Thứ 2 10/01/2022

NOTE

I. Protocols & extensions

1. Protocols

- Protocols là 1 thiết kế mô tả 1 kiểu nên có các property hoặc method như thế nào. Nếu như kiểu nào mà muốn conform 1 protocol thì bắt buộc phải áp dụng các method và property được định nghĩa trên protocol đó để thoả mãn điều kiện của protocol.
- Ví dụ:

```
protocol Identifiable {
    var id: String { get set }
}
```

- Quy luật:
  + Trong 1 protocol mà ta muốn định nghĩa 1 property thì phải định nghĩa read & write hoặc read-only
  + Property bắt buộc phải là var
  + Với method thì ta chỉ cần khai báo method với kiểu trả về chứ không cần phải có phần thân do method trong này chỉ là điều kiện.
- Protocol không thể nào được khởi tạo thành 1

giá trị giống như các kiểu thông thường nhưng có thể dùng làm như 1 kiểu. Các công dụng:

+ Dùng làm tham số để chỉ nhận các kiểu nào mà conform cho protocol.
+ Dùng làm 1 delegate để thực hiện công việc uỷ quyền (Cho các class nào mà đã conform cho protocol).

- Ví dụ:

```
protocol Cleaning {
    func clean()
}

struct Man {
    var delegate: Cleaning?
}

struct Roomba: Cleaning {
    func clean() {
        //vacuuming the house
    }
}

var luke = Man()
let r2d2 = Roomba()
luke.delegate = r2d2
luke.delegate?.clean() //Turn on the roomba.
```

- Protocol hoạt động giống như 1 kiểu kế thừa của class với các điểm khác biệt:

- Khác nhau:
    + Ta có thể conform protocol cho nhiều kiểu khác nhau không chỉ với class như Struct, Enum.
    + Các method và property trong protocol được thiết kế để cho các kiểu nào mà conform thì sẽ để cho kiểu đó tự định nghĩa các method và property đó theo ý thích chỉ giới hạn cách khai báo và kiểu trả về.
- Giống nhau:
    + Giống như kế thừa của class ta có thể dùng protocol giống như 1 superclass mà có thể dùng cho mọi class mà conform protocol đó (với giới hạn là chỉ dùng được các thành phần bên trong protocol).

2. Protocol inheritance
    - Kế thừa của protocol hoạt động giống như kiểu của class với 1 điểm khác biệt: protocol có thể kế thừa nhiều hơn 1 protocol.
    - Ví dụ:

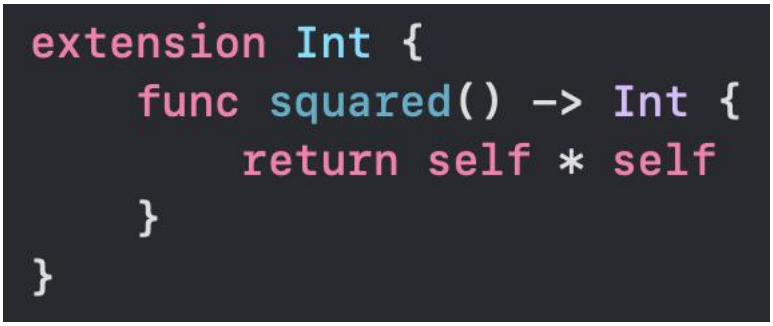
```
protocol Employee: Payable, NeedsTraining, HasVacation { }
```

    - Chức năng này giúp ta phân mảnh các protocol

thành các thành phần cho dễ tái sử dụng lại. Chức năng này cũng giúp cho logic của ta hợp lý hơn khi xây dựng protocol.

3. Extensions

- Extension là 1 cách để xây dựng thêm tính năng cho các kiểu sẵn có.
- Ví dụ:



- Chú ý:
  + Bên trong extension không cho phép định nghĩa property kiểu dữ liệu mới. Swift chỉ cho phép property kiểu computed.
- Extension cũng cho phép ta conform protocol bên trong đó.
- Ví dụ:

```
protocol Lumberjack {
    func chopWood()
}

struct Woody {
    let name = "Woody"
}
extension Woody: Lumberjack {
    func chopWood() {
        //Chopping wood
    }
}
```

- Extension thường được dùng để chia các thành phần của 1 kiểu thành nhiều phần khác nhau cho dễ đọc code hơn.
- Extension có thể dùng để thêm các tính năng mà cho các kiểu mà ta không sở hữu.

4. Protocol Extensions
    - Ta có thể dùng extension để cho protocol có thể định nghĩa method/property riêng cho protocol giống như 1 kiểu.
    - Sử dụng chức năng này giúp ta giảm thiểu viết các code thường dùng do ta có thể chia sẻ các chức năng trong extension trong protocol cho các kiểu nào mà conform cho nó.

- Các thành phần bên trong extension là không bắt buộc.

5. Protocol-oriented programming
    - Ta có thể áp dụng thêm để cho protocol có thể có method mặc định nếu như mà kiểu đang cho protocol không có định nghĩa method đó.
    - Chức năng này giúp cho ta có thể chia sẻ code giữa nhiều kiểu khác nhau và hạn chế lặp lại code.

II. Optionals

1. Handling missing data
    - Kiểu Optional trong swift tượng trưng cho giá trị có thể nil (không tồn tại).
2. Unwrapping optionals
    - Để có thể sử dụng giá trị của optional ta phải unwrap thành phần bên trong optional.
    - Cách thường dùng để unwrap 1 optional là sử dụng conditional binding (if-let) mà cho phép ta tạo 1 biến mới với giá trị của optional và bỏ qua nếu giá trị là nil.
    - Ví dụ:

```
if let unwrapped = name {
    print("\(unwrapped.count) letters")
} else {
    print("Missing name")
}
```

3. Unwrapping with guard
    - 1 cách khác để unwrap optional 1 cách an toàn là sử dụng 1 guard-let.
    - Guard-let khác với if-let là nếu giá trị là optional thì guard-let bắt buộc ta phải đi ra khỏi scope hiện tại. Còn với if let thì cho phép ta bỏ qua phần bên trong if-let.
    - Guard let thường dùng ở phần đầu của các phương thức để kiểm tra tất cả các giá trị.
4. Force unwrapping
    - Swift cho phép ta unwrap giá trị mà không cần phải xử lý optional sử dụng dấu !.
    - Cách này thường chỉ sử dụng nếu như biết chắc 100% giá trị sẽ không bao giờ bị nil.
5. Implicitly unwrapped optionals
    - Swift cho phép ta định nghĩa 1 kiểu optional mà đã unwrap sẵn. Ví dụ:

```
let age: Int! = nil
```

- Cách này thường dùng cho các @IBOutlet do gía trị của các thành phần UI thường trước khi sử dụng là nil, tới lúc mà trước khi sử dụng như ở viewDidLoad() thì đã có giá trị và không bao giờ bị thay đổi thành nil.

6. Nil coalescing
   - Ta cũng có thể unwrap 1 cách an toàn bằng cách cho 1 giá trị mặc định nếu giá trị là nil sử dụng toán tử ??.
   - Ta có thể nối nhiêù toán tử này lại với nhau.
   - Ví dụ: first() ?? second ?? “”
   - Nếu mà phương thức first() trả về giá trị là nil thì kế tiếp sẽ chạy phương thức second() và nếu second() cũng trả về nil thì cuối cùng sẽ trả về giá trị “”.
7. Optional chaining
   - Ta có thể truy cập nhiều thành phần khác nhau của 1 object mà optional bằng cách sử dụng optional chaining. Khi ta gọi 1 property hay method của 1 object mà optional ta có thể gọi

tắt bằng cách nối đuôi bằng dấu ?.

- Ví dụ:

```
let names = ["John", "George", "Paul", "Ringo"]
let beatle = names.first?.uppercased()
```

  + Nếu trong lúc đang đi xuống mà bất cứ thành phần nào trong chuỗi là nil thì sẽ ngừng đi xuống và trả về giá trị là nil.
  + Nếu ta đang truy cập 1 giá trị thì giá trị sẽ được bọc bởi 1 optional.

- Optional chaining thường được dùng chung với nil coalescing để đi xuống nhiều tầng của 1 object và có 1 giá trị đề phòng 1 trong các thành phần là optional.

8. Optional try

- Ta có thể thực hiện các method mà throw sử dụng try? và try!
- Try? được sử dụng như safe safe unwrap (?) mà sẽ trả về nil nếu như mà có lỗi. Tuy nhiên sẽ không truy cập được lỗi.
- Cách này thường sử dụng để thực hiện 1 method mà chỉ quan tâm xem method có bị lỗi không và không quan tâm đến lỗi.

- Try! Giống như force unwrap chỉ sử dụng cho method mà biết rõ sẽ không bao giờ bị lỗi.

9. Failable initializers
    - Ta cũng có thể sử dụng optional để cho khi khởi tạo 1 kiểu. Để sử dụng ta phải nối ? sau lệnh init? Trả về nil cho điều kiện sai.
    - Nếu như mà ta sử dụng chức năng này thì kiểu object mà được tạo là sẽ được để bên trong 1 optional.
    - Chức năng này thường được sử dụng để quản lý xem object có khởi tạo đúng tiêu chuẩn không.
10. Type casting
    - Trong swift ta có thể đổi 1 kiểu thành kiểu khác nếu kiểu mà ta định đổi liên quan đến kiểu ta sắp đổi.
    - Giống như optional ta có thể ép kiểu (type cast) sử dụng safe downcast hoặc forced downcast. 2 lệnh này thường dùng để đi xuống từ 1 superclass xuống 1 class mà subclass nó.
    - Nếu ta dùng lệnh as (upcast) thường dùng để đi từ 1 child class đi lên super class.