**[Introducing Protocol-Oriented Programming in Swift 3](https://www.raywenderlich.com/148448/introducing-protocol-oriented-programming" \o "Introducing Protocol-Oriented Programming in Swift 3)**

Hãy tưởng tượng bạn đang phát triển game đua xe. Bạn có thể lái xe ô tô, xe máy hoặc máy bay. Thì bạn nên nghĩ đến thiết kế các đối tượng, nhưng bình thường thì chúng ta sẽ tạo một lớp base rồi tất cả các class khác sẽ kế thừa.

Nhưng nó cũng đem lại một số khó khăn. Ví dụ, nếu bạn muốn tạo một cái phương tiện mà yêu cầu chạy bằng gas hoặc bất kể thứ gì mà bạn muốn sử dụng lại game logic, thì khá là khó khan để tách các thành phần của các phương tiện để có thể tái sử dụng.

Với trường hợp này bạn có thể nghĩ đến protocol.

Swift luôn cho phép bạn chỉ rõ ra interface của class, struct và enum bằng cách sử dụng protocols, Swift 2 đã cho phép bạn extend protocols và với Swift 3 cung cấp cho bạn them các toán tử cơ bản và sử dụng chúng để có thể tạo ra các protocols mới giựa trên các thư viện tiêu chuẩn. Trong bài này sẽ tìm hiểu cách tạo và sử dụng protocol để giúp code của bạn dễ mở rộng.

Bắt đầu

Bạn cần tạo một playground, File\New\Playground… và đặt tên là SwiftProtocols. Bạn viết các đoạn mã sau:

|  |
| --- |
| protocol Bird {  var name: String { get }  var canFly: Bool { get }  }    protocol Flyable {  var airspeedVelocity: Double { get }  } |

Nó định nghĩa một protocol có tên là Bird với các thuộc tính là name và canFly cũng như protocol có tên Flyable nó định nghĩa thuộc tính airSpeedVelocity.

Chúng going như khi bạn lập trình như trước thì protocol Flyable như là một base class và sau đó bạn định nghĩa ra những loài nào có thể bay ví dụ như máy bay. Nhưng hãy chú ý là ở đây tất cả thao tác với protocol. Bạn sẽ thấy các đoạn mã bạn viết ra sẽ mềm dẻo hơn khi bạn định nghĩa ra các kiểu ở phần tiếp theo.

**Định nghĩa protocol-conforming types**

Thêm một struct sau:

|  |
| --- |
| struct FlappyBird: Bird, Flyable {  let name: String  let flappyAmplitude: Double  let flappyFrequency: Double  let canFly = true    var airspeedVelocity: Double {  return 3 \* flappyFrequency \* flappyAmplitude  }  } |

Ở đây tôi định nghĩa một struct mới FlappyBird, nó tuân thủ theo Bird và Flyable protocols. Vận tốc bay của nó được đính toán như trên. Và thuộc tính canFly = true nghĩa là nó có thể bay. Tiếp theo chúng ta sẽ định nghĩa thêm 2 structs:

struct Penguin: Bird {

let name: String

let canFly = false

}

struct SwiftBird: Bird, Flyable {

var name: String { return "Swift \(version)" }

let version: Double

let canFly = true

*// Swift is FASTER every version!*

var airspeedVelocity: Double { return version \* 1000.0 }

}

Dĩ nhiên chim cánh cụt là một loại chim nhưng nó không thể bay. May quá ở đây chúng ta không sử dụng kế thừa, việc sử dụng protocol cho phép chúng ta chỉ rõ ra đối tượng nào phù hợp với protocol đó.

**Extending protocols với default implementations**Chúng ta sử dụng extension cho protocol, bạn có thể định nghĩa giá trị mặc định với các thuộc tính:

extension Bird {

*// Flyable birds can fly!*

var canFly: Bool { return self is Flyable }

}

Với đoạn mã trên chúng ta định nghĩa giá trị mặc định của thuộc tính canFly là true nếu đối tượng nào tuân thủ theo protocol Flyable.

i)

Xoá let canFly=…. ở FlappyBird, SwiftBird, Penguin ở các structs đã tạo. Vì chúng ta chỉ cần set nó ở extension.

**Tại sao không sử dụng các Base Classes?**

Với Protocol extensions và default implementations trông có vẻ như sử dụng một base class hoặc abstract classes trong các ngôn ngữ khác, nhưng đối với Swift nó có vài lợi thế sau:

* Đối với một đối tượng bạn có thể tuân thủ theo nhiều protocols nhưng việc kế thừa thì các ngôn ngữ thường chỉ cho kế thừ trực tiếp từ một lớp cha.
* Protocols có thể được tuần thủ các kiểu như class, struc, enum. Đối với việc sử dụng base classes và kế thừa thì nó hạn chế chỉ sử dụng ở đối với class.

Nói cách khác thì protocol extensions cho phép định nghĩa giá trị mặc định cho thuộc tính mà không chỉ class làm được.

Ở trên chúng ta đã sử dụng structs. Tiếp theo chúng ta sẽ thử thao tác với kiểu enum:

enum UnladenSwallow: Bird, Flyable {

case african

case european

case unknown

var name: String {

switch self {

case .african:

return "African"

case .european:

return "European"

case .unknown:

return "What do you mean? African or European?"

}

}

var airspeedVelocity: Double {

switch self {

case .african:

return 10.0

case .european:

return 9.9

case .unknown:

fatalError("You are thrown from the bridge of death!")

}

}

}

Ở đây tôi định nghĩa UnladenSwallow có kiểu enum và nó tuân thủ theo 2 protocols, vì nó là chim và nó có thể bay, và dĩ nhiên thuộc tính canFly sẽ là true.

**Overriding Default Behavior**

Với UnladenSwallow nó có kiều loại khác nhau thì trong trường hợp này nếu không biết nó là loại của vùng nào tôi sẽ cho thuộc tính canFly là false, và điều đó toàn toàn có thể làm được bằng cách override lại default implementation.

extension UnladenSwallow {

var canFly: Bool {

return self != .unknown

}

}

Bây giờ chỉ có .african và .european sẽ trả về true đối với thuộc tính canFly. Để kiểm tra bạn thêm đoạn mã sau:

UnladenSwallow.unknown.canFly *// false*

UnladenSwallow.african.canFly *// true*

Penguin(name: "King Penguin").canFly *// false*

Đây là cách bạn có thể override các thuộc tính và methods cũng giống như object oriented programming.

**Extending Protocols**

Bạn có thể tận dụng các protocols từ các thư viện mặc định và hoàn toàn có thể định nghĩa default behaviors cho nó.

Chỉnh lại protocol Bird và cho nó tuân thủ theo CustomStringConvertible protocol:

protocol Bird: CustomStringConvertible {

Việc tuân thủ theo CustomStringConvertitle có nghĩa là bạn cần thuộc tính description. Có phải nó có nghĩa là đối với tất cả các nơi đã tuần thủ theo Bird ?

Bạn hoàn toàn có thể làm như sau để set giá trị cho các đối tượng tuân thủ theo Bird như sau:

extension CustomStringConvertible where Self: Bird {

var description: String {

return canFly ? "I can fly" : "Guess I’ll just sit here :["

}

}

Bạn có thể thử thay đổi kiểu của UnladenSwallow xem description thay đổi như nào.

## Protocol Comparators

Như đã nói ở đầu bài thì swift 3 cho phép bạn có thể thêm các toán tử so sánh vào protocol.

Các bạn them các đoạn mã sau:

protocol Score {

var value: Int { get }

}

struct RacingScore: Score {

let value: Int

}

Vì ở bài này chúng ta đang thử làm một game đua xe nên chúng ta cần điểm, với protocol Score thì đôi tượng nào tuân thủ theo thì sẽ có điểm, vậy khi chúng ta muốn so sánh điểm giữa các đối tượng thì chúng ta có thể làm như sau:

protocol Score: Equatable, Comparable {

var value: Int { get }

}

struct RacingScore: Score {

let value: Int

static func ==(lhs: RacingScore, rhs: RacingScore) -> Bool {

return lhs.value == rhs.value

}

static func <(lhs: RacingScore, rhs: RacingScore) -> Bool {

return lhs.value < rhs.value

}

}

Bạn có thể tham khảo bài này để hiểu rõ hơn về lập trình generic

Bạn thêm đoạn mã sau để kiểm tra:

RacingScore(value: 150) >= RacingScore(value: 130) *// true*