Unidade 2

Aula 1

# Swift – Parte 1

Nesta aula vamos aprender o essencial sobre a linguagem Swift. Primeiro vamos apresentar uma introdução e depois iremos aprender um pouco de sua sintaxe. Vamos lá!

## O que é Swift?

Swift é uma linguagem de programação de propósito geral construída utilizando uma abordagem moderna que se adequam aos padrões de segurança, desempenho e design de um software.

O objetivo do projeto Swift é criar a melhor linguagem disponível para usos que vão desde programação de sistemas, para aplicativos móveis e de desktop, até serviços em nuvem de alta escala. Mais importante ainda, Swift foi concebido para tornar a escrita e manutenção de programas fáceis para os desenvolvedores. Para atingir este objetivo, o código Swift deve ser:

* **Seguro:** Quando temos uma linguagem fácil e óbvia de se escrever, ela também deve se comportar de maneira segura. Comportamento instável é o inimigo da segurança e erros do desenvolvedor devem ser travados antes que o software esteja em ambiente de produção (lançado para o mercado).
* **Rápido:** Swift foi criado como um substituto para as linguagens baseadas em C (C, C++, e Objective-C). Para assumir esta responsabilidade, espera-se que o Swift possua um desempenho melhor ou igual a estas linguagens.
* **Expressivo:** Décadas de avanço na ciência da computação vieram para oferecer ao Swift uma sintaxe que é uma alegria para usar, com características modernas que desenvolvedores esperam. Mas Swift nunca foi terminado, pois ainda está em desenvolvimento e alterações constantes. Vamos acompanhar os avanços da língua e abraçar o que funciona, em constante evolução para fazer o Swift ainda melhor.

### Características

Swift inclui recursos que tornam código mais fácil de ler e escrever, dando ao desenvolvedor o controle necessário em uma verdadeira linguagem de programação. Swift suporta **tipos inferidos** para tornar o código mais limpo e menos propenso a erros, e módulos de eliminar os cabeçalhos e fornecem **namespaces**. A memória é gerenciada automaticamente, e você nem precisa digitar ponto e vírgula. Swift também toma emprestado de outras línguas, por exemplo, parâmetros nomeados trazidos de Objective-C são expressos em uma sintaxe limpa que faz APIs em Swift fácil de ler e manter.

As características do Swift são projetadas criar uma linguagem poderoso, mas ao mesmo tempo é divertida de usar. Alguns recursos adicionais de Swift incluem:

* Closures unificadas com ponteiros de função
* Tuplas e múltiplos valores de retorno
* Generics
* Iteração rápida e concisa em um espaço ou uma coleção
* Estruturas que suportam **métodos**, **extensões** e **protocolos**
* Padrões de programação funcional, por exemplo, map e filter
* Tratamento de erros em tempo de execução
* Fluxo de controle avançado com palavras-chave **do**, **guard**, **defer** e **repeat**

Vamos então aprender um pouco de Swift. Abra um Playground e siga os exemplos seguintes.

As lições a seguir, serão guiadas a partir de um Playground. Como dizemos anteriormente, o Playground é um tipo de arquivo que permite que você interaja com o código diretamente no Xcode, com resultado imediato, sem a necessidade de simuladores. Playgrounds são ótimos para aprender e experimentar, então sinta-se a vontade de explorar e ir além do que for apresentado em aula. Vamos então começar a aprender Swift 3.0.

## Tipos básicos

Uma **constante** é um valor que permanece imutável após ter sido declarada pela primeira vez, enquanto que uma **variável** é um valor que pode ser alterado livremente. Se você sabe que um valor não precisa ser alterado no seu código, declare-o como uma constante ao invés de uma variável.

Para fazer isto, use let para declarar uma constante e **var** para uma variável.

var myVariable = 42

myVariable = 50

let myConstant = 42

Cada constante ou variável em Swift possui um tipo, mas nem sempre você precisa definir este tipo explicitamente. Fornecendo um valor ao criar uma constante ou variável permite que o compilador infira seu tipo. No exemplo acima, o compilador infere que myVariable é um número inteiro (Int), por que seu valor é um número inteiro. Isto é chamado de **Inferência de Tipos.** Uma vez que uma constante ou variável tem um tipo, este não pode ser mais alterado.

Se o valor inicial não fornece informações suficientes (não fornece nenhum valor) para que a inferência de tipo seja feita, você deve especificar o tipo, escrevendo-o após a variável, separados por dois pontos.

let implicitInteger = 70

let implicitDouble = 70.0

let explicitDouble: Double = 70

**DICA**: No Xcode, use o comando **Option + clique** no nome de uma constante ou variável para ver seu tipo inferido. Tente fazer isto com as constantes acima.

Valores nunca podem ser convertidos implicitamente. Se você precisa fazer uma conversão de um tipo para outro, faça explicitamente uma instância do tipo desejado. No exemplo, você pode converter um Int para uma **String.**

let label = "The width is "

let width = 94

let widthLabel = label + String(width)

**DICA**: Tente remover a conversão para **String** da última linha. Que erro acontece?

Há uma maneira ainda mais simples para incluir valores em strings: Escrever o valor entre parênteses, e escrever uma barra invertida ( \) antes dos parênteses. Isto é conhecido como *interpolação de string*.

let apples = 3

let oranges = 5

let appleSummary = "I have \(apples) apples."

let fruitSummary = "I have \(apples + oranges) pieces of fruit."

### Opcionais

Use **optionals** para trabalhar com valores que podem estar ausentes. Um valor opcional pode conter um valor ou pode conter **nil** (nenhum valor) para indicar que o valor está ausente. Coloque um ponto de interrogação ( **?**) após o tipo de um valor para marcá-lo como opcional.

* let optionalInt: Int? = 9

Para obter o valor subjacente de um opcional, “desembrulhe-o”. Você vai aprender a desembrulhar uma variável mais tarde, mas a maneira mais simples para fazer isto é usando o operador **unwrap ( !).** Só utilize o operador **unwrap** se tiver certeza que o valor subjacente não é nil**.**

let actualInt: Int = optionalInt!

Opcionais estão presentes em Swift, e são muito úteis para muitas situações em que um valor pode ou não estar presente. Eles são especialmente útil para tentativas de conversões.

var myString = "7"

var possibleInt = Int(myString)

print(possibleInt)

### Arrays

Uma matriz (ou mais popularmente chamado de array) é um tipo de dados que mantém o controle de uma coleção ordenada de itens. Crie arrays utilizando colchetes ( []), e acesse seus elementos escrevendo o índice entre estes colchetes. Como em Java, arrays sempre começam no índice 0.

var ratingList = ["Poor", "Fine", "Good", "Excellent"]

ratingList[1] = "OK"

ratingList

Para criar uma matriz vazia, use a sintaxe de inicialização. Você vai aprender mais sobre inicialização daqui a pouco.

// Cria um array vazio

let emptyArray = [String]()

Você vai notar que o código acima tem um **comentário**. Como você já conhece do Java, um comentário é um pedaço de texto em um arquivo de código fonte que não seja compilado como parte do programa, mas fornece o contexto ou informações úteis sobre peças individuais de código. Um comentário de uma única linha aparece após duas barras ( //) e um comentário de várias linhas aparece entre um conjunto de barras e asteriscos ( /\* ... \*/). Você vai ver e gravar os dois tipos de comentários durante todo o código-fonte nas aulas.

## Controle de fluxo

Swift tem dois tipos de demonstrações de fluxo de controle. *As indicações condicionais* , como if e switch, verifica se a condição é verdadeira, isto é, se o seu valor avaliado como o booleano é true, antes de executar um pedaço de código. *Loops* , como for-in e while, executam um trecho de código em determinadas vezes.

### If, else

Uma declaração if verifica se uma determinada condição é verdadeira, e se for, o código dentro da declaração if é executado. Você pode adicionar uma clausula else cláusula a uma if declaração para definir o comportamento mais complexo. Uma else cláusula pode ser usado para cadeia if declarações juntos, ou ele pode ficar em sua própria, caso em que a else cláusula é executada se nenhuma das encadeadas if declarações avaliar a true.

let number = 23

if number < 10 {

  print("The number is small")

} else if number > 100 {

  print("The number is pretty big")

} else {

  print("The number is between 10 and 100")

}

**DICA**: Mude o valor de number para ver os outros resultados dos prints.

As declarações podem ser aninhadas para o tratamento de comportamentos mais complexos e interessantes em um programa. Aqui está um exemplo da declaração if com uma cláusula else aninhada em uma declaração for-in (que percorre ordenadamente cada item um uma coleção, um por um).

let individualScores = [75, 43, 103, 87, 12]

var teamScore = 0

for score in individualScores {

  if score > 50 {

      teamScore += 3

  } else {

      teamScore += 1

  }

}

print(teamScore)

### If - let

Use um *optional binding* (if-let) em uma declaração if para verificar se um *optional* contém um valor.

var optionalName: String? = "John Appleseed"

var greeting = "Hello!"

if let name = optionalName {

  greeting = "Hello, \(name)"

}

DICA: Mude o valor de optionalName para nil. Qual vai ser a saudação (greeting) obtida? Adicione um else, para tratar o caso em que optionalName é nil.

Se o valor opcional é nil, a condição é false, e o código entre chaves é ignorada. Caso contrário, o valor opcional é desembrulhado e atribuído a constante definida com let (no exemplo acima é name), o que torna o valor desembrulhado disponível dentro do bloco de código.

### Where

Você pode usar uma única instrução if para ligar vários valores. Uma cláusula where pode ser adicionada para ampliar o escopo da instrução condicional. Neste caso, o if é executado apenas se a as condições forem true para todos estes valores definidos no where.

var optionalHello: String? = "Hello"

if let hello = optionalHello where hello.hasPrefix("H"), let name = optionalName {

  greeting = "\(hello), \(name)"

}

### Switch

Switches em Swift são muito poderosos. Uma declaração switch suporta qualquer tipo de dado e uma grande variedade de operações de comparação não se limitando a números inteiros e testes de igualdade como em Java. Neste exemplo, o switch passa sobre o valor da string vegetable, comparando o valor de cada um dos seus casos e realizando o tratamento correspondente.

let vegetable = "red pepper"

switch vegetable {

case "celery":

  let vegetableComment = "Add some raisins and make ants on a log."

case "cucumber", "watercress":

  let vegetableComment = "That would make a good tea sandwich."

case let x where x.hasSuffix("pepper"):

  let vegetableComment = "Is it a spicy \(x)?"

default:

  let vegetableComment = "Everything tastes good in soup."

}

DICA: Tente remover o caso default. Que erro você recebe?

Note como let pode ser usado em um teste padrão para atribuir o valor correpondente a parte de da constante vegetableComment. Assim como em um if, a cláusula where pode ser adicionada a um caso (case) para ampliar o escopo da instrução condicional. No entanto, ao contrário de um if, um switch-case que tem múltiplas condições separados por vírgulas (case "cucumber", "watercress":) é executado quando qualquer uma das condições é atendida.

Depois de executar o código dentro do switch-case correspondente, o programa sai switch-case. A execução não continua para o próximo case, para que você não precisa quebra-la (break) explicitamente no final do código de cada case.

As instruções switch podem ser exaustivas, e demos definir um caso default ao menos que todas os possíveis casos do switch sejam cobertos. O uso de enumerações (enum) pode facilitar neste caso, já que o Xcode completa automaticamente os casos do switch relacionando-os com os casos da enumeração.

### Loops

Agora falando de lopps for, em Swift é um pouco diferente. Você pode manter um índice em um loop usando um *Range* (gama). Use o *operador semi-aberto gama* ( ..<) para fazer uma série de índices.

var firstForLoop = 0

for i in 0..<4 {

  firstForLoop += i

}

print(firstForLoop)

O operador de range semi-aberto ( ..<) não inclui o número superior, de modo que este intervalo vai desde 0 a 3 para um total de quatro iterações do loop. Use o *operador de range fechado* ( ...) para fazer um intervalo que inclui ambos os valores.

var secondForLoop = 0

for \_ in 0...4 {

  secondForLoop += 1

}

print(secondForLoop)

Esta gama vai de 0 a 4 para um total de cinco iterações do loop. O *sublinhado* ( \_) representa um curinga, que você pode usar quando você não precisa saber qual iteração do loop está sendo executado.

## Funções e Métodos

A uma hora destas você já deve saber o que é um método e uma função, mas vamos relembrar.

### Funções

Uma *função* é uma peça reutilizável, com um nome que pode ser referido a partir de muitos lugares em um programa. Em Swift os métodos são chamados de funções pois não está diretamente relacionado a uma classe ou objeto, isto quer dizer que você pode declarar uma função em qualquer em um arquivo **.swift**. Estas funções declaradas fora de classe se tornam globais e podem ser acessadas por qualquer outro objeto.

Use func para declarar uma função. A declaração da função pode incluir zero ou mais *parâmetros* , redigidas de forma name: Type, que são elementos de informação adicional que devem ser passados para a função quando é chamada. Opcionalmente, uma função pode ter um tipo de retorno, escrito após o operador ->, o que indica que tipo de objeto a função retorna. A implementação de uma função vai dentro de um par de chaves ( {}).

func greet(name: String, day: String) -> String {

  return "Hello \(name), today is \(day)."

}

Para chamar uma função, use seu nome seguido de uma lista de *argumentos* (os valores que você passa para satisfazer os parâmetros de uma função) entre parênteses. Quando você chamar uma função, você deve passar cada valor subsequente com o seu nome.

greet(name: "Anna", day: "Tuesday")

greet(name: "Bob", day: "Friday")

greet(name: "Charlie", day: "a nice day")

### Métodos

Funções que são definidas dentro de uma classe específica são chamadas *métodos*. Métodos são explicitamente vinculados ao tipo ou classe onde estão definidos, e só podem ser chamados nesta classe (ou uma de suas subclasses, que você vai aprender em breve). No anterior exemplo switch, você viu um método que é definido na classe String chamado hasSuffix(), mostrando novamente aqui:

let exampleString = "hello"

if exampleString.hasSuffix("lo") {

  print("ends in lo")

}

Como você vê, para chamar um método de uma classe você deve usar a sintaxe de ponto ( .), como em Java. Você pode ver que em certos casos, quando você chama um método ou função não há a necessidade de inserir o nome do primeiro parâmetro, pois o nome do método é semântico o suficiente para isto.

var array = ["apple", "banana", "dragonfruit"]

array.insert("cherry", at: 2)

Você pode declarar métodos assim, utilizando um sublinhado ao invés do nome do parâmetro. O método insert do Array é declarado da seguinte maneira:

public func insert(\_ newElement: Element, at i: Index) { ... }

Veja que podemos também definir um nome de parâmetro (rótulo) diferente do nome da variável a ser manipulada no método. Então na chamada do método insert usamos o at, e dentro do método usamos o i. Neste caso o at é somente um rótulo do parâmetro i.

Coisas pontuais como esta veremos no decorrer do curso.

Aula 2

# Swift – Parte 2

Nesta aula iremos aos passos finais para entendermos o fundamental de Swift. O que você está achando da linguagem até aqui? Parece mais produtiva, não é? Você reparou que quase não precisamos utilizar parênteses? Ou, que o ponto-e-vírgula foi embora de vez? Antes de continuarmos a aula, saiba que se você está acostumado com estas características herdadas de outras linguagens, você pode usá-la. Sim o ponto-e-vírgula e os parênteses das instruções são opcionais. Agora vamos entrar nos assuntos de **orientação a objetos** do Swift.

### Classes e Initializers---------------------------------------------------

Na programação orientada a objeto, o comportamento de um programa baseia-se em grande parte em interações entre objetos. Um *objeto* é uma instância de uma *classe* , o que pode ser pensado como um modelo para esse objeto. Classes armazenar informações adicionais sobre si mesmos na forma de*propriedades* e definir o seu comportamento usando métodos.

Use classseguido pelo nome da classe para definir uma classe. A declaração de propriedade em uma classe é escrito da mesma forma que uma declaração constante ou variável, exceto que é no contexto de uma classe. Da mesma forma, o modo ea função declarações são escritos da mesma maneira. Este exemplo declara uma Shapeclasse com uma numberOfSidespropriedade e um simpleDescription()método.

class Shape {

  var numberOfSides = 0

  func simpleDescription() -> String {

      return "A shape with \(numberOfSides) sides."

  }

}

Criar uma instância de uma classe um objeto-colocando parênteses após o nome da classe. Use a sintaxe de ponto para acessar as propriedades e métodos da instância. Aqui, shapeé um objeto que é uma instância daShapeclasse.

* var shape = Shape()
* shape.numberOfSides = 7
* var shapeDescription = shape.simpleDescription()

Esta Shapeclasse está faltando alguma coisa importante: um inicializador. Um *inicializador* é um método que prepara uma instância de uma classe para uso, que envolve a definição de um valor inicial para cada propriedade e realizar qualquer outra configuração. Use initpara criar uma. Este exemplo define uma nova classe, NamedShapeque tem um inicializador que leva um nome.

* class NamedShape {
* var numberOfSides = 0
* var name: String
* init(name: String) {
* self.name = name
* }
* func simpleDescription() -> String {
* return "A shape with \(numberOfSides) sides."
* }
* }

Observe como selfé usado para distinguir a namepropriedade do nameargumento para o inicializador. Cada propriedade tem um valor atribuído, seja na sua declaração (como numberOfSides) ou na inicializador (comoname).

Você não chama um inicializador por escrito init; você chamá-lo, colocando parênteses com os argumentos apropriados após o nome da classe. Quando você chama um inicializador, você incluir todos os argumentos nomes, juntamente com os seus valores.

* let namedShape = NamedShape(name: "my named shape")

Classes *herdar* o seu comportamento de sua classe pai. Uma classe que herda o comportamento de outra classe é chamada de *subclasse* dessa classe, ea classe pai é chamado de uma *superclasse* . Subclasses incluem seu nome superclasse depois de seu nome da classe, separados por dois pontos. Uma classe pode herdar de apenas uma superclasse, embora essa classe pode herdar de outra superclasse, e assim por diante, resultando em uma *hierarquia de classes* .

Métodos em uma subclasse que *se sobrepõem a* implementação da superclasse são marcados comoverride-overriding um método por acidente, sem override, é detectada pelo compilador como um erro. O compilador também detecta métodos com overrideque realmente não substituir qualquer método na superclasse.

Este exemplo define a Squareclasse, uma subclasse de NamedShape.

* class Square: NamedShape {
* var sideLength: Double
* init(sideLength: Double, name: String) {
* self.sideLength = sideLength
* super.init(name: name)
* numberOfSides = 4
* }
* func area() ->  Double {
* return sideLength \* sideLength
* }
* override func simpleDescription() -> String {
* return "A square with sides of length \(sideLength)."
* }
* }
* let testSquare = Square(sideLength: 5.2, name: "my test square")
* testSquare.area()
* testSquare.simpleDescription()

Observe que o inicializador para a Squareclasse tem três etapas distintas:

1. A definição do valor de propriedades que a subclasse, Squaredeclara.
2. Chamando o inicializador da superclasse, NamedShape.
3. Alterar o valor de propriedades definidas pela superclasse, NamedShape. Qualquer trabalho de configuração adicional que utiliza métodos, getters ou setters também pode ser feito neste momento.

Por vezes, a inicialização de um objecto precisa de falha, tal como quando os valores fornecidos como os argumentos estão fora de um determinado intervalo, ou quando os dados que se espera estar ali está ausente. Initializers que podem falhar para inicializar com êxito um objeto são chamados initializers failable.Um *inicializador failable* pode retornar nilapós a inicialização. Use init?a declarar o initializer failable.

* class Circle: NamedShape {
* var radius: Double
* init?(radius: Double, name: String) {
* self.radius = radius
* super.init(name: name)
* numberOfSides = 1
* if radius <= 0 {
* return nil
* }
* }
* override func simpleDescription() -> String {
* return "A circle with a radius of \(radius)."
* }
* }
* let successfulCircle = Circle(radius: 4.2, name: "successful circle")
* let failedCircle = Circle(radius: -7, name: "failed circle")

Inicializadores também pode ter um número de palavras-chave associadas com eles. Um *inicializador designado* não necessita de quaisquer palavras-chave. Este inicializador atua como um dos initializers primárias para uma classe; qualquer inicializador dentro de uma classe deve finalmente chamada para um inicializador designado.

A conveniencepalavra-chave ao lado de um inicializador indica um *inicializador de conveniência* . Initializers de conveniência são initializers secundárias. Eles podem adicionar comportamento ou personalização adicional, mas deve eventualmente chamar através de um inicializador designado.

A requiredpalavra-chave ao lado de um inicializador indica que cada subclasse da classe que tem que inicializador deve implementar sua própria versão do inicializador (se ele implementa qualquer inicializador).

*Tipo de carcaça* é uma maneira de verificar o tipo de uma instância, e tratar essa instância como se fosse uma superclasse diferente ou subclasse de outro lugar em sua própria hierarquia de classes.

Uma constante ou variável de um determinado tipo de classe pode realmente se referir a uma instância de uma subclasse nos bastidores. Onde você acha que esse é o caso, você pode tentar *downcast* para o tipo de subclasse usando um operador de conversão de tipo.

Porque downcasting pode falhar, o operador de conversão de tipo vem em duas formas diferentes. A forma opcional, as?, retorna um valor opcional do tipo que você está tentando downcast para. A forma forçada, as!, tenta os abatidos e força-desembrulha o resultado como uma única ação composto.

Use o *operador de conversão de tipo opcional* ( as?) quando você não tem certeza se os abatidos terá sucesso. Esta forma de o operador sempre retornará um valor opcional, eo valor será nilse a entrada de ar não foi possível. Isto permite-lhe verificar se há uma entrada de ar bem sucedido.

Use o *operador de conversão de tipo forçado* ( as!) somente quando tiver certeza de que os abatidos sempre terá êxito. Esta forma de o operador irá disparar um erro de execução se você tentar downcast a um tipo de classe incorreta.

Este exemplo mostra a utilização do operador tipo moldável opcional ( as?) para verificar se a forma de uma matriz de formatos é um quadrado ou um triângulo. Está incrementar a contagem do squaresetrianglesvariáveis por um cada vez que o formato correspondente for encontrado, a impressão dos valores no final.

* class Triangle: NamedShape {
* init(sideLength: Double, name: String) {
* super.init(name: name)
* numberOfSides = 3
* }
* }
* let shapesArray = [Triangle(sideLength: 1.5, name: "triangle1"), Triangle(sideLength: 4.2, name: "triangle2"), Square(sideLength: 3.2, name: "square1"), Square(sideLength: 2.7, name: "square2")]
* var squares = 0
* var triangles = 0
* for shape in shapesArray {
* if let square = shape as? Square {
* squares++
* } else if let triangle = shape as? Triangle {
* triangles++
* }
* }
* print("\(squares) squares and \(triangles) triangles.")

EXPERIMENTAR

Tente substituir as?com as!. O erro que você recebe?

### Enumerações e estruturas

Classes não são as únicas maneiras de definir tipos de dados na Swift. Enumerações e estruturas têm capacidades semelhantes às aulas, mas pode ser útil em diferentes contextos.

*Enumerações* definir um tipo comum para um grupo de valores relacionados e permitir-lhe trabalhar com esses valores de uma maneira de tipo seguro dentro de seu código. Enumerações pode ter métodos associados a eles.

Use enumpara criar uma enumeração.

* enum Rank: Int {
* case Ace = 1
* case Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten
* case Jack, Queen, King
* func simpleDescription() -> String {
* switch self {
* case .Ace:
* return "ace"
* case .Jack:
* return "jack"
* case .Queen:
* return "queen"
* case .King:
* return "king"
* default:
* return String(self.rawValue)
* }
* }
* }
* let ace = Rank.Ace
* let aceRawValue = ace.rawValue

No exemplo acima, o tipo de valor bruto da enumeração é Int, então você tem que especificar apenas o primeiro valor bruto. O resto dos valores brutos são atribuídos em ordem. Você também pode usar cordas ou números de ponto flutuante como o tipo crua de uma enumeração. Use a rawValuepropriedade para acessar o valor bruto de um membro de enumeração.

Use o init?(rawValue:)inicializador para fazer uma instância de uma enumeração de um valor bruto.

* if let convertedRank = Rank(rawValue: 3) {
* let threeDescription = convertedRank.simpleDescription()
* }

Os valores de membro de uma enumeração são valores reais, e não apenas uma outra maneira de escrever seus valores brutos. Na verdade, nos casos em que não é um valor bruto significativa, você não tem que fornecer um.

* enum Suit {
* case Spades, Hearts, Diamonds, Clubs
* func simpleDescription() -> String {
* switch self {
* case .Spades:
* return "spades"
* case .Hearts:
* return "hearts"
* case .Diamonds:
* return "diamonds"
* case .Clubs:
* return "clubs"
* }
* }
* }
* let hearts = Suit.Hearts
* let heartsDescription = hearts.simpleDescription()

Observe as duas maneiras que o Heartsmembro da enumeração é acima referidos: Quando um valor é atribuído ao heartsconstante, o membro de enumeração Suit.Heartsé referido por seu nome completo porque a constante não tem um tipo explícito especificado. No interior do comutador, o membro de enumeração é referido pela sua forma abreviada, .Heartsporque o valor de selfjá é conhecido por ser um terno. Você pode usar o formulário a qualquer hora abreviada o tipo do valor já é conhecido.

*Estruturas* apoiar muitos dos mesmos comportamentos como classes, incluindo métodos e inicializadores.Uma das diferenças mais importantes entre estruturas e classes é que as estruturas são sempre copiados quando eles são passados em torno de seu código, mas as classes são passados por referência. Estruturas são grandes para a definição de tipos de dados leves que não precisa ter capacidades como herança e tipo de fundição.

Use structpara criar uma estrutura.

* struct Card {
* var rank: Rank
* var suit: Suit
* func simpleDescription() -> String {
* return "The \(rank.simpleDescription()) of \(suit.simpleDescription())"
* }
* }
* let threeOfSpades = Card(rank: .Three, suit: .Spades)
* let threeOfSpadesDescription = threeOfSpades.simpleDescription()

### protocolos

Um *protocolo* define um modelo de métodos, propriedades e outros requisitos que se adequam a uma tarefa ou um pedaço de funcionalidade particular. O protocolo na verdade não fornecer uma implementação para qualquer um destes requisitos de apenas descreve o que uma implementação será semelhante. O protocolo pode então ser *adotada* por uma classe, estrutura ou enumeração para fornecer uma implementação efectiva dessas exigências. Qualquer tipo que satisfaça os requisitos de um protocolo é dito para *estar em conformidade* com esse protocolo.

Use protocola declarar um protocolo.

* protocol ExampleProtocol {
* var simpleDescription: String { get }
* func adjust()
* }

NOTA

A { get }sequência da simpleDescriptionpropriedade indica que ele é *somente leitura* , o que significa que o valor da propriedade pode ser visto, mas nunca mudou.

Os protocolos podem exigir que tipos conformes têm propriedades específicas de instância, métodos de instância, métodos de tipo, operadores e subscritos. Os protocolos podem exigir métodos de instância e métodos de tipo específico a ser implementado, conformando tipos. 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 de método.

Classes, estruturas e enumerações adotar um protocolo listando seu nome após o seu nome, separados por dois pontos. Um tipo pode adotar qualquer número de protocolos, que aparecem em uma lista separada por vírgulas. Se uma classe tem uma superclasse, nome da superclasse deve aparecer em primeiro lugar na lista, seguido por protocolos. Você conformidade com o protocolo através da implementação de todos os seus requisitos.

Aqui, SimpleClassadota o ExampleProtocolprotocolo, e em conformidade com o protocolo através da implementação da simpleDescriptionpropriedade e adjust()método.

* class SimpleClass: ExampleProtocol {
* var simpleDescription: String = "A very simple class."
* var anotherProperty: Int = 69105
* func adjust() {
* simpleDescription += "  Now 100% adjusted."
* }
* }
* var a = SimpleClass()
* a.adjust()
* let aDescription = a.simpleDescription

Os protocolos são os tipos de primeira classe, o que significa que eles podem ser tratados como outros tipos nomeados. Por exemplo, você pode criar um ExampleProtocolarray e chamar adjust()em cada um dos casos em que (porque qualquer instância dessa matriz seria garantida a implementar adjust(), um dos requisitos do protocolo).

* class SimpleClass2: ExampleProtocol {
* var simpleDescription: String = "Another very simple class."
* func adjust() {
* simpleDescription += "  Adjusted."
* }
* }
* var protocolArray: [ExampleProtocol] = [SimpleClass(), SimpleClass(), SimpleClass2()]
* for instance in protocolArray {
* instance.adjust()
* }
* protocolArray

### Swift e Cocoa Touch

Swift é projetado para fornecer interoperabilidade perfeita com Cocoa Touch, o conjunto da Apple Estruturas você usa para desenvolver aplicativos para iOS. Como você anda pelo resto das lições, que ajuda a ter uma compreensão básica de como Swift interage com Cocoa Touch.

Até agora, você tem trabalhado exclusivamente com tipos de dados da biblioteca padrão Swift. A *biblioteca padrão Swift* é um conjunto de tipos de dados e recursos projetados para Swift e assados para a língua. Tipos como Stringe Arraysão exemplos de tipos de dados que você vê na biblioteca padrão.

* let sampleString: String = "hello"
* let sampleArray: Array = [1, 2, 3.1415, 23, 42]

EXPERIMENTAR

Leia sobre tipos na biblioteca padrão de Option-clicando no tipo no Xcode. Opção clique sobre StringeArrayno código acima enquanto olha para este playground no Xcode.

Ao escrever aplicativos iOS, você poderá usar mais do que a biblioteca padrão Swift. Uma das estruturas mais frequentemente utilizados no desenvolvimento de aplicativos iOS é UIKit. *UIKit* contém classes úteis para trabalhar com o *(interface de usuário) UI* camada de seu aplicativo.

Para ter acesso a UIKit, basta importá-lo como um módulo em qualquer arquivo Swift ou playground.

* import UIKit

Depois de importar UIKit, você pode usar a sintaxe de Swift com tipos de UIKit e com os seus métodos, propriedades, e assim por diante.

* let redSquare = UIView(frame: CGRect(x: 0, y: 0, width: 44, height: 44))
* redSquare.backgroundColor = UIColor.redColor()

Muitas das classes você vai ser introduzidas no lições vêm de UIKit, então você verá essa declaração de importação, muitas vezes.

Com esta amplitude de conhecimento sobre Swift, você está prestes a saltar para fazer um app de pleno direito na próxima lição. Embora esta lição é a extensão do campo de jogos você vai trabalhar com, por agora, lembre-se que eles podem ser uma ferramenta poderosa no desenvolvimento de aplicativos para qualquer coisa de depuração, para visualização de código complexo, a prototipagem rápida.