Unidade 5

Aula 1

# Protocolos

Protocolos são como interfaces. Eles definem um esqueleto de métodos, propriedades e outros requisitos que conformam com uma tarefa em particular ou um pedaço de funcionalidade. O protocolo podem então ser adotados por uma classe, estrutura (struct) ou enumerações (enum) para prover 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 Protocol

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 um 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 propriedade 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 vim com o modificador **static**, mas isto não significa que o protocolo possui uma variável estática, as 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 comple”) e sua propriedade (fullName, que significa “Nome completo”).

O protocolo estabelece que qualquer um que conforme com FullyNamed deve ter ma 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, o que representa uma pessoa com nome específica. 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. (Swift relata um erro em tempo de compilação se um requisito protocolo não é cumprido.)

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

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"

DICA: 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 nave estelar. 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 anexa-o 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 defini um protocolo com um único requisito de método de instância:

protocol RandomNumberGenerator {

  func random() -> Double

}

O protocolo **RandonNumberGeneretor** que o tipo que o conforme tenha um método chamado **random()**, que deve retornar um valor Double sempre que é 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 com uma interface em Java?

### Initializers

Os protocolos podem exigir Initializers (inicializadores) específicos a serem implementadas e conformando. 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 em 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 dado para ser utilizado no seu código.

Pondendo 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 conincidir 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 n-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 uma dado.

A propriedade generator é do tipo RandomNumberGenerator. Portanto, você pode atribuir a 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 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 ou adicionar funcionalidade adicional que tipos conformes pode tirar vantagem. Veremos isto mais adiante em Extensões de Protocolo.

### Sintaxe da Extension

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

extension SomeType {

  // Novas funcionalidades a serem adicionadas a SomeType vão aqui

}

Uma extenção 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 adoptar 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 de mais cedo pode ser estendida para adoptar 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 extender. Estamos acostumado 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 segue 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 ultimas classes criadas anteriormente:

let objects: [AnyObject] = [

  Circle(radius: 2.0),

  Country(area: 243\_610),

  Animal(legs: 4)

]

A array **objects** pode agora ser iterado, e cada objeto no array pode ser verificado 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

Aula 2

# 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 torna nosso código.

Uma alternativa as funções aninhadas é o uso de closures. Closures são utilizadas para escrever blocos de código inline em uma sintaxe breve e focada. Closures são utilizados para otimizar a escrita de certos algoritmos sem diminuir sua clareza ou intenção. Os exemplos abaixo 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, com seus elementos em uma ordem especificada pelo closure. O array original não é modificado depois deste processo.

O exemplo abaixo usa o método usa o método **sorted(by:)** para ordenar um array de Strings em ordem alfabética reversa (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 deve estar antes do segundo, e **false** caso contrário.

Este exemplo está classificando um array de String, e este 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)

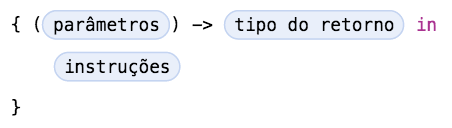
// reversedNames é igual a ["Ewa", "Daniella", "Chris", "Barry", "Alex"]

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 alfabética inversa então “Bruno” será apresentado antes de “Alex”.

## Sintaxe de um closure

Closure expression syntax has the following general form:



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 é idêntico 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 inicio 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:)** permance 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 passar 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 realmente isto é o desejável. Abaixo 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 quer 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 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** referen-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 ultima maneira de reduzir a definição de um closure. Em Swift, os operadores condicionais básicos (==, >, <, <=, >=) se comportam como métodos. Com o exemplo de uma String é como 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 ressaltando que a melhor coisa a se fazer é utilizar a forma completa de declaração de um closure. Ao 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 logo, 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”. Leva este nome pois nós literalmente a 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 relebrar 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 proveniente 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.

Nesta aula ainda não iremos implementar os métodos de Networking, mas iremos apenas preparar as chamadas destes blocos nos View Controllers.

Aula 2

# Delegation

Na unidade 4 falamos um pouco sobre delegates e data source para implementação das Table Views. Pois bem, chegou a hora de explorar mais a fundo este conceito que é utilizado exaustivamente no desenvolvimento de apps iOS. Os Delegates são feitos por meio de protocolos, e já que aprendemos as várias funcionalidades dos protocolos, vamos utilizá-lo para aprender este novo padrão de projeto: o Delegate.