Thứ 5 06/01/2022

Note

1. Closures phần 1:

a) Creating basic closures:

- Phương thức trong swift cũng là 1 kiểu. Ta có thể gán 1 hằng/biến là 1 phương thức và để gọi thực hiện phương thức thì ta phải thêm 2 dấu ngoặc đằng sau hằng/biến giống như gọi phương thức thông thường. Ví dụ:

    - let stat = { print(“This is a closure”) }
    - stat()

b) Accepting parameters in a closures:

- Giống như phương thức thông thường với closure ta cũng có thể truyền dữ liệu vào bên trong. Lệnh (in) dùng để tách biệt phần tham số và phần thân của closure. Ví dụ:

    - let method = { (entry: String) in
    - print(“Go \(entry)”) }

c) Returning values from a closures:

- Cũng giống như phương thức thông thường ta có thể thực hiện để cho closures trả về giá trị. Ví dụ:

    - let method = { (entry: String) → String in
    - return “Go \(entry)” }

d) Closures as parameters:

- Vì closures có thể được gán cho hằng/biến nên ta có thể đặt closures làm tham số cho các phương thức.
- Cú pháp cho kiểu của tham số chia làm 2 phần:
    - Phần tham số: () bên trong sẽ gọi kiểu của từng tham số mà closures này gọi.
    - Phần giá trị trả về: → và nối theo là kiểu của giá trị trả về
    - Ví dụ: (String)→Void là 1 kiểu closures có 1 tham số kiểu String và không có giá trị trả về

e) Trailing closures syntax:

- Nếu ta gọi 1 phương thức mà có tham số cuối cùng là 1 closures thì ta có thể bỏ closures đó nối sau phương thức mà không cần bên trong 2 dấu ngoặc của phương thức.
- Nếu phương thức đó chỉ có 1 tham số là kiểu closures thì ta có thể loại bỏ 2 dấu ngoặc của phương thức để chứa tham số.
- Ví dụ:

```
travel { ///The parentheses can be ommitted if there is only a
    single parameter and it's a closure.
    print("I'm driving my car")
}
```

2. Closures phần 2:

a) Using closures as parameters when they accept parameters:

- Ta có thể sử dụng tham số cho closures giống như tham số bình thường của phương thức với 1 điểm khác biệt. Ta không thể đặt label cho tên của tham số của closures do closures là 1 kiểu phương thức không có tên.
- Sử dụng tham số cho closures:

```
func travel(action: (String)->Void ) {
    print("I'm getting ready to go")
    action("London") ///Give the closure value
    print("I have arrived")
}

travel { (place) in
    print("I'm driving to \(place)") ///Calling the value
}
```

- Ở trong ví dụ trên ta sử dụng tham số cho phương thức travel là closure mà nhận 1 tham số kiểu chuỗi và trả về không.
- Kế tiếp ở trong phương thức travel ta cho giá trị London vào bên trong closures.
- Cuối cùng ở hàng dưới ta gọi phương thức travel và thực hiện cho closures in 1 chuỗi sử dụng với giá trị London mà đưa vào từ bên travel.

- Ta có thể đặt tên cho tham số của closures mà không cần khai báo kiểu do Swift sẽ tự suy ra kiểu của tham số.

b) Using closures as parameters when they return values:

- Thực hiện trả về giá trị từ closures:

```
func travel(action: (String)->String) {
    print("I'm getting ready to move")
    let description = action("London")
    print(description)
    print("I have arrived")
}

travel { (place: String) in
    return ("I'm traveling to \(place)")
}
```

c) Shorthand parameter names:

- Các tham số của closure trong Swift có thể gọi bằng tên viết tắt mà Swift tự động tạo. Cú pháp $0, $1, … tham số đầu tiên của closures đếm bắt đầu từ 0.
- Sử dụng chức năng này ta có thể loại bỏ dòng in khai báo tham số.
- Chức năng này thường dùng nếu closure mà đang gọi chỉ gọi các tham số 1, 2 lần. Nếu cần gọi các tham số của closure nhiều lần thì tốt nhất là đặt tên cho các tham số để không bị nhầm lẫn.

d) Closures with multiple parameters:

- Nếu 1 closures mà có nhiều hơn 1 tham số thì ta tách chúng ra bằng , ở phần khai báo kiểu của chúng. Ví dụ:

    - let closure = (String, Int) → String

e) Returning closures from functions:

- Phương thức cũng có thể trả về giá trị là kiểu closure.
- Ta cũng có thể thực hiện gọi closures mà phương thức trả về trong cùng 1 dòng. Ví dụ:

- func method() → (String) → Void { }
- method()(“Hello”)

f) Capturing values:

- Do closures được thiết kế là 1 phương thức mà ta có thể chuyền đi khắp nơi và thực hiện nên closures cần phải thực hiện được ở mọi nơi. Do vậy nên closures thuộc kiểu con trỏ.
- Closures cần phải trỏ đến các giá trị bên trong nó nên đó là lý do mà closures bắt lại giá trị.
- Tính năng này của closures là cần thiết để giúp cho Swift đảm bảo cho code bên trong closures luôn được thực hiện.

3. Struct phần 1:

a) Creating your own structs:

- Structs trong swift là kiểu dữ liệu thường được dùng nhiều nhất do chúng nhẹ dễ tạo, xử lý và là kiểu gía trị nên chúng cũng dễ dàng bị loại bỏ khi cần. Struct cũng là 1 kiểu bất biến nên chúng thường an toàn khi sử dụng.
- Struct là giống như 1 kiểu Tuples mà có tên. Ta có thể gọi một struct mà có access level là internal ở mọi nơi trong 1 project so với 1 Tuples mà chỉ có thể gọi ở trong 1 file.
- Quy luật sử dụng: Khi nào mà ta cần gộp nhiều giá trị có kiểu khác nhau mà cần gọi nhiều lần trong 1 project thì ta sẽ dùng đến Struct, còn nếu mà ta chỉ cần gọi 1, 2 lần thì tốt nhất vẫn dùng Tuples.

b) Computed property:

- So với kiểu property thông thường của 1 Structs mà ta bắt buộc phải init giá trị hoặc cho giá trị mặc định. Computed property có thể sử dụng giá trị của các property khác bên trong Structs do giá trị này thuộc kiểu lazy.
- Kiểu computed property có 2 phần writer và read (get & set).
- Getter sẽ thực hiện công việc đưa dữ liệu cho property.
- Setter sẽ cho phép thực hiện công việc sau khi mà property đã có giá trị.
- 1 Computed property có thể chỉ có getter (read-only) nhưng nếu ta muốn sử dụng setter thì ta bắt buộc phải có getter.
- Computed property thường được dùng để theo dõi tình trạng của 1 hay nhiều property mà được liên tục cập nhật giá trị.

c) Property observer:

- Property observer là chức năng của mỗi property giúp ta thực hiện công việc trước khi hay sau khi ta đặt giá trị cho property.
- DidSet sẽ được gọi sau khi mà ta thực hiện đặt giá trị với giá trị đó sẽ là 1 hằng tên newValue.
- WillSet ngược lại sẽ gọi trước khi mà giá trị sẽ được đặt với giá trị cũ tên là oldValue.
- Property observer không bao giờ được chạy khi lúc bắt đầu đặt giá trị cho property.

d) Methods:

- Methods là 1 kiểu phương thức mà thuộc về 1 kiểu. Khác biệt giữa method và phương thức thông thường là method có thể gọi tất cả các property của Struct để sử dụng. Tuy nhiên chỉ có thể gọi method của từng kiểu sử dụng chính kiểu ấy.
- Methods là 1 cách để chuyển giá trị có access level private sang nơi khác bằng cách cho method trả về giá trị đó (Kiểu giống như read-only).

e) Mutating methods:

- Do tính bất biến của Struct nên để đảm bảo an toàn Swift bắt buộc mọi method nào mà thực hiện thay đổi property trong Structs bắt buộc phải có mác mutating. Mác này dùng làm cho người lập trình để ý rằng đang thực hiện thay đổi cho property.
- Nếu ta mà gọi 1 method mà có gọi 1 method mutating thì ta phải gắn mác mutating cho cả 2.
- Swift không cho phép 1 thực thể kiểu hằng gọi 1 mutating method và cũng không cho phép 1 method thay đổi 1 property kiểu let.

f) Property & methods of strings:

- Kiểu chuỗi là 1 kiểu dữ liệu cơ bản của Swift dạng Struct mà có thể chứa nhiều ký tự lại thành dạng mảng các ký tự.
- Kiểu chuỗi là dạng Struct thay vì là ký tự để cho Swift thực hiện nhiều việc sử dụng method và property cho làm việc với chuỗi 1 cách thuận tiện hơn.
- Các method và property hay sử dụng: .count, .isEmpty, .upperCased(), .sorted(), .firstIndexOf() …

- Khi ta muốn kiểm tra 1 chuỗi nào mà trống thì ta sử dụng property .isEmpty do 1 chuỗi vẫn luôn có count > 0 cho dù chuỗi đó trống.
- Swift hạn chế sử dụng subscript sử dụng Int và Range mà bắt buộc sử dụng String.Index do các ký tự đặc biệt như emoji mà làm cho chuỗi có độ dài khác nhau.

g) Property & methods of arrays:

- Mảng cũng là 1 dạng dữ liệu cơ bản mà ta thường dùng dạng Structs.
- Các method và property hay sử dụng: .append(_/contensOf:), .remove, .first, .isEmpty, .sorted, .map, …
- Mảng và chuỗi khác nhau ở 1 điểm : ta có thể sử dụng subscript với mảng do ta chỉ cần sử dụng 1 phần tử bên trong mảng mỗi lần, còn với chuỗi thường ta cần dùng tất cả các thành phần trong chuỗi dẫn đến giảm hiệu suất do mỗi lần phải đi qua tất cả các thành phần trong chuỗi nên do đó chỉ có mảng là sử dụng cách subscript bình thường để truy cập các phần tử.