

A Swift Tour



Conhecendo a
Linguagem

Valores
Simples

Funções

Objetos e
Classes

Enumerações e
Estruturas

Simultaneidade

Protocolo e
Extensões

Manipulacao de
erros

Genéricos



Swift



Swift é uma nova linguagem de programação para o desenvolvimento de aplicativos iOS, macOS, watchOS e tvOS. No entanto, muitas partes do Swift serão familiares a partir da sua experiência de desenvolvimento em C e Objective-C.

Swift fornece suas próprias versões de todos os tipos fundamentais de C e Objective-C, incluindo Int para inteiros, Double e Float para valores de ponto flutuante, Bool para valores booleanos e String para dados textuais. O Swift também fornece versões poderosas dos três tipos principais de coleção, Array, Set e Dictionary, conforme descrito em [Tipos de coleção](#).

Como C, o Swift usa variáveis para armazenar e se referir a valores por um nome de identificação. O Swift também faz uso extensivo de variáveis cujos valores não podem ser alterados. Estas são conhecidas como constantes e são muito mais poderosas do que as constantes em C. As constantes são usadas em todo o Swift para tornar o código mais seguro e mais claro quando você trabalha com valores que não precisam ser alterados.

Além dos tipos familiares, o Swift introduz tipos avançados não encontrados no Objective-C, como tuplas. As tuplas permitem que você crie e passe agrupamentos de valores. Você pode usar uma tupla para retornar vários valores de uma função como um único valor composto.

Swift também apresenta tipos opcionais, que lidam com a ausência de um valor. Os opcionais dizem “existe um valor e é igual a x” ou “não existe nenhum valor”. Usar opcionais é semelhante a usar nil com ponteiros em Objective-C, mas eles funcionam para qualquer tipo, não apenas para classes. Os opcionais não são apenas mais seguros e expressivos do que os ponteiros nil em Objective-C, mas também estão no centro de muitos dos recursos mais poderosos do Swift.

Swift é uma linguagem type-safe, o que significa que a linguagem ajuda você a ser claro sobre os tipos de valores com os quais seu código pode trabalhar. Se parte do seu código exigir uma String, o tipo de segurança impede que você passe um Int por engano. Da mesma forma, a segurança de tipo evita que você passe accidentalmente uma String opcional para um trecho de código que requer uma String não opcional. A segurança de tipo ajuda você a detectar e corrigir erros o mais cedo possível no processo de desenvolvimento

A tradição sugere que o primeiro programa em um novo idioma deve imprimir as palavras “Olá, mundo!” na tela. Em Swift, isso pode ser feito em uma única linha:

```
1 print("Hello, world!")
2 // Prints "Hello, world!"
```

Se você escreveu código em C ou Objective-C, essa sintaxe parece familiar para você—em Swift, essa linha de código é um programa completo. Você não precisa importar uma biblioteca separada para funcionalidades como entrada/saída ou manuseio de strings. O código escrito no escopo global é usado como ponto de entrada para o programa, para que você não precise de uma função main(). Você também não precisa escrever ponto-e-vírgula no final de cada declaração.

Este passeio fornece informações suficientes para começar a escrever código em Swift, mostrando como realizar uma variedade de tarefas de programação. Não se preocupe se você não entender algo - tudo o que é apresentado nesta turnê é explicado em detalhes no resto deste livro.

VALORES SIMPLES

Use `let` para fazer uma constante e `var` para fazer uma variável. O valor de uma constante não precisa ser conhecido em tempo de compilação, mas você deve atribuir um valor exatamente uma vez. Isso significa que você pode usar constantes para nomear um valor que você determina uma vez, mas usa em muitos lugares.

```
1 var myVariable = 42
2 myVariable = 50
3 let myConstant = 42
```

Uma constante ou variável deve ter o mesmo tipo do valor que você deseja atribuir a ela. No entanto, você nem sempre precisa escrever o tipo explicitamente. Fornecer um valor ao criar uma constante ou variável permite que o compilador infera seu tipo. No exemplo acima, o compilador infere que `myVariable` é um número inteiro porque seu valor inicial é um número inteiro.

Se o valor inicial não fornecer informações suficientes (ou se não houver um valor inicial), especifique o tipo escrevendo-o após a variável, separada por dois pontos.

```
1 let implicitInteger = 70
2 let implicitDouble = 70.0
3 let explicitDouble: Double = 70
```

Os valores nunca são implicitamente convertidos para outro tipo. Se você precisar converter um valor para um tipo diferente, faça explicitamente uma instância do tipo desejado.

```
1 let label = "The width is "
2 let width = 94
3 let widthLabel = label + String(width)
```

Há uma maneira ainda mais simples de incluir valores em strings: Escreva o valor entre parênteses e escreva uma barra invertida (\) antes dos parênteses. Por exemplo:

```
1 let apples = 3
2 let oranges = 5
3 let appleSummary = "I have \(apples) apples."
4 let fruitSummary = "I have \(apples + oranges) pieces of fruit."
```

Use três aspas duplas (""""") para strings que assumem várias linhas. O recuo no início de cada linha entre aspas é removido, desde que corresponda ao recuo das aspas de fechamento. Por exemplo:

```
1 let quotation = """
2 I said "I have \(apples) apples."
3 And then I said "I have \((apples + oranges) pieces of fruit."
4 """
```

Crie matrizes e dicionários usando colchetes ([]), e acesse seus elementos escrevendo o índice ou a chave entre parênteses. Uma vírgula é permitida após o último elemento.

```
1 var fruits = ["strawberries", "limes", "tangerines"]
2 fruits[1] = "grapes"
3
4 var occupations = [
5     "Malcolm": "Captain",
6     "Kaylee": "Mechanic",
7 ]
8 occupations["Jayne"] = "Public Relations"
```

As matrizes crescem automaticamente à medida que você adiciona elementos.

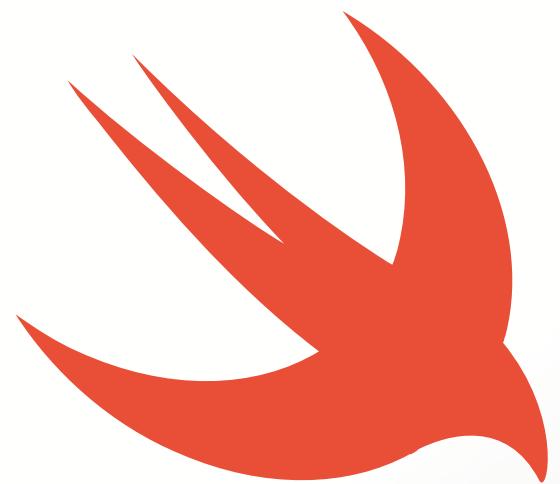
```
1 fruits.append("blueberries")
2 print(fruits)
```

Para criar uma matriz ou dicionário vazio, use a sintaxe do inicializador.

```
1 let emptyArray: [String] = []
2 let emptyDictionary: [String: Float] = [:]
```

Se as informações de tipo puderem ser inferidas, você pode escrever uma matriz vazia como [] e um dicionário vazio como [:]—por exemplo, quando você define um novo valor para uma variável ou passa um argumento para uma função.

```
1 fruits = []
2 occupations = [:]
```



Fluxo de Controle

Use if e switch para fazer condicionais e use for-in, while e repeat-while para fazer loops. Parênteses ao redor da condição ou variável de loop são opcionais.

Suspensórios ao redor do corpo são necessários.

```
1 let individualScores = [75, 43, 103, 87, 12]
2 var teamScore = 0
3 for score in individualScores {
4     if score > 50 {
5         teamScore += 3
6     } else {
7         teamScore += 1
8     }
9 }
10 print(teamScore)
11 // Prints "11"
```

Em uma instrução if, a condicional deve ser uma expressão booleana — isso significa que um código como if score { ... } é um erro, não uma comparação implícita com zero.

Você pode usar if e let juntos para trabalhar com valores que podem estar faltando. Esses valores são representados como opcionais. Um valor opcional contém um valor ou contém nil para indicar que um valor está ausente. Escreva um ponto de interrogação (?) após o tipo de valor para marcar o valor como opcional.

```
1 var optionalString: String? = "Hello"
2 print(optionalString == nil)
3 // Prints "false"
4
5 var optionalName: String? = "John Appleseed"
6 var greeting = "Hello!"
7 if let name = optionalName {
8     greeting = "Hello, \(name)"
9 }
```

Se o valor opcional for nulo, a condicional será falsa e o código entre chaves será ignorado. Caso contrário, o valor opcional é desempacotado e atribuído à constante após let, o que torna o valor desempacotado disponível dentro do bloco de código.

Outra maneira de lidar com valores é fornecer um valor padrão usando o ?? operador. Se o valor opcional estiver faltando, o valor padrão será usado.

```
1 let nickname: String? = nil
2 let fullName: String = "John Appleseed"
3 let informalGreeting = "Hi \(nickname ?? fullName)"
```

Você pode usar uma ortografia mais curta para desembrulhar um valor, usando o mesmo nome para valor unwrapped .

Funções e Encerramentos

Use **func** para declarar uma função. Chame uma função seguindo seu nome com uma lista de argumentos entre parênteses. Use -> para separar os nomes e tipos de parâmetros do tipo de retorno da função.

```
1 func greet(person: String, day: String) -> String {  
2     return "Hello \(person), today is \(day)."  
3 }  
4 greet(person: "Bob", day: "Tuesday")
```

Por padrão, as funções usam seus nomes de parâmetro como rótulos para seus argumentos. Escreva um rótulo de argumento personalizado antes do nome do parâmetro ou escreva _ para não usar nenhum rótulo de argumento.

```
1 func greet(_ person: String, on day: String) -> String {  
2     return "Hello \(person), today is \(day)."  
3 }  
4 greet("John", on: "Wednesday")
```

Use uma tupla para fazer um valor composto—por exemplo, para retornar vários valores de uma função. Os elementos de uma tupla podem ser referidos por nome ou por número.

```
1 func calculateStatistics(scores: [Int]) -> (min: Int, max: Int, sum: Int) {  
2     var min = scores[0]  
3     var max = scores[0]  
4     var sum = 0  
5  
6     for score in scores {  
7         if score > max {  
8             max = score  
9         } else if score < min {  
10            min = score  
11        }  
12        sum += score  
13    }  
14  
15    return (min, max, sum)
```

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

```
1 func returnFifteen() -> Int {  
2     var y = 10  
3     func add() {  
4         y += 5  
5     }  
6     add()  
7     return y  
8 }  
9 returnFifteen()
```

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

```
1 func makeIncrementer() -> ((Int) -> Int) {  
2     func addOne(number: Int) -> Int {  
3         return 1 + number  
4     }  
5     return addOne  
6 }  
7 var increment = makeIncrementer()  
8 increment(7)
```

Uma função pode tomar outra função como um de seus argumentos.

```
1 func hasAnyMatches(list: [Int], condition: (Int) -> Bool) -> Bool {  
2     for item in list {  
3         if condition(item) {  
4             return true  
5         }  
6     }  
7     return false  
8 }  
9 func lessThanTen(number: Int) -> Bool {  
10    return number < 10  
11 }  
12 var numbers = [20, 19, 7, 12]
```

As funções são na verdade um caso especial de fechamentos: blocos de código que podem ser chamados mais tarde. O código em um fechamento tem acesso a coisas como variáveis e funções que estavam disponíveis no escopo onde o fechamento foi criado, mesmo que o fechamento esteja em um escopo diferente quando é executado - você viu um exemplo disso já com funções aninhadas. Você pode escrever um encerramento sem um nome pelo código ao redor com chaves ({}). Use para separar os argumentos e retornar o tipo do corpo.

```
1 numbers.map({ (number: Int) -> Int in  
2     let result = 3 * number  
3     return result  
4 })
```

EXPERIMENTO

Reescreva o fechamento para retornar zero para todos os números ímpares.

Você tem várias opções para escrever encerramentos de forma mais concisa. Quando o tipo de um fechamento já é conhecido, como o retorno de chamada para um delegado, você pode omitir o tipo de seus parâmetros, seu tipo de retorno ou ambos. Fechamentos de instrução única retornam implicitamente o valor de sua única declaração.

```
1 let mappedNumbers = numbers.map({ number in 3 * number })  
2 print(mappedNumbers)  
3 // Prints "[60, 57, 21, 36]"
```

Você pode se referir aos parâmetros por número em vez de por nome - essa abordagem é especialmente útil em fechamentos muito curtos. Um fechamento passado como o último argumento para uma função pode aparecer imediatamente após os parênteses. Quando um fechamento é o único argumento para uma função, você pode omitir os parênteses completamente.

```
1 let sortedNumbers = numbers.sorted { $0 > $1 }  
2 print(sortedNumbers)  
3 // Prints "[20, 19, 12, 7]"
```

OBJETOS E CLASSES

Use class seguido pelo nome da classe para criar uma classe. Uma declaração de propriedade em uma classe é escrita da mesma forma que uma declaração de constante ou variável, exceto que está no contexto de uma classe. Da mesma forma, as declarações de método e função são escritas da mesma maneira.

```
1 class Shape {  
2     var numberOfSides = 0  
3     func simpleDescription() -> String {  
4         return "A shape with \(numberOfSides) sides."  
5     }  
6 }
```

Crie uma instância de uma classe colocando parênteses após o nome da classe. Use a sintaxe de ponto para acessar as propriedades e métodos da instância.

```
1 var shape = Shape()  
2 shape.numberOfSides = 7  
3 var shapeDescription = shape.simpleDescription()
```

Esta versão da classe Shape está faltando algo importante: um inicializador para configurar a classe quando uma instância é criada. Use o init para criar um.

```
1 class NamedShape {  
2     var numberOfSides: Int = 0  
3     var name: String  
4  
5     init(name: String) {  
6         self.name = name  
7     }  
8  
9     func simpleDescription() -> String {  
10        return "A shape with \(numberOfSides) sides."  
11    }  
12 }
```

Observe como **self** é usado para distinguir a propriedade name do argumento name para o inicializador. Os argumentos para o inicializador são passados como uma chamada de função quando você cria uma instância da classe. Cada propriedade precisa de um valor atribuído — seja em sua declaração (como em `numberOfSides`) ou no inicializador (como em `name`).

Use **deinit** para criar um deinicializador se precisar executar alguma limpeza antes que o objeto seja desalocado.

As subclasses incluem seu nome de superclasse após o nome de classe, separadas por dois pontos. Não há necessidade de que as classes subclassifiquem qualquer classe raiz padrão, para que você possa incluir ou omitir uma superclasse conforme necessário.

Os métodos em uma subclasse que substituem a implementação da superclasse são marcados com **override** — substituir um método por acidente, sem substituição, é detectado pelo compilador como um erro. O compilador também detecta métodos com **override** que na verdade não substituem nenhum método na superclasse.

```
1 class Square: NamedShape {
2     var sideLength: Double
3
4     init(sideLength: Double, name: String) {
5         self.sideLength = sideLength
6         super.init(name: name)
7         number0fSides = 4
8     }
9
10    func area() -> Double {
11        return sideLength * sideLength
12    }
13
14    override func simpleDescription() -> String {
15        return "A square with sides of length \(sideLength)."
16    }
17}
18let test = Square(sideLength: 5.2, name: "my test square")
19test.area()
20test.simpleDescription()
```

Além das propriedades simples que são armazenadas, as propriedades podem ter um getter e um setter.

```
1 class EquilateralTriangle: NamedShape {
2     var sideLength: Double = 0.0
3
4     init(sideLength: Double, name: String) {
5         self.sideLength = sideLength
6         super.init(name: name)
7         number_of_sides = 3
8     }
9
10    var perimeter: Double {
11        get {
12            return 3.0 * sideLength
13        }
14        set {
15            sideLength = newValue / 3.0
16        }
17    }
18
19    override func simpleDescription() -> String {
20        return "An equilateral triangle with sides of length \
21        \(sideLength)."
22    }
23
24    var triangle = EquilateralTriangle(sideLength: 3.1, name: "a t
25    print(triangle.perimeter)
26    // Prints "9.3"
27    triangle.perimeter = 9.9
28    print(triangle.sideLength)
29    // Prints "3.30000000000003"
```

No setter para perimeter, o novo valor tem o nome implícito newValue. Você pode fornecer um nome explícito entre parênteses após o set.

Notice that the initializer for the EquilateralTriangle class has three different steps:

1. Definindo o valor das propriedades que a subclasse declara.
2. Chamando o inicializador da superclasse.
3. Alterando o valor das propriedades definidas pela superclasse.

Qualquer trabalho de configuração adicional que use métodos, getters ou setters também pode ser feito neste momento.

Se você não precisa calcular a propriedade, mas ainda precisa fornecer o código que é executado antes e depois de definir um novo valor, use willSet e didSet. O código que você fornece é executado sempre que o valor é alterado fora de um inicializador. Por exemplo, a classe abaixo garante que o comprimento do lado de seu triângulo seja sempre igual ao comprimento do lado de seu quadrado.

```

1 class TriangleAndSquare {
2     var triangle: EquilateralTriangle {
3         willSet {
4             square.sideLength = newValue.sideLength
5         }
6     }
7     var square: Square {
8         willSet {
9             triangle.sideLength = newValue.sideLength
10        }
11    }
12    init(size: Double, name: String) {
13        square = Square(sideLength: size, name: name)
14        triangle = EquilateralTriangle(sideLength: size, name: name)
15    }
16}
17 var triangleAndSquare = TriangleAndSquare(size: 10, name: "another test
shape")
18 print(triangleAndSquare.square.sideLength)
// Prints "10.0"
19 print(triangleAndSquare.triangle.sideLength)
// Prints "10.0"
20 triangleAndSquare.square = Square(sideLength: 50, name: "larger square")
21 print(triangleAndSquare.triangle.sideLength)
// Prints "50.0"

```

Ao trabalhar com valores opcionais, você pode escrever `?` antes de operações como métodos, propriedades e subscriptos. Se o valor antes do `?` é nil, tudo depois do `?` é ignorado e o valor de toda a expressão é nil.

Caso contrário, o valor opcional é desembrulhado, e tudo depois do `?` atua no valor desembrulhado. Em ambos os casos, o valor de toda a expressão é um valor opcional.

```

1 let optionalSquare: Square? = Square(sideLength: 2.5, name: "optional
square")
2 let sideLength = optionalSquare?.sideLength

```

Enumerações e Estruturas

Use **enum** para criar uma enumeração. Como as classes e todos os outros tipos nomeados, as enumerações podem ter métodos associados a elas.

```
1 class TriangleAndSquare {
2     var triangle: EquilateralTriangle {
3         willSet {
4             square.sideLength = newValue.sideLength
5         }
6     }
7     var square: Square {
8         willSet {
9             triangle.sideLength = newValue.sideLength
10        }
11    }
12    init(size: Double, name: String) {
13        square = Square(sideLength: size, name: name)
14        triangle = EquilateralTriangle(sideLength: size, name: name)
15    }
16 }
17 var triangleAndSquare = TriangleAndSquare(size: 10, name: "another test
shape")
18 print(triangleAndSquare.square.sideLength)
19 // Prints "10.0"
20 print(triangleAndSquare.triangle.sideLength)
21 // Prints "10.0"
22 triangleAndSquare.square = Square(sideLength: 50, name: "larger square")
23 print(triangleAndSquare.triangle.sideLength)
24 // Prints "50.0"
```

Por padrão, o Swift atribui os valores brutos começando em zero e incrementando em um a cada vez, mas você pode alterar esse comportamento especificando explicitamente os valores. No exemplo acima, `Ace` recebe explicitamente um valor bruto de 1 e o restante dos valores brutos são atribuídos em ordem. Você também pode usar strings ou números de ponto flutuante como o tipo bruto de uma enumeração. Use a propriedade `rawValue` para acessar o valor bruto de um caso de enumeração.

Use o inicializador `init?(rawValue:)` para criar uma instância de uma enumeração a partir de um valor bruto. Ele retorna o caso de enumeração correspondente ao valor bruto ou nil se não houver classificação correspondente.

```
1 if let convertedRank = Rank(rawValue: 3) {
2     let threeDescription = convertedRank.simpleDescription()
3 }
```

Os valores de caso de uma enumeração são valores reais, não apenas outra maneira de escrever seus valores brutos. Na verdade, nos casos em que não há um valor bruto significativo, você não precisa fornecer um.

```
1 enum Suit {  
2     case spades, hearts, diamonds, clubs  
3  
4     func simpleDescription() -> String {  
5         switch self {  
6             case .spades:  
7                 return "spades"  
8             case .hearts:  
9                 return "hearts"  
10            case .diamonds:  
11                return "diamonds"  
12            case .clubs:  
13                return "clubs"  
14        }  
15    }  
16}  
17 let hearts = Suit.hearts  
18 let heartsDescription = hearts.simpleDescription()
```

Observe as duas maneiras como o caso da enumeração de corações é referido acima: Ao atribuir um valor à constante de corações, o caso de enumeração `Suit.hearts` é referido por seu nome completo porque a constante não tem um tipo explícito especificado. Dentro do `switch`, o caso de enumeração é referido pela forma abreviada `.hearts` porque o valor de `self` já é conhecido como um naipe. Você pode usar a forma abreviada sempre que o tipo do valor já for conhecido.

Se uma enumeração tiver valores brutos, esses valores são determinados como parte da declaração, o que significa que cada instância de um caso de enumeração específico sempre tem o mesmo valor bruto. Outra opção para casos de enumeração é ter valores associados ao caso - esses valores são determinados quando você faz a instância, e eles podem ser diferentes para cada instância de um caso de enumeração. Você pode pensar nos valores associados como se comportando como propriedades armazenadas da instância de caso de enumeração. Por exemplo, considere o caso de solicitar os horários do nascer e do pôr do sol de um servidor. O servidor responde com as informações solicitadas ou responde com uma descrição do que deu errado.

```

1 enum ServerResponse {
2     case result(String, String)
3     case failure(String)
4 }
5
6 let success = ServerResponse.result("6:00 am", "8:09 pm")
7 let failure = ServerResponse.failure("Out of cheese.")
8
9 switch success {
10 case let .result(sunrise, sunset):
11     print("Sunrise is at \(sunrise) and sunset is at \(sunset).")
12 case let .failure(message):
13     print("Failure... \(message)")
14 }
15 // Prints "Sunrise is at 6:00 am and sunset is at 8:09 pm."

```

Observe como os horários do nascer e do pôr do sol são extraídos do valor ServerResponse como parte da correspondência do valor com os casos de alternância.

Use **struct** para criar uma estrutura. As estruturas suportam muitos dos mesmos comportamentos das classes, incluindo métodos e inicializadores. Uma das diferenças mais importantes entre estruturas e classes é que as estruturas são sempre copiadas quando são passadas em seu código, mas as classes são passadas por referência.

```

1 struct Card {
2     var rank: Rank
3     var suit: Suit
4     func simpleDescription() -> String {
5         return "The \(rank.simpleDescription()) of \
6             \(suit.simpleDescription())"
7     }
8     let threeOfSpades = Card(rank: .three, suit: .spades)
9     let threeOfSpadesDescription = threeOfSpades.simpleDescription()

```

Simultaneidade

Use **async** para marcar uma função que é executada de forma assíncrona..

```
1 func fetchUserID(from server: String) async -> Int {  
2     if server == "primary" {  
3         return 97  
4     }  
5     return 501  
6 }
```

Você marca uma chamada para uma função assíncrona escrevendo await na frente dela.

```
1 func getUsername(from server: String) async -> String {  
2     let userID = await fetchUserID(from: server)  
3     if userID == 501 {  
4         return "John Appleseed"  
5     }  
6     return "Guest"  
7 }
```

Você marca uma chamada para uma função assíncrona escrevendo await na frente dela.

```
1 func getUsername(from server: String) async -> String {  
2     let userID = await fetchUserID(from: server)  
3     if userID == 501 {  
4         return "John Appleseed"  
5     }  
6     return "Guest"  
7 }
```

Use **async let** para chamar uma função assíncrona, permitindo que ela seja executada em paralelo com outro código assíncrono. Quando você usar o valor que ele retorna, escreva await.

```
1 func connectUser(to server: String) async {  
2     async let userID = fetchUserID(from: server)  
3     async let username = getUsername(from: server)  
4     let greeting = await "Hello \(username), user ID \(userID)"  
5     print(greeting)  
6 }
```

Use **Task** para chamar funções assíncronas do código síncrono, sem esperar que elas retornem.

```
1 Task {  
2     await connectUser(to: "primary")  
3 }  
4 // Prints "Hello Guest, user ID 97"
```

Protocolos e Extensões

Use `protocol` to declare a protocol.

```
1 protocol ExampleProtocol {  
2     var simpleDescription: String { get }  
3     mutating func adjust()  
4 }
```

Classes, enumerações e estruturas podem adotar protocolos.

```
1 class SimpleClass: ExampleProtocol {  
2     var simpleDescription: String = "A very simple class."  
3     var anotherProperty: Int = 69105  
4     func adjust() {  
5         simpleDescription += " Now 100% adjusted."  
6     }  
7 }  
8 var a = SimpleClass()  
a.adjust()  
let aDescription = a.simpleDescription  
  
12 struct SimpleStructure: ExampleProtocol {  
13     var simpleDescription: String = "A simple structure"  
14     mutating func adjust() {  
15         simpleDescription += " (adjusted)"  
16     }  
17 }  
18 var b = SimpleStructure()  
b.adjust()  
let bDescription = b.simpleDescription
```

Observe o uso da palavra-chave `mutating` na declaração de `SimpleStructure` para marcar um método que modifica a estrutura. A declaração de `SimpleClass` não precisa de nenhum de seus métodos marcados como mutantes porque os métodos em uma classe sempre podem modificar a classe.

Use a extensão para adicionar funcionalidade a um tipo existente, como novos métodos e propriedades computadas. Você pode usar uma extensão para adicionar conformidade de protocolo a um tipo declarado em outro lugar ou até mesmo a um tipo que você importou de uma biblioteca ou estrutura.

```
1 extension Int: ExampleProtocol {  
2     var simpleDescription: String {  
3         return "The number \(self)"  
4     }  
5     mutating func adjust() {  
6         self += 42  
7     }  
8 }  
9 print(7.simpleDescription)  
10 // Prints "The number 7"
```

Você pode usar um nome de protocolo como qualquer outro tipo nomeado—por exemplo, para criar uma coleção de objetos que têm tipos diferentes, mas que todos estão em conformidade com um único protocolo. Quando você trabalha com valores cujo tipo é um tipo de protocolo, métodos fora da definição de protocolo não estão disponíveis.

```
1 let protocolValue: ExampleProtocol = a
2 print(protocolValue.simpleDescription)
3 // Prints "A very simple class. Now 100% adjusted."
4 // print(protocolValue.anotherProperty) // Uncomment to see the error
```

Embora a variável `protocolValue` tenha um tipo de tempo de execução de `SimpleClass`, o compilador a trata como o tipo fornecido de `ExampleProtocol`. Isso significa que você não pode acessar acidentalmente métodos ou propriedades que a classe implementa além de sua conformidade de protocolo.



Manipulação de erros

Você representa erros usando qualquer tipo que adote o protocolo de **Error**.

```
1 enum PrinterError: Error {  
2     case outOfPaper  
3     case noToner  
4     case onFire  
5 }
```

Use **throw** para lançar um erro e **throws** para marcar uma função que pode gerar um erro. Se você lançar um erro em uma função, a função retornará imediatamente e o código que chamou a função manipulará o erro.

```
1 func send(job: Int, toPrinter printerName:  
2         String) throws -> String {  
3     if printerName == "Never Has Toner" {  
4         throw PrinterError.noToner  
5     }  
6     return "Job sent"  
7 }
```

Existem várias maneiras de lidar com erros. Uma maneira é usar do-catch. Dentro do bloco do, você marca o código que pode lançar um erro escrevendo try na frente dele. Dentro do bloco catch, o erro recebe automaticamente o nome error, a menos que você dê um nome diferente

```
1 do {  
2     let printerResponse = try send(job: 1040,  
3                                     toPrinter: "Bi Sheng")  
4     print(printerResponse)  
5 } catch {  
6     print(error)  
7 }  
// Prints "Job sent"
```

Você pode fornecer vários blocos catch que lidam com erros específicos. Você escreve um padrão após o catch, assim como faz após o case em um switch.

```
1 do {
2     let printerResponse = try send(job: 1440,
3         toPrinter: "Gutenberg")
4     print(printerResponse)
5 } catch PrinterError.onFire {
6     print("I'll just put this over here, with the
7     rest of the fire.")
8 } catch let printerError as PrinterError {
9     print("Printer error: \(printerError).")
10 } catch {
11     print(error)
12 }
13 // Prints "Job sent"
```

Outra maneira de lidar com erros é usar a try? para converter o resultado em um opcional. Se a função gerar um erro, o erro específico será descartado e o resultado será nil. Caso contrário, o resultado é um opcional contendo o valor que a função retornou.

```
1 let printerSuccess = try? send(job: 1884, toPrinter:
2     "Mergenthaler")
3 let printerFailure = try? send(job: 1885, toPrinter:
4     "Never Has Toner")
```

Use **defer** para escrever um bloco de código que é executado depois de todos os outros códigos na função, logo antes do retorno da função. O código é executado independentemente de a função lançar um erro. Você pode usar defer para escrever o código de configuração e limpeza um ao lado do outro, mesmo que precisem ser executados em momentos diferentes.

```
1 var fridgeIsOpen = false
2 let fridgeContent = ["milk", "eggs", "leftovers"]
3
4 func fridgeContains(_ food: String) -> Bool {
5     fridgeIsOpen = true
6     defer {
7         fridgeIsOpen = false
8     }
9
10    let result = fridgeContent.contains(food)
11    return result
12 }
13 fridgeContains("banana")
14 print(fridgeIsOpen)
15 // Prints "false"
```

Genéricos

Escreva um nome dentro de **colchetes angulares** para criar uma função ou tipo genérico.

```
1 func makeArray<Item>(repeating item: Item, number0fTimes: Int) -> [Item] {  
2     var result: [Item] = []  
3     for _ in 0..<number0fTimes {  
4         result.append(item)  
5     }  
6     return result  
7 }  
8 makeArray(repeating: "knock", number0fTimes: 4)
```

Você pode fazer formas genéricas de funções e métodos, bem como classes, enumerações e estruturas.

```
1 // Reimplement the Swift standard library's optional type  
2 enum OptionalValue<Wrapped> {  
3     case none  
4     case some(Wrapped)  
5 }  
6 var possibleInteger: OptionalValue<Int> = .none  
possibleInteger = .some(100)
```

Use **where** logo antes do corpo para especificar uma lista de requisitos — por exemplo, para exigir que o tipo implemente um protocolo, para exigir que dois tipos sejam iguais ou para exigir que uma classe tenha uma superclasse específica.

```
1 func anyCommonElements<T: Sequence, U: Sequence>(_ lhs: T, _ rhs: U) ->  
2     Bool  
3     where T.Element: Equatable, T.Element == U.Element  
4 {  
5     for lhsItem in lhs {  
6         for rhsItem in rhs {  
7             if lhsItem == rhsItem {  
8                 return true  
9             }  
10        }  
11    }  
12    return false  
13 }  
anyCommonElements([1, 2, 3], [3])
```

Tradução - A Swift Tour

Fonte:

<https://docs.swift.org/swift-book/GuidedTour/GuidedTour.html>

