

cit 

Hilton Pintor

Desenvolvedor (iOS/tvOS/watchOS)



hiltonpintor@gmail.com

// Aula 03

// Resoluções da Aula 02

// Enumerações

// enum

Enumerações são uma forma de de criar um **novo tipo** para **valores relacionados** de alguma forma.

Apresentam várias funcionalidades que geralmente são atribuídas a classes, como **métodos** e **propriedades** computadas.

// enum - simples

```
enum Familia {  
    case pai  
    case mae  
    case filho  
    case filha  
}
```

// enum - definição

```
enum Familia {  
    case pai  
    case mae  
    case filho  
    case filha  
}
```

Palavra-reservada
(*keyword*)

// enum - definição

```
enum Familia {  
    case pai  
    case mae  
    case filho  
    case filha  
}
```

Palavra-reservada
(*keyword*)

Identificador
(novo tipo)

// enum - definição

```
enum Familia {  
    case pai  
    case mae  
    case filho  
    case filha  
}
```

Palavra-reservada
(*keyword*)

Identificador
(novo tipo)

Palavra-reservada
(*keyword*)

// enum - definição

```
enum Familia {  
    case pai  
    case mae  
    case filho  
    case filha  
}
```

Palavra-reservada
(*keyword*)

Identificador
(novo tipo)

Palavra-reservada
(*keyword*)

Identificador
(possível valor)

```
// enum - acesso
```

```
let membro = Familia.pai  
var mensagem = ""
```

// enum - acesso

```
let membro = Familia.pai  
var mensagem = ""
```

Identificador
(tipo)

```
// enum - acesso
```

```
let membro = Familia.pai  
var mensagem = ""
```

Identificador ▪
(tipo)

// enum - acesso

```
let membro = Familia.pai  
var mensagem = ""
```

Identificador
(tipo)

▪

Identificador
(valor)

```
// enum - switch
```

```
switch membro {  
  case .pai, .mae:  
    mensagem = "primeira geração"  
  case .filho, .filha:  
    mensagem = "segunda geração"  
}
```

```
mensagem // tem valor de "primeira geração"
```


// enum - valores associados

```
enum NovoTipo {  
    case palavra(valor: String)  
    case inteiro(valor: Int)  
}
```

// enum - valores associados

```
let numero = NovoTipo.inteiro(valor: 8)  
mensagem = ""
```

```
// enum - valores associados
```

```
switch numero {  
  
  case .palavra(let valor):  
    mensagem = "a palavra tem valor: \(valor)"  
  
  case .inteiro(let valor):  
    mensagem = "o inteiro tem valor: \(valor)"  
}
```

```
mensagem // "o inteiro tem valor: 8"
```

```
// enum - valores associados
```

```
if case let NovoTipo.inteiro(valor: x) = numero {  
    x // tem valor de 8  
}
```

// enum - raw value

```
enum Ordem: Int {  
    case primeiro = 1  
    case segundo = 2  
    case terceiro  
}
```

// enum - raw value

Ordem.primeiro.rawValue // tem valor de 1 (Int)

Ordem.terceiro.rawValue // tem valor de 3 (Int)

```
// enum - raw value
```

```
switch posicao {
```

```
case .primeiro:
```

```
    mensagem = "número \ (posicao.rawValue)"
```

```
default:
```

```
    break
```

```
}
```

```
// enum - raw value
```

```
switch posicao.rawValue {
```

```
case 1:
```

```
    mensagem = "número 1"
```

```
default:
```

```
    break
```

```
}
```



```
// enum - raw value
```

```
switch posicao {
```

```
case _ where posicao.rawValue == 1:  
    mensagem = "número 1"
```

```
default:  
    break  
}
```

// Exercício

// Exercício 06

Pedra, Papel e Tesoura

1. Defina uma **enumeração** para representar os três possíveis **movimentos** do jogador (.pedra, .papel, .tesoura)
2. Defina uma **enumeração** para representar os possíveis **resultados** do jogo (.vitoria, .derrota, .empate) com **rawValues** do tipo **String**, contendo uma mensagem anunciando o resultado.
3. Defina uma **função** que recebe os **movimentos dos dois jogadores**, e retorna o uma **mensagem de resultado da partida** (referente ao primeiro jogador)

```
partida(primeiroJogador: .pedra, segundoJogador: .tesoura) // -> "Você ganhou"
```



// Structs e Classes

// Structs e Classes


```
struct Endereco {  
    var cidade: String  
    var logradouro: String  
    var numero: Int?  
}  
  
class Pessoa {  
    let nome: String  
    var endereco: Endereco?  
    var telefone: String?  
}
```

// Construtor

```
class Pessoa {  
    let nome: String  
    var endereco: Endereco?  
    var telefone: String?  
  
    init(nome: String, endereco: Endereco?, telefone: String?) {  
        self.nome = nome  
        self.endereco = endereco  
        self.telefone = telefone  
    }  
}
```


// Construtor

```
class Pessoa {  
    let nome: String  
    var endereco: Endereco?  
    var telefone: String?  
  
    init(nome: String, endereco: Endereco?, telefone: String?) {  
        self.nome = nome  
        self.endereco = endereco  
        self.telefone = telefone  
    }  
}
```




// Construtor

```
class Pessoa {  
    let nome: String  
    var endereco: Endereco?  
    var telefone: String?  
  
    init(nome: String, endereco: Endereco?, telefone: String?) {  
        self.nome = nome  
        self.endereco = endereco  
        self.telefone = telefone  
    }  
}
```



// Construtor

```
class Pessoa {  
    let nome: String  
    var endereco: Endereco?  
    var telefone: String?  
  
    init(nome: String, endereco: Endereco?, telefone: String?) {  
        self.nome = nome  
        self.endereco = endereco  
        self.telefone = telefone  
    }  
}
```



// Inicializando

```
var apollo235 = Endereco(cidade: "Recife",  
                           logradouro: "Rua do Apolo",  
                           numero: 235)
```

```
let joao = Pessoa(nome: "João",  
                  endereco: apollo235,  
                  telefone: nil)
```

// acessando propriedades

joao.nome // tem valor de "João"

joao.telefone // não tem valor (nil)

joao.endereco?.logradouro // "Rua do Apolo"

apolo235.logradouro // também "Rua do Apolo"

// método de instância

```
class Pessoa {  
    let nome: String  
    var endereco: Endereco?  
    var telefone: String?  
  
    init(nome: String, endereco: Endereco?, telefone: String?) {  
        self.nome = nome  
        self.endereco = endereco  
        self.telefone = telefone  
    }  
    // método de instância  
    func apresentar() -> String {  
        return "Oi, meu nome é \(self.nome)"  
    }  
}
```

// método de instância

joao.apresentar() // retorna "Oi, meu nome é João"

// propriedades computadas

```
struct Point {  
    var x = 0.0  
    var y = 0.0  
}
```

```
struct Size {  
    var width = 0.0  
    var height = 0.0  
}
```



// propriedades computadas

```
struct Rect {  
    var origin = Point()  
    var size = Size()  
    var center: Point {  
        get {  
            let centerX = origin.x + (size.width / 2)  
            let centerY = origin.y + (size.height / 2)  
            return Point(x: centerX, y: centerY)  
        }  
        set(newCenter) {  
            origin.x = newCenter.x - (size.width / 2)  
            origin.y = newCenter.y - (size.height / 2)  
        }  
    }  
}
```



// classes e structs

- **Propriedades** para armazenar valores
- **Métodos** para prover funcionalidade
- **Construtores** para configurar o estado inicial
- **Extensões** para fazerem mais que sua definição padrão
- Conformam com **Protocolos**

// por cópia (*value type*)

```
var copiaApolo = apolo235
```

```
apolo235.cidade = "Olinda"
```

```
apolo235.cidade  
copiaApolo.cidade
```

```
// tem valor de "Olinda"
```

```
// tem valor de "Recife"
```

// por referência (*reference type*)

```
var referenciaJoao = joao
```

```
joao.telefone = "999999999"
```

```
joao.telefone // tem valor de "999999999"
```

```
referenciaJoao.telefone // tem valor de "999999999"
```

// quando usar structs

- **Encapsular** alguns valores de dados relativamente **simples**
- É razoável esperar que os **valores** sejam **copiados**
- Qualquer propriedade da struct são também **value types**
- **Não** é preciso **herdar** propriedades ou comportamento de outro tipo

// exemplos de structs

- O **tamanho** de uma **forma geométrica**:
 - propriedades largura e altura
 - Double
- **Pontos** num sistema de coordenadas **3D**:
 - x, y, z
 - Double

// revisando

```
class Pessoa {  
    let nome: String  
    var endereco: Endereco?  
    var telefone: String?  
  
    init(nome: String, endereco: Endereco?, telefone: String?) {  
        self.nome = nome  
        self.endereco = endereco  
        self.telefone = telefone  
    }  
  
    func apresentar() -> String {  
        return "Oi, meu nome é \(self.nome)"  
    }  
}
```

Propriedades

// revisando

```
class Pessoa {  
    let nome: String  
    var endereco: Endereco?  
    var telefone: String?  
  
    init(nome: String, endereco: Endereco?, telefone: String?) {  
        self.nome = nome  
        self.endereco = endereco  
        self.telefone = telefone  
    }  
  
    func apresentar() -> String {  
        return "Oi, meu nome é \ \(self.nome)"  
    }  
}
```

Construtor

// revisando

```
class Pessoa {  
    let nome: String  
    var endereco: Endereco?  
    var telefone: String?  
  
    init(nome: String, endereco: Endereco?, telefone: String?) {  
        self.nome = nome  
        self.endereco = endereco  
        self.telefone = telefone  
    }  
  
    func apresentar() -> String {  
        return "Oi, meu nome é \(self.nome)"  
    }  
}
```

Método



// herança



// herança

```
class Professor: Pessoa {  
    var titulo = "Prof."  
}  
  
var pedro = Professor(nome: "Pedro",  
                      endereco: apolo235,  
                      telefone: "1234567")  
  
pedro.nome    // tem valor de "Pedro"  
pedro.titulo  // tem valor de "Prof."
```

```
// sobrescrevendo método
```

```
class Professor: Pessoa {  
    var titulo = "Prof."
```

```
//sobrescrevendo método
```

```
    override fun apresentar() -> String {  
        return "Oi, meu nome é \$(self.titulo) \$(self.nome)"  
    }  
}
```

```
pedro.apresentar() // retorna "Oi, meu nome é Prof. Pedro"
```

// adicionando propriedade

```
class Professor: Pessoa {
```

```
    //...
```

```
    // nova propriedade
```

```
    var universidade: String
```

```
    init(nome: String, endereco: Endereco?, telefone: String?, universidade: String) {
```

```
        self.universidade = universidade
```

```
        //chamando construtor da superclasse
```

```
        super.init(nome: nome, endereco: endereco, telefone: telefone)
```

```
    }
```

```
}
```

// adicionando propriedade

```
var pedro = Professor(nome: "Pedro",  
                      endereco: apolo235,  
                      telefone: "1234567",  
                      universidade: "UFPE")
```

pedro.universidade // tem valor de "UFPE"

// Exercício

// Exercício 07

Meios de Locomoção

1. Crie classes para os tipos de **Veículos**: Carro, Ônibus, Navio, Barco, Motocicleta
2. Todos os Veículos tem:
 1. **velocidade**
 2. capacidade de calcular o **tempo** que leva para chegar num destino
3. Todos os Veículos, menos Motocicleta e Navio tem: **número de janelas**
4. Só **Veículos Terrestres** tem: **número de rodas**.
5. Só Ônibus tem **capacidade de passageiros**
6. Só **Veículos Aquáticos** tem **largura**



// Casting de Tipos

// casting de tipos

Forma de **conferir** o tipo de uma instância, ou de **tratar** uma instância como sendo de uma subclasse

// checagem de tipos (*is*)

```
let maria = Pessoa(nome: "Maria",  
                    endereco: apolo235,  
                    telefone: nil)
```

```
maria is Pessoa    // retorna true
```

```
maria is Endereco  // retorna false
```

// downcasting (*as*)

```
var pessoas: [Pessoa] = [pedro, maria]
var mensagem = "título(s): "

for pessoa in pessoas {
    if let professor = pessoa as? Professor {
        mensagem = mensagem + professor.titulo +
        professor.nome
    }
}

mensagem // tem valor de "título(s): Prof."
```

// Protocolos

Swift is a **Protocol-Oriented** Programming Language



<https://developer.apple.com/videos/play/wwdc2015/408/>

// protocolo

```
protocol Mestre {  
    var universidadeMestrado: String? {get set}  
  
    func receberTitulo(universidade: String)  
}
```

```
// protocolo
```

```
class Professor: Pessoa, Mestre {  
    // requisitos do protocolo  
    var universidadeMestrado: String?  
  
    func receberTitulo(universidade: String) {  
        self.universidadeMestrado = universidade  
        self.titulo = "Ms."  
    }  
  
    var titulo = "Prof."  
    // ...  
}
```

// protocolo

pedro.receberTitulo(universidade: "MIT")

pedro.titulo // tem valor de "Ms."

pedro.universidadeMestrado // tem valor de "MIT"

```
// protocolo como tipo
```

```
let mestre: Mestre = pedro
```


// Exercício

// Exercício 08

Formas Geométricas

1. Crie um protocolo **Forma**, que tem os métodos:
 1. `area()` -> `Float`
 2. `perimetro()` -> `Float`
2. Crie as classes: **Circulo**, **Quadrado**, **Retângulo** que implementam o protocolo **Forma** de acordo com suas características.

DÚVIDAS

