



Hilton Pintor

Desenvolvedor (iOS/tvOS/watchOS)



hiltonpintor@gmail.com

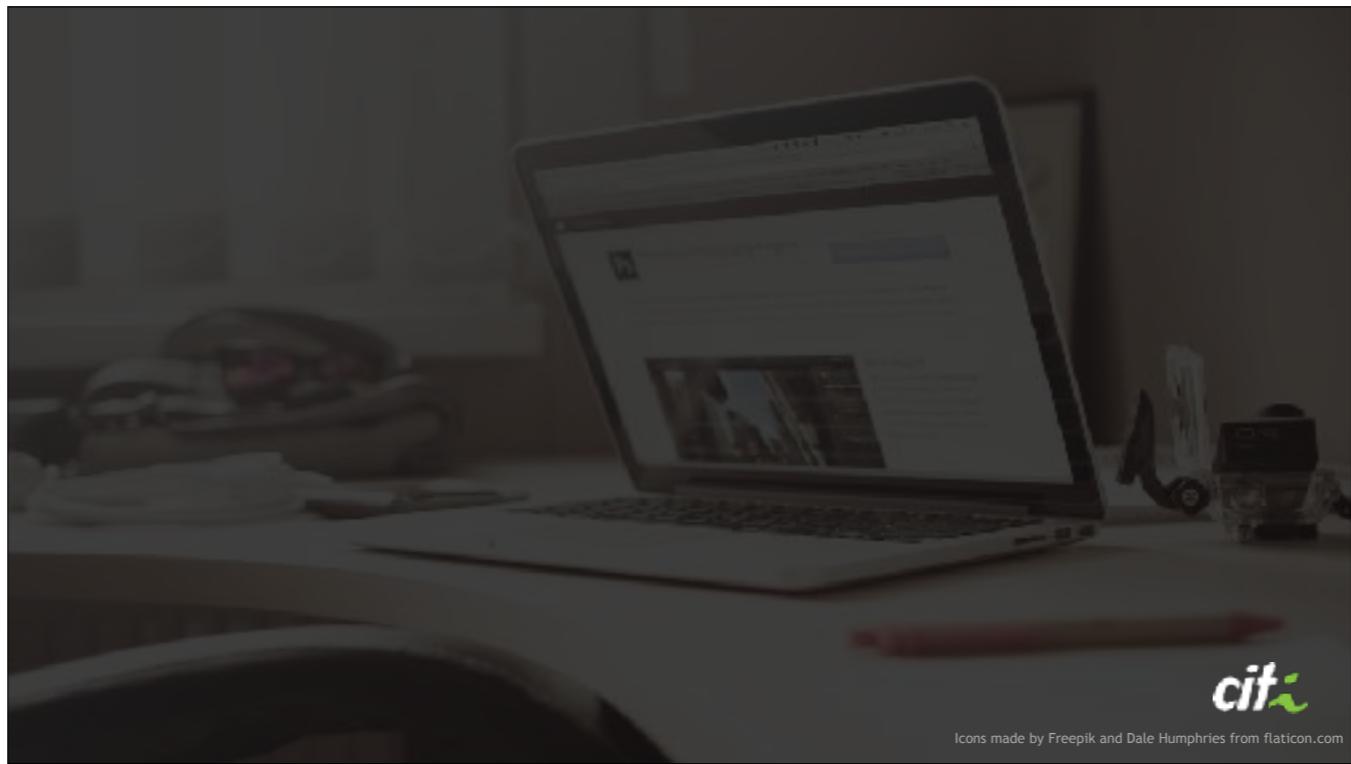
citi

// Semana 01



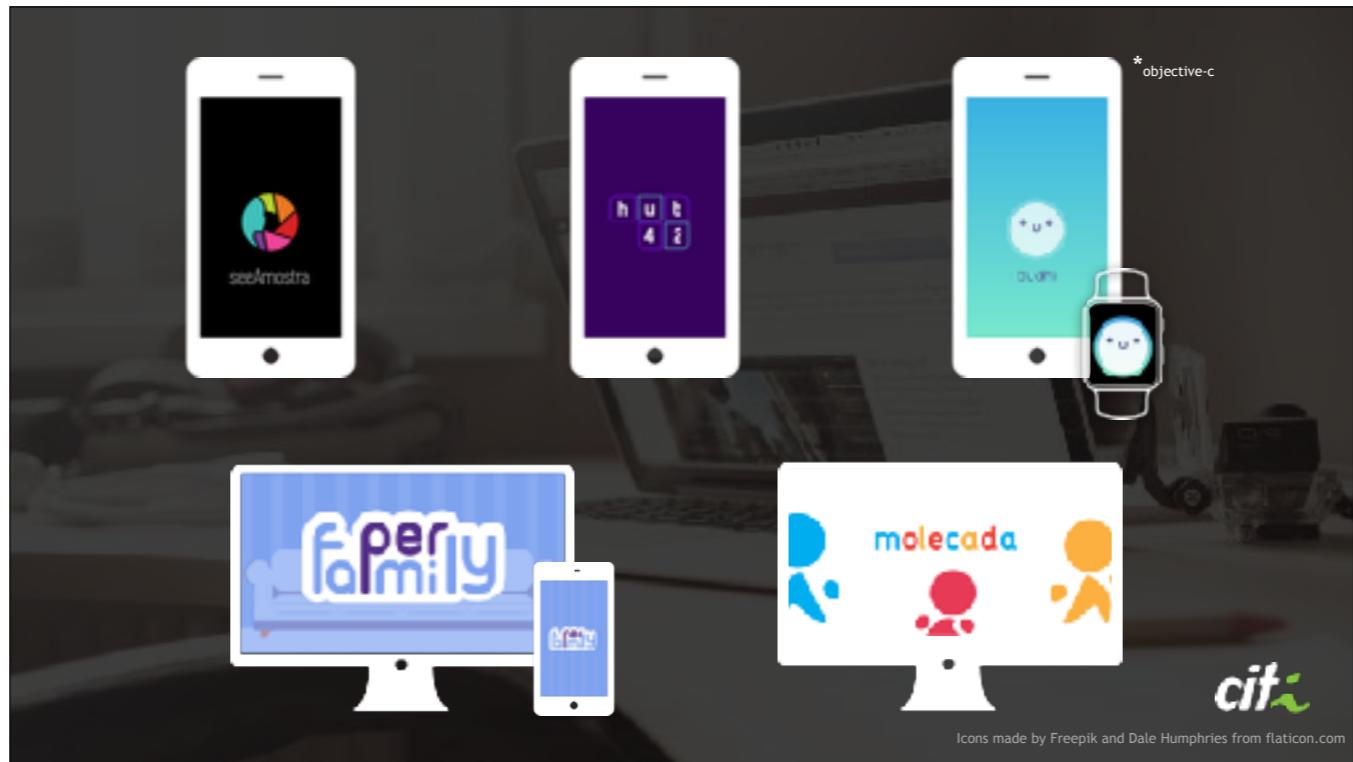
// Aula 01





citi

Icons made by Freepik and Dale Humphries from flaticon.com

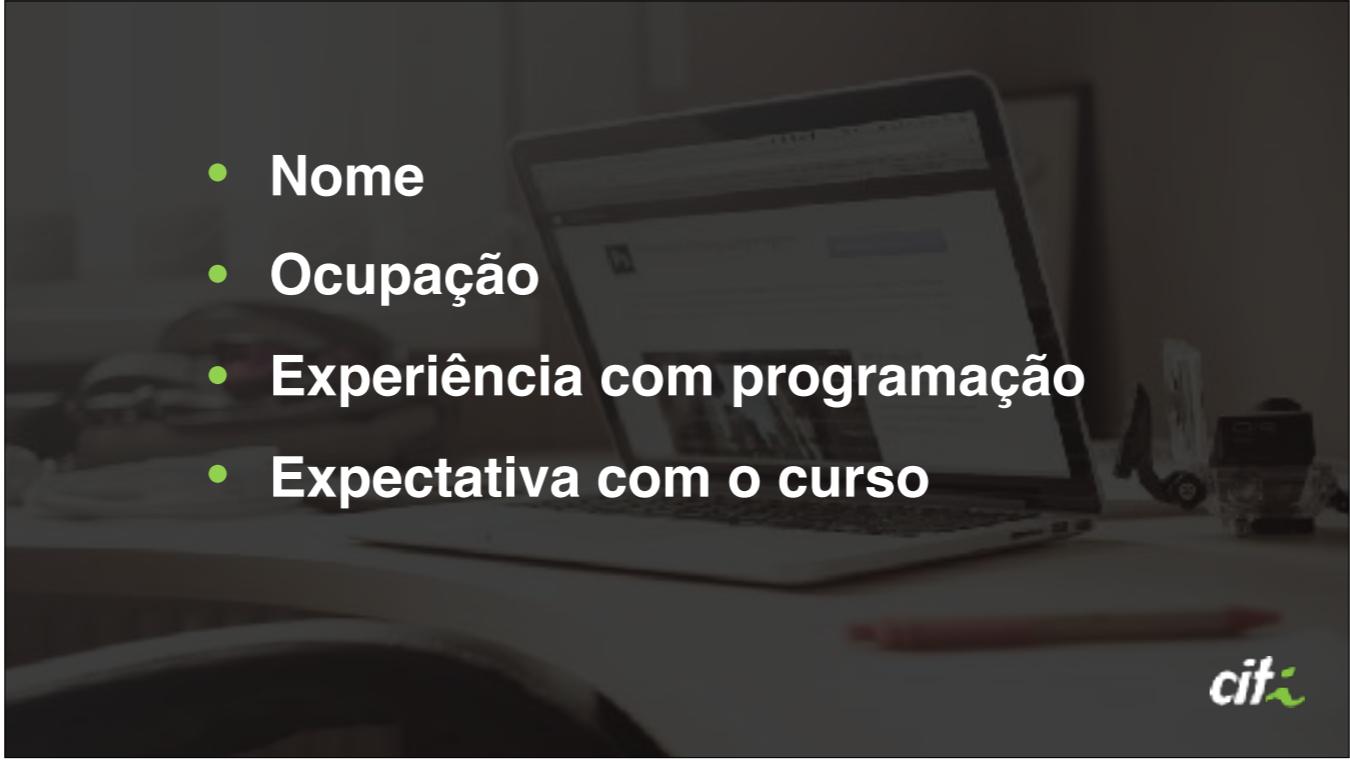


Icons made by Freepik and Dale Humphries from flaticon.com

iOS
com Swift



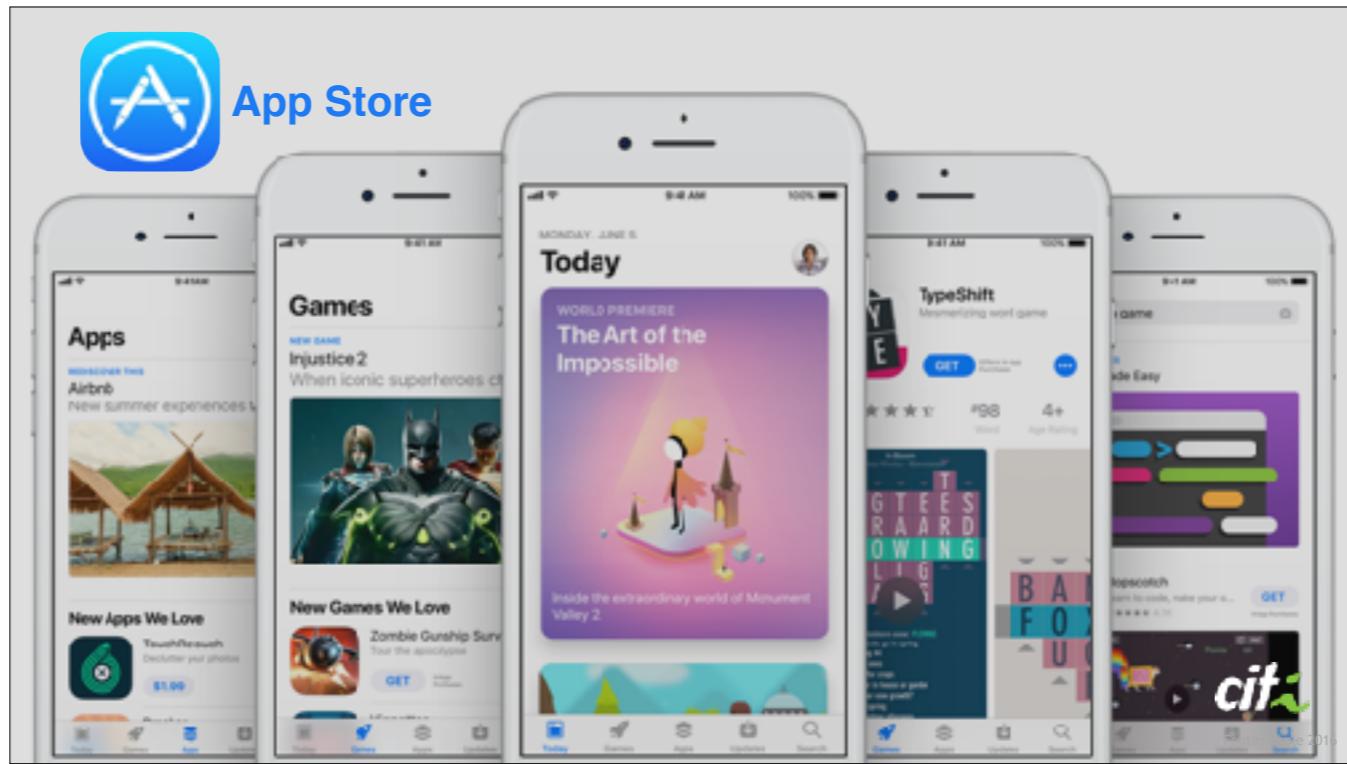
- Nome
- Ocupação
- Experiência com programação
- Expectativa com o curso



cit

iOS
com Swift

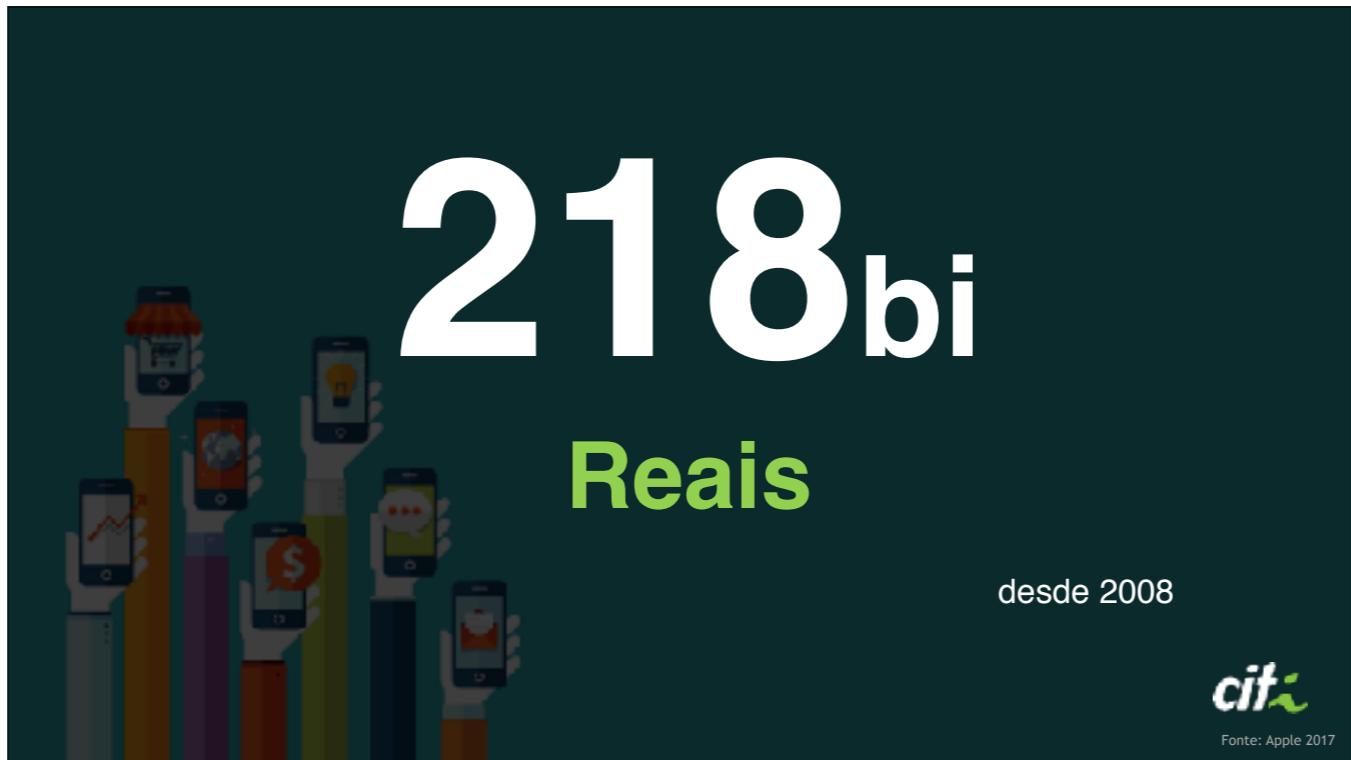




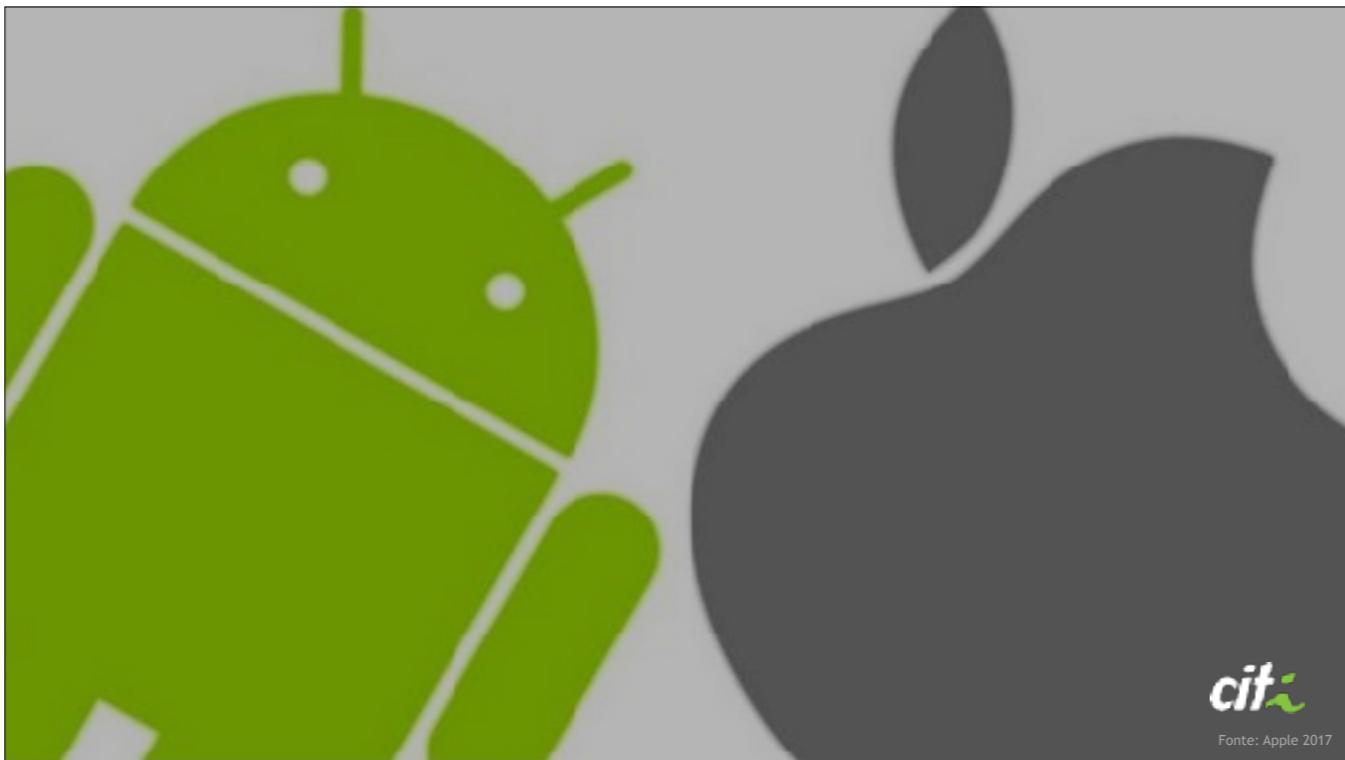
App store é a loja de apps da Apple



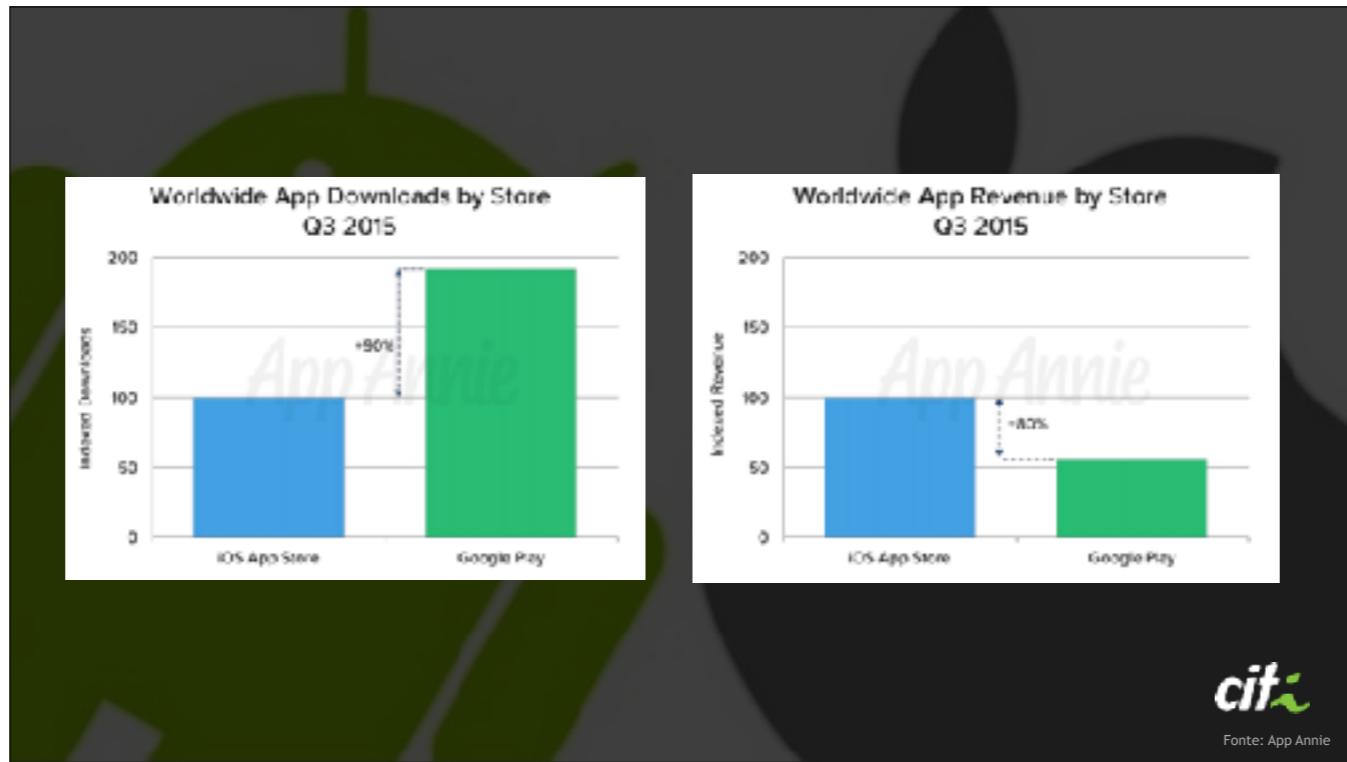
Apple announced on 2016 WWDC that there are over 2 million apps available on the app store



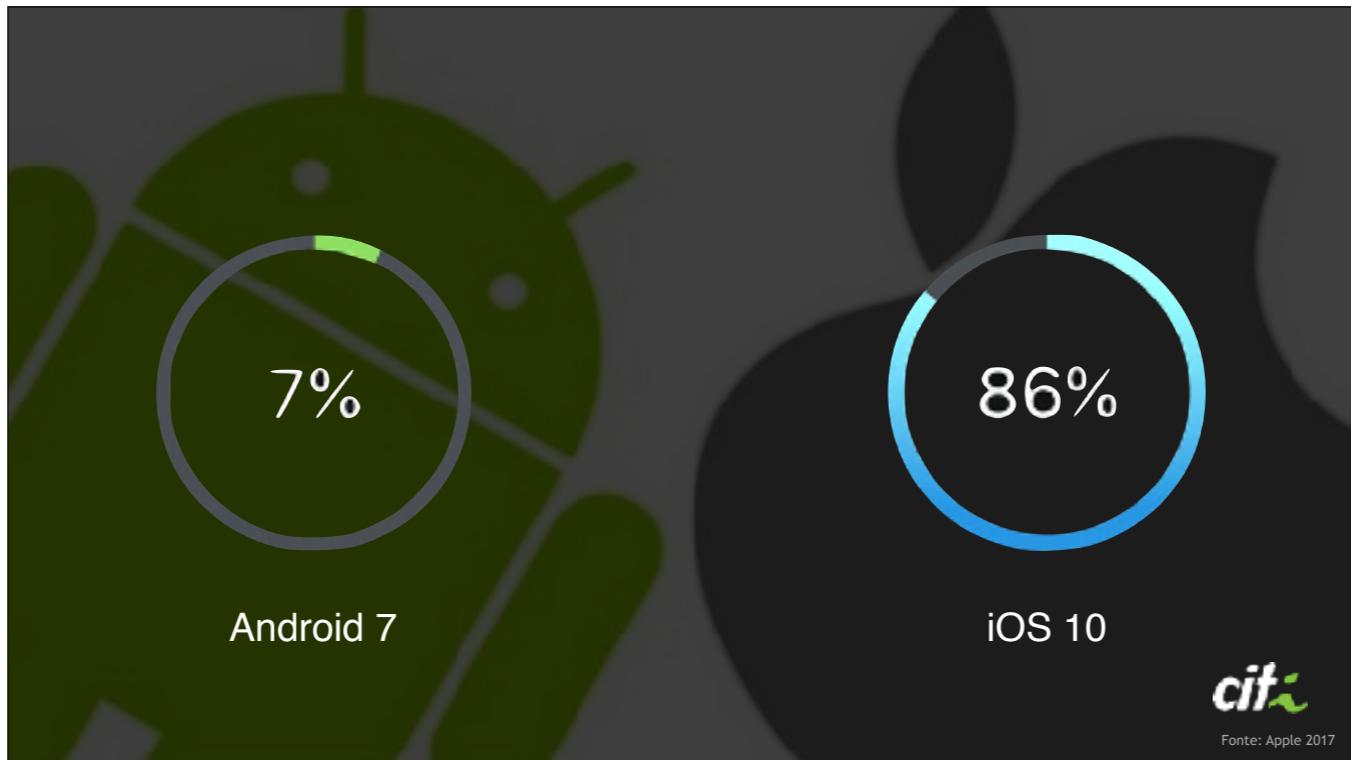
Apple today announced that its global **developer** community has earned over **\$70 billion** since the App Store launched in **2008**.



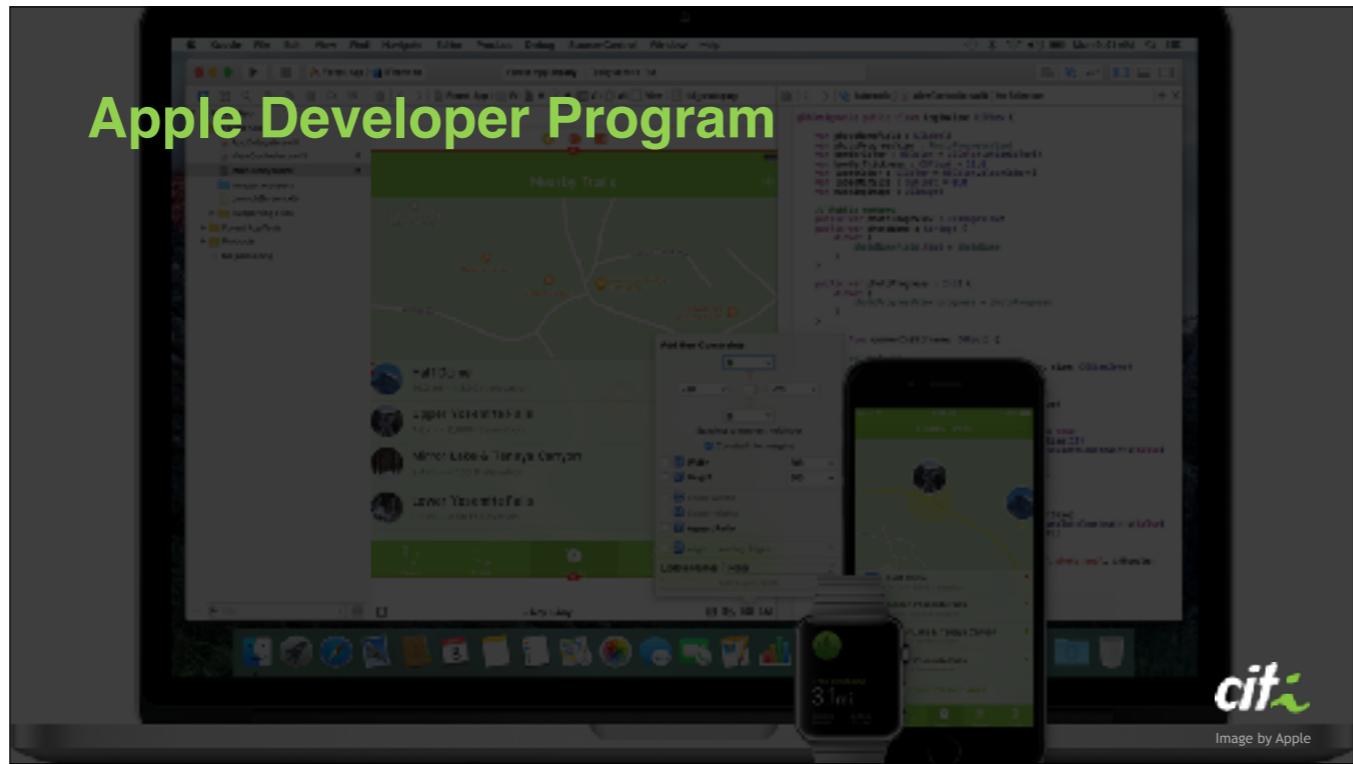
There are many **more devices out there running Android**
the App Store still brings in significantly **more revenue** - to the tune of about **75%** according to a report from App Annie.



There are many **more devices out there running Android**
 the **App Store** still brings in significantly **more revenue** - to the tune of about 75% according to a report from App Annie.



Menor fragmentação no iOS.
Usuários fazem **updates** com mais frequência.



Processo de submissão

Apple Developer Program

- Acesso plataformas Apple



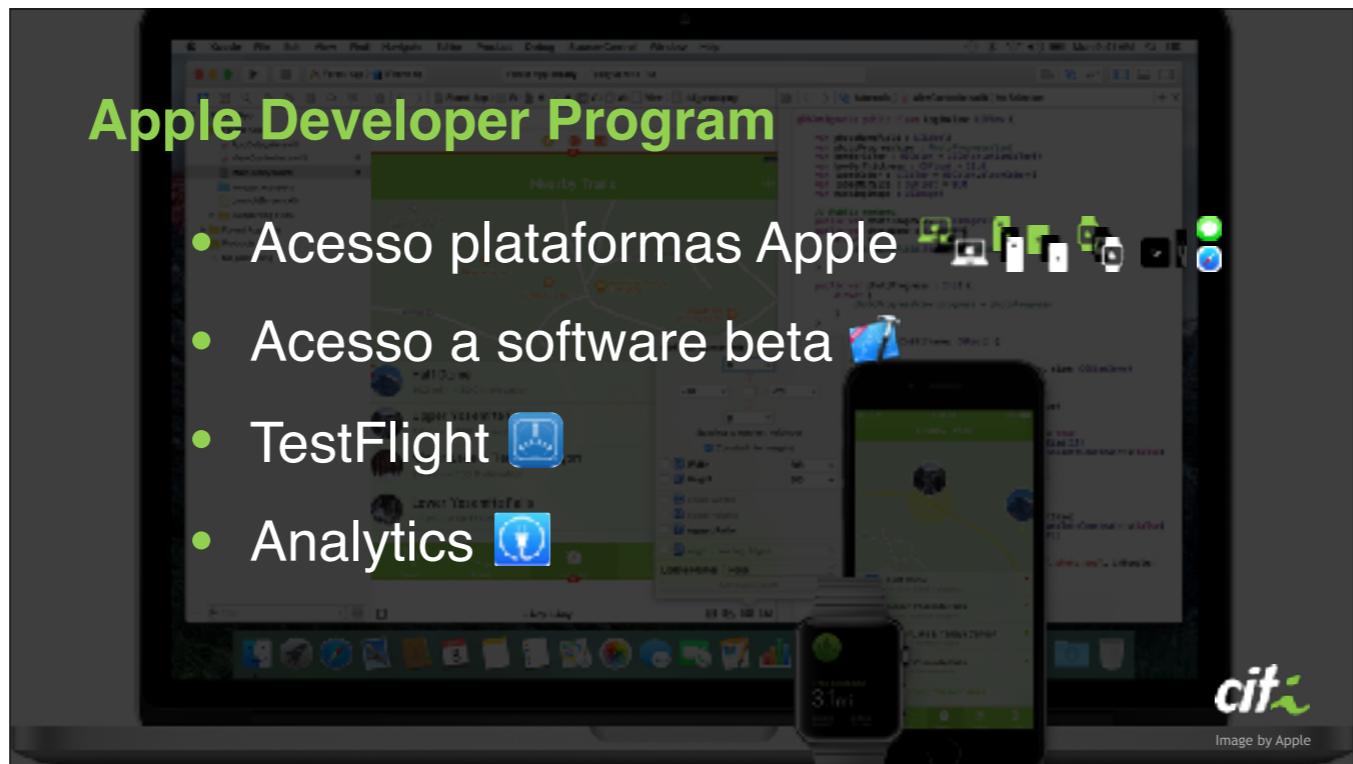
Processo de submissão



Processo de submissão



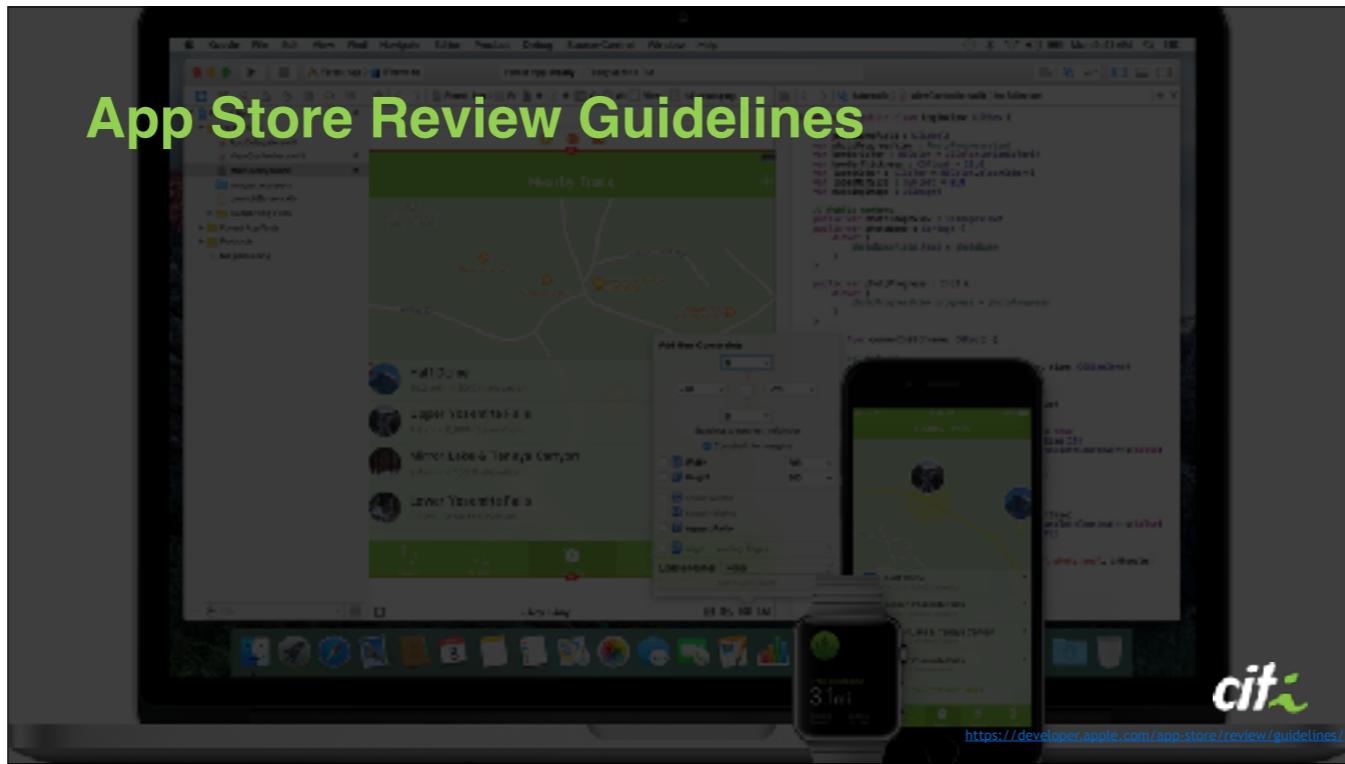
Processo de submissão



Processo de submissão



Processo de submissão



1. Safety

- 1.1 Objectionable Content
- 1.2 User Generated Content
- 1.3 Kids Category
- 1.4 Physical Harm
- 1.5 Developer Information

2. Performance

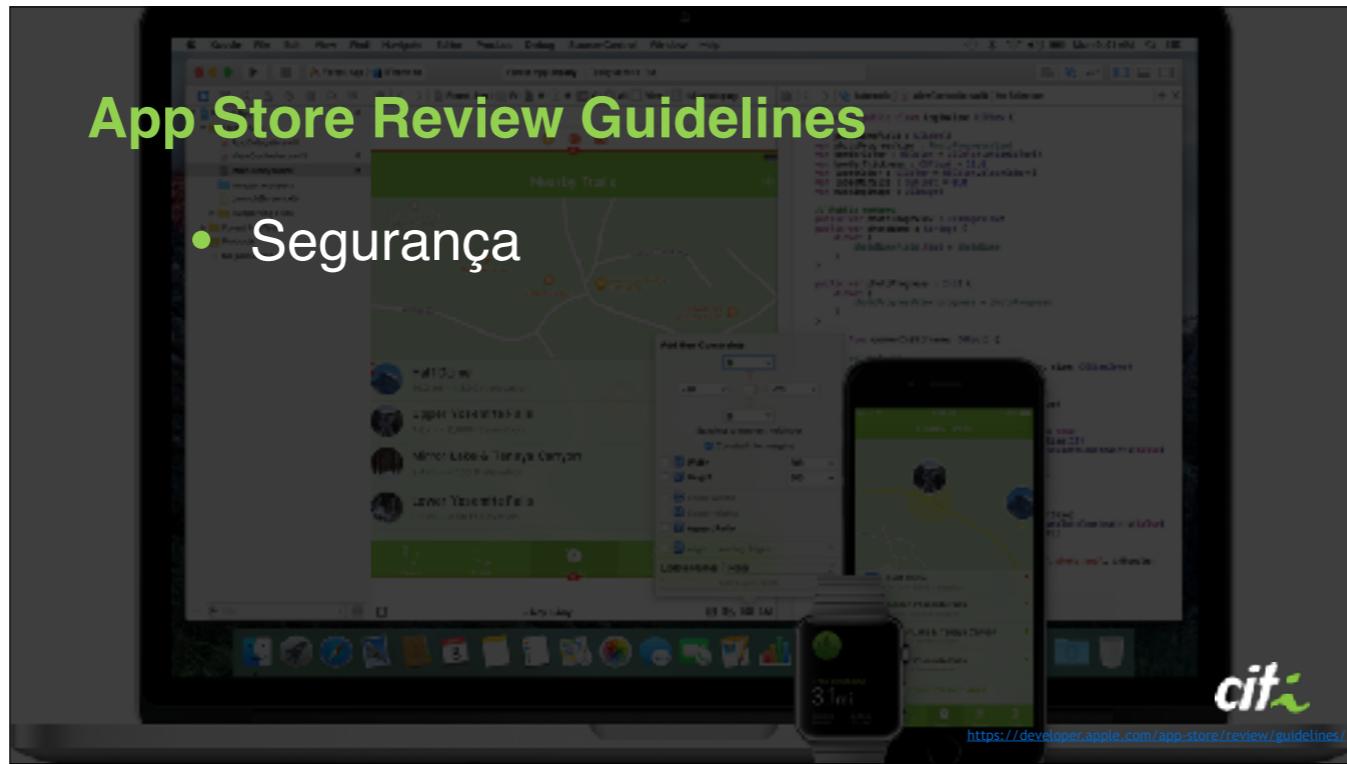
- 2.1 App Completeness
- 2.2 Beta Testing
- 2.3 Accurate Metadata
- 2.4 Hardware Compatibility
- 2.5 Software Requirements

3. Business

- 3.1 Payments
 - 3.1.1 In-App Purchase
 - 3.1.2 Subscriptions
 - 3.1.3 Content-based “Reader” Apps
 - 3.1.4 Content Codes
 - 3.1.5 Physical Goods and Services Outside of the App
 - 3.1.6 Apple Pay
- 3.2 Other Business Model Issues
 - 3.2.1 Acceptable
 - 3.2.2 Unacceptable

4. Design

- 4.1 Copycats



1. Safety

- 1.1 Objectionable Content
- 1.2 User Generated Content
- 1.3 Kids Category
- 1.4 Physical Harm
- 1.5 Developer Information

2. Performance

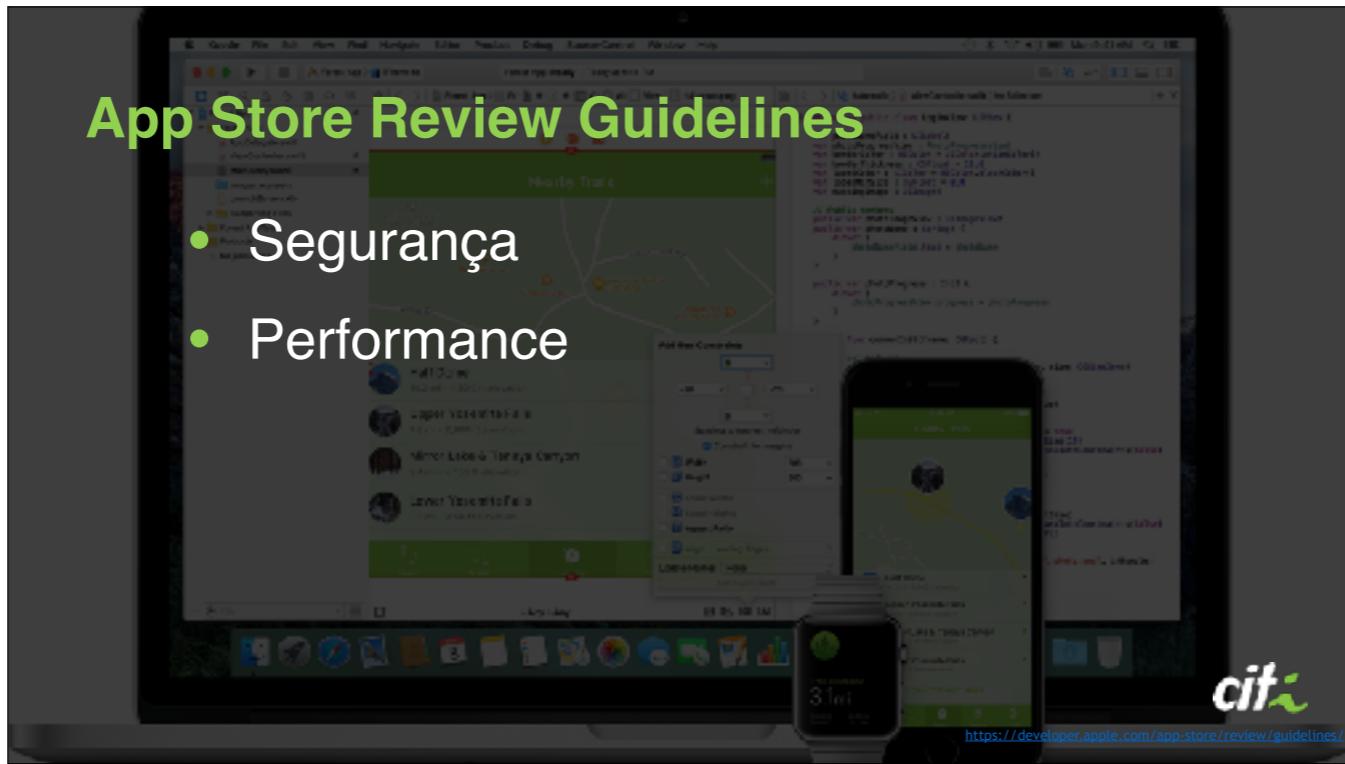
- 2.1 App Completeness
- 2.2 Beta Testing
- 2.3 Accurate Metadata
- 2.4 Hardware Compatibility
- 2.5 Software Requirements

3. Business

- 3.1 Payments
 - 3.1.1 In-App Purchase
 - 3.1.2 Subscriptions
 - 3.1.3 Content-based “Reader” Apps
 - 3.1.4 Content Codes
 - 3.1.5 Physical Goods and Services Outside of the App
 - 3.1.6 Apple Pay
- 3.2 Other Business Model Issues
 - 3.2.1 Acceptable
 - 3.2.2 Unacceptable

4. Design

- 4.1 Copycats



1. Safety

- 1.1 Objectionable Content
- 1.2 User Generated Content
- 1.3 Kids Category
- 1.4 Physical Harm
- 1.5 Developer Information

2. Performance

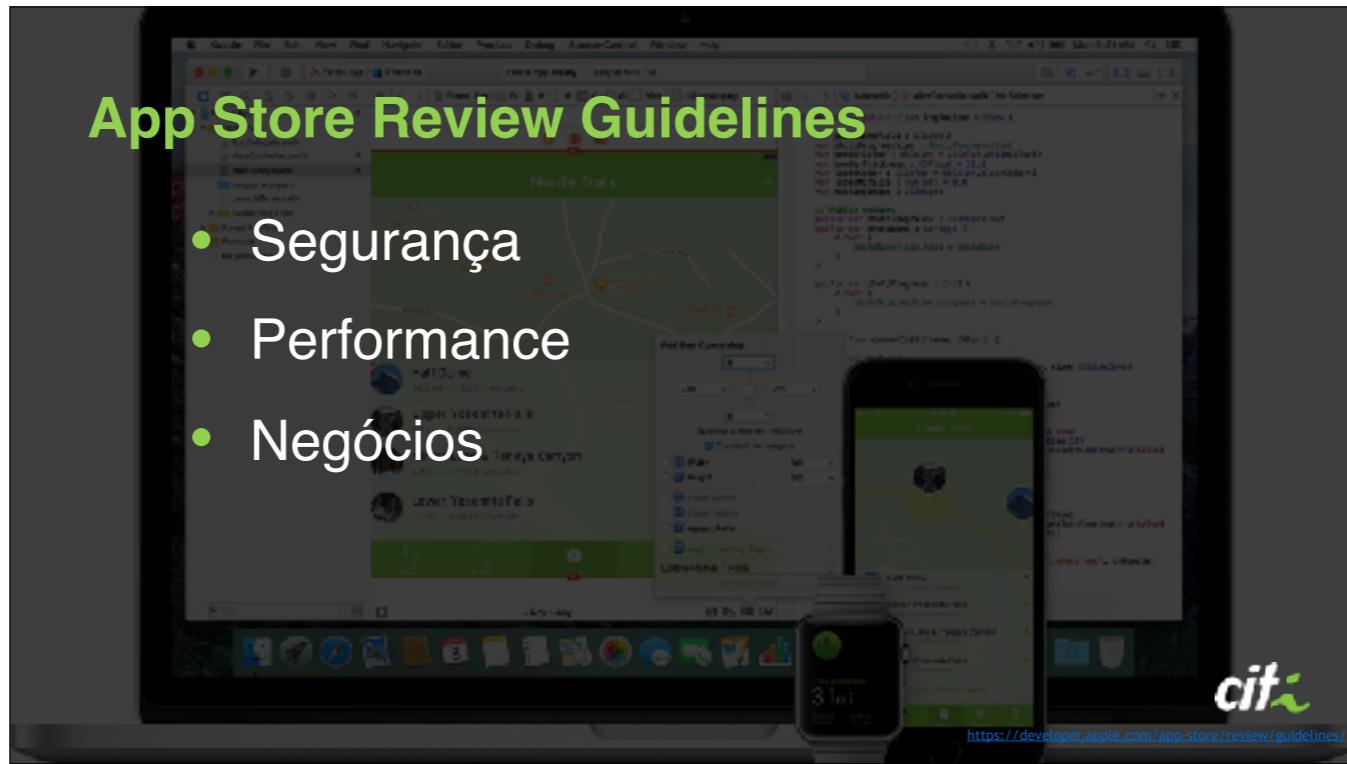
- 2.1 App Completeness
- 2.2 Beta Testing
- 2.3 Accurate Metadata
- 2.4 Hardware Compatibility
- 2.5 Software Requirements

3. Business

- 3.1 Payments
 - 3.1.1 In-App Purchase
 - 3.1.2 Subscriptions
 - 3.1.3 Content-based “Reader” Apps
 - 3.1.4 Content Codes
 - 3.1.5 Physical Goods and Services Outside of the App
 - 3.1.6 Apple Pay
- 3.2 Other Business Model Issues
 - 3.2.1 Acceptable
 - 3.2.2 Unacceptable

4. Design

- 4.1 Copycats



1. Safety

- 1.1 Objectionable Content
- 1.2 User Generated Content
- 1.3 Kids Category
- 1.4 Physical Harm
- 1.5 Developer Information

2. Performance

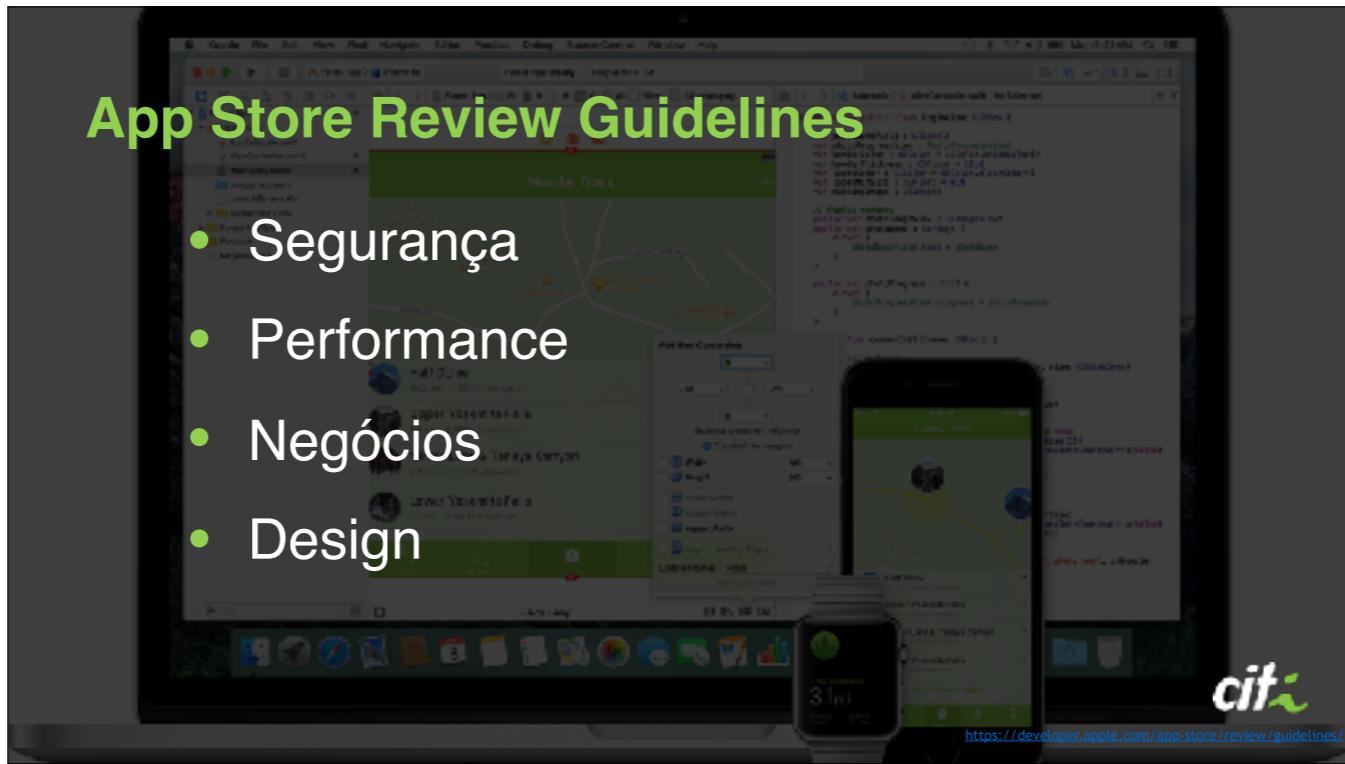
- 2.1 App Completeness
- 2.2 Beta Testing
- 2.3 Accurate Metadata
- 2.4 Hardware Compatibility
- 2.5 Software Requirements

3. Business

- 3.1 Payments
 - 3.1.1 In-App Purchase
 - 3.1.2 Subscriptions
 - 3.1.3 Content-based “Reader” Apps
 - 3.1.4 Content Codes
 - 3.1.5 Physical Goods and Services Outside of the App
 - 3.1.6 Apple Pay
- 3.2 Other Business Model Issues
 - 3.2.1 Acceptable
 - 3.2.2 Unacceptable

4. Design

- 4.1 Copycats



1. Safety

- 1.1 Objectionable Content
- 1.2 User Generated Content
- 1.3 Kids Category
- 1.4 Physical Harm
- 1.5 Developer Information

2. Performance

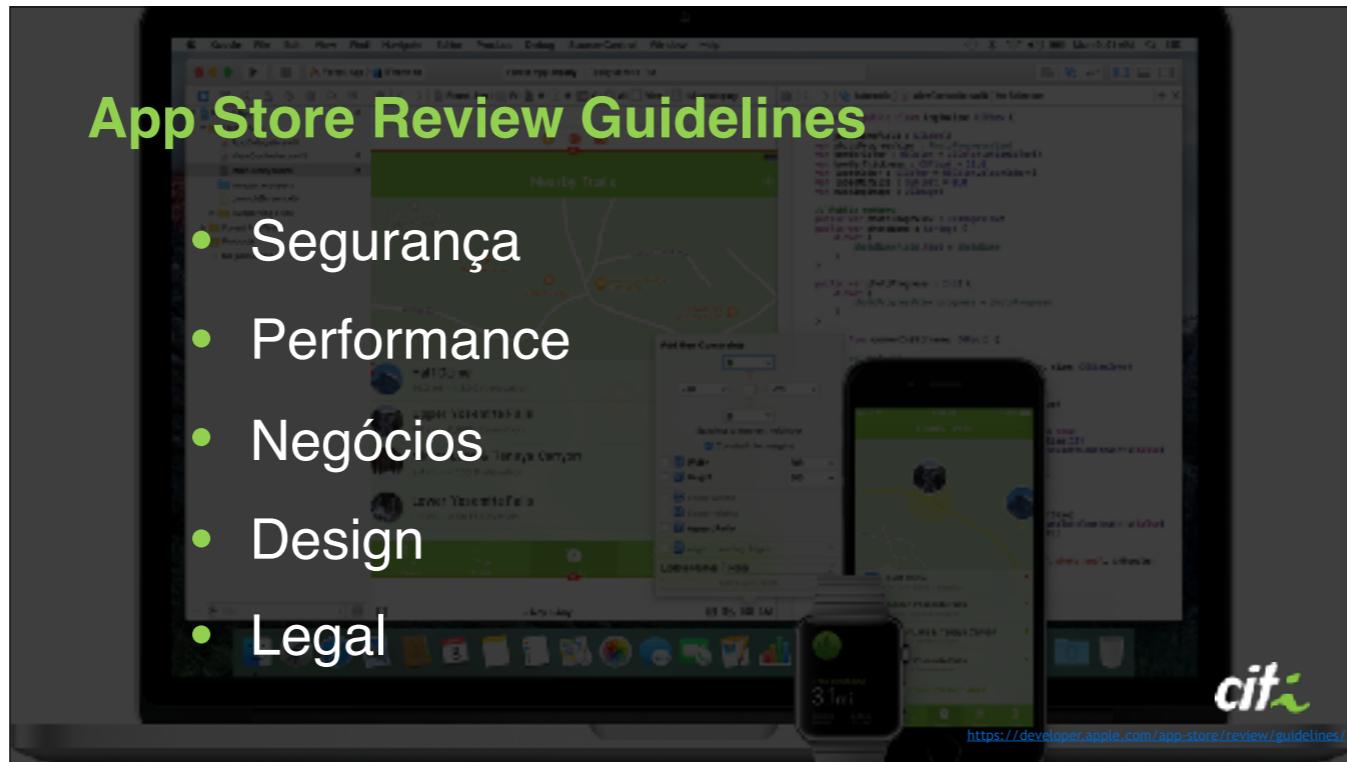
- 2.1 App Completeness
- 2.2 Beta Testing
- 2.3 Accurate Metadata
- 2.4 Hardware Compatibility
- 2.5 Software Requirements

3. Business

- 3.1 Payments
 - 3.1.1 In-App Purchase
 - 3.1.2 Subscriptions
 - 3.1.3 Content-based “Reader” Apps
 - 3.1.4 Content Codes
 - 3.1.5 Physical Goods and Services Outside of the App
 - 3.1.6 Apple Pay
- 3.2 Other Business Model Issues
 - 3.2.1 Acceptable
 - 3.2.2 Unacceptable

4. Design

- 4.1 Copycats



1. Safety

- 1.1 Objectionable Content
- 1.2 User Generated Content
- 1.3 Kids Category
- 1.4 Physical Harm
- 1.5 Developer Information

2. Performance

- 2.1 App Completeness
- 2.2 Beta Testing
- 2.3 Accurate Metadata
- 2.4 Hardware Compatibility
- 2.5 Software Requirements

3. Business

- 3.1 Payments
 - 3.1.1 In-App Purchase
 - 3.1.2 Subscriptions
 - 3.1.3 Content-based “Reader” Apps
 - 3.1.4 Content Codes
 - 3.1.5 Physical Goods and Services Outside of the App
 - 3.1.6 Apple Pay
- 3.2 Other Business Model Issues
 - 3.2.1 Acceptable
 - 3.2.2 Unacceptable

4. Design

- 4.1 Copycats

13 de mai de 2016 às 18:54
De Apple

14. PERSONAL ATTACKS



14.3 Details

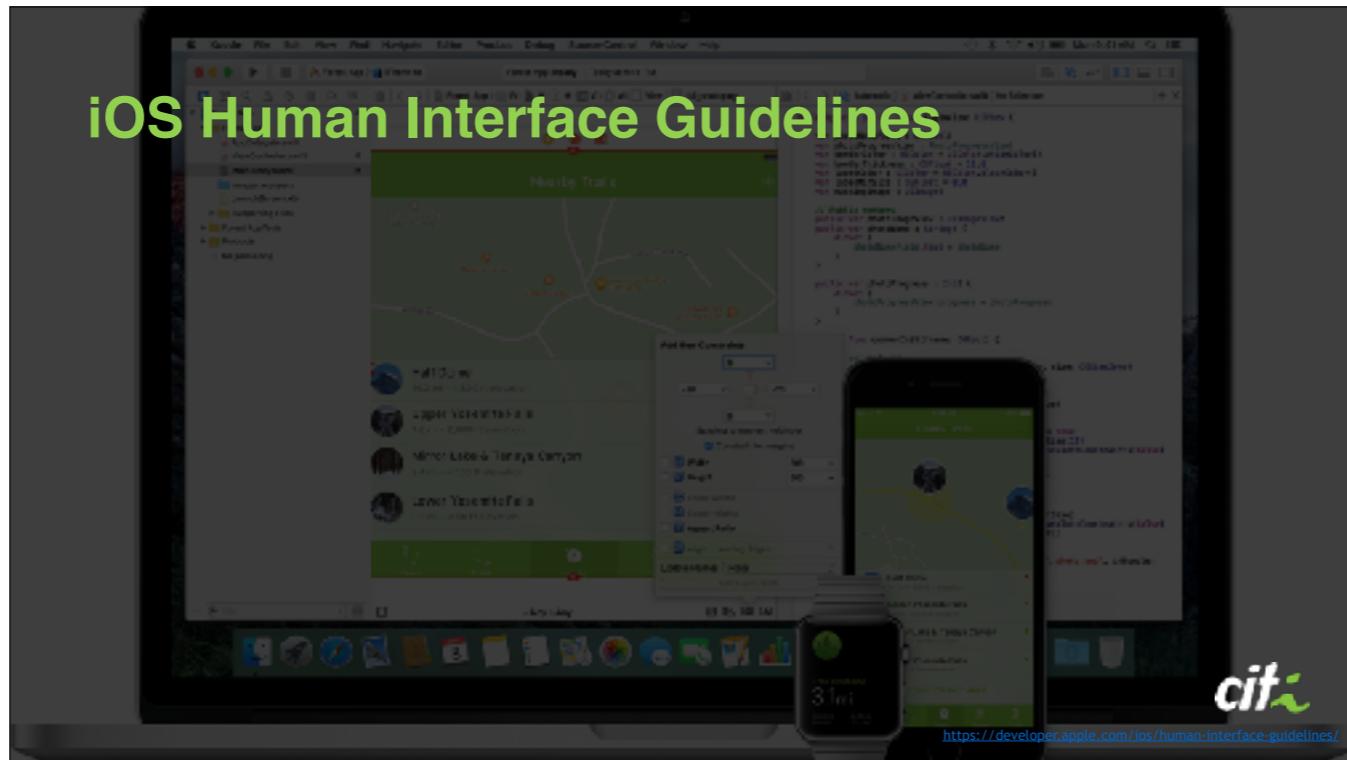
Your app enables users to post content anonymously but does not have the required precautions in place.

Next Steps

It is necessary that you put all of the following precautions in place:

- Age rating must reflect 17+
- The ability to report a post
- The ability to block a user
- The ability to immediately remove a post from the feed
- Contact information available in the app for users to report inappropriate activity
- A backend mechanism for identifying and blocking users who violate terms and conditions

PRJ CII



Aesthetic Integrity

Aesthetic integrity represents how well an app's appearance and behavior integrate with its function. For example, an app that helps people perform a serious task can keep them focused by using subtle, unobtrusive graphics, standard controls, and predictable behaviors. On the other hand, an immersive app, such as a game, can deliver a captivating appearance that promises fun and excitement, while encouraging discovery.

Consistency

A consistent app implements familiar standards and paradigms by using system-provided interface elements, well-known icons, standard text styles, and uniform terminology. The app incorporates features and behaviors in ways people expect.

Direct Manipulation

The direct manipulation of onscreen content engages people and facilitates understanding. Users experience direct manipulation when they rotate the device or use gestures to affect onscreen content. Through direct manipulation, they can see the immediate, visible results of their actions.

Feedback

Feedback acknowledges actions and shows results to keep people informed. The built-in iOS apps provide perceptible feedback in response to every user action. Interactive elements are highlighted briefly when tapped, progress indicators communicate the status of long-running operations, and animation and sound help clarify the results of actions.

Metaphors

People learn more quickly when an app's virtual objects and actions are metaphors for familiar experiences—whether rooted in the real or digital world. Metaphors work well in iOS because people physically interact with the screen. They move views out of the way to expose content beneath. They drag and swipe content. They toggle switches, move sliders, and scroll through picker values. They even flick through pages of books and magazines.

User Control

Throughout iOS, people—not apps—are in control. An app can suggest a course of action or warn about dangerous consequences, but it's usually a mistake for the app to take over the decision-making. The best apps find the correct balance between enabling users and avoiding unwanted outcomes. An app can make people feel like they're in control by keeping interactive elements familiar and predictable, confirming destructive actions, and making it easy to cancel operations, even when they're already underway.



Aesthetic Integrity

Aesthetic integrity represents how well an app's appearance and behavior integrate with its function. For example, an app that helps people perform a serious task can keep them focused by using subtle, unobtrusive graphics, standard controls, and predictable behaviors. On the other hand, an immersive app, such as a game, can deliver a captivating appearance that promises fun and excitement, while encouraging discovery.

Consistency

A consistent app implements familiar standards and paradigms by using system-provided interface elements, well-known icons, standard text styles, and uniform terminology. The app incorporates features and behaviors in ways people expect.

Direct Manipulation

The direct manipulation of onscreen content engages people and facilitates understanding. Users experience direct manipulation when they rotate the device or use gestures to affect onscreen content. Through direct manipulation, they can see the immediate, visible results of their actions.

Feedback

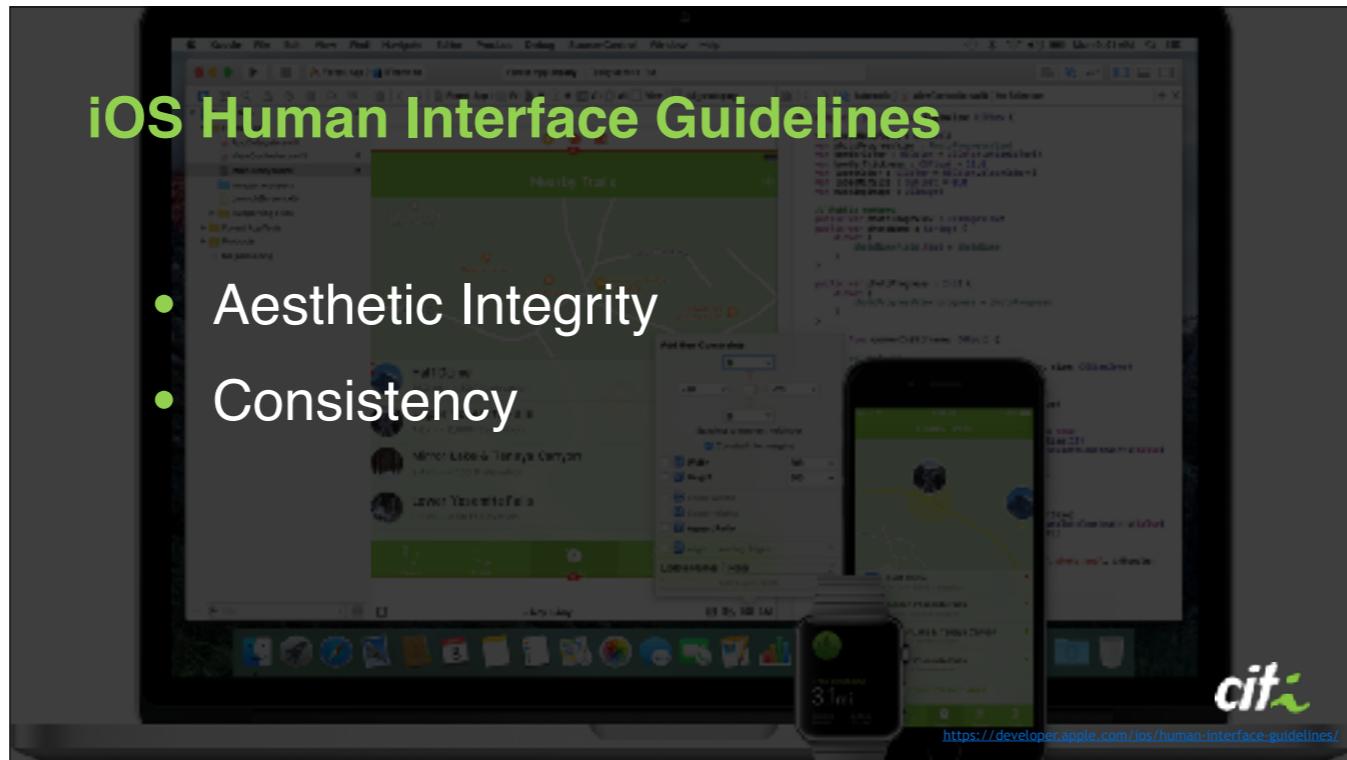
Feedback acknowledges actions and shows results to keep people informed. The built-in iOS apps provide perceptible feedback in response to every user action. Interactive elements are highlighted briefly when tapped, progress indicators communicate the status of long-running operations, and animation and sound help clarify the results of actions.

Metaphors

People learn more quickly when an app's virtual objects and actions are metaphors for familiar experiences—whether rooted in the real or digital world. Metaphors work well in iOS because people physically interact with the screen. They move views out of the way to expose content beneath. They drag and swipe content. They toggle switches, move sliders, and scroll through picker values. They even flick through pages of books and magazines.

User Control

Throughout iOS, people—not apps—are in control. An app can suggest a course of action or warn about dangerous consequences, but it's usually a mistake for the app to take over the decision-making. The best apps find the correct balance between enabling users and avoiding unwanted outcomes. An app can make people feel like they're in control by keeping interactive elements familiar and predictable, confirming destructive actions, and making it easy to cancel operations, even when they're already underway.



Aesthetic Integrity

Aesthetic integrity represents how well an app's appearance and behavior integrate with its function. For example, an app that helps people perform a serious task can keep them focused by using subtle, unobtrusive graphics, standard controls, and predictable behaviors. On the other hand, an immersive app, such as a game, can deliver a captivating appearance that promises fun and excitement, while encouraging discovery.

Consistency

A consistent app implements familiar standards and paradigms by using system-provided interface elements, well-known icons, standard text styles, and uniform terminology. The app incorporates features and behaviors in ways people expect.

Direct Manipulation

The direct manipulation of onscreen content engages people and facilitates understanding. Users experience direct manipulation when they rotate the device or use gestures to affect onscreen content. Through direct manipulation, they can see the immediate, visible results of their actions.

Feedback

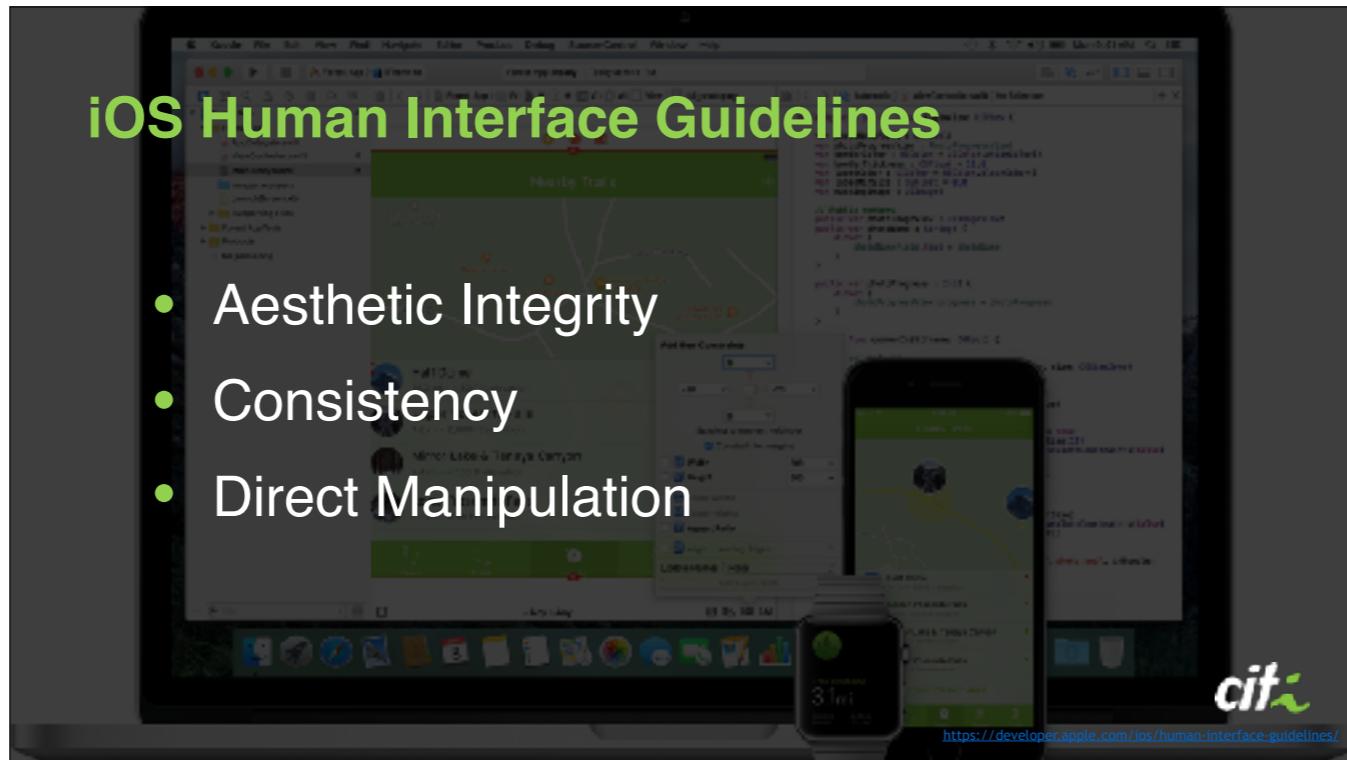
Feedback acknowledges actions and shows results to keep people informed. The built-in iOS apps provide perceptible feedback in response to every user action. Interactive elements are highlighted briefly when tapped, progress indicators communicate the status of long-running operations, and animation and sound help clarify the results of actions.

Metaphors

People learn more quickly when an app's virtual objects and actions are metaphors for familiar experiences—whether rooted in the real or digital world. Metaphors work well in iOS because people physically interact with the screen. They move views out of the way to expose content beneath. They drag and swipe content. They toggle switches, move sliders, and scroll through picker values. They even flick through pages of books and magazines.

User Control

Throughout iOS, people—not apps—are in control. An app can suggest a course of action or warn about dangerous consequences, but it's usually a mistake for the app to take over the decision-making. The best apps find the correct balance between enabling users and avoiding unwanted outcomes. An app can make people feel like they're in control by keeping interactive elements familiar and predictable, confirming destructive actions, and making it easy to cancel operations, even when they're already underway.



Aesthetic Integrity

Aesthetic integrity represents how well an app's appearance and behavior integrate with its function. For example, an app that helps people perform a serious task can keep them focused by using subtle, unobtrusive graphics, standard controls, and predictable behaviors. On the other hand, an immersive app, such as a game, can deliver a captivating appearance that promises fun and excitement, while encouraging discovery.

Consistency

A consistent app implements familiar standards and paradigms by using system-provided interface elements, well-known icons, standard text styles, and uniform terminology. The app incorporates features and behaviors in ways people expect.

Direct Manipulation

The direct manipulation of onscreen content engages people and facilitates understanding. Users experience direct manipulation when they rotate the device or use gestures to affect onscreen content. Through direct manipulation, they can see the immediate, visible results of their actions.

Feedback

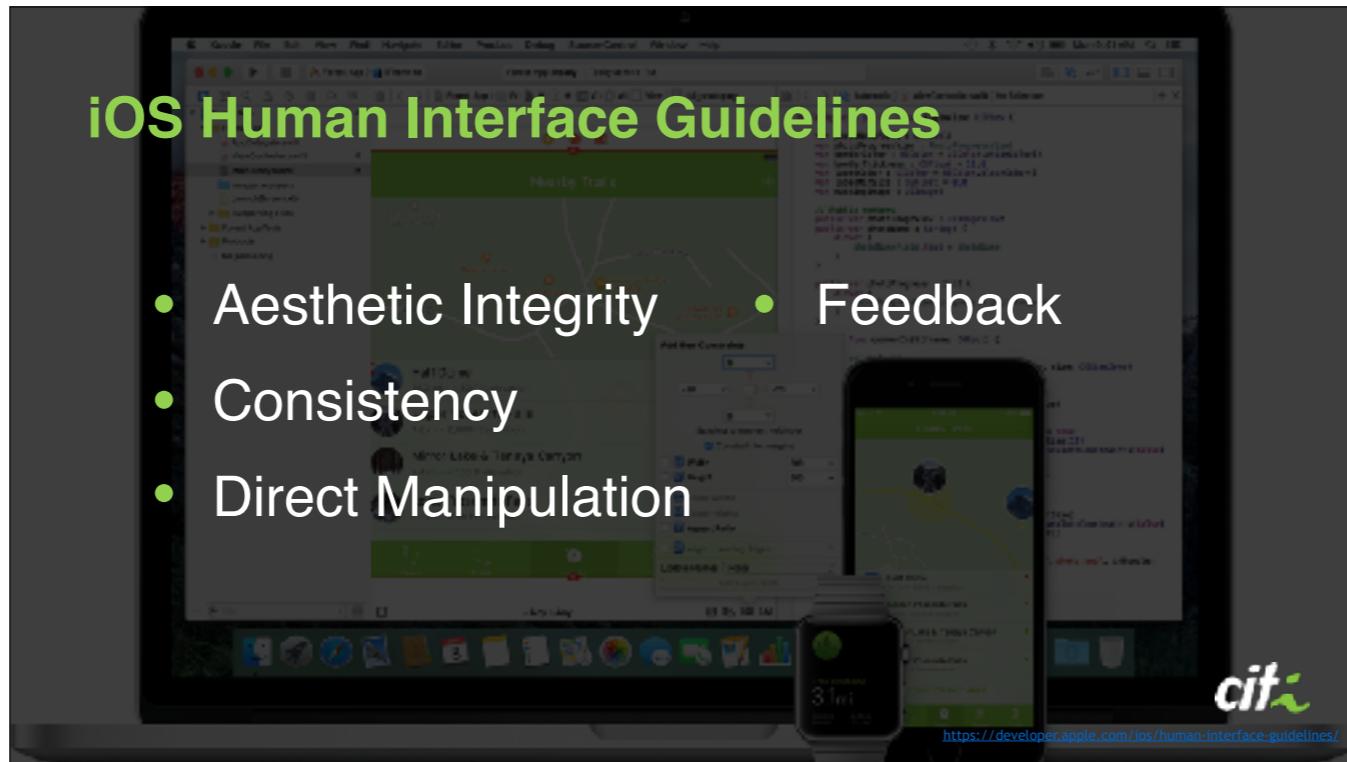
Feedback acknowledges actions and shows results to keep people informed. The built-in iOS apps provide perceptible feedback in response to every user action. Interactive elements are highlighted briefly when tapped, progress indicators communicate the status of long-running operations, and animation and sound help clarify the results of actions.

Metaphors

People learn more quickly when an app's virtual objects and actions are metaphors for familiar experiences—whether rooted in the real or digital world. Metaphors work well in iOS because people physically interact with the screen. They move views out of the way to expose content beneath. They drag and swipe content. They toggle switches, move sliders, and scroll through picker values. They even flick through pages of books and magazines.

User Control

Throughout iOS, people—not apps—are in control. An app can suggest a course of action or warn about dangerous consequences, but it's usually a mistake for the app to take over the decision-making. The best apps find the correct balance between enabling users and avoiding unwanted outcomes. An app can make people feel like they're in control by keeping interactive elements familiar and predictable, confirming destructive actions, and making it easy to cancel operations, even when they're already underway.



Aesthetic Integrity

Aesthetic integrity represents how well an app's appearance and behavior integrate with its function. For example, an app that helps people perform a serious task can keep them focused by using subtle, unobtrusive graphics, standard controls, and predictable behaviors. On the other hand, an immersive app, such as a game, can deliver a captivating appearance that promises fun and excitement, while encouraging discovery.

Consistency

A consistent app implements familiar standards and paradigms by using system-provided interface elements, well-known icons, standard text styles, and uniform terminology. The app incorporates features and behaviors in ways people expect.

Direct Manipulation

The direct manipulation of onscreen content engages people and facilitates understanding. Users experience direct manipulation when they rotate the device or use gestures to affect onscreen content. Through direct manipulation, they can see the immediate, visible results of their actions.

Feedback

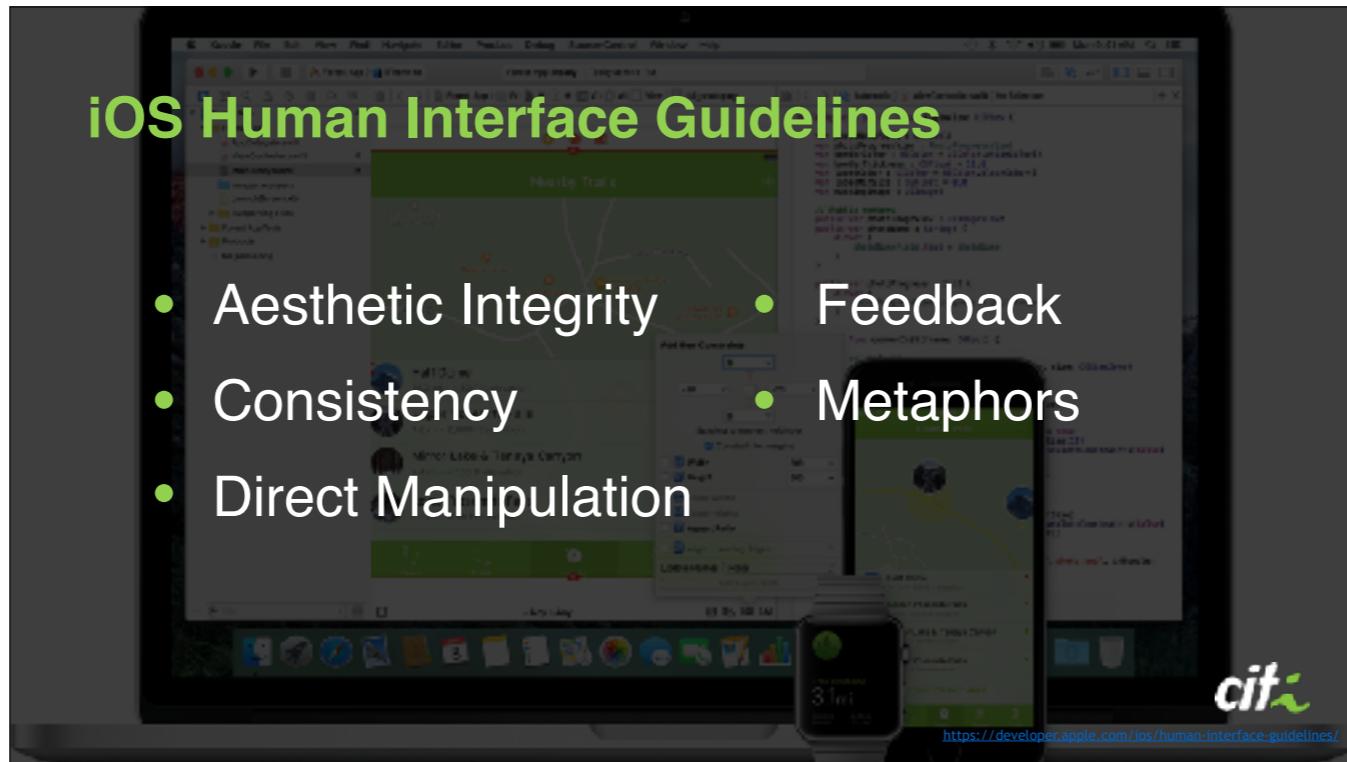
Feedback acknowledges actions and shows results to keep people informed. The built-in iOS apps provide perceptible feedback in response to every user action. Interactive elements are highlighted briefly when tapped, progress indicators communicate the status of long-running operations, and animation and sound help clarify the results of actions.

Metaphors

People learn more quickly when an app's virtual objects and actions are metaphors for familiar experiences—whether rooted in the real or digital world. Metaphors work well in iOS because people physically interact with the screen. They move views out of the way to expose content beneath. They drag and swipe content. They toggle switches, move sliders, and scroll through picker values. They even flick through pages of books and magazines.

User Control

Throughout iOS, people—not apps—are in control. An app can suggest a course of action or warn about dangerous consequences, but it's usually a mistake for the app to take over the decision-making. The best apps find the correct balance between enabling users and avoiding unwanted outcomes. An app can make people feel like they're in control by keeping interactive elements familiar and predictable, confirming destructive actions, and making it easy to cancel operations, even when they're already underway.



Aesthetic Integrity

Aesthetic integrity represents how well an app's appearance and behavior integrate with its function. For example, an app that helps people perform a serious task can keep them focused by using subtle, unobtrusive graphics, standard controls, and predictable behaviors. On the other hand, an immersive app, such as a game, can deliver a captivating appearance that promises fun and excitement, while encouraging discovery.

Consistency

A consistent app implements familiar standards and paradigms by using system-provided interface elements, well-known icons, standard text styles, and uniform terminology. The app incorporates features and behaviors in ways people expect.

Direct Manipulation

The direct manipulation of onscreen content engages people and facilitates understanding. Users experience direct manipulation when they rotate the device or use gestures to affect onscreen content. Through direct manipulation, they can see the immediate, visible results of their actions.

Feedback

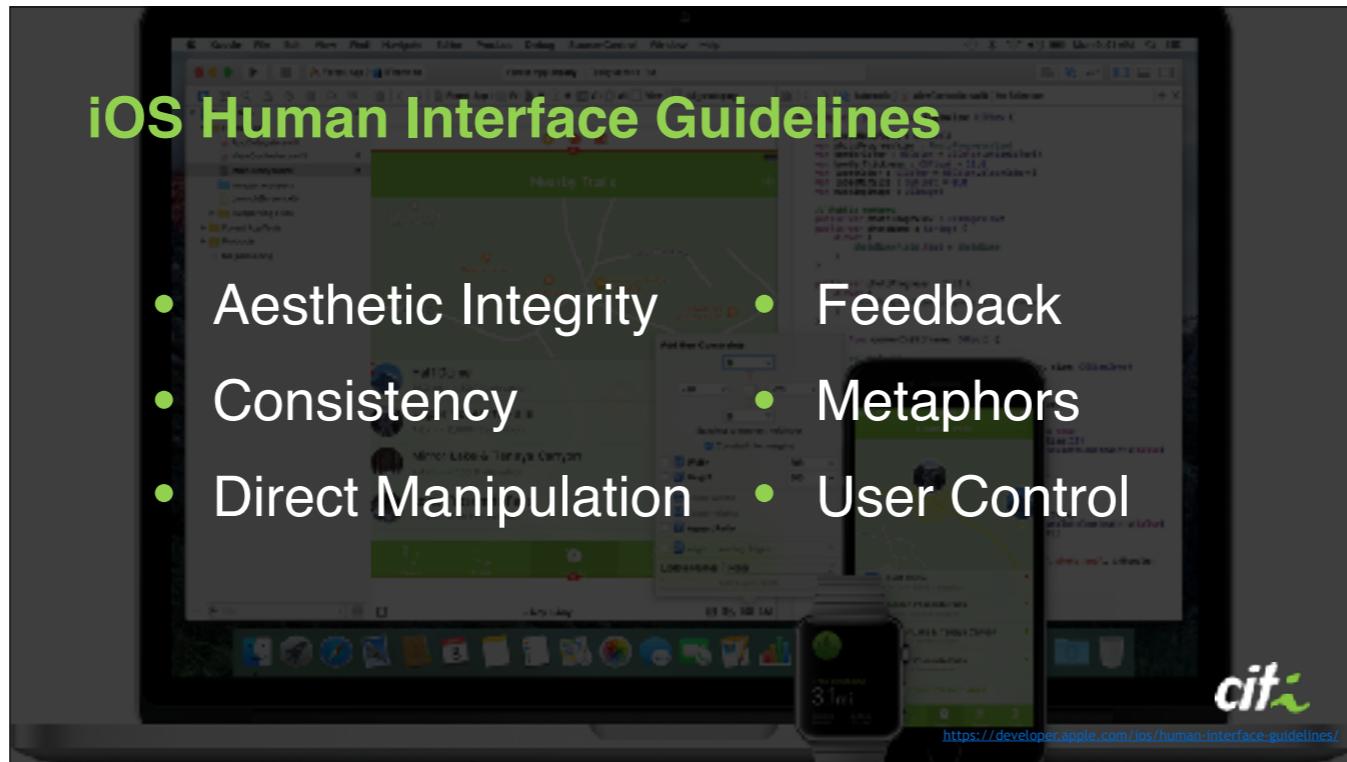
Feedback acknowledges actions and shows results to keep people informed. The built-in iOS apps provide perceptible feedback in response to every user action. Interactive elements are highlighted briefly when tapped, progress indicators communicate the status of long-running operations, and animation and sound help clarify the results of actions.

Metaphors

People learn more quickly when an app's virtual objects and actions are metaphors for familiar experiences—whether rooted in the real or digital world. Metaphors work well in iOS because people physically interact with the screen. They move views out of the way to expose content beneath. They drag and swipe content. They toggle switches, move sliders, and scroll through picker values. They even flick through pages of books and magazines.

User Control

Throughout iOS, people—not apps—are in control. An app can suggest a course of action or warn about dangerous consequences, but it's usually a mistake for the app to take over the decision-making. The best apps find the correct balance between enabling users and avoiding unwanted outcomes. An app can make people feel like they're in control by keeping interactive elements familiar and predictable, confirming destructive actions, and making it easy to cancel operations, even when they're already underway.



Aesthetic Integrity

Aesthetic integrity represents how well an app's appearance and behavior integrate with its function. For example, an app that helps people perform a serious task can keep them focused by using subtle, unobtrusive graphics, standard controls, and predictable behaviors. On the other hand, an immersive app, such as a game, can deliver a captivating appearance that promises fun and excitement, while encouraging discovery.

Consistency

A consistent app implements familiar standards and paradigms by using system-provided interface elements, well-known icons, standard text styles, and uniform terminology. The app incorporates features and behaviors in ways people expect.

Direct Manipulation

The direct manipulation of onscreen content engages people and facilitates understanding. Users experience direct manipulation when they rotate the device or use gestures to affect onscreen content. Through direct manipulation, they can see the immediate, visible results of their actions.

Feedback

Feedback acknowledges actions and shows results to keep people informed. The built-in iOS apps provide perceptible feedback in response to every user action. Interactive elements are highlighted briefly when tapped, progress indicators communicate the status of long-running operations, and animation and sound help clarify the results of actions.

Metaphors

People learn more quickly when an app's virtual objects and actions are metaphors for familiar experiences—whether rooted in the real or digital world. Metaphors work well in iOS because people physically interact with the screen. They move views out of the way to expose content beneath. They drag and swipe content. They toggle switches, move sliders, and scroll through picker values. They even flick through pages of books and magazines.

User Control

Throughout iOS, people—not apps—are in control. An app can suggest a course of action or warn about dangerous consequences, but it's usually a mistake for the app to take over the decision-making. The best apps find the correct balance between enabling users and avoiding unwanted outcomes. An app can make people feel like they're in control by keeping interactive elements familiar and predictable, confirming destructive actions, and making it easy to cancel operations, even when they're already underway.



Objective-C vs Swift



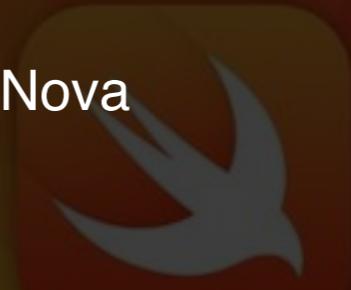
citr

Objective-C vs Swift

- Tradicional



- Nova



citr

Objective-C vs Swift

- Tradicional
- Menos Mudanças
- Nova
- Open source

citiz

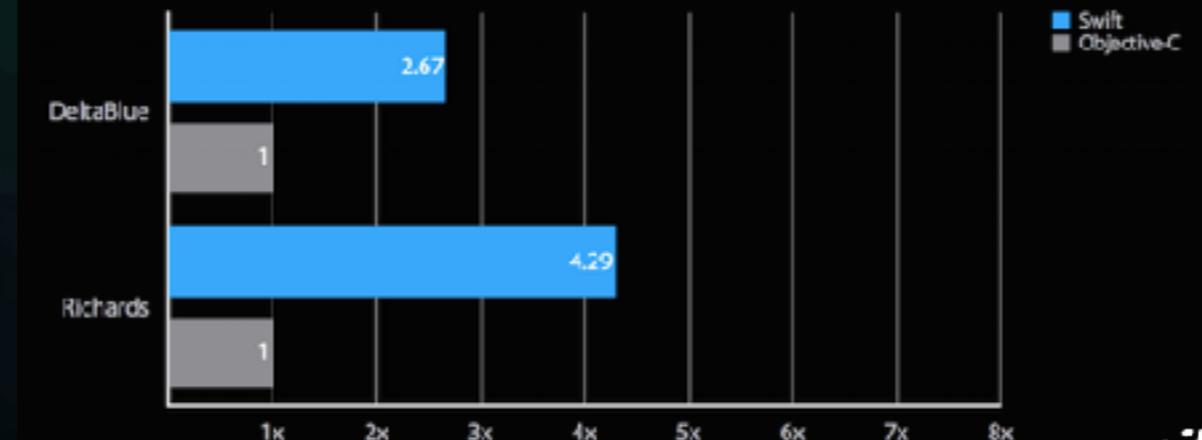
Objective-C vs Swift

- Tradicional
- Menos Mudanças
- Mais Adotada
- Nova
- Open source
- Adotada pela Apple

citiz

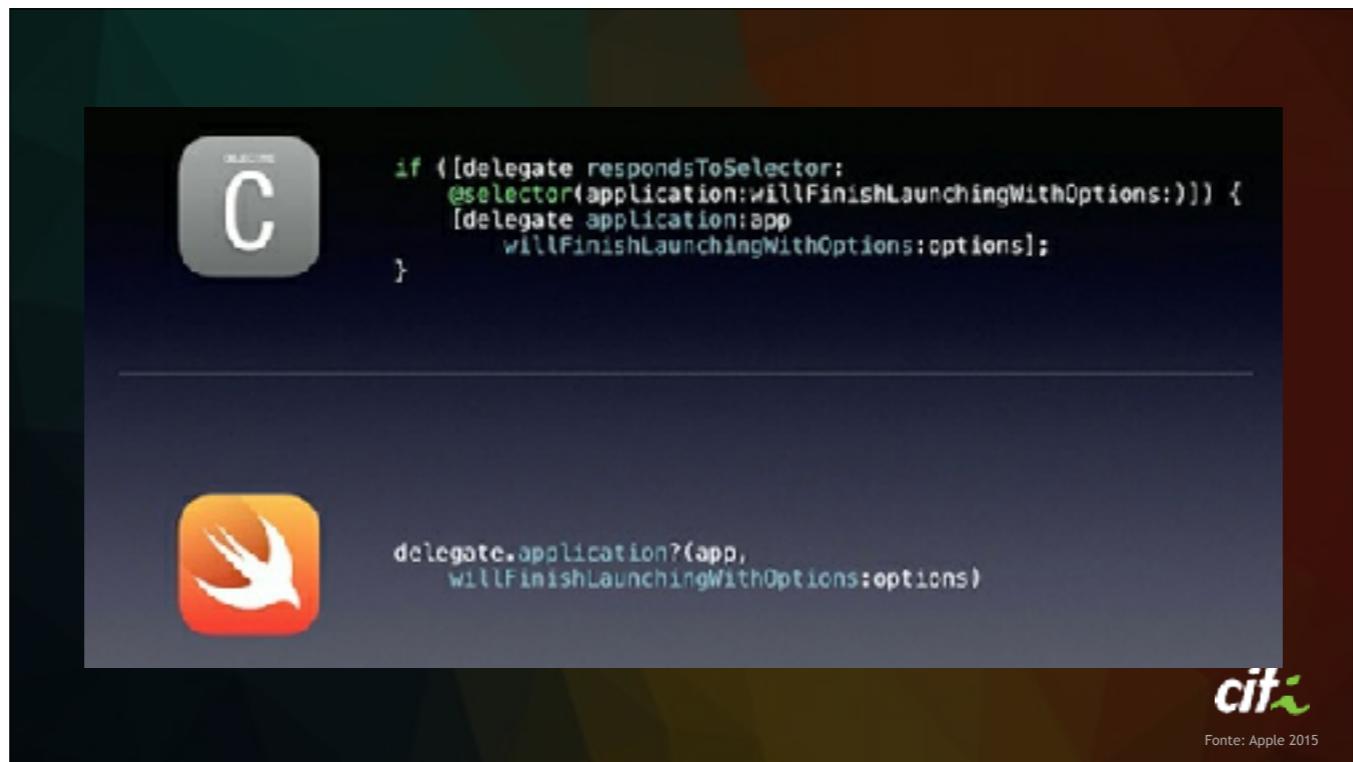
Swift vs. Objective-C

Program speed (higher is better)



Fonte: Apple 2015







Swift



Xcode

citi



```
// Comentários em Swift
```

```
/*  
Comentários  
com várias linhas  
*/
```



// Memória



// Variáveis

```
var nome = "Hilton"  
var idade: Int = 21
```



// Variáveis

```
var nome = "Hilton"  
var idade: Int = 21
```

Palavra-reservada
(keyword)



// Variáveis

```
var nome = "Hilton"  
var idade: Int = 21
```

Palavra-reservada **Identificador**
(keyword)



// Variáveis

```
var nome = "Hilton"  
var idade: Int = 21
```

Palavra-reservada Identificador Tipo
(keyword)



// Variáveis

```
var nome = "Hilton"  
var idade: Int = 21
```

Palavra-reservada Identificador Tipo operador
(keyword)



// Variáveis

```
var nome = "Hilton"  
var idade: Int = 21
```

Palavra-reservada <i>(keyword)</i>	Identificador	Tipo	Operador de atribuição	Valor (String) Valor (Int)
---------------------------------------	---------------	------	------------------------------	---



// Atribuição de novo valor

idade = 22

Palavra-reservada <i>(keyword)</i>	Identificador	Tipo	Operador de atribuição	Valor (Int)
---------------------------------------	---------------	------	------------------------------	-------------



// Atribuição de novo valor

idade = 22

Palavra reservada
(keyword) **X** Identificador **X** Operador de atribuição Valor (Int)



```
// Constantes
```

```
let diaNascimento = 2  
let nacionalidade: String = "Brasileiro"
```



```
// Constantes
```

```
let diaNascimento = 2  
let nacionalidade: String = "Brasileiro"
```

Palavra-reservada
(*keyword*)



```
// Constantes
```

```
let diaNascimento = 2  
let nacionalidade: String = "Brasileiro"
```

Palavra-reservada
(keyword)

Identificador



```
// Constantes
```

```
let diaNascimento = 2  
let nacionalidade: String = "Brasileiro"
```

Palavra-reservada
(keyword)

Identificador

Tipo



// Constantes

```
let diaNascimento = 2  
let nacionalidade: String = "Brasileiro"
```

Palavra-reservada
(keyword)

Identificador

Tipo

Operador
de
atribuição



```
// Constantes
```

```
let diaNascimento = 2  
let nacionalidade: String = "Brasileiro"
```

Palavra-reservada
(keyword)

Identificador

Tipo

Operador
de
atribuição

Valor (Int)
Valor (String)



// Tipos



// Tipos numéricos

```
let inteiro: Int = 10  
let decimal: Double = 3.14159265359  
let pi: Float = 3.14159265359
```

Valor (Int)
Valor (Double)
Valor (Float)



// Tipos numéricos

```
let inteiro: Int = 10
let decimal: Double = 3.14159265359
let pi: Float = 3.14159265359
                           //arredondado para 3.141593
```



```
// Booleans
```

```
let souRecifense: Bool = true  
var odeioSwift = false
```

Valor (**Bool**)



```
// Strings
```

```
let sobrenome: String = "Pintor"  
let cpf = "111-404-222.16"
```

Valor (String)



```
// Tuplas
```

```
let estado: (String, String) = ("Recife", "PE")
```



// Tuplas

```
let estado: (String, String) = ("Recife", "PE")
estado.0 // acessando primeiro valor da tupla estado
```

Identificador . Índice
Ponto



// Tuplas

```
let rg = (numero: 9153865, orgao: "SDS", UF: "PE")
```



// Tuplas

```
let rg = (numero: 9153865, orgao: "SDS", UF: "PE")
rg.numero // acessando valor "numero" da tupla rg
```

Identificador . Label
Ponto



// Optionals

```
var sentidoVida:Int?  
sentidoVida // nesse ponto não tem valor (nil)
```

Tipo ?
Interrogação



// Optionals

```
sentidoVida = 42  
sentidoVida // nesse ponto o valor é 42
```



```
// Forced unwrapping
```



```
var titulo: String?  
titulo = "PhD"  
var unwrapped = titulo! // extrai o valor do opcional
```

titulo é do tipo String?
unwrapped é do tipo String



```
// Optional binding

var opcional: Bool?
opcional = true

if let concreto = opcional {
    concreto // valor extraído: true
}
```

opcional é do tipo Bool?
concreto é do tipo Bool



// Operadores

cit

// Atribuição

nome = "Hilton"
idade = 21



// Aritmética

```
let dez: Int = 10  
let dois: Int = 2
```



// Aritmética

```
dez + dois    // 12
dez - dois    // 8
dez / dois    // 5
dez * dois    // 20
```

dez e dois são do mesmo tipo



// Resto

5 % 2 // 1

5 % 5 // 0

2 % 5 // 2

citi

// Resto

5

2

citi

// Resto

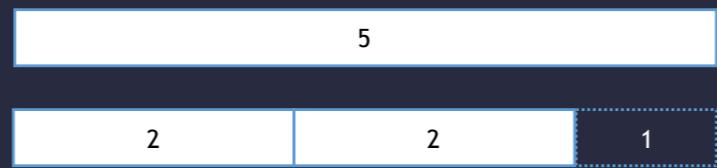
5

2

2

citi

// Resto



5 % 2 // 1

citi

// Concatenação

"olá" + "mundo" // resulta em: "olámundo"

"oi " + nome // resulta em: "oi Elton"

operador **+** realiza operações diferentes de acordo
com o **tipo dos operandos (Int, String, ...)**



// Comparação

2 == 2 // dois é igual a dois? true

10 < 1 // 10 é menor que 1? false



// Comparação

```
nome != "Hilton" /* o valor de nome é diferente de  
"Hilton"? false */  
idade >= 21 /* o valor de idade é maior ou igual a  
21? true */
```



// Negação

```
odeioSwift = false  
!odeioSwift // retorna true
```



// Conjunção (*And*)

```
true && true // retorna true  
false && true // retorna false  
false && false // retorna false
```



// Disjunção (*Or*)

```
true || true // retorna true  
false || true // retorna true  
false || false // retorna false
```



// Intervalos

0...3 // representa 0, 1, 2, 3

0..<3 // representa 0, 1, 2



// Coleções



```
// Arrays
```

```
var cidades = ["Recife", "Olinda"]  
cidades[0] // retorna "Recife"
```



// Arrays

```
var cidades = [ 0 | "Recife" | 1 | "Olinda"  
cidades[0] // retorna "Recife"
```



// Arrays - adição

```
cidades.append("Jaboatão")
cidades[2] // retorna "Jaboatão"
```



// Arrays - adição

```
0 | "Recife" | 1 | "Olinda" | .append("Jaboatão")
```

```
cidades[2] // retorna "Jaboatão"
```



// Arrays - adição

```
0 | "Recife" | 1 | "Olinda" | .append( 2 | "Jaboatão" | )
```

```
cidades[2] // retorna "Jaboatão"
```



// Arrays - remoção

0	"Recife"	1	"Olinda"	2	"Jaboatão"
---	----------	---	----------	---	------------

cidades.**removeLast()** // remove "Jaboatão"

cidades.**removeFirst()** // remove "Recife"

cidades.**removeAll()** // remove todos os elementos



// Arrays - remoção

0	"Recife"	1	"Olinda"
---	----------	---	----------

cidades.**removeLast()** // remove "Jaboatão"

cidades.**removeFirst()** // remove "Recife"

cidades.**removeAll()** // remove todos os elementos



// Arrays - remoção

0	"Olinda"
---	----------

```
cidades.removeLast() // remove "Jaboatão"  
cidades.removeFirst() // remove "Recife"  
cidades.removeAll() // remove todos os elementos
```



// Arrays - remoção

```
cidades.removeLast() // remove "Jaboatão"  
cidades.removeFirst() // remove "Recife"  
cidades.removeAll() // remove todos os elementos
```



// Arrays - contagem

```
var cidadesVisitadas: [String] = []
cidadesVisitadas.count // retorna 0 (zero)
```



// Arrays - contagem

```
cidadesVisitadas.append("Recife")  
cidadesVisitadas.count // retorna 1
```

0	"Recife"
---	----------



// Arrays - modificação

```
cidadesVisitadas[0] = "Olinda"  
cidadesVisitadas // ["Olinda"]
```

0	"Recife"
---	----------



// Arrays - modificação

```
cidadesVisitadas[0] = "Olinda"  
cidadesVisitadas // ["Olinda"]
```

0	"Olinda"
---	----------



// Arrays - acesso inválido

```
cidadesVisitadas[cidadesVisitadas.count]  
cidades[-1]
```

0	"Olinda"
---	----------



// Arrays - acesso inválido

cidadesVisitadas[cidadesVisitadas.count]
cidades[~~X~~]

0	"Olinda"
---	----------



// Dicionários

```
var capitais: [String: String] = ["PE": "Recife"]  
capitais["PE"] // retorna "Recife"
```

Chave	Valor
"PE"	"Recife"
...	nil



// Dicionários - adição

capitais["PB"] = "João Pessoa"

capitais // ["PB": "João Pessoa", "PE": "Recife"]

Chave	Valor
"PE"	"Recife"
"PB"	"João Pessoa"
...	nil



// Dicionários - modificação

capitais["PB"] = "Maria Pessoa"

capitais // ["PB": "Maria Pessoa", "PE": "Recife"]

Chave	Valor
"PE"	"Recife"
"PB"	"João Pessoa"
...	nil



// Dicionários - modificação

capitais["PB"] = "Maria Pessoa"

capitais // ["PB": "Maria Pessoa", "PE": "Recife"]

Chave	Valor
"PE"	"Recife"
"PB"	"Maria Pessoa"
...	nil



// Dicionários - remoção

```
capitais["PB"] = nil  
capitais.removeValue(forKey: "PE")
```

Chave	Valor
"PE"	"Recife"
"PB"	"João Pessoa"
...	nil



// Dicionários - remoção

```
capitais["PB"] = nil  
capitais.removeValue(forKey: "PE")
```

Chave	Valor
"PE"	"Recife"
"PB"	nil
...	nil



// Dicionários - remoção

```
capitais["PB"] = nil  
capitais.removeValue(forKey: "PE")
```

Chave	Valor
"PE"	nil
"PB"	nil
...	nil



// Dicionários - contagem

```
var dictVazio: [Int: String] = [:]  
dictVazio.count // retorna 0
```

Chave	Valor
...	nil



// Dicionários - acesso sem valor

```
dictVazio[0] // retorna nil  
dictVazio[-1] // retorna nil
```

Chave	Valor
0	nil
-1	nil
...	nil



// Controle de Fluxo



// Desvios Condicionais

citiz

```
// If (se)

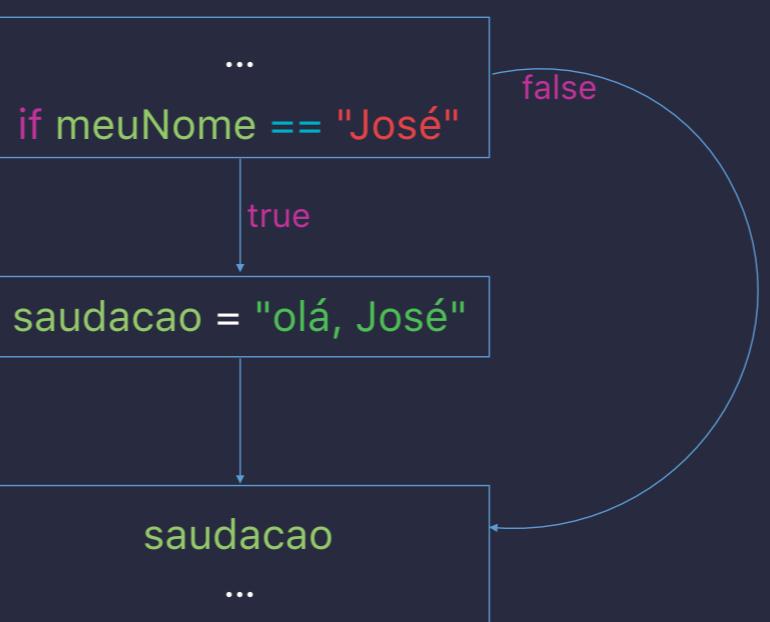
var meuNome = "José"
var saudacao = ""

if meuNome == "José" {
    // executar caso verdadeiro:
    saudacao = "Olá, José"
}

saudacao // tem valor de "Olá, José"
```



// If (se)



citi

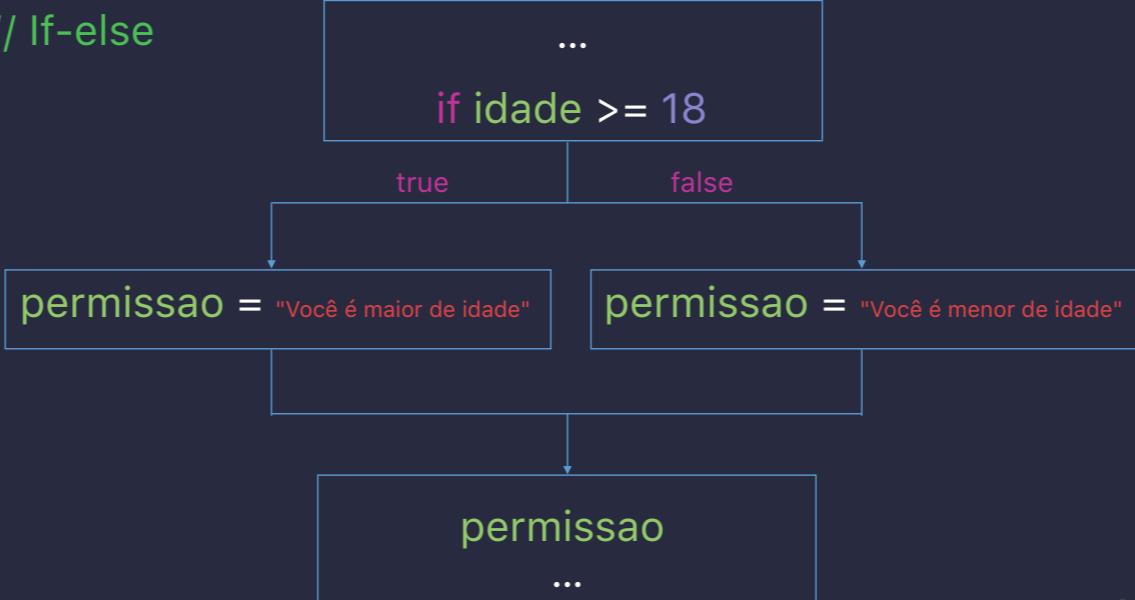
```
// If-else  idade = 17
var permissao = ""

if idade >= 18 {
    permissao = "Você é maior de idade"
} else {
    // executar caso falso:
    permissao = "Você é menor de idade"
}

permissao // tem valor de "Você é menor
de idade"
```



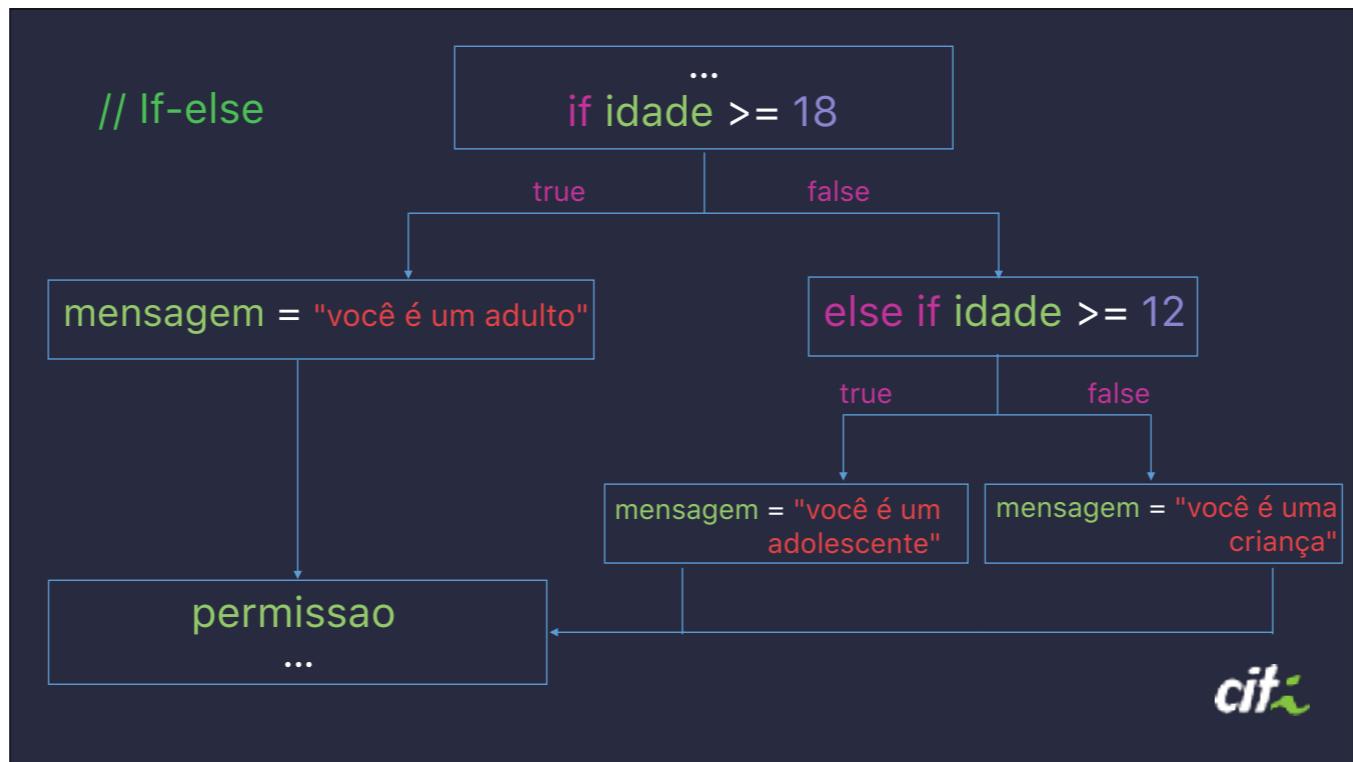
// If-else



citi

```
// else if      idade = 16
                var mensagem = ""
                if idade >= 18 {
                    mensagem = "você é um adulto"
                } else if idade >= 12{
                    mensagem = "você é um adolescente"
                } else {
                    mensagem = "você é uma criança"
                }
                mensagem // tem valor de "você é um adolescente"
```





cit

// switch

```
var animal = "Cachorro"
var som: String

switch animal {
    case "Gato":
        som = "miado"
    case "Cachorro":
        som = "latido"
    case "Leão":
        som = "rugido"
    default:
        // se for qualquer outro animal
        som = "indefinido"
}
som // tem valor de "latido"
```



```
// switch
animal = "Gata"
som = ""
switch animal {
    case "Gato", "Gata":
        som = "miado"
    case "Cachorro", "Cadela":
        som = "latido"
    case "Leão", "Leoa":
        som = "rugido"
    default:
        // se for qualquer outro animal
        som = "indefinido"
}
som // tem valor de "miado"
```



// Loops



```
// for in
```

```
cidadesVisitadas = ["Recife", "Olinda", "Jaboatão"]
mensagem = "nós visitamos:"
for cidade in cidadesVisitadas {
    mensagem = mensagem + " " + cidade
}
mensagem // tem valor de "nós visitamos: Recife Olinda Jaboatão"
```



// for in

```
for cidade in cidadesVisitadas {  
    mensagem = mensagem + " " + cidade  
}
```

Execução	Valor de cidade	Valor de mensagem
Antes do loop	Não existe	nós visitamos:
Fim da 1a Iteração	Recife	nós visitamos: Recife
Fim da 2a Iteração	Olinda	nós visitamos: Recife Olinda
Fim da 3a Iteração	Jaboatão	nós visitamos: Recife Olinda Jaboatão
Após o loop	Não existe	nós visitamos: Recife Olinda Jaboatão

cidadesVisitadas = ["Recife", "Olinda", "Jaboatão"]



```
// for in range

var quadrados: [Int] = []

for numero in 1...10 {
    let quadrado = numero * numero
    quadrados.append(quadrado)
}

quadrados // tem valor de [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```



```
// while (enquanto)

    var preco = 100
    var devoComprar = false
    //enquanto o preço for caro não devo comprar
    while !devoComprar {

        preco = preco - 1 // preço diminui
        //se o preço chegar em 50, eu devo comprar.
        if preco == 50 {
            devoComprar = true
        }
    }

    preco      // tem valor de 50
    devoComprar // tem valor de true
```



// Funções e Closures



// funções - declaração

```
func maior(primeiro: Int, segundo: Int) -> Int{  
    if primeiro > segundo {  
        return primeiro  
  
    } else {  
        return segundo  
    }  
}
```



```
// funções - declaração

func maior(primeiro: Int, segundo: Int) -> Int{
    if primeiro > segundo {
        return primeiro
    }
    else {
        return segundo
    }
}
```



// funções - declaração

```
func maior(primeiro: Int, segundo: Int) -> Int{  
    if primeiro > segundo {  
        return primeiro  
  
    } Palavra-reservada else { Identificador  
        (keyword) return segundo  
    }  
}
```



// funções - declaração

func maior(primeiro: Int, segundo: Int) -> Int{

 if primeiro > segundo {

 return primeiro

} else {
 Palavra-reservada
(keyword)

 Identificador (nome dos
parâmetros

 return segundo

}

}



// funções - declaração

func maior(primeiro:Int, segundo:Int) -> Int{

 if primeiro > segundo {

 return primeiro

} else {
 Palavra-reservada
 (keyword) Identificador (nome dos parâmetros Tipo dos parâmetros)

 return segundo

}

}



// funções - declaração

func maior(primeiro: Int, segundo: Int) -> Int{

 if primeiro > segundo {

 return primeiro

} else {

 Identificador (nome dos parâmetros Tipo dos parâmetros) ->

 return segundo

}

}



// funções - declaração

Palavra-reservada **(keyword)** Identificador $\left(\begin{array}{cc} \text{nome dos} & \text{Tipo dos} \\ \text{parâmetros} & \text{parâmetros} \end{array} \right)$ -> **Tipo de retorno** $\left\{ \text{corpo} \right\}$

func maior(primeiro: Int, segundo: Int) -> Int {

 if primeiro > segundo {
 return primeiro

 } else {
 return segundo
 }
}



```
// funções - chamada

let um = 1
let dois = 2

let maximo = maior(primeiro: um, segundo: dois)

maximo // tem valor de 2
```



```
// funções - chamada

let um = 1
let dois = 2

let maximo = maior(primeiro: um, segundo: dois)

maximo // tem identificador de 2
```



```
// funções - chamada
```

```
let um = 1  
let dois = 2
```

```
let maximo = maior(primeiro: um, segundo: dois)
```

```
maximo // tem valor de   
nome dos parâmetros
```



// funções - chamada

```
let um = 1  
let dois = 2
```

```
let maximo = maior(primeiro:um, segundo:dois)
```

maximo // Identificador de (nome dos parâmetros argumentos)



// funções são tipos

O **tipo** de uma função é dado pelo tipo de seus **parâmetros** e de seu **retorno**, separados por uma seta.



```
// funções são tipos  
  
func maior(primeiro: Int, segundo: Int) -> Int{...}
```



```
// funções são tipos

func maior(primeiro: Int, segundo: Int) -> Int{...}

(primeiro: Int, segundo: Int) -> Int
```



// funções são tipos

func maior(primeiro: Int, segundo: Int) -> Int{...}

(primeiro: Int, segundo: Int) -> Int

(Int, Int) -> Int



// funções de primeira ordem

Funções que recebem como
parâmetro, ou **retornam**, outras
funções.



```
// funções de primeira ordem

var array = [1, 2, 3]

func somaUm(valor: Int) -> Int {
    return valor + 1
}

// map aplica a função somaUm a cada elemento do array
var novoArray = array.map(somaUm)

novoArray // tem valor de [2, 3, 4]
```



```
// funções de primeira ordem
```

```
func somaUm(valor: Int) -> Int {  
    return valor + 1  
}
```

```
(Int) -> Int
```



```
// map
```

```
var array = [1, 2, 3]  
var novoArray = array.map(somaUm)
```



```
// map
```

```
var array = [1, 2, 3]
```

```
var novoArray = [ somaUm(valor: array[0]), somaUm(valor: array[1]), somaUm(valor: array[2]) ]
```



// closures

Funções **anônimas** que podem ser definidas no mesmo lugar em que são chamados



```
// map com closure

array = [1, 2, 3]
novoArray = []

novoArray = array.map({ (numero: Int) -> Int in
    return numero + 1
})

novoArray // tem valor de [2, 3, 4]
```



```
// closures

novoArray = array.map({(numero: Int) -> Int in
    return numero + 1
})

{                                         }
```

citiz

```
// closures
```

```
novoArray = array.map({ (numero: Int) -> Int in  
    return numero + 1  
})
```

{ (nome dos
parâmetros) }

citiz

```
// closures
```

```
novoArray = array.map({ (numero:Int) -> Int in  
    return numero + 1  
})
```

$\left\{ \begin{pmatrix} \text{nome dos} & \text{Tipo dos} \\ \text{parâmetros} & \text{parâmetros} \end{pmatrix} \right\}$

citiz

```
// closures
```

```
novoArray = array.map({ (numero: Int) -> Int in  
    return numero + 1  
})
```

{ $\begin{pmatrix} \text{nome dos} & \text{Tipo dos} \\ \text{parâmetros} & \text{parâmetros} \end{pmatrix}$ -> Tipo de
 retorno }



```
// closures
```

```
novoArray = array.map({ (numero: Int) -> Int  
    return numero + 1  
})
```

{ (nome dos Tipo dos
parâmetros parâmetros) -> Tipo de Palavra-reservada
 retorno (keyword) }



```
// closures
```

```
novoArray = array.map({ (numero: Int) -> Int in  
    return numero + 1  
})
```

{ (nome dos parâmetros Tipo dos parâmetros) -> { Tipo de retorno Palavra-reservada (keyword) corpo } }



```
// função vs closure
```

```
func somaUm(valor: Int) -> Int {  
    return valor + 1  
}
```

==

```
{ (numero: Int) -> Int in  
    return numero + 1  
})
```



```
// closure enxuto  
  
novoArray = array.map({ (numero: Int) -> Int in  
    return numero + 1  
})
```



```
// closure enxuto

novoArray = array.map({ (numero: Int) -> Int in
    return numero + 1
})
```



```
// closure enxuto  
  
novoArray = array.map({ numero in  
    return numero + 1  
})
```



```
// closure enxuto  
  
novoArray = array.map({ numero in  
    return numero + 1  
})
```



```
// closure enxuto  
  
novoArray = array.map({  
    return numero + 1  
})
```



```
// closure enxuto  
  
novoArray = array.map({  
    return      $0 + 1  
})
```



```
// closure enxuto
```

```
novoArray = array.map({  
    $0 + 1  
})
```



```
// closure enxuto
```

```
novoArray = array.map({ $0 + 1 })
```



```
// closure enxuto
```

```
novoArray = array.map { $0 + 1 }
```



```
// closure enxuto

array = [1, 2, 3]
novoArray = []

novoArray = array.map {$0 + 1}

novoArray // tem valor de [2, 3, 4]
```



// função vs closure II

```
func somaUm(valor: Int) -> Int {  
    return valor + 1  
}  
===  
{ (numero: Int) -> Int in  
    return numero + 1  
}  
===  
{$0 + 1}
```



// Exercícios



// Exercício 01-a

Ano Bissexto

Chama-se ano bissexto o ano ao qual é acrescentado um dia extra, ficando ele com 366 dias.

Ocorrendo a cada **quatro anos** (exceto anos **múltiplos de 100** que não são **múltiplos de 400**).

Dado um ano como entrada,
identifique se o ano é
bissexto.



// Exercício 01-b

Ano Bissexto

Chama-se ano bissexto o ano ao qual é acrescentado um dia extra, ficando ele com 366 dias.

Ocorrendo a cada **quatro anos** (exceto anos **múltiplos de 100** que não são **múltiplos de 400**).

Calcule os anos bissextos de
1988 a 2017



// Exercício 01-c

Ano Bissexto

Chama-se ano bissexto o ano ao qual é acrescentado um dia extra, ficando ele com 366 dias.

Ocorrendo a cada **quatro anos** (exceto anos **múltiplos de 100** que não são **múltiplos de 400**).

Filtre um array de anos,
deixando somente os
bissextos



// Aula 02



// Dúvidas da Aula 01



Dúvidas em aberto, esclarecimentos, e correções.

```
// operações de tipos diferentes
```

```
var opcional: Int? = 1
```

```
var concreto: Int = 1
```

```
opcional + concreto ! Value of optional type not unwrapped
```



- Operações devem ser com operandos do mesmo tipo
- Opcional e concretos não são do mesmo tipo
- erro de compilação

```
// operações de tipos diferentes
```

```
var opcional: Int? = 1
```

```
var concreto: Int = 1
```

```
opcional! + concreto // 2
```



- é preciso extrair o valor de opcional
- force unwrapping
- para realizar a operação

```
// operações de tipos diferentes
```

```
var int: Int = 1  
var double: Double = 1
```

```
int + double
```

Binary operator '+' cannot be applied to operands of type 'Int' and 'Double'



- dois tipos diferentes
- erro de compilação

```
// operações de tipos diferentes
```

```
var int: Int = 1  
var double: Double = 1
```

```
Double(int) + double // 2  
int + Int(double) // 2
```



- conversão de um tipo em outro

```
// operações de tipos diferentes
```

```
var string: String = "1"  
int = 1
```

```
string + int
```

Binary operator '+' cannot be applied to operands of type 'String' and 'Int'



- dois tipos diferentes
- erro de compilação

```
// operações de tipos diferentes
```

```
var string: String = "1"
int = 1

Int(string)! + int    // 2
string + String(int) // "11"
```



- converter String em Int
 - gera opcional
 - pode ter ou não um valor numérico na string
- converter de Int para String
 - o + agora é concatenação

```
// switch com intervalo
    var valor = 11
    var mensagem = ""
    switch valor {
        case Int.min..<0:
            mensagem = "negativo"
        case 0...Int.max:
            mensagem = "positivo"
        default:
            mensagem = "impossível"
    }
```

Int.max = 9223372036854775807
Int.min = -9223372036854775808



<https://stackoverflow.com/questions/36476599/swift-switch-statement-considered-all-cases-of-int-but-compiler-still-display-e>

- Foi falado sobre usar switch para verificar se um valor pertence a um intervalo.
- Essa é uma forma de se fazer.
- Nunca vai entrar no caso default, mas mesmo assim o compilador obriga.

// compilador não é tão esperto

"Unfortunately, integer operations like '...' and '<' are just plain functions to Swift, so it'd be difficult to do this kind of analysis.

Even with special case understanding of integer intervals, **I think there are still cases in the full generality of pattern matching for which exhaustiveness matching would be undecidable.**

We may eventually be able to handle some cases, but there will always be special cases involved in doing so."



<https://stackoverflow.com/questions/36476599/swift-switch-statement-considered-all-cases-of-int-but-compiler-still-display-e>

- Mensagem de um engenheiro da Apple.
- Compilador não é tão inteligente para operadores de intervalo e comparação.
- ele não tem como saber se o switch está exaustivo.

```
// switch com intervalo
    var valor = 11
    var mensagem = ""
    switch valor {
        case Int.min..<0:
            mensagem = "negativo"
        case 0...Int.max:
            mensagem = "positivo"
        default:
            mensagem = "impossível"
    }
```

Int.max = 9223372036854775807
Int.min = -9223372036854775808



<https://stackoverflow.com/questions/36476599/swift-switch-statement-considered-all-cases-of-int-but-compiler-still-display-e>

- como o caso default nunca poderá ser atingido
- podemos usar break (ignorar o caso)
- encerra a execução do switch sem realizar nenhuma ação

```
// switch com intervalo
    var valor = 11
    var mensagem = ""
    switch valor {
        case Int.min..<0:
            mensagem = "negativo"
        case 0...Int.max:
            mensagem = "positivo"
        default:
```

```
Int.max = 9223372036854775807
Int.min = -9223372036854775808
```

```
}
```



<https://stackoverflow.com/questions/36476599/swift-switch-statement-considered-all-cases-of-int-but-compiler-still-display-e>

- como o caso default nunca poderá ser atingido
- podemos usar break (ignorar o caso)
- encerra a execução do switch sem realizar nenhuma ação

```
// switch com intervalo
    var valor = 11
    var mensagem = ""
    switch valor {
        case Int.min..<0:
            mensagem = "negativo"
        case 0...Int.max:
            mensagem = "positivo"
        default:
            break
    }
    Int.max = 9223372036854775807
    Int.min = -9223372036854775808
```

<https://stackoverflow.com/questions/36476599/swift-switch-statement-considered-all-cases-of-int-but-compiler-still-display-e>



- como o caso default nunca poderá ser atingido
- podemos usar break (ignorar o caso)
- encerra a execução do switch sem realizar nenhuma ação

```
// compilador não é tão esperto
```

Although **break** is not required in Swift, you can use a **break** statement to match and **ignore a particular case** or to break out of a matched case before that case has completed its execution.



https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/ControlFlow.html

- Break não é necessário, é opcional
- usado para ignorar um caso
 - ou para sair do caso antes do fim da execução

```
// switch com where
```

```
var someVar = 3
switch someVar {
    case let x where x < 0:
        mensagem = "\(x) é menor que zero"
    case let x where x == 0:
        mensagem = "\(x) é igual a zero"
    case let x where x > 0:
        mensagem = "\(x) é maior que zero"
    default:
        break
}
```

<https://stackoverflow.com/questions/36476599/swift-switch-statement-considered-all-cases-of-int-but-compiler-still-display-e>



- switch com comparações
- é necessário o uso do where
- captura o valor em X, e compara no where
- interpolação de strings
- break

```
// switch com Where
```

```
switch someVar {  
    case _ where someVar < 0:  
        mensagem = "\$(someVar) é menor que zero"  
    case _ where someVar == 0:  
        mensagem = "\$(someVar) é igual a zero"  
    case _ where someVar > 0:  
        mensagem = "\$(someVar) é maior que zero"  
    default:  
        break  
}
```



<https://stackoverflow.com/questions/36476599/swift-switch-statement-considered-all-cases-of-int-but-compiler-still-display-e>

- não precisamos dar um novo nome ao valor
 - ignoramos com _
 - utilizamos o identificador original

```
// switch com fallthrough
```

In Swift, **switch** statements **don't fall through** the bottom of each case and into the next one.

That is, the entire **switch** statement **completes its execution as soon as the first matching case is completed**



https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/ControlFlow.html

- Só cai em um caso.
- finaliza o switch após executar o caso

```
// switch com fallthrough

let inteiro = 5
var descricao = "O número \(inteiro) é"
switch inteiro {
    case 2, 3, 5, 7, 11, 13, 17, 19:
        descricao += " primo, e também"
    default:
        descricao += " um inteiro."
}
descricao // "O número 5 é primo, e também um inteiro."
```



https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/ControlFlow.html

- para fazer com que execute mais de um caso
- que depois de um caia no outro
 - fallthrough

```
// switch com fallthrough

let inteiro = 5
var descricao = "O número \u201c(inteiro) é"
switch inteiro {
    case 2, 3, 5, 7, 11, 13, 17, 19:
        descricao += " primo, e também"
        fallthrough
    default:
        descricao += " um inteiro."
}
descricao // "O número 5 é primo, e também um inteiro."
```



https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/ControlFlow.html

- para fazer com que execute mais de um caso
- que depois de um caia no outro
 - fallthrough

```
// switch com fallthrough
```

The **fallthrough** keyword **does not check** the case **conditions** for the **switch** case that it causes execution to fall into.

The **fallthrough** keyword simply causes code execution to **move directly to the statements inside** the next case (or **default** case) block, as in C's standard **switch** statement behavior.



https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/ControlFlow.html

- não checa o caso
 - cai direto no corpo do próximo case
 - igual a C

// repeat-while

Executa o que tem dentro do **repeat** antes de conferir a condição do **while**

Similar ao do-while de outras linguagens.



- Foi perguntado sobre do-while
- Repeat
 - Executa a ação
 - depois confere a condição (while)

```
// repeat-while

var valorAtual = 0
var valorEsperado = 10

repeat {
    valorAtual = valorAtual + 1
} while valorAtual != valorEsperado

valorAtual // 10
```



REPETE X ENQUANTO Y
• incrementa o valor atual
• até ser igual ao esperado

// Exercícios



// Exercício 02

Scrabble

No jogo scrabble, procura-se fazer palavras com o máximo de pontos. Cada letra da palavra tem uma pontuação

Letter	Value
A, E, I, O, U, L, N, R, S, T	1
B, G	2
D, C, M, P	3
F, H, V, W, Y	4
K	5
J, X	8
Q, Z	10

Faça uma função que dado uma palavra, retorne a sua pontuação correspondente.



adaptado de: <http://exercism.io/exercises/swift/scrabble-score/readme>

// Exercício 03

Calculadora Escrita

Avalie o resultado das seguintes operações:

```
let soma = "Quanto é 5 mais 13?"  
let subtracao = "Quanto é 15 menos 10?"  
let multiplicacao = "Quanto é 1 vezes 13?"  
let divisao = "Quanto é 15 sobre 5?"
```

O resultado da expressão deve ser impresso (print) no console.

//Dica: procure por split



adaptado de: <http://exercism.io/exercises/swift/wordy/readme>

// Exercício 04

Dígitos

Implemente uma **função recursiva** chamada `digitos`, que recebe como entrada um `numero` positivo e retorna um `Array` contendo os dígitos desse número, em ordem.

// exemplo:

```
digitos(213) // retorna [2, 1, 3]
```



adaptado de: <https://www.weheartswift.com/recursion/>

// Exercício 05-a Array de Dicionários

Dado um **array de dicionários**, onde cada dicionário contém exatamente duas chaves: "nome", e "sobrenome".

Armazene em uma variável **nomes**, todos os nomes contidos nos dicionários, e em uma variável **sobrenomes**, todos os sobrenomes contidos nos dicionários.

```
// exemplo:  
var pessoas: [[String:String]] = [  
    [ "nome": "Hilton",  
      "sobrenome": "Pintor"  
    ],  
    [ "nome": "Elton",  
      "sobrenome": "Santana"  
    ],  
    [ "nome": "Daniel",  
      "sobrenome": "Oliveira"  
    ],  
    [ "nome": "Clarissa",  
      "sobrenome": "Pessoa"  
    ],  
    [ "nome": "Fanny",  
      "sobrenome": "Chien"  
  ]
```



adaptado de: <https://www.weheartswift.com/recursion/>

// Exercício 05-b Array de Dicionários

construindo acima do exemplo anterior, construa um array nomeCompleto onde cada elemento é uma string que contem o nome e sobrenome de cada pessoa separados por espaço.



adaptado de: <https://www.weheartswift.com/recursion/>

// Aula 03



// Dúvidas da Aula 02



Dúvidas em aberto, esclarecimentos, e correções.

```
// alterando valor de parâmetro
```

```
func funcao(parametro: Int) {  
    parametro = 10  
}
```



- Parâmetros são constantes
 - não é possível mudar o seu valor

```
// alterando valor de parâmetro
```

```
func funcao(parametro: Int) {  
    parametro = 10 // Cannot assign to value: 'parametro' is a 'let' constant  
}
```



- Parâmetros são constantes
 - não é possível mudar o seu valor

```
// atribuição a variável
```

```
func funcao(parametro: Int) {  
    var parametroLocal = parametro  
    parametroLocal = 10  
}
```



- para mudar o valor de um parâmetro dentro da função
 - atribuímos o mesmo a uma variável

```
// mudanças locais

var valor = 1

funcao(parametro: valor)

valor // continua sendo 1
```



- mudanças feitas no escopo de uma função
 - não alteram valores de fora dele

```
// parâmetros inout
```

```
func funcao(parametro: inout Int) {  
    parametro = 10  
}
```

```
var valor = 1  
funcao(parametro: &valor)  
valor // agora é 10
```



- se quisermos mudar o valor de um parâmetro
- e que essa mudança reflita fora da função
- usamos o modificador inout
 - passamos o parametro com um & na frente

```
// ordenação

var array = [2, 1, 3, 5, 4]

array.sort()

array // [1, 2, 3, 4, 5]
```



- sort por padrão
 - do menor pro maior

```
// ordenação com closure
```

```
var array = [2, 1, 3, 5, 4]
```

```
array.sort(by: { $0 > $1 })
```

```
array // [5, 4, 3, 2, 1]
```



- sort ascendente
 - recebendo um closure
 - que retorne um bool

```
// ordenação com operador
```

```
var array = [2, 1, 3, 5, 4]
```

```
array.sort(by: >)
```

```
array // [5, 4, 3, 2, 1]
```



- sort ascendente
 - recebendo um closure
 - que retorne um bool

```
// ordenação - sorted

var array = [2, 1, 3, 5, 4]

var sortedArray = array.sorted()
sortedArray // [1, 2, 3, 4, 5]

array          // [2, 1, 3, 5, 4]
```



- não faz o sort in-place
- retorna um novo array ordenado
- não modifica o original
- introsort $O(n \log n)$

```
// String.CharacterView

var string = "abc😱"

for char in string.characters {
    print(char) // a
    // b
    // c
    // 😱
}
```



```
// String.CharacterView
```

String



```
var string = "abc"
```

```
for char in string.characters {  
    print(char)  
}
```

citiz

```
// String.CharacterView  
var string = "abc"  
for char in string.characters {  
    print(char)  
}
```

cit

- A view of a string's contents as a collection of characters

```
// String.CharacterView  
var string = "abc"  
for char in string.characters {  
    print(char)  
}
```

citi

// Enumerações



```
// enum
```

Enumerações são uma forma 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.



- Novo tipo
 - possíveis valores relacionados
- tem métodos e propriedades

```
// enum - simples
```

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



- novo tipo Família (Maiúscula)
 - possíveis valores (Minúscula)

// 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 = ""
```



Atribuindo a uma variável
um valor do tipo família

// 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 - Identificador
(tipo) (valor)



```
// enum - switch

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



membro é do tipo Familia
omitimos o tipo nos casos
não precisa do default (já é exaustivo)

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

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



Redefinir os tipos de swift
fazer eles em português

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

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



```
numero é do tipo NovoTipo
numero tem valor de .inteiro
com valor associado de 8
```

```
// 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"
```



- cada case do switch
 - considera um case (valor possível) do enum
 - o valor associado vai ser armazenado na constante valor
 - e adicionada à mensagem

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

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



- outra forma de comparar o enum
 - e acessar valor associado
 - casamento de padrão
 - if case let

```
// enum - raw value

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



O enum com valores padrões (raw value)
de um mesmo tipo
implícito no terceiro

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

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

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



acessando o raw value

o terceiro implicitamente atribuído a 3

```
// enum - raw value

switch posicao {

    case .primeiro:
        mensagem = "número \u201c" + posicao.rawValue + "\u201d"

    default:
        break
}
```



podemos comparar pelo case do enum
e acessar o rawvalue no corpo

```
// enum - raw value

switch posicao.rawValue {
    case 1:
        mensagem = "número 1"

    default:
        break
}
```



podemos comparar diretamente pelo rawValue
ignorando os cases do enum

```
// enum - raw value

switch posicao {

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

    default:
        break
}
```



podemos verificar se o raw value respeita uma condição

// 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 uma **mensagem de resultado da partida** (referente ao primeiro jogador)

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



adaptado de: <https://www.weheartswift.com/tuples-enums/>

// 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 apolo235 = Endereco(cidade: "Recife",
                        logradouro: "Rua do Apolo",
                        numero: 235)

let joao = Pessoa(nome: "João",
                  endereco: apolo235,
                  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
}
```



Trecho de: Apple Inc. “The Swift Programming Language (Swift 3.1)”. iBooks.

```
// propriedades computadas
```

```
struct Rect {  
    var origin = Point()  
    var size = Size()  
    var center: Point {  
        get {  
            let (x, y) = origin  
            let (w, h) = size  
            return Point(x + w / 2, y + h / 2)  
        }  
        set {  
            center = value  
        }  
    }  
}
```

Trecho de: Apple Inc. “The Swift Programming Language (Swift 3.1)”. iBooks.



```
// 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)  
        }  
    }  
}
```

Trecho de: Apple Inc. “The Swift Programming Language (Swift 3.1)”. iBooks.



```
// 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)  
        }  
    }  
}
```



Trecho de: Apple Inc. “The Swift Programming Language (Swift 3.1)”. iBooks.

// 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**



<https://goo.gl/Q9DoXP>

// por cópia (*value type*)

```
var copiaApolo = apolo235  
apolo235.cidade = "Olinda"  
apolo235.cidade      // tem valor de "Olinda"  
copiaApolo.cidade    // tem valor de "Recife"
```



// por referência (reference type)

```
var referenciaJoao = joao  
joao.telefone = "99999999"  
  
joao.telefone // tem valor de "99999999"  
referenciaJoao.telefone // tem valor de "99999999"
```



// 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



<https://stackoverflow.com/questions/24232799/why-choose-struct-over-class>

```
// exemplos de structs
```

- O **tamanho** de uma **forma geométrica**:

- propriedades largura e altura
 - Double

- **Pontos** num sistema de coordenadas **3D**:

- x, y, z
 - Double



<https://stackoverflow.com/questions/24232799/why-choose-struct-over-class>

// 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

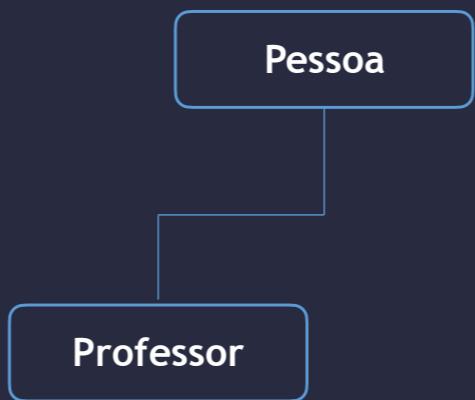


// herança

Pessoa



// herança



// herança



// herança

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



// herança

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



// 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."  
  
}
```



modifica o comportamento de um método da superclasse
Override

```
// sobrescrevendo método

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

    //sobrescrevendo método
    override func apresentar() -> String {
        return "Oi, meu nome é \(self.titulo) \(self.nome)"
    }
}
```



modifica o comportamento de um método da superclasse
Override

```
// sobrescrevendo método

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

    //sobrescrevendo método
    override func apresentar() -> String {
        return "Oi, meu nome é \(self.titulo) \(self.nome)"
    }
}

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



modifica o comportamento de um método da superclasse
Override

```
// adicionando propriedade  
  
class Professor: Pessoa {  
    //...  
  
}
```



```
// adicionando propriedade

class Professor: Pessoa {
    //...
    // nova propriedade
    var universidade: String

}
```



```
// 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")
```



```
// 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**



adaptado de: <https://goo.gl/bQicMH>

// Casting de Tipos

cit

// 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)
```



// checagem de tipos (*is*)

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

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



// 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): Ms."
```



// 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.



adaptado de: <https://www.weheartswift.com/swift-classes-part-2/>

// Aula 04



// Protocolos (//)



// extensões

Extensões **adicionam nova funcionalidade** a uma classe, estrutura, enumeração, ou protocolo existente.

É possível também extender tipos dos quais não temos acesso ao código fonte (*retroactive modeling*)



<https://goo.gl/D8Uf2Z>

// extensões

- Adicionar **propriedades computadas**
- Definir **métodos**
- Prover novos **construtores**
- Definir **tipos aninhados**
- Conformar com **protocolo**



adaptado de: <https://goo.gl/D8Uf2Z>

// extensões

Extensões podem adicionar novas funcionalidades a um tipo, mas **não podem sobrescrever** (*override*) funcionalidades existentes.



<https://goo.gl/D8Uf2Z>

```
// nova classe

class Pessoa {
    let nome: String
    var anoNascimento: Int
    var telefone: String?

    init(nome: String, anoNascimento: Int, telefone: String? = nil) {
        self.nome = nome
        self.anoNascimento = anoNascimento
        self.telefone = telefone
    }
}
```



- Default value
 - telefone

```
// default value

var joao = Pessoa(nome: "João",
                  anoNascimento: 1996)

var marina = Pessoa(nome: "Marina",
                     anoNascimento: 1990,
                     telefone: "94382712")

joao.telefone // nil
marina.telefone // "94382712"
```



```
// estendendo classe

extension Pessoa {
    var idade: Int {

        let date = Date()
        let calendar = Calendar.current
        let year = calendar.component(.year, from: date)

        return year - self.anoNascimento
    }
}
```



```
// acessando propriedade da extensão
```

```
joao.idade // 21  
marina.idade // 27
```



```
// subclasse
```

```
class Professor: Pessoa {  
    var tituloP = "Prof."  
}  
  
var jose = Professor(nome: "José", anoNascimento: 1980)  
jose.tituloP // "Prof."
```



```
// novo protocolo

protocol Mestre {
    var tituloM: String { get }
}
```



```
// estendendo protocolo

protocol Mestre {
    var tituloM: String { get }
}

extension Mestre {
    var tituloM: String {
        return "Ms."
    }
}
```



```
// estendendo classe para aderir a protocolo
```

```
extension Professor: Mestre { }
```

```
jose.tituloM // "Ms."
```



// MVC



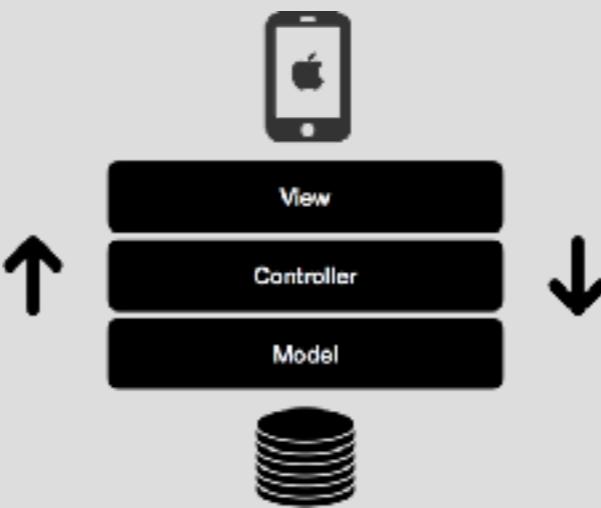
// MVC

Para desenvolver um aplicativo, a arquitetura mais comumente utilizada, e recomendada pela própria Apple é o padrão **Model-View-Controller** (MVC).

Nessa arquitetura temos **três camadas** principais com responsabilidades definidas para nos ajudar a ter um código organizado e menos propenso a erros e inconsistências.



// MVC



citi

// MVC



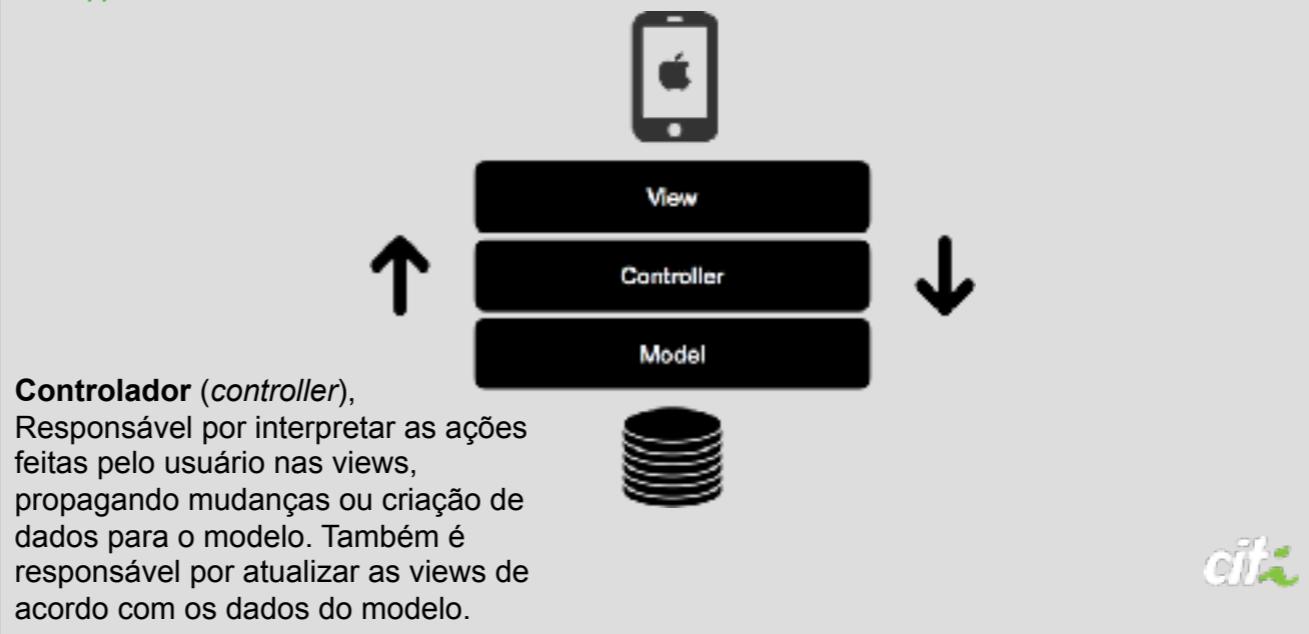
Modelo (model),
responsável por manter os dados,
objetos, comunicação com a rede,
e regras de negócio da aplicação.



// MVC



// MVC



// UIKit

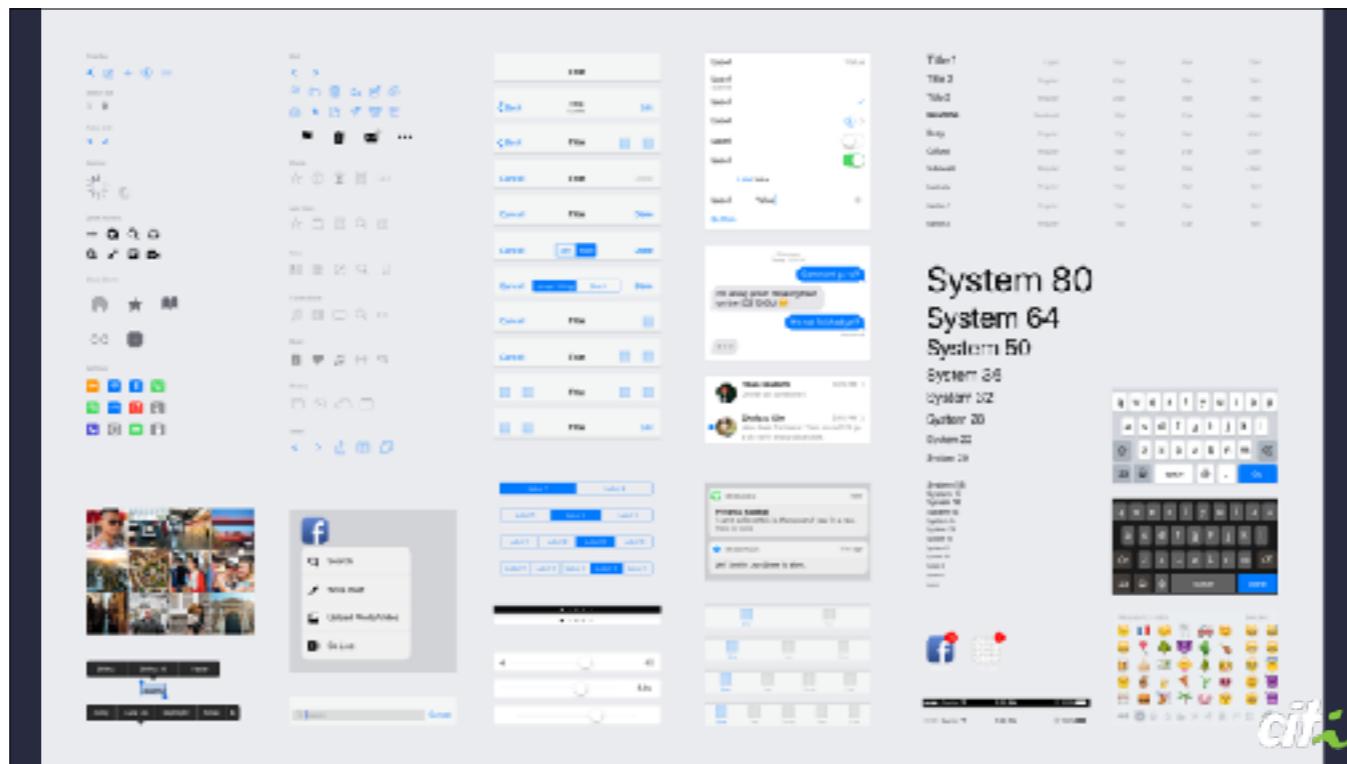


// UIKit

Framework para construir e gerenciar
interfaces gráficas orientadas a
eventos, em suas aplicações iOS.



adaptado de: <https://developer.apple.com/documentation/uikit>



// Views e Controls

Apresentam o seu **conteúdo** na tela, em maneiras específicas e definem **interações** permitidas com este conteúdo.



adaptado de: https://developer.apple.com/documentation/uikit/views_and_controls

// UIView

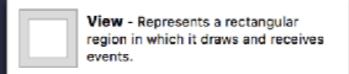


View - Represents a rectangular region in which it draws and receives events.

cif

adaptado de: <https://developer.apple.com/documentation/uikit/uiview>

//



View - Represents a rectangular region in which it draws and receives events.

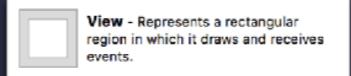
Desenho e Animação

- Views desenham conteúdo em sua área retangular usando UIKit ou Core Graphics.
- Algumas propriedades de uma view podem ser animadas.



adaptado de: <https://developer.apple.com/documentation/uikit/uiview>

//



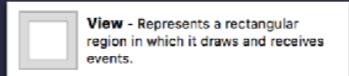
Layout e subviews

- Views podem ter várias subviews.
- Views podem ajustar posição e tamanho de suas subviews.
- Podem utilizar Auto-layout para fazer o posicionamento responsivo



adaptado de: <https://developer.apple.com/documentation/uikit/uiview>

//



Event handling

- Views são subclasses de UIResponder podendo responder a toques e outros eventos.
- Views podem adicionar reconheceres de gestos para lidar com gestos comuns.



adaptado de: <https://developer.apple.com/documentation/uikit/uiview>

```
// UILabel
```

Label Label - A variably sized amount of static text.



// **Label** Label - A variably sized amount of static text.

```
class UILabel : UIView {  
  
    var text: String? // default is nil  
    // ...  
}
```



```
// UIButton
```

Button **Button** - Intercepts touch events and
sends an action message to a target
object when it's tapped.



// **Button** - Intercepts touch events and
Button sends an action message to a target
object when it's tapped.

```
class UIButton : UIControl{

    func setTitle(_ title: String?,
                  for state: UIControlState)
        // default is nil. title is assumed to be single line
        // ...
}
```



// StoryBoard



// storyBoard

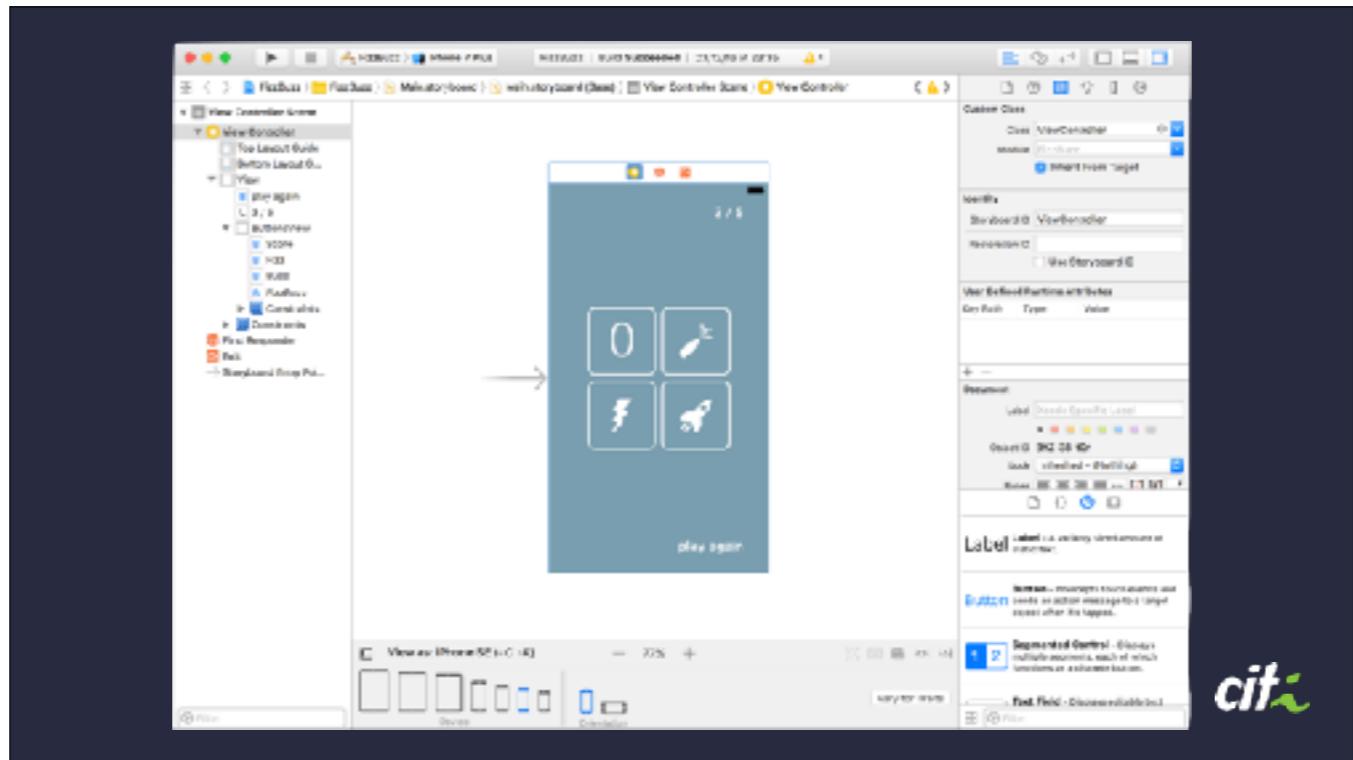
Interface gráfica para montar suas GUIs



// storyBoard

Uma forma mais fácil de dispor as views
e suas subviews, customizá-las, e ver o
resultado enquanto faz.





// ViewController



// viewControllers

View controllers são os fundamentais para a estrutura interna de um app.

Todo app tem pelo menos um view controller, ou vários.

Cada viewController gerencia uma porção da GUI, além das interações entre a interface e os dados.



adaptado de: adaptado de: https://developer.apple.com/documentation/uikit/views_and_controls

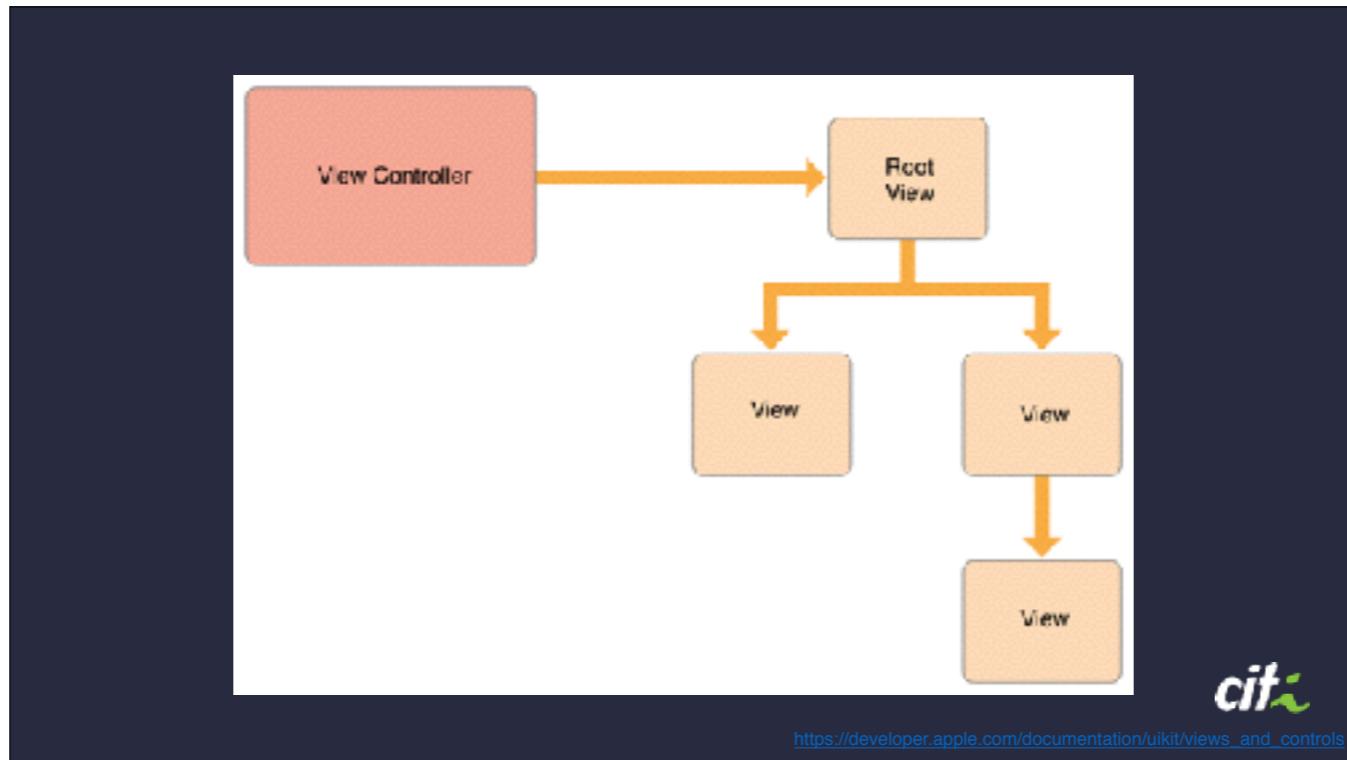
```
// viewControllers
```

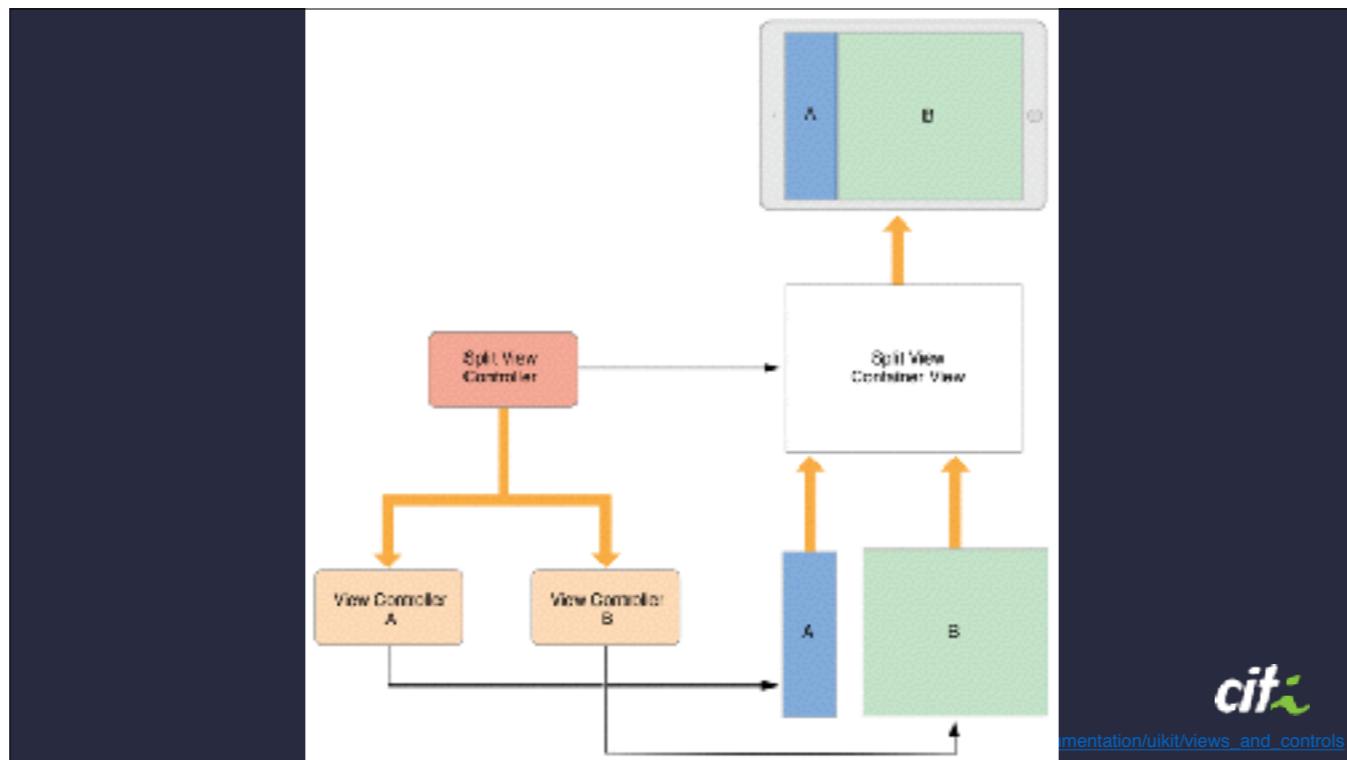
A classe `UIViewController` define métodos e propriedades para gerenciar as views, tratar eventos, transicionar entre view controllers, e coordenar com outras partes do app.

Fazemos subclasses de `UIViewController` para implementar o comportamento customizado da nossa aplicação.



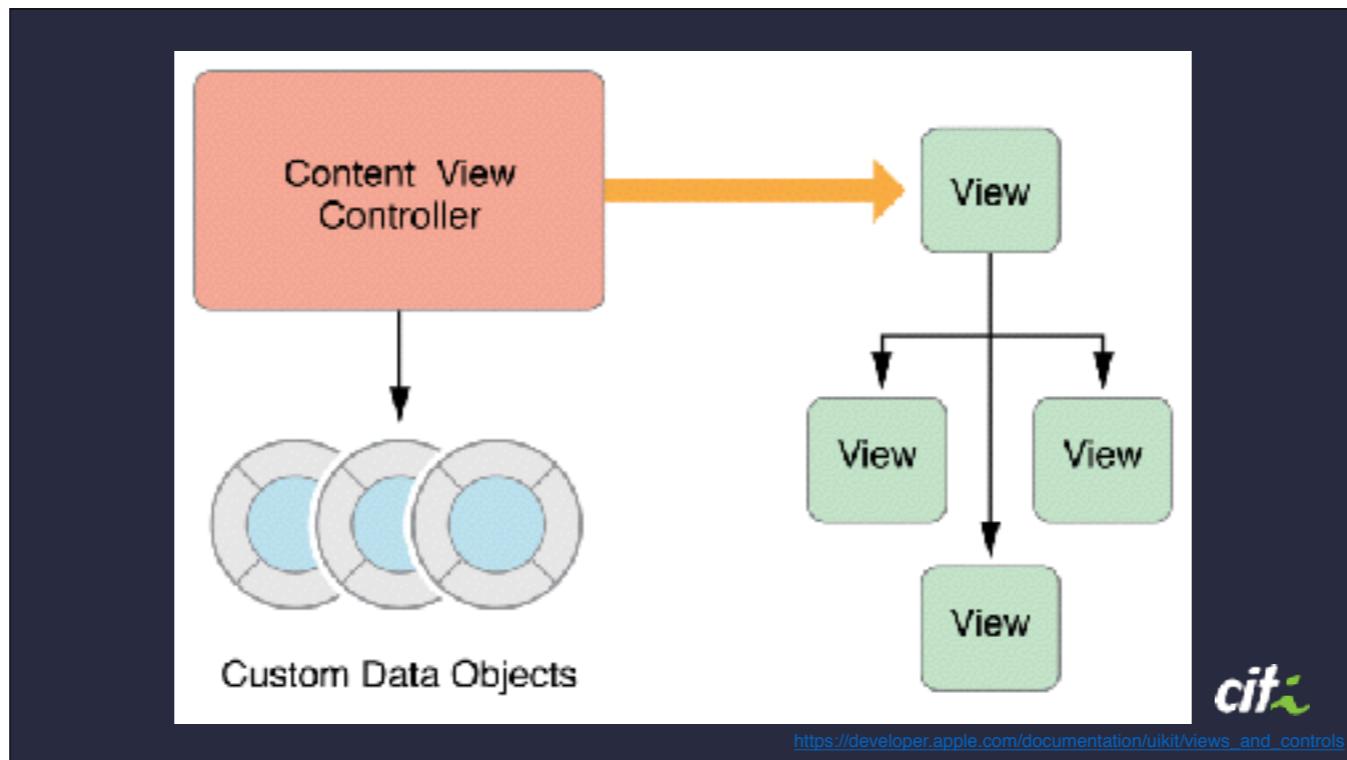
adaptado de: adaptado de: https://developer.apple.com/documentation/uikit/views_and_controls





citi

[Implementation/uikit/views_and_controls](#)



https://developer.apple.com/documentation/uikit/views_and_controls



// viewControllers

View controllers tem um ciclo de vida, com métodos que são chamados sempre que algum marco acontece.





citi

```
// viewControllers

import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view,
        // typically from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
}
```



// ViewController/StoryBoard



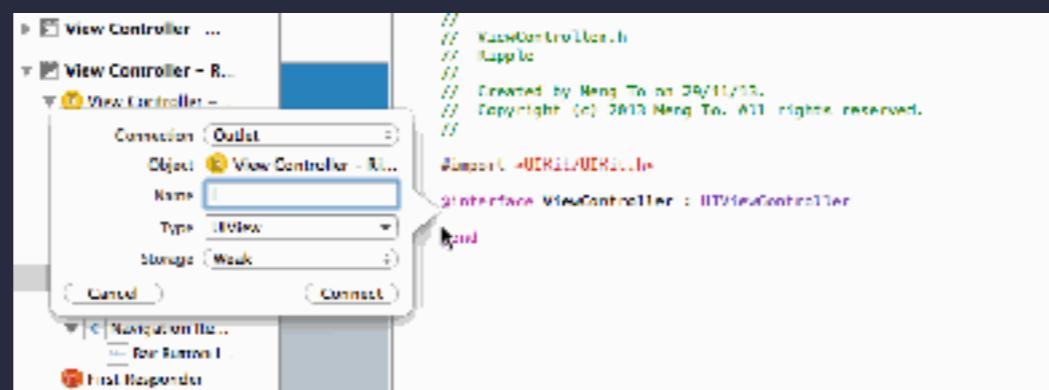
// problemática

Como dizer que um ViewController é responsável por uma view?

Como referenciar views adicionadas ao StoryBoard pelo código (ex: mudar conteúdo de um label, ação ao apertar um botão, ...)?

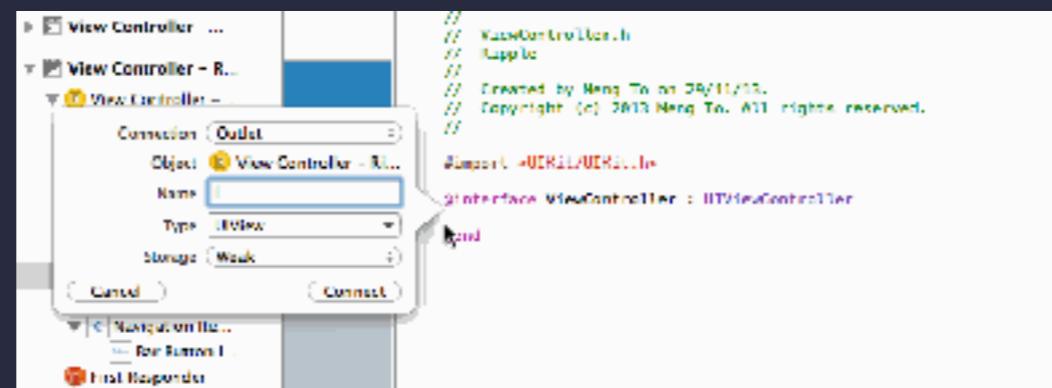


// IBOutlets & IBActions



citi

// IBOutlets & IBActions



citi

```
// IBOutlet
```

Dá um nome a uma view do StoryBoard para que possa ser acessada por código no ViewController



```
// IBAction
```

Configura funções a serem executadas quando acontece
alguma ação.



// Exercício



// Exercício 09

Button and Label

1. No **StoryBoard**:

1. adicione um botão
2. adicione uma label

2. No **view controller**:

- 2.1 ao **abrir o app**, mude o texto da label para a **data atual**
- 2.2 ao **apertar o botão**, mude o **texto da label** para a quantidade de vezes que o botão já foi apertado



// Delegate



// Delegate

Delegação é um padrão de projeto que permite que uma classe ou estrutura, passe (**delege**) parte de suas **responsabilidades** para uma instância de um outro tipo



// Delegate

1. Definição do protocolo
2. Implementar protocolo em uma classe
3. adicionar propriedade delegate
4. chamar métodos do delegate



```
// Delegate
```

```
protocol DiceGame {  
    var dice: Dice { get }  
    func play()  
}
```



```
// Delegate

protocol DiceGameDelegate {
    func gameDidStart(_ game: DiceGame)
    func game(_ game: DiceGame,
didStartNewTurnWithDiceRoll diceRoll: Int)
    func gameDidEnd(_ game: DiceGame)
}
```



```
// Delegate

class SnakesAndLadders: DiceGame {
    // ...
    var delegate: DiceGameDelegate?
}
```



```
// Delegate
class DiceGameTracker: DiceGameDelegate {
    var number0fTurns = 0
    func gameDidStart(_ game: DiceGame) {
        number0fTurns = 0
        if game is SnakesAndLadders {
            print("Started a new game of Snakes and Ladders")
        }
        print("The game is using a \(game.dice.sides)-sided dice")
        "(game.dice.sides)-sided dice"
    }
    func game(_ game: DiceGame, didStartNewTurnWithDiceRoll diceRoll:
    Int) {
        number0fTurns += 1
        print("Rolled a \(diceRoll)")
    }
    func gameDidEnd(_ game: DiceGame) {
        print("The game lasted for \(number0fTurns) turns")
    }
}
```



```
// Delegate
```

```
let tracker = DiceGameTracker()  
let game = SnakesAndLadders()  
game.delegate = tracker  
game.play()
```



// AppDelegate



// Exercício



// Exercício 10

Text Field

Faça um App com um botão e um campo de texto.

Inicie o **botão** como **desabilitado**.

Quando tiver texto escrito no campo de texto, **habilite** o **botão**.

Ao **apertar** o **botão**:

- se o usuário tiver digitado uma cor válida, modifique a **cor** do **background** da **view**.
- caso contrário, **notifique** o usuário que escreveu uma **cor inválida**

UIColor e shouldChangeCharactersIn range 

// Aula 05



// Recapitulando



// Delegate



// Delegate

1. Definição do protocolo
2. Implementar protocolo em uma classe
3. adicionar propriedade delegate
4. chamar métodos do delegate



```
// delegate

import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var colorTextField: UITextField!

    @IBOutlet weak var colorBtn: UIButton!

    @IBAction func changeColor(_ sender: Any) {
        var novaCor: UIColor
        // ...
    }
}
```



```
// delegate  
extension ViewController: UITextFieldDelegate {  
  
}  
}
```



```
// delegate

extension ViewController: UITextFieldDelegate {

    func textField(_ textField: UITextField,
                  shouldChangeCharactersIn range: NSRange,
                  replacementString string: String) -> Bool {

        guard let texto = textField.text else {
            return true
        }

        if texto != ""{
            self.colorBtn.isEnabled = true
        }
        return true
    }
}
```



```
// delegate

class ViewController: UIViewController {
    @IBOutlet weak var colorTextField: UITextField!

    override func viewDidLoad() {
        super.viewDidLoad()

    }
}
```



```
// delegate

class ViewController: UIViewController {
    @IBOutlet weak var colorTextField: UITextField!

    override func viewDidLoad() {
        super.viewDidLoad()

        self.colorTextField.delegate = self
    }
}
```



```
// delegate
```

1. UITextFieldDelegate
2. ViewController: UITextFieldDelegate
3. self.colorTextField.delegate = self
4. chamar métodos do delegate (responsabilidade do Text Field)



```
// delegate

public protocol UITextFieldDelegate : NSObjectProtocol {

    @available(iOS 2.0, *)
    optional public func textFieldShouldBeginEditing(_ textField: UITextField) -> Bool // return NO to disallow editing.

    @available(iOS 2.0, *)
    optional public func textFieldDidBeginEditing(_ textField: UITextField) // became first responder

    @available(iOS 2.0, *)
    optional public func textFieldShouldEndEditing(_ textField: UITextField) -> Bool // return YES to allow editing to stop and to
    resign first responder status. NO to disallow the editing session to end

    @available(iOS 2.0, *)
    optional public func textFieldDidEndEditing(_ textField: UITextField) // may be called if forced even if shouldEndEditing
    returns NO (e.g. view removed from window) or endEditing:YES called

    @available(iOS 10.0, *)
    optional public func textFieldDidEndEditing(_ textField: UITextField, reason: UITextFieldDidEndEditingReason) // if
    implemented, called in place of textFieldDidEndEditing:

    // ...
}
```



// Navegação

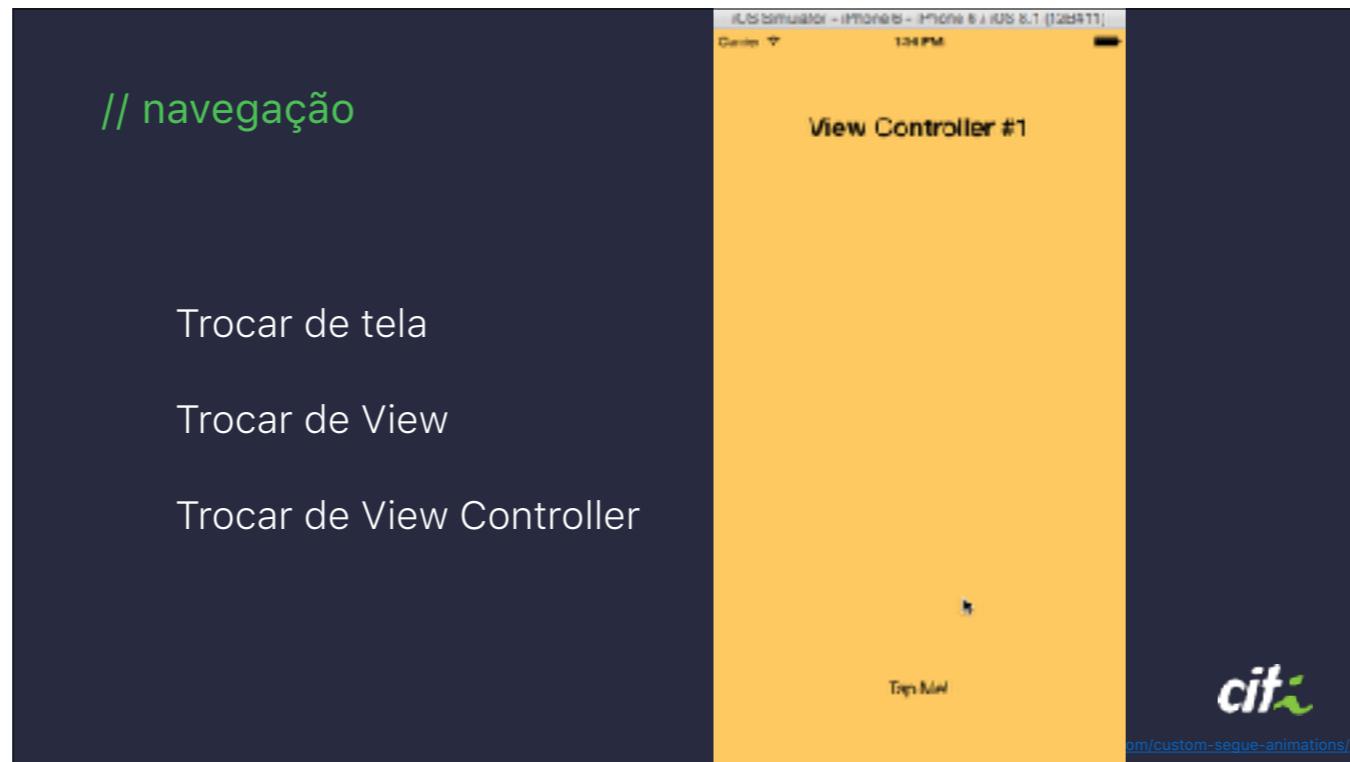


// navegação

Trocar de tela

Trocar de View

Trocar de View Controller

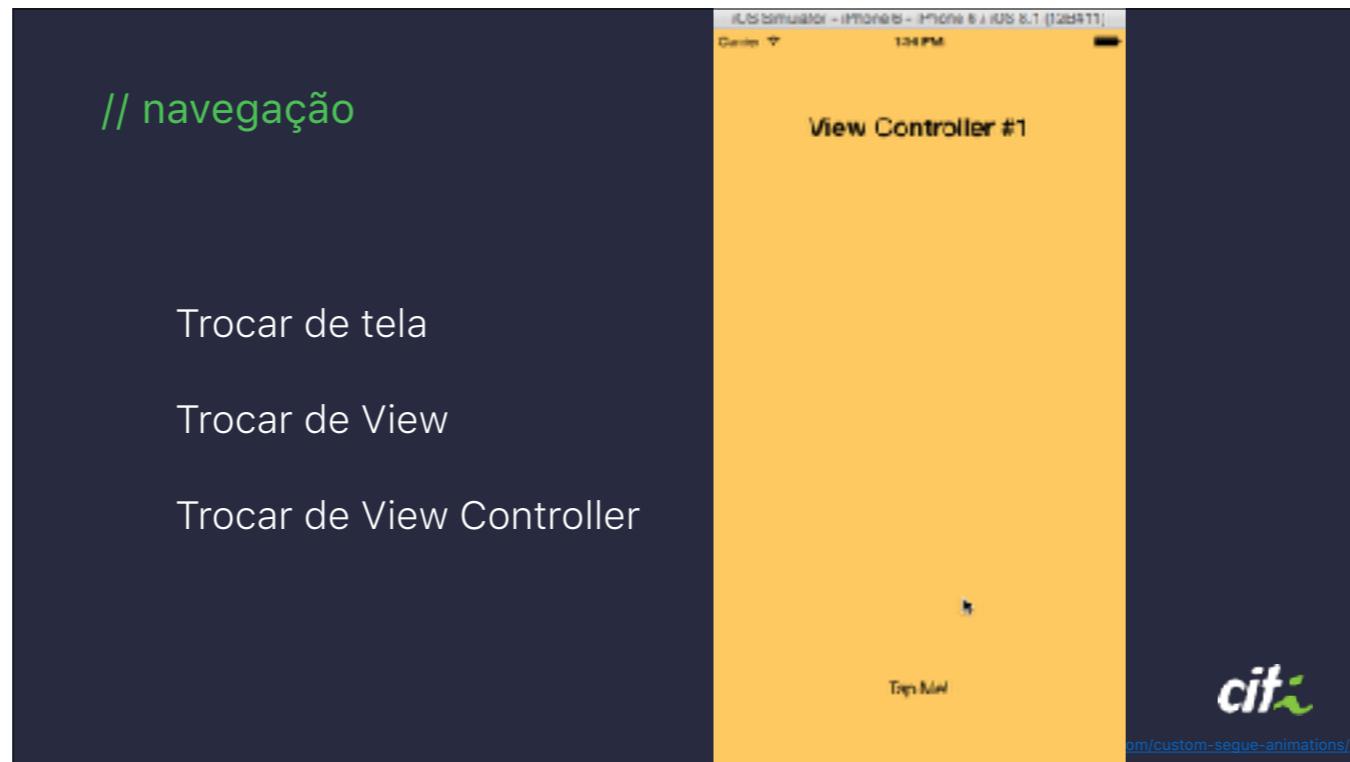


// navegação

Trocar de tela

Trocar de View

Trocar de View Controller



// Segues

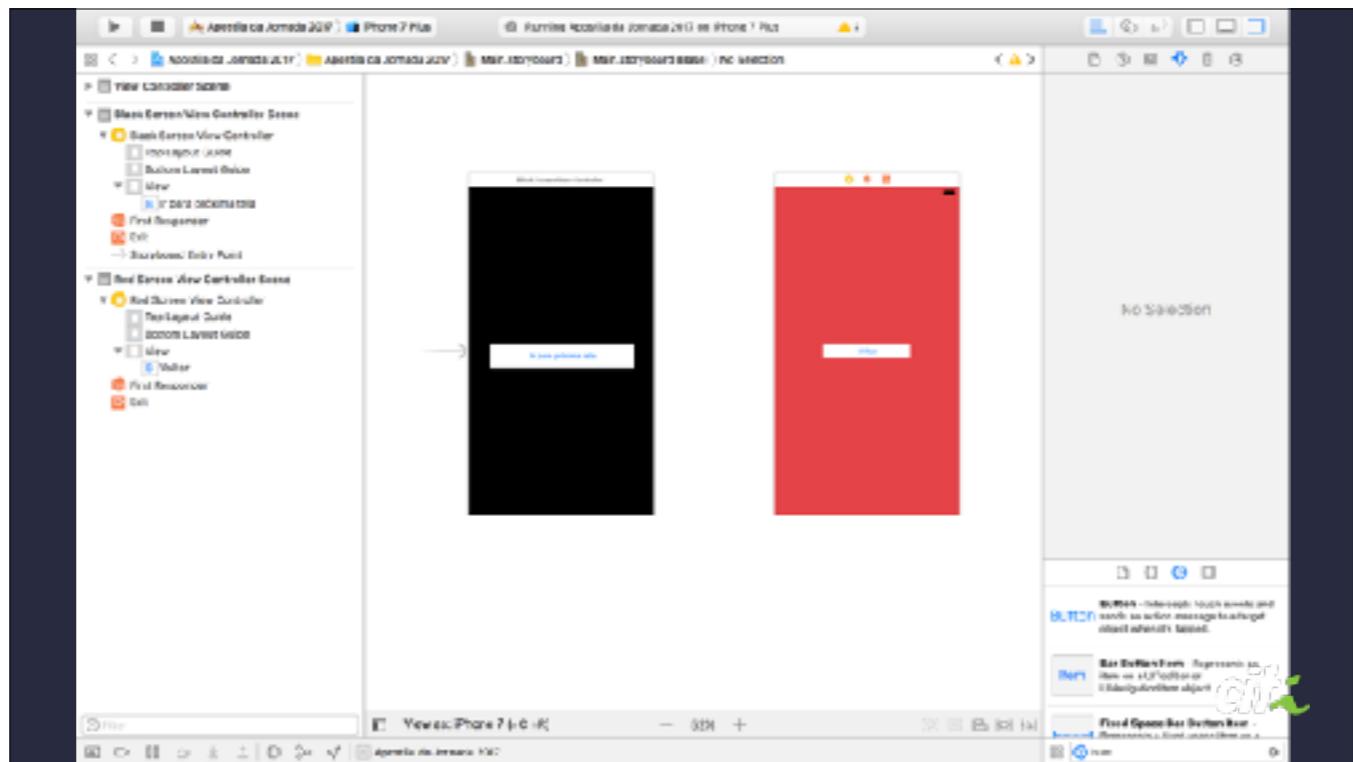


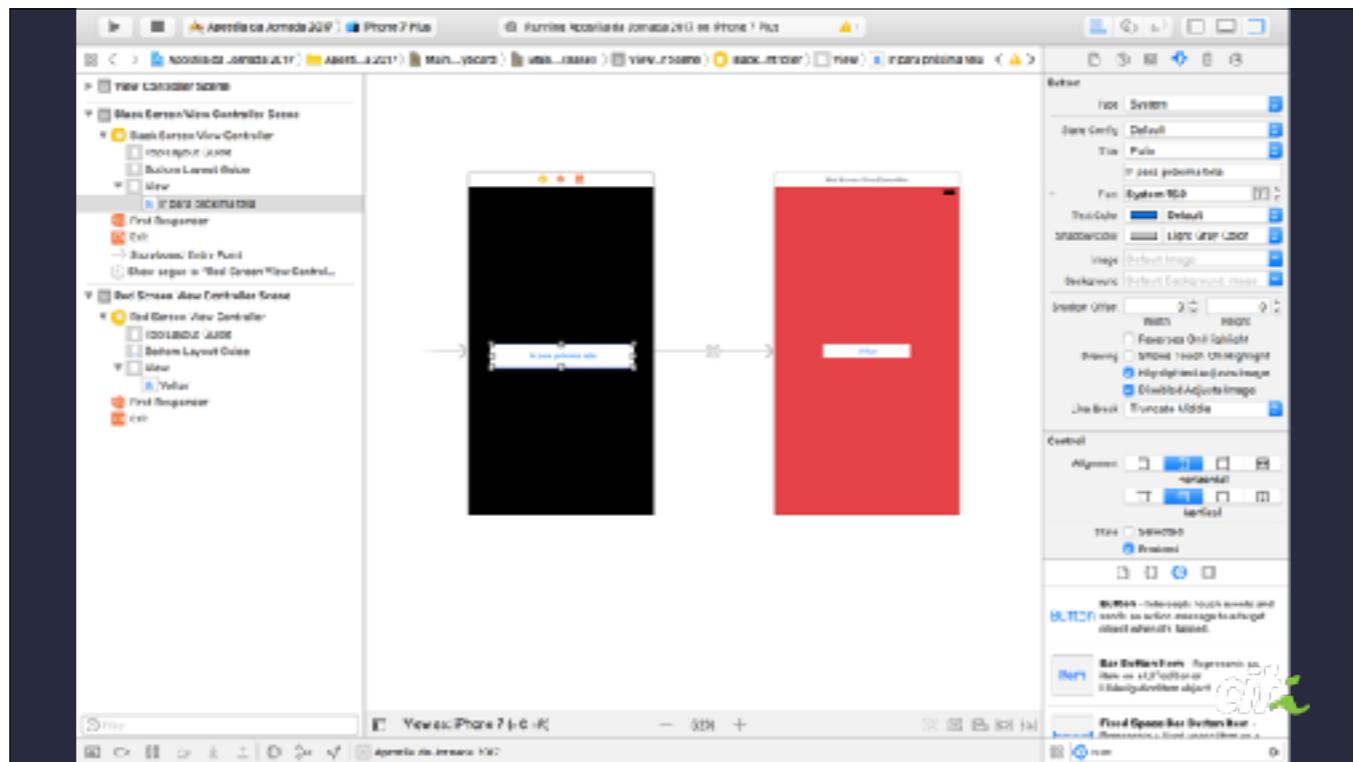
// Segues

- Definem o fluxo de **navegação** do app
- **Transição** entre ViewControllers do StoryBoard
- **Início**: button, table row, or gesture recognizer
- **Destino**: ViewController a ser mostrado



<https://goo.gl/bRnvzY>





// Segues

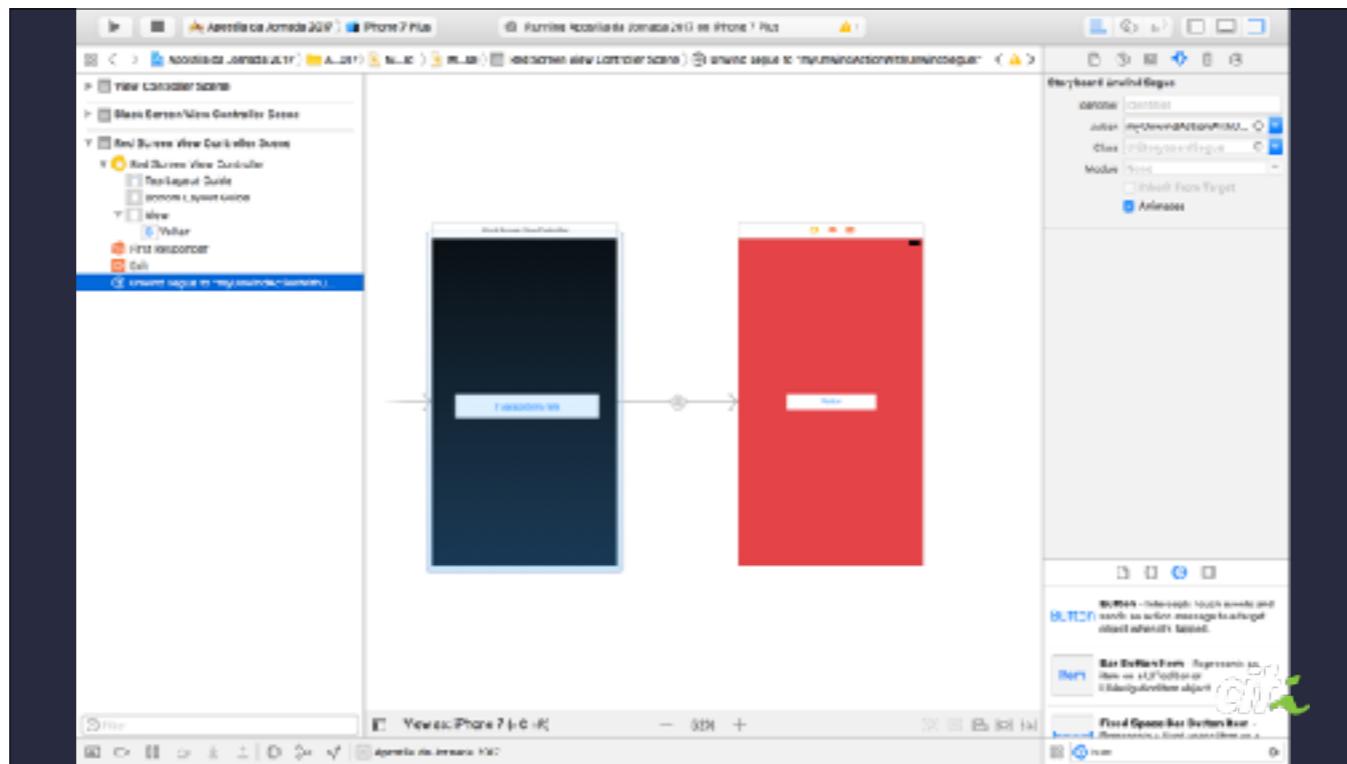
- Apresentação Modal



// Segues

- Apresentação Modal





You must define an unwind action method in one of your view controllers before trying to create the corresponding unwind segue in Interface Builder. The presence of that method is required and tells Interface Builder that there is a valid target for the unwind segue.

Use the implementation of your unwind action method to perform any tasks that are specific to your app. You do not need to dismiss any view controllers involved in the segue yourself; UIKit does that for you. Instead, use the segue object to fetch the view controller being dismissed so that you can retrieve data from it. You can also use the unwind action to update the current view controller before the unwind segue finishes.

// Segues

- Unwind

```
@IBAction func myUnwindAction(unwindSegue: UIStoryboardSegue)
```



You must define an unwind action method in one of your view controllers before trying to create the corresponding unwind segue in Interface Builder. The presence of that method is required and tells Interface Builder that there is a valid target for the unwind segue.

Use the implementation of your unwind action method to perform any tasks that are specific to your app. You do not need to dismiss any view controllers involved in the segue yourself; UIKit does that for you. Instead, use the segue object to fetch the view controller being dismissed so that you can retrieve data from it. You can also use the unwind action to update the current view controller before the unwind segue finishes.

// Segues

- Unwind

```
@IBAction func myUnwindAction(unwindSegue: UIStoryboardSegue)
```



You must define an unwind action method in one of your view controllers before trying to create the corresponding unwind segue in Interface Builder. The presence of that method is required and tells Interface Builder that there is a valid target for the unwind segue.

Use the implementation of your unwind action method to perform any tasks that are specific to your app. You do not need to dismiss any view controllers involved in the segue yourself; UIKit does that for you. Instead, use the segue object to fetch the view controller being dismissed so that you can retrieve data from it. You can also use the unwind action to update the current view controller before the unwind segue finishes.

// Chamando por Código

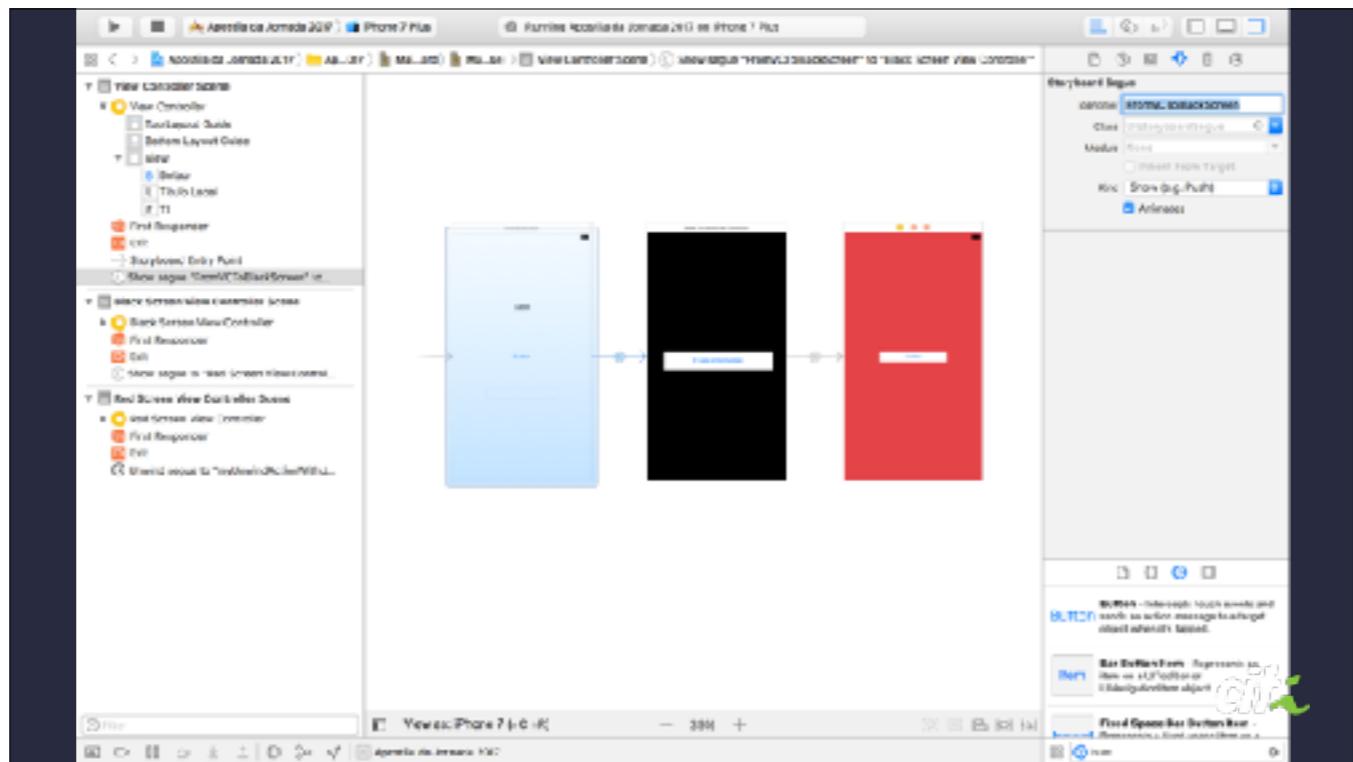


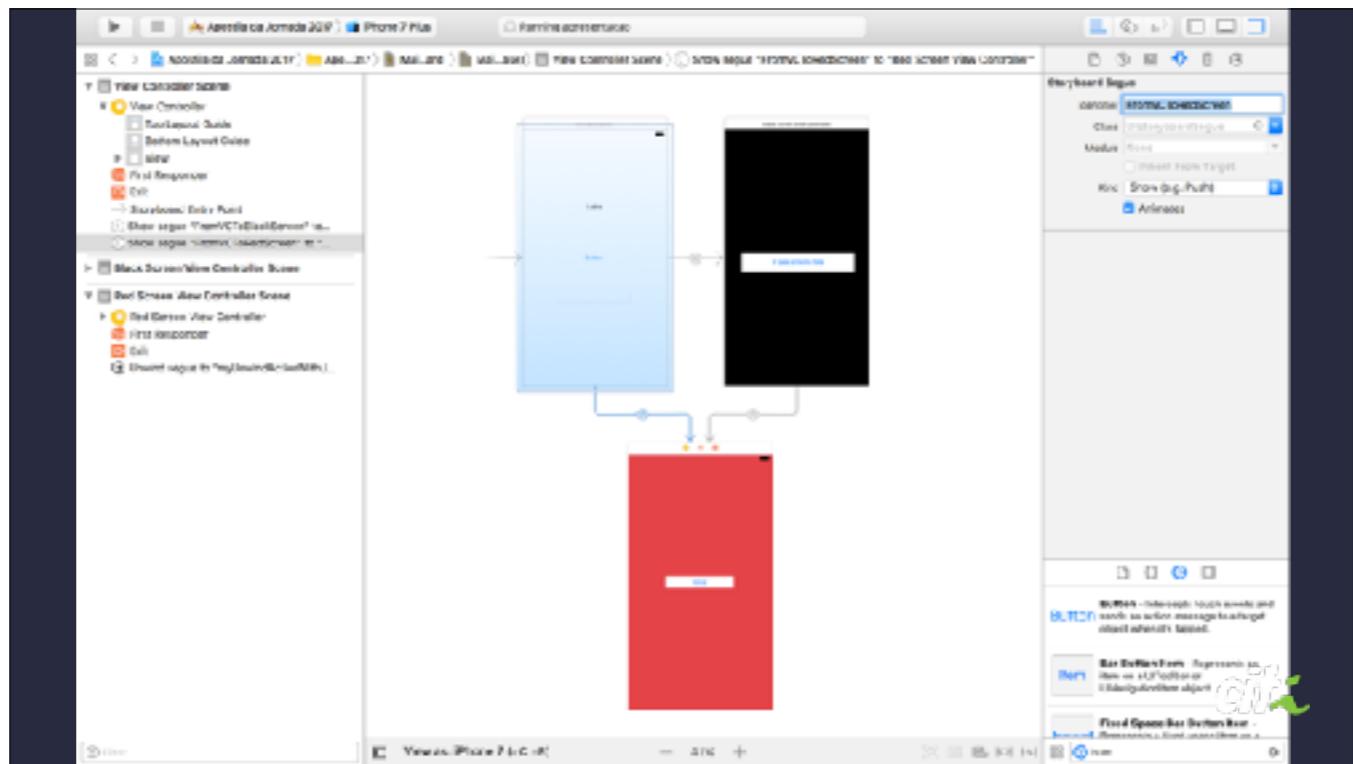
// Segues programáticas

Às vezes é necessário **chamar** uma segue pelo código



<https://goo.gl/bRnvzY>





// chamando segue



citi

// chamando segue



citi

```
// chamando segue
```

```
if tituloLabel.text == "Black" {  
    self.performSegue(withIdentifier: "FromVCToBlackScreen", sender: self)  
  
} else if tituloLabel.text == "Red" {  
    self.performSegue(withIdentifier: "FromVCToRedScreen", sender: self)  
}
```



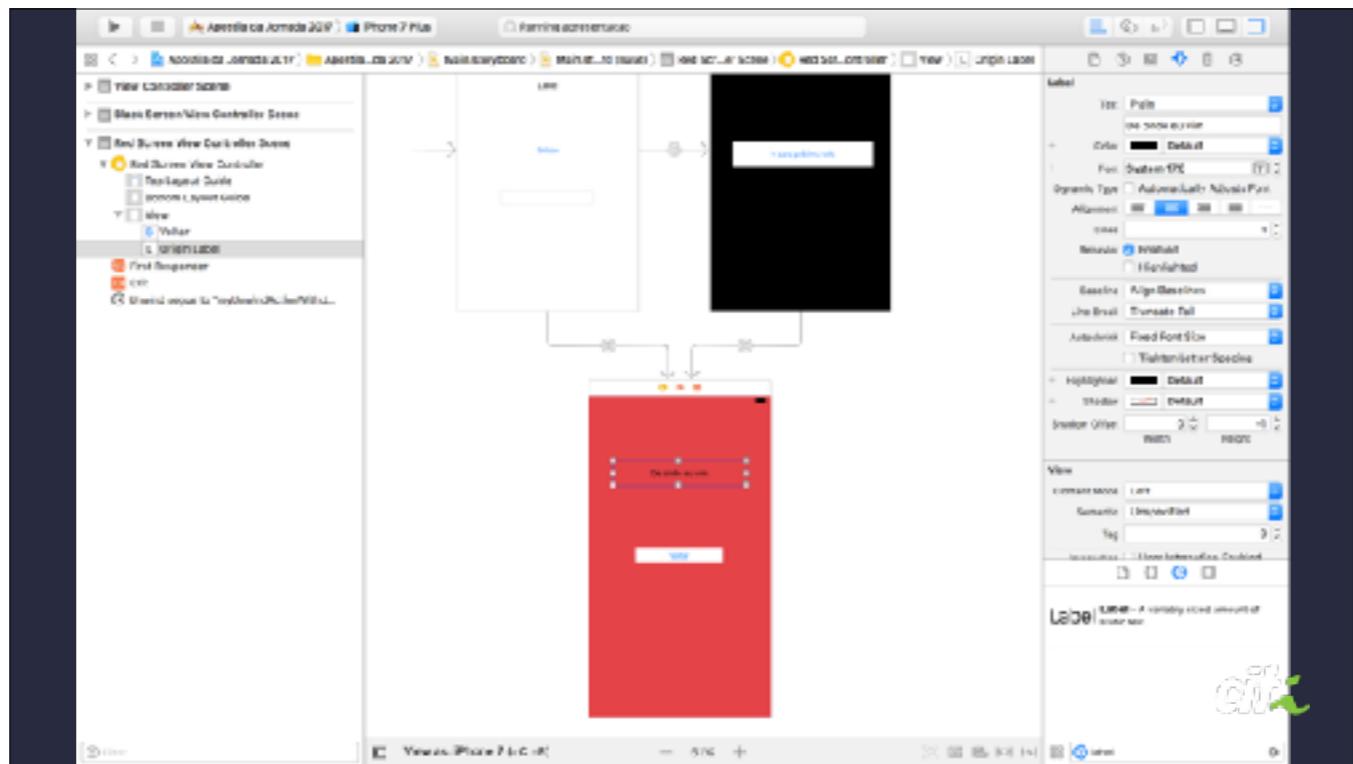
// Passando Dados

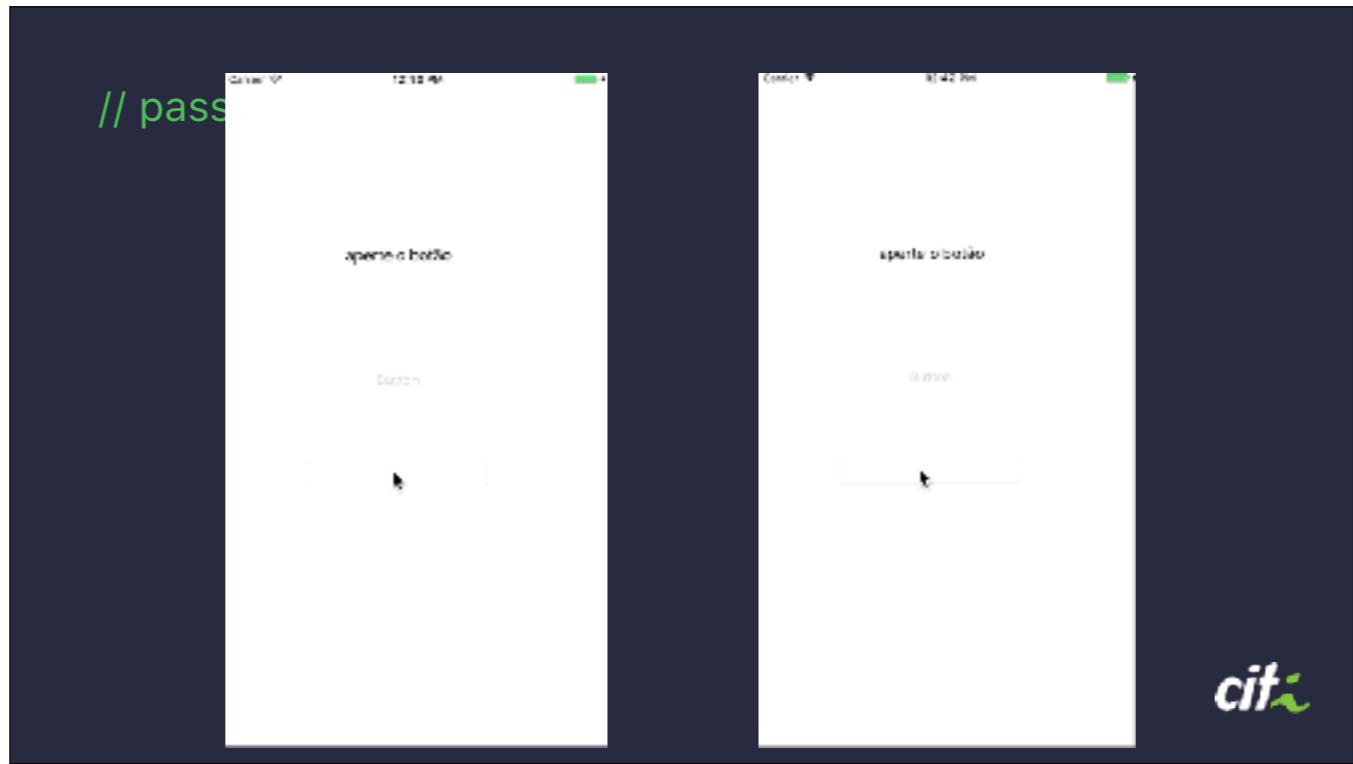


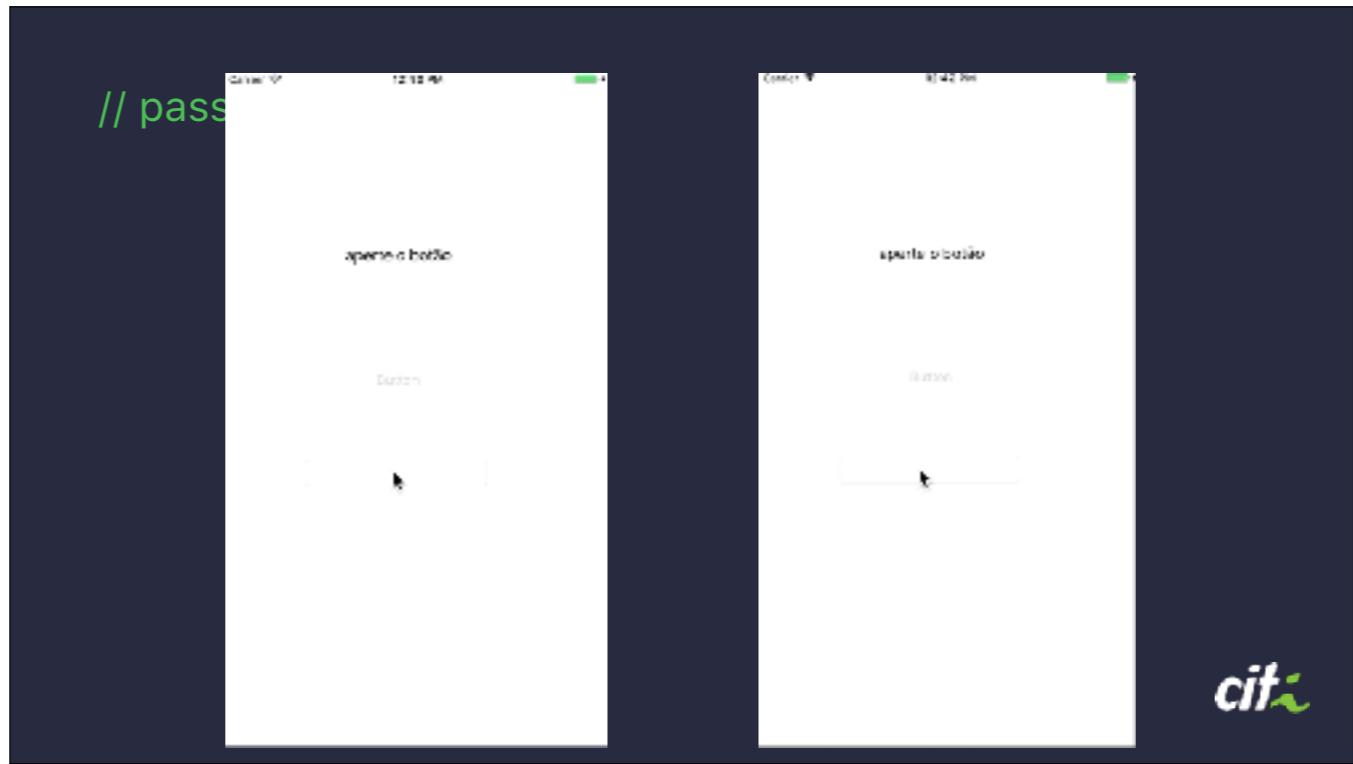
// passando dados

- Se precisarmos passar informações entre view controllers
- Podemos passá-las pela segue









```
// passando dados

class RedScreenViewController: UIViewController {
    var recievedData: String?

    @IBOutlet weak var originLabel: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()

        self.originLabel.text = self.recievedData
    }
}
```



// passando dados

No ViewController:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {  
    if let redVC = segue.destination as? RedScreenViewController {  
        redVC.recievedData = "Vim do VC"  
    }  
}
```

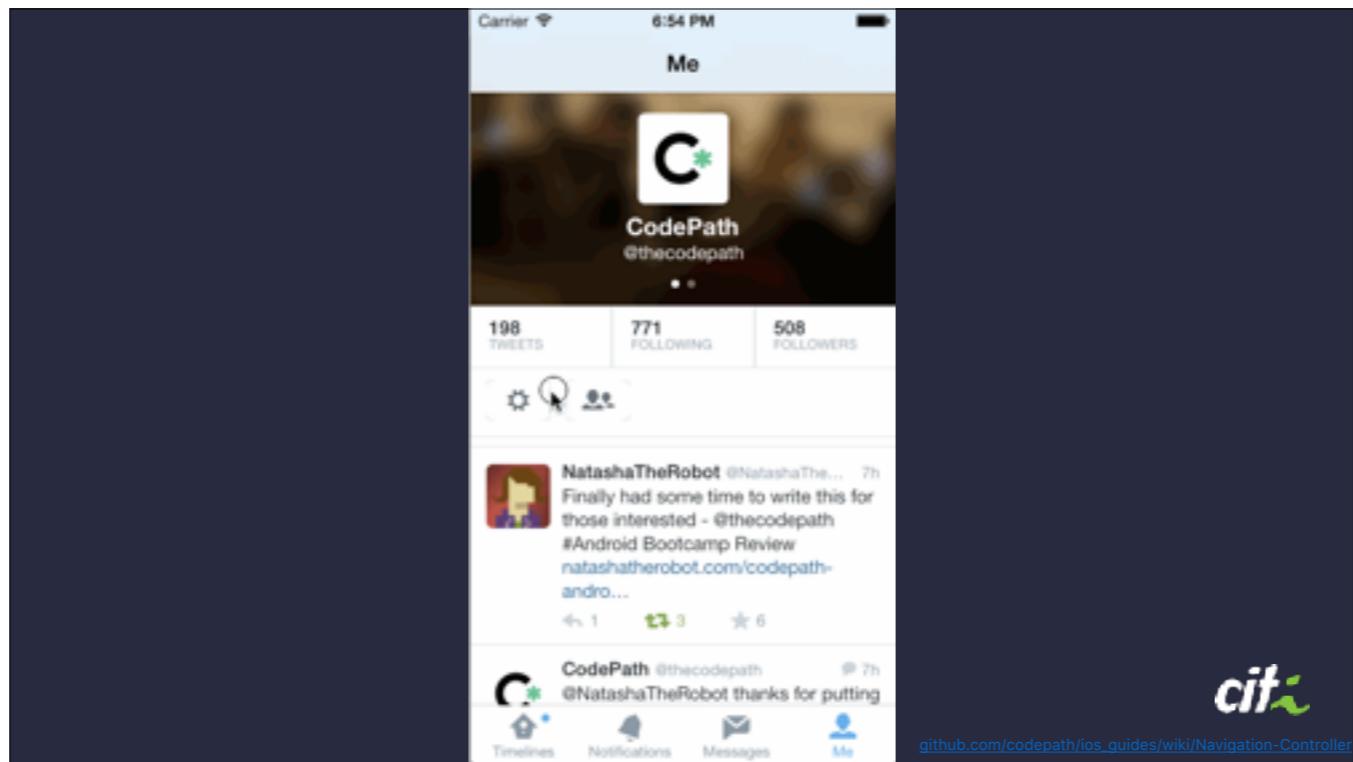
No BlackScreenViewController:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {  
    if let redVC = segue.destination as? RedScreenViewController {  
        redVC.recievedData = "Vim do BlackVC"  
    }  
}
```

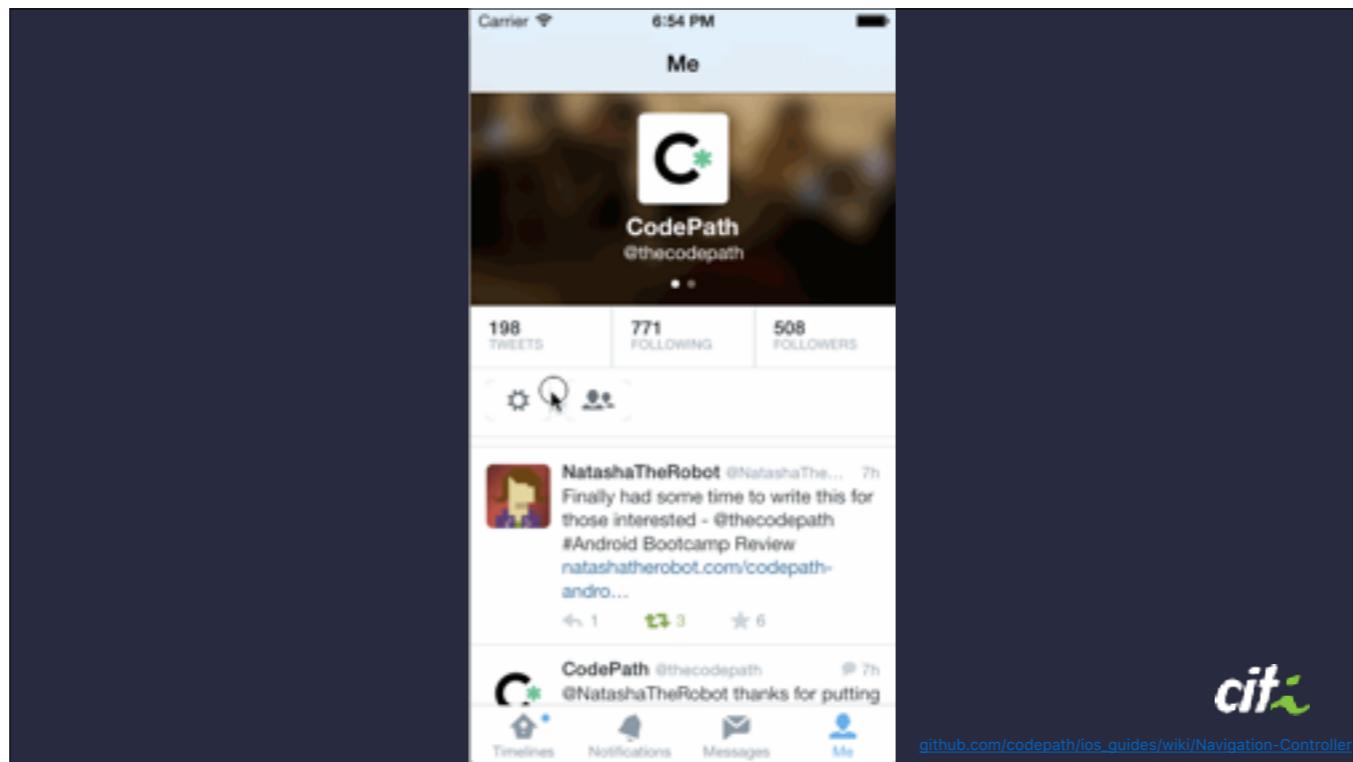


// Navigation Controller



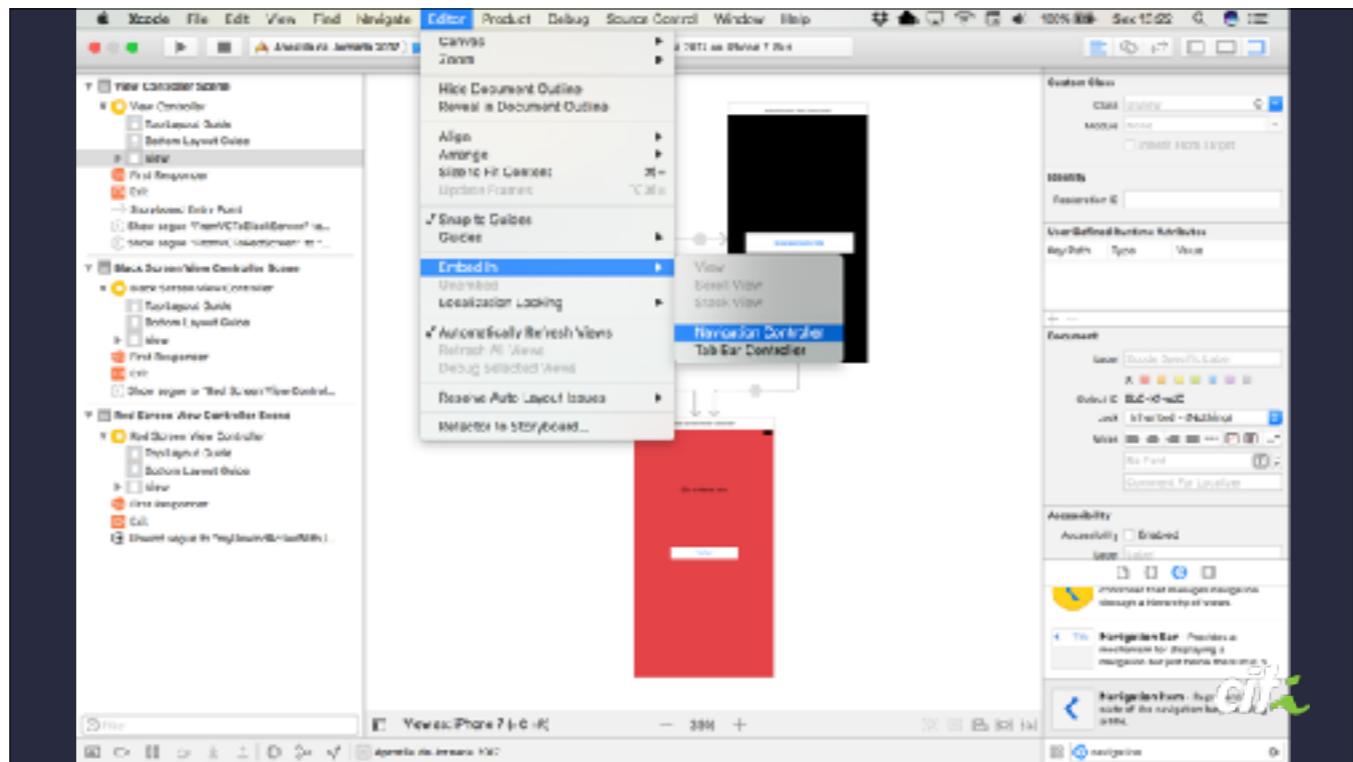


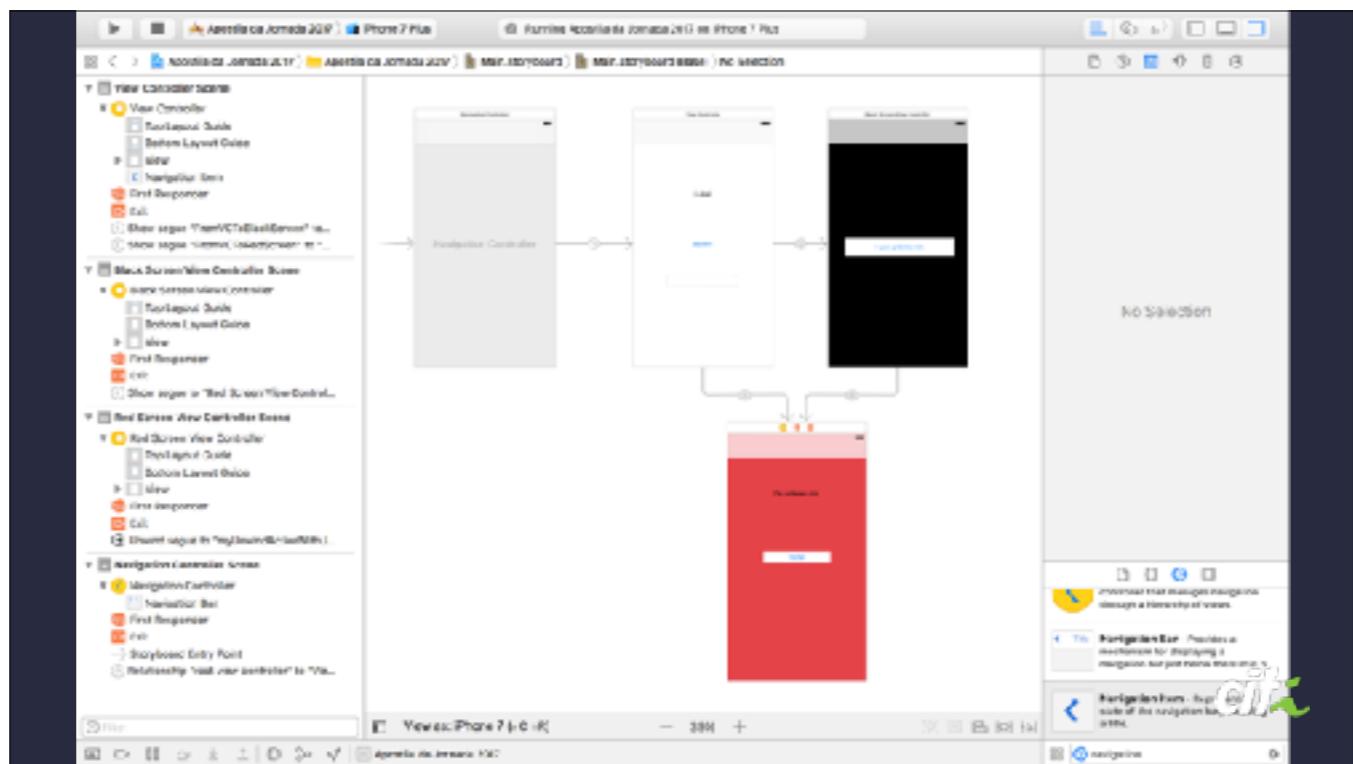
github.com/codepath/ios_guides/wiki/Navigation-Controller

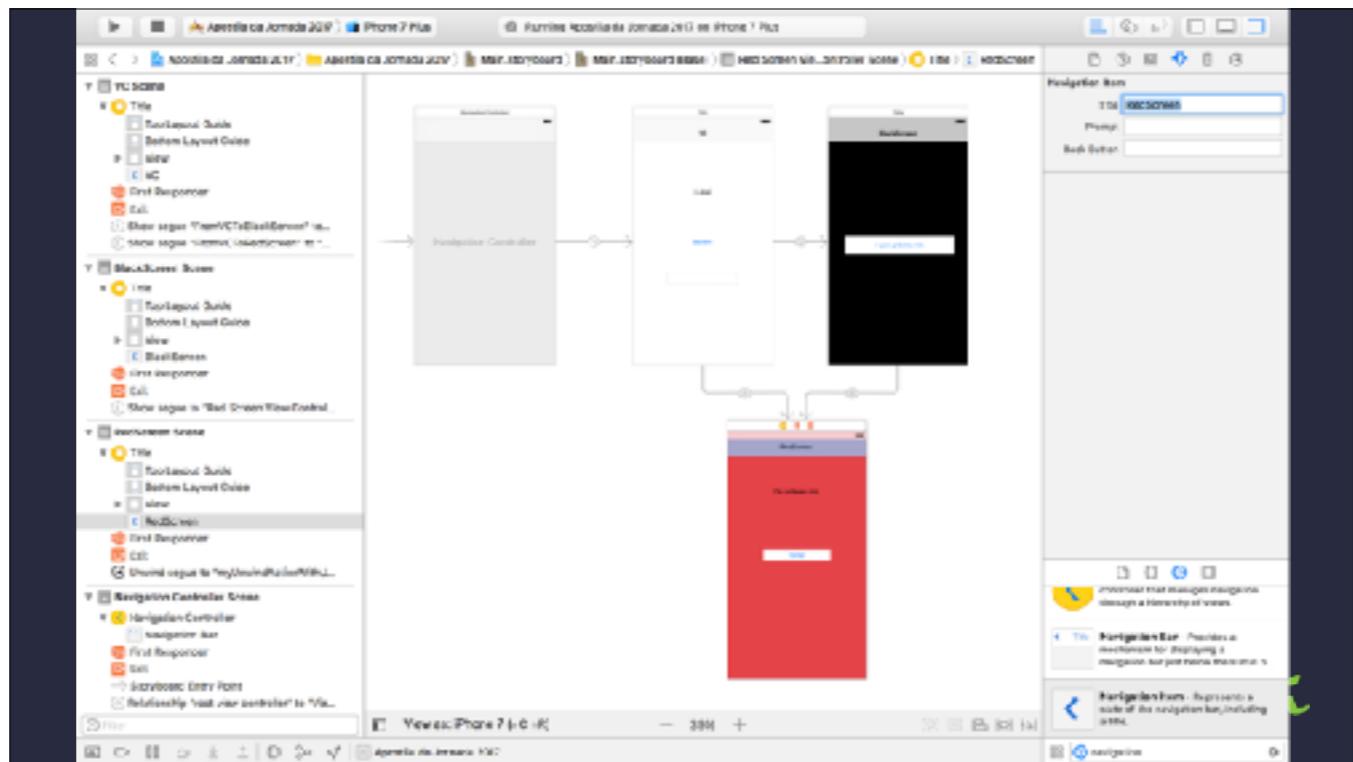


citi

github.com/codepath/ios_guides/wiki/Navigation-Controller







// Navigation controller

- UINavigationController
- Navegação hierárquica
- Pilha de VCs

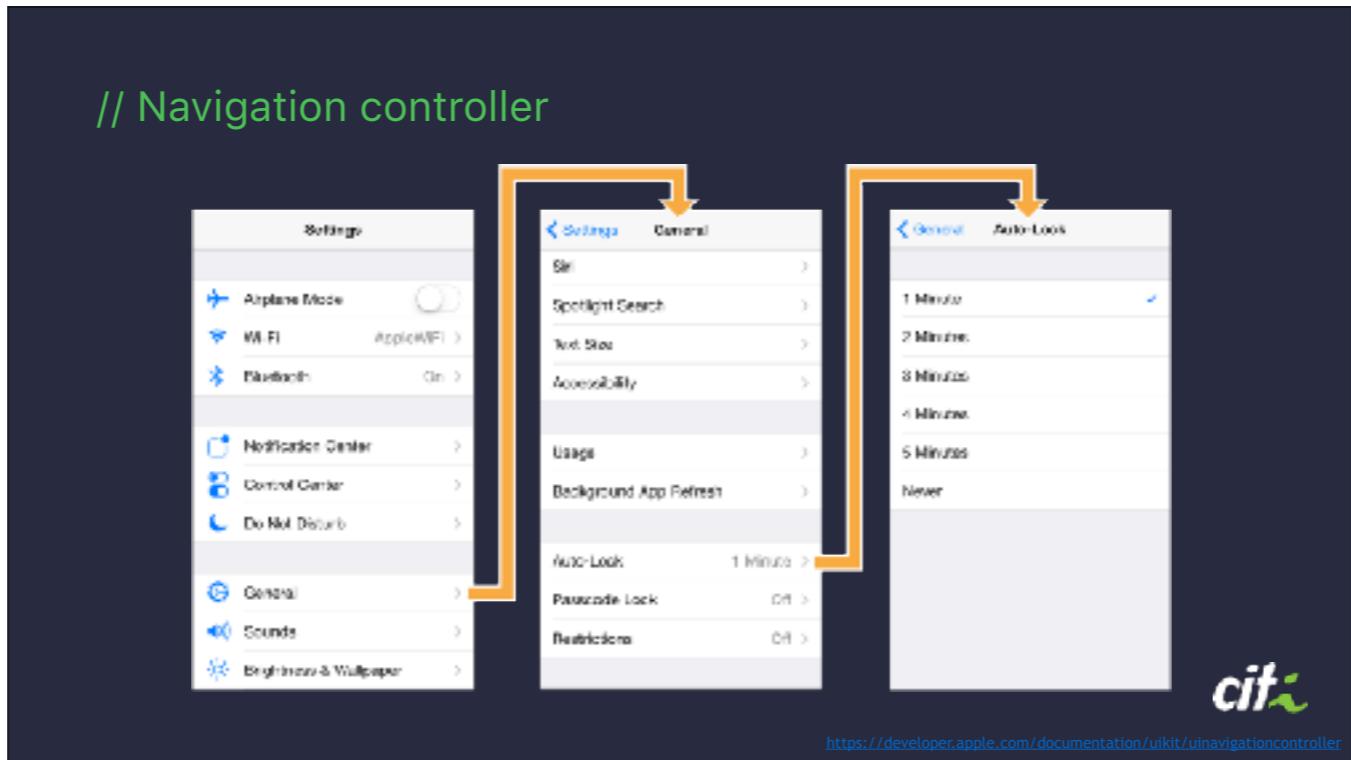


// Navigation controller

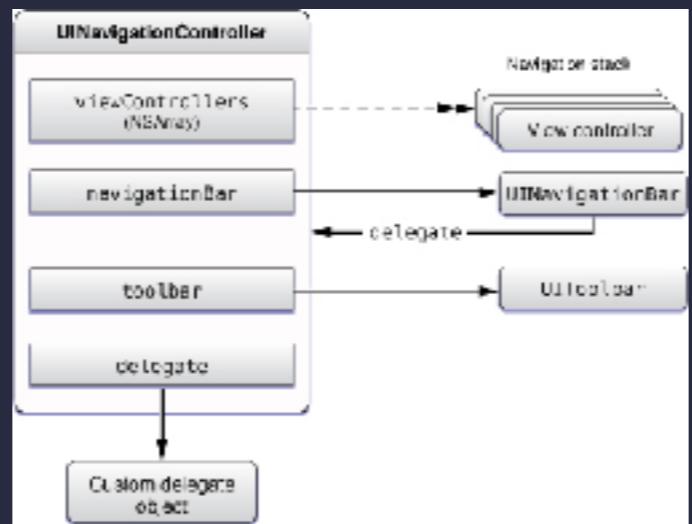
- UINavigationController
- Navegação hierárquica
- Pilha de VCs



// Navigation controller



// Navigation controller



<https://developer.apple.com/documentation/uikit/uinavigationcontroller>



// Exercício



// Exercício 11-a

Navegação



Apertar em Escolher cor leva para ColorVC

O background de MainVC deve ser da cor escolhida, em ColorVC, e a label deve indicar o nome da cor.



Apertar em um botão de cor muda o conteúdo da label.

Apertar em save retorna para o MainVC, passando a cor escolhida



adaptado de: <https://goo.gl/Bxaeob>

// Exercício 11-b Navegação com Delegate



Implementa um protocolo ColorVCDelegate:

```
func chosen(color: UIColor, by colorVC: ColorViewController)
```

retira o ColorViewController da **pilha** de navegação e atualiza interface do MainViewController



tem uma propriedade **delegate** do tipo ColorVCDelegate

ao apertar **save**, chama método chosen(color: by:) de seu **delegate**

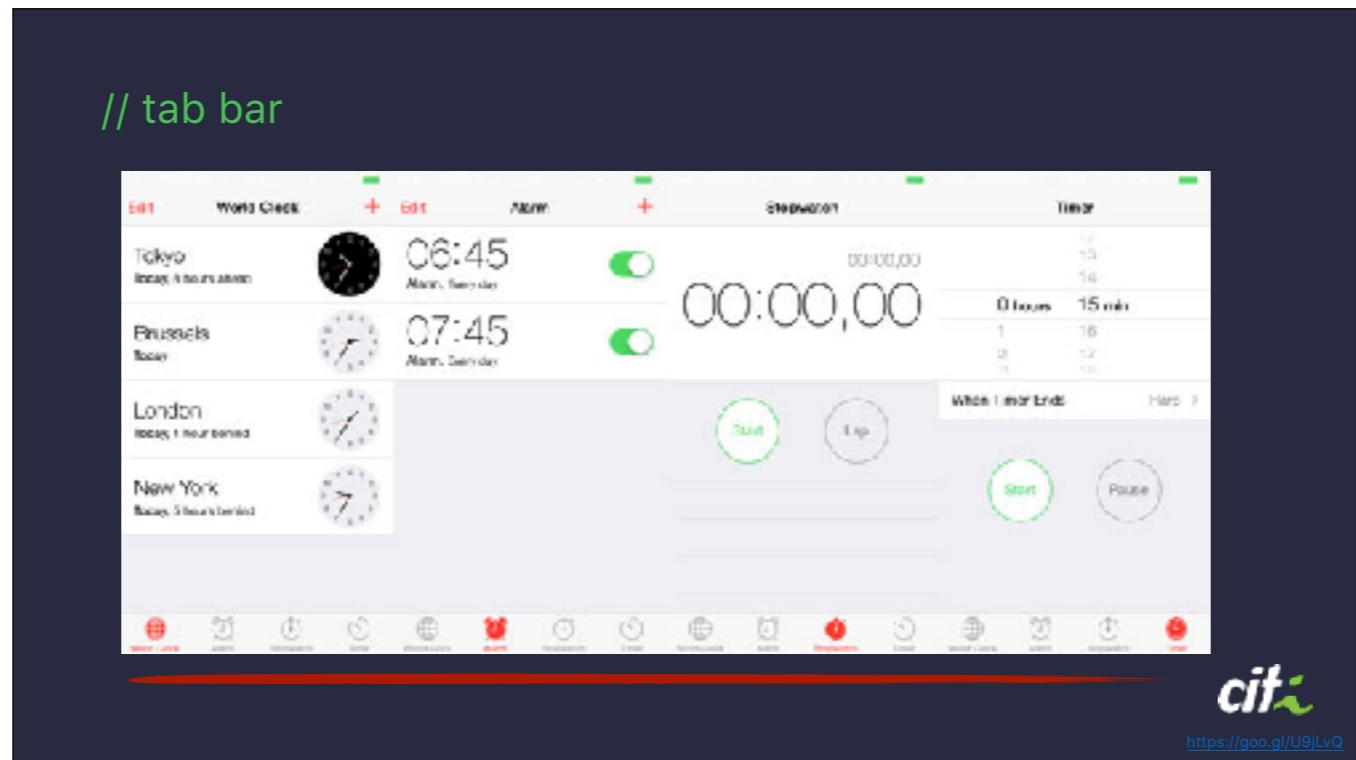


adaptado de: <https://goo.gl/Bxaeob>

// Tab Bar



// tab bar



// tab bar

Gerencia navegação

View Controllers não relacionados

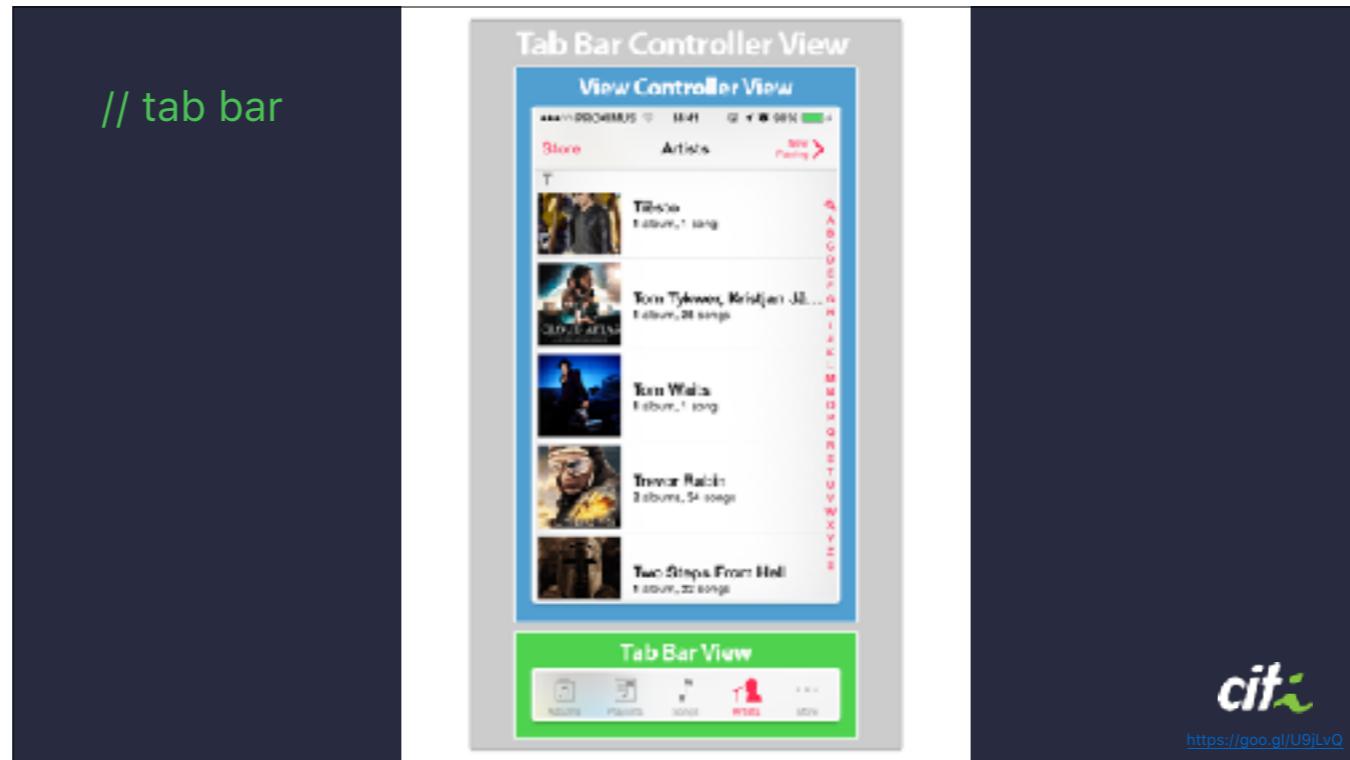
Tab bar vs Nav bar



<https://goo.gl/U9LvQ>

UITabBarController is another UIViewController subclass.
While navigation controllers manage a stack of related view controllers,
tab bar controllers manage an array of view controllers that have no explicit relation to one another.

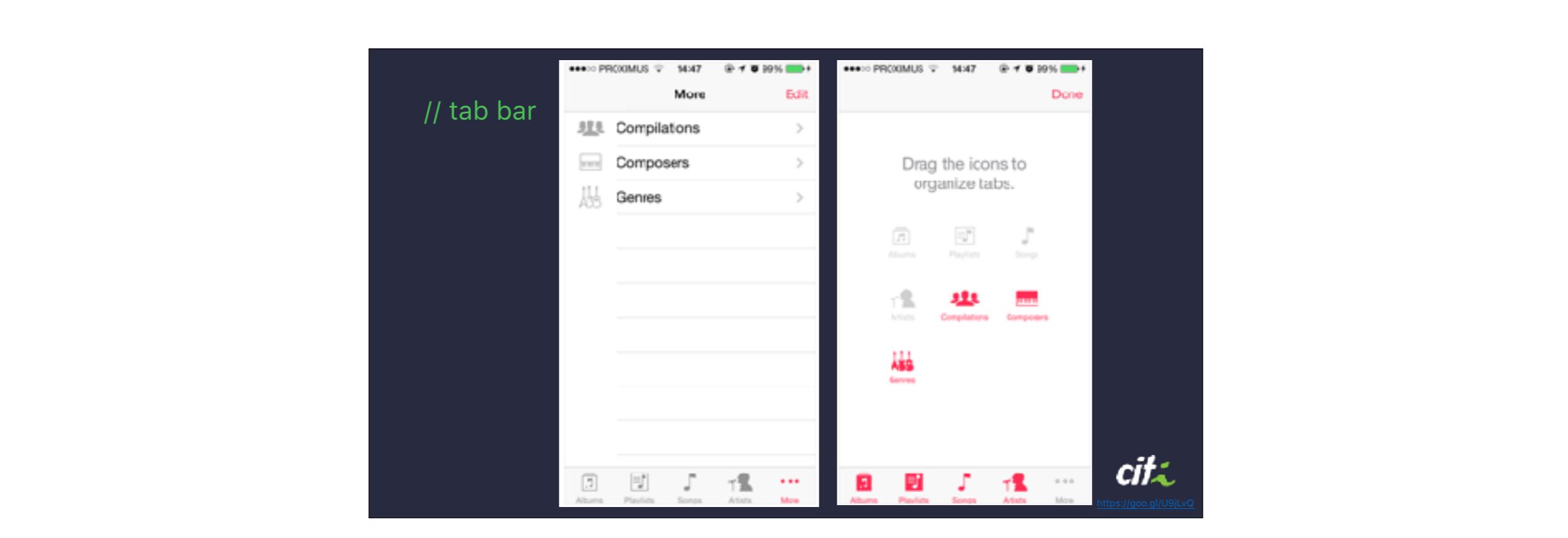
// tab bar



<https://goo.gl/U9LvQ>

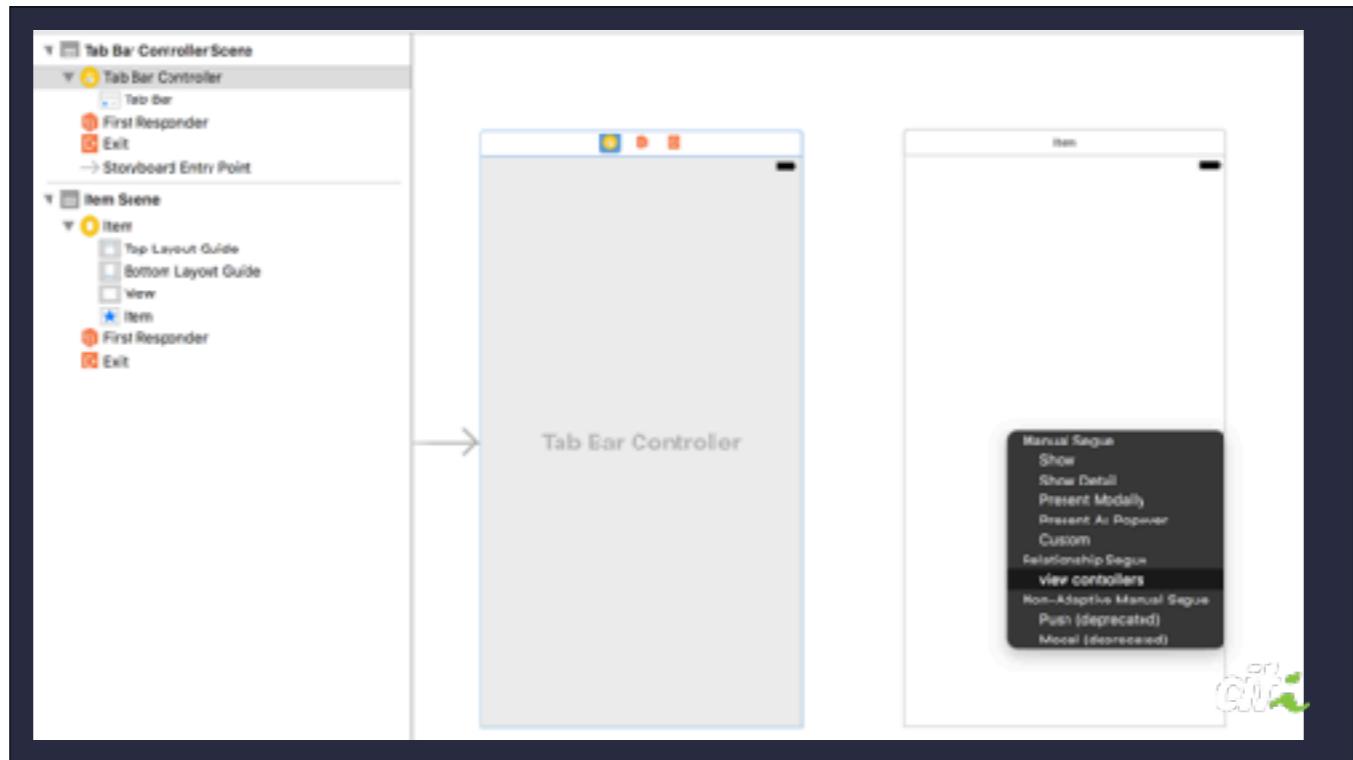
toda subclasse de UIViewController gerencia views
Tab bar controller gerencia duas:

- A da própria tab bar
- a view do viewController atual

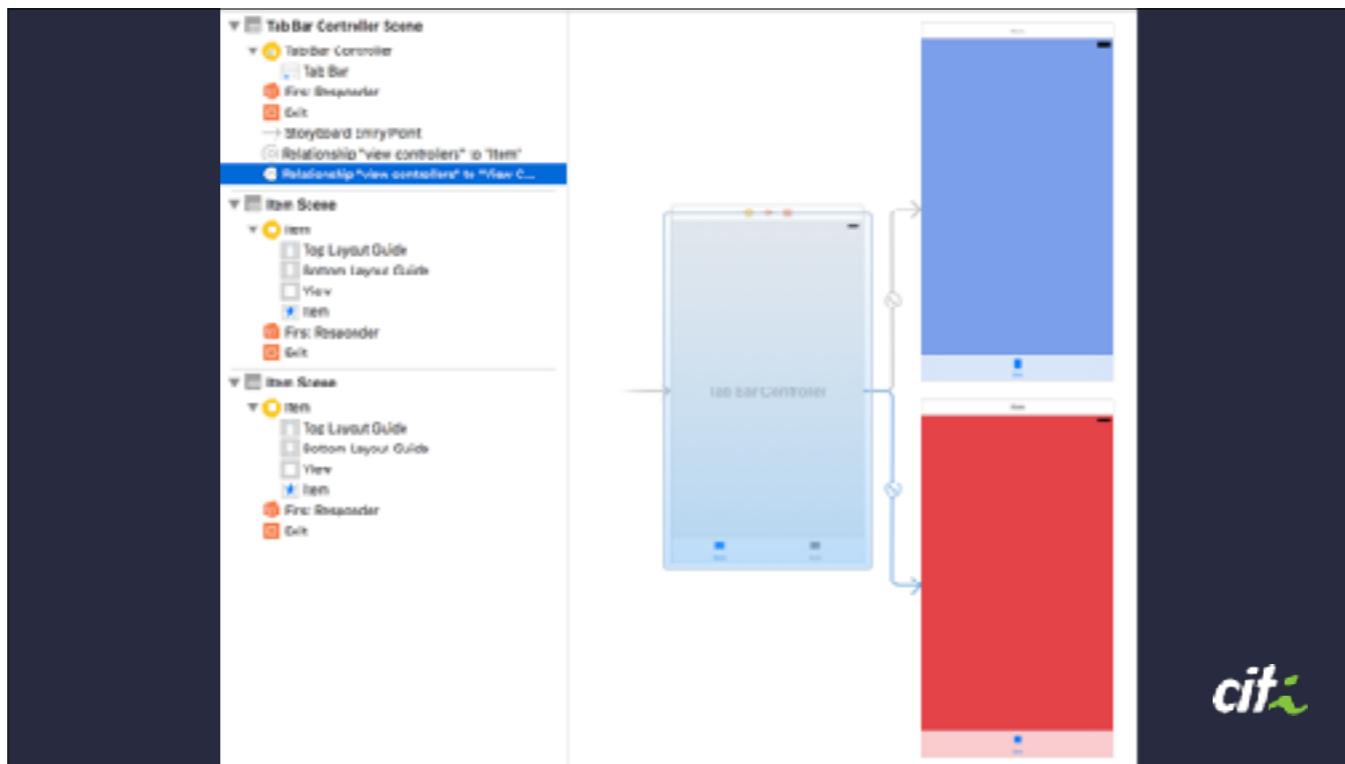


tab bar view:

- só mostra até 5 itens
- pode gerenciar mais de 5 VCs



relationship segue



adicionando segundo VC

```
// tab bar item

class RedViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        // Do any additional setup after loading the view.
        self.tabBarItem = UITabBarItem(title: "Fire", image: #imageLiteral(resourceName: "fire"), tag: 2)
    }

class BlueViewController: UIViewController {

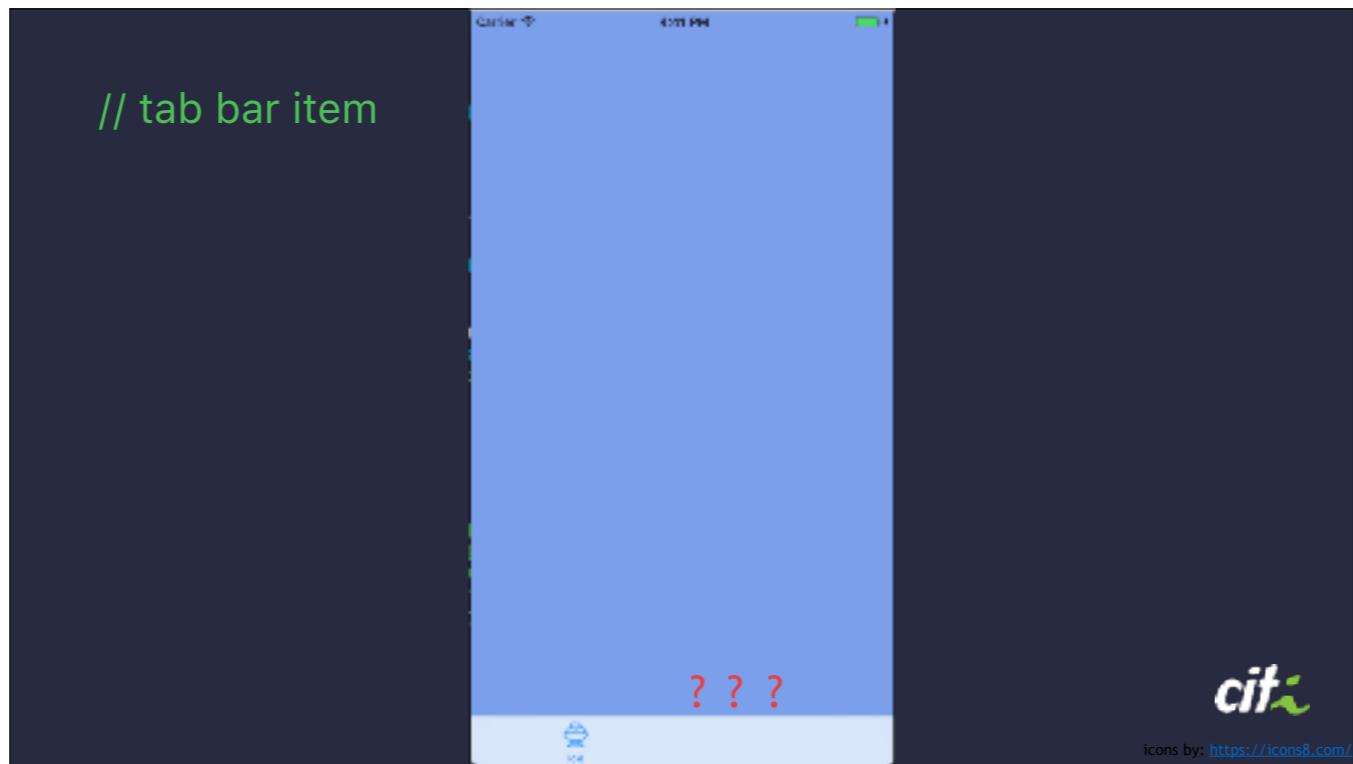
    override func viewDidLoad() {
        super.viewDidLoad()

        // Do any additional setup after loading the view.
        self.tabBarItem.title = "Ice"
        self.tabBarItem.image = #imageLiteral(resourceName: "ice")
    }
}
```

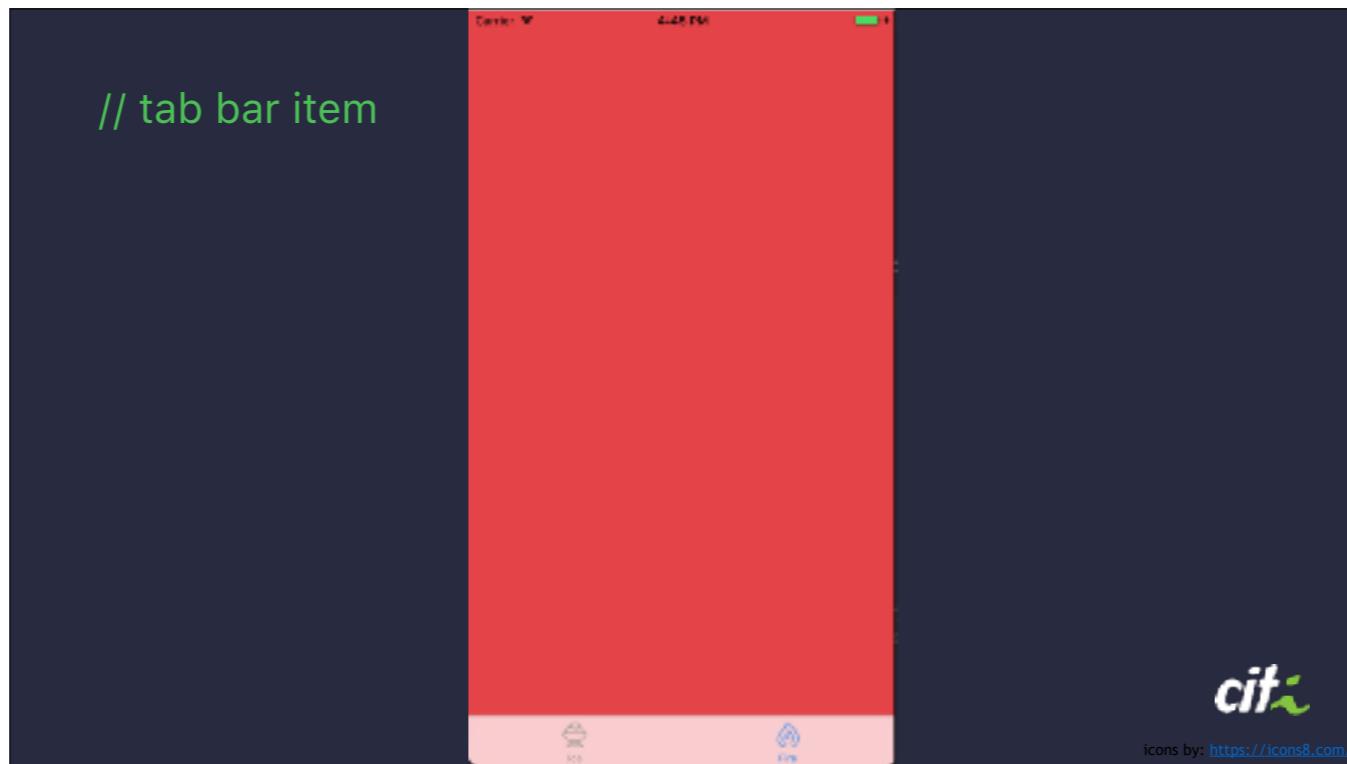


icons by: <https://icons8.com/>

adicionando item



adicionando item



adicionando item

```
// tab bar item

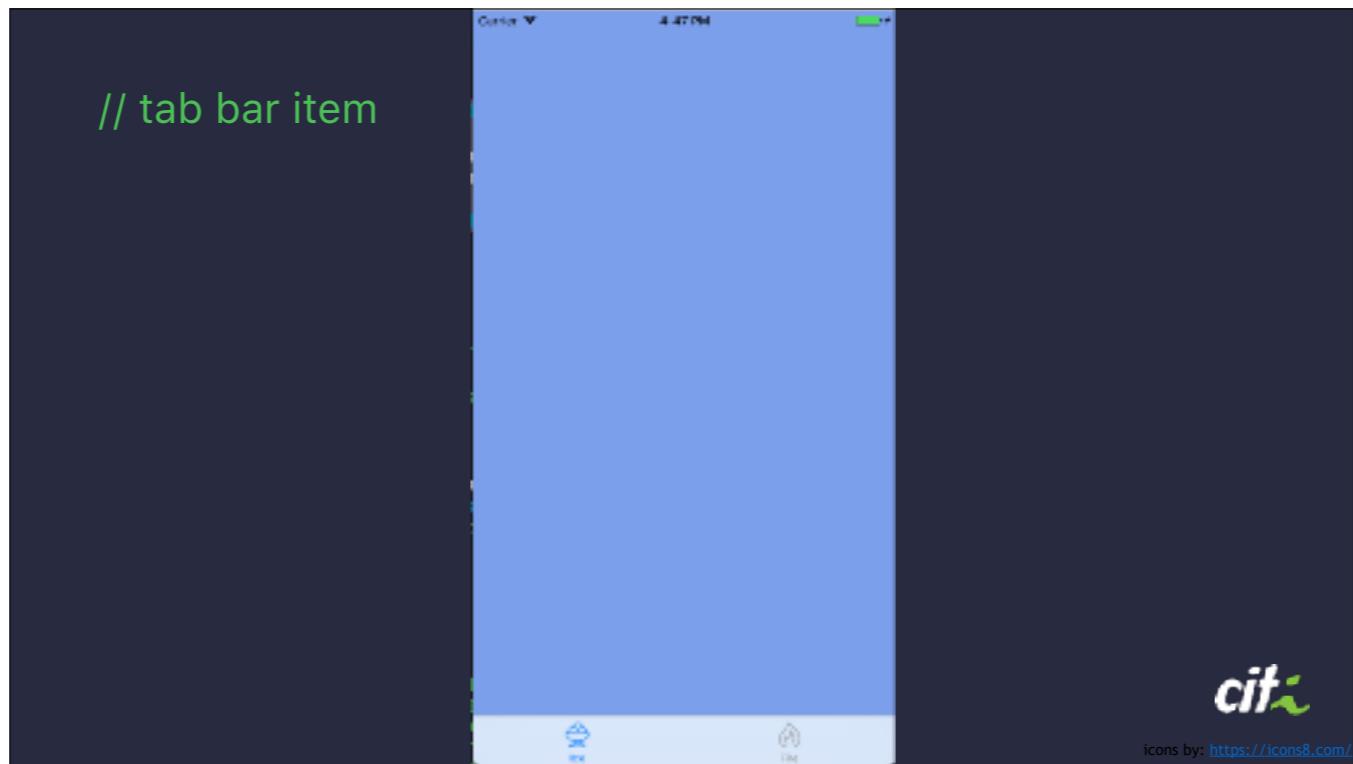
class RedViewController: UIViewController {

    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)

        self.tabBarItem = UITabBarItem(title: "Fire",
                                       image: #imageLiteral(resourceName: "fire"),
                                       tag: 2)
    }
}
```



adicionando item



adicionando item

```
// tab bar badge

class RedViewController: UIViewController {

    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)

        self.tabBarItem = UITabBarItem(title: "Fire",
                                       image: #imageLiteral(resourceName: "fire"),
                                       tag: 2)

        self.tabBarItem.badgeValue = "消防安全"
    }
}
```

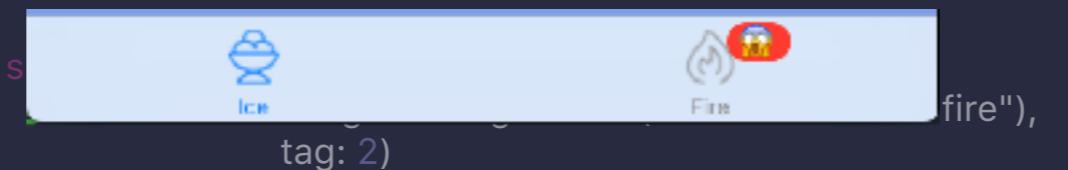


adicionando item

```
// tab bar badge

class RedViewController: UIViewController {

    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
        self.tabBar.items![1].badgeValue = "2"
    }
}
```



adicionando item

// Exercício



// Exercício 12

Tab bar grande

Fazer uma aplicação com um **TabBarController**

Colocar os seguintes **ViewControllers**:

1. CarroViewController
2. MotoViewController
3. BarcoViewController
4. NavioViewController
5. TremViewController
6. OnibusViewController

Colocar um ícone e um título no **construtor** de cada VC

Colocar uma **ImageView** em cada ViewController com uma imagem correspondente

Tentar **reajustar** os itens mostrados na tab bar



// Desafio 01

Tab e Nav

Junte os últimos dois exercícios, para que sua aplicação final contenha uma **Tab Bar**, e uma **Nav Bar** ao mesmo tempo.



// Ementa



EMENTA PRELIMINAR DO CURSO:

SWIFT	STORYBOARD
Variáveis, constantes e operadores	Views
Tipos de variáveis (números, strings, entre outras)	InputView
Coleções (array, dicionário)	ScrollView
Controle de fluxo (condicionais e loops)	TableView
Funções e closures	Navigation Controller
Enums	Tab Bar
Classes e structs	Labels
Protocols	Botões
Casting de tipos	Trocando de telas
Optionals	Integrando o storyboard com o código (outlets, buttons, entre outros)
Persistência	Webviews

COMPLEMENTO: XCODE, MONETIZAÇÃO E DESIGN DE INTERFACE.



EMENTA PRELIMINAR DO CURSO:

SWIFT	STORYBOARD
Variáveis, constantes e operações	Views
Tipos de variáveis (números, strings, entre outras)	InputView TextField
Collecções (array, dicionário)	ScrollView
Controle de fluxo (condicionais e loops)	TableView
Funções e classes	Navigation-Controller
Enums	Tab-Bar
Classes e structs	Table
Protocolos	Button
Costing/códigos	Transando-de-views
Opções	Integrando o storyboard com o código (outlets, buttons, entre outros)
-Delegação	
Parcialidade	Webviews

COMPLEMENTO: ~~WEEBS~~ MONETIZAÇÃO E DESIGN DE INTERFACE.



// Proposta semana 02

Segunda

TableView

ScrollView

Terça

WebView

Persistência

Quarta

Projeto

Quinta

Projeto

Extra

Sexta

Projeto

Extra



// Aula 06



// ScrollView



// quando usar

- Conteúdo maior que a tela
- Zoom



// UIKit

- UIScrollView
- Table Views e Collection Views



// características

- A scroll view itself has **no appearance**, but does display transient scrolling indicators as people interact with it.
- **Don't place a scroll view inside of another scroll view.** Doing so creates an unpredictable interface that's difficult to control.



<https://goo.gl/QJFYfC>

// implementação

1. mudar tamanho do **ViewController**
2. adicionar **ScrollView**
 - 2.1. fazer com que cubra a tela toda
3. adicionar **View** de conteúdo
 - 3.1. fazer com que cubra a tela toda
 - 3.2. adicionar conteúdo



<https://goo.gl/QJFYfC>

// Auto-Layout



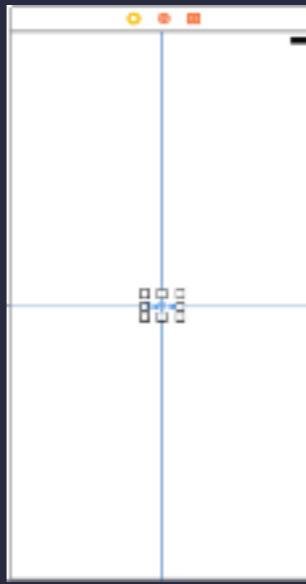
```
// auto-layout
```

Auto Layout **dynamically** calculates the **size** and **position** of all the views in your view hierarchy, based on **constraints** placed on those views



<https://goo.gl/2Rx9Jr>

```
// constraints
```



citi

This constraint-based approach to design allows you to build user interfaces that dynamically respond to both internal and external changes.

// mudanças externas

- Rotação do dispositivo
- Diferentes tamanhos de tela
- ...



<https://goo.gl/2Rx9Jr>

// mudanças internas

- Mudança no conteúdo do app
- Internacionalização
- Suporte a Dynamic Type



<https://goo.gl/2Rx9Jr>

When your app's content changes, the new content may require a different layout than the old. This commonly occurs in apps that display text or images.

Third, changing the language can affect not just the size of the text, but the organization of the layout as well. Different languages use different layout directions. English, for example, uses a left-to-right layout direction, and Arabic and Hebrew use a right-to-left layout direction.

If your iOS app supports dynamic type, the user can change the font size used in your app. This can change both the height and the width of any textual elements in your user interface. If the user changes the font size while your app is running, both the fonts and the layout must adapt.

// atributos



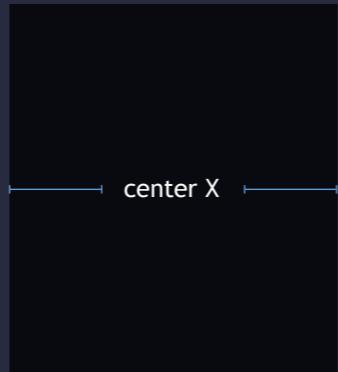
citi

// atributos



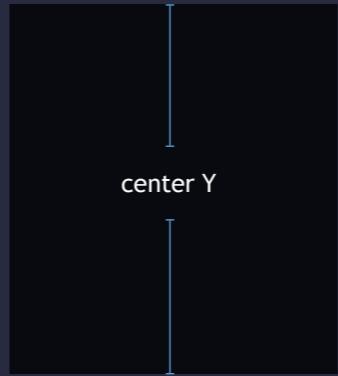
citi

// atributos



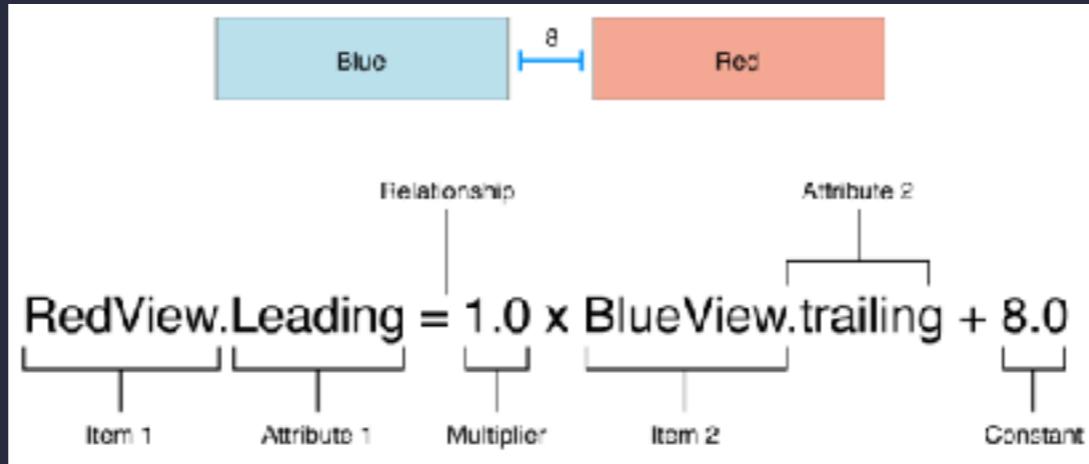
citi

// atributos



citi

// constraint



<https://goo.gl/2Rx9Jr>

When your app's content changes, the new content may require a different layout than the old. This commonly occurs in apps that display text or images.

Third, changing the language can affect not just the size of the text, but the organization of the layout as well. Different languages use different layout directions. English, for example, uses a left-to-right layout direction, and Arabic and Hebrew use a right-to-left layout direction.

If your iOS app supports dynamic type, the user can change the font size used in your app. This can change both the height and the width of any textual elements in your user interface. If the user changes the font size while your app is running, both the fonts and the layout must adapt.

// ScrollView no Xcode



// Zoom



// implementação

1. adicionar **ScrollView**
2. adicionar **Image View**
3. criar **Outlets**
4. colocar **valores** do **zoom**
5. implementar **delegate**



cif

// implementação

1. adicionar **ScrollView**
2. adicionar **Image View**
3. criar **Outlets**
4. colocar **valores** do **zoom**
5. implementar **delegate**



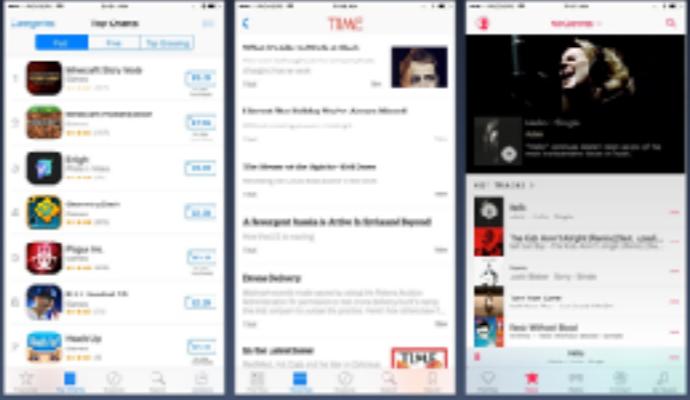
cif

// Table View



// table view

- Conteúdo em lista
- Customizável
- Subclasse de UIScrollView



cite

<https://designcode.io/cloud/chapter1/Design-TableExamples.jpg>

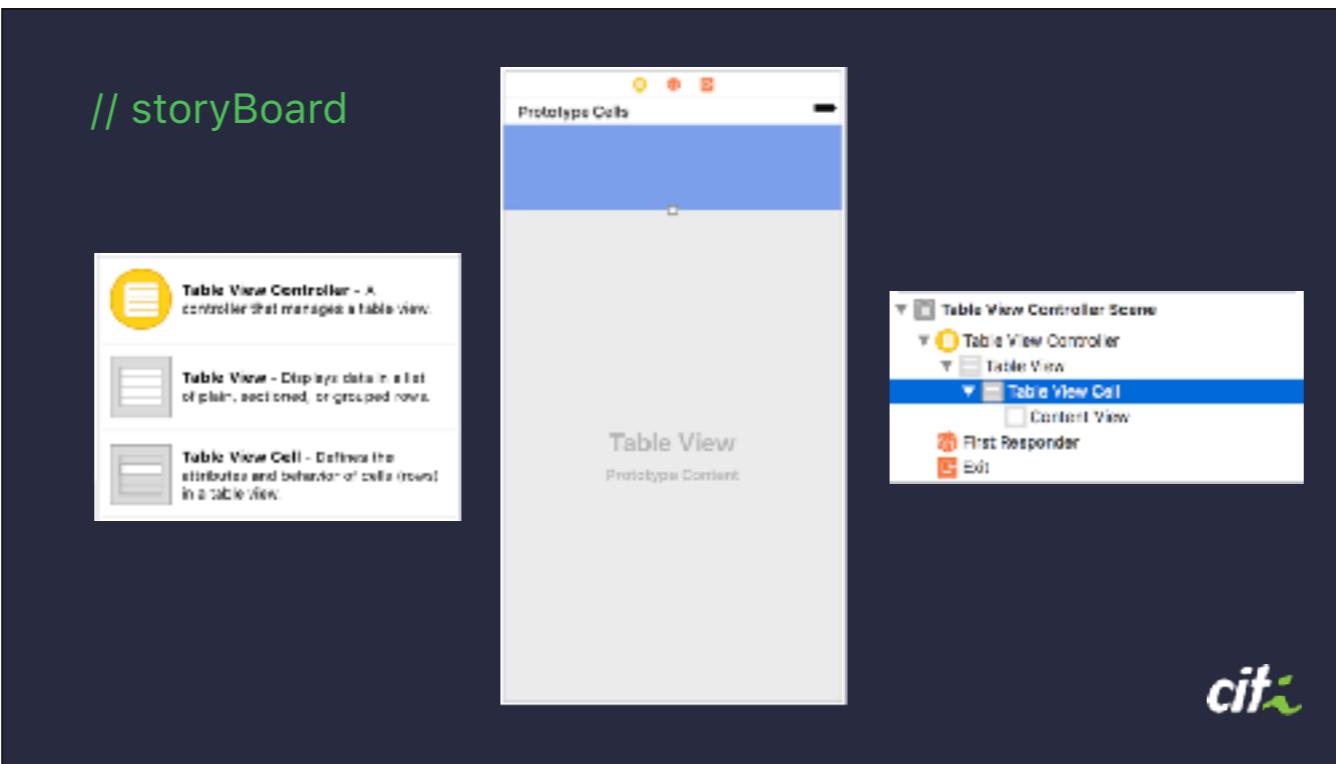
// anatomia

- Seções
- Linhas
- Células



citi

```
// storyBoard
```



// Implementação



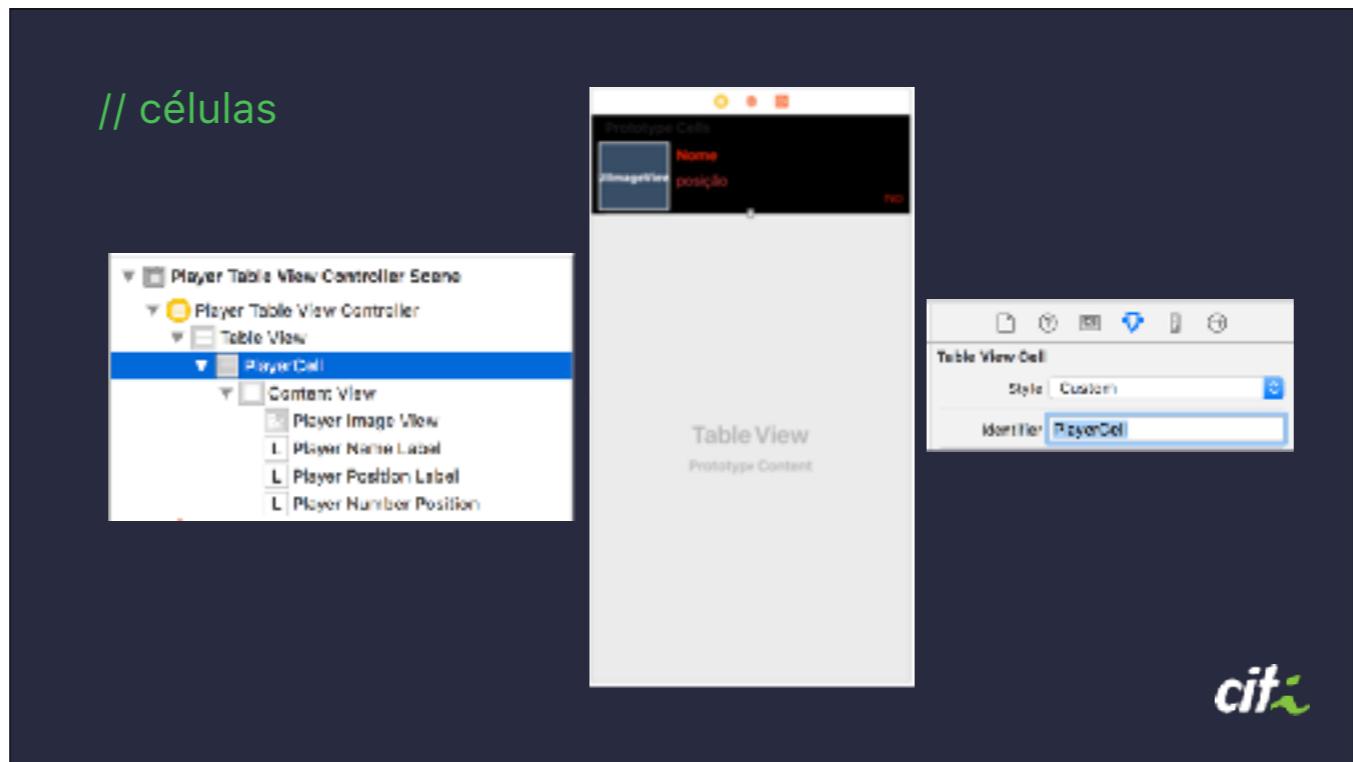
```
// tipo dos dados
```

```
class Player {  
    var nome: String  
    var posicao: String  
    var numero: Int  
    var foto: UIImage  
  
    init(nome: String, posicao: String, numero: Int, foto: UIImage) {  
        self.nome = nome  
        self.posicao = posicao  
        self.numero = numero  
        self.foto = foto  
    }  
}
```

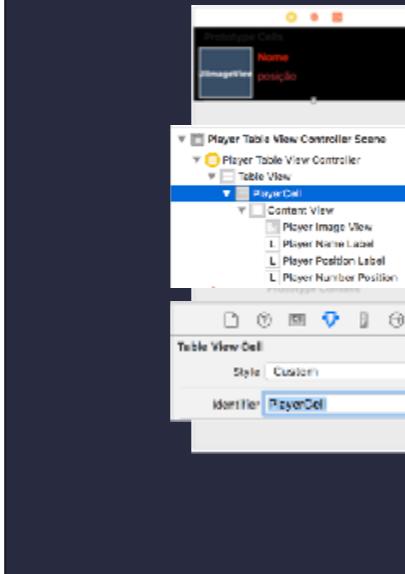


citi

```
// células
```



// células



```
class PlayerTableViewCell: UITableViewCell {  
    @IBOutlet weak var playerImageView: UIImageView!  
    @IBOutlet weak var playerNameLabel: UILabel!  
    @IBOutlet weak var playerPositionLabel: UILabel!  
    @IBOutlet weak var playerNumberPosition: UILabel!  
    // ...  
}
```



```
// controlador
```

```
class PlayerTableViewController: UITableViewController {  
    var players: [Player] = []  
  
}
```



```
// dados a serem mostrados

class PlayerTableViewController: UITableViewController {
    var players: [Player] = []

}
```



// criando/carregando dados

```
// MARK: - Cell Setup
private func loadPlayers() {
    // create player objects
    let diegoSouza = Player(nome: "Diego Souza", posicao: "Meia", numero: 97, foto: #diego)
    let magrao = Player(nome: "Magrão", posicao: "Goleiro", numero: 1, foto: #magrao)
    let durval = Player(nome: "Durval", posicao: "Zagueiro", numero: 4, foto: #durval)

    // add them to players property
    players.append(contentsOf: [diegoSouza, magrao, durval])
}
```



// chamando função

```
class PlayerTableViewController: UITableViewController {  
    var players: [Player] = []  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
  
        self.loadPlayers()  
    }  
}
```



```
// data source
```

- protocolo `UITableViewDataSource`
- provedor de dados
- implementado por `UITableViewController`



```
// data source

class PlayerTableViewController: UITableViewController {

    override func numberOfSections(in tableView: UITableView) -> Int {
        // return the number of sections
        return 1
    }

}
```

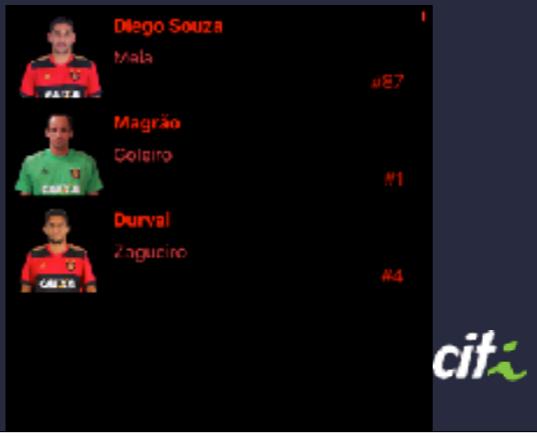


```
// data source

class PlayerTableViewController: UITableViewController {

    override func numberOfSections(in tableView: UITableView) -> Int {
        // return the number of sections
        return 1
    }

}
```



```
// data source

class PlayerTableViewController: UITableViewController {
    var players: [Player] = []

    override func tableView(_ tableView: UITableView,
                          numberOfRowsInSection section: Int) -> Int {

        // return the number of rows
        return self.players.count
    }
}
```

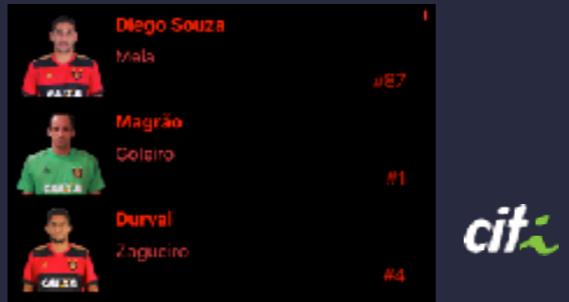


```
// data source
```

```
class PlayerTableViewController: UITableViewController {
    var players: [Player] = []

    override func tableView(_ tableView: UITableView,
                          numberOfRowsInSection section: Int) -> Int {

        // return the number of rows
        return self.players.count
    }
}
```



```
// data source
override func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell {
}

}
```



```
// data source
override func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cellIdentifier = "PlayerCell"
}

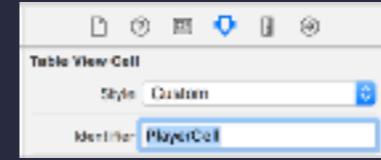
}
```



```
// data source
```

```
override func tableView(_ tableView: UITableView,  
                      cellForRowAt indexPath: IndexPath) -> UITableViewCell {
```

```
    let cellIdentifier = "PlayerCell"
```



```
}
```

```
// data source
override func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cellIdentifier = "PlayerCell"
}

}
```



```
// data source
override func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cellIdentifier = "PlayerCell"

    guard let cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifier,
        for: indexPath)

    }

}
```



```
// data source
override func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cellIdentifier = "PlayerCell"

    guard let cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifier,
        for: indexPath) as? PlayerTableViewCell
    }

}
```



```
// data source
    override func tableView(_ tableView: UITableView,
                          cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        let cellIdentifier = "PlayerCell"

        guard let cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifier,
                                                      for: indexPath) as? PlayerTableViewCell
        else {
            fatalError("Tipo de célula não é PlayerTableViewCell")
        }

    }
```



```
// data source
override func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cellIdentifier = "PlayerCell"

    guard let cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifier,
                                                for: indexPath) as? PlayerTableViewCell
    else {
        fatalError("Tipo de célula não é PlayerTableViewCell")
    }

    // Configure the cell...
    let currentPlayer = self.players[indexPath.row]

}
```



```
// data source
    override func tableView(_ tableView: UITableView,
                          cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        let cellIdentifier = "PlayerCell"

        guard let cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifier,
                                                      for: indexPath) as? PlayerTableViewCell
        else {
            fatalError("Tipo de célula não é PlayerTableViewCell")
        }

        // Configure the cell...
        let currentPlayer = self.players[indexPath.row]

        cell.playerImageView.image = currentPlayer.foto
        cell.playerNameLabel.text = currentPlayer.nome
        cell.playerPositionLabel.text = currentPlayer.posicao
        cell.playerNumberPosition.text = "#" + String(currentPlayer.numero)
    }
}
```



```
// data source
    override func tableView(_ tableView: UITableView,
                          cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        let cellIdentifier = "PlayerCell"

        guard let cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifier,
                                                      for: indexPath) as? PlayerTableViewCell
        else {
            fatalError("Tipo de célula não é PlayerTableViewCell")
        }

        // Configure the cell...
        let currentPlayer = self.players[indexPath.row]

        cell.playerImageView.image = currentPlayer.foto
        cell.playerNameLabel.text = currentPlayer.nome
        cell.playerPositionLabel.text = currentPlayer.posicao
        cell.playerNumberPosition.text = "#" + String(currentPlayer.numero)

        return cell
    }
```



// resultado final

	Diego Souza	8.67
	Magno	8.1
	Duvval	8.0

↳

citi

// resultado final

	Diego Souza	8.67
	Magno	8.1
	Duvval	8.0



citi

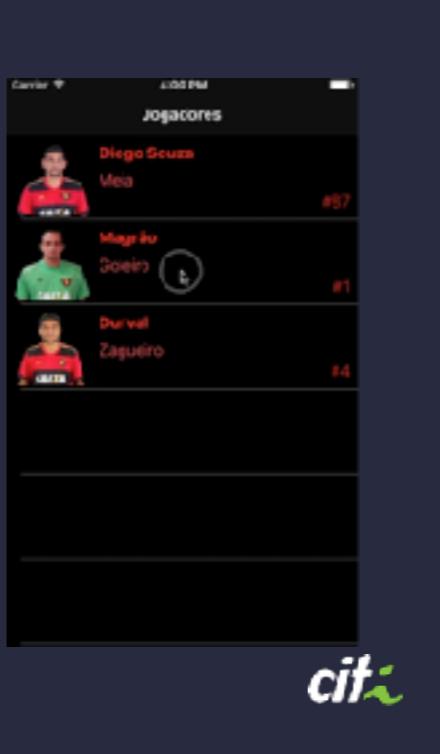
// Exercício



// Exercício 13

Lista de coisas

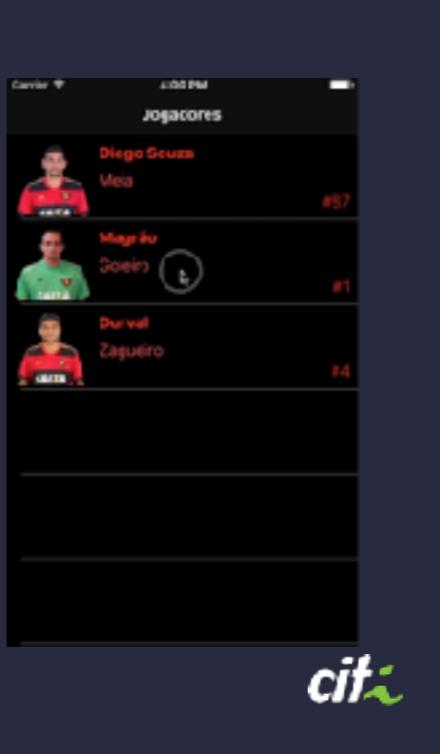
1. Faça uma **Table View** que mostre pelo menos **3 itens** de uma lista (contendo pelo menos foto e nome)
2. ao tocar num item, **direcione o usuário** para uma tela com os detalhes do item selecionado
 - 2.1. Permita que o usuário possa dar **zoom** na foto mostrada



// Exercício 13

Lista de coisas

1. Faça uma **Table View** que mostre pelo menos **3 itens** de uma lista (contendo pelo menos foto e nome)
2. ao tocar num item, **direcione o usuário** para uma tela com os detalhes do item selecionado
 - 2.1. Permita que o usuário possa dar **zoom** na foto mostrada



// Edições



// remoção

Jogadores			
	Diego Souza	Mata	487
	Magrão	Goleiro	01
	Durval	Zagueiro	44

citi

// remoção

Jogadores			
	Diego Souza	Mata	487
	Magrão	Goleiro	01
	Durval	Zagueiro	44

citi

```
// remoção

override func tableView(_ tableView: UITableView,
    commit editingStyle: UITableViewCellEditingStyle,
    forRowAt indexPath: IndexPath) {

    if editingStyle == .delete {
        // Delete the row from the data source
        self.players.remove(at: indexPath.row)
        tableView.deleteRows(at: [indexPath], with: .fade)

    }
}
```

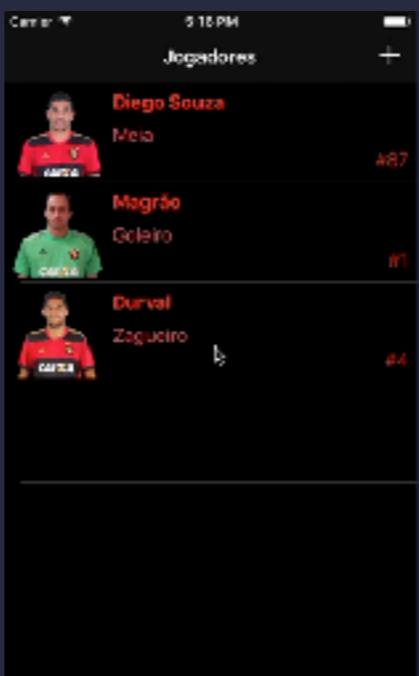


// inserção

Jogadores		
Diego Souza	Méio	487
Magrão	Goleiro	101
Durval	Zagueiro	64

citi

// inserção



citi

```
// inserção

@IBAction func addPlayer(_ sender: Any) {
    let newIndexPath = IndexPath(row: players.count, section: 0)

    let andre = Player(nome: "André",
                        posicao: "Atacante",
                        numero: 90,
                        foto: #imageLiteral(resourceName: "andre"))

    players.append(andre)

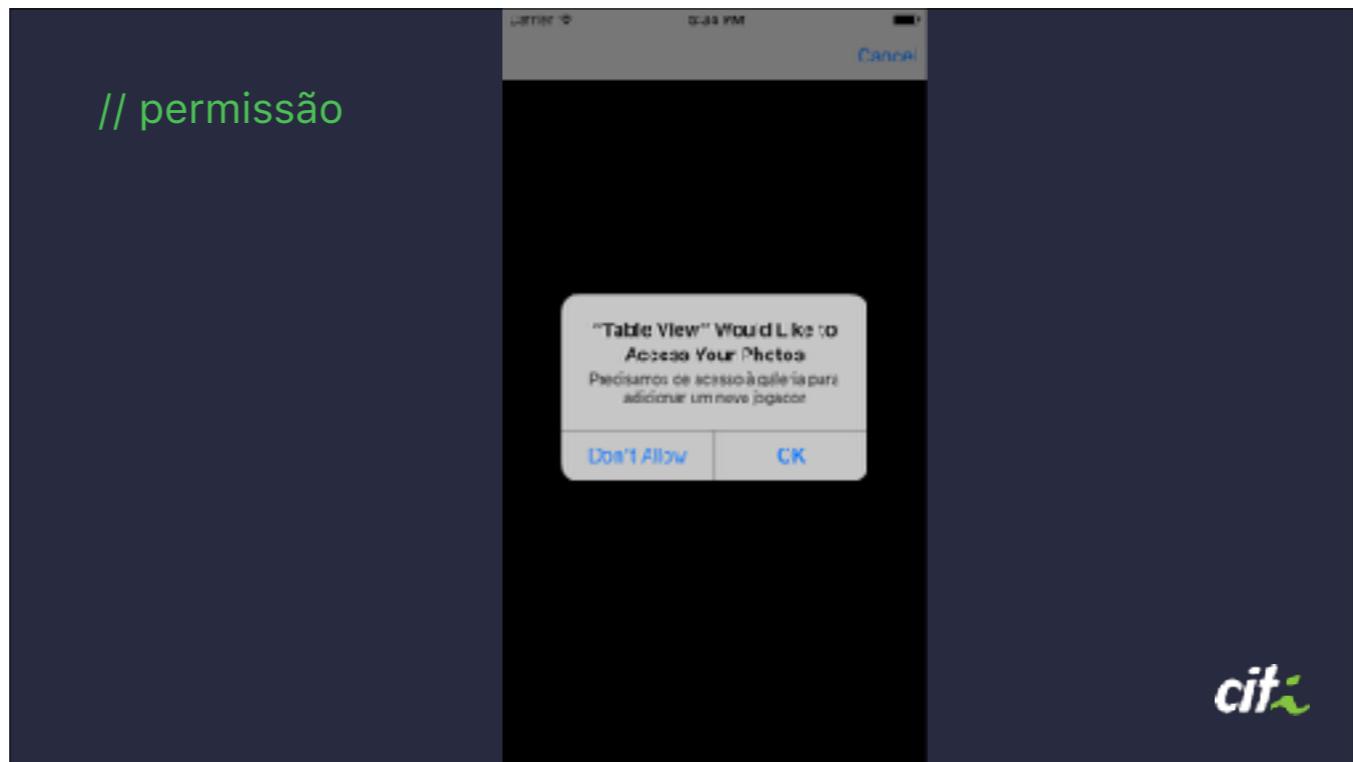
    self.tableView.insertRows(at: [newIndexPath], with: .automatic)
}
```



// Acesso à galeria



// permissão

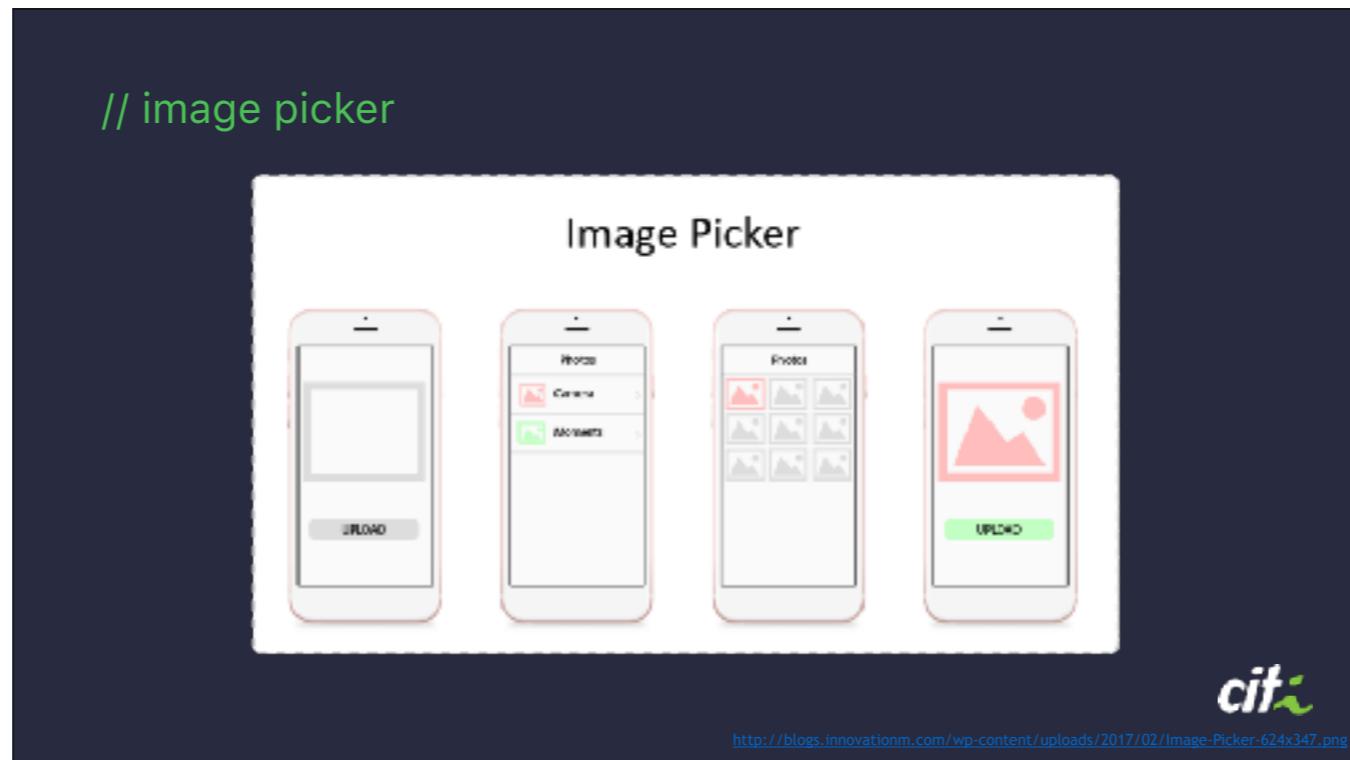


// info.plist

Key	Type	Value
Information Property List	Dictionary	{15 items}
Localization native development region	String	en
Executable file	String	\$(EXECUTABLE_NAME)
Bundle Identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	0.0
Bundle name	String	\$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle version string, short	String	1.0
Bundle version	String	1
Application requires iPhone environment	Boolean	YES
Launch screen interface file base name	String	LaunchScreen
Main storyboard file base name	String	Main
Required device capabilities	Array	{1 item}
Status bar is initially hidden	Boolean	NO
Privacy - Photo Library Usage Description	String	Permissão de acesso à galeria para adicionar um novo jogador
Supported Interface orientations	Array	{15 items}

citi

// image picker



```
// abrindo galeria
```

```
if UIImagePickerController.isSourceTypeAvailable(.photoLibrary) {  
    let imagePicker = UIImagePickerController()  
  
    imagePicker.delegate = self  
  
    imagePicker.sourceType = .photoLibrary  
  
    imagePicker.allowsEditing = true  
  
    self.present(imagePicker, animated: true, completion: nil)  
}
```



```
// capturando imagem selecionada

extension PlayerTableViewController: UIImagePickerControllerDelegate ,  
    UINavigationControllerDelegate {  
  
    func imagePickerController(_ picker: UIImagePickerController,  
                           didFinishPickingMediaWithInfo info: [String : Any]) {  
  
        if let pickedImage = info[UIImagePickerControllerOriginalImage] as? UIImage {  
  
            let imagemSelecionada = pickedImage  
            // adicionar na tabela  
        }  
        picker.dismiss(animated: true, completion: nil)  
    }  
}
```



// Exercício



// Exercício 14

Listas de coisas II

1. ponha um botão de **adicionar**, que leva o usuário para uma tela onde ele escreverá um **nome**, e uma **foto** para um novo item a ser **inserido** na tabela tabela.
2. permita que o usuário possa **deletar** um item da tabela

// Extra

3. permita que o usuário possa **editar** um item da tabela



DÚVIDAS

?

citi