

Swift Protocols

Using Protocols

Prof. Joaquim Pessoa Filho



Agenda

Introdução

O que é Protocolo

Syntax de Protocol Syntax

Property Requirements

Method Requirements

Initializer Requirements

Delegation

Extensions & Protocol Extensions

Inheritance

Protocol Composition

Tips and Good habits

Conclusão

Introdução

- Protocolo está disponível na linguagem Swift desde a versão inicial
- É considerado similar a `Interface` contida em outras linguagens de programação
- Maneira inteligente de encapsulamento de funcionalidades
- Faz parte do padrão de projeto *Delegation*
- Auxilia na prevenção de vazamento de memória e referencias fortes.

O que é Protocol

- Um protocolo define um modelo;
- É composto de métodos, propriedades e outros requisitos que se adequam a uma tarefa ou um pedaço de funcionalidade.
- O protocolo pode ser adotada por uma classe , estrutura ou enumerador;
- Pode-se estender um protocolo para implementações extras;

Sintaxe do Protocolo

Primeiro contato

A definição de protocolo é muito parecida com Estruturas, Classes e Enumeradores

```
Protocol MyFirstProtocol {  
    //Code...  
}
```

Adoção de Protocolo

Custom Types

Para que um *Custom Type* adote um protocolo, basta inserir o nome do protocolo logo após o nome do *Custom Type* e separá-los com dois pontos.

```
struct MyCustomStruct: MyFirstProtocol {  
    //Code...  
}
```

Adoção de vários Protocolos

Múltiplos Protocolos

Para uma adoção de mais de um protocolo, basta lista-los e separa-los com uma vírgula.

```
struct MyCustomStruct: MyFirstProtocol, MySecondProtocol {  
    //Code...  
}
```

Protocolos & Super Classes

Hierarquia

Para uma adoção de uma classe que possui uma Super Classe, basta listar e separar todos os parentescos e protocolos com uma vírgula.

```
class MyCustomClass: SuperClass, MyFirstProtocol, MySecondProtocol {  
    //Code...  
}
```


Property Requirements

Propriedade

Um protocolo pode exigir qualquer tipo em conformidade para fornecer uma propriedade de instância.

O protocolo também especifica se cada propriedade deve ser apenas do tipo `get` ou se é do tipo `get` e `set`.

Property requirements são representadas através da palavra `var`.

```
protocol MyFirstProtocol {  
    var age: Int { get set }  
}
```

Property Requirements

```
protocol MyFirstProtocol {  
    var age: Int { get set }  
}
```

Criando uma estrutura que adote o protocolo criado anteriormente, sendo assim obrigatoriamente deve existir uma variável chamada age do tipo Int:

```
struct Animal: MyFirstProtocol {  
    var age: Int  
}
```

Criando e preenchendo valores de uma estrutura do tipo Animal

```
let rex = Animal(age:3)
```

Method Requirements

Métodos e Protocolos

Protocolos podem solicitar métodos específicos, esses métodos fazem parte da definição do protocolo e devem ser escritos da mesma forma pela qual foram declarados

Na hora de definir os métodos, as chaves devem ser retiradas

Veja abaixo a declaração de um método dentro de um protocolo.

```
protocol MyFirstProtocol {  
    var myVariable: Int { get set }  
    func myMethod() -> String  
}
```

Method Requirements

Exemplo

```
protocol MyFirstProtocol {  
    var myVariable: Int { get set }  
    func myMethod() -> String  
}
```

Veja abaixo uma classe qualquer implementando o protocolo acima

```
class MyClass: MyFirstProtocol{  
    var myVariable: Int = 0  
    func myMethod() -> String{  
        return "Hello World"  
    }  
}
```

Initializer Requirements

Construtores e Protocolos

Protocolos podem solicitar construtores específicos

Esses construtores são declarados normalmente dentro do protocolo, e igualmente aos métodos, eles devem ser declarados sem as chaves.

```
protocol MyFirstProtocol {  
    init (someParameter:String)  
}  
  
class MyClass: MyFirstProtocol{  
    required init (someParameter:String) {}  
}
```

Initializer Requirements

Construtores e Protocolos

O uso do `required` garante que todas as subclasses também estejam em conformidade com o protocolo.

```
class MyClass: MyFirstProtocol{  
    required init (someParameter:String) { }  
}
```

O que é Delegation?

- É um *design pattern* que permite entregar responsabilidades a uma instância de outro tipo.
- Funciona através de eventos dentro do processo de execução.
- É possível prover funcionalidades por trás do código.
- Permite uma abstração maior de funcionalidades.
- Feito através de protocolos, com o diferencial de conseguir cumprir suas responsabilidades ao longo da implementação.

Delegation

Exemplo de um protocolo delegate

Protocolo Comum

```
protocol RunningPlayer {  
    var currentSpeed: Float { get set }  
}
```


Delegation

Exemplo de um protocolo delegate

Delegate Protocol

```
protocol RunningPlayerDelegate {  
    func gainSpeed (player: RunningPlayer, withSpeed speed: Float)  
    func startRunning (player: RunningPlayer)  
    func stopRunning (player: RunningPlayer)  
}
```

Delegation

Exemplo de um protocolo delegate

Já no *delegate*, você terá a oportunidade de criar um objeto que servirá como *trigger* para determinadas ações dentro da estrutura de seu código. Esse *delegate* será declarado da seguinte maneira.

```
weak var delegate: RunningPlayerDelegate?
```

Se você criar um protocolo classe, você deve criar um `weak delegate` para ter maior controle sobre referências. Deixar como opcional faz com que o *optional-chaining* do Swift entre em ação para possíveis erros.

Delegation

Exemplo de um protocolo delegate

```
struct Player: RunningPlayer {
    var currentSpeed: Float = 0.0
    weak var delegate: RunningPlayerDelegate?

    mutating func run () {
        delegate?.startRunning(player: self)

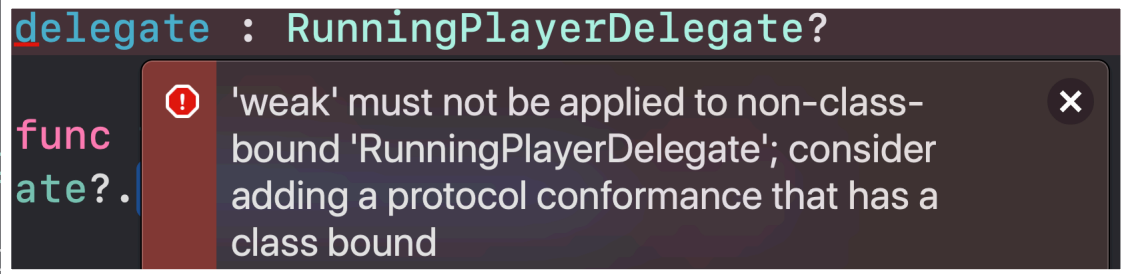
        for increasingSpeed in 1...10 {
            delegate?.gainSpeed(player: self, withSpeed: Float(increasingSpeed))
            self.currentSpeed += Float(increasingSpeed)
        }

        delegate?.stopRunning(player: self)
    }
}
```

Delegation

Exemplo de um protocolo delegate

```
struct Player: RunningPlayer {  
    var currentSpeed: Float = 0.0  
    weak var delegate : RunningPlayerDelegate?  
  
    mutating func delegate?.  
    for {  
        delegate?.gainSpeed(player: self, withSpeed: Float(increasingSpeed))  
        self.currentSpeed += Float(increasingSpeed)  
    }  
  
    delegate?.stopRunning(player: self)  
}  
}
```



Delegation

Exemplo de um protocolo delegate

```
struct Player: RunningPlayer {
    var currentSpeed: Float = 0.0
    weak var delegate: RunningPlayerDelegate?

    mutating func run () {
        delegate?.startRunning(player: self)

        for increasingSpeed in 1...10 {
            delegate?.gainSpeed(player: self, withSpeed: Float(increasingSpeed))
            self.currentSpeed += Float(increasingSpeed)
        }

        delegate?.stopRunning(player: self)
    }
}
```

```
protocol RunningPlayerDelegate: AnyObject {
    ...
}
```

Delegation

Exemplo de um protocolo delegate

```
struct Player: RunningPlayer {
    var currentSpeed: Float = 0.0
    weak var delegate: RunningPlayerDelegate?

    mutating func run () {
        delegate?.startRunning(player: self)

        for increasingSpeed in 1...10 {
            delegate?.gainSpeed(player: self, withSpeed: Float(increasingSpeed))
            self.currentSpeed += Float(increasingSpeed)
        }

        delegate?.stopRunning(player: self)
    }
}
```

```
protocol RunningPlayerDelegate: AnyObject {
    ...
}
```

```
var player = Player()
player.delegate = DelegateConformance()
player.run()
```



Instance will be immediately deallocated because property 'delegate' is 'weak'



Delegation

Exemplo de um protocolo delegate

```
struct Player: RunningPlayer {
    var currentSpeed: Float = 0.0
    weak var delegate: RunningPlayerDelegate?

    mutating func run () {
        delegate?.startRunning(player: self)

        for increasingSpeed in 1...10 {
            delegate?.gainSpeed(player: self, withSpeed: Float(increasingSpeed))
            self.currentSpeed += Float(increasingSpeed)
        }

        delegate?.stopRunning(player: self)
    }
}
```

```
protocol RunningPlayerDelegate: AnyObject {
    ...
}
```

```
var player = Player()
let delegate = DelegateConformance()
player.delegate = delegate
player.run()
```

Delegation

Exemplo de um protocolo delegate

```
class DelegateConformance: RunningPlayerDelegate {  
    func startRunning(player: RunningPlayer) {  
        print("Running!")  
    }  
  
    func gainSpeed(player: RunningPlayer, withSpeed speed: Float) {  
        print(player.currentSpeed)  
        print("Speed engaging: \(speed)")  
    }  
  
    func stopRunning(player: RunningPlayer) {  
        print("Stopped Running!")  
    }  
}
```


Delegation

Exemplo de um protocolo delegate

```
Running!
0.0
Speed engaging: 1.0
1.0
Speed engaging: 2.0
3.0
Speed engaging: 3.0
6.0
Speed engaging: 4.0
10.0
Speed engaging: 5.0
15.0
Speed engaging: 6.0
21.0
Speed engaging: 7.0
28.0
Speed engaging: 8.0
36.0
Speed engaging: 9.0
```

```
45.0
Speed engaging: 10.0
Stopped Running!
```

```
var player = Player()
let delegate = DelegateConformance()
player.delegate = delegate
player.run()
```

O que é Protocol Extensions?

- Protocolos podem fazer uso de `extensions` para entrarem em conformidade com determinados métodos e propriedades.
- Essa extensão serviria para definir um comportamento padrão para determinado método/função, caso não houvesse sobrescrita do mesmo.
- Nesse caso, o protocolo definiria o seu próprio comportamento ao invés da conformidade de cada um dos tipos.

Protocol Extensions

Exemplo

```
protocol Person {  
    var name: String { get set }  
    var age: Int { get set }  
    func aboutMe()  
}  
  
extension Person {  
    func aboutMe () {  
        // Default implementation  
        print("My name is \(name) and I'm \(age)!")  
    }  
}  
  
struct Human: Person{  
    var name: String  
    var age: Int  
}
```

Protocol Extensions

Exemplo

```
let human = Human(name: "Marcos", age: 21)
human.aboutMe()
//Output
//My name is Marcos and I'm 21!
```

O que é Herança em Protocolos?

- Um protocolo pode herdar de um ou mais protocolos.
- Sendo assim, novos requerimentos podem ser exigidos pelo protocolo herdeiro.
- Ele deverá satisfazer todos os outros requerimentos dos protocolos herdados.

Herança de Protocolos

Exemplo

```
protocol ParentProtocol {  
    // Primeiro Protocolo a ser Herdado  
}  
  
protocol SuperParentProtocol {  
    // Segundo Protocolo a ser Herdado  
}  
  
protocol ChildProtocol: ParentProtocol, SuperParentProtocol {  
    // Definições do protocolo "filho"  
}
```

Protocol Composition

Entrando em
conformidade com
múltiplos Protocolos

- Você pode juntar múltiplos protocolos em um só requerimento com a Composição de Protocolos.
- Para fazer a composição de protocolos, você colocará o seguinte formato:

```
protocol <PrimeiroProtocolo,SegundoProtocolo,...>
```

- É ideal para definir um protocolo temporário que está em conformidade com os protocolos declarados.
- Um exemplo seria em enviar um parâmetro de um protocolo em conformidade com outros protocolos.

Protocol Composition

Exemplo

```
protocol Job {  
    var salary: Float { get set }  
}  
  
protocol Employee {  
    var name: String { get set }  
    var age: Int { get set }  
    var gender: String { get set }  
}  
  
struct Programmer: Job, Employee {  
    var salary: Float  
    var name: String  
    var age: Int  
    var gender: String  
}
```


Protocol Composition

Exemplo

```
func hire (candidate: Job & Employee) {  
    print("You're hired, \(candidate.name), with a  
        \(candidate.salary) salary!")  
}  
  
let programmer = Programmer(salary: 6000, name: "John Appleseed",  
    age: 22, gender: "Male")  
  
hire(candidate: programmer)  
//You're hired, John Appleseed, with a 6000.0 salary!
```

Protocolos Conclusão

- Protocolos facilitam a vida de um programador Swift.
- O encapsulamento de funcionalidades faz com que o código fique limpo, organizado e bem construído.
- *Delegation* é uma mão na roda na hora de trabalhar com eventos e *triggers*.
- A implementação de protocolos permite um controle muito maior de suas classes e, conseqüentemente, de suas instâncias.
- A programação orientada a protocolos é o futuro.

Recapitulando

What is Protocol?

Protocol Syntax

Protocol Requirements

Delegation

Protocols Extensions & Inheritance

Protocol Composition



Universidade Presbiteriana
Mackenzie

150 anos
1870 - 2020



Faculdade de
Computação e Informática

