Unidade 5

Aula 1

Nesta aula iremos um pouco a fundo no Swift. Falaremos sobre os protocolos, que é um recurso importantíssimo no desenvolvimento iOS. Eles são interfaces que definem um esqueleto de métodos, propriedades e outros requisitos.

# Protocolos

Protocolos são como interfaces do Java. Um tipo de dado que definem um esqueleto de métodos, propriedades e outros requisitos que conformam com uma tarefa em particular ou um pedaço de funcionalidade. Os protocolos podem então ser adotados por uma classe, estrutura (struct) ou enumerações (enum) para providenciar uma implementação real destes requisitos. Qualquer **tipo personalizado (classe, struct ou enum)** que satisfaça os requisitos de um protocolo é dito que **conforma com o protocolo.**

## Sintaxe do Protocolo

Os protocolos podem ser definidos de um jeito muito similar as classes, structs e enums:

protocol SomeProtocol {

  // A definição do protocolo é feita aqui

}

Para fazermos com que um tipo personalizado adote um protocolo, utilizamos a mesma sintaxe de uma herança. Basta colocar o nome do protocolo após o nome do tipo personalizado, separando por vírgula, caso já haja uma herança ou adoção de outros protocolos.

struct SomeStructure: FirstProtocol, AnotherProtocol {

  // A definição da struct é feita aqui

}

Vale lembrar que, o primeiro nome que vem depois dos dois pontos (: ) deve ser o nome da classe herdada, caso haja uma herança:

class SomeClass: SomeSuperclass, FirstProtocol, AnotherProtocol {

  // A definição da classe é feita aqui

}

## Requisitos de conformidade

O protocolo pode requerer qualquer tipo de conformidade para prover uma propriedade de instância ou um tipo de propriedade com um nome e tipo particular. O protocolo não especifica se a propriedade deve ser de armazenamento ou computada, ele apenas especifica seu nome e tipo. O protocolo também pode especificar o nível de acesso da propriedade, ou seja, se ela é somente **gettable** ou se é **gettable** e **settable.**

### Propriedades

Se um protocolo requer que uma propriedade seja *gettable* e *settable*, este requisito não pode ser satisfeito por uma constante *stored* ou uma propriedade computada *read-only*. Se o protocolo apenas requer que a propriedade seja *gettable*, o requisito pode ser satisfeito por qualquer tipo de propriedade.

Requisitos de propriedades devem sempre ser declarados como variáveis (usando a palavra-chave **var** como prefixo). Propriedades gettables e settables devem ser indicadas escrevendo **{ get set }** depois da declaração de seu tipo e uma propriedade apenas gettable deve ser indicada escrevendo **{ get** **}**, como no exemplo abaixo.

protocol SomeProtocol {

  var mustBeSettable: Int { get set }

  var doesNotNeedToBeSettable: Int { get }

}

Requisitos de propriedades podem vir com o modificador **static**, mas isto não significa que o protocolo possui uma variável estática, e que o tipo personalizado que o adota deverá ter:

protocol AnotherProtocol {

  static var someTypeProperty: Int { get set }

}

Aqui está um exemplo de um protocolo que possui um único requisito de propriedade:

protocol FullyNamed {

  var fullName: String { get }

}

O protocolo **FullyNamed** exige um tipo de conformação para fornecer um nome completo de alguém. O protocolo não especifica nada sobre a natureza da conformação do tipo, ou seja, não restringe que uma certa String deva conter o nome completo de alguém. Neste caso o protocolo indica isto semanticamente através de seu nome (FullyNamed, que significa “Com nome completo”) e sua propriedade (fullName, que significa “Nome completo”).

O protocolo estabelece que qualquer um que conforme com FullyNamed deve ter uma propriedade de instância (atributo) gettable chamada **fullName**, que é do tipo String.

Veja abaixo o exemplo de uma struct simples que adota e está em conformidade com o protocolo FullyNamed:

struct Person: FullyNamed {

  var fullName: String

}

let john = Person(fullName: "John Appleseed")

// john.fullName é "John Appleseed"

Este exemplo define uma *struct* chamada **Person**, que representa uma pessoa com nome específico. Ela adota o protocolo FullyNamed na primeira linha de sua definição.

Cada instância Person terá uma única propriedade chamada fullName que é do tipo String. Isto coincide com a única exigência do protocolo FullyNamed.

DICA: O Swift relata um erro em tempo de compilação se um requisito do protocolo não é cumprido.

Aqui está uma classe mais complexa que também adota o protocolo FullyNamed e está em conformidade com ele:

class Starship: FullyNamed {

  var prefix: String?

  var name: String

  init(name: String, prefix: String? = nil) {

      self.name = name

      self.prefix = prefix

  }

  var fullName: String {

      return (prefix != nil ? prefix! + " " : "") + name

  }

}

var ncc1701 = Starship(name: "Enterprise", prefix: "USS")

// ncc1701.fullName é "USS Enterprise"

Veja que a propriedade fullName possui um corpo. Propriedades com este corpo se chamam **Propriedades computadas**. Elas tem este nome pois não armazenam valores, mas calculam. Quando temos a palavra **return**, significa que esta propriedade é apenas **gettable.** Se ela fosse gettable e settable, teria o seguinte formato:

var fullName: String {

get {

return (prefix != nil ? prefix! + " " : "") + name

}

set (newValue) {

// Computa o que vazer quando é atribuído algum valor a fullName

}

}

Esta é a maneira de se encapsular os atributos em Swift.

Essa classe implementa o requisito de propriedade fullName como uma propriedade de somente leitura computada para uma Starship. Cada instância da classe Starship armazena um nome (name) obrigatório e um prefixo (prefix) opcional. A propriedade fullName utiliza o valor de prefix, se ele existir, e o anexa no início de name para criar um nome completo para a nave.

### Métodos

Os protocolos podem exigir métodos de instância e métodos de tipo específico a serem implementados e conformados. Estes métodos são escritos como parte da definição do protocolo, exatamente da mesma forma que para os métodos de instância e tipo normal, mas sem chaves ou um corpo do método.

O exemplo a seguir define um protocolo com um único requisito de método de instância:

protocol RandomNumberGenerator {

  func random() -> Double

}

O protocolo **RandonNumberGeneretor** requer que o tipo (classe, struct, enum) que o conforme tenha um método chamado **random()**, que deve retornar um valor Double sempre que o mesmo for chamado.

O protocolo **RandonNumberGeneretor** não faz quaisquer suposições sobre como será cada número aleatório gerado. Ele simplesmente exige um método gerador para fornecer uma maneira padrão de gerar um novo número aleatório.

Aqui está uma implementação de uma classe que o adota e está em conformidade com ele. Esta classe implementa um algoritmo pseudo-aleatório do gerador de número conhecido como um gerador linear congruente:

class LinearCongruentialGenerator: RandomNumberGenerator {

  var lastRandom = 42.0

  let m = 139968.0

  let a = 3877.0

  let c = 29573.0

  func random() -> Double {

      lastRandom = ((lastRandom \* a + c).truncatingRemainder(dividingBy:m))

      return lastRandom / m

  }

}

let generator = LinearCongruentialGenerator()

print("Número aleatório: \(generator.random())")

// Mostrará "Número aleatório: 0.37464991998171"

print("Outro número aleatório: \(generator.random())")

// Mostrará "Outro número aleatório: 0.729023776863283"

Você está percebendo a semelhança de uma interface em Java?

### Initializers

Os protocolos podem exigir Initializers (inicializadores) específicos a serem implementados. Escrevemos estes initializers como parte da definição do protocolo, exatamente da mesma forma que para initializers normais, mas sem chaves ou um corpo inicializador:

protocol SomeProtocol {

  init(someParameter: Int)

}

### Implementando um initializer requerido pelo protocolo

Você pode implementar um requisito inicializador para uma classe tanto como **um inicializador designado** ou um **inicializador de conveniência**. Em ambos os casos, você deve marcar a implementação do inicializador com o modificador **required**:

class SomeClass: SomeProtocol {

  required init(someParameter: Int) {

      // A implementação do inicializador entra aqui

  }

}

O uso do modificador required garante que você forneça uma implementação explícita ou herdada do inicializador em todas as subclasses das classes conformes, de tal forma que eles também estão em conformidade com o protocolo.

Se uma subclasse anula um inicializador designado de uma superclasse e também implementa um requisito inicializador que corresponde a um protocolo, marque a implementação inicializador com ambos os modificadores **required** e **override**:

protocol SomeProtocol {

  init()

}

class SomeSuperClass {

  init() {

      // A implementação do inicializador entra aqui

  }

}

class SomeSubClass: SomeSuperClass, SomeProtocol {

  // "required" de SomeProtocol; "override" de SomeSuperClass

  required override init() {

      // A implementação do inicializador entra aqui

  }

}

## Protocolos como tipos de dados

Semelhante as interfaces, os protocolos podem ser utilizados para determinar tipos de dados para serem utilizados no seu código.

Podendo ser um tipo, você pode utilizar os protocolos para os mesmos fins que os outros tipos são utilizados, incluindo:

* Como um tipo de parâmetro ou tipo de retorno em uma função, método ou inicializador;
* Como tipo de uma constante, variável ou propriedade;
* Como tipo de itens em um array, dicionário ou outro tipo de recipiente.

DICA: Como os protocolos podem ser utilizados como tipos, sua nomenclatura deve aconselhavelmente ser iniciadas com letra maiúscula (como em FullyName e RandomNumberGenerator) para coincidir com os demais tipos em Swift (como Int , String e Double).

Abaixo temos o exemplo do protocolo RandomNumberGenerator, que criamos anteriormente, sendo utilizado como um tipo:

class Dice {

  let sides: Int

  let generator: RandomNumberGenerator

  init(sides: Int, generator: RandomNumberGenerator) {

      self.sides = sides

      self.generator = generator

  }

  func roll() -> Int {

      return Int(generator.random() \* Double(sides)) + 1

  }

}

Este exemplo define uma nova classe chamada **Dice** (dado), o que representa um dado de diversos lados para uso em jogos de tabuleiro. Instâncias de Dice têm uma propriedade Int chamada *sides*, que representa o número de lados que eles têm e uma propriedade chamada *generator*, que fornece um gerador de números aleatórios a partir do qual para criar os valores de cada jogada de um dado.

A propriedade *generator* é do tipo RandomNumberGenerator. Portanto, você pode atribuir a um *generator* qualquer instância que adote e esteja em conformidade com RandomNumberGenerator.

Dice também possui um inicializador para configurar seu estado inicial. Este inicializador tem um parâmetro chamado *generator*, o qual é também do tipo RandomNumberGenerator. Você pode passar um valor de qualquer tipo desde que esteja em conformidade com este parâmetro ao inicializar uma nova instância de Dice.

A classe Dice fornece um método de instância chamado *roll*, que retorna um valor inteiro entre 1 e o número de lados nos dados. Este método faz chamada do método gerador **random()** para criar um novo número aleatório entre 0.0 e 1.0, e usa esse número aleatório para criar um valor baseado no número de lados do dado. Como o *generator* é do tipo **RandomNumberGenerator** e neste caso a instância que será atribuída obrigatoriamente deverá estar em conformidade com este protocolo, é garantido que a propriedade tenha o método **random()** para ser chamado.

Veja como a classe Dice pode ser usada para criar um dado de seis lados com a classe de exemplo LinearCongruentialGenerator como o seu gerador de números aleatórios:

var d6 = Dice(sides: 6, generator: LinearCongruentialGenerator())

for \_ in 1...5 {

  print("O número tirado foi \(d6.roll())")

}

// O número tirado foi 3

// O número tirado foi 5

// O número tirado foi 4

// O número tirado foi 5

// O número tirado foi 4

## Extensions

Extensões adicionam novas funcionalidades a uma classe, estrutura, enumeração ou tipo de protocolo existente. Isto inclui a capacidade de estender os tipos para os quais você não tem acesso ao código fonte original (conhecido como modelagem retroativa). Extensões são semelhantes às categorias em Objective-C. (Ao contrário categorias Objective-C, extensões Swift não têm nomes).

Extensões em Swift podem:

* Adicionar propriedades de instância computadas e propriedades do tipo computadas;
* Definir métodos de instância e métodos de tipo;
* Fornecer novos *initializers*;
* Definir e usar novos tipos aninhados;
* Fazer um tipo existente entrar em conformidade com um protocolo.

Em Swift você pode até mesmo estender um protocolo para fornecer implementações de seus requisitos. Veremos isto mais adiante em Extensões de Protocolo.

### Sintaxe da Extension

Declare uma extensão com a palavra chave **extension:**

extension SomeType {

  // Novas funcionalidades a serem adicionadas a SomeType vão aqui

}

Uma extensão pode estender um tipo existente para que este adote um ou mais protocolos. Neste caso os nomes dos protocolos são escritos exatamente da mesma maneira do que em uma classe ou *struct*:

extension SomeType: SomeProtocol, AnotherProtocol {

  // A implementação dos requisitos dos protocolos vão aqui

}

O uso de extensões é uma forma de organizar o código também.

### Conformidade de protocolos com Extensions

Você pode estender um tipo existente para adotar e estar em conformidade com um novo protocolo, mesmo se você não tem acesso ao código-fonte deste tipo. As extensões podem adicionar novas propriedades, métodos e subscritos para um tipo existente, e são, portanto, capaz de adicionar todos os requisitos que um protocolo pode exigir.

Por exemplo, este protocolo, chamado **TextRepresentable**, pode ser implementado por qualquer tipo que tem uma forma representável por texto. Esta poderia ser uma descrição de si mesmo, ou de uma versão textual do seu estado atual:

protocol TextRepresentable {

  var textualDescription: String { get }

}

A classe *Dice* que vimos mais cedo pode ser estendida para adotar e estar em conformidade com TextRepresentable:

extension Dice: TextRepresentable {

  var textualDescription: String {

      return "Um dado de \(sides) lado(s)"

  }

}

DICA: Uma coisa que pode haver confusão é o termo “estender”. Estamos acostumados a utilizá-lo para referir a uma relação de herança, mas neste caso estamos nos referindo ao ato de usar o **extension** do Swift.

Esta extensão adota o novo protocolo exatamente da mesma maneira como se *Dice* tivesse fornecido na sua implementação original. O nome do protocolo é fornecido após o nome do tipo, separados por dois pontos e uma implementação de todos os requisitos do protocolo é fornecido dentro de chaves da extensão.

Qualquer instância de *Dice* pode agora ser tratada como TextRepresentable:

let d12 = Dice(sides: 12, generator: LinearCongruentialGenerator())

print(d12.textualDescription)

// Mostrará "Um dado de 12 lados"

Similarmente, a classe SnakesAndLadders pode ser estendida para adotar e conformar ao protocolo TextRepresentable:

extension SnakesAndLadders: TextRepresentable {

  var textualDescription: String {

      return "Um jogo de cobras e escadas com \(finalSquare) quadrados"

  }

}

print(game.textualDescription)

// Mostrará "Um jogo de cobras e escadas com 25 quadrados"

## Checando conformidade com protocolo

Você pode usar os operadores **is** e **as,** descritos na conversão de tipos, para verificar a conformidade de protocolo e para converter um protocolo específico. Verificação e conversão de um protocolo seguem exatamente a mesma sintaxe que a verificação e conversão a um tipo:

* O operador **is** retorna **true** se uma instância está em conformidade com um protocolo e **false** se isso não acontecer;
* A versão **as?** do operador de downcast retorna um valor opcional do tipo do protocolo e este valor é **nil** se a instância não se conforma com esse protocolo;
* A versão as! do operador de downcast obriga o downcast para o tipo de protocolo e dispara um erro de execução se a entrada de ar não tiver êxito.

Este exemplo define um protocolo chamado HasArea, com um único requisito de propriedade de uma propriedade gettable com retorno Bool chamada area:

protocol HasArea {

  var area: Double { get }

}

Abaixo temos duas classes, Circle e Country que estão ambas em conformidade com o protocolo HasArea:

class Circle: HasArea {

  let pi = 3.1415927

  var radius: Double

  var area: Double { return pi \* radius \* radius }

init(radius: Double) { self.radius = radius }

}

class Country: HasArea {

  var area: Double

init(area: Double) { self.area = area }

}

A classe Circle implementa a propriedade exigida **area** como uma propriedade computada, com base no valor armazenado na propriedade radius. A classe Country implementa a propriedade exigida **area** diretamente como uma propriedade armazenada. Ambas as classes conformam corretamente para o protocolo HasArea.

Aqui está uma classe chamada Animal, que não se conforma com o protocolo HasArea:

class Animal {

  var legs: Int

  init(legs: Int) { self.legs = legs }

}

Agora vamos inicializar um array com uma instância de cada uma destas três últimas classes criadas anteriormente:

let objects: [AnyObject] = [

  Circle(radius: 2.0),

  Country(area: 243\_610),

  Animal(legs: 4)

]

O array **objects** pode agora ser iterado e cada objeto no array pode ser verificado ao validarmos a conformidade com o protocolo HasArea:

for object in objects {

  if let objectWithArea = object as? HasArea {

      print("A Área é \(objectWithArea.area)")

  } else {

      print("Alguma coisa não possui área")

  }

}

// A Área é 12.5663708

// A Área é 243610.0

// Alguma coisa não possui área

## Resumo

Protocolos são como as Interfaces em Java, porém mais poderosos. Com eles podemos tornar nosso código mais maleável e inverter as dependências. Nesta aula você aprendeu a fazer uso dele e a criar seus próprios protocolos. Na aula seguinte aprenderemos uma técnica que utiliza protocolos, o padrão de projeto Delegate.

Aula 2

Depois de tanto falar, vamos finalmente entender o que é o padrão de projeto delegate. Este padrão é muito utilizado em toda a API do iOS, então este conceito será importantíssimo. Além disso, aprenderemos a utilizar bibliotecas de terceiros para poupar nosso tempo de desenvolvimento, igual fazíamos com o Gradle do Android, utilizando o Cocoapods.

# Delegates e Gerenciamento de Dependências

## Padrão de projeto Delegate

Delegação é um mecanismo no qual um objeto age em nome de, ou em coordenação com, outro objeto. Por exemplo, quando você usa uma Table View, um dos métodos que você deve implementar é **tableView(numberOfRowsInSection:).**

Você não esperava que a Table View saiba quantas linhas você quer para cada seção dela, não é? Sendo assim, a tarefa de calcular a quantidade de linhas em cada seção é passada para o protocolo **UITableViewDelegate.** Isto permite que a classe UITableView seja independente dos dados que ela exibe. É bem parecido com o padrão Adapter do Android.

O objeto UITableView faz unicamente seu trabalho de exibir dados na tabela, mas eventualmente ele precisará de informações que ele não tem, como o número de linhas em cada seção. Quando assim, ele passa esta responsabilidade para o delegate que, por sua vez, envia uma mensagem pedindo estas informações adicionais (para um View Controller, por exemplo). As definições podem parecer difíceis no início, mas iremos aprender na prática.

Fazer uma subclasse para substituir os métodos necessários pode até parecer mais fácil do que o delegate, mas você deve levar em consideração que uma subclasse pode ser baseada em somente uma superclasse. Se você pretendesse que um objeto seja o delegado de dois ou mais objetos, você não seria capaz de conseguir isto com subclasses.

DICA: Este é um padrão importante. A Apple usa esta abordagem na maioria das classes UIKit: UITableView, UITextView, UITextField, UIWebView, UIAlert, UIActionSheet, UICollectionView, UIPickerView, UIGestureRecognizer, UIScrollView, e a lista continua.

### Delegates e DataSources

Com o padrão delegate criamos dois objetos: o delegate e o datasource. O delegate está diretamente relacionado a manipulação da interface, ou seja, ele delega a tarefa de como uma view que será apresentada mediante as situações. Já o datasource usa o padrão Delegate para delegar a tarefa de fornecer os dados para a aplicação.

No exemplo da TableView o delegate cuida do *layout* da célula e o datasource cuida de informações específicas como a altura da célula, a quantidade de células e etc.

## CocoaPods

Muito se ouve falar do **CocoaPods**. Se você costuma pesquisar tutoriais na internet ou conversar com desenvolvedores iOS, com certeza já deve ter ouvido falar neste termo. Mas o que é exatamente o CocoaPods?

Em uma tradução livre do inglês, isto significa **vagem de cacau**. Mas é claro que não estamos falando de vagens que irão te ajudar em iOS. A definição dada pelo próprio site do CocoaPods (https://cocoapods.org/) deve ser a melhor para providenciar a resposta que procuramos:

“O CocoaPods é um gerenciador de dependências para projetos Cocoa (iOS, macOS, tvOS, watchOS, etc). Ele possui mais de 25 mil de bibliotecas e está sendo usado em mais de 1.2 milhões de aplicativos. O CocoaPods poderá te ajudar a escalar seus projetos de forma elegante”

Mas o que é gerenciamento de dependências? Ouvimos muito falar no curso de Android, mas parece que ainda precisamos entender melhor.

Não importa que tipo de app você está criando, você poderá utilizar o código de outros desenvolvedores no formato de frameworks ou bibliotecas. O que o gerenciamento de dependências faz é cuidar destes frameworks para você.

No caso do CocoaPods podemos atualizar a versão da biblioteca, adicionar novas bibliotecas e escolher qual versão utilizar, isto tudo de forma separada do seu projeto. Chamados de dependência, uma parte do seu projeto que está sendo resolvida por uma solução de terceiros, e não está necessariamente no mesmo local do seu projeto. Se esta dependência não for instalada, seu projeto simplesmente não funciona. E é isto que o gerenciador de dependências faz, ele instala códigos de terceiros no seu projeto.

### Porque utilizar bibliotecas de terceiros

Pelo que você sentiu até aqui, desenvolver para iOS aparenta ser muito mais simples do que para Android. Engana-se. No iOS existem prós e contras. Um exemplo de tarefa difícil de se realizar em iOS é a manipulação de datas; outra coisa complicada é lidar com chamadas HTTP.

Não estamos restritos a utilizar bibliotecas de terceiros, mas elas definitivamente podem reduzir em muito o tempo de desenvolvimento e nos dar mais chances de melhoramentos nosso app ao invés de ficar escrevendo incontáveis linhas de código que alguém já escreveu.

Podemos utilizar também bibliotecas de terceiros sem um gerenciador de dependências, simplesmente baixando o código de algum lugar e copiando no seu projeto, mas isto pode ser perigoso e demorado. Esta abordagem traz uma série de desvantagens, como:

* Atualizar uma biblioteca para uma versão mais nova pode ser uma tarefa difícil, principalmente se esta biblioteca possui subdependências que precisam ser atualizadas também;
* Manter seu aplicativo sempre com as versões mais recentes da biblioteca pode ser difícil também, principalmente se você não tem o costume de verificar sempre se há algo novo no *site* do desenvolvedor;
* Encontrar novas bibliotecas é frustrante sem utilizar um *site* que centraliza todas as bibliotecas disponíveis como o CocoaPods;
* A Swift está em constante mudança e adicionar uma biblioteca com uma versão antiga da linguagem pode se tornar um pesadelo, pois você terá que traduzir todo o código para a versão mais nova. O CocoaPods cuida do intercambiamento entre as versões da linguagem.

O CocoaPods pode te ajudar a se livrar de todos estes problemas e muitos outros. Ele instala as bibliotecas no projeto resolvendo a questão das subdependências entre elas, te ajudando a descobrir novas bibliotecas e configura para você o ambiente de desenvolvimento apropriado para o uso destas.

### Instalando o CocoaPods

A primeira coisa que devemos fazer é instalar o CocoaPods no nosso Mac. Felizmente o CocoaPods é escrito na linguagem Ruby e as versões mais recentes do macOS (desde a versão OS X 10.7) já vem com o compilador para Ruby instalado. Execute os seguintes passos:

1. Abra o terminal do Mac e entre com o seguinte comando:

**sudo gem install cocoapods**

1. Você precisará inserir a senha de usuário do seu Mac, pois este comando exige privilégios de administrador da máquina. Insira sua senha e a instalação começará.
2. Você precisará utilizar **sudo** somente na instalação, para os demais comandos ele não será preciso. Por último, precisamos das últimas configurações do CocoaPods. Então execute o seguinte comando no terminal para terminarmos a instalação:

**pod setup --verbose**

DICA: O comando sudo garante permissões de administrador ao usuário, dando a ele acesso a diretórios restritos pelo sistema, como por exemplo o diretório bin que é onde o CocoaPods precisa ser salvo.

Aguarde um instante e pronto. Agora poderemos utilizar o CocoaPods no nosso projeto.

### Configurando nosso projeto

Com o CocoaPods instalado no seu Mac agora precisamos configurar nosso projeto para receber as bibliotecas. Para isto será necessário um pouco de conhecimento sobre o uso do terminal.

1. Abra o terminal e navegue até o diretório do seu projeto:

**cd ~/caminho/para/o/diretório/do/seu/projeto**

1. Execute o comando de inicialização:

**pod init**

1. Com o último comando o arquivo chamado **Podfile** é criado. É nele que especificamos quais bibliotecas queremos instalar no nosso projeto. Abra este arquivo com o comando:

**open –a XCode Podfile**

1. O conteúdo virá neste padrão:

# Uncomment the next line to define a global platform for your project

# platform :ios, '9.0'

target 'MyProject' do

# Comment the next line if you're not using Swift and don't want to use dynamic frameworks

use\_frameworks!

# Pods for MyProjectTest

end

Na próxima unidade iremos utilizar uma biblioteca **Firebase**. Vamos adicioná-la ao projeto.

1. Insira dentro do bloco target a seguinte instrução:

pod 'Firebase/Core'

1. Agora execute o comando de instalação no terminal:

**pod install**

1. Você terá o seguinte resultado:

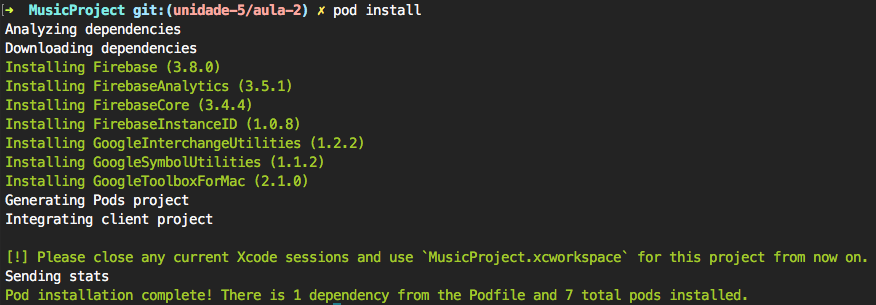


Figura - Tela de progresso da instalação do Firebase utilizando o CocoaPods

Veja que foi instalada uma série de subdependências.

Feito este processo, nosso projeto estará pronto e configurado com o CocoaPods, mas antes de finalizarmos veja como ficou o diretório do projeto:

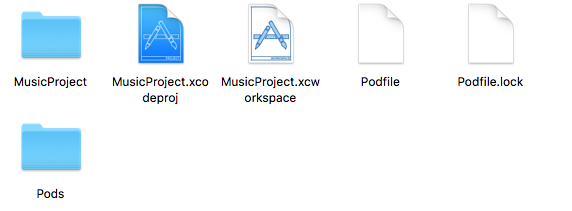


Figura - Nova estrutura de diretórios após a instalação do CocoaPods no projeto

O arquivo **MusicProject.xcworkspace** foi criado. A partir de agora você só deverá abrir este arquivo. Nunca mais use o **arquivo .xcodeproj**, senão as bibliotecas não irão funcionar.

## Resumo

Nesta aula aprendemos um pouco sobre um dos principais padrões de projeto do iOS, o Delegate. Vimos que com ele podemos preencher tabelas e delegar atividades a outras classes. Também conhecemos o CocoaPods, um incrível gerenciador de dependências para iOS. Com ele podemos instalar bibliotecas de terceiro e economizar bastante tempo de desenvolvimento.

Aula 3

Outro recurso bastante importante no Swift é o uso de Closures. Eles são utilizados para escrever blocos de código, como se fossem métodos, que otimizam a escrita de certos algoritmos sem diminuir a clareza ou intenção. Em outras linguagens o Closure pode ser comparado com Callbacks (que falamos no curso de Android).

# Closures

Closures são blocos independentes de funcionalidade que podem ser transferidos de um lugar para o outro e utilizados em seu código. Com eles podemos implementar o tratamento de um objeto em tempo de execução. Em outras palavras, podemos criar métodos e passa-los via parâmetro em algum método ou atribuí-lo em alguma variável/constante. Os closures são utilizados principalmente para realizar operações assíncronas.

## Expreções Closure

Para dar continuidade nos exemplos desta aula, abra novamente o seu Playground e execute os códigos mostrados aqui.

Em Swift podemos criar funções aninhadas, ou seja, é possível definir funções dentro de outra função. Isto é feito para organizar melhor o código ou alguma outra aplicação específica que se faz necessário este uso. Mas convenientemente não é muito aconselhável pela orientação a objetos, pois diminui a legibilidade do código e quanto mais aninhamos funções dentro de outras funções aninhadas, mais complexo se tornará nosso código.

Uma alternativa as funções aninhadas é o uso de closures. Closures são utilizadas para escrever blocos de código *inline* (em uma só linha) em uma sintaxe breve e focada. Closures são utilizados para otimizar a escrita de certos algoritmos sem diminuir sua clareza ou intenção. Os exemplos a seguir ilustram estas otimizações refinando o método de ordenação de arrays **sorted(by:)** com diversas maneiras para realizar a mesma funcionalidade.

### O método de ordenação

O Swift apresenta um método chamado **sorted(by:)**, que classifica um array de valores de um tipo conhecido com base na produção de uma saída ordenada por um closure que você fornece. Quando este método completa o processo de classificação, é retornado um novo array de mesmo tipo que o antigo, porém com seus elementos em uma ordem especificada pelo closure. O array original não é modificado depois deste processo.

O exemplo abaixo usa o método **sorted(by:)** para ordenar um array de Strings em ordem decrescente (de Z a A). Este é o array inicial a ser classificado:

let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]

O método **sorted(by:)** aceita um bloco closure que recebe dois argumentos com mesmo tipo do conteúdo do array e retorna um valor Bool que indica se o primeiro argumento deve ser posicionado no novo array antes do segundo argumento. O closure de classificação deve retornar **true** se o primeiro valor estar antes do segundo e **false** caso contrário.

No exemplo a seguir o array de String está sendo classificado e o closure de classificação deve possuir o tipo **(String, String) -> Bool**.

Uma maneira de providenciar uma closure de classificação é escrever uma função normal com o tipo que corresponde ao tipo que o método **sorted(by:)** aceita. O tipo de uma função é definido pela sua assinatura (conjunto de argumentos) e tipo de retorno. Ao definir esta função, basta passarmos esta via parâmetro do método **sorted(by:).**

func backward(\_ s1: String, \_ s2: String) -> Bool {

  return s1 > s2

}

var reversedNames = names.sorted(by: backward)

// reversedNames é igual a ["Ewa", "Daniella", "Chris", "Barry", "Alex"]

Veja que não invocamos **backward()** mas passamos apenas o **nome** do método via parâmetro de **sorted(by:)**, isto porque este método de classificação não espera um valor ou retorno, mas sim a definição de como ele irá classificar o array.

Se a primeira string (s1) é maior do que a segunda (s2), a função **backward()** irá retornar **true**, indicando que s1 deve aparecer antes de s2 no array classificado. Para caracteres em strings, “maior que” significa “aparece antes no alfabeto que”. Isto significa que “B” é maior do que “A”, e a string “Tom” é maior que a string “Tim”. Como este exemplo ordena as strings na ordem decrescente então “Bruno” será apresentado antes de “Alex”.

## Sintaxe de um closure

A sintaxe de um closure possui a seguinte estrutura básica:

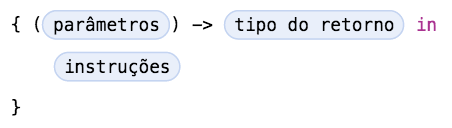


Figura - Sintaxe geral do closure

Veja que esta sintaxe se assemelha a declaração de uma função, porém não precisamos definir um nome a ela (e nem utilizar a palavra reservada **func**).Para definir um closure, especificamos um conjunto de parâmetros, o tipo de retorno e as instruções no interior do bloco.

O exemplo abaixo mostra o exemplo do **sorted(by:)** com o método **backward()** sendo substituído por um closure inline.

reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in

return s1 > s2

})

Note que os parâmetros e retorno deste closure inline são idênticos ao da função **backward()**. Em ambos os casos eles são escrito com **(s1: String, s2: String) -> Bool.** Entretanto, para o caso inline, o parâmetro e o tipo de retorno estão escritos dentro de chaves ({ }) e não fora delas.

O início do corpo do closure é introduzido pela palavra-chave **in**. Esta palavra chave indica que a definição dos parâmetros e tipo de retorno do closure terminou e daí em diante o corpo do closure começa.

Como o corpo do closure é muito curto, ele pode ser escrito em uma única linha:

reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in return s1 > s2 } )

Isto ilustra que em geral a chamada de **sorted(by:)** permanece a mesma. Um par de parênteses envolve inteiramente o argumento do método. Entretanto, este argumento agora é um closure inline.

### Inferindo tipo pelo contexto

Como o closure de classificação é passado via argumento para um método, o Swift consegue inferir o tipo de seus parâmetros e o seu tipo de retorno. O método **sorted(by:)** é chamado de um array de strings, logo seu argumento deve ser a função do tipo **(String, String) -> Bool**, pois é isto que o método **sorted(by:)** espera. Isto significa que os tipos **(String, String)** e **Bool** não precisam ser escritos como parte da definição do closure. Uma vez que todos os tipos podem ser inferidos, a seta de retorno (->) e os parênteses que ficam em torno dos nomes dos parâmetros também podem ser omitidos. Veja a diferença:

reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 } )

Sempre será possível inferir os tipos de parâmetros e o tipo de retorno ao fornecer um closure via argumento. Como resultado você nunca precisará escrever a definição de um closure inline em sua forma completa.

Você ainda pode continuar explicitando os tipos se desejar e isto é realmente o desejável. A seguir você verá formas de resumir ainda mais a definição de um closure, mas a sua forma completa é a melhor maneira para evitar ambiguidade e confusão para quando estiver lendo seu código. No caso do método **sorted(by:)** o propósito do closure é reduzir ou limpar a quantidade de código que uma função de ordenação normalmente requer.

### Retornos implícitos em closures com uma única expressão

Closures que podem ser definidos em apenas uma linha podem implicitar o retorno de sua expressão omitindo a palavra chave **return** de sua declaração. Veja a diferença:

reversedNames = names.sorted(by: { s1, s2 in s1 > s2 } )

Aqui o tipo de função dos argumentos do método **sorted(by:)** torna claro que um valor **Bool** deve ser devolvido pelo closure. Uma vez que o corpo do closure contém apenas uma expressão (**s1 > s2**) que devolve um valor *booleano*, não existe ambiguidade e a palavra **return** pode ser omitida. Mas ainda podemos reduzir isto.

### Nomes de argumentos taquigráficos

O Swift automaticamente fornece taquigrafias (abreviações) do nome dos argumentos dos métodos para *closures inline*. Estes podem ser utilizados para se referir aos valores dos argumentos dos closures pelos nomes **$0, $1, $2** e por aí em diante.

Se você usa estes argumentos de taquigrafia, você pode omitir a lista de argumentos da definição de uma *closure* e o número e tipo do argumento taquigráfico será inferido para o tipo de função que se espera. Com isto a palavra-chave **in** também pode ser omitida pois agora a *closure* está definida inteiramente pelo seu corpo. Veja:

reversedNames = names.sorted(by: { $0 > $1 } )

Aqui, **$0** e **$1** referem-se ao primeiro e segundo argumento String do *closure*. Mas acalme-se, existe uma última forma de redução.

### Métodos de operador

Na verdade há ainda esta última maneira de reduzir a definição de um *closure*. Em Swift os operadores condicionais básicos (==, >, <, <=, >=) se comportam como métodos. Exemplificando com uma String, se a condição **s1 > s2** fosse equivalente a **s1.maiorQue(s2)**, inclusive o Swift nos dá a possibilidade de sobrescrevermos os operadores e adicionar o comportamento/retorno que bem quisermos, mas isto fica para uma pesquisa mais avançada que você pode fazer. No caso da comparação de uma String, este método espera um parâmetro do tipo String, que é a string a ser comparada, e emite um retorno do tipo Bool, afirmando se a comparação é verdadeira ou não. Sendo assim o Swift consegue inferir que os valores que estarão do lado esquerdo e direito do operador **>** serão do tipo String. Como estamos comparando um conjunto de strings dentro de um array de strings, o método **sorted(by:)** consegue inferir tudo e a única coisa que precisamos nos preocupar é como o método irá classificar estes valores.

Portanto você pode apenas passar o operador **>** como parâmetro do método:

reversedNames = names.sorted(by: >)

Mais uma vez ressaltamos que a melhor coisa a se fazer é utilizar a forma completa de declaração de um closure. A menos que estejamos implementando um método banal como este de classificação, devemos sempre priorizar pela legibilidade, então escolha sempre pelo que é mais legível para a próxima pessoa que irá ler seu código, seja este o modo resumido ou completo.

## Trailling Closures

Se você precisa passar um closure como argumento final de uma função, mas este closure é muito extenso, ou caso você queira reaproveita-lo em diversos lugares diferentes, você pode escrevê-lo como um trailling closure, ou em uma tradução direta, um “closure de arraste”, que leva este nome por literalmente o arrastamos para fora do método. Uma tralling closure é escrita depois dos parênteses da chamada da função, mas ainda continua sendo um argumento da função. Quando você usa a sintaxe de trailling closure, não é necessário escrever o rótulo do argumento para a parte do closure na chamada da função.

Primeiro veja uma a definição de um closure convencional que não possui parâmetros e nem retorno:

func someFunctionThatTakesAClosure(closure: () -> Void) {

  // O corpo da função entra aqui

}

Em seguida, a sua chamada convencional:

someFunctionThatTakesAClosure(closure: {

  // O corpo do método entra aqui

})

Agora veja com a sintaxe de trailling closure:

someFunctionThatTakesAClosure() {

  // trailing closure's body goes here

}

A nossa clássica função **sorted(by:)**  também pode ser escrita com a sintaxe de trailling closure:

reversedNames = names.sorted() { $0 > $1 }

Se a closure é o único argumento do método, você pode fornecer esta expressão como trailling closure sem precisar escrever o par de parênteses após o nome do método. Como os parênteses estarão vazios, basta omitirmos eles. A função **sorted(by:)** se enquadra neste caso:

reversedNames = names.sorted { $0 > $1 }

Aprendemos o básico sobre os closures. Existem ainda conceitos mais avançados na documentação do Swift, mas os descritos aqui irão cobrir a maioria das necessidades na hora de desenvolver um app iOS.

Agora vamos aplicar estes conceitos no nosso projeto:

### Preparando a camada de serviço

No curso de Android aprendemos o conceito de Networking, o que é REST e HTTP. Antes de entrarmos nestes conceitos vamos relembrar como eram feitas as requisições lá.

Primeiramente utilizamos a biblioteca Ion, que a partir de uma série de configurações e parâmetros, conseguíamos fazer requisições GET de uma determinada URL.

DICA: Relembrando o conceito de métodos HTTP, o GET obtém dados provenientes de uma URL. Estes dados geralmente vêm no formato JSON para comunicação com aplicativos mobile.

Outro ponto é que as chamadas eram feitas de maneira assíncrona, ou seja, após a requisição ter sido feita, o código seguia e quando uma resposta era obtida o bloco de código do Ion implementado como um observer realizava o tratamento necessário.

O lugar que queremos chegar é que iremos reproduzir isto através de chamadas assíncronas utilizando os closures.

## Resumo

Vimos nesta aula a utilização e utilidade de closure. Apesar de parecer diferente de tudo que vimos em Java, mas podemos ver uma grande vantagem. Os closure permitem que injetemos trechos de código para “ensinar” como o objeto deve se comportar mediante a certas situações por exemplo: **“como ordenar este array?”**. Em comparação com Java, utilizávamos interfaces e podíamos implementá-las inline, porém aumenta consideravelmente a quantidade de arquivos e esforço gerados em comparação com os closures no Swift.