23. Asynchronous Closures and Memory Management

Bạn đã học những điều cơ bản về quản lý bộ nhớ trong Chương 14 "**Advanced Classes**", ở đó bạn đã khám phá vòng đời của lớp và việc đếm tham chiếu tự động hoặc ARC. Trong hầu hết các trường hợp, quản lý bộ nhớ hoạt động bên ngoài box trong Swift và bạn không cần phải lo lắng về nó.

Tuy nhiên, có một số trường hợp chắc chắn khi ARC không thể nội suy ra mói quan hệ thích hợp giữa các đối tượng và bạn cần phải nói cho nó . Trong chương này, bạn sẽ xem lại khái niệm về các vòng tròn tham chiếu và học cách giải quyết chúng với các lớp và các **closure**. Bạn cũng sẽ học quản lý mã không đồng bộ với **Grand Central Dispatch** và sử dụng các danh sách nắm giữ trong các **closure** để nắm giữ các giá trị từ phạm vi đóng. (enclosing)

Trước khi bạn bắt đầu khám phá vòng tròn tham chiếu, điều quan trọng phải hiểu chúng được tạo ra như thế nào.

Reference cycles for classes

Hai thể hiện của lớp mà giữ một tham chiếu mạnh tới mỗi thể hiện của lớp khác tạo ra một vòng tròn tham chiếu mạnh (strong reference cycle) là dẫn tới việc rò rỉ bộ nhớ (memory leak). Đó là bởi vì mỗi thể hiện giữ một thể hiện khác tồn tại, vì vậy số tham chiếu của chúng không bao giờ đi tới **0**

Ví dụ , website của chúng ta có rất nhiều các bài tập (tutorial) lập trình nâng cao và mỗi bài tập được chỉnh sửa bởi một người chỉnh sửa (editor) cụ thể. Bạn có thể mô hình những bài tập này với lớp như sau :

```
class Tutorial {
    let title: String
    var editor: Editor?

    init(title : String) {
        self.title = title
    }
    deinit {
        print("Goodbye Tutorial : \(title)!")
    }
}
```

Ngoài việc bổ sung một thuộc tính title , một bài tập có thể có một người biên dịch (editor) vì vậy nó được đánh dấu là **optional**. Bạn có thể có các thuộc tính bổ sung như là ngày tạo , danh mục , thân văn bản ... nhưng hãy loại bỏ chúng ra khỏi ví dụ này để giữ mọi thứ đơn giản. Nhớ rằng từ Chương 14 , "**Advanced Classes**", mà Swift gọi **deinitializer** tự động ngay trước khi nó giải phóng đối tượng từ bộ nhớ và số tham chiếu của nó trở về **0**.

Bạn đã định nghĩa một người chỉnh sửa (editor) cho mỗi bài tập (tutorial) , nhưng bạn chưa khai báo lớp **Editor**. Đây là những gì nó cần :

```
class Editor {
    let name: String
    var tutorials : [Tutorial] = []

    init(name: String) {
        self.name = name
    }
    deinit {
        print("Goodbye Author : \(name)!")
    }
}
```

Mỗi người chỉnh sửa có một tên (name) và một danh sách các bài tập họ đã chỉnh sửa. Thuộc tính tutorials là một mảng vì vậy bạn có thể bổ sung tới nó.

Bây giờ định nghĩa một bài tập mới (tutorial) để xuất bản và một người biên dịch để mang nó tới tiêu chuẩn chất lượng cao của Site :

```
do {
    let tutorial : Tutorial = Tutorial(title: "Memory managent")  Immutable valu
    let editor : Editor = Editor(name: "Ray")  Immutable value 'editor' was never used;
}
```

Chúng được đặt vào trong một phạm vi (được tạo với **do {}**) để mà ngay khi chúng đi ra ngoài phạm vi chúng bị giải phóng (deallocated). Mọi thứ làm việc tốt , nhưng điều gì xảy ra nếu bạn thay thế code mà tạo ra một mối quan hệ giữa hai đối tượng như vậy :

```
do {
    let tutorial : Tutorial = Tutorial(title: "Memory managent")
    let editor : Editor = Editor(name: "Ray")
    tutorial.editor = editor
    editor.tutorials.append(tutorial)
}
```

Mặc dù cả hai đối tượng bạn đi ra ngoài phạm vi các phương thức **deinitializer** không bao giờ được gọi và không có gì được in tới console. Đó là vì bạn vừa tạo ra một vòng tròn tham chiếu giữa bài tập (tutorial) và người chỉnh sửa tương ứng của nó. Bạn không giải phóng thực sự các đối tượng từ bộ nhớ, mằng dù bạn không cần chúng nữa.

Bây giờ bạn hiểu cách các vòng tròn tham chiếu xảy ra, đây là thời gian để phá chúng. Tham chiếu yếu (Weak) để giải thoát điều này.

- **Weak references**

Weak reference - các tham chiếu yếu không thực hiện bất kỳ phần nào trong số tham chiếu (reference count) của một đối tượng nhất định. Bạn khai báo

chúng là **optional**, vì vậy chúng trở thành **nil** khi số tham chiếu chuyển về **0**

Một bài tập không phải luôn có một người chỉnh sửa (editor) được gán, vì vậy nó tạo ra ý nghĩa hoàn hảo để mô hình nó là một tham chiếu yếu. Thay đổi sự khai báo của thuộc tính trong lớp Tutorial thành như sau :

```
  weak var editor: Editor?
```

Bạn phá vỡ vòng tròn tham chiếu với từ khoá weak; cả hai phương thức **deinitializer** chạy và in ra đầu ra sau tới console bây giờ :

```
Goodbye Author : Ray!
Goodbye Tutorial : Memory managent!
```

Ghi chú : Bạn không thể định nghĩa một tham chiếu **weak** là một hằng bởi vì nó sẽ được thiết lập là **nil** tại một số thời điểm trong thời gian chạy. Vì thực tế, nếu bạn xoá mảng bài tập (toturials) từ một người biên dịch (editor) tham chiếu trong bài tập tự động trở thành **nil** . Rất tuyệt.

Có một cách nữa để phá vỡ vòng tròn tham chiếu giữa các thể hiện lớp : các tham chiếu **unowned**.

- **Unowned references**

Unowned references - các tham chiếu **Unowned** hành động giống như một tham chiếu **weak** : chúng không tăng số tham chiếu của đối tượng. Tuy nhiên không như tham chiếu yếu, chúng luôn mong đợi để có một giá trị - bạn không thể khai báo chúng là các **optional**. Nó không có ý nghĩa với một bài tập (tutorial) tồn tại mà không có một tác giả (author). Việc chỉnh sửa lớp Tutorial như thể hiện bên dưới :

```
class Tutorial {
    let title: String
    let author : Author
    weak var editor: Editor?
    init(title : String, author : Author) {
        self.title = title
        self.author = author
    }
    deinit {
        print("Goodbye Tutorial : \(title)!")
    }
}
```

```
class Author {
    let name : String
    var tutorials : [Tutorial] = []

    init(name : String) {
        self.name = name
    }
    deinit {
        print("Goodbye Author: \(name)!")
    }
}
```

Điều này đảm bảo rằng các bài tập (tutorials) luôn có một tác giả. Vì với điều đó, nó không được khai báo là **optional**. Nói cách khác , một tác giả tutorials là một biến để mà nó có thể được bổ sung sau khi khởi tạo.

Tuy nhiên, vẫn có một lỗi trong code của bạn. Trước đây tạo ra bài tập chưa có một tác giả, vì vậy chỉnh sửa khai báo của nó như sau :

```
do {
    let author = Author(name: "Cosmin")
    let tutorial : Tutorial = Tutorial(title: "Memory managent",author : author)
    let editor : Editor = Editor(name: "Ray")
    tutorial.editor = editor
    editor.tutorials.append(tutorial)
    author.tutorials.append(tutorial)
}
```

Chỉ người chỉnh sửa (editor) được giải phóng thay vì tất cả các đối tượng. Bạn vừa tạo ra một vòng tròn tham chiếu khác , lần này giữa bài tập (tutorial) và tác giả (author) tương ứng của nó

```
Goodbye Editor : Ray!
```

Mỗi bài tập trên website có một tác giả. Không có các tác giả ẩn danh ở đây!. Thuộc tính author của bài tập là phù hợp hoàn toàn với một tham chiếu không được sở hữu (unowned) vì nó không bao giờ **nil**. Thay đổi khai báo của thuộc tính trong lớp Tutorial thành như sau :

```
class Tutorial {
    let title: String
    unowned let author : Author
    weak var editor: Editor?
    //...
```

Bạn phá vỡ vòng tròn tham chiếu với từ khoá unowned và tất cả các phương thức **deinit** chạy và in ra đầu ra như sau tới console bây giờ :

```
Goodbye Editor : Ray!
Goodbye Author: Cosmin!
Goodbye Tutorial : Memory managent!
```

Đó là cho vòng tròn tham chiếu với các lớp. Đây là lúc chuyển tới vòng tròn

tham chiếu với các closure.

References cycles for closures

Bạn đã học trong chương 8 "**Collection Iteration with Closures**" rằng các **closure** nắm giữ các giá trị từ phạm vi bao quanh (enclosing scope). Bạn tạo ra một vòng tròn tham chiếu nếu bạn gán một **closure** mà nắm giữ một thể hiện của lớp tới một thuộc tính, vì các **closure** là kiểu tham chiếu kết nối bên dưới.

Ví dụ, bổ sung một thuộc tính mà tính toán mô tả của **tutorial** từ lớp Tutorial như thế này :

```
lazy var createDescription: (()->String ) = {
    return "\(self.title) by \(self.author.name)"
}
```

Nhớ rằng trong Chương 11 "**Properties**" rằng bạn định nghĩa một **lazy properties** để đảm bảo bạn không gán nó cho đến khi bạn sử dụng nó với ngay lần đầu tiên và rằng **self** chỉ hiển thị sau khi khởi tạo.

Bây giờ in mô tả (description) của tutorial tới console. Bổ sung code sau ngay sau khai báo của đối tượng tutorial.

```
print(tutorial.createDescription())
```

```
"Thanos by Titan\n"
```

Bạn đã tạo ra vòng tròn tham chiếu **strong** khác giữa đối tượng tutorial và **closure** bằng cách nắm giữ **self**, vì vậy chỉ phương thức deinit của author chạy. Để phá vỡ nó, bạn cần phải biết một hoặc cả hai danh sách nắm giữ (capture list)

Ghi chú : Swift yêu cầu bạn sử dụng **self** bên trong các **closure**, như một lời nhắc nhở rằng một tham chiếu tới đối tượng hiện tại đang bị nắm giữ. Ngoại lệ duy nhất đối với quy tắc này là trong trường hợp các **closure non-escaping**, điều này bạn sẽ tìm hiểu sau.

- **Capture lists**

Một danh sách nắm giữ (capture list) là một mảng của các biến mà được nắm giữ bởi một **closure**. Bạn sử dụng từ khoá **in** để ngăn cách các biến giữa cặp ngoặc vuông với phần còn thân của **closure**. Nếu **closure** có cả hai : danh sách tham số và danh sách nắm giữ, thì danh sách nắm giữ đứng trước .

Hãy xem xét đoạn code sau :

```
var counter = 0
var closure = { print(counter) }
counter = 1
closure()
```

```
0
(2 times)
1
```

Closure in ra giá trị được cập nhật của biến counter, như mong đợi, bởi vì **closure** nắm giữ một tham chiếu tới biến counter.

Nếu bạn muốn hiển thị giá trị ban đầu thay vào đó, định nghĩa một danh sách nắm giữ cho **closure** như thế này :

```
counter = 0
closure = { [counter] in print(counter)} // prints 0
counter = 1
closure()
```

```
0
(2 times)
1
```

Closure bây giờ sẽ sao chép giá trị của counter vào trong một hằng cục bộ mới với tên tương tự. Kết quả là , giá trị **0** ban đầu được in ra.

Khi làm việc với các đối tượng, nhớ rằng "**constant** - hằng" có một ý nghĩa khác với các kiểu tham chiếu. Với các kiểu tham chiếu, một danh sách nắm giữ sẽ làm cho **closure** nắm bắt và lưu trữ tham chiếu hiện tại được lưu trữ trong biến được nắm giữ. Những thay đổi làm cho đối tượng thông qua tham chiếu này sẽ vẫn được hiển thị từ bên ngoài của **closure**.

Bây giờ bạn hiểu cách danh sách nắm giữ làm việc, thời gian này để tìm hiểu cách bạn có thể sử dụng chúng để phá vỡ vòng tròn tham chiếu cho các **closure**.

- **Unowned self**

Closure mà chỉ định desciption của **tutorial** nắm giữ một tham chiếu **strong** của **self** và tạo ra một vòng tròn tham chiếu. Khi **closure** không tồn tại sau khi bạn giải phóng đối tượng tutorial từ bộ nhớ, **self** sẽ không bao giờ là **nil** vì vậy bạn có thể thay đổi tham chiếu **strong** tới một đối tượng không được sở hữu (unowned) bằng cách sử dụng một danh sách nắm giữ :

```
lazy var createDescription: (()->String ) = {
    [unowned self] in
    return "\(self.title) by \(self.author.name)"
}
```

Điều này phá vỡ vòng tròn tham chiếu. Tất cả các phương thức deinit làm việc như trước và đầu ra tới console như sau :

```
Thanos by Titan
Gooodbye Editor BangNguyen
Gooodbye Author Titan!
Gooodby Tutorial Thanos
```

Mã không đồng bộ sử dụng các **closure** để thực hiện công việc và mã nguồn khác của việc tạo ra các vòng tròn tham chiếu.

Handling asynchronous closures.

Môi trường hoạt động hiện tại là đa luồng. Điều này có nghĩa là công việc có thẻ xảy ra không đồng thời (simultaneously) trên nhiều luồng thực hiện. Ví dụ, tất cả các hoạt động nối mạng thực hiện trong một luồng nền vì vậy chúng không khoá giao diện người dùng mà xảy ra trên luồng chính. Trong thực tế , làm việc trong môi trường đa luồng có thể rất phức tạp. Ví dụ , chỉ khi một luồng đang ghi một số dữ liệu, luồng khác có thể đang cố gắng để đọc nó và nhận một giá trị chưa hoàn thành (half-backed) nhưng chỉ rất ngẫu nhiên làm cho nó cực khi khó đoán.

Để tránh các điều kiện cuộc đua (race conditions) như thế này , **synchronization** - sự đồng bộ trở nên cần thiết. Mặc dù Swift chưa có một mô hình đồng bộ tự nhiên, bạn có thể sử dụng một framework được gọi là **Grand Central Dispatch** hoặc **GCD** để đơn giản hoá nhiều các vấn đề này. **GCD** là một sự trừu tượng bên trên của các luồng mà làm cho việc thực hiện làm việc dưới nền nghiêng về lỗi ít.

Ghi chú : Sự khởi tạo của dữ liệu global và việc đếm tham chiếu **Swift** là luồng an toàn vì vậy bạn không cần phải lo lắng về những điều cơ bản đó.

Race condition : xảy ra khi nhiều các luồng cùng truy cập và cùng lúc muốn thay đổi dữ liệu. Vì thuật toán chuyển đổi việc thực thi giữa các luồng có thể xảy ra bất cứ lúc nào nên không thể biết được thứ tự của các luồng truy cập và thay đổi dữ liệu đó sẽ dẫn đến giá trị của dữ liệu sẽ không như mong muốn. Kết quả sẽ phụ thuộc vào thuật toán đặt lịch luồng (**thread scheduling**) của hệ điều hành.

Vì **GCD** cho phép bạn truyền đi các **closure** , và các **closure** đó có thể chứa các tham chiếu tới các đối tượng, bạn càn phải nhận thức được những gì đang được truyền đi để không tạo ra các vòng tròn ngoài ý muốn (unintended). Nguyên tắc sở hữu đã mô tả từ trước làm việc chính xác giống như với code đồng bộ. Sự nguy hiểm đến là khi thứ tự của sự hoàn thành thay đổi từ trong lúc chạy , bạn cần phải đặc biệt cần thận để không truy cập một tham chiếu không được sở hữu (unowned) mà có thể đã biến mất.

- **GCD**

Thay vì tìm ra (exposing) các luồng thô cho bạn, **GCD** cung cấp khái niệm về một hàng đợi (**queue**) công việc. Bạn đặt công việc lên một hàng đợi bằng cách sử dụng một **closure** và **closure** đó trong thân của nó có thể chuyển hướng công việc vào hàng đợi **GCD** khác.

- Nếu một hàng đợi là tiếp nối (serial) nó thực hiện các closure lên nó tuần tự.
- Nếu một hàng đợi là đồng thời (concurrency) nó có thể gửi nhiều các **closure** tại cùng thời điểm.

Các hàng đợi **GCD** là các luồng an toàn - bạn có thể bổ sung các **closure** khác tới một hàng đợi từ bất kỳ các hàng đợi khác.

Bạn sẽ tạo ra một hàm gửi đi execute mà chạy một **closure** trên hàng đợi nền để thực hiện một tính toán dài và sau đó truyền kết quả tới một **closure** trên hàng đợi chính khi nó hoàn thành. Dữ liệu được sao chép, không phải chia sẻ, để tránh các điều kiện đua truy cập dữ liệu (data race condition).

Trước hết, định nghĩa một số hàm bạn sẽ cần :

```
func log(message : String) {
    let thread = Thread.current.isMainThread ? "Main" : "Background"
    print("\(thread) thread \(message)")
}
func addNumbers(upTo range : Int) -> Int {
    log(message : "Adding numbers ...")
    return (1...range).reduce(0,+)
}
```

- log(message:) : sử dụng toán tử tam phân để kiểm tra nếu luồng hiện tại là hàng đợi **main** hoặc **background** và in ra một thông điệp tới tới màn hình console phù hợp.

- addNumbers(upTo:) : tính toán tổng của một phạm vi số cho trước. Bạn sẽ sử dụng hàm này để đại diện cho một nhiệm vụ phức tạp mà phải được chạy trên một luồng nền.

Để chạy các nhiệm vụ trên một hàng đợi nền, đầu tiên bạn cần phải tạo ra một hàng đợi :

```
let queue = DispatchQueue(label: "queue")
```
<OS_dis

Ở đây bạn tạo ra một hàng đợi nối tiếp (serial queue) , nơi các nhiệm vụ thực hiện mỗi cái tại một thời điểm theo thứ tự **FIFO** (first in first out). Nếu bạn định

nghĩa một hàng đợi đồng thời , bạn sẽ phải giải quyết (deal) với tất cả các vấn đề đồng thời, mà vượt qua phạm vi của cuốn sách này. Công việc được gửi đi từ một hàng đợi nối tiếp cụ thể không cần biết về sự can thiệp cùng một lúc từ các **closure** khác trên cùng một hàng đợi nối tiếp. Các hàng đợi đồng thời và việc chia sẻ dữ liệu chung giữa các hàng đợi là một câu chuyện khác và yêu cầu phải xem xét.

Tiếp theo, tạo ra một phương thức như sau :

```
//1:
func execute<Result>(backgroundWork : @escaping ()-> Result,
                     mainWork : @escaping (Result)->()) {
    //2:
    queue.async {
        let result = backgroundWork()
        //3:
        DispatchQueue.main.async {
            mainWork(result)
        }
    }
}
```

Có rất nhiều điều xảy ra ở đây , vì vậy hãy thực hiện nó từng bước :

1. Bạn tạo ra một hàm **generic** bởi vì **closure** backgroundWork trả về một kết quả **generic** trong khi **closure** mainWork làm việc với kết quả đó. Bạn đánh dấu cả hai **closure** với đặc tính @escaping bởi vì chúng thoát khỏi hàm : bạn sử dụng chúng không đồng bộ , vì vậy chúng ta nhận được cuộc gọi sau khi hàm trả về. Các **closure** là **non-ascaping** là mặc định. **Non-escaping** nghĩa là khi hàm đang sử dụng **closure** trả về nó sẽ không bao giờ được sử dụng lại một lần nữa.

2. Bạn chạy **closure** backgroundWork không đồng bộ trên hàng đợi nối tiếp đã định nghĩa trước và lưu trữ giá trị trả về của nó.

3. Bạn gửi đi **closure** mainWork không đồng bộ tới hàng đợi chính và bạn sử dụng kết quả của **closure** backgroundWork là đối số của nó.

Bạn không thể chạy code không đồng bộ trong một **playground out of box**; bạn phải bật chế độ không đồng bộ của **playground** đầu tiên :

```
PlaygroundPage.current.needsIndefiniteExecution = true
```

Bạn cũng sẽ cần phải nhập framework **PlaygroundSupport** để làm việc với lớp PlaygroundPage.

Thời gian để thấy phương thức mới của bạn hoạt động :

```
execute(backgroundWork: {addNumbers(upTo: 100)},
        mainWork: { log(message: "The sum is \($0)")})
```
```
5050
()
```

Ở đây bạn bổ sung các số lên luồng nền và in ra kết quả tới màn hình console lên luồng chính. Điều này cho đầu ra như sau :

```
Background thread: Adding numbers...
Main thread: The sum is 5050
```

Bây giờ bạn biết cách **GCD** làm việc, đây là thời gian để tìm hiểu cách xử lý với các vòng tròn tham chiếu với code không đồng bộ.

- **Weak self**

Có những thời điểm nhất định khi bạn không thể nắm giữ self như một tham chiếu unowned bởi vì nó có thể trở thành **nil** tại một số thời điểm trong khi chạy một nhiệm vụ không đồng bộ. Ví dụ, một trình chỉnh sửa chỉnh sửa một tutorial cụ thể và trình tự phản hồi chỉnh sửa lên thẻ của tutorial như sau :

```
extension Editor {
    func feedback(for tutorial : Tutorial) -> String {
        let tutorialStatus : String
        // Should use the tutorial.content hear, really.
        //Instead , flip a coin
        if arc4random_uniform(10) % 2 == 0 {
            tutorialStatus = "published"
        } else {
            tutorialStatus = "rejected"
        }
        return "Tutorial \(tutorialStatus) by \(self.name)"
    }
}
```

Editor này không thực hiện một công việc tốt và quyết định phải **publish** hoặc **reject** một tutorial một cách ngẫu nhiên.

Bổ sung phương thức editTutorial tới **extentsion** Editor :

```
func editTutorial(_ tutorial : Tutorial) {
    queue.async {
        [unowned self] in
        let feedback = self.feedback(for: tutorial)
        DispatchQueue.main.async {
            print(feedback)
        }
    }
}
```

Phương thức này sử dụng cùng code **GCD** như execute(backgroundWork:mainWork:) mà bạn đã sử dụng khi trước. Việc chỉnh

sửa xảy ra trên một luồng nền và kết quả phản hồi in ra trên luồng chính.

Mặc dù không có vòng tròn tham chiếu xảy ra trong trường hợp này, bởi vì bạn đã quyết định không muốn phản hồi để kéo dài vòng đời của một đối tượng **Editor**, bạn đã khai báo self là unowned. Tuy nhiên, code này không an toàn ! với một số lý do, nếu editor đã vượt ra ngoài phạm vi trước khi editTutorial được hoàn thành, self.feedback sẽ crash khi nó được thực hiện cuối cùng.

- **The Strong weak pattern**

Để cho phép các editor biến mất nếu chúng vượt ra ngoài phạm vi nhưng thực hiện chính xác chỉnh sửa nếu chúng vẫn còn xung quanh khi **closure** thực thi, bạn có thể tạo ra một tham chiếu **strong** tạm thời tới self. Cập nhật editTotorial như sau :

```
func editTutorial(_ tutorial : Tutorial) {
    queue.async {
        [weak self] in

        guard let strongSelf = self else {
            print("i no longer exist so no feedback for you")
            return
        }
        DispatchQueue.main.async {
            print(strongSelf.feedback(for: tutorial))
        }
    }
}
```

Code này là tất cả hoặc không có gì. Nếu **editor** đã biến mất do thời gian **closure** chạy, nó ẽ in ra một thông điệp đầy ý nghĩa. Nếu strongSelf không **nil** nó sẽ dính lên xung quanh trong khi lệnh in hoàn thành.