Unidade 2

Aula 1

# Swift – Parte 1

Nesta aula você vai aprender o essencial sobre a linguagem Swift. Primeiro será apresentada uma introdução e depois você vai aprender um pouco de sua sintaxe.

## O que é Swift?

Swift é uma linguagem de programação de propósito geral, ou seja, não é utilizada apenas para criar aplicativos iOS, mas, como o Java, pode ser usada em diversas aplicações da computação (como *sites*, programas em *desktop*, servidores, jogos etc.). É construída utilizando uma abordagem moderna que se adequa 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 desktop até serviços em nuvem de alta escala. Mais importante ainda, a Swift foi concebida para tornar a escrita e a manutenção de programas fáceis para os desenvolvedores. Para atingir esse objetivo, o código Swift deve ser:

* **Seguro:** quando há uma linguagem fácil e óbvia de se escrever, ela também deve se comportar de maneira segura. Comportamento instável é o inimigo da segurança e os erros do desenvolvedor devem ser corrigidos antes que o *software* esteja em ambiente de produção (lançado para o mercado).
* **Rápido: a** Swift foi criada como uma substituta para as linguagens baseadas em C (C, C++, e Objective-C). Para assumir essa responsabilidade, espera-se que a Swift possua um desempenho melhor ou igual a essas linguagens.
* **Expressivo:** décadas de avanço na Ciência da Computação vieram para oferecer à Swift as características de linguagens de programação modernas que os desenvolvedores esperam. Você verá em breve de quais modernidades se tratam. A Swift ainda está em desenvolvimento e sofre alterações constantes, então é possível esperar novidades ainda melhores que a Swift já proporciona.

### Características

Alguns recursos adicionais de Swift que você aprenderá no decorrer do curso incluem:

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

Você vai 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 dito 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 a programar, então se sinta à vontade para explorar e ir além do que for apresentado nas aulas. Agora você vai começar a aprender Swift 3.0.

A Apple tem uma documentação completa sobre Swift, porém não está totalmente atualizada para Swift 3, então será feito um compilado do essencial da Swift descrita nesta documentação, atualizando para Swift 3 para você.

## Tipos básicos

Uma **constante** é um valor que permanece imutável após ter sido declarado 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 isso, 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 de valor, mas nem sempre você precisa definir este tipo explicitamente. Fornecer 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), pois seu valor é um número inteiro. Isso 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 integerImplicito = 70

let doubleImplicito = 70.0

var doubleExplicito: Double = 70

**DICA**: No Xcode, use o comando **Option + clique** no nome de uma constante ou variável para ver seu tipo inferido. Tente fazer isso com as constantes e variáveis acima. Veja que apenas a variável doubleExplicito possui um tipo explícito, pois foi declarada com o tipo Double em sua declaração.

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 deles. Isso 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 isso é usando o operador ***unwrap* ( !).** Só utilize o operador ***unwrap*** se tiver certeza de que o valor subjacente não é nil**.**

let actualInt: Int = optionalInt!

DICA: O conceito de embrulhar e desenbrulhar (ou encapsular e desencapsular) em Swift é diferente em relação à Java. Em Swift não há *getters* e *setters* dos seus atributos, pois eles na verdade são **propriedades** (você entenderá a diferença entre propriedades e atributos nas aulas seguintes), então quando se diz embrulhar/desembrulhar (*wrap*/*unwrap*) em Swift, trata-se de estar protegendo e obtendo (respectivamente) os valores de uma variável/constante **opcional.** Quando um valor é protegido, se está afirmando que ele pode ser **nil** ou não (uso da sintaxe “ **?”** visto anteriormente).

Os opcionais estão presentes em Swift e são úteis para situações em que um valor pode ou não estar presente. Eles são especialmente úteis para tentativas de conversões.

var myString = "7"

var possibleInt = Int(myString)

print(possibleInt)

### *Arrays*

Uma matriz (ou mais popularmente chamada de *array*) é um tipo de dado que mantém o controle de uma coleção ordenada de itens. Crie *arrays* utilizando colchetes ([]) e acesse seus elementos escrevendo o índice entre eles. 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 isso daqui a pouco.

// Cria um array vazio

let emptyArray = [String]()

Você vai notar que o código acima tem um **comentário**. Como você já conhece de Java, um comentário é um pedaço de texto em um arquivo de código-fonte que não é 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 utilizar os dois tipos de comentários durante todo o código-fonte nas aulas.

## Controle de fluxo

A linguagem Swift tem dois tipos de demonstrações de fluxo de controle:

* *As indicações condicionais*, como if e switch, que avaliam se a condição é verdadeira antes de executar um pedaço de código.
* *Loops*, como for-in e while, que executam um trecho de código repetidas vezes.

### *If*, *else*

Uma cláusula if verifica se uma determinada condição é verdadeira e, se for, o código dentro dela é executado. Você pode adicionar uma cláusula else a uma cláusula if para manipular condições mais complexas. Uma cláusula else pode ser usada para encadear cláusulas if para avaliar mais condições em casos mais complexos. Entenda com o exemplo:

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 numa 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 ignorado. Caso contrário, o valor opcional é desembrulhado e atribuído à 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. Nesse caso, o if é executado apenas se a as condições forem true para todos os 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 correspondente a parte 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, separadas por vírgulas (case "cucumber", "watercress":), é executado quando qualquer uma das condições é atendida.

Depois de executar o código dentro do switch-case correspondente, a linha de execução do programa sai do switch-case, ou seja, há uma quebra que faz com que os demais casos (case) não sejam executados. A execução não continua para o próximo case para que você não precise quebrá-la (break) explicitamente no final do código de cada case.

As instruções switch podem ser exaustivas e deve-se definir um caso default, ao menos que todas os possíveis casos do switch sejam cobertos (veja o uso do default no exemplo acima). 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.

enum Vegetable {

case carrot

case chicory

case brocoli

case pumpkin

}

let vegetable = Vegetable.carrot

switch vegetable {

case .brocoli, .carrot:

  let vegetableComment = "I like it.”

default:

  let vegetableComment = "Arrgh!!"

}

Acima, um exemplo de uso do switch para permutar entre valores de um enum.

### *Loops*

Agora falando de *loops* for, em Swift, você pode manter um índice em um *loop* usando um *Range* (gama). Use o *operador semiaberto gama* ( ..<) para fazer uma série de índices.

var firstForLoop = 0

for i in 0..<4 {

  firstForLoop += i

}

print(firstForLoop)

O operador de *range* semiaberto ( ..<) não inclui o número superior, de modo que esse intervalo vai de 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)

Essa gama vai de 0 a 4 para um total de cinco iterações do *loop*. O *sublinhado* (\_) é um caractere que você pode usar quando não precisar saber qual iteração do *loop* está sendo executada.

## Funções e métodos

É preciso relembrar o que é um método e uma função para dar continuidade ao seu aprendizado.

### Funções

Uma *função* é um trecho de código de programação 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 estão diretamente relacionados a uma classe ou objeto, isso quer dizer que você pode declarar uma função em qualquer arquivo **.swift**. Essas funções declaradas fora da classe tornam-se globais e podem ser acessadas por qualquer outro objeto.

Use func para declarar uma função. A declaração da função pode incluir nenhum ou mais *parâmetros* redigidos de forma name: Type, que são elementos de informação adicionais que devem ser passados para a função quando são chamados. Opcionalmente, uma função pode ter um tipo de retorno, escrito após o operador ->, o que indica que tipo de objeto a função retornará. A implementação de uma função é inserida 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 deve fornecer 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*. Eles são explicitamente vinculados ao tipo ou à classe em que estão definidos e só podem ser chamados nesta classe (ou em uma de suas subclasses, que você aprenderá em breve). No exemplo anterior de switch, você viu um método que é definido na classe String, chamado hasSuffix(), mostrado 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 notar que em certos casos, quando você chama um método ou uma função, não há a necessidade de inserir o nome do primeiro parâmetro, pois o nome do método é semântico o suficiente para isso.

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

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

Você pode declarar métodos utilizando um sublinhado em vez do nome do parâmetro. O método *insert* do *Array* é declarado da seguinte maneira:

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

Veja que é possível também definir um nome de parâmetro (rótulo) diferente do nome da variável a ser manipulada no método. Então na chamada do método insert usa-se o at e dentro do método usa-se o i. Nesse caso o at é somente um rótulo do parâmetro i.

No decorrer do curso, você verá mais itens pontuais.

Aula 2

Nesta aula você percorrerá os passos finais para entender o fundamental de Swift. Agora você vai entrar nos assuntos de **orientação a objetos** do Swift.

# Swift – Parte 2

## Classes e *initializers*

Em programação orientada a objetos, o comportamento de um programa baseia-se em grande parte em interações entre objetos. Um *objeto* é uma instância de uma *classe* que pode ser pensada como um modelo para esse objeto. Classes armazenam informações adicionais sobre si mesmas na forma de *propriedades* e definem os seus próprios comportamentos utilizando métodos.

DICA: Em iOS, em vez de atributos de classe, há as propriedades. Esse é um jargão utilizado no iOS desde a linguagem Objective-C. As propriedades diferem-se dos atributos na maneira em que são encapsuladas. As propriedades podem ter valores explícitos e não encapsulados, chamados de **propriedades de armazenamento** (*stored properties*), ou podem ter seus valores encapsulados em sua própria definição, sem a necessidade de utilizar métodos *getters* e *setters*, deixando o código mais limpo. Propriedades desse tipo são chamadas de propriedades computadas (*computed properties*), pois não possuem e nem armazenam valor factível, mas sempre calculam um valor novo no *getter* ou realizam algum cálculo no seu *setter*. Você verá mais sobre isso na Unidade 5.

### Classes

Use a palavra-chave class seguida de um nome para definir uma classe. A declaração das propriedades numa classe é escrita da mesma forma que uma constante ou uma variável, que você aprendeu anteriormente, exceto pelo fato de que essas propriedades são válidas apenas no contexto da classe em que foram declaradas. Da mesma forma, os métodos são declarados como as funções aprendidas na aula passada.

O exemplo a seguir declara uma classe Shape (forma) com a propriedade numberOfSides (número de lados) e o método simpleDescription() (descrição simples).

class Shape {

  var numberOfSides = 0

  func simpleDescription() -> String {

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

  }

}

Criar uma instância de uma classe é uma tarefa bem fácil. Basta colocar um conjunto de abre e fecha parênteses após o nome da classe. Para acessar as propriedades e os métodos da instância, use a sintaxe de ponto. Aqui shape é um objeto que é instância da classe Shape.

var shape = Shape()

shape.numberOfSides = 7

var shapeDescription = shape.simpleDescription()

### *Initializers*

Ainda está faltando uma coisa importante para a classe Shape: um inicializador, que é um método que prepara uma instância de uma classe para o uso, envolvendo a definição de valores iniciais de cada propriedade e outras configurações iniciais do objeto. Trata-se do conhecido **construtor** em Java, porém em Swift há um novo jeito para defini-lo: utilizando a palavra reservada init. Nesse caso o init entra como nome do método.

O exemplo a seguir define uma nova classe NamedShape, que 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 que foi utilizada a palavra reservada self para distinguir a propriedade name do argumento name. Cada propriedade tem um valor atribuído, seja na sua declaração (como em numberOfSides) seja no inicializador (como em name). Mesmo que esse valor seja explicitamente nil, todas as propriedades devem ter um valor atribuído em si.

Ao chamar o inicializador na criação da instância, não se utiliza a palavra init, deve-se chamá-lo colocando os parênteses após o nome da classe. Quando se chama um inicializador, devem ser incluídos todos os argumentos e nomes, juntamente com os seus valores.

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

As classes podem herdar o comportamento de sua classe mãe. Uma classe que herda o comportamento de outra é chamada de **subclasse** e a classe mãe é chamada de **superclasse**. Uma subclasse inclui o nome da superclasse depois de seu nome, separados por dois-pontos (:), como no exemplo abaixo. Uma classe pode herdar apenas uma superclasse, embora esta possa herdar outra superclasse e assim por diante, resultando em uma **hierarquia de classes**.

Métodos de uma subclasse que **sobrepõem** a implementação da superclasse são marcados com a palavra reservada override antecedendo o método. Quando não é anotada a palavra override, o compilador emitirá um erro. O compilador detecta um erro quando se coloca o override desnecessariamente em métodos que de fato não sobrepõem nenhum método da superclasse.

Este exemplo define a classe Square, que é 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 classe Square tem três etapas distintas:

1. A definição do valor da propriedade que a subclasse Square declara (sideLength);
2. A chamada do inicializador da superclasse NamedShape usando a palavra reservada super;
3. A alteração do valor da propriedade que a superclasse NamedShape define (numberOfSides). Qualquer trabalho de configuração adicional, *getters* ou *setters,* também pode ser feito neste momento.

Por vezes, a inicialização de um objeto pode falhar, por exemplo, quando os valores fornecidos como argumentos estão fora de um determinado intervalo ou quando por qualquer descuido os dados recebidos são inválidos ou não esperados.

Inicializadores que podem falhar com êxito são chamados de ***failable initializer*** e podem retornar nil após a inicialização.

Use init? para declarar um *failable initializer*.

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 possuem uma série de modificadores. O inicializador padrão é o **designado** e não precisa de qualquer palavra-chave para determinar isso. Esse inicializador atua primariamente nas classes e todas elas possuem definitivamente um inicializador designado, mesmo que ele esteja em suas superclasses.

O modificador convenience ao lado do inicializador indica **inicialização de conveniência,** ou seja, adiciona comportamento ou personalização inicial, mas deve necessariamente ser chamado por meio de um inicializador designado.

O required indica que cada subclasse deve implementar sua própria versão de inicializador.

## *Casting*

**Conversão de tipo** (*casting*) é uma maneira de verificar o tipo de uma instância e tratá-la como se fosse uma superclasse ou subclasse de sua própria hierarquia de classes.

Nos bastidores, uma constante ou variável de um determinado tipo de classe pode realmente se referir a uma subclasse. Quando você julgar que esse é o caso, pode-se tentar fazer um ***downcast*** para o tipo de subclasse utilizando o operador de conversão de tipo.

Mas fique atento, pois o *downcasting* pode falhar, uma vez que não é garantido que o valor da variável ou constante é do tipo da subclasse que se espera. Nesse caso o operador de conversão de tipo vem em duas formas diferentes. A forma opcional, as?, que retorna um valor opcional do tipo que você está tentando fazer o *downcast*. A forma forçada, as!, que tenta desencapsular o objeto sofrendo o *downcasting* à força.

Use o operador de conversão de tipo opcional quando você não tiver certeza de que os *downcast* terão sucesso. Essa forma de operador sempre retornará um valor opcional e o valor será nil se o *downcast* não for possível. Isso lhe permite verificar se há um *downcast* bem-sucedido.

O operador de conversão de tipo forçada deve ser utilizado somente quando houver certeza de que o *downcast* será bem-sucedido. Essa forma de operador disparará um erro de execução (***crash***) se você tentar fazer um *downcast* para um tipo de classe incorreta.

Este exemplo mostra a utilização do operador de conversão de tipo opcional para verificar se a forma de um *array* de formatos é um quadrado ou um triângulo. A cada correspondência de triângulos ou quadrados, incrementa-se um contador respectivo e faz-se *print* 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.")

DICA: Tente substituir as? com as!. Qual o erro que você recebe?

## Enumerações e estruturas

Classes não são as únicas maneiras de definir tipos de dados em Swift. Enumerações e estruturas têm capacidades semelhantes às classes, mas podem ser úteis em diferentes contextos.

### Enumerações

**Enumerações** definem um tipo comum para um grupo de valores relacionados e permite trabalhar com esses valores de uma maneira segura em seu código. Enumerações podem ter métodos associados a elas.

Use enum para 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 (rawValue) da enumeração é Int, mas você pode definir um enum utilizando outros tipos de classe como Strings. Utilize a propriedade rawValue para obter o valor bruto do membro do enum.

Utilize o inicializador init?(rawValue:) para criar uma instância de uma enumeração a partir de um valor bruto.

if let convertedRank = Rank(rawValue: 3) {

  let threeDescription = convertedRank.simpleDescription()

}

Os valores membros de uma enumeração são valores legítimos e não apenas uma outra maneira de escrever seus valores brutos. Na verdade, quando não é explicitado um tipo para o enum, ele não poderá fornecer um valor bruto.

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 membro Hearts é referido na enumeração acima. Quando um valor é atribuído à constante hearts, o membro Suit.Hearts é referido por seu nome completo porque a constante não tem um tipo explícito, pois a enumeração também não tem. No interior do comutador (switch), o membro da enumeração é referido pela sua forma abreviada .Hearts, uma vez que o valor de self já é conhecido por ser um termo. Você pode utilizar a forma abreviada em qualquer lugar desde que o tipo dos membros da enumeração seja especificado.

### *Structs*

**Estruturas** apresentam muitos dos comportamentos das classes, incluindo métodos e inicializadores. Uma das diferenças mais importantes entre ***structs*** e classes é que as estruturas são sempre copiadas quando passadas em torno do seu código, enquanto as classes são passadas por referência. Estruturas são grandes aliadas para definição de tipos de dados leves que não precisam ter capacidade de herança e *casting*.

Use a palavra reservada struct para 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 a um pedaço de funcionalidade particular. O protocolo na verdade não fornece uma implementação para qualquer um desses requisitos, apenas descreve como uma implementação deverá se assemelhar. O protocolo pode então ser *adotado* por uma classe, estrutura ou enumeração para fornecer uma implementação efetiva dessas exigências. Qualquer um que satisfaça os requisitos de um protocolo é dito que está *em conformidade* com esse protocolo.

O protocolo lembra-lhe de algo? Sim. Eles são as **interfaces** do Java em Swift. O grande diferencial, além do nome, é que os protocolos podem definir propriedades e métodos, enquanto uma interface define apenas métodos. Usar o termo protocolo em vez de interface possui, semanticamente, mais conformidade com seu papel, já que ambos (protocolo e interfaces) são regras que deverão entrar em conformidade.

Use protocol quando declarar um protocolo.

protocol ExampleProtocol {

  var simpleDescription: String { get }

  func adjust()

}

DICA: A sequência { get } da propriedade simpleDescription indica que é somente leitura, o que significa que o valor da propriedade pode ser visto, mas nunca alterado.

O protocolo pode exigir que tipos (classes, estruturas ou enumerações) em conformidade com ele tenham propriedades, métodos e operadores específicos. Esses métodos são escritos como parte da definição do protocolo, exatamente da mesma forma que se definem métodos em uma interface Java, escrevendo somente o cabeçalho do método. Protocolos podem exigir conformidades com variáveis também, diferente das interfaces Java, que só exigem conformidade com métodos.

Classes, estruturas e enumerações podem adotar um protocolo listando o nome dele (do protocolo) após o seu nome (das classes, estruturas ou enumerações), separados por dois-pontos. Um tipo (classes, estruturas ou enumerações) pode adotar qualquer quantidade de protocolos, os quais aparecem em uma lista separados por vírgulas. Se uma classe possui uma superclasse, o nome da superclasse deve aparecer em primeiro lugar na lista, seguido dos protocolos. Você garante a conformidade com o protocolo por meio da implementação de todos os seus requisitos.

Aqui, SimpleClass adota o protocolo ExampleProtocol e entra em conformidade com o protocolo por meio da implementação da propriedade simpleDescription e do método adjust().

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 tipos de primeira classe, o que significa que eles podem ser tratados como outros tipos nomeados. Por exemplo, você pode criar um *array* de ExampleProtocol e chamar adjust() em cada um dos itens dele, pois qualquer item desse *array* deve garantir a implementação do método 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

## Funções (avançado) e *closures*

Relembrando, use func para declarar uma função. Chame uma função pelo seu nome com uma lista de argumentos entre parênteses. Use -> para separar os nomes de parâmetros e tipo de retorno da função.

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

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

}

greet(person: "Bob", day: "Tuesday")

Por padrão, as funções usam seus nomes de parâmetros como etiquetas ou rótulos para seus argumentos. Você pode escrever um rótulo personalizado de um argumento antes do nome do parâmetro, ou escrever um sublinhado para ausência de rótulo no argumento.

func greet(\_ person: String, on day: String) -> String {

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

}

greet("John", on: "Wednesday")

Use uma tupla para definir um valor composto que possibilite retornar vários valores de uma só vez na mesma função. Os elementos de uma tupla podem ser referidos pelo nome ou pelo número.

DICA: Na matemática, uma tupla é uma sequência finita (também chamada de "lista ordenada") de objetos. Em Ciência da Computação, tupla tem três significados distintos: 1- Um objeto de dados que guarda diversos objetos. Em swift a definição matemática faz mas sentido, pois uma tupla Swift não é um objeto, e sim uma lista ordenada e finita de objetos.

func calculateStatistics(scores: [Int]) -> (min: Int, max: Int, sum: Int) {

  var min = scores[0]

  var max = scores[0]

  var sum = 0

  for score in scores {

      if score > max {

          max = score

      } else if score < min {

          min = score

      }

      sum += score

  }

  return (min, max, sum)

}

let statistics = calculateStatistics(scores: [5, 3, 100, 3, 9])

print(statistics.sum)

print(statistics.2)

Funções também podem ter um número variável de argumentos coletados de um *array*.

func sumOf(numbers: Int...) -> Int {

  var sum = 0

  for number in numbers {

      sum += number

  }

  return sum

}

sumOf()

sumOf(numbers: 42, 597, 12)

As funções podem ser aninhadas. Funções aninhadas têm acesso a variáveis ​​que foram declaradas na função externa. Você pode usar funções aninhadas para organizar o código em uma função que é longa ou complexa.

func returnFifteen() -> Int {

  var y = 10

  func add() {

      y += 5

  }

  add()

  return y

}

returnFifteen()

As funções são um tipo de primeira classe. Isso significa que uma função pode retornar outra função como o seu valor.

func makeIncrementer() -> ((Int) -> Int) {

  func addOne(number: Int) -> Int {

      return 1 + number

  }

  return addOne

}

var increment = makeIncrementer()

increment(7)

Uma função pode levar outra função como um dos seus argumentos.

func hasAnyMatches(list: [Int], condition: (Int) -> Bool) -> Bool {

  for item in list {

      if condition(item) {

          return true

      }

  }

  return false

}

func lessThanTen(number: Int) -> Bool {

  return number < 10

}

var numbers = [20, 19, 7, 12]

hasAnyMatches(list: numbers, condition: lessThanTen)

As funções são na verdade um caso especial de *closures*, que são blocos de código que podem ser chamados mais tarde. O código em um *closure* tem acesso a coisas como variáveis e funções que estavam disponíveis no escopo em que o closurefoi criado, mesmo que ele esteja em um escopo diferente do qual é executado. Você viu um exemplo disso já com funções aninhadas. Você pode escrever um fechamento sem um nome cercando o código com chaves ({}). Use in para separar os argumentos e o tipo de retorno do corpo.

numbers.map({

  (number: Int) -> Int in

  let result = 3 \* number

  return result

})