

MovieUP

(iOS)

Apresentação

200

MARCELLO

Marcelo Fabri

- [@marcelofabri_](#)
- Na Movile desde 2012
 - Unicamp
 - COTUCA

Objetivos

- Ensinar o básico de iOS
- Dar o pontapé inicial
- Mostrar um pouco como trabalhamos

Dinâmica

- Apresentações teóricas (não muito longas)
- Projeto prático durante o curso

Contato

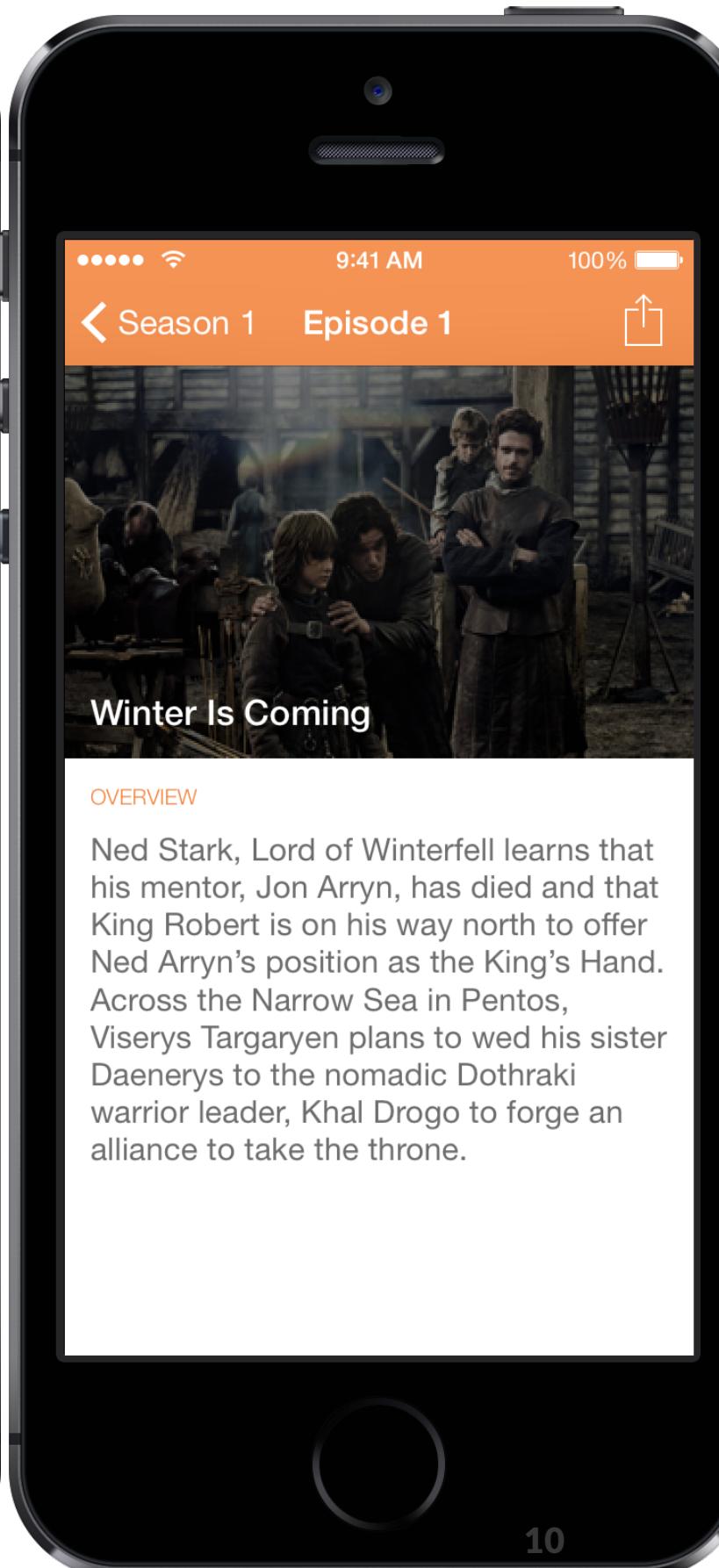
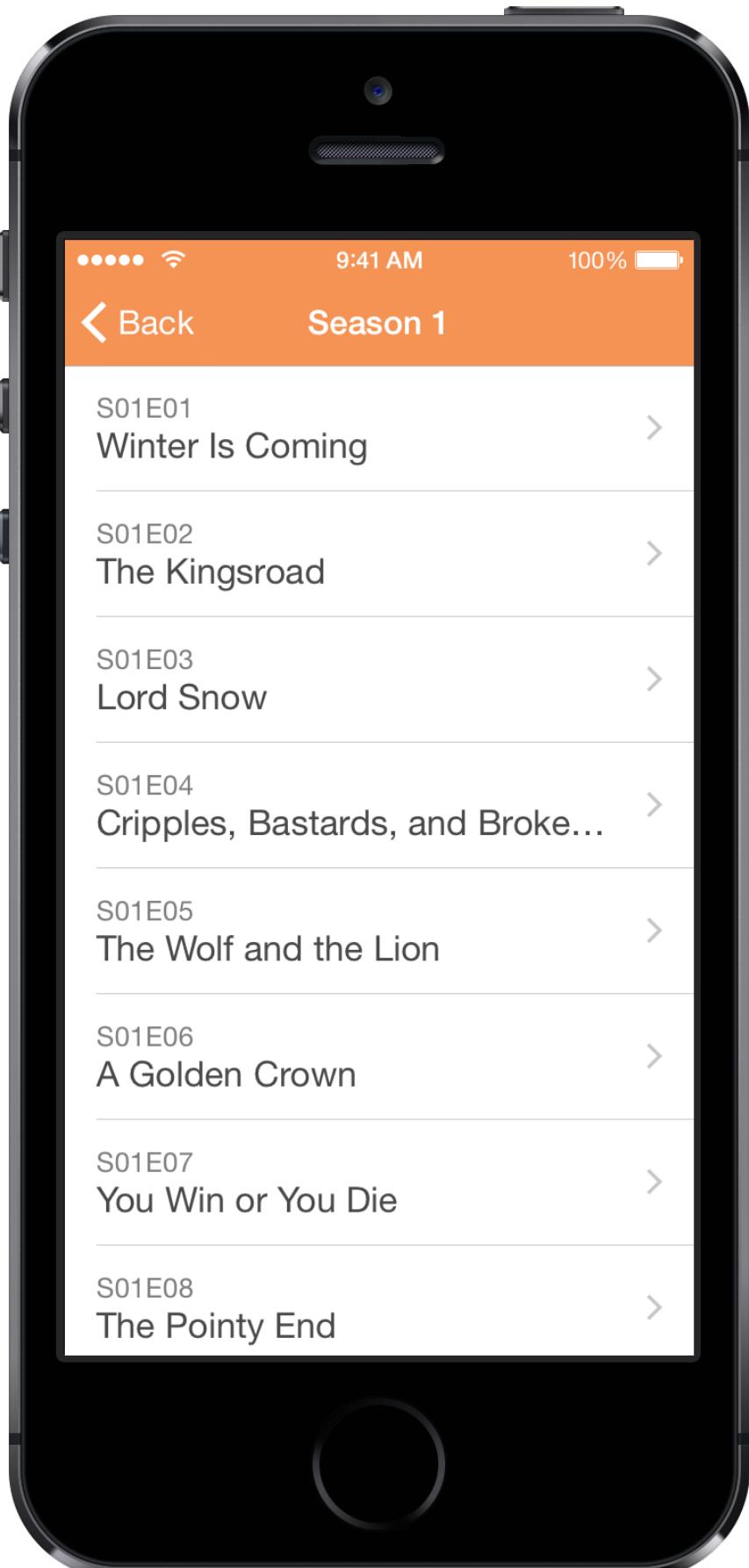
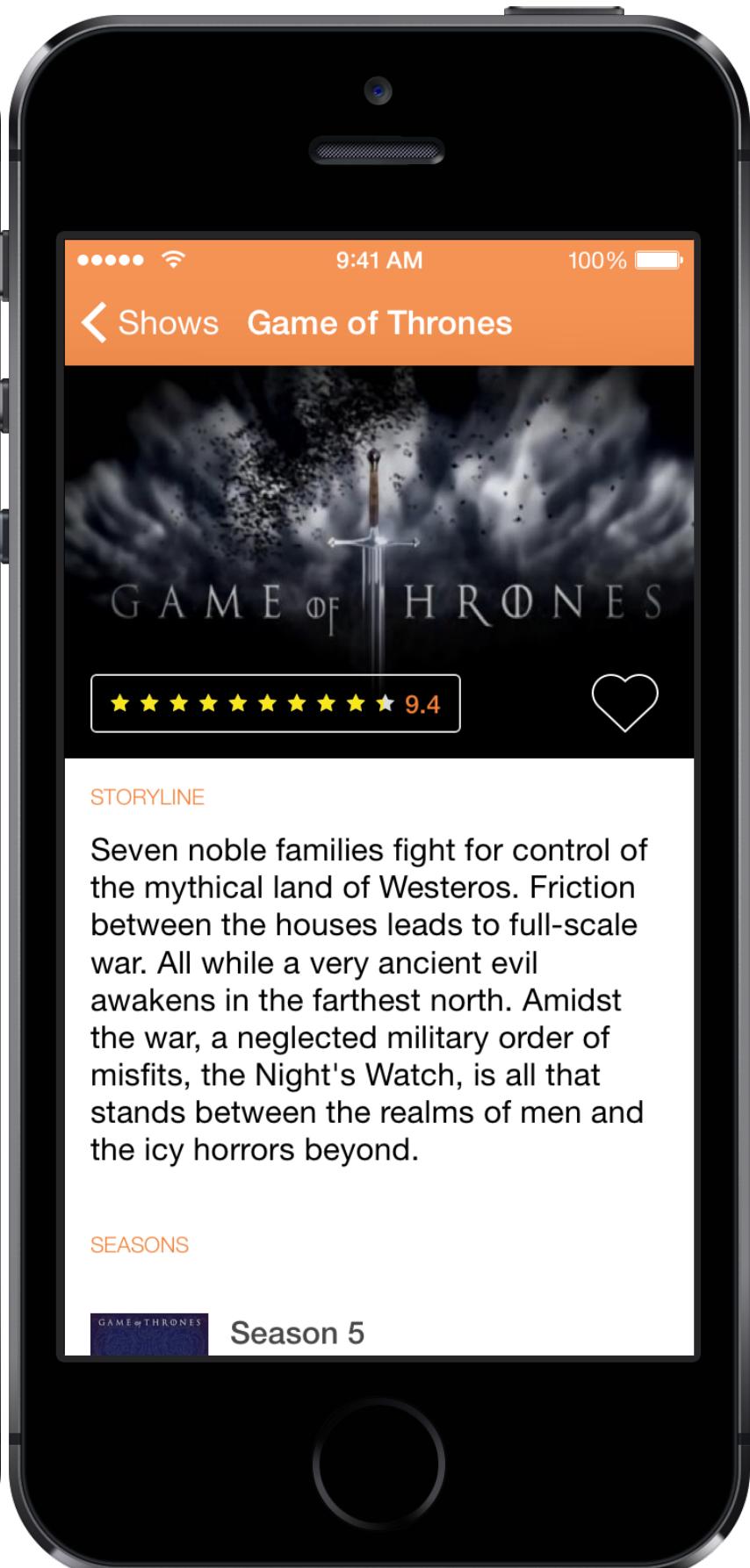
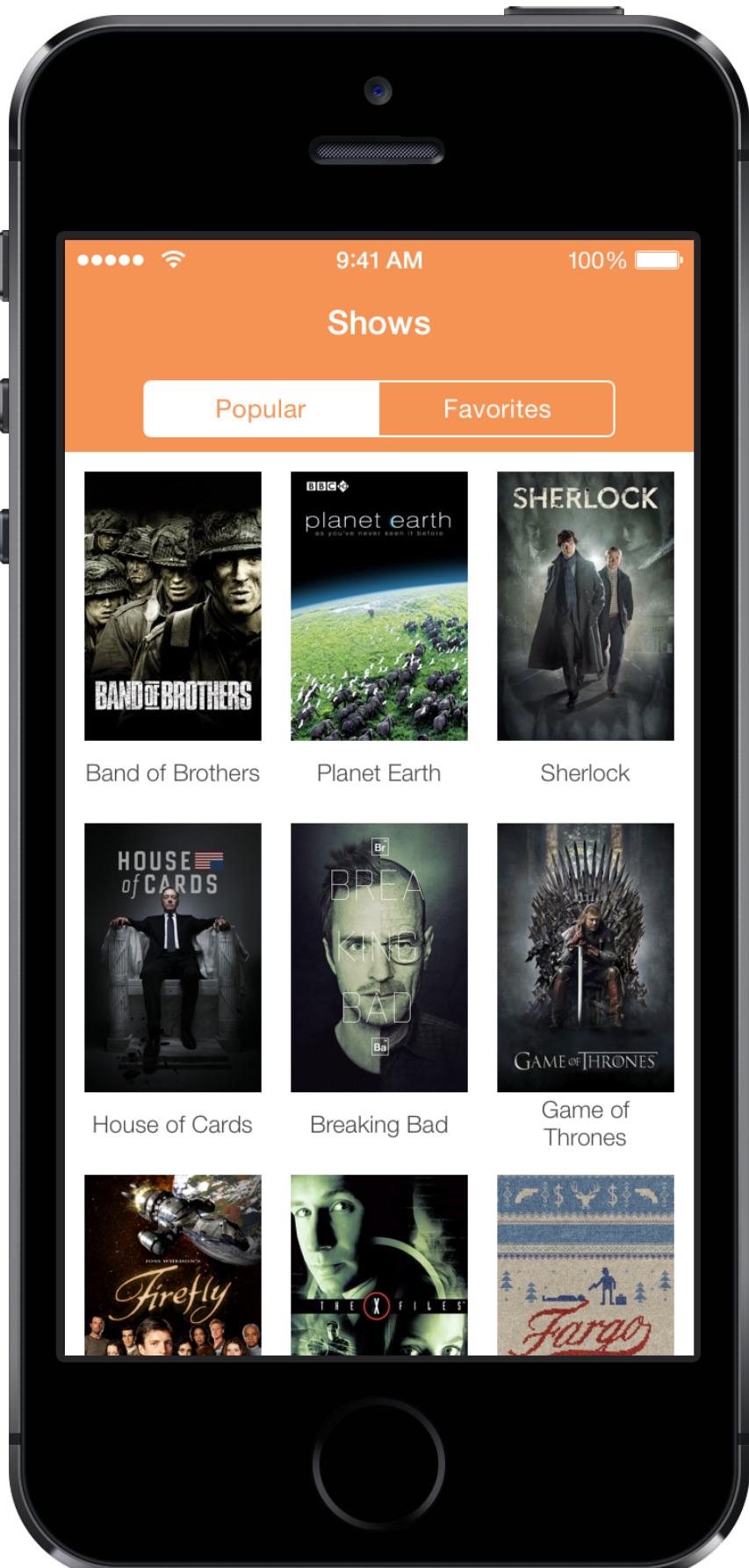
- Só chamar!
- marcelo.fabri@movile.com
- me@marcelofabri.com
- Grupo de email

Material

- <https://github.com/marcelofabri/mobile-up>
- Slides, códigos auxiliares e imagens

Projeto prático

- Listagem de episódios de séries
- Integração com o [trakt.tv](#) para obter os dados

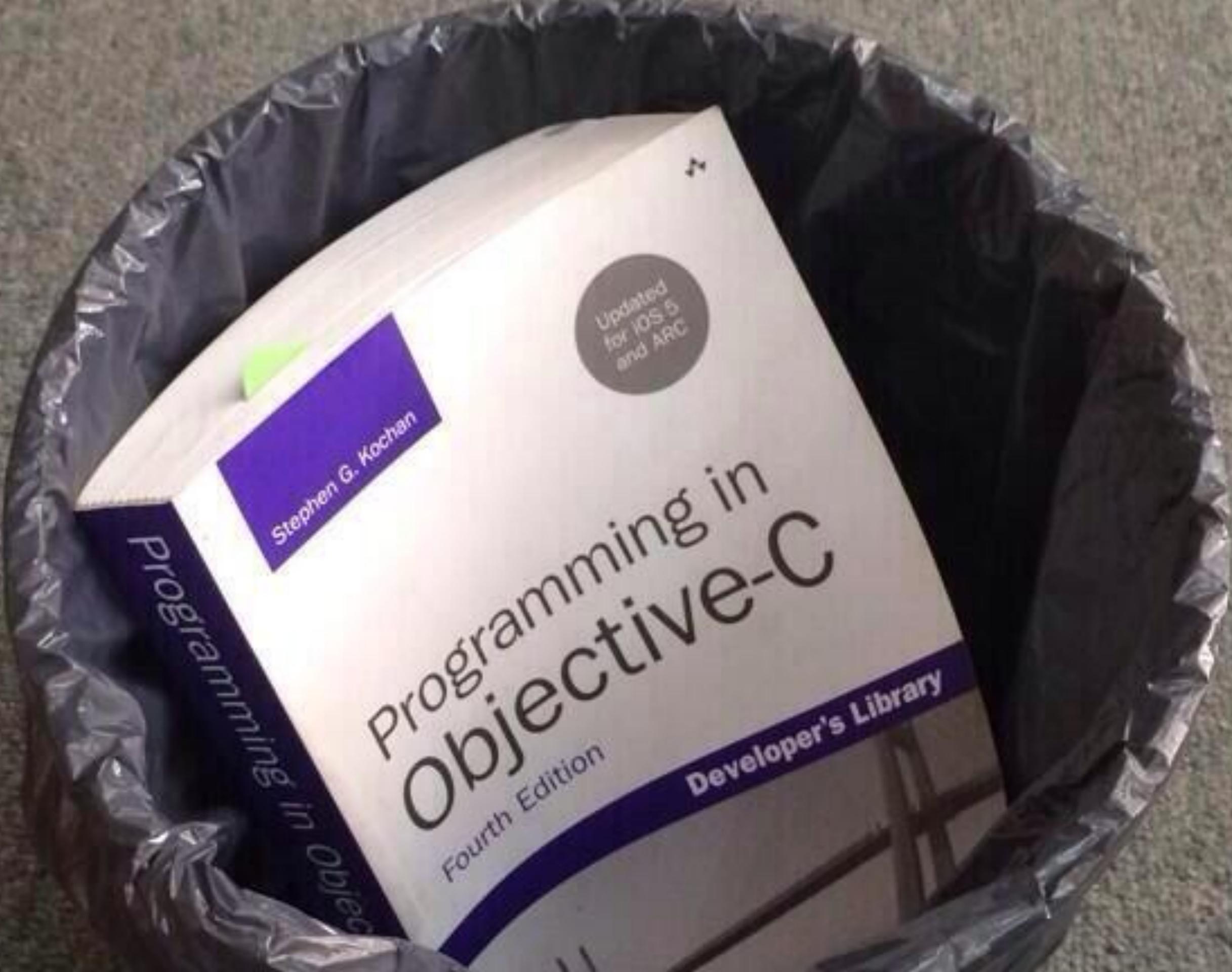


"Bibliografia"

- Apple Developer Portal
- Ray Wenderlich
- NSHipster
- Little Bites of Cocoa
- Natasha The Robot
- iOS Dev Weekly
- objc.io

Swift × Objective-C

Swift x Objective-C





Chris Lattner

@clattner_llvm



Seguir

Looking forward to next month: I'll be the first and only guy with 4 years of swift programming experience :-)

Responder

Retweetar

Curtir

Mais

RETWEETS

1.144

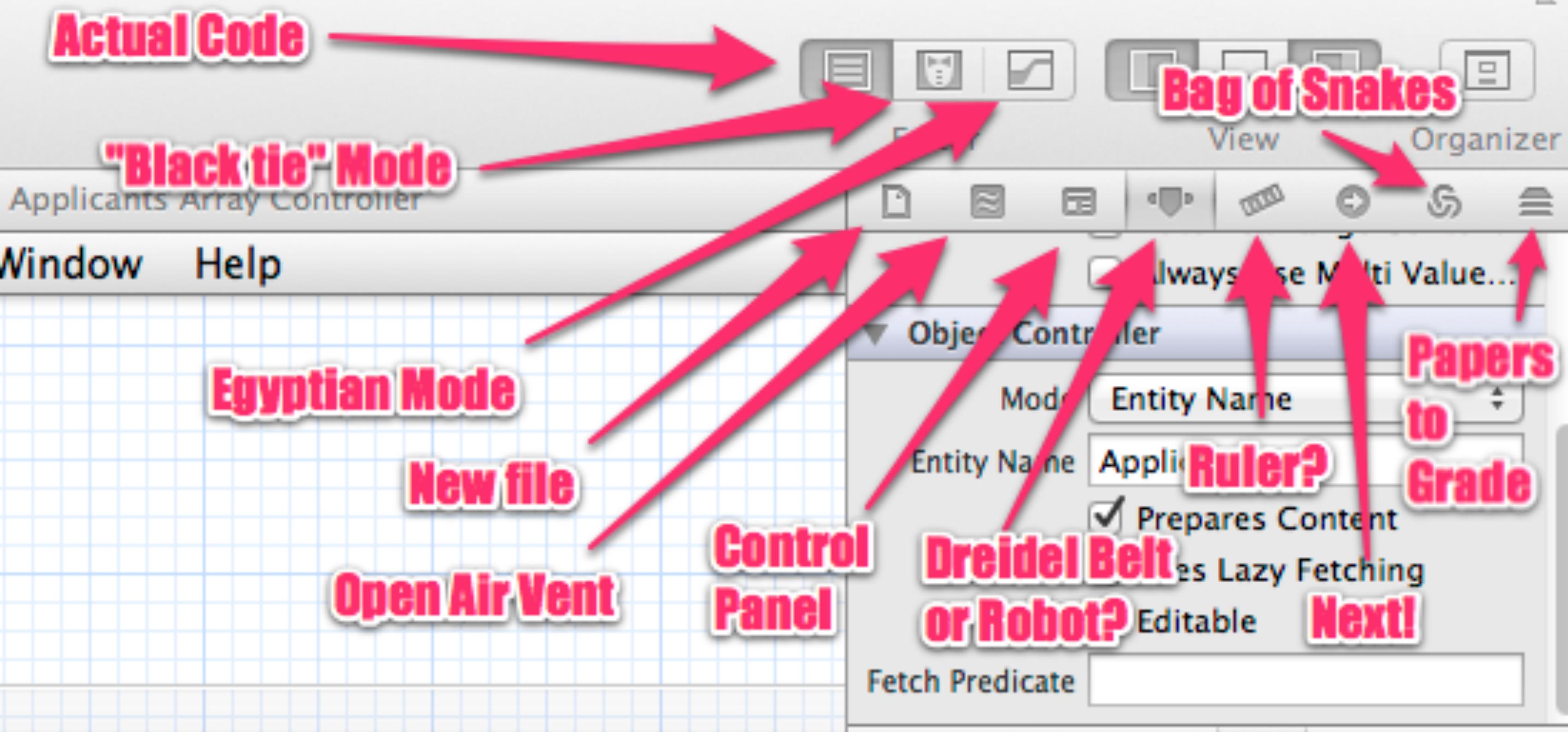
CURTIRAM

896



07:35 - 3 de jun de 2014







Welcome to Xcode

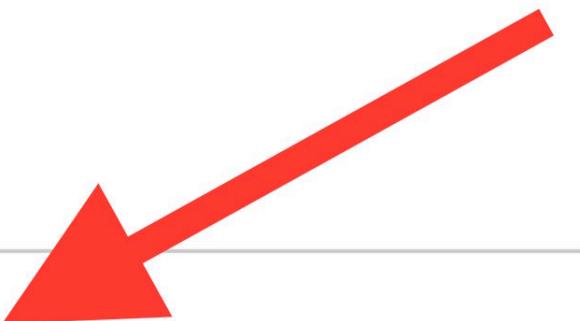
Version 6.0 (6A215l)

No Recent Projects



Get started with a playground

Explore new ideas quickly and easily



Create a new Xcode project

Start building a new iPhone, iPad or Mac application.



Check out an existing project

Start working on something from an SCM repository.



Show this window when Xcode launches

Open another project...

Slow Swift

Functional patterns	Protocols and extensions on structs	Pattern matching
Concise syntax	Closures	Generics
Native collections		Fast iteration
Operator overloading		Optional types
Namespaces	Tuples	Object orientation
Clear mutability syntax		Type inference
Interactive playground	Multiple return types	Read-Eval-Print-Loop (REPL)
		Compile to native code



Moderna

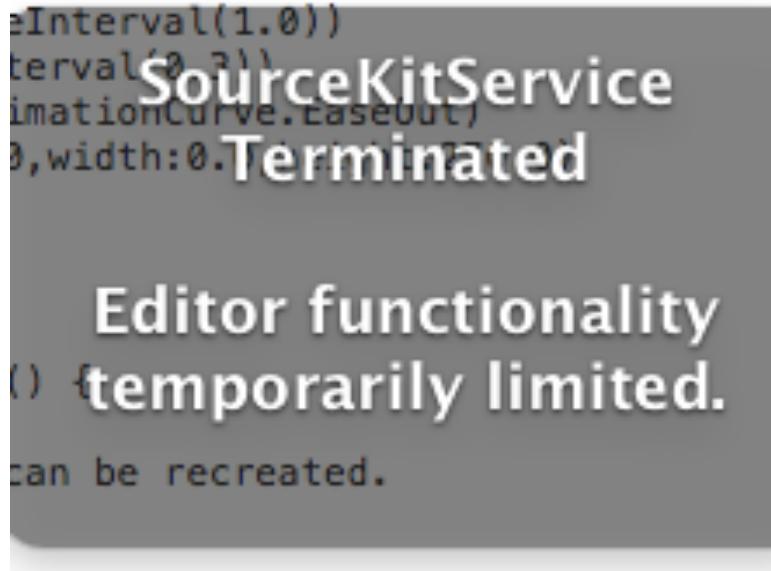
- Closures
- Tuplas
- Generics
- Structs/enums
- Características de programação funcional
- Sem colchetes, ponto-e-vírgula ou headers

Segura

- Variáveis sempre inicializadas antes de uso
- Tipagem forte
- Gerenciamento de memória automático (ARC)
- Erros em tempo de compilação, não de execução
- Inferência de tipos
- Conversão de tipos deve ser sempre explícita

Maaaaaaas...

- Ferramentas instáveis
- Xcode, clang, SourceKit, frameworks, etc
- Preparem-se para ficar um pouco estressados 😬



Sintaxe

Inspirado **fortemente** em "A Swift Tour"¹ e "Language Guide"²

¹ https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/GuidedTour.html

² https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

Hello, world!

```
println("Hello, world!")
```

Variáveis

```
var myVariable = 42  
myVariable = 50  
let myConstant = 42
```

```
let orangesAreOrange = true  
let turnipsAreDelicious = false
```

Observe a inferência de tipos!

Tipos explícitos

```
let implicitInteger = 70
let implicitDouble = 70.0
let explicitDouble: Double = 70
```

Conversão de tipos sempre explícita

```
let label = "The width is " // String
let width = 94 // Int
let widthLabel = label + String(width) // String
```

Vetores

```
var shoppingList = ["catfish", "water", "tulips", "blue paint"]
shoppingList[1] = "bottle of water"
```

Dicionários

```
var occupations = [
  "Malcolm": "Captain",
  "Kaylee": "Mechanic",
]
occupations["Jayne"] = "Public Relations"
```

Controle de fluxo

```
let individualScores = [75, 43, 103, 87, 12]
var teamScore = 0
for score in individualScores {
    if score > 50 {
        teamScore += 3
    } else {
        teamScore += 1
    }
}
println(teamScore)
```

Chaves sempre obrigatórias
Parênteses opcionais

for vs for-in

```
var firstForLoop = 0
for i in 0..<4 {
    firstForLoop += i
}
println(firstForLoop)
```

```
var secondForLoop = 0
for var i = 0; i < 4; ++i {
    secondForLoop += i
}
println(secondForLoop)
```

while vs do-while

```
var n = 2
while n < 100 {
    n = n * 2
}
println(n)
```

```
var m = 2
do {
    m = m * 2
} while m < 100
println(m)
```

Switch

```
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."
}
```

Não tem break (igual em C).
Se quiser continuar, use fallthrough.

Tuplas

```
let http404Error = (404, "Not Found") // (Int, String)
let (statusCode, statusMessage) = http404Error

println("The status code is \(statusCode)")
println("The status message is \(statusMessage)")

println("The status code is \(http404Error.0)")
println("The status message is \(http404Error.1)")

// Só queremos o status code
let (justTheStatusCode, _) = http404Error
```

Optionals

- Se um valor pode ser nulo, ele tem que ser anotado de forma diferente

```
let possibleNumber = "123"  
let convertedNumber = possibleNumber.toInt()
```

- convertedNumber é do tipo Int?, já que possibleNumber pode não ser um inteiro
- Qualquer tipo pode ter sua variante optional

Forced Unwrapping

```
if convertedNumber != nil {  
    println("Value == \(convertedNumber!)")  
}
```

Se convertedNumber fosse nil, um erro (irrecuperável) em runtime seria disparado.

Optional Binding

```
if let actualNumber = possibleNumber.toInt() {  
    println("Value == \(actualNumber)")  
} else {  
    println("Not an integer")  
}  
  
// Multiple optional binding  
if let constantName = someOptional,  
    anotherConstantName = someOtherOptional {  
}
```

Implicitly Unwrapped Optionals

- Todas variáveis de um objeto devem estar instanciadas ao fim de seu construtor
- Mas às vezes só conseguimos fazer isso em algum momento posterior
 - Exemplo: carregamento de telas

```
@IBOutlet weak var label: UILabel!
```

weak vs strong

```
class Person {  
    let name: String  
    init(name: String) { self.name = name }  
    var apartment: Apartment?  
    deinit { println("\(name) is being deinitialized") }  
}
```

```
class Apartment {  
    let number: Int  
    init(number: Int) { self.number = number }  
    weak var tenant: Person?  
    deinit { println("Apartment #\(number) is being deinitialized") }  
}
```

```
var john: Person?
```

```
var number73: Apartment?
```

```
john = Person(name: "John Appleseed")
```

```
number73 = Apartment(number: 73)
```

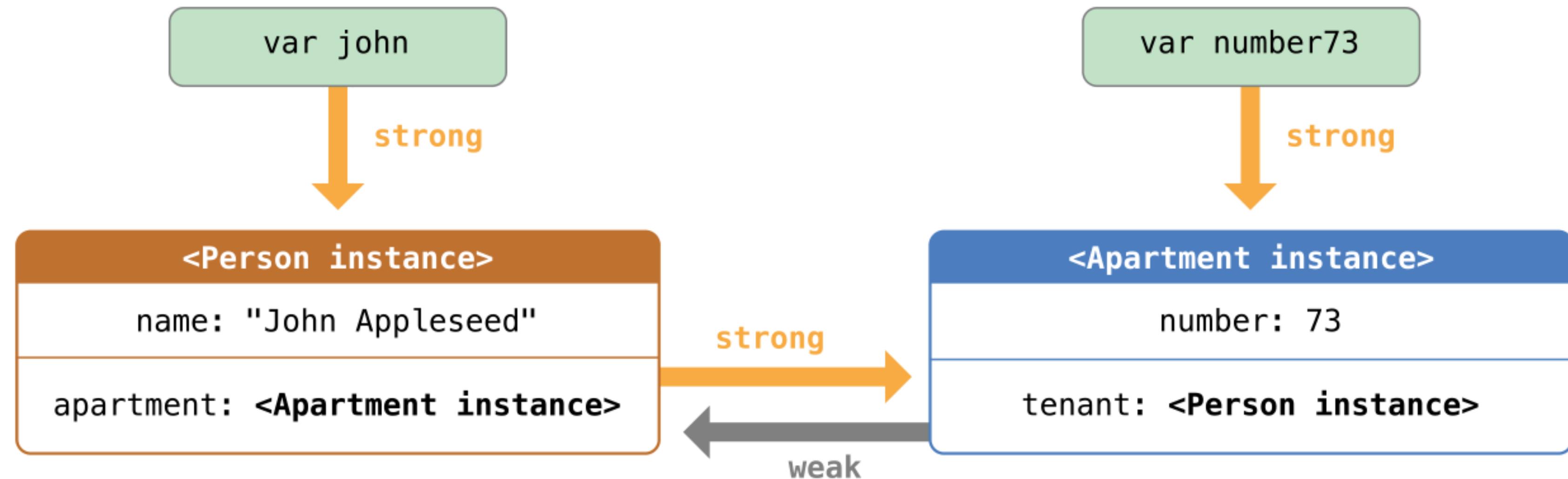
```
john!.apartment = number73
```

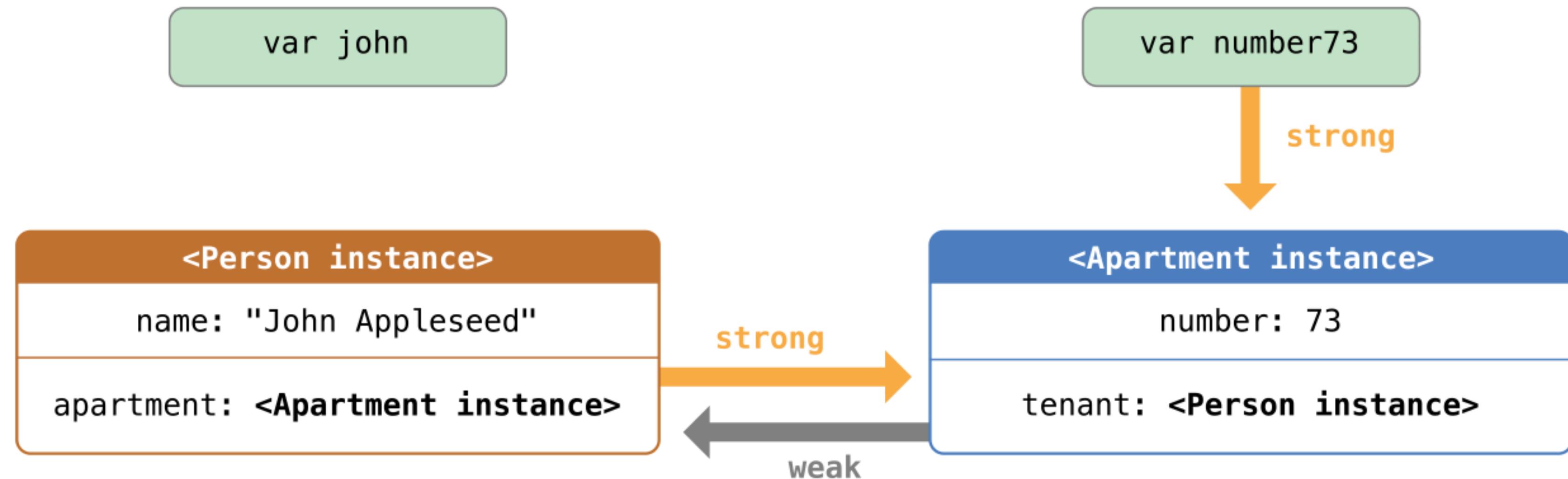
```
number73!.tenant = john
```

```
john = nil
```

```
// prints "John Appleseed is being deinitialized"
```

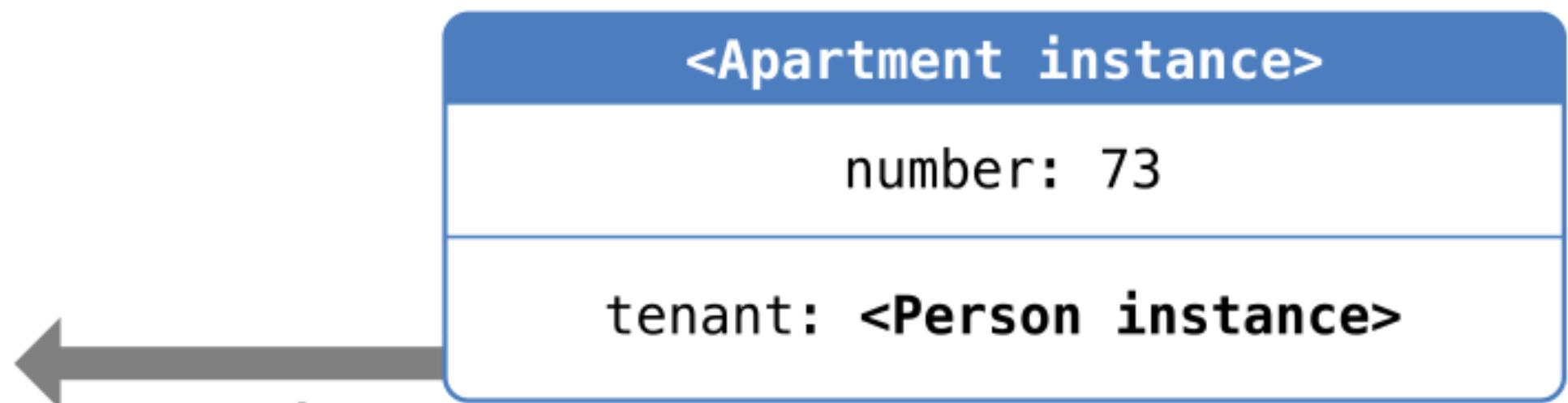
```
println(number73!.apartment) // "nil"
```





var john

var number73



weak

weak vs strong

- strong é o padrão (implícito)
- Veremos mais sobre gerenciamento de memória mais tarde
- Em geral, properties de elementos de tela são weak (a hierarquia de view impede que seja desalocadas imediatamente)

```
@IBOutlet weak var label: UILabel!
```

Optional Chaining

```
if let beginsWithThe =  
    john.residence?.address?.buildingIdentifier()?.hasPrefix("The") {  
    if beginsWithThe {  
        println("John's building identifier begins with \"The\".")  
    } else {  
        println("John's building identifier does not begin with \"The\".")  
    }  
}
```

Só imprimirá algo se residence, address e buildingIdentifier() forem diferente de nil.

protocol

- Definição de interface, não implementação
- Diminuição de acoplamento
- Similar ao interface de Java

```
protocol Named {  
    var name: String { get set }  
  
    func printName() -> Void  
}
```

Lição de casa: ver como tornar um método opcional.

struct vs class

- Guardar informações
- Métodos
- Construtores (init)
- Extensions
- Protocols

class

- Herança
- Destrutores (`deinitializer`)
- Contagem de referência: passagem por referência
- *reference type*

struct

- Passagem por valor (ou seja, sempre são copiadas)
- *value type*

```
class PersonClass: Named {
    var name: String = ""

    init(name: String) {
        self.name = name
    }

    func printName() {
        println("I'm named \(name) and I'm a class.")
    }
}

struct PersonStruct: Named {
    var name: String = ""

    func printName() {
        println("I'm named \(name) and I'm a struct.")
    }
}
```

```
func changeName(var obj: Named) {  
    let oldName = obj.name  
    obj.name = "Someone Else"  
    println("Just changed my name from \(oldName) to \(obj.name).")  
}
```

```
var p1 = PersonClass(name: "Marcelo")  
p1.printName() // I'm named Marcelo and I'm a class.  
changeName(p1) // Just changed my name from Marcelo to Someone Else.  
p1.printName() // I'm named Someone Else and I'm a class.
```

```
var p2 = PersonStruct(name: "John")  
p2.printName() // I'm named John and I'm a struct.  
changeName(p2) // Just changed my name from John to Someone Else.  
p2.printName() // I'm named John and I'm a struct.
```

<http://stackoverflow.com/questions/27241411/cannot-assign-to-property-in-protocol-swift-compiler-error>

enum

```
enum ShowStatus: String {
    case Returning = "returning series"
    case InProduction = "in production"
    case Canceled = "canceled"
    case Ended = "ended"

    func isFinished() -> Bool {
        switch self {
            case .Returning, .InProduction:
                return false
            case .Canceled, .Ended:
                return true
        }
    }
}
```

Usando enum

```
let returning = ShowStatus.Returning
if let returningFromString = ShowStatus(rawValue: "returning series") {
    println("Yey!")
}

println(ShowStatus.Ended.rawValue) // "ended"
```

enum com valores associados

```
enum Router {  
    case PopularShows  
    case Show(String)  
    case Seasons(showId: String)  
    // ...  
}
```

```
let popular = Router.PopularShows  
let got = Router.Show("game-of-thrones")  
let gotSeasons = Router.Seasons(showId: "game-of-thrones")
```

```
extension Router {  
    var path: String {  
        switch self {  
        case .PopularShows:  
            return "shows/popular"  
        case .Show(let id):  
            return "shows/\\(id)"  
        case .Seasons(let showId):  
            return "shows/\\(showId)/seasons"  
        }  
    }  
}
```

```
Router.PopularShows.path // "shows/popular"  
Router.Show("game-of-thrones").path // "shows/game-of-thrones"
```

extension

- Adicionar funcionalidade a um tipo (class, struct ou enum)
- Não precisa ter o código fonte original
- Não consegue sobrescrever funcionalidades existentes
- Pode ser usada para fazer um tipo implementar um protocol

Exemplo de extension

```
extension Int {  
    mutating func square() {  
        self = self * self  
    }  
  
    func half() -> Int {  
        return self / 2  
    }  
}  
  
var someInt = 4  
someInt.square() // someInt is now 16  
  
var anotherInt = someInt.half()  
// anotherInt is 8, but someInt is 16
```

Melhores práticas/convenções

<http://ericasadun.com/2015/05/05/swift-dont-do-that/>

- Don't fight type inference.
- Don't use var when let is appropriate.
- Don't use classes when structs will do.
- Do use Swift native types where possible.
- Do prefer if-let to forced unwrapping whenever possible.
- Don't add code cruft.

Exercício

Criar modelo Show que será usado pelo app

Dica: criar modelos auxiliares para os ids e URLs de imagens

Documentação do Trakt.tv: <http://docs.trakt.apiary.io>