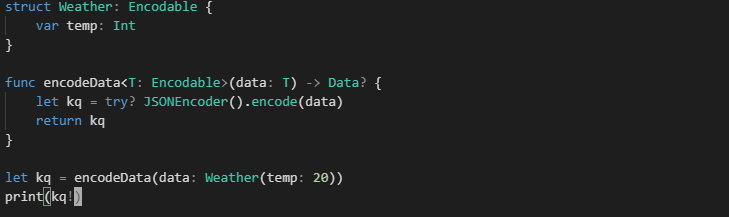
Output:



Phân tích:

* Để sử dụng Generic function, ta thêm <T>
* Khi ta truyền biến vào parameters của function generic đó, nó sẽ tự hiểu T là kiểu dữ liệu gì, trong đây T sẽ được hiểu là kiểu Int
* Ta có từ khóa **Equatable**. Tại sao lại cần từ khóa này? Ví do bởi vì chương trình đâu biết được kiểu dữ liệu này có thể so sánh hay ko, ta thêm từ khóa này để constrain người dùng phải truyền vào 1 kiểu dữ liệu có thể Equatable.
* Các từ khóa khác có thể được dùng là **Codable, Encodable, Decodable, Comparable,…**

VD2:



Output:



Ta thấy nếu ko them từ khóa Encodable ở T: Encodable thì sẽ sai, bởi vì chương trình đâu biết kiểu dữ liệu T có thể Encode được hay không mà thực hiện. Ở đây ta truyền 1 struct Weather mà đã comform với Encodable nên thành công

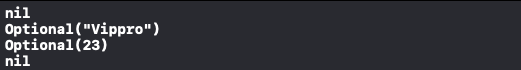
# VI. Declare let, var, lazy

* Chúng ta có 1 chân lý như sau: Với biến được khai báo là **let(let! Và let?)** , thì biến đó phải được gán default value hoặc phải được gán giá trị trong hàm **init().** Nhưng với biến **var? hay var!** thì chúng ko cần phải gán default value và cũng không cần phải gán giá trị cho chúng trong hàm **init().**

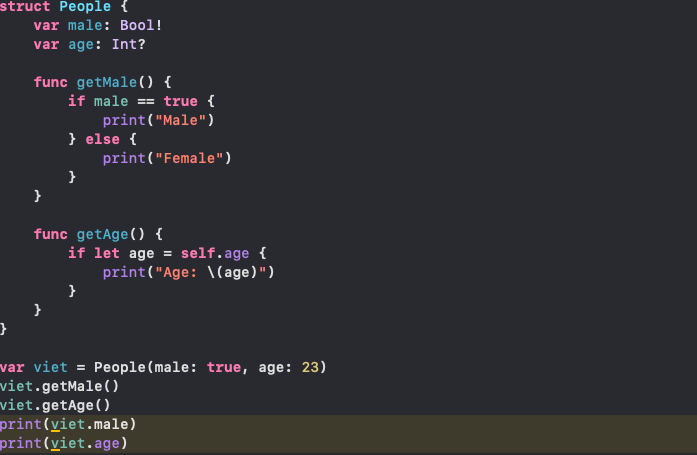


Ta thấy đó, trong hàm **init()** ta không cần phải gán giá trị cho biến **var! và var?** nhưng ta phải gán giá trị cho **let? Và let!**

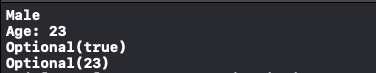
Output:



* Ta có biến **lazy** chỉ được khởi tạo khi nó được truy cập 🡪 Nên biến **lazy không thể đi với let.**
* Ta chỉ có lazy var button: UIButton = {}()
* Vậy thì điểm khác biệt giữa **var, var? và var!**
  + Biến **var? và var!** thì không cần được gán value trong hàm **init().** Còn biến **var** thì bắt buộc phải được assign value trong hàm **init()** (nếu chưa có default value)
  + Vậy thì điểm khác biệt giữa **var? và var!.** Như đã nói bên trên, khi ta không muốn gán giá trị cho chúng trong hàm **init()** thì có thể dùng **var? hoặc var!.** Điểm khác biệt giữa 2 biến này là biến **var?** thì khi sử dụng trong **class** ta phải unwrap nó**,** còn biến var! ở trong **class** thì không cần unwarpp(nó sẽ tự động được implicit unwrap). Nhưng chú ý khi đi ngoài class(ý là truy cập từ bên ngoài class) thì cả **var? và var!** đều phải được unwrap.



Output:



# VII. Xử lý bất đồng bộ IOS

Mỗi ứng dụng của IOS đều có 1 Main Thread, nhiệm vụ chính của Main Thread là thay đổi UI. Để tránh những công việc mất thời gian như request API phải xử lý trên Main Thread thì thì IOS cung cấp cho ta các thread khác gọi chung là Backgroud Thread.

* Queue: là 1 hàng đợi chứa những block code để đợi Thread thực thi, Queue có 2 loại chính
  + Serial Queue: Các block code ở đây được thực thi lần lượt
  + Concurrent Queue: Các block code ở đây thực thi ko theo thứ tự, ko cần đợi Task trước kết thúc để thực thi
* GCD: là 1 low level API được API cung cấp để quản lý Queue, công việc chính của GCD là:
  + Tạo Queue
  + Đưa các block code vào Queue
  + GCD cho phép ta tạo 2 Queue chính:
    - Là Main Queue: là 1 Serial Queue, các block code ở đây sẽ được đẩy vào **Main Thread** để thực thi
    - Global Queue: là Concurrent Queue: các Task trong này được đẩy vào bacgroud Thread

**Chú ý:** Ta có 1 điều quan trọng cần hiểu: **Main Thread** sẽ lấy các block code trong **Main Queue** để thực thi theo thứ tự**, Block code** nào vào trước sẽ được **Main Thread** thực thi trước. Thực ra đó là tính chất của tất cả Serial Queue

* Ta có 2 khái niệm mới đó là **sync và async:** 
  + Sync: Thực thi Task(Task là block code) một cách đồng bộ với hàng đợi hiện tại. Điều này có nghĩa là gì ? Giả dụ ta đang chạy trên Main Thread, ta khai báo 1 Thread khác, sau đó chạy thread đó với sync**, thì thread đó sẽ block Main Thread để thực thi, sau khi thực thi xong thì sẽ tiếp tục chạy Main Thread.**
  + Async: Cái này thì tự hiểu 😊))). Nó sẽ không block Main Thread, mà cả 2 Task sẽ chạy song song với nhau.

## 7.1 Thực hành

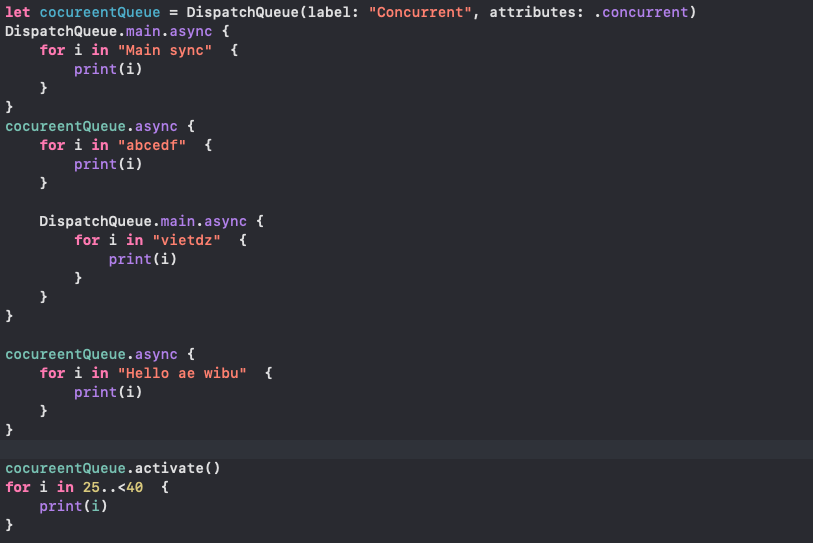
Không có mô tả.

Khi khởi tạo 1 Queue, ở phần **attributes** ta có 2 option là **concurrent và initiallyInactive.**

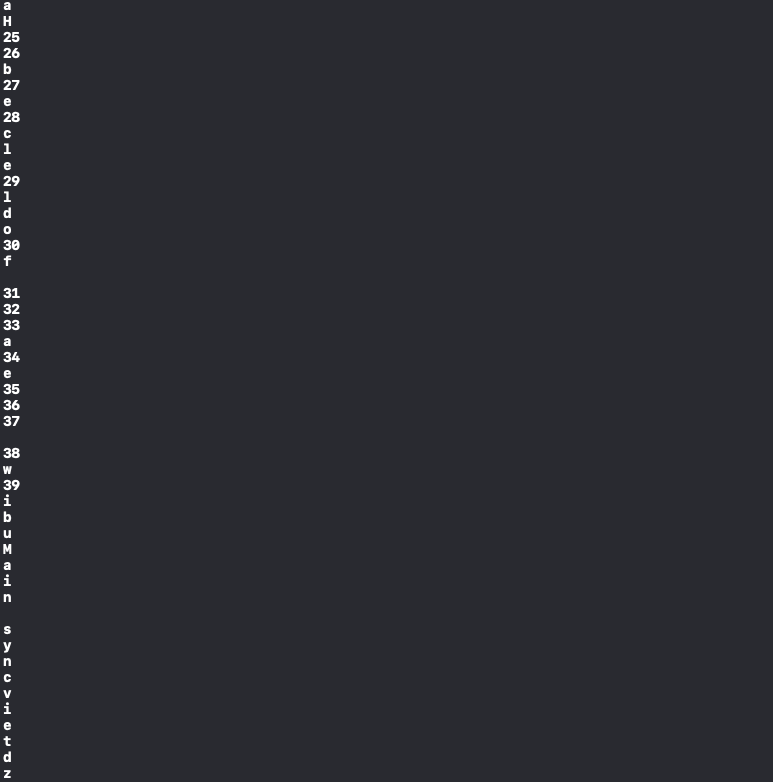
* Với **concurrent** thì queue đó hiển nhiên thuộc loại concurrent 😊)))
* Với **initiallyInactive** thì khi khởi tạo queue đó chưa được **active,** ta phải **activate().** Và queue đó là Serial Queue
* Với không có attributes thì Queue đó là SerialQueue.

Ta phải làm rõ lại 1 lần nữa 2 khái niệm sau:

* **Sync, Async** là sử dụng giữa các Thread với nhau. **Sync** sẽ **block Thread nó đang đứng và thực thi đoạn code của mình, sau khi thực thi đoạn code đó xong sẽ unblock Main Thread. Còn Async sẽ không block Thread nó đang đứng**
* SerialQueue và Concurrent là sử dụng với các block code trong Queue đó. Với SerialQueue thì các block code trong đó được thực thi lần lượt, còn Concurrent thì ko theo thứ tự. Hmm **nói thế này thì chưa rõ và còn quá đơn giản vấn đề.** Ta sẽ giải thích với ví dụ này:
  + Ta có Main Thread và 1 Thread A thuộc loại concurrentQueue. Trong concurrenQueue ta có 2 block code print(“1”) và print(“2”).
    - Với Thread A chạy với sync. Thì nó sẽ block Main Thread và thực thi 2 block code print() bên trong, và 2 block code này được in theo kiểu concurrent, nghĩa là in 1 và 2 lần lộn chứ không phải in hết 1 mới in 2(vì nó là kiểu concurrentQueue). Khi thực thi **hết** block code trong Thread A thì Main Thread mới tiếp tục được thực thi.
    - Với Thread A chạy với async. Thì cả Main Thread và Block code trong Thread A sẽ chạy đồng thời với nhau.
    - Nếu Thread A là SerialQueue thì sẽ print(“1”) xong mới print(“2”)
* Ta có 1 hàm quen thuộc: **DispatchQueue.Main.asyn(print(“Hello”)),** ý nghĩa của đoạn code này là gì.
  + Đơn giản thôi, hàm này sẽ thêm block code print(“Hello”) vào đít Main Queue. Sau khi các block trước đó của Main Queue được thực thi hết bởi Main Thread thì print(“Hello”) mới được thực hiện.
  + Chú ý: lệnh này ko có ý nghĩa bảo Main Thread hãy thực thi block code này ngay lập tức. Hmm ta thường thấy block code này được thực thi ngay lập tức vì các block code trước đó đã được thực thi hết rồi.
* Ta có 1 câu văn xuôi: Thực thi hết Block code, thực thi xong Block là thế nào? Thực thi hết Block code là Thread chạy được xuống dòng code cuối cùng block code đó, như thế là thực thi xong. Cái này quan trong vì dễ gây nhầm: Ta có block code có mỗi URLSession.shared.dataTask().resume(), bản chất là chạy đến lệnh resume() **là xong rồi và nếu có sync thì code ở ngoài Block đã được unblock và đã được thực thi**, còn việc trả về kết quả trong closure completion là câu chuyện của Thread khác trả về.
* VD1: Với Queue là ConcurrentQueue



Output:



Giải thích:

Ta thấy đó, “Main Sync” không được in ra ngay đầu tiên, bởi vì nó được add vào Main Queue, và phải chờ Main thread thực thi nó.

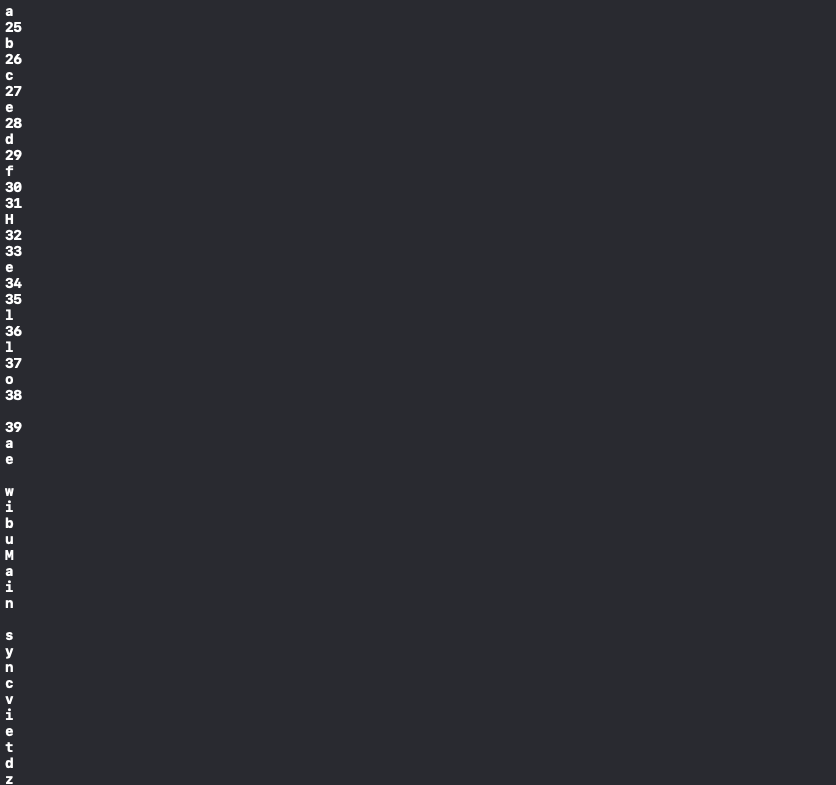
Ta cũng thấy luôn vì Queue ta tạo ra là concurrentQueue, nên nó thực thi song song với Main Thread, kết quả là in được lẫn lộn.

Ta cũng thấy luôn trong concurrentQueue, ta có add 1 block code print “vietdz” vào Main Queue, và vì nó được add sau print(“Main sync”) ở đầu nên nó được thực thi sau, và được kết quả như trên.

* VD2: Với Queue là Serial Queue



Output:



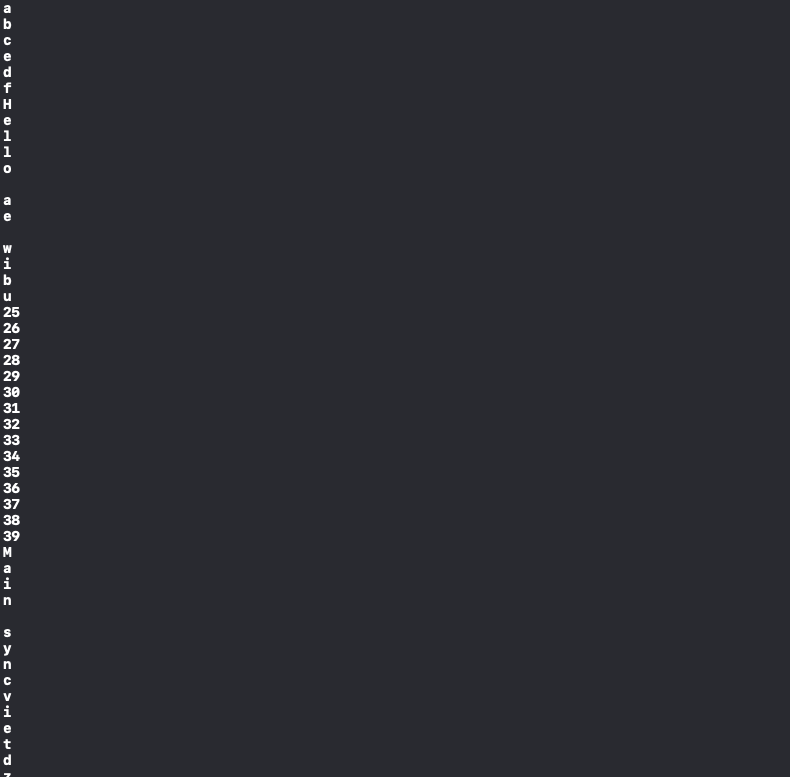
Lần này vì nó là serialQueue, nên đoạn code print(“Hello ae wibu”) và print(“abcdef”) không được chạy đồng thời với nhau. Mà đoạn code print(“abcdef”) chạy xong thì print(“Hello ae wibu”) mới được chạy

Và vì cả 2 đoạn này là async, nên chúng chạy song song với Main Thread.

* VD3:



Ouput:

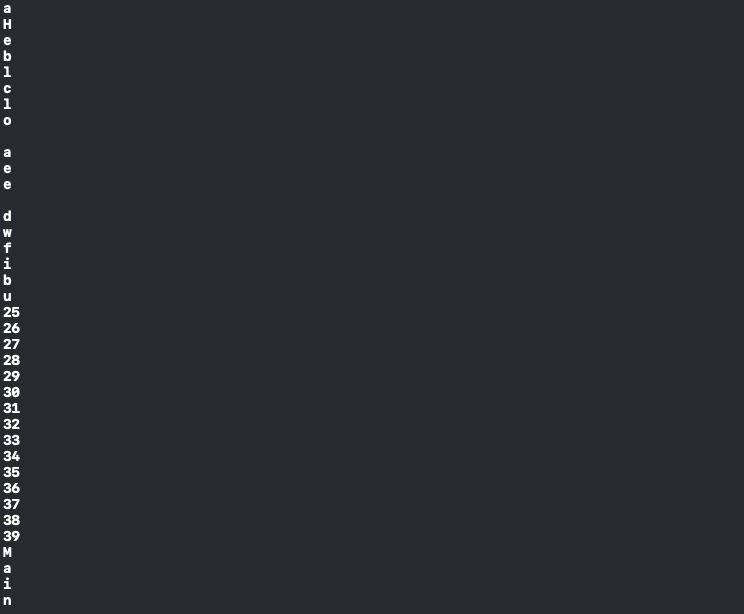


Doạn **Async** đầu sẽ không block Main Thread, nên đoạn code sync sau sẽ được add vào ConcurrentQueue, đến đoạn sync thì sẽ Block Main Thread.

* VD4:



Output:



Tóm Lại:

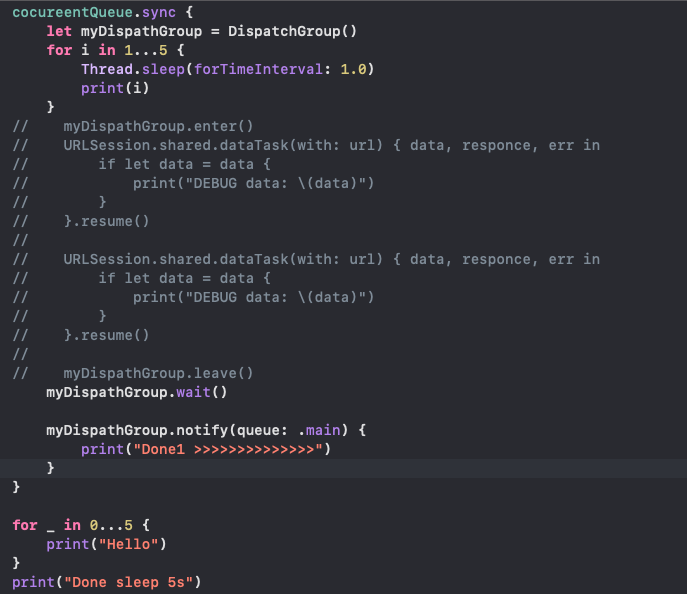
* Cứ gặp sync thì Thread đang đứng bị block để thực thi Thread sync đó, Thread sync đó thực thi xong thì unblock đang đứng
* Gặp async thì sẽ không block Thread đang đứng, mà sẽ chạy xong xong với nhau
* Concurrent thì cho phép các block code trong Queue chạy song song.
* Serial Queue thì các Block code trong đó chạy lần lượt.
* **Ta không bao giờ được dùng: DispathQueue.Main.sync()** bởi vì nó sẽ gây crash chương trình, lý do bởi vì khi đi với sync, nó sẽ block Thread hiện tại, mà Thread hiện tai là Thread Main, nên Thread Main bị block để thực thi đoạn code trong Main.sync, mà bản chất code trong đó được thực thi bởi Main Thread, mà Main Thread bị block bên trên rồi nên đel được thực thi nữa 😊)))) Nên gây ra hiện tượng DeadLock.

## 7.2 DispatchGroup

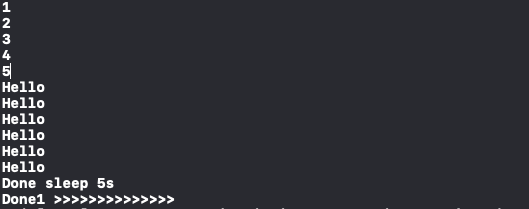
Dispatch group cho phép nhóm nhiều task vụ với nhau và chờ cho chúng hoàn thành hoặc nhận thông báo khi chúng hoàn thành mới thực hiện tiếp login của bạn. Những task vụ này có thể chạy đồng bộ hoặc bất đồng bộ và có thể chạy trên nhiều queue khác nhau. Ví dụ: khi open 1 màn hình, bạn cần phải call nhiều api để lấy dữ liệu, sau khi hoàn thành call tất cả api này mới update data lên giao diện.

Dispatch group cung cấp hàm wait() , hàm này block thread hiện tại cho tới khi các task vụ trong group được hoàn thành.

VD1: Với sync



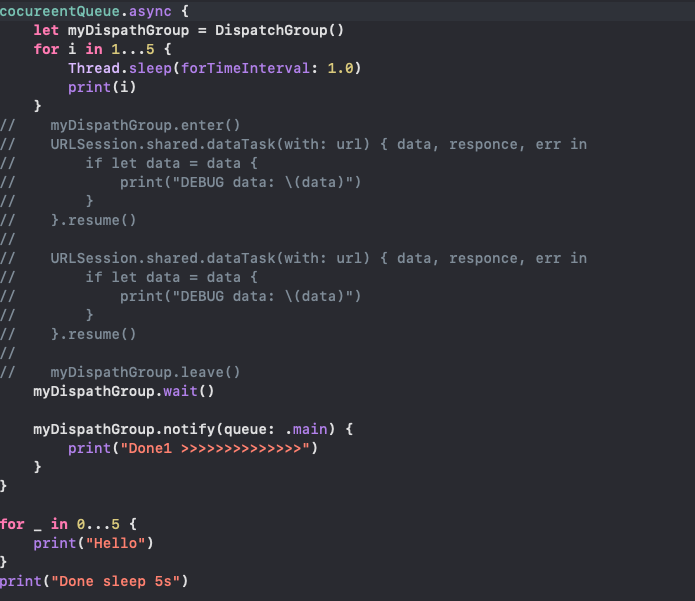
Output:



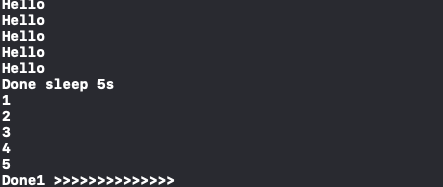
Giải thích:

* .enter(): là bắt đầu nhập đoạn code 1 vào Group
* .leave(): Là kết thúc việc nhập đoạn code 1 vào Group
* .notify(): Sau khi **tất cả Task**  trong **myDispathGroup** thực thi xong thì sẽ gửi **1 block code** là closure đến **Main Queue.** Ta thấy sau khi đoạn code Main Queue trước đó là print(“Done sleep 5s”) thực thi xong thì sẽ chạy đoạn code mới được đẩy vào là print(“Done1 >>>>>>>>>>>>> “)
* .wait(): Sẽ block Thread hiện tại cho đến kho hoàn thành xong hết việc mới cho chạy tiếp Thread đó, Thread hiên tại trong trường hợp này là Concurrent đấy.

VD2: Với async:

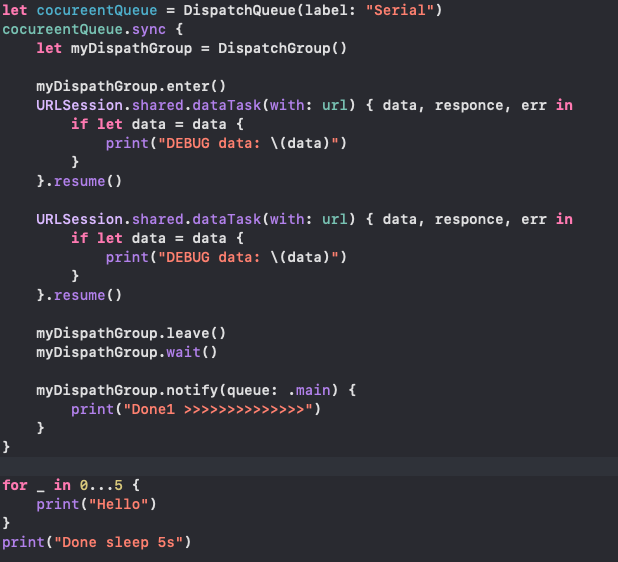


Output:

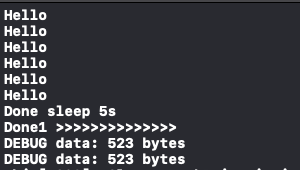


VD3:

* Ta có 1 câu văn xuôi: Thực thi hết Block code, thực thi xong Block là thế nào? Thực thi hết Block code là Thread chạy được xuống dòng code cuối cùng block code đó, như thế là thực thi xong. Cái này quan trong vì dễ gây nhầm: Ta có block code có mỗi URLSession.shared.dataTask().resume(), bản chất là chạy đến lệnh resume() **là xong rồi và nếu có sync thì code ở ngoài Block đã được unblock và đã được thực thi**, còn việc trả về kết quả trong closure completion là câu chuyện của Thread khác trả về. Xem ví dụ:



Output:



Đấy, ta thấy kết quả trả về Debug Data ở cuối cùng, vì bản chất nó đoạn closure completion là câu truyện khác, đoạn code trong Queue đã được chạy hết(chạy đến dòng cuối cùng) nên đoạn kết quả trả về đel liên quan gì.