

iOS Swift - UI 200





Copyright © Quattro Treinamentos Empresariais LTDA

Todos os direitos autorais reservados. Este manual não pode ser copiado, fotocopiado, reproduzido, traduzido ou convertido em qualquer forma eletrônica, ou legível por qualquer meio, em parte ou no todo, sem a aprovação prévia, por escrito, da Quattro Treinamentos Empresariais Ltda, estado o contrafator sujeito a responder por crime de Violação do Direito Autoral, conforme o art.184 do Código Penal Brasileiro, além de responder por Perdas e Danos. Todos os logotipos e marcas utilizados neste material pertencem às suas respectivas empresas.

Autoria:

Tiago Santos de Souza

Roberto Rodrigues Junior

Revisão:

Tiago Santos de Souza

Design, edição e produção:

Tiago Santos de Souza

"As marcas registradas e os nomes comerciais citados nesta obra, mesmo que não sejam assim identificados, pertencem aos seus respectivos proprietários nos termos das leis, convenções e diretrizes nacionais e internacionais."

Edição nº3

Janeiro 2018

Sumário



Apresentação.....	3
Capítulo 1 - Simulação e debugging.....	5
Seção 1-1 - Simulador X Devices.....	7
Seção 1-2 - Debugging.....	10
Capítulo 2 - Protocolos.....	15
Seção 2-1 - Protocolos.....	17
Seção 2-2 - Protocolos recorrentes.....	18
Capítulo 3 - Extensões.....	19
Seção 3-1 - Extensões.....	21
Seção 3-2 - Extensões de tipos existentes.....	22
Seção 3-3 - Extensões para adoção de protocolos.....	23
Capítulo 4 - UIResponder.....	25
Seção 4-1 - UIResponder.....	27
Capítulo 5 - Trabalhando com textos.....	29
Seção 5-1 - UITextField.....	31
Seção 5-2 - UITextView.....	33
Capítulo 6 - Trabalhando com Imagens.....	35
Seção 6-1 - UIImage.....	37
Seção 6-2 - UIImageView.....	39
Seção 6-3 - Definindo o ícone da aplicação.....	41
Capítulo 7 - Barras de ferramentas - Toolbar.....	43
Seção 8-1 - UIToolBar.....	45
Seção 7-2 - UIBarButtonItem.....	46
Seção 7-3 - Fixed Space e Flexible Space.....	47
Capítulo 8 - UIProgressView.....	49
Seção 8-1 - UIProgressView.....	51
Capítulo 9 - UIActivityIndicatorView.....	53
Seção 9-1 - UIActivityIndicatorView.....	55
Capítulo 10 - Autolayout.....	57
Seção 10-1 - Introdução ao Autolayout.....	59
Seção 10-2 - Trabalhando com previews.....	61
Seção 10-3 - Constraints.....	62
Capítulo 11 - UIStackView.....	67
Seção 11-1 - UIStackView.....	69
Seção 12-2 - Embed in Stack.....	70
Capítulo 12 - UIScrollView.....	71
Seção 17-1 - UIScrollView.....	73
Capítulo 13 - Closures.....	75
Seção 13-1 - Sintaxe das closures.....	77
Capítulo 14 - Trabalhando com Alertas.....	79
Seção 14-1 - UIAlertController.....	81
Seção 14-2 - UIAlertAction.....	83
Capítulo 15 - UIPickerView.....	85
Seção 15-1 - UIPickerView.....	87
Capítulo 16 - UIDatePicker, Date e Timer.....	91
Seção 16-1 - UIDatePicker.....	93
Seção 16-2 - Date.....	95
Seção 16-3 - DateFormatter.....	96
Seção 16-4 - Timer.....	98
Capítulo 17 - UITableView.....	99
Seção 17 -1 - UITableView.....	101
Seção 17 -2 - UITableViewCell.....	104
Seção 17 -3 - UITableViewController.....	106
Capítulo 18 - UICollectionView.....	107
Seção 18 -1 - UICollectionView.....	109
Seção 18 -2 - UICollectionViewCell.....	111
Seção 18 -3 - UICollectionViewController.....	112
Capítulo 19 - UIVisualEffects.....	113
Seção 19 -1 - UIVisualEffect.....	115
Seção 19 -2 - UIVisualEffectView.....	117
Capítulo 20 - Eventos de toque e UITapGestureRecognizer.....	119
Seção 20 -1 - Respondendo a eventos de toque.....	121
Seção 20 -2 - UITapGestureRecognizer.....	122
Capítulo 21 - Trabalhando com multitelas.....	123
Seção 21 -1 - UIStoryboard.....	125
Seção 21 -2 - UIStoryboardSegue.....	126
Seção 21 -3 - Dispensando View Controllers.....	128
Seção 21 -4 - Criando novos Storyboards.....	130
Seção 21 -5 - Storyboard Reference.....	131
Capítulo 22 - Container View.....	133
Seção 22 -1 - Container View.....	135
Capítulo 23 - UINavigationController.....	137
Seção 23 -1 - UINavigationController.....	139
Seção 23 -1 - UINavigationControllerItem.....	140
Capítulo 24 - UITabBarController.....	141
Seção 24 -1 - UITabBarController.....	143
Capítulo 25 - Encapsulamento e Subscripts.....	145
Seção 25 -1 - Encapsulamento.....	147
Seção 25 -2 - Subscripts - Sintaxe.....	148
Capítulo 26 - Tratamento de erros.....	151
Seção 26 -1 - Tratamento de erros.....	153
Apendice 1 - Caderno de Exercícios.....	155



Apresentação

Agora que já conhecemos as sintaxes do Core da linguagem **Swift** e aprendemos a utilizar seus principais recursos, estamos prontos para partir na direção do desenvolvimento da interfaces mais avançadas dos nossos aplicativos.

A partir deste ponto, vamos alinhar a codificação com o desenvolvimento da interface gráfica das aplicações. Além de testar os recursos implementados no simulador de iPhone ou iPad, podemos acompanhar o desenvolvimento diretamente nos nossos dispositivos físicos.

Vamos aprofundar nossos conhecimentos na ferramenta **Xcode**, descobrir quanto o trabalho pode ser produtivo com os seus recursos para interface. Trabalhar com aplicativos multi vistas, botões, textos , imagens, etc.

Nosso principal foco nesse momento do processo será o framework **UIKit** que auxilia na criação de elementos visuais e interações com o usuário através de diversos elementos.

Também veremos como fazer entrada de dados e permitir que o usuário insira informações, trabalhos simples com animações, como utilizar a poderosa ferramenta de autolayout, recurso de extrema importância dada a atual diversidade de resoluções dos dispositivos Apple.

Temos que ter em mente que apenas o elemento visual em si não garante a resposta do sistema, claro que por trás temos toda a codificação necessária para cada funcionalidade.

Não se esqueça de consultar as documentações oficiais da Swift e Xcode para complementar os seus conhecimentos.

<https://developer.apple.com/swift/>

<https://developer.apple.com/xcode/>

Bons estudos!



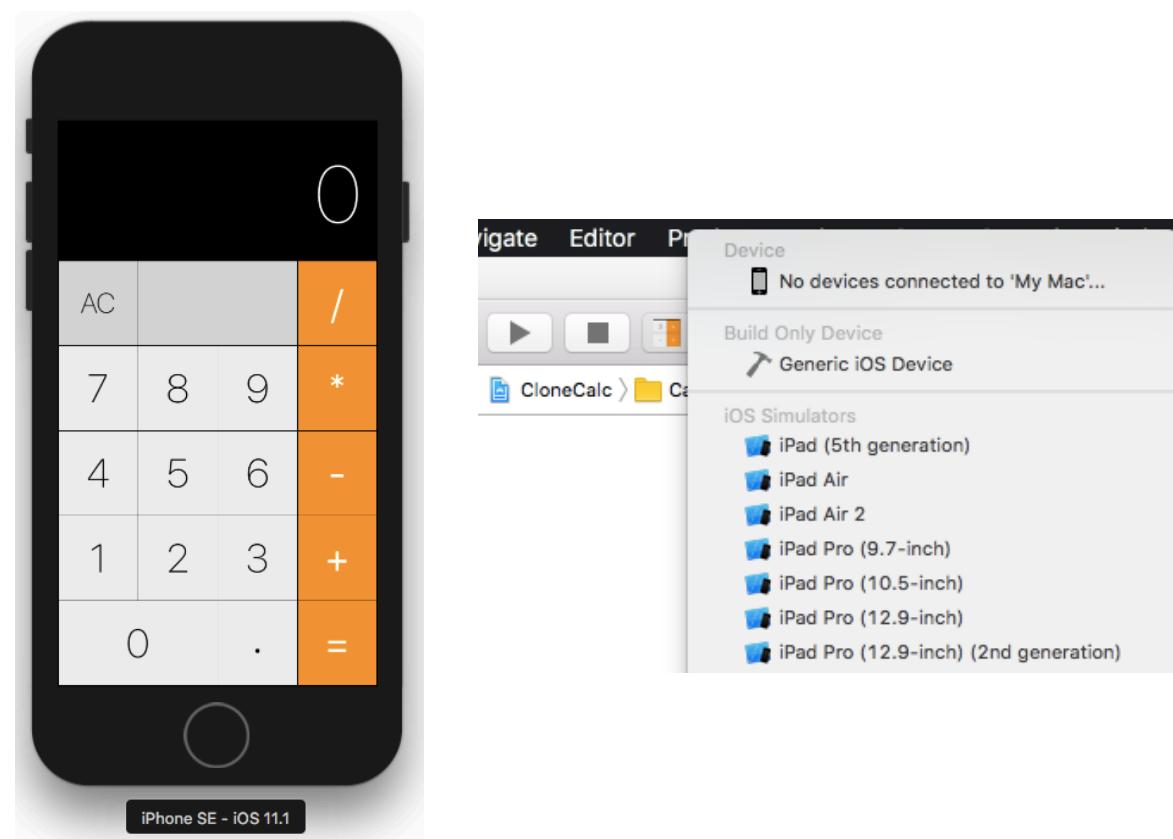
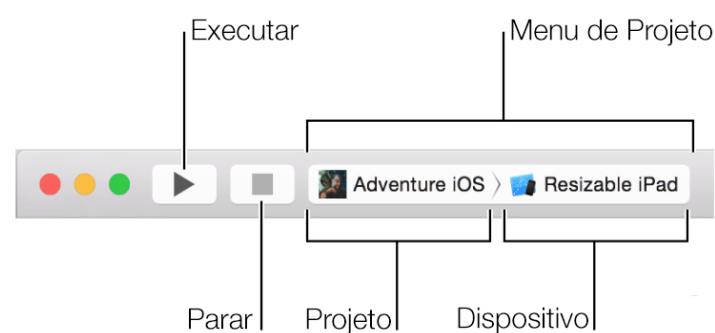
Simulação e debugging

Seção 1-1

Simulador X Devices



Como vimos no módulo anterior, podemos fazer a simulação das nossas aplicações em um simulador, que tem a capacidade proporcionar uma vasta experiência com recursos semelhantes aos dispositivos físicos.



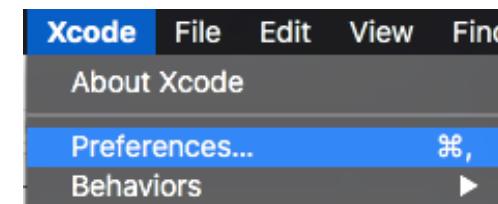
Claro que o simulador tem limitações, se comparado a um dispositivo. Com o simulador não podemos acessar algumas regiões do sistema, como a captação de imagens com a câmera, ou a captação de movimentos. É nesse momento que entra em ação a simulação diretamente em um dispositivo físico, iPhone ou iPad.

Para que possamos executar a aplicação diretamente em um dispositivo, temos que plugar o aparelho no computador através do cabo de alimentação.

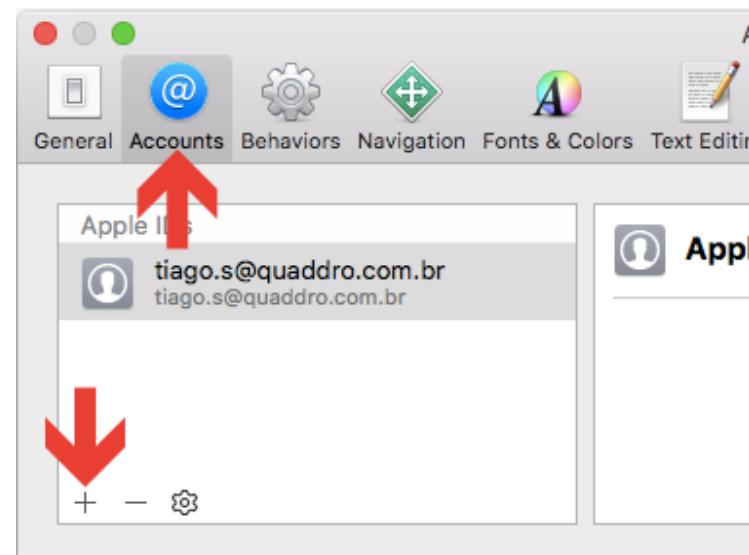
Além do dispositivo, seu Apple ID deve estar inscrito no programa de desenvolvimento, que pode ser feito de forma gratuita em <https://developer.apple.com/account>.

Feito isso, devemos adicionar nossa conta dentro do Xcode:

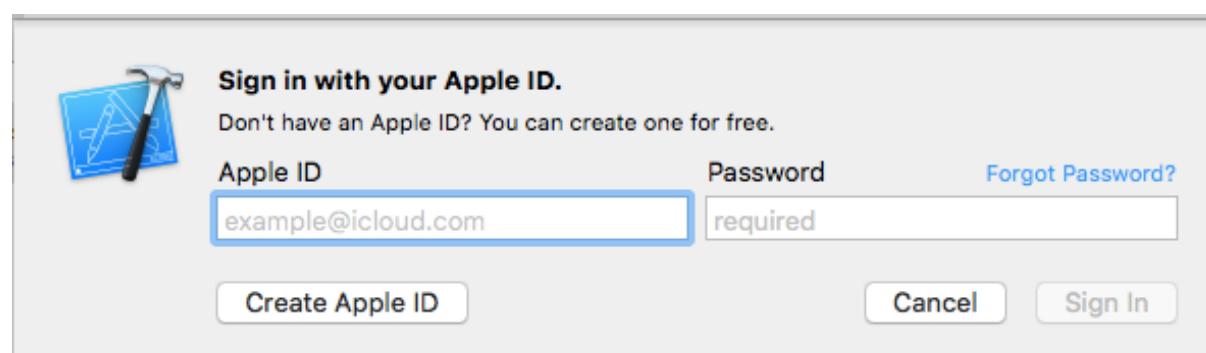
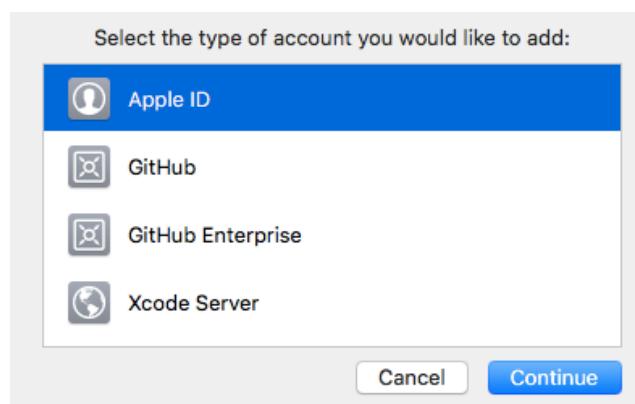
1. Acesse o menu **Xcode**, e em seguida a opção **Preferences**;



2. Na janela exibida, vamos para a guia **Accounts**. Nela encontraremos o botão para adicionar (+);



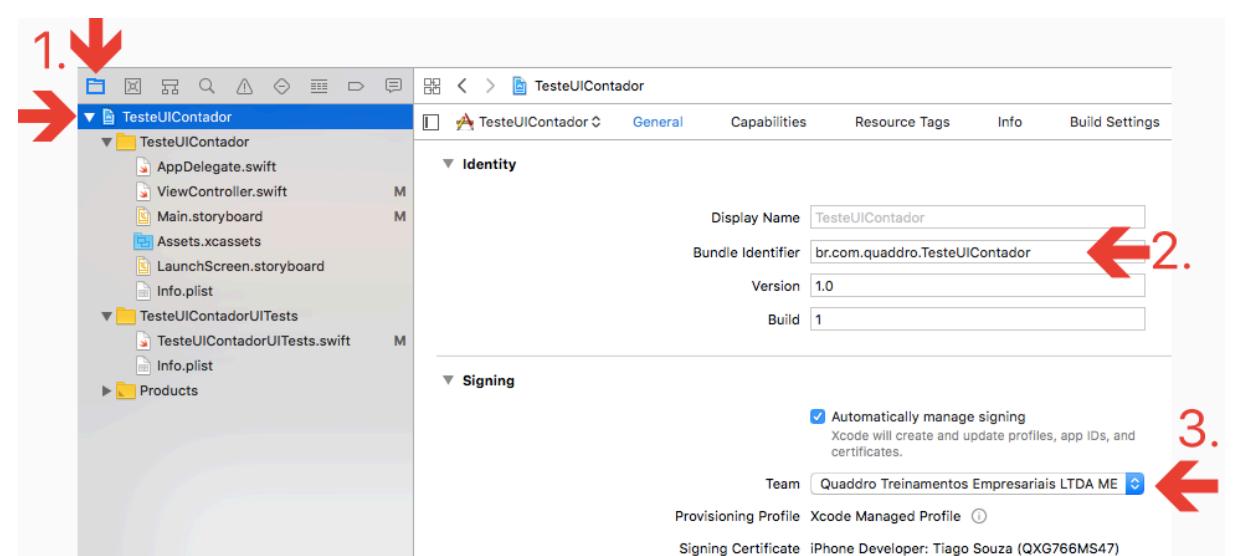
3. Na caixa exibida, escolha a opção **Apple ID**, e entre com os dados da sua conta.



Criando um certificado para a aplicação

Com a conta inserida no Xcode, podemos gerar um certificado para nossa aplicação, e em seguida fazer um Run dela no nosso dispositivo. Vamos aos passos:

1. Em **Project Navigator**, vamos escolher o arquivo que presenta o **Bundle** da aplicação;
2. Em Bundle Identifier defina um identificador para o pacote;
3. Em Team escolha a sua conta.

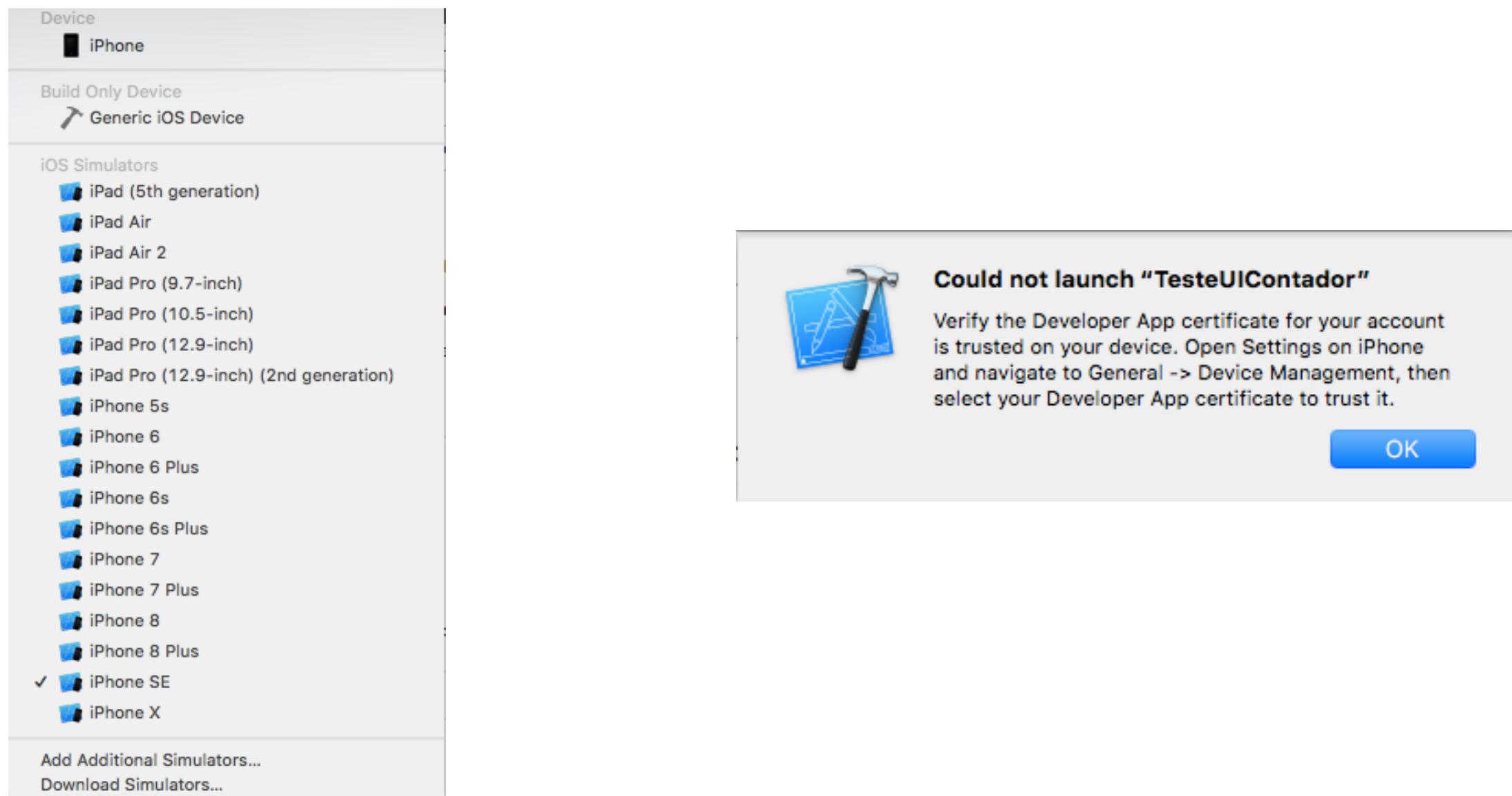


Ao término do processo em certificado será criado para a aplicação. É importante salientar, que para contas gratuitas os certificados são válidos por 7 dias.

Rodando a aplicação no dispositivo

Após conectar um dispositivo ao computador, e gerar o certificado para a aplicação, podemos utilizar o dispositivo ao invés do simulador.

O dispositivo plugado ficará disponível como o primeiro elemento da lista de possíveis meios de saída, junto aos simuladores. Basta escolher o dispositivo, rodar a aplicação, e aguardar o resultado na tela. Algumas aplicações exigirão que uma permissão seja dada pelo usuário do sistema. Basta proceder conforme a imagem indicar.



Seção 1-2

Debugging



Um dos itens mais importantes no desenvolvimento é o **Debug**, que é o processo de desvendar as possíveis causas de quebra ou erros na aplicação.

O Xcode conta com ferramentas de debugging em tempo real. Dentre essas ferramentas, um grande aliado é a já conhecida área de console.

Enquanto estamos entrando com códigos, a leitura do que fazemos é realizada, e caso o sistema encontre alguma discrepância, são passados alguns tipos de alertas para o programador, que auxiliam na busca por soluções.

Em outros casos, erros podem acontecer somente quando rodamos a aplicação. Nesse caso não ocorrerá uma correção automatizada por parte do Xcode, acontecerá apenas um aviso na região onde a aplicação está quebrando.

Outro passo importante do debug, é a quebra da sequência do código em seções. Essas quebras são chamadas de **Breakpoints**.

Tipos de alertas

Podemos classificar os alertas em dois tipos:

- **Alertas de Prevenção:** São indicados na cor amarela. Geralmente fazem sugestões de comportamento relacionados a forma de utilização de partes do código. Esses tipos de alerta não impedem que a aplicação seja compilada e executada. Quando clicamos sobre um alerta de prevenção, uma sugestão é indicada:

```
let constante = "Ainda não utilizei"
```

⚠ Initialization of immutable value 'constante' was never used; ✖ consider replacing with assignment to '_' or removing it

Replace 'let constante' with '_'

Fix

- **Alertas de Erro de Compilação:** São indicados na cor vermelha. Podem tratar tanto de um erro onde não se encontra alguma solução dentro core (indicados por uma exclamação - !), ou indicar alguma solução, quando for encontrada. Esse tipo de alerta impede que a aplicação seja executada.

```
let numero = Numero()
```

⚠ Use of unresolved identifier 'Numero'

```
@IBOutlet weak var labelValor: UILabel
```

● @IBOutlet property has non-optional type 'UILabel'✖

Add '?' to form the optional type 'UILabel?'

Add '!' to form the implicitly unwrapped optional type 'UILabel!'

Fix Fix

Breakpoints e Breakpoint Navigator

Um **Breakpoint** é um mecanismo para pausar uma aplicação em um determinado ponto durante a execução. É uma ferramenta muito útil no sistema de debugging.

Utilizamos um breakpoint para verificar o estado das variáveis e o fluxo de controle da aplicação.

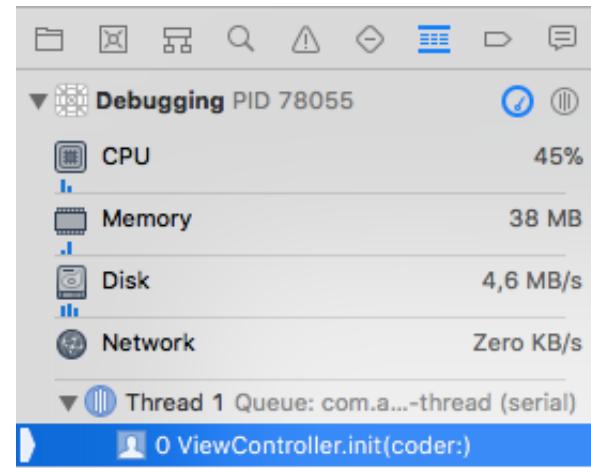
Para adicionar um breakpoint, basta dar um clique sobre o número da linha que deseja utilizar como ponto de pausa:

```
12  
13 var arrayNumeros = ["um", "dois", "três", "quatro"]  
14
```

Quando a aplicação for executada, acontecerá uma pausa na execução utilizando o código da linha indicada:

```
12  
13 var arrayNumeros = ["um", "dois", "três", "quatro"]  
14
```

Outro ponto a destacar, quando estamos depurando com breakpoints é o painel Breakpoint Navigator, que é acionado quando o sistema pausa na primeira ponto encontrado. Através desse navegador podemos ter uma ideia de das porcentagens dos recursos disponíveis, como CPU, memória, armazenamento e recursos de internet:



Podemos desabilitar um breakpoint com um clique. Nesse caso o ícone apresentará uma coloração mais fraca, indicando que o ponto está desabilitado. Quando desabilitado, um ponto passa a ser desprezado quando a execução ocorrer.

```
12  
13 var arrayNumeros = ["um", "dois", "três", "quatro"]  
14
```

Caso queira remover um breakpoint, basta arrastá-lo para qualquer região dentro do código.

Praticando o debugging

Como exemplo prático, vamos utilizar um **array** de strings e comandos **prints** para captar diferentes momentos nas pausas da aplicação com breakpoints.

Vamos criar uma projeto **Single View App**, e na classe viewController adicionar os seguintes códigos e breakpoints:

```

9 import UIKit
10
11 class ViewController: UIViewController {
12
13     var arrayNumeros = ["um", "dois", "três", "quatro"]
14
15     override func viewDidLoad() {
16         super.viewDidLoad()
17
18         print(arrayNumeros)
19
20         for numero in 0...4{
21
22             print(arrayNumeros[numero])
23         }
24     }
25 }
```

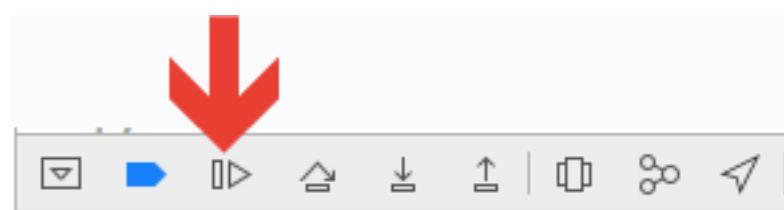
Ao executar a aplicação, acontecerá a primeira pausa na linha **13**:

```

12
13     var arrayNumeros = ["um", "dois", "três", "quatro"]
14
```

A linha é destacada na **cor verde**, indicando que não ocorreram erros, e que a aplicação pode continuar sendo executada até o próximo breakpoint.

Utilizamos o botão **Continue program execution**, localizado na barra de ferramentas acima do console para dar continuidade ao processo:



12

Avance o processo até que o sistema atinja o próximo breakpoint:

```

19
20     for numero in 0...4{ Thread 1: breakpoint...
21
22         print(arrayNumeros[numero])
23     }
24 }
25 }
```

self = (Debugging.ViewController) 0x00007fc...
numero = (Int) 0

["um", "dois", "três", "quatro"]
(lldb)

Observe que em console foi exibido o primeiro **print** solicitado. Como colocamos o segundo breakpoint sobre uma linha sem códigos, o sistema pausará a execução no primeiro código que encontrar pela frente, no nosso caso um **for**.

Como esse **for** depende no nosso **array**, e em ambos os casos temos um **breakpoint**, cada vez que avançarmos a execução, acontecerá um laço do for.

```

Debugging Thread 1 0 ViewController.arrayNumeros.getter
self = (Debugging.ViewController) 0x00007fc...

```

["um", "dois", "três", "quatro"]
um
dois
três
quatro
(lldb)

Porém, temos um erro no nosso código, e esse erro causa uma **exceção fatal**, ao tentar acessar um **índice vazio** em um array. Para o **for**, foi indicado um intervalo de **0 até 4**, onde teríamos cinco posições, mas nosso array possui apenas **quatro posições**.

```
19
20     for numero in 0...4{
21
22         print(arrayNumeros[numero])
23     }    ⚡ Thread 1: Fatal error: Index out of range ✘
24 }
```

Erros como esse são comuns no dia a dia da programação, e quando trabalhamos com o debugging, temos a disposição meios mais fáceis de identificar e resolver esses problemas.

Capítulo 2

Protocolos



Seção 2-1

Protocolos



Protocolos funcionam como um guia a ser seguido por uma classe, estrutura ou enumeração, determinando quais métodos e propriedades serão necessários para a sua concepção.

A classe, estrutura ou enumeração deverá estar em conformidade com o protocolo para que não aconteçam erros de compilação, ou seja, o protocolo realmente determinará quais tarefas deverão ser completadas dentro desses padrões. Vamos a sintaxe:

```
protocol NomeDoProtocolo {  
    //Definições do Protocolo  
}
```

Podemos utilizar múltiplos protocolos em um mesmo processo e quando trabalhamos com classes podemos, caso seja necessário, determinar qual será nossa superclasse e também a utilização de múltiplos protocolos:

```
class NomeDaClasse: NomeDaSuperClass,  
PrimeitoProtocolo, OutroProtocolo {  
  
    //Definições da Classe atendendo aos protocolos  
}
```

A tipagem das propriedades também deve estar em conformidade com os protocolos:

```
protocol Pao {  
    var nome : String {get set}  
    var farinha : Int {get set}  
    var fermento : Int {get set}  
    var leite : Int {get set}  
}  
  
struct DeLeite : Pao {  
    var nome : String  
    var farinha : Int  
    var fermento : Int  
    var leite : Int  
}  
  
let paoDeLeite = DeLeite(nome: "da vovó", farinha:  
200, fermento: 2, leite: 1)  
  
print("Na receita de pão de leite \(paoDeLeite.  
nome) teremos \(paoDeLeite.farinha) gramas de  
farinha, \(paoDeLeite.fermento) colheres de fermento  
e \(paoDeLeite.leite) copo de leite")  
  
//Resultado impresso em console: Na receita de pão  
de leite da vovó teremos 200 gramas de farinha, 2  
colheres de fermento e 1 copo de leite
```

Seção 2-2

Protocolos recorrentes



Muitas das classes que utilizaremos daqui em diante utilizarão alguns protocolos para determinar alguns métodos de execução.

Os nomes mais comuns que veremos pela frente são:

- **Data Source:** Um tipo de protocolo, que quando adotado, fornece base de dados para objetos como listas;
- **Delegate:** Um tipo de protocolo, que quando adotado, fornece informações sobre a interação com elementos de interface.

É bom termos em mente, que alguns dos protocolos adotados terão métodos obrigatórios a serem inseridos para que aconteça a conformidade com o protocolo. No Xcode 9 temos um alerta de erro de compilação que nos auxilia a implementar esses métodos:

```
class ViewController: UIViewController, UITableViewDataSource {  
     Type 'ViewController' does not conform to protocol 'UITableViewDataSource'   
}  
Do you want to add protocol stubs? 
```

Capítulo 3



Extensões

Seção 3-1

Extensões



Extensões adicionam novas funcionalidades para classes, estruturas ou enumerações já existentes, dando a possibilidade de **estender** seus métodos e propriedades sem que aconteça uma sobreposição do original.

Os tipos de classes já existentes na **Swift** como **String**, **Date**, **Int**, **Double**, **Float**, entre outros, também podem ser ampliados para realizarem novas funcionalidades com as extensões.

Além dos tipos existentes, podemos utilizar a palavra **extension** aos tipos desenvolvidos pelo programador. Veja o exemplo abaixo:

```
//Definição da classe programada
class Produto {
    var nome = String()
    var preco = Float()
}

//Estendendo a Classe
extension Produto{
    func mostrar() {
        print("Nome do Produto: \(nome)")
        print("Preço do Produto: R$ \(preco)")
    }
}
```

```
//Instância da Classe
var leite = Produto()
leite.nome = "Da Fazenda"
leite.preco = 3.99
leite.mostrar()
```

```
//Resultado impresso em console:
Nome do Produto: Da Fazenda
Preço do Produto: R$ 3.99
```

#Dica!

Novas **propriedades** a serem acrescentadas em uma extensão funcionarão apenas como **somente leitura**. Caso queira acrescentar novas propriedades a uma classe que tenham como premissa a entrada de novos dados, utilize a classe desejada como **herança**.

Extensões de tipos existentes



Com **extension** é possível ampliar os tipos já existentes na Swift, como Double, Int, String, Date, entre outros, elaborando, por exemplo, estruturas que calculam e retornam valores, ou trabalhar com acréscimo de informações

Primeiro vamos trabalhar com a extensão da classe **String** da Swift, dando a ela a função de duplicar uma string:

```
extension String{
    var maiusculo : String{
        return self.uppercased()
    }
}

var teste = String()

teste = "teste de maiúsculo"

print(teste.maiusculo)

//Resultado impresso em console: TESTE DE MAIÚSCULO
```

Agora vamos adicionar propriedades ao **Double**, ampliando suas funções para realizarem conversões de temperatura:

```
extension Double{
    var fahrenheitToCelsius : Double{
        return (self - 32.00) / 1.8000
    }

    var celsiusToFahrenheit : Double{
        return (self * 1.80) + 32.00
    }

    let temp1Fahrenheit : Double = 212
    let temp2Celsius : Double = 100

    print(temp1Fahrenheit.fahrenheitToCelsius)
    //Resultado impresso em console: 100.0

    print(temp2Celsius.celsiusToFahrenheit)
    //Resultado impresso em console: 212.0
```

Seção 3-3

Extensão para adoção de protocolos



Com as declarações de extensão podemos adicionar conformidades com protocolo a uma classe, estrutura ou enum.

As declarações de extensão não podem adicionar heranças nas classes porém, podemos especificar uma lista de protocolos após o nome do tipo e dois pontos:

```
extension ElementoParaEstender : ProtocolosAdotados {  
    //Declarações  
}
```

Estender elementos é para adoção de protocolo, e consequentemente colocar esse elemento em conformidade com o referido protocolo, consiste em uma boa prática de codificação. Com isso mantemos os métodos relacionados, agrupados com o protocolo de origem.

```
class ViewController : UIViewController {  
    //Declarações  
}  
  
// MARK: - Métodos de UITableViewDataSource  
extension ViewController : UITableViewDataSource {  
    // Declarações dos métodos  
}  
  
// MARK: - Métodos de UIScrollViewDelegate  
extension ViewController : UIScrollViewDelegate {  
    // Declarações dos métodos  
}
```


Capítulo 4



UIResponder

Seção 4-1

UIResponder



Herança: NSObject > UIResponder

A classe **UIResponder** é responsável pela interação entre os objetos e o usuário final da aplicação. Basicamente ela define um meio para que os objetos possam responder a diferentes ações e lidar com eventos.

Objetos como buttons, sliders, segmented controls e outros que veremos adiante podem utilizar os eventos provenientes de **UIResponder** para interagir no sistema.

Podemos separar esses eventos em dois conjuntos distintos: Eventos de **toque** e eventos de **movimentação** (**touch** e **motion**). A princípio vamos trabalhar apenas com eventos de toque.

Gerenciando a cadeia de resposta

Nessa etapa vamos tratar como será o foco de interação com os objetos presentes na nossa janela.

Temos a disposição métodos e propriedades que podem ser implementados com a utilização de dot syntax (.), como veremos a seguir:

```
func becomeFirstResponder() -> Bool  
//Método que indica foco para o objeto
```

```
var canBecomeFirstResponder : Bool { get }  
//Propriedade que responde se um objeto pode entrar em foco
```

```
func resignFirstResponder() -> Bool  
//Método que retira o foco de um objeto
```



```
var canResignFirstResponder: Bool { get }  
//Propriedade que responde se um objeto pode sair de foco
```


Capítulo 5



Trabalhando com textos

Seção 5-1

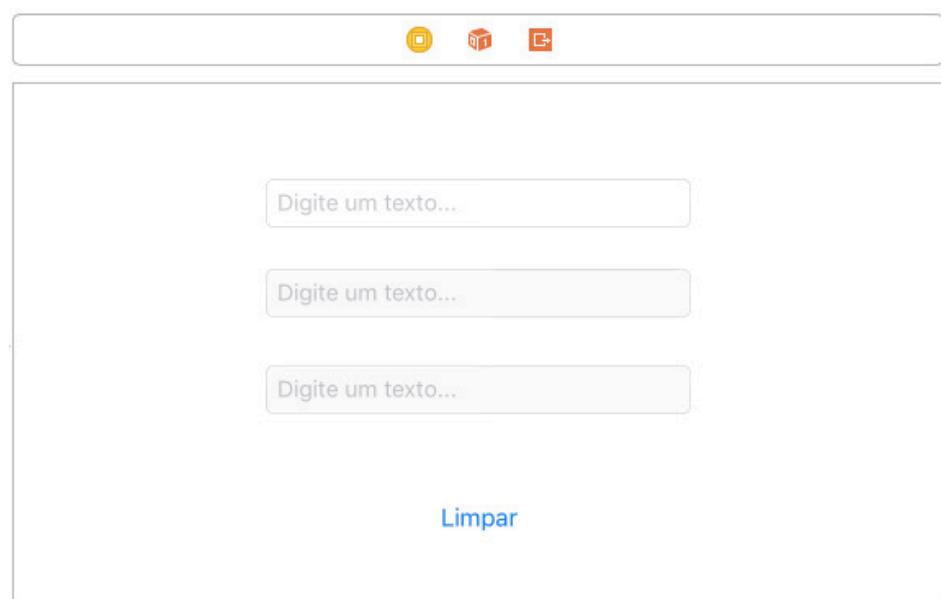
UITextField



Herança: NSObject > UIResponder > UIView > UIControl > UITextField

Text
Text Field - Displays editable text and sends an action message to a target object when Return is tapped.

A classe **UITextField** é responsável por permitir a entrada de textos a partir da tela do dispositivo. Devemos utilizar esse componente quando o usuário precisa digitar pequenas quantidades de texto para utilização dentro da aplicação.



De forma similar a outros componentes, a **UITextField** possui seu **delegate (Protocolo UITextFieldDelegate)**, que é responsável por receber eventos relacionados ao seu comportamento.

A classe **ViewController** que receberá **text fields**, poderá adotar o protocolo **UITextFieldDelegate** para que os recursos de **delegate** fiquem disponíveis, e dessa forma possamos captar as interações com os campos.

```
// MARK: - Métodos de UITextFieldDelegate  
extension ClasseViewController : UITextFieldDelegate {  
    // Declarações dos métodos  
}
```

Principais propriedades

Propriedade	Tipo	Descrição
delegate	UITextFieldDelegate	Responde a edição relacionada ao campo de texto.
isEnabled	Bool	Define / Retorna se a caixa de texto está habilitada ou não.
text	String?	Define / Retorna o texto da TextField.
placeholder	String?	Define / Retorna um breve texto que é esperado para a caixa de texto.

Interações de um campo com o teclado

Quando tocamos um **text field**, automaticamente o objeto ganha foco, e o teclado do dispositivo se torna aparente e pronto para interagir com o campo.

Podemos forçar um campo de texto para se tornar o primeiro que entre em foco, chamando o método **becomeFirstResponder**, visto no capítulo anterior.

Podemos pedir ao sistema para dispensar o teclado chamando o método **resignFirstResponder** para o campo de texto. Normalmente, dispensamos o teclado em resposta às interações específicas, como um toque ou em resposta a tecla **Enter/Return**.

O sistema também pode dispensar o teclado em resposta a ações do usuário, como tocar diferentes áreas da tela por exemplo.

Métodos de Interação UITextFieldDelegate

Podemos controlar as interações do usuário com a área do **text field**, como quando a caixa de texto é tocada e se inicia a digitação, quando a caixa de texto é deixada de lado e perde o foco de digitação ou quando a tecla **Return** é utilizada.

```
//Método disparado quando um text field entra em foco
optional func textFieldDidBeginEditing(_ textField:
UITextField) {
    //Declarações
}
```

```
//Método disparado quando um text field perde o foco
optional func textFieldDidEndEditing(_ textField:
UITextField) {
    //Declarações
}

//Método disparado quando acontecem alterações dentro de um
text field
optional func textField(_ textField: UITextField,
shouldChangeCharactersIn range: NSRange, replacementString
string: String) -> Bool {
    //Declarações
    return true
}

//Método disparado quando a tecla Return é pressionada
optional func textFieldShouldReturn(_ textField: UITextField)
-> Bool {
    //Declarações
    return true
}
```

Seção 5-2

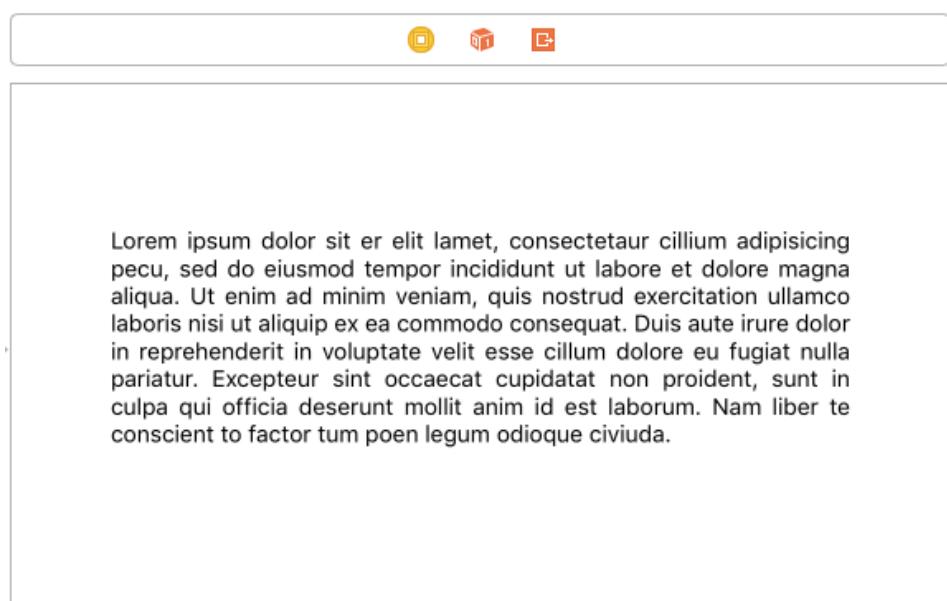
UITextView



Herança: NSObject > UIResponder > UIView > UIScrollView > UITextView



UITextView é responsável por exibir caixas de texto que podem ou não ser editadas pelo usuário final. A edição, quando possível, poderá ser feita através de um duplo toque na caixa de texto, ou o texto poderá ser inserido e travado conforme a necessidade do programador.



Placeholder
Lorem ipsum dolor sit er elit lamet, consectetaur cillum adipisicing
pecu, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco
laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor
in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla
pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim id est laborum. Nam liber te
conscient to factor tum poen legum odioque civiuda.

Outro ponto importante a ser destacado, é a possibilidade de deixarmos o texto indicado na caixa como selecionável ou não.

Caso a quantidade de texto exceda a caixa indicada, automaticamente teremos a disposição uma barra de rolagem e a possibilidade de aplicar um scroll no texto.

Assim como o UITextField, o **UITextView** possui seu **delegate** (Protocolo **UITextViewDelegate**) que é responsável por receber eventos relacionados ao seu comportamento, tais como um toque dentro da caixa de texto, início de digitação ou o pressionamento do Enter/Return.

A classe a **ViewController** que receberá **text views**, deverá implementar o protocolo **UITextViewDelegate** para que os recursos de **delegate** fiquem disponíveis.

```
extension ClasseViewController : UITextViewDelegate {  
    //Declarações dos métodos  
}
```

Principais Propriedades

Propriedade	Tipo	Descrição
delegate	UITextFieldDelegate?	Responde a edição relacionada ao campo de texto.
isEditable	Bool	Define / Retorna se a edição de texto está habilitada ou não.
isSelectable	Bool	Define / Retorna se a seleção de texto está habilitada ou não.
text	String!	Define / Retorna o texto da TextField.

```
//Método disparado quando um text view perde o foco
optional func textViewDidEndEditing(_ textView: UITextView) {
    //Declarações
}

//Método disparado quando ocorre alguma alteração em um text view
optional func textViewDidChange(_ textView: UITextView) {
    //Declarações
}
```

Métodos de Interação UITextViewDelegate

Podemos controlar as interações do usuário com a área do **text field**, como quando a caixa de texto é tocada e se inicia a digitação, quando a caixa de texto é deixada de lado ou perde o foco de digitação.

```
//Método disparado quando um text view entra em foco
optional func textViewDidBeginEditing(_ textView: UITextView) {
    //Declarações
}
```

Capítulo 6



Trabalhando com imagens



Herança: NSObject > UIImage

Um objeto **UIImage** faz o gerenciamento de dados de imagem em seu aplicativo. Ou seja, esse tipo de elemento guarda as informações do objeto imagem, como o seu endereço, tamanho, tipo, etc.

A classe **UIImage** controla apenas o objeto que representa uma imagem, e não serve para exibir a imagem na tela propriamente dita.

Para exibirmos uma imagem na tela, precisamos de um objeto **UIImageView**, que herda as propriedades de **UIView**. Veremos esse objeto mais adiante.

Um objeto desse tipo pode conter uma única imagem ou uma sequência de imagens que podemos usar em uma animação..

A Swift suporta os tipos de imagens mais utilizados como **.png**, **.jpg**, **.tif**, **.gif**, etc.

A extensão mais indicada é **.png**, pois sua se torna fácil, desde que a imagem faça parte do **Bundle** (pacote do app), bastando apenas indicar nome do arquivo, sem a necessidade da extensão.

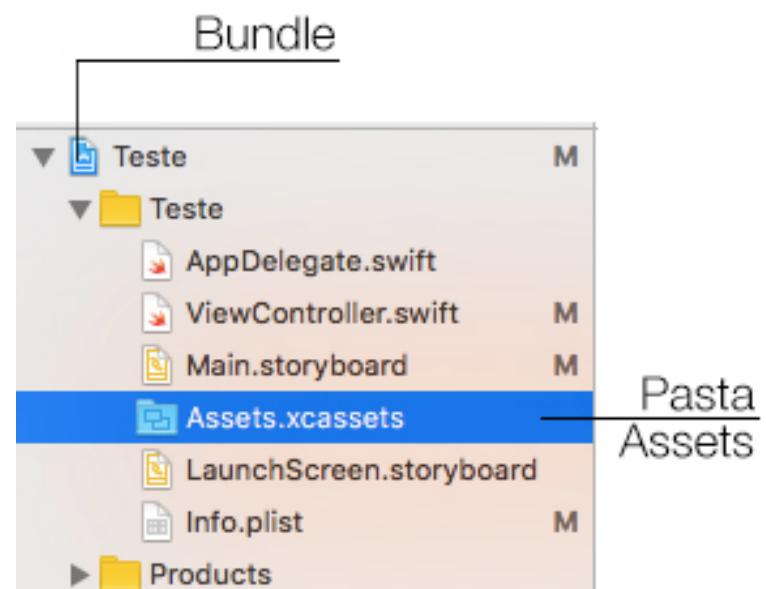
Incorporando um arquivo de imagem ao seu projeto

Podemos deixar um arquivo de imagem solto no bundle, e utilizá-lo a qualquer momento dentro o app, porém essa não é uma prática recomendada.

Quando iniciamos um novo projeto, automaticamente é criada uma espécie de pasta chamada **Assets.xcassets**. Esse espaço é reservado para incorporar arquivos diversos que venham a complementar nosso aplicativo, como imagens por exemplo.

Podemos utilizar esse espaço e nele concentrar as imagens que farão parte do nosso aplicativo.

Para incorporar um arquivo ao pacote do aplicativo, basta arrastar os arquivos desejados para a área do bundle, inclusive a pasta **Assets.xcassets**.



Principais atributos

Propriedade	Tipo	Descrição
imageOrientation	UIImageOrientation	Define a orientação e rotação da imagem.
size	CGSize	Define o tamanho da imagem.
scale	CGFloat	Define a escala da imagem para diferentes tipos de tamanhos de dispositivos.
images	UIImage	Define um array de imagens para a criação de animações.
duration	TimeInterval	Retorna o tempo de duração de uma sequência animada

//Método que busca uma sequência de imagens provenientes de um array. Também define o tempo de duração da animação
animatedImage(with images: [UIImage], duration: TimeInterval)
-> UIImage?

Principais métodos

//Inicia um objeto UIImage com o nome do arquivo fornecido
init?(named name: String) -> UIImage

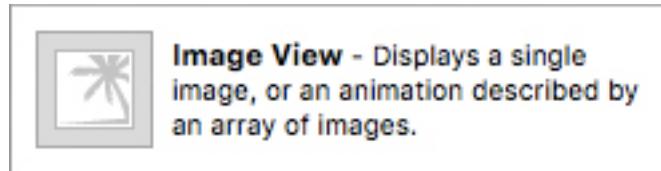
//Método que busca uma sequência de imagens que tenha um mesmo nome como prefixo e números iniciados em 0. Exemplo:
imagem_0, imagem_1, etc. Também define o tempo de duração da animação
animatedImageNamed(_ name: String, duration: TimeInterval)
-> UIImage?

Seção 6-2

UIImageView

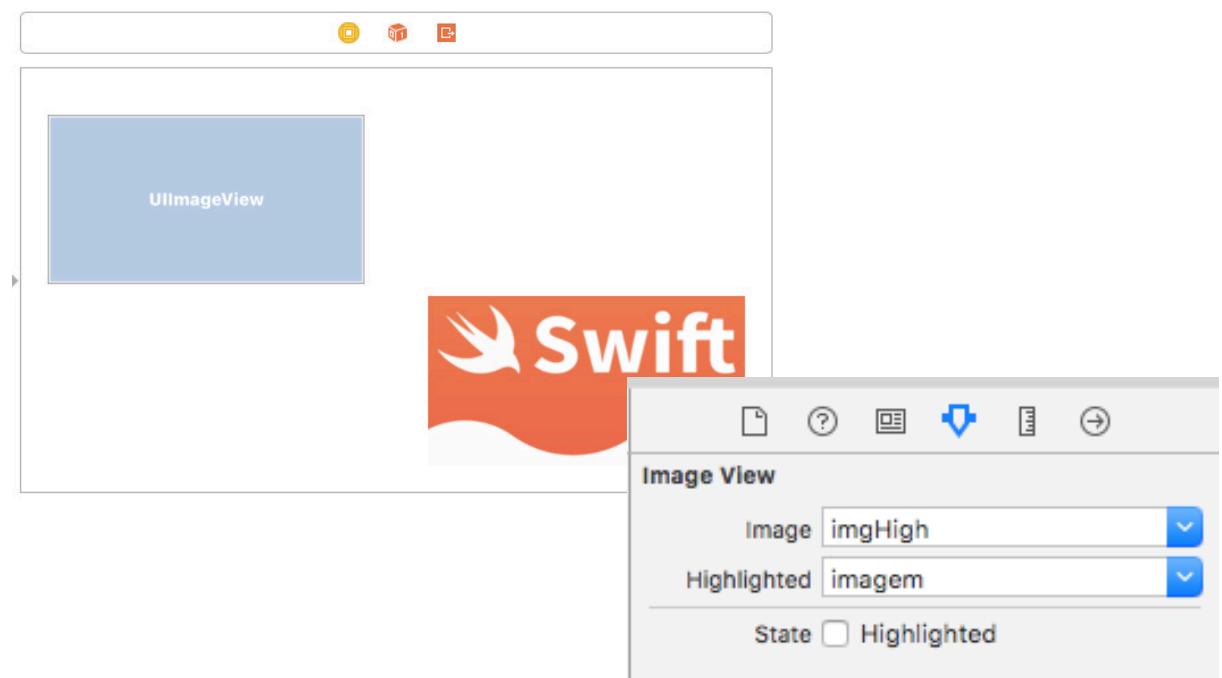


Herança: NSObject > UIResponder > UIView > UIImageView



Um UIImageView é um **container** responsável por exibir um objeto do tipo **UIImage**, que pode representar uma imagem estática ou um conjunto de imagens formando uma animação.

Outra propriedade interessante é escolhermos uma segunda imagem para destaque (**Highlighted**), podendo assim alternar entre duas imagens conforme a necessidade:



Principais propriedades

Propriedade	Tipo	Descrição
image	UIImage?	Define qual objeto UIImage será utilizado
highlightedImage	UIImage?	Define que a imagem de destaque será exibida
animationImages	[UIImage]?	Define qual array de imagens será utilizado
highlightedAnimationImages	[UIImage]?	Define qual array de imagens será utilizado para destaque
animationDuration	TimeInterval	Define o tempo de duração total da animação
animationRepeatCount	Int	Define a quantidade de vezes que a animação será repetida.
isHighlighted	Bool	Define se o objeto está ou não em destaque.

Principais métodos

```
//Método que retorna um objeto UIImageView com o objeto  
UIView fornecido.  
init(image: UIImage?)
```

```
//Inicia a animação de imagens na UIImageView.  
.startAnimating()
```

```
//Para a animação de imagens na UIImageView.  
.stopAnimating()
```

```
//Retorna um booleano indicando se a UIImageView está  
animando as imagens  
.isAnimating()
```

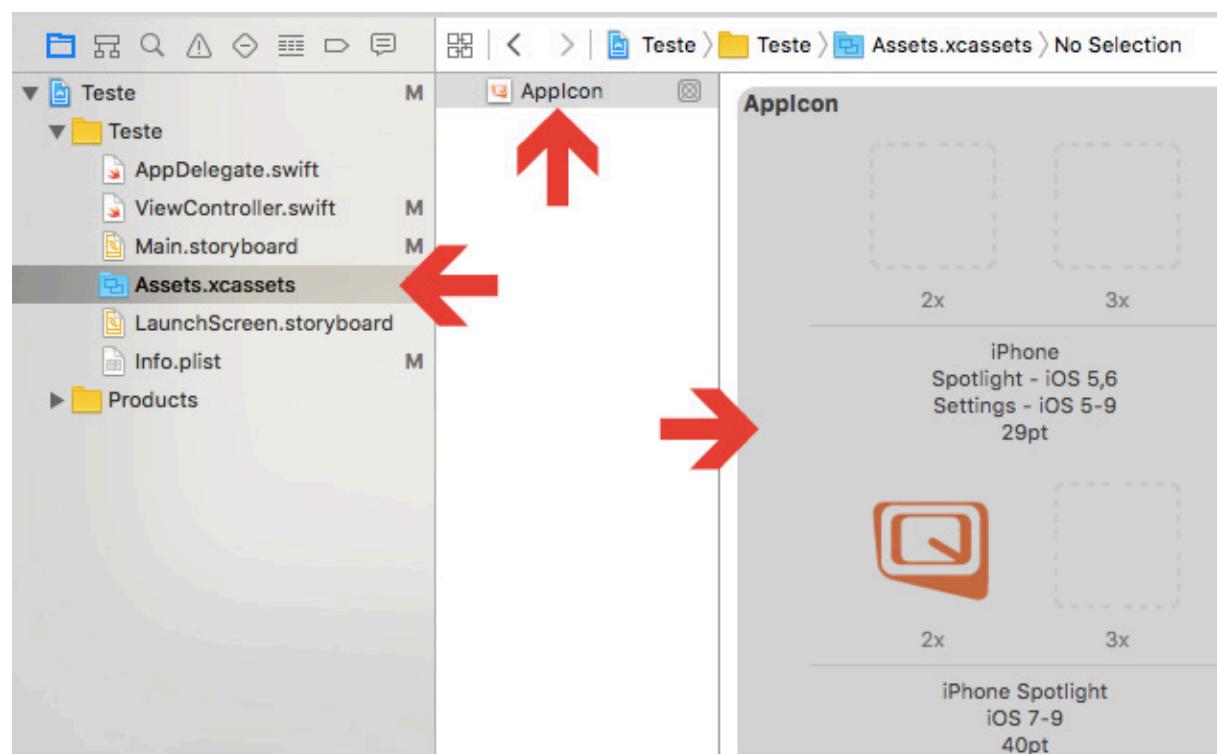
Seção 6-3

Definindo o ícone da aplicação



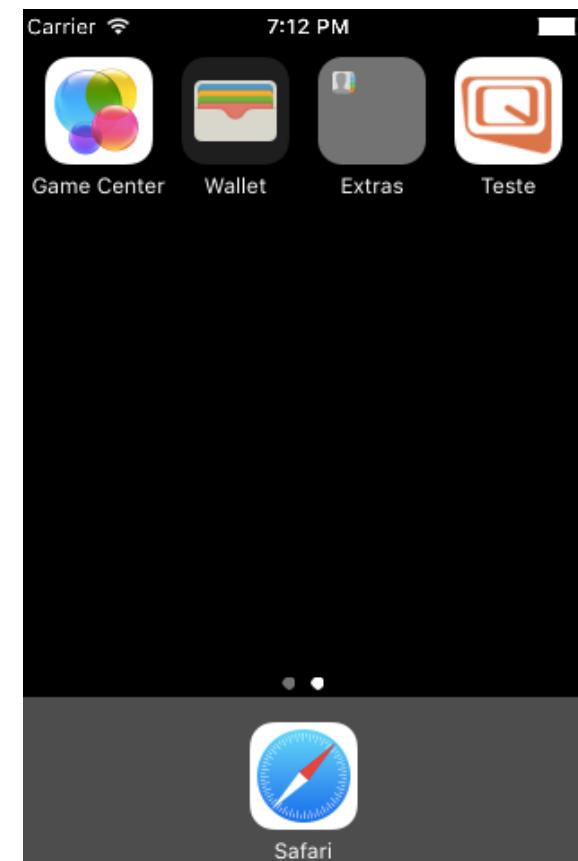
Anteriormente falamos acerca da pasta **Assets.xcassets**. Uma das opções que podemos encontrar dentro dessa pasta é a alocação de uma imagem como ícone da nossa aplicação.

O processo é bem simples: Basta arrastarmos o arquivo de imagem que será o ícone para as opções que estão dispostas de acordo com a versão do iOS e tamanhos dos dispositivos.



#Dica!

As imagens apresentadas deverão respeitar os tamanhos em pixels de acordo com o tamanho e versão do iOS de cada dispositivo.



Capítulo 7



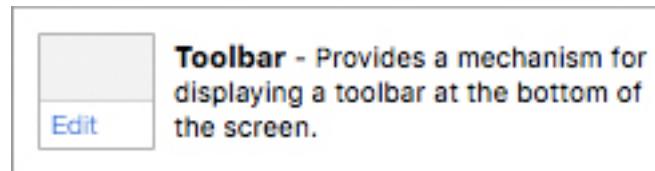
Barra de ferramentas - Toolbar

Seção 7-1

UIToolbar



Herança: NSObject > UIResponder > UIView > UIToolbar



A classe **UIToolbar** controla o objeto de interface responsável por agrupar botões em uma área controlada.

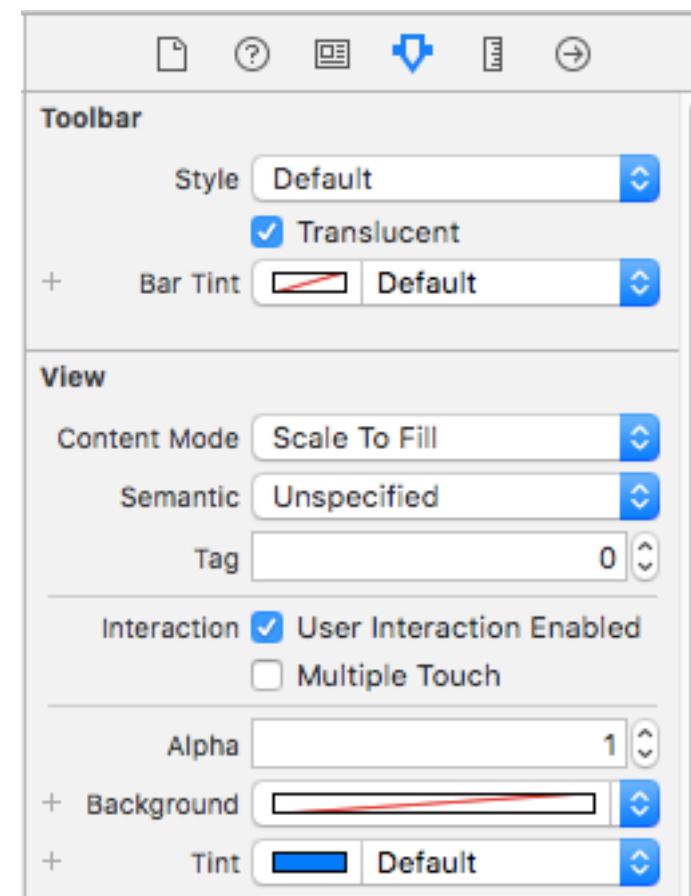
A posição dessa barra flexível pode ser definida tanto na parte superior quanto inferior do layout do aplicativo e é útil para incluir novas funcionalidades de maneira organizada para nossa aplicação.



#Dica!

Quando colocamos um objeto **UIToolbar** no nosso aplicativo, temos a disposição apenas um botão. Veremos adiante como acrescentar mais itens.

Podemos customizar a aparência das tollbars no Attributes Inspector:



Seção 7-2

UIBarButtonItem



Herança: NSObject > UIBarButtonItem > UIBarButtonItem



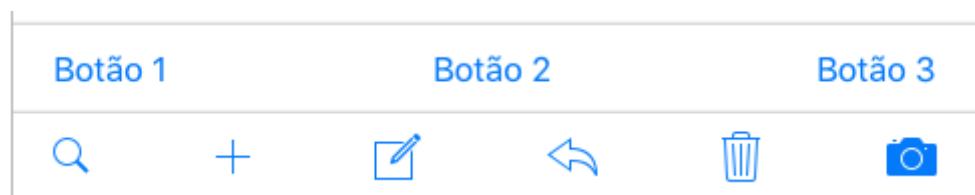
A classe **UIBarButtonItem** controla o objeto de interface responsável por exibir botões específicos para serem inseridos em uma **UIToolbar** ou em uma **UINavigationBar**.

Esse tipo especial de botão também possui uma propriedade chamada que possibilitam a exibição de ícones no lugar dos textos nos títulos.

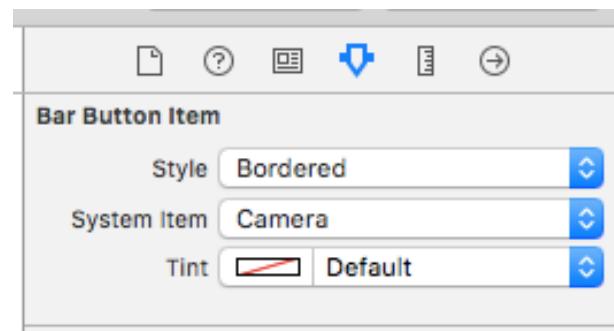
Podemos acrescentar novos botões a nossa toolbar **arrastando e soltando** novos itens dentro dela. Podemos também **alterar a ordem** dos botões selecionando e arrastando o botão desejado para a área a ser utilizada dentro da toolbar.

Utilizando ícones do sistema como botões

Os botões de uma toolbar podem ter títulos com **textos** ou **ícones** disponíveis no sistema.



No painel **Attributes** podemos escolher o estilo (**Style**), tipo de item (**System Item**) e a cor do botão (**Tint**). Em System Item temos a disposição alguns ícones que podem compor os botões.



#Dica!

Os **itens** que compõem uma **toolbar** se comportam de forma semelhante a um **botão (button)**. Podemos codificar esses itens com **Outlets** ou **Actions** conforme a necessidade de diferentes propriedades e métodos, tratando cada um dos itens de **forma independente**.

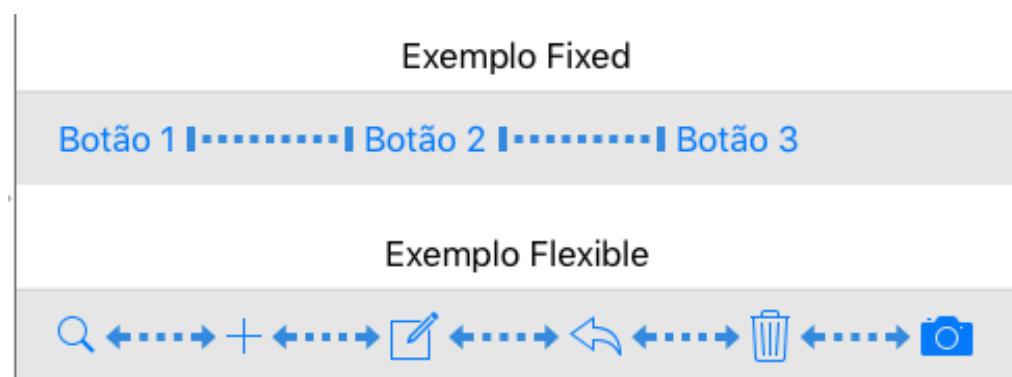
Seção 7-3

Fixed Space e Flexible Space



Fixed Space Bar e **Flexible Space Bar** são dois tipos de **UIBarButtonItem** responsáveis por gerar espaçamento entre botões de uma **UIToolbar** ou em uma **UINavigationBar**.

Com o **Fixed** podemos determinar a distância entre os botões. Já com **Flexible** atua como uma mola distanciando o máximo possível os botões, e quando temos mais de dois botões e utilizamos um **Flexible** entre eles, as distâncias se manterão iguais.



#Dica!

Podemos transformar qualquer botão em **Fixed** ou **Flexible**. Basta selecionar o objeto e escolher a opção no painel **Attributes**, menu drop down **System Item**, conforme vimos na seção anterior.

Capítulo 8



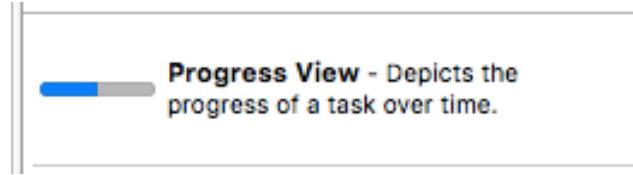
UIProgressView

Seção 8-1

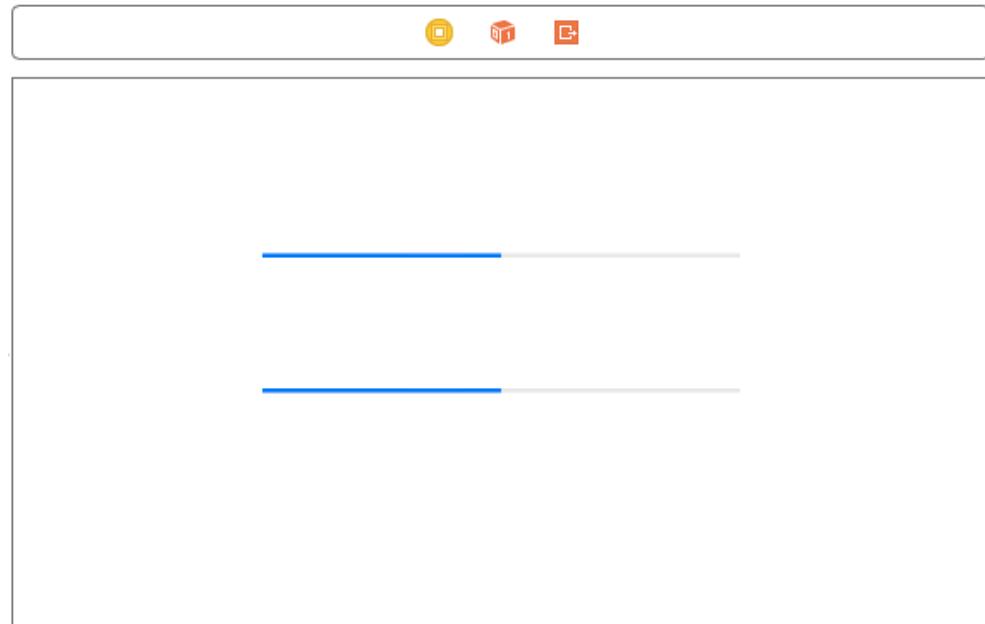
UIProgressView



Herança: NSObject > UIResponder > UIView > UIProgressView

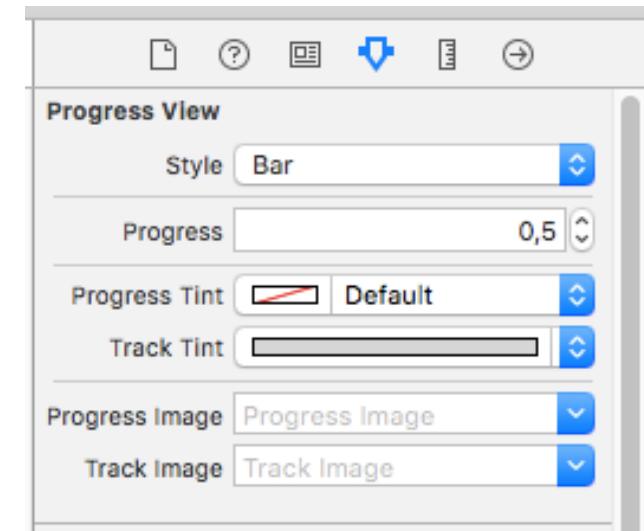


Podemos utilizar a classe **UIProgressView** para criar **barras de progresso**.



Temos a disposição dois estilos diferentes de barras, um mais fino e compacto, e outro um pouco mais aparente.

A barra pode ter a cor dos seus elementos customizadas conforme a necessidade de cada projeto. Imagens também podem ser utilizadas para a formação da aparência da barra.



#Dica!

Os valores de progressão por padrão, variam de 0.0 a 1.0

Principais propriedades

Propriedade	Tipo	Descrição
progress	Float	Define um valor de progressão
observedProgress	Progress	Define um valor de progressão de acordo com um objeto NSProgress

Principais métodos

```
//Inicia e retorna um objeto UIProgressView  
convenience init(progressViewStyle style:  
UIProgressViewStyle)
```

```
//Define um valor de progressão com opção de animar  
.setProgress(_ progress: Float, animated animated: Bool)
```

Capítulo 9



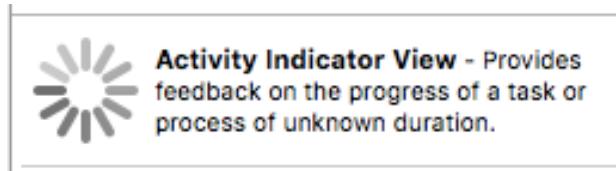
UIActivityIndicatorView

Seção 9-1

UIActivityIndicatorView

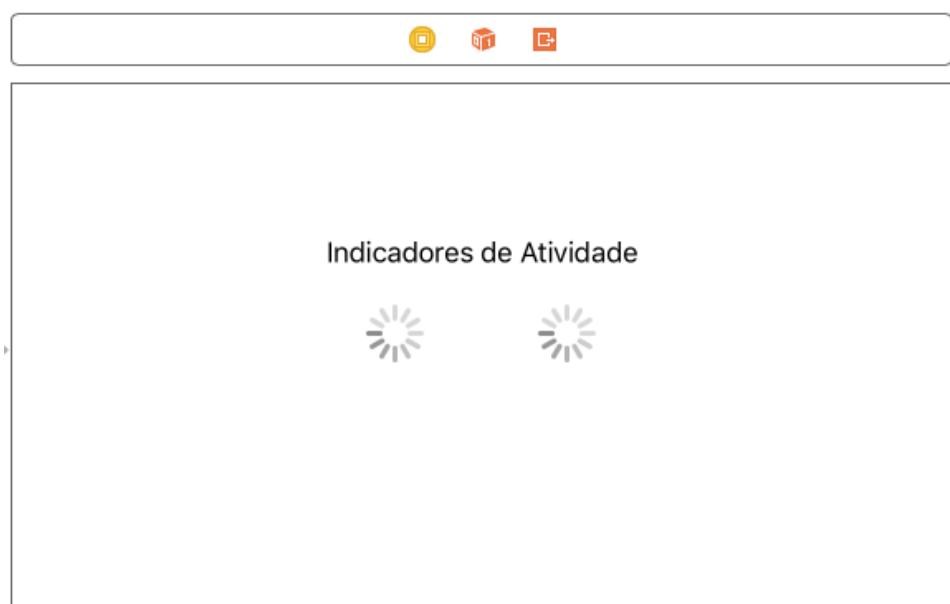


Herança: NSObject > UIResponder > UIView > UIActivityIndicatorView

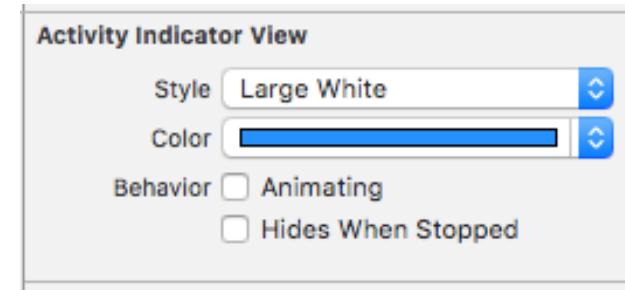


Podemos usar um indicador de atividade (**UIActivityIndicatorView**) para mostrar que uma tarefa está em progresso.

Podemos controlar quando um indicador de atividade será chamado, animado, e quando será ocultado.



Assim como os outros objetos vistos até aqui, podemos trabalhar com a aparência do indicador, modificando suas cores e tamanhos. Também é possível controlar os comportamentos do objeto como a animação e como será exibido.



Principais propriedades

Propriedade	Tipo	Descrição
hidesWhenStopped	Bool	Define o objeto será ocultado quando estiver parado
color	UIColor?	Define a cor base do objeto

Principais métodos

```
.startAnimating()  
//Define o inicio da animação do objeto  
  
.stopAnimating()  
//Defini o fim da animação do objeto  
  
.isAnimating()  
//Retorna se o obejto está animando ou não
```


Capítulo 10

Autolayout



Seção 10-1

Introdução ao Autolayout

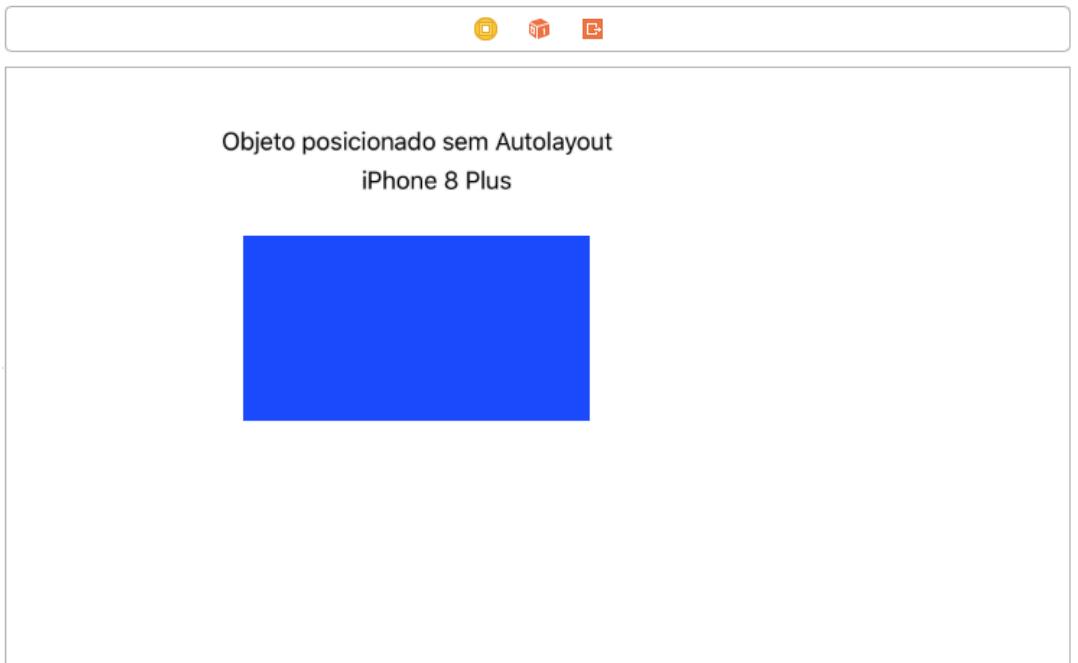


O **Autolayout** é uma ferramenta do **XCode** que permite a criação de relacionamentos entre elementos da interface para que se ajustem a diferentes resoluções em diferentes dispositivos de forma responsiva.

Através dele é possível fazer com que seu aplicativo adapte a interface de acordo com o dispositivo utilizado, evitando problemas de compatibilidade do layout e distribuindo os elementos corretamente na tela.

Quando iniciamos um novo projeto, os elementos lançados respeitam dois pontos fundamentais de inserção: **Dimensões** (comprimento e largura do objeto), e **posicionamento** (coordenadas X e Y).

Quando não impomos um esquema de Autolayout, esses objetos ficam fixos em suas posições sem alterações das suas dimensões.



No exemplo anterior, temos os mesmos objetos dispostos em dispositivos com tamanhos diferentes, um com 4.0 polegadas e outro com 5.5 polegadas. Observe que a dimensão e posição os objetos não foram alteradas nem ajustadas de acordo com a mudança de dispositivo.

O trabalho com **Autolayout** é uma premissa importante no desenvolvimento de aplicativos, é com ele que vamos organizar nossos layouts, inclusive com conceitos de responsividade.

Adiante temos o conceito aplicado, novamente a dois dispositivos com tamanhos distintos:



#Dica!

É importante ter em mente logo no início do projeto quais dispositivos (tamanho e versões o iOS) seu projeto atenderá.



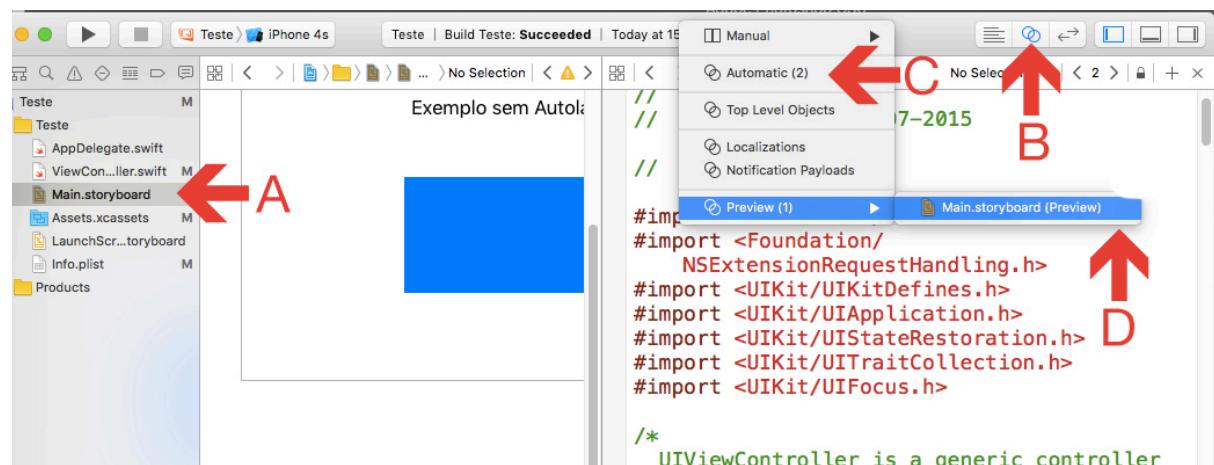
Seção 10-2

Trabalhando com previews

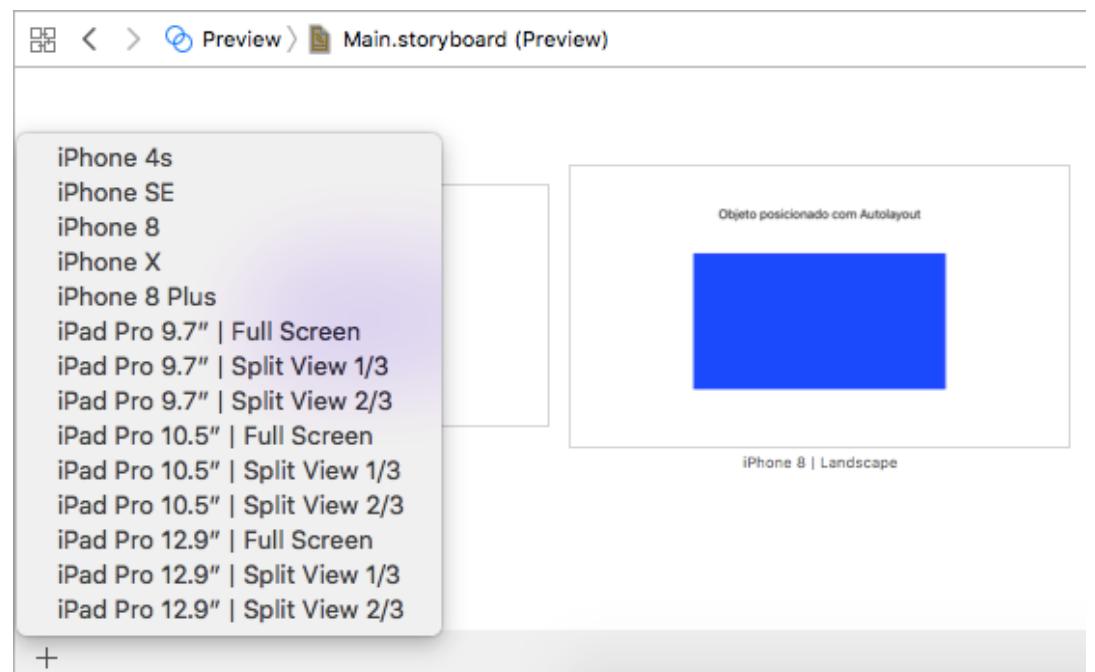


No **Xcode** temos a disposição uma sequencia de **previews** para diferentes tamanhos de dispositivos. É importante salientar que uma quantidade grande de elementos e diferentes janelas de previews podem ocasionar travamentos ou lentidão do Xcode. Vamos ao procedimento:

- A. Escolha o esquema de layout a ser previsto;
- B. Clique no botão Assistant editor;
- C. Clique no menu drop down de previews (Automatic);
- D. Escolha a opção Preview (1), e escolha a opção Main.storyboard (Preview).



Em seguida, podemos acessar o menu com sinal de mais (+), localizado no canto inferior esquerdo da janela de previews e escolher quais dispositivos queremos visualizar:



Seção 10-3 Constraints

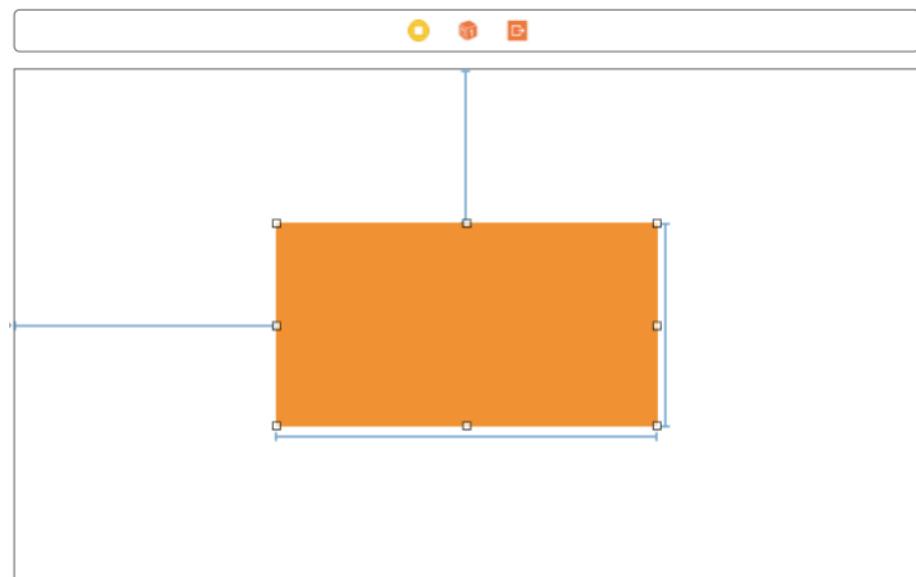


O conceito de **Autolayout** é baseado em **Constraints**, que são regras definidas pelo desenvolvedor para determinar as dimensões e posicionamento dos objetos.

Essas regras descrevem o comportamento do elemento com relação a elementos vizinhos quando existirem, ou com relação a borda ou margens da View Controller.

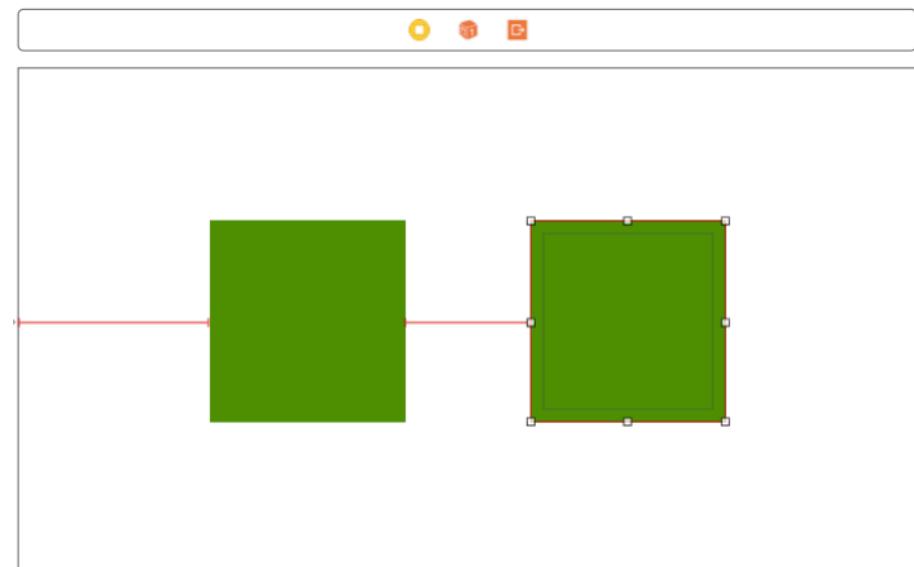
Cada elemento de interface pode ter suas **Constraints** e essas constraints por sua vez, podem referenciar outros elementos.

Temos que ter em mente que precisamos amarrar duas sequências de informações para que as constraints de um objeto funcionem. Devemos amarrar as suas dimensões e seu posicionamento com relação a pontos satisfatórios no layout.

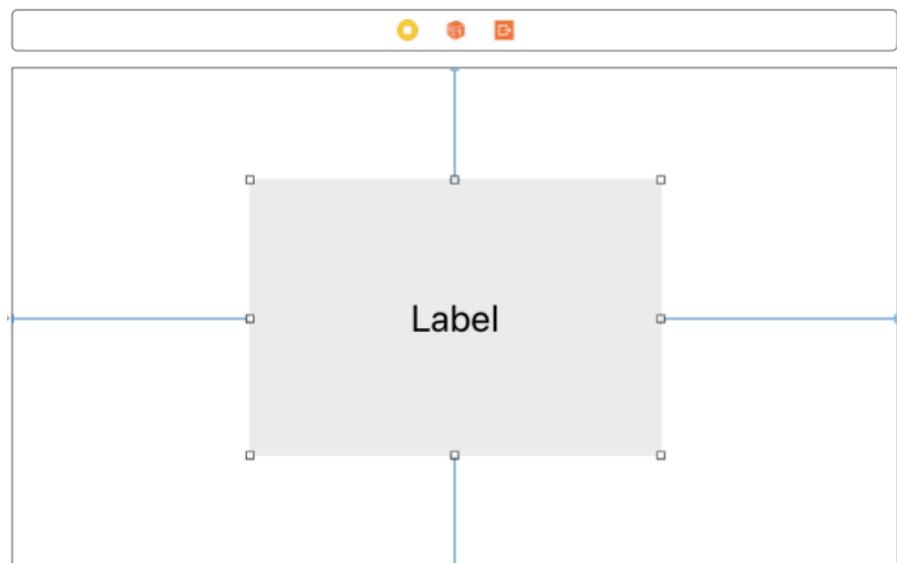


No exemplo anterior, podemos observar que o objeto **View** possui alguns elementos a sua volta que determinam as distâncias da borda da janela, e também as dimensões. Esses objetos são as constraints.

Quando criamos constraints, seus parâmetros serão tomados a partir do objeto imediatamente mais próximo do objeto atual. Se a constraint não encontrar nenhum objeto no caminho, automaticamente a referência será a borda da janela.



Quando trabalhamos com elementos que estão dentro de outros elementos, temos que alocar constraints para ambos:



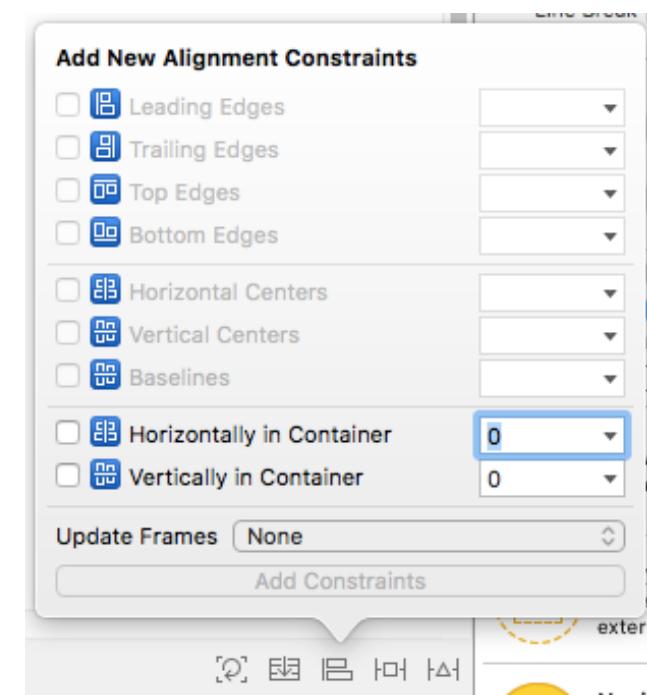
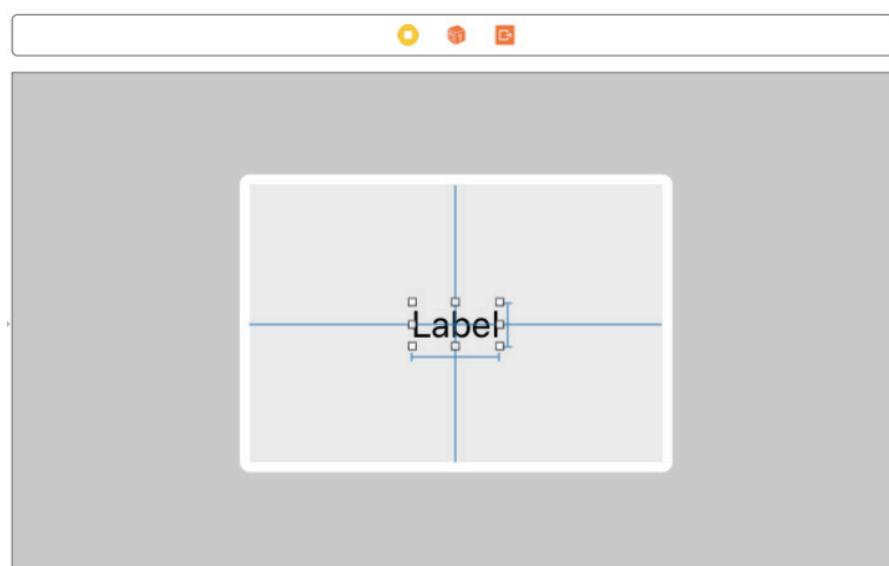
Definindo constraints

No canto inferior direito da janela do nosso **storyboard** temos a disposição os **cinco botões** que cuidam das definições de constraints para os objetos selecionados e outros recursos de Autolayout.

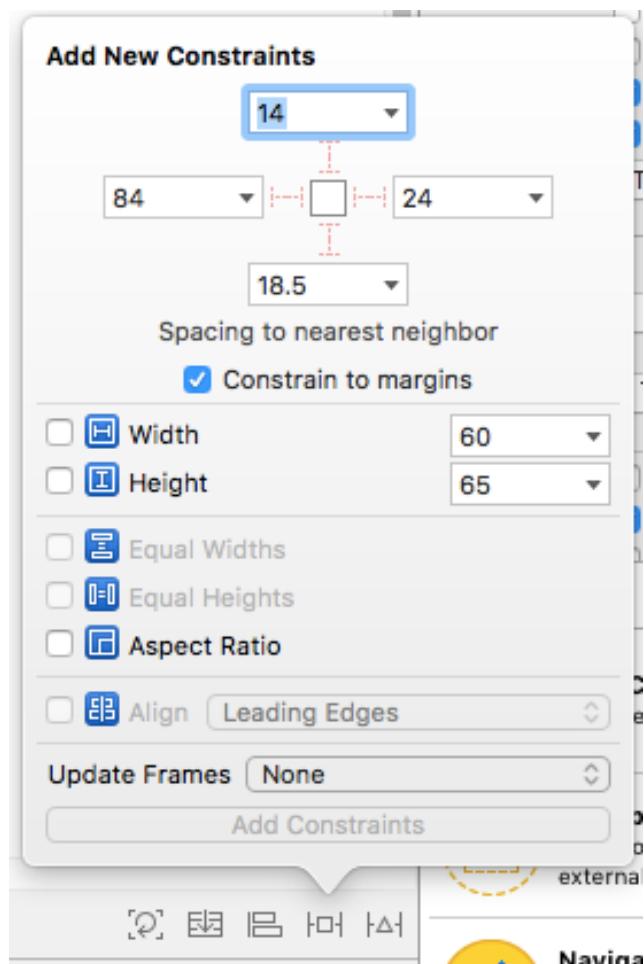


Os botões **Align**, **Pin** e **Resolve Autolayout Issues**, interagem diretamente na criação, alteração e exclusão de uma constraint:

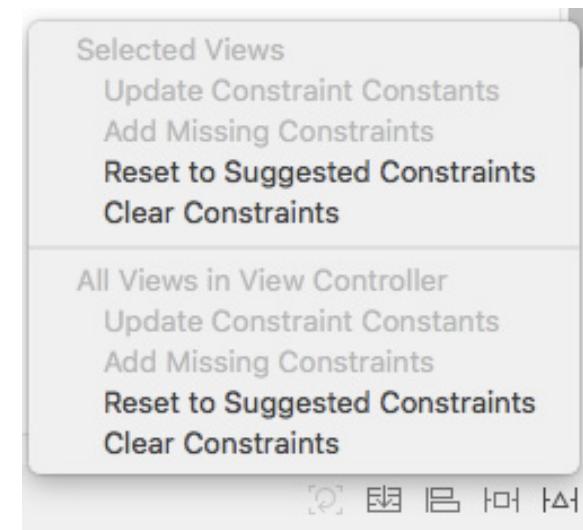
- **Align:** Cria constraints de alinhamento. É utilizado quando se precisa centralizar um objeto em relação ao seu container, ou especificar que um elemento esteja alinhado a esquerda do outro, por exemplo. Depois de escolher as constraints desejadas, devemos clicar no botão **Add Constraints**



- **Pin:** ⌂ Cria constraints de dimensionamento e espaçamento. É usado quando se precisa especificar o tamanho de um objeto, ou indicar deslocamentos no espaço horizontal/vertical. Depois de escolher as constraints desejadas, devemos clicar no botão **Add Constraints**;



- **Resolve Autolayout Issues:** ⌧ Menu que fornece de opções para resolver problemas comuns no Autolayout, ou limpar as constraints. As opções na metade superior do menu afetam apenas os objetos selecionados. As opções na metade inferior afetam todos os objetos de uma mesma View Controller.



#Dica!

A opção **Reset to Suggested Constraints** adiciona constraints automaticamente aos objetos selecionados. Quando aplicado é importante que se confira o resultado em diferentes tamanhos de dispositivos para identificação de possíveis falhas.

Já o recurso **Add Missing Constraints** ajuda a colocar as constraints que estão faltando para completar um layout.

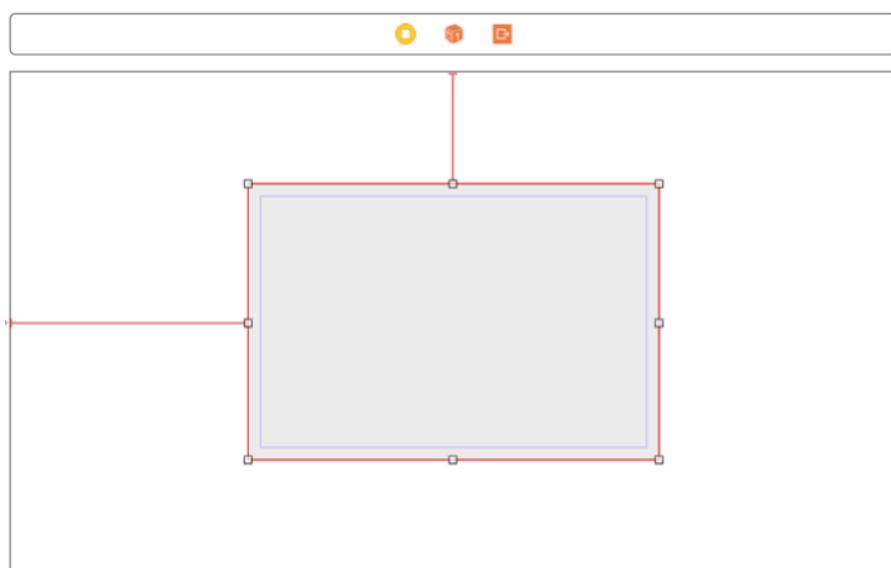
- **Update Frames:** ⌘ Botão que atualiza a interface de acordo com as constraints estabelecidas.

Comportamento das Constraints

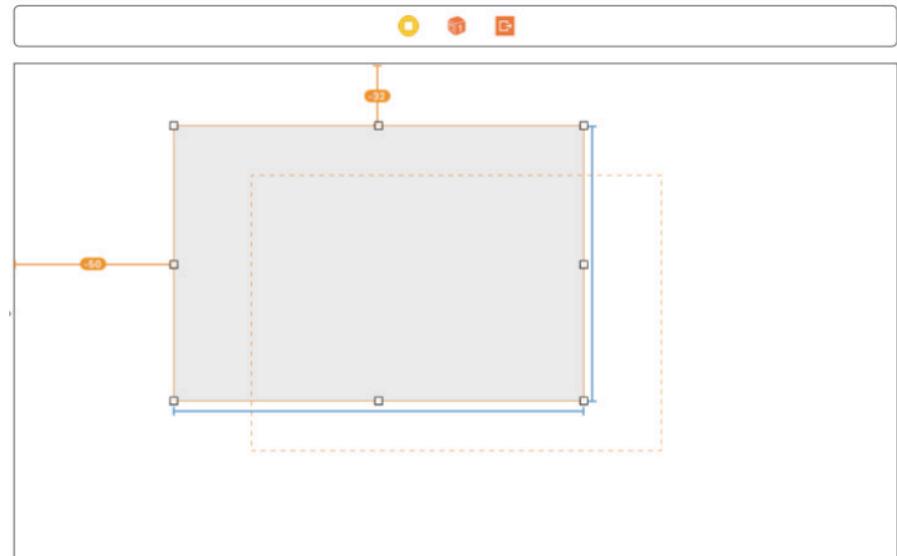
Conforme adicionamos constraints ao nosso projeto, elas se comportam de maneiras diversas, e dessa forma podemos entender quais informações estão faltando para completar o processo de Autolayout.

Basicamente as constraints podem apresentar três cores quando estão sendo criadas e o seu respectivo objeto é selecionado. Cada uma das cores nos revela qual o status atual do objeto com relação as constraints:

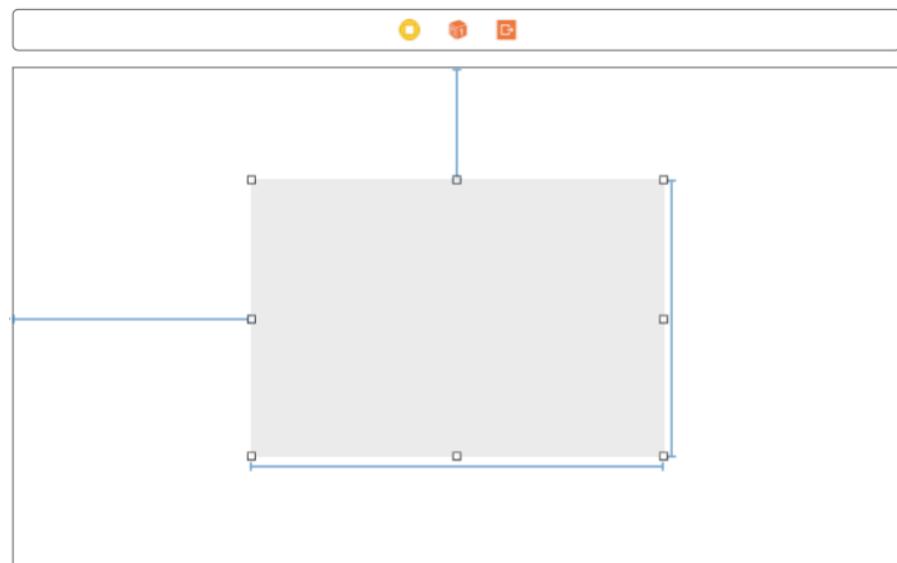
- **Constraints na cor vermelha:** Indicam que ainda faltam constraints para terminar de definir o layout dos objetos selecionados;



- **Constraints na cor Laranja:** Indicam que um objeto foi movido para outra posição, diferente da definida pelas suas constraints. Nessa caso, podemos utilizar o botão **Update Frames** para concertar o ocorrido;



- **Constraints na cor Azul:** Indicam que o obejto selecionado está com todas as constraints necessárias para a formação do seu layout.



Capítulo 11



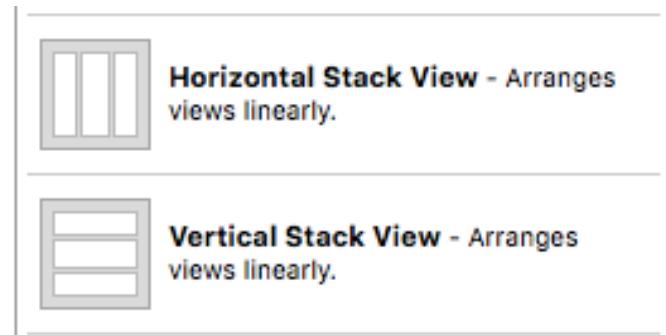
UIStackView

Seção 11-1

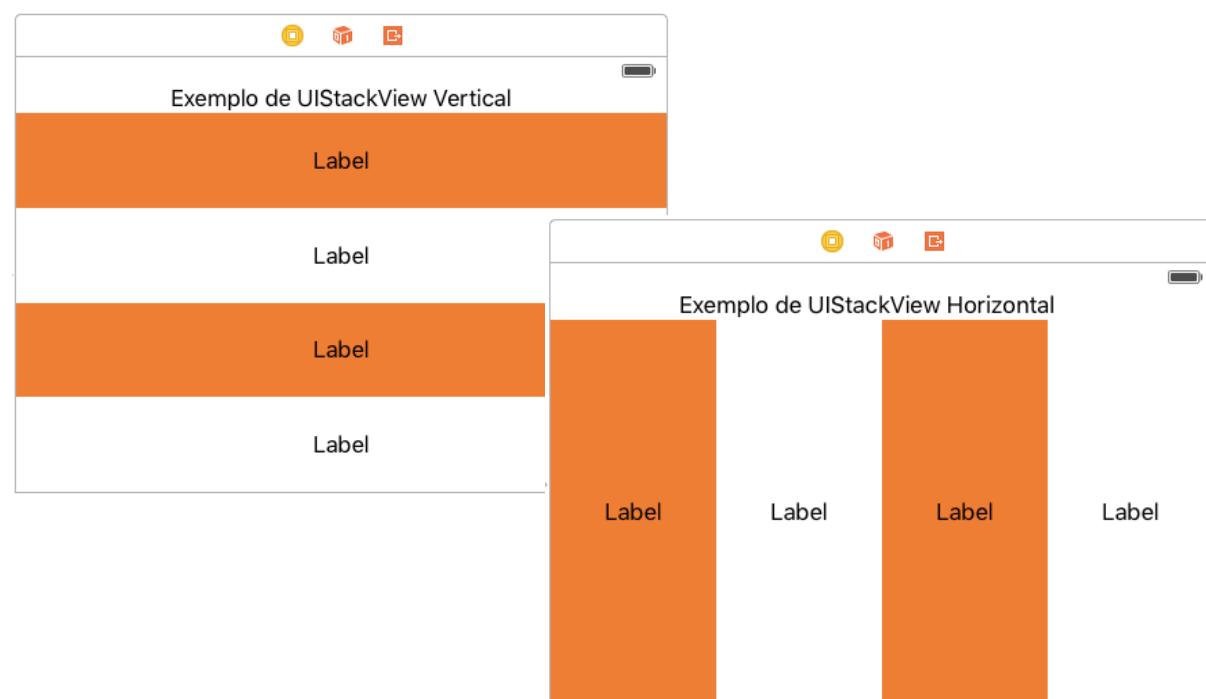
UIStackView



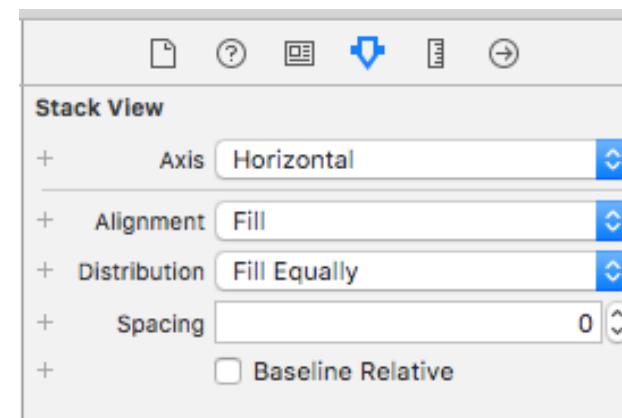
Herança: NSObject > UIResponder > UIView > UIStackView



A classe **UIStackView** fornece uma forma simples de fazer a divisão de um layout em linhas ou colunas. Objetos postos diretamente como subobjetos de uma **UIStackView** se adequam automaticamente ao esquema proposto, seja **horizontal** ou **vertical**.



O trabalho com stack view é bem simples, basta escolhermos com qual tipo vamos trabalhar (horizontal ou vertical), fazer a definição da distribuição dos objetos no painel Attributes, e arrastar os elementos necessários para dentro do stack view.



A cada novo elemento, os demais já postos na vista, se acomodarão automaticamente. O espaçamento entre os elementos também podem ser definidos conforme as necessidades.

#Dica!

Apesar dos elementos internos ao stack view não precisarem da aplicação de **Autolayout**, é importante que o elemento **UIStackView** tenha suas **constraints** configuradas.

Seção 11-2

Embed in Stack

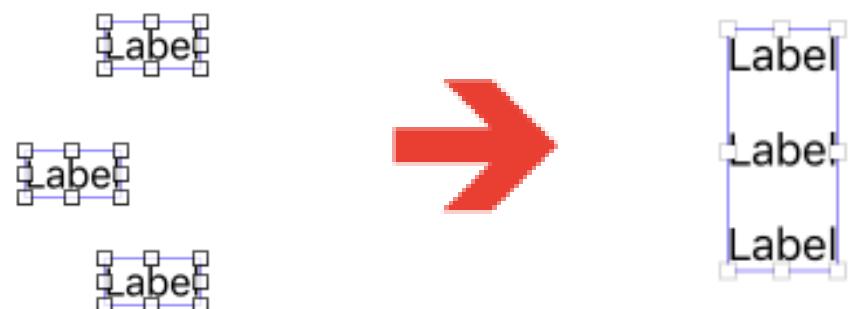


Podemos embutir elementos já já foram colocados na interface previamente em uma Stack View. Para tal, podemos utilizar o recurso **Embed in Stack**, disponível na barra de Autolayout.

As stacks criadas com o Embed in Stack podem ter seus atributos alterados como qualquer outra Stack View.



Para embutir objetos em uma stack, basta escolher os elementos desejados, e clicar no botão:



Capítulo 12



UIScrollView

Seção 12-1

UIScrollView



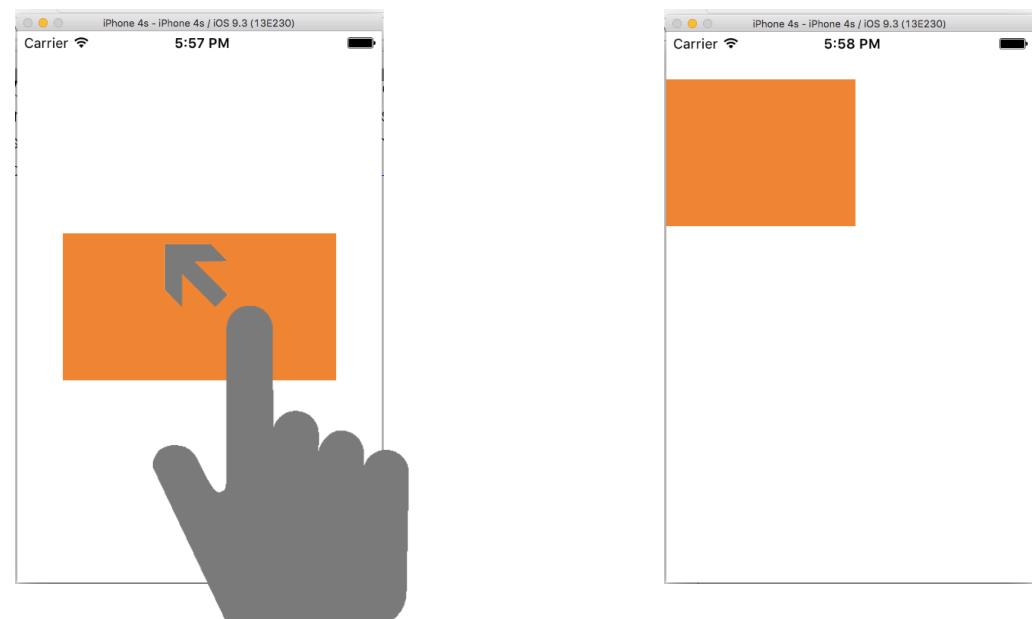
Herança: NSObject > UIResponder > UIView > UIScrollView



A classe **UIScrollView** controla o objeto de interface responsável por fornecer uma **View** com barra de rolagem que pode receber conteúdos maiores do que ela.

A classe **UIScrollView** também possibilita **Zoom In** e **Zoom Out** usando o gesto de pinça com os dedos (assunto abordado em outro módulo).

A **UIScrollView** é a base para outras classes, como a **UICollectionView**, **UITableView** e o **UITextView**.



Principais propriedades

Propriedade	Tipo	Descrição
contentSize	CGSize	Define a quantidade de deslocamento do scroll
isDragging	Bool	Retorna quando o scroll é acionado
delegate	UIScrollViewDelegate?	Define/Retorna o objeto delegate

A classe **ViewController** que receberá **scroll views**, poderá adotar o protocolo **UIScrollViewDelegate** para que os recursos de **delegate** fiquem disponíveis, e dessa forma possamos captar as interações com os objetos ao efetuar arrastos.

```
// MARK: - Métodos de UIScrollViewDelegate
extension ClasseViewController : UIScrollViewDelegate {

    // Declarações dos métodos
}
```

Principais métodos de UIScrollViewDelegate

```
//Método que notifica o delegate que a barra de rolagem foi  
utilizada:  
func scrollViewDidScroll(_ scrollView: UIScrollView)
```

```
//Método que notifica o delegate que a barra de rolagem  
começou a desacelerar:  
func scrollViewWillBeginDecelerating(_ scrollView:  
UIScrollView)
```

```
//Método que comunica o delegate que a barra de rolagem foi  
até o topo do conteúdo  
func scrollViewDidScrollToTop(_ scrollView: UIScrollView)
```

Capítulo 13



Closures

Seção 13-1

Sintaxe das closures



Closures são ferramentas para a criação de funções **in-line**. Com elas podemos criar blocos de códigos que podem atuar como variáveis ou funções.

Elas são muito similares aos blocos em Objective-C ou, fazendo uma analogia com outras linguagens, podemos compará-las com callbacks e lambdas.

Como fazemos com funções e métodos, as **closures** podem receber parâmetros (argumentos) e também podem possuir um retorno de dados, de qualquer tipo:

Observe a sintaxe de uma **Closure**:

```
{(parametros) -> tipoDeRetorno in  
  //Declarações  
}
```

Vamos ver como acontece a declaração padrão, sem closure:

```
// Declaração da função  
func saudacaoMatinal() {  
  print("Bom Dia")  
}
```

```
//Exibir o resultado final  
saudacaoMatinal()
```

Vamos ao mesmo exemplo em uma **closure**:

```
//Declaração da variável, junto com a função  
var saudacaoNoturna = {() -> Void in print("Boa  
Noite")}
```

```
//Exibição do resultado  
saudacaoNoturna()
```

Observe que obtivemos resultados idênticos, porém reduzindo a quantidade de códigos.

#Dica!

Lembre-se que **Void** é um tipo, portanto mesmo que a **closure** não retorne nada, você poderá utilizar o tipo **Void**.

Podemos indicar um tipo de retorno para uma closure, Vamos a alguns exemplos:

```
//Closure para somar +1 a um número indicado:  
var maisUm = {(a: Int) -> Int in  
  
    let soma = a + 1  
    return soma  
}  
  
print(maisUm(10))  
//Resultado: 11  
  
//Closure para multiplicar dois números:  
var multiplicar = {(a: Int, b: Int) -> Int in  
  
    return a * b  
}  
  
print(multiplicar(5,5))  
//Resultado: 25  
  
//Closure para repetir mensagens por quantidade de vezes:  
var repetirMsg = {(mensagem: String, repetir: Int) in  
  
    for _ in 1...repetir{  
  
        print(mensagem)  
    }  
}  
  
repetirMsg("Olá", 5)  
//Resultado: Olá  
    Olá  
    Olá  
    Olá  
    Olá
```

Utilizando Closure como parâmetro de uma função

Podemos passar uma closure como um parâmetro de um função. Em assuntos que serão abordados mais adiante, nos depararemos com muitos casos onde uma closure é um dos elementos que compõem um método.

```
//Closure como parâmetro de um método:  
func minhaClosure(umaClosure: () -> Void) {  
    umaClosure()  
}  
  
//Exemplo de utilização em um método inicializador:  
let acaoOK = UIAlertAction(title: "OK", style: .default,  
handler: ((UIAlertAction) -> Void)?)  
  
//Nesse caso a closure pode ser utilizada pra complementar  
a ação a ser disparada com o método.
```

Capítulo 14



Trabalhando com Alertas

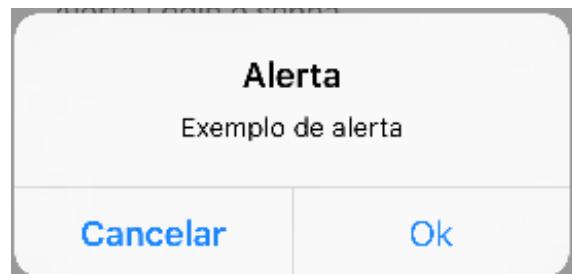
Seção 14-1

UIAlertController



Herança: NSObject > UIResponder > UIViewController > UIAlertController

Um **UIAlertController** exibe uma mensagem de alerta para o usuário. Devemos configurar o controlador de alerta com as ações e estilos necessários, e depois utilizar o método específico para apresentá-lo.



Todo o desenvolvimento do sistema de alertas deve ser feito de forma programática. Não temos a disposição um objeto de interface que possa ser arrastada para uma view controller. Todo o sistema de alertas segue o padrão estabelecido pelo iOS, e pode fariar de aspecto dependendo da versão do sistema,

Os elementos presentes em um alerta podem ser criados conforme a necessidade de cada mensagem. Podemos complementar o alerta com campos de texto e botões com diversas finalidades.



Declarando um alerta

Podemos declarar um objeto **UIAlertController** dentro da classe que executará o alerta, vamos usar como exemplo a classe padrão da nossa ViewController. O alerta pode ser declarado como uma constante conforme podemos ver no exemplo:

```
let nomeDoAlerta = UIAlertController (title: "Título do Alerta", message: "Mensagem do alerta", preferredStyle: UIAlertControllerStyle.alert)
```

Observe que junto com a declaração do alerta, podemos definir o título e a mensagem que serão exibidos.

Para que o alerta seja exibido, devemos declarar a sua apresentação com um método específico de **UIViewController**. Esse método deve ser declarado dentro do método que executará a ação do alerta.

O método **present** de **UIViewController** exibirá o alerta como uma modal, e nele temos a disposição uma closure para disparar um método complementar a exibição do alerta. Vamos ao exemplo:

```
present(nomeDoAlerta, animated: Bool){  
    //Bloco da closure a ser executado com a apresentação do alerta, como um método complementar  
}
```

Principais propriedades

Propriedade	Tipo	Descrição
title	String	Define um título para o alerta
message	String	Define uma mensagem para o alerta
textFields	[UITextField]	Faz a leitura a partir de um UITextField a partir do seu índice

Principais métodos

```
.addAction(_ action: UIAlertAction)  
//Adiciona um objeto de ação definida ao alerta  
  
.addTextField(configurationHandler: ((UITextField) -> Void)?  
= nil)  
//Adiciona campos de texto ao alerta
```

Seção 14-2

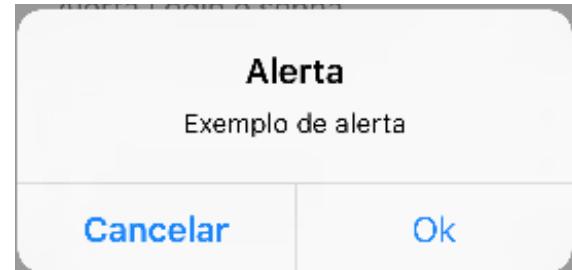
UIAlertAction



Herança: NSObject > UIAlertAction

Utilizamos a classe **UIAlertAction** para criar botões em um alerta e definir uma ação para esse botão.

Esta classe é utilizada para configurar as informações de um elemento por vez, definindo o título que será apresentado no botão, informações de estilo, e uma closure que pode ser executado quando o botão é pressionado.



Declarando uma ação

Cada botão que fará parte do nosso alerta deve ser declarado como uma constante:

```
let nomeDoBotão = UIAlertAction(title: "Título do botão",
style: UIAlertActionStyle.default) { (umNomeParaAcao) in
    //Bloco da closure que será executado ao clicar no botão
}
```

Para que o botão seja exibido junto ao alerta, devemos declarar a sua apresentação com um método específico de **UIAlertController**. Esse método deve ser declarado dentro do método que executará a ação do alerta.

O método **addAction** exibirá o botão junto com o alerta, vamos ao exemplo:

```
nomeDoAlerta.addAction(nomeDoBotão)
```


Capítulo 15



UIPickerView

Seção 15-1

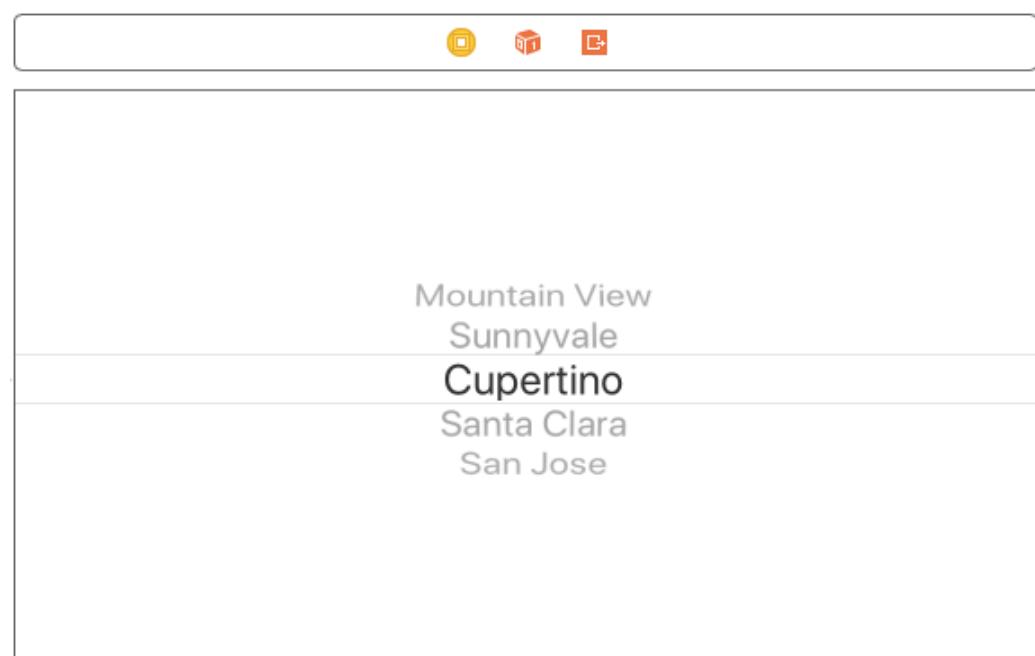
UIPickerView



Herança: NSObject > UIResponder > UIView > UIPickerView



A classe **UIPickerView** controla o objeto de interface responsável por exibir um menu vertical de opções. Esse tipo de objeto é popularmente conhecido como **menu roleta**. Entre outras customizações, podemos definir múltiplas colunas para um **PickerView**.



Protocolos para DataSource e Delegate

A definição de valores para um PickerView faz uso de uma forma bem específica de uso de dados: o protocolo **Data Source**.

Um **data source** é um protocolo que tem como função fornecer meios para que um objeto receba dados. No caso de um PickerView, precisamos definir quantas colunas e linhas ela possuirá. Essas são exatamente as tarefas do **data source**.

Quando um objeto de tela precisa de definições de dados, usamos **Data Source**. Quando precisa de definições de interações, usamos o **Delegate**.

Ambos protocolos devem ser adotados pela classe **View Controller** ou pela classe que será utilizada para controlar a View que conterá o picker view:

```
//MARK:- Métodos de UIPickerViewDataSource
extension ViewController : UIPickerViewDataSource {
    //Declarações
}

//MARK:- Métodos de UIPickerViewDelegate
extension ViewController : UIPickerViewDelegate {
    //Declarações
}
```

Principais propriedades

Propriedade	Tipo	Descrição
numberOfComponents	Int	Retorna a quantidade de colunas existentes em um picker view
delegate	UIPickerViewDelegate?	Define/retorna o sistema de interação do picker view
dataSource	UIPickerViewDataSource?	Define o sistema de entrada de dados para o picker view

Principais métodos

```
//Método que recarrega uma coluna específica do PickerView  
.reloadComponent(_ component: Int)
```

```
//Método que recarrega todos os elementos do PickerView  
.reloadAllComponents()
```

```
//Método que retorna a linha selecionada de uma coluna definida  
.selectedRow(inComponent component: Int) -> Int
```

```
//Método que seleciona uma linha em uma coluna específica.  
Este método pode ser utilizado com animação.  
.selectRow(_ row: Int, inComponent component: Int, animated: Bool)
```

Métodos de UIPickerViewDataSource

O protocolo **UIPickerViewDataSource** define os métodos obrigatórios que um PickerView deve implementar para definir a quantidade de linhas e colunas a serem exibidas. Os dois métodos que compõem o protocolo são obrigatórios:

```
//Método que retorna a quantidade de colunas de um PickerView  
func numberOfComponents(in pickerView: UIPickerView) -> Int
```

```
//Método que define a quantidade de linhas de uma coluna específica  
func pickerView(pickerView: UIPickerView,  
numberOfRowsInComponent component: Int) -> Int
```

Métodos de UIPickerViewDelegate

O protocolo **UIPickerViewDelegate** define os métodos opcionais que um PickerView pode implementar para resgatar a interação do usuário com as opções exibidas.

```
//Método que notifica que o usuário selecionou uma linha  
optional func pickerView(_ pickerView: UIPickerView,  
didSelectRow row: Int, inComponent component: Int)
```

```
//Método que retorna o título da linha específica na coluna  
//definida  
optional func pickerView(_ pickerView: UIPickerView,  
titleForRow row: Int, forComponent component: Int) -> String?
```



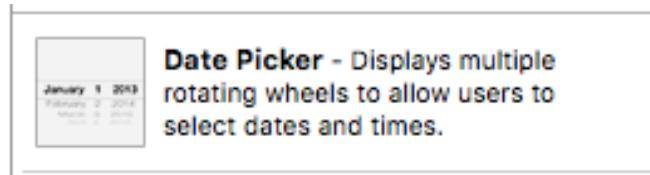

UIDatePicker, Date e Timer

Seção 16-1

UIDatePicker



Herança: NSObject > UIResponder > UIView > UIControl > UIDatePicker

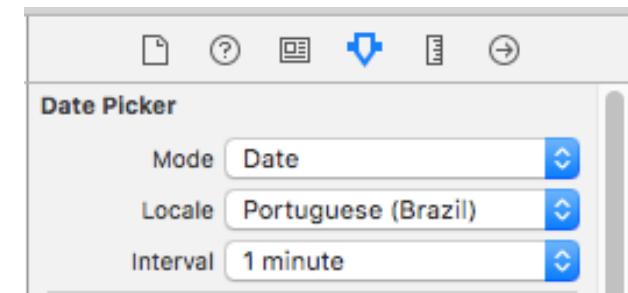


Date Picker - Displays multiple rotating wheels to allow users to select dates and times.

A classe **UIDatePicker** controla o objeto de interface responsável por exibir um menu para seleção de datas. A base de um UIDatePicker é a classe **UIPickerView**, customizada para exibir colunas referentes as informações de data e horário.

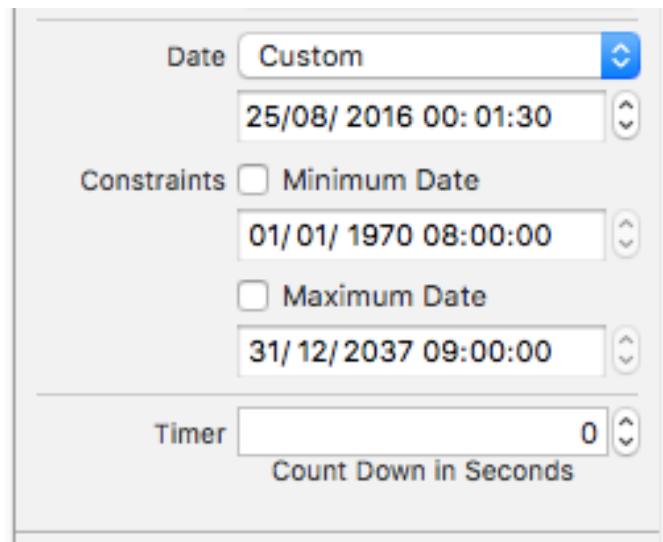


Podemos formatar o DatePicker para mostrar diferentes tipos de informações, como datas, horas ou contadores de tempo. Outro ponto importante é a localização do idioma do Picker, que também pode ser selecionado.



00	55
01	56
02	57
03	58
04	59
05	00
06	01
sáb 20 de ago	21
dom 21 de ago	22
seg 22 de ago	23
ter 23 de ago	00
qua 24 de ago	01
qui 25 de ago	02
Hoje	03
0	58
1	59
2	0
3 hours	1 min
4	2
5	3
6	4

As datas mínimas e máximas podem ser definidas para o objeto, bem como com qual data o picker será iniciado, podendo ser definidas tanto a data atual como uma data customizada.



Podemos trabalhar com Outlets ou Actions do DatePicker para captar datas, e utilizá-las na nossa aplicação com finalidades diversas.

Seção 16-2

Date



A estrutura **Date** representa um momento específico de horário e data. A captação desse momento é feita assim que a instância do objeto Date é conformada.

Devido a essa estrutura de captação de informações de horário e data, um objeto Date é **imutável**, e por isso, para fazermos resgates específicos (como segundos e minutos), precisamos usar a classe **DateFormatter**, que veremos mais adiante.

A estrutura ainda fornece métodos para comparar datas, fazer cálculos de intervalo de tempo entre datas, e criar novas datas de intervalo de tempo em relação a uma outra data.

Principais métodos

```
//Método inicializador que retorna um objeto que representa  
a diferença em segundos a partir do tempo atual  
init(timeIntervalSinceNow: TimeInterval)
```

```
//Método inicializador que retorna um objeto que representa  
a diferença em segundos a partir de uma data especificada  
init(timeInterval: TimeInterval, since date: Date)
```

```
//Método inicializador que retorna um objeto de data zerado  
em 01/01/1970 as 0:00h  
init(timeIntervalSince1970: TimeInterval)
```

```
//Método que retorna a diferença do intervalo de tempo em  
segundos de um data definida  
func timeIntervalSince(_ anotherDate: Date) -> TimeInterval
```

Seção 16-3

DateFormatter



Herança: NSObject > Formatter > DateFormatter

A classe **NSDateFormatter** tem como objetivo permitir o trabalho de datas em formato **String**, convertendo datas em texto, ou transformando textos em data.

Através de métodos específicos e definições de estilos, podemos extrair de datas completas apenas os trechos desejados tais como minutos, segundos ou dias da semana.

Como exemplo, podemos extrair a data de um **Date Picker**, e transportá-la para uma **label**, formatando a data da forma desejada.

4	junho	1819
5	julho	1820
6	agosto	1821
7	setembro	1822
8	outubro	1823
9	novembro	1824
10	dezembro	1825

sábado, 7 de setembro de 1822

[Mostrar](#)

Principais propriedades

Propriedade	Tipo	Descrição
dateFormat	String!	Formata a data string recebida para diferentes formatos
dateStyle	NSDateFormatterStyle	Formata a data string recebida para esquemas pré estabelecidos
locate	Locate	Define a localização da data recebida

Principais métodos

//Método que retorna um NSDate a partir de uma String
date(from string: String) -> Date?

//Método que retorna uma String a partir de um objeto NSDate
string(from date: Date) -> String

DateFormatterStyle

As constantes de **DateFormatterStyle** definem um estilo controlado para o resgate de informações de data.

Através destes estilos, podemos fazer resgates curtos, médios, longos ou completos de datas e horários.

- **none:** Não Especifica estilo.
- **short:** Normalmente apenas números. Ex: 28/01/84
- **medium:** Usa abreviações. Ex: 28 Jan 1984
- **long:** Descrição extensa. Ex: 28 de Janeiro de 1984
- **full:** Descrição completa. Ex: Terça-feira 28 de Janeiro de 1984

Seção 16-4

Timer



Herança: NSObject > Timer

Com a classe **Timer** podemos criar objetos **temporizadores**.

Um timer de espera um determinado intervalo de tempo decorrido e depois dispara, enviando uma mensagem especificada para um objeto de destino.

Por exemplo: podemos criar um objeto Timer que envia uma mensagem para uma janela, acionando uma atualização após um determinado intervalo de tempo.

Principais propriedades

Propriedade	Tipo	Descrição
timeInterval	TimeInterval	Retorna o intervalo de tempo definido
fireDate	Date	Define a data para disparo do timer

Principais métodos

```
//Cria e retorna um novo objeto Timer e agenda um loop  
de execução no modo padrão  
.scheduledTimer(ti: TimeInterval, target:  
aTarget: Any, selector aSelector: Selector, userInfo: Any?,  
repeats yesOrNo: Bool) -> Timer
```

```
//Dispara uma mensagem de início do timer para ser enviado  
para seu alvo  
.fire()
```

```
//Pausa a execução de um timer  
.invalidate()
```

Capítulo 17



UITableView e UITableViewController

Seção 17-1

UITableView



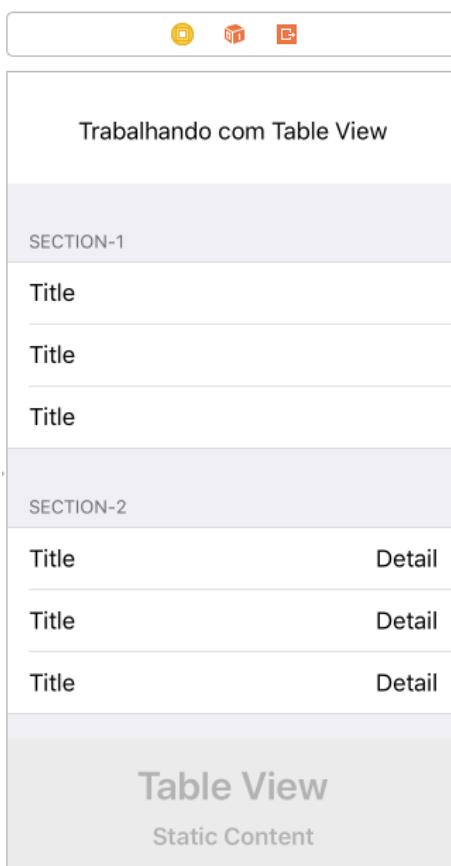
Herança: NSObject > UIResponder > UIView > UIScrollView > UITableView



Table View - Displays data in a list of plain, sectioned, or grouped rows.

A classe **UITableView** controla o objeto de interface responsável por exibir uma lista hierárquica de informações. Por ser uma subclasse de **UIScrollView**, o objeto já possui nativamente a opção de rolagem para seu conteúdo.

Montamos uma lista com a concentração de objetos **UITableViewCell**, que podem ser divididos em seções para uma ordenação visual.



UITableViewDataSource e UITableViewDelegate

A definição de dados para um UITableView é feita através do protocolo **data source**. A interação e resgate de seleções são feitas usando o protocolo **delegate**. É importante que esses protocolos sejam adotados pela classe que interpretará os métodos

```
extension ClasseViewController : UITableViewDataSource {  
    //Declarações dos métodos  
}
```

```
extension ClasseViewController : UITableViewDelegate {  
    //Declarações dos métodos  
}
```

Todo o trabalho de definição e resgate de células é baseado em um tipo de dados específico chamado **IndexPath**. Ele concentra as informações de seções e linhas em uma única variável, veremos nas principais propriedades.

Tabelas estáticas e dinâmicas

Existem basicamente dois tipos de tabelas: as **dinâmicas** e as **estáticas**.

Tabelas dinâmicas tem seus conteúdos definidos dinamicamente através do **código programático**. Já uma tabela estática pode ser definida visualmente no **storyboard**.

Principais propriedades

Propriedade	Tipo	Descrição
dataSource	UITableViewDataSource?	Define/Retorna o objeto delegate
delegate	UITableViewDelegate?	Define/Retorna o objeto data source
rowHeight	CGFloat	Define/Retorna a altura das celulas
numberOfSections	Int	Retorna a quantidade de seções da tabela
isEditing	Bool	Retorna se a tabela está sendo editada

//Método de UITableViewDataSource responsável por definir e retornar a quantidade de seções:

```
func numberOfSections(in tableView: UITableView) -> Int
```

//Método de UITableViewDataSource que define títulos para as seções:

```
func tableView(_ tableView: UITableView,  
titleForHeaderInSection section: Int) -> String?
```

//Método de UITableViewDataSource responsável por definir e retornar se conteúdos das células podem ser editados (apagados ou inseridos):

```
func tableView(_ tableView: UITableView, canEditRowAt  
indexPath: IndexPath) -> Bool
```

//Método de UITableViewDataSource responsável por definir comandos de exclusão ou inclusão de novas células e conteúdo:

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

//Método de UITableViewDataSource responsável por definir novas posições para as células:

```
func tableView(_ tableView: UITableView, canMoveRowAt  
indexPath: IndexPath) -> Bool
```

//Método de UITableViewDataSource responsável por definir comandos de exclusão ou inclusão de novas células e conteúdo:

```
func tableView(_ tableView: UITableView, moveRowAt  
fromIndexPath: IndexPath, to: IndexPath)
```

Principais métodos de UITableViewDelegate

```
//Método que notifica ao delegate que uma célula da tabela  
foi selecionada:  
func tableView(_ tableView: UITableView, didSelectRowAt  
indexPath: IndexPath)
```

```
//Método que diz ao delegate que foi removida a seleção de  
uma célula da tabela:  
func tableView(_ tableView: UITableView, didDeselectRowAt  
indexPath: IndexPath)
```

Métodos de UITableView

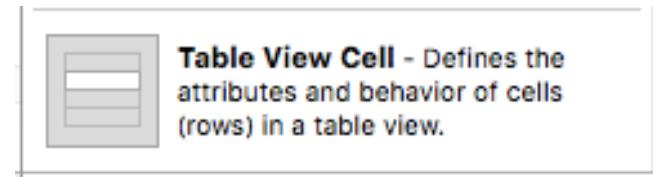
```
//Método que de aplica um conteúdo no formato de String a  
uma célula pré identificada:  
.dequeueReusableCell(withIdentifier identifier: String) ->  
UITableViewCell?
```

Seção 17-2

UITableViewCell



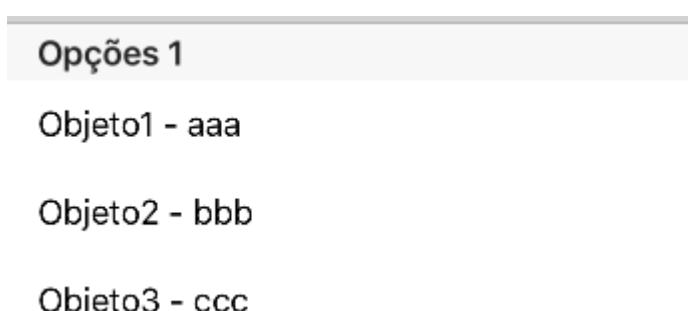
Herança: NSObject > UIResponder > UIView > UITableViewCell



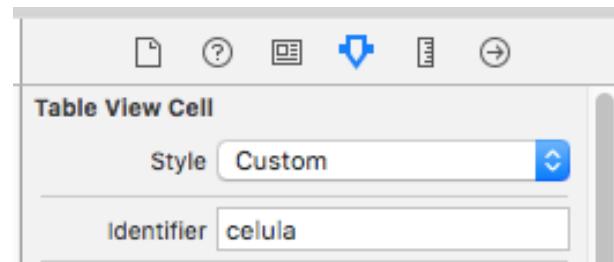
A classe **UITableViewCell** define as propriedades e o comportamento das células em uma tabela.

Basicamente uma célula é feita utilizando um fundo que seria uma **UIView**. Podemos colocar dentro dessa célula objetos de texto, como uma **UILabel**, ou até mesmo um objeto de imagem **UIImage**.

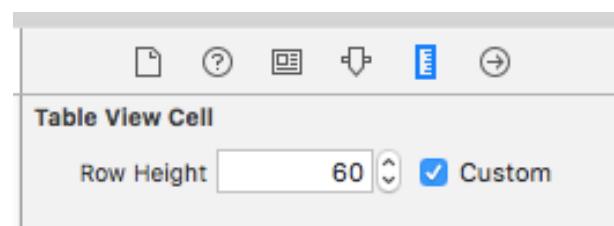
Podemos adicionar células a uma tabela conforme a necessidade de cada projeto.



No painel **Attributes** temos a disposição a opção **Identifier**, que pode indicar um **identificador** para que a célula seja utilizada no código programático:



Podemos utilizar as opções do painel **Size** para indicar a altura da linha da célula:



Principais propriedades

Propriedade	Tipo	Descrição
isSelected	Bool	Retorna se uma célula está selecionada
isEditing	Bool	Retorna se uma célula está sendo editada
textLabel	UILabel?	Retorna o valor que está sendo usado na label
reuseldentifier	String?	Retorna o identificador da célula
imageView	UIImageView?	Retorna a imagem usada na célula

Seção 17-3

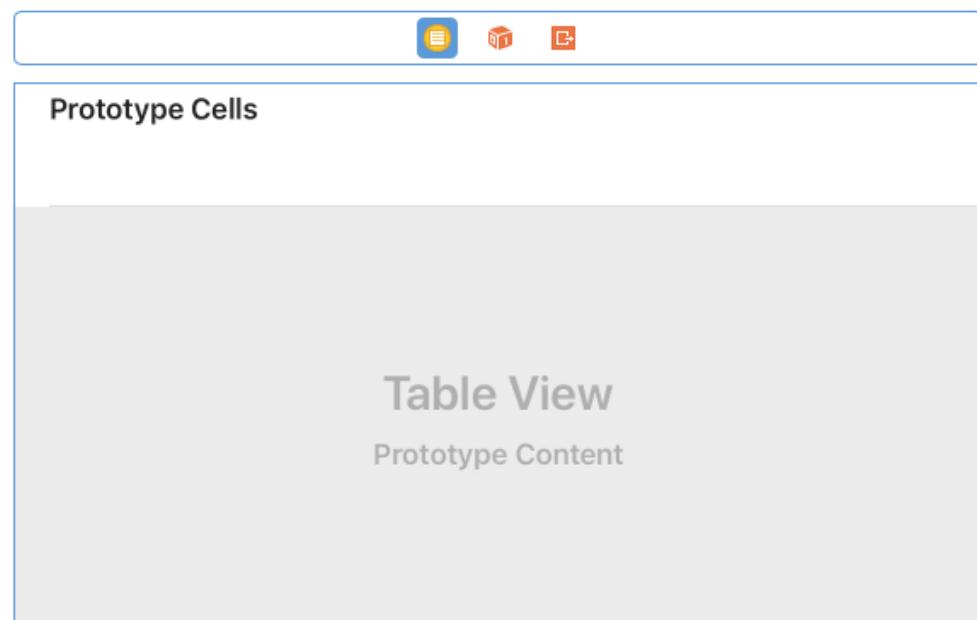
UITableViewController



Herança: NSObject > UIResponder > UIViewController > UITableViewController



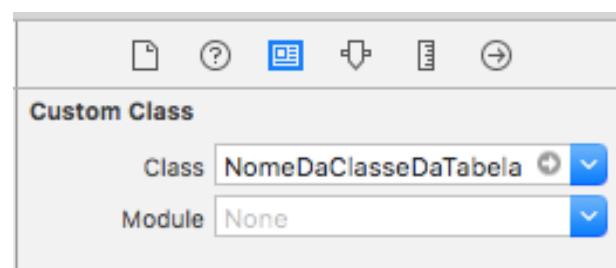
A classe **UITableViewController** cria um objeto controlador (janela) que gerencia uma exibição de tabela. Ou seja, temos uma view controller interligada diretamente e automaticamente com uma tabela.



Quando criamos um objeto UITableViewController, a classe que comanda esse controlador não está especificada nem criada. Ou seja, diferente do que acontece com um view controller inicial, não temos ainda um arquivo de códigos que comande o funcionamento do objeto, isso pode ser observado no navegador **Identify**

Devemos então criar uma classe que será responsável pelo funcionamento do nossa table view controller, que herde a classe **UITableViewController**, e em seguida agregar essa classe ao nosso objeto, através da opção **Class** do navegador **Identify**:

```
class NomeDaClasseDaTabela : UITableViewController {  
    //Conteúdo da classe  
}
```



#Dica!

Como a classe criada herda características de **UIViewController**, todas as funções da mesma, como **ViewDidLoad** por exemplo, estarão disponíveis na nova classe.

Outro ponto interessante é que os protocolos de **Delegate** e **DataSource** serão adotados e implementados automaticamente.

Capítulo 18



UICollectionView

Seção 18-1

UICollectionView

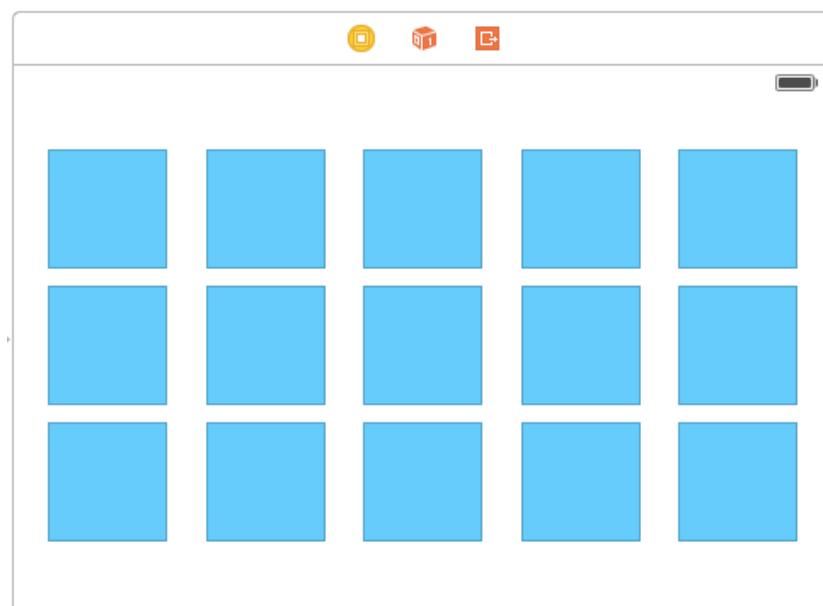


Herança: NSObject > UIResponder > UIView > UIScrollView > UICollectionView



A classe **UICollectionView** administra uma coleção ordenada de itens de dados de forma a apresentá-los usando layouts personalizados compostos por células em formato multicoluna.

Por ser uma subclasse de **UIScrollView**, o objeto já possui nativamente a opção de rolagem para seu conteúdo.



UICollectionViewDataSource e UICollectionViewDelegate

Assim como o UITableView, a definição de dados para um UICollectionView é feita através do protocolo **data source**. A interação e resgate de seleções são feitas usando o protocolo **delegate**. É importante que esses protocolos sejam adotados pela classe que interpretará os métodos.

```
extension ClasseViewController : UICollectionViewDataSource{  
    //Declaração dos métodos  
}  
  
extension ClasseViewController : UICollectionViewDelegate{  
    //Declaração dos métodos  
}
```

Assim como nas tabelas, todo o trabalho de definição e resgate de células é baseado em dados do tipo **IndexPath**.

#Dica!

Os índices seguirão a ordem de disposição das linhas e das colunas de forma crescente.

Principais propriedades

Propriedade	Tipo	Descrição
dataSource	UICollectionViewDataSource?	Define/Retorna o objeto data source
delegate	UICollectionViewDelegate?	Define/Retorna o objeto delegate

Principais métodos de UICollectionViewDataSource

```
//Método obrigatório de UICollectionViewDataSource  
responsável por definir e retornar a quantidade de itens em  
uma seção:  
func collectionView(_ collectionView: UICollectionView,  
numberOfItemsInSection section: Int) -> Int
```

```
//Método obrigatório de UICollectionViewDataSource  
responsável por implementar os dados de um array a uma  
célula a partir de um índice:  
func collectionView(_ collectionView: UICollectionView,  
cellForItemAt indexPath: IndexPath) -> UICollectionViewCell
```

```
//Método de UITableViewDataSource responsável por definir e  
retornar a quantidade de seções:  
func numberOfSections(in collectionView: UICollectionView)  
-> Int
```

Principais métodos de UICollectionViewDelegate

```
//Método que diz ao delegate que um item da coleção foi  
selecionada:  
func collectionView(_ collectionView: UICollectionView,  
didSelectItemAt indexPath: IndexPath)
```

```
//Método que diz ao delegate que foi removida a seleção de  
um item da coleção:  
func collectionView(_ collectionView: UICollectionView,  
didDeselectItemAt indexPath: IndexPath)
```

Métodos de UICollectionView

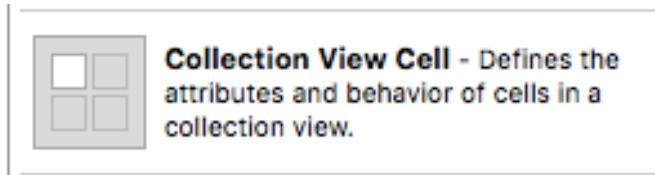
```
//Método que de aplica um conteúdo no formato de String a  
uma célula pré identificada:  
.dequeueReusableCell(withIdentifier identifier: String,  
for indexPath: IndexPath) -> UICollectionViewCell
```

Seção 18-2

UICollectionViewCell



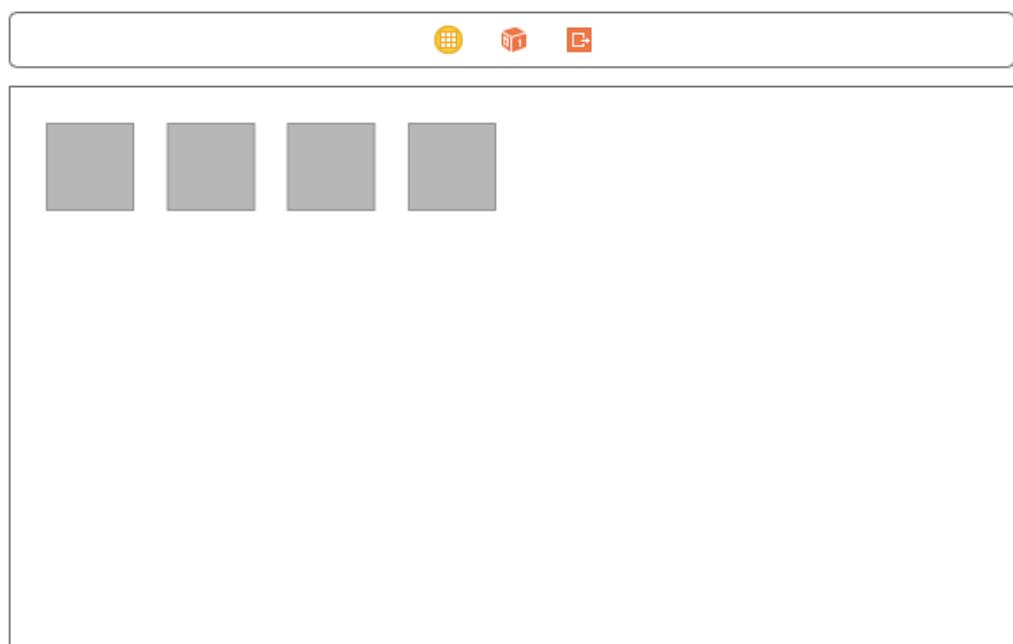
Herança: NSObject > UIResponder > UIView > UICollectionViewReusableView > UICollectionViewCell



Collection View Cell - Defines the attributes and behavior of cells in a collection view.

A classe **UICollectionViewCell** define as propriedades e o comportamento das células em uma collection view.

Assim como nas tabelas, as células de uma coleção são feitas utilizando um fundo que seria uma **UIView**. Podemos colocar dentro dessa célula objetos de texto, como uma **UILabel**, ou até mesmo um objeto de imagem **UIImage**. Podemos adicionar células a uma coleção conforme a necessidade de cada projeto.



Principais propriedades

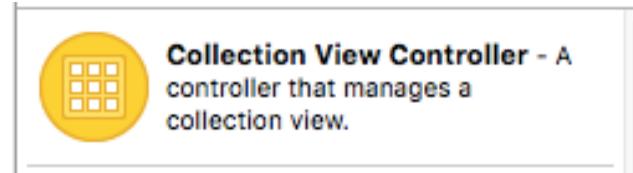
Propriedade	Tipo	Descrição
isSelected	Bool	Retorna se uma célula está selecionada
isHighlighted	Bool	Retorna se uma célula está em destaque

Seção 18-3

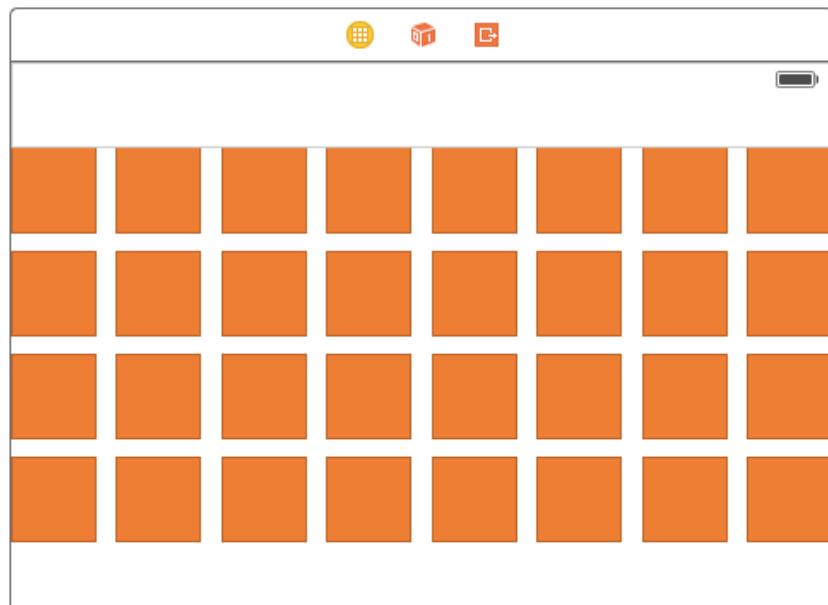
UICollectionViewController



Herança: NSObject > UIResponder > UIViewController > UICollectionViewController



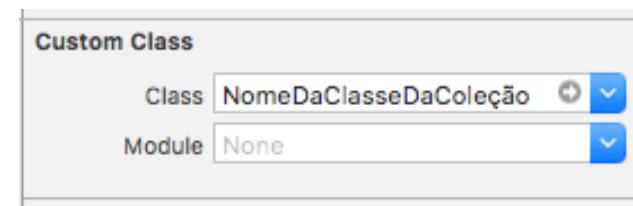
A classe **UICollectionViewController** cria um objeto controlador (janela) que gerencia uma exibição de coleções. Ou seja, temos uma view controller interligada diretamente e automaticamente com uma coleção



Quando criamos um objeto UICollectionViewController, a classe que comanda esse controlador não está especificada nem criada. Ou seja, diferente do que acontece com um view controller inicial, não temos ainda um arquivo de códigos que comande o funcionamento do objeto, isso pode ser observado no painel **Identify**

Devemos então criar uma classe que será responsável pelo funcionamento do nossa collection view controller, que herde a classe **UICollectionViewController**, e em seguida agregar essa classe ao nosso objeto, através da opção **Class** do painel **Identify**:

```
class NomeDaClasseDaColecao : UICollectionViewController {  
    //Conteúdo da classe  
}
```



#Dica!

Como a classe criada herda características de **UIViewController**, todas as funções da mesma, como **ViewDidLoad** por exemplo, estarão disponíveis na nova classe.

Outro ponto interessante é que os protocolos de **Delegate** e **DataSource** serão adotados e implementados automaticamente.

Capítulo 19



UIVisualEffects

Seção 19-1

UIVisualEffect



Herança: NSObject > UIVisualEffect > UIBlurEffect / UVibrancyEffect

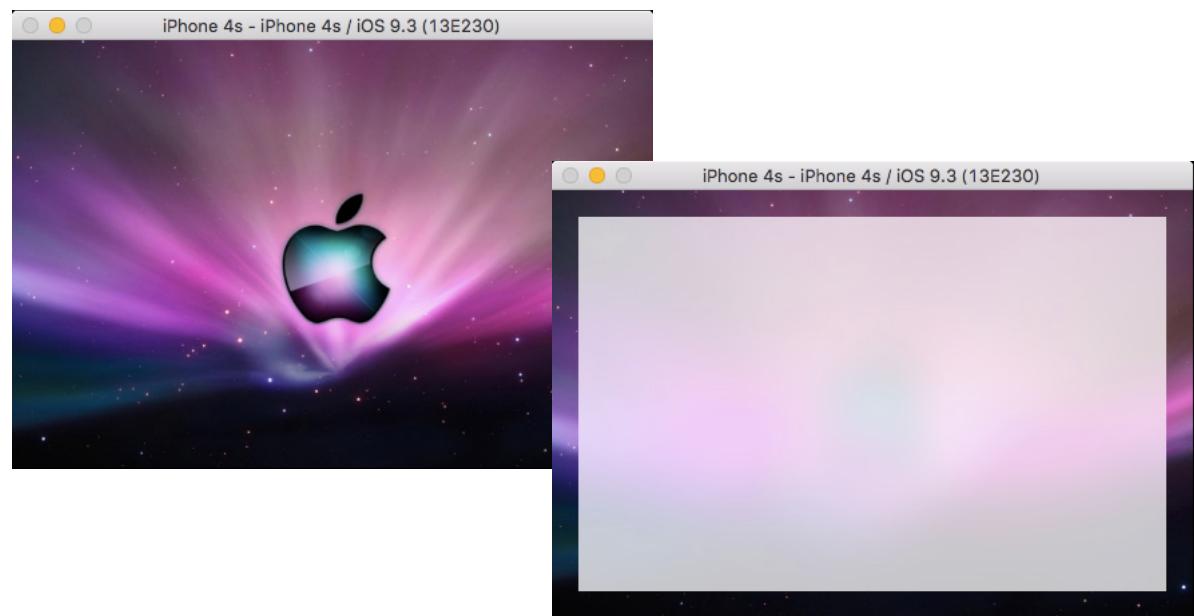
A classe **UIVisualEffect** não contém métodos e destina-se como uma forma de inicializar um objeto **UIVisualEffectView** com recursos de **UIBlurEffect** e **UVibrancyEffect**.

Antes de mais nada, temos que entender a função de cada um dos dois efeitos propostos pela classe:

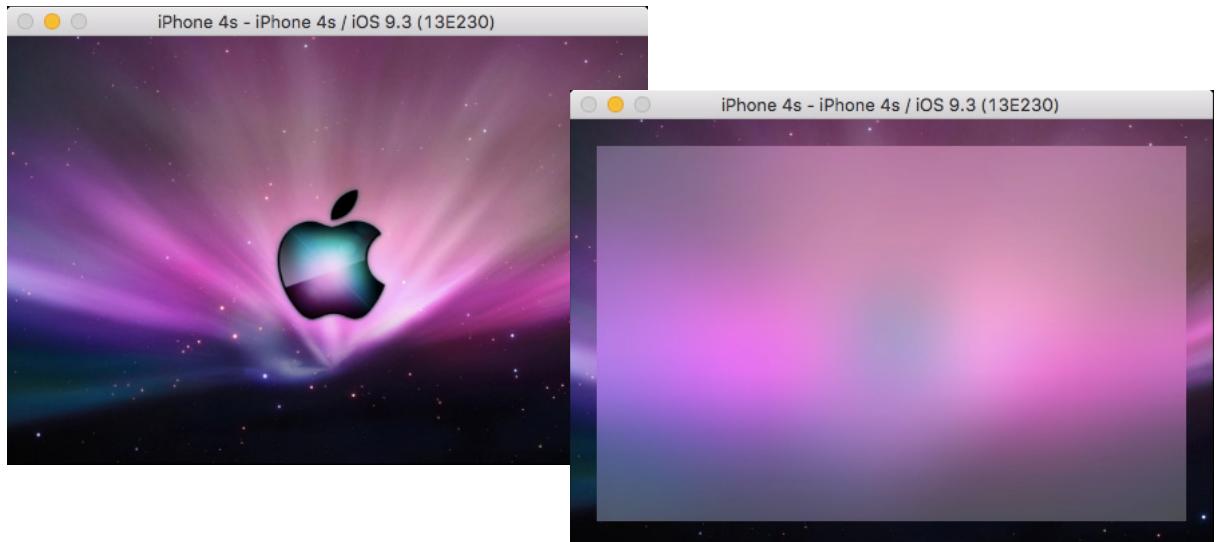
UIBlurEffect

Um objeto **UIBlurEffect** aplica um efeito **blur (embaçar)** no conteúdo que está por **trás** de uma **UIVisualEffectView** (veremos a classe adiante). É possível também escolher estilos diferentes de blur, sendo:

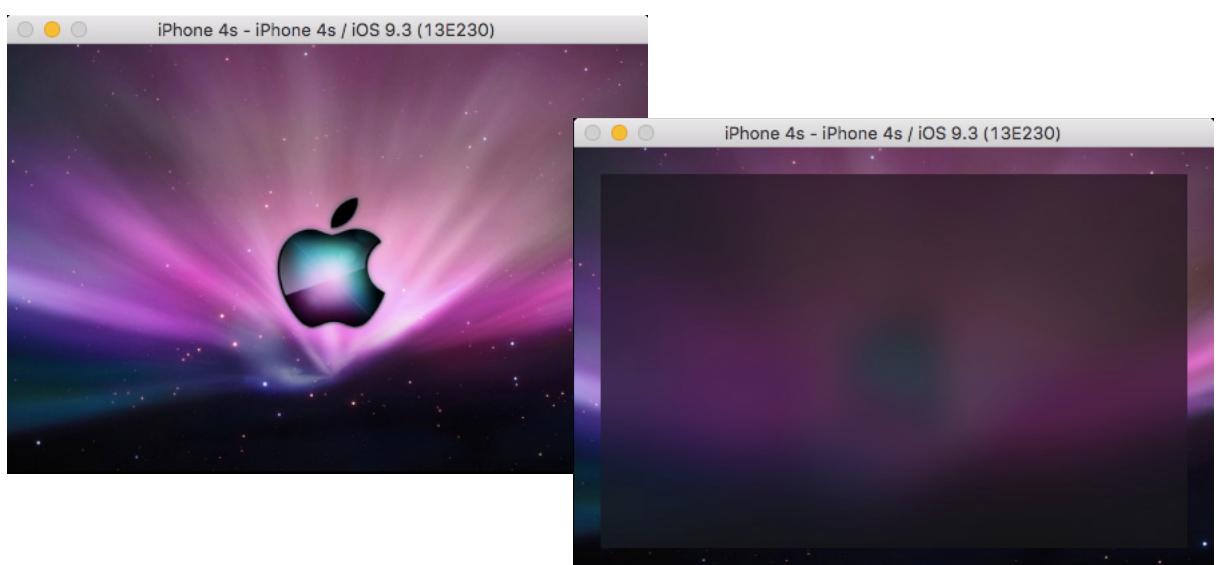
- **ExtraLight**: para um efeito mais claro, próximo do branco;



- **Light**: para um efeito claro, com uma transparência maior;



- **Dark**: para um efeito escuro.



UVibrancyEffect

Um objeto **UVibrancyEffect** amplia e ajusta as cores do conteúdo que está por trás de uma **UIVisualEffectView**, tomando como base o subobjeto que está diretamente posto dentro do objeto **UIVisualEffectView**.

É necessário utilizar as propriedades de **UIBluerEffect** para obtermos diferentes resultados com o **UVibrancyEffect**.



Seção 19-2

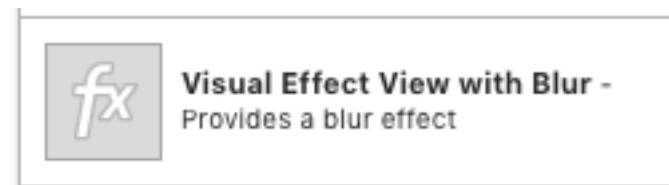
UIVisualEffectView



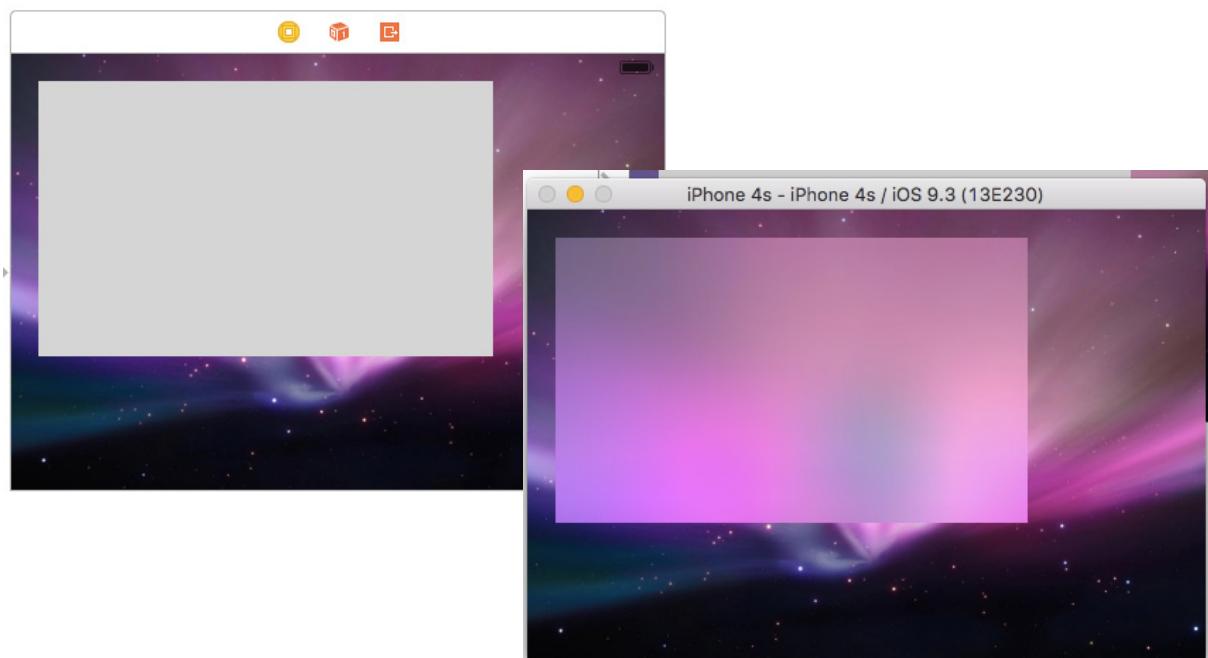
Herança: NSObject > UIResponder > UIView > UIVisualEffectView

Um objeto **UIVisualEffectView** é uma maneira fácil implementar efeitos visuais. Dependendo do efeito desejado (blur ou vibrancy), o efeito pode afetar o conteúdo em camadas atrás da view ou conteúdo adicionado ao contentView da vista que está acima do objeto.

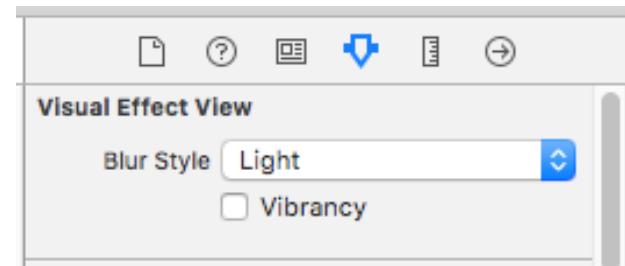
Trabalhando o efeito Blur



Para indicar a área onde o efeito atuará, devemos arrastar o elemento, e soltá-lo acima do objeto desejado, e dimensionar conforme as necessidades.



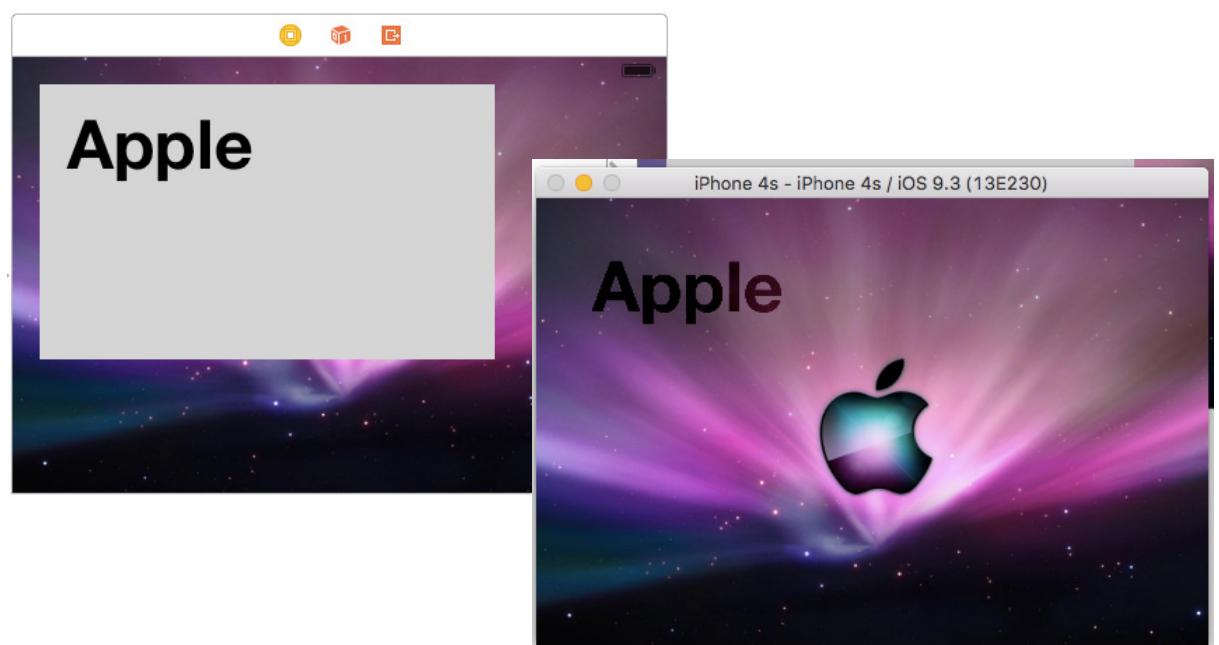
No painel **Attributes** podemos escolher estilo do blur em **Blur Style**:



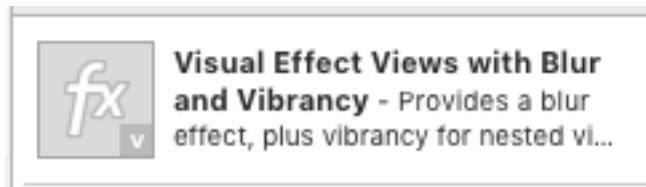
Trabalhando o efeito Vibrancy

Para o efeito **Vibrancy**, devemos proceder da mesma forma vista com o efeito Blur. No painel Attributes a opção Vibrancy deve permanecer ligada.

Devemos lembrar que o efeito atuará a partir do objeto posto diretamente acima do **UIVisualEffectView**. Como exemplo utilizamos um objeto **UILabel**.

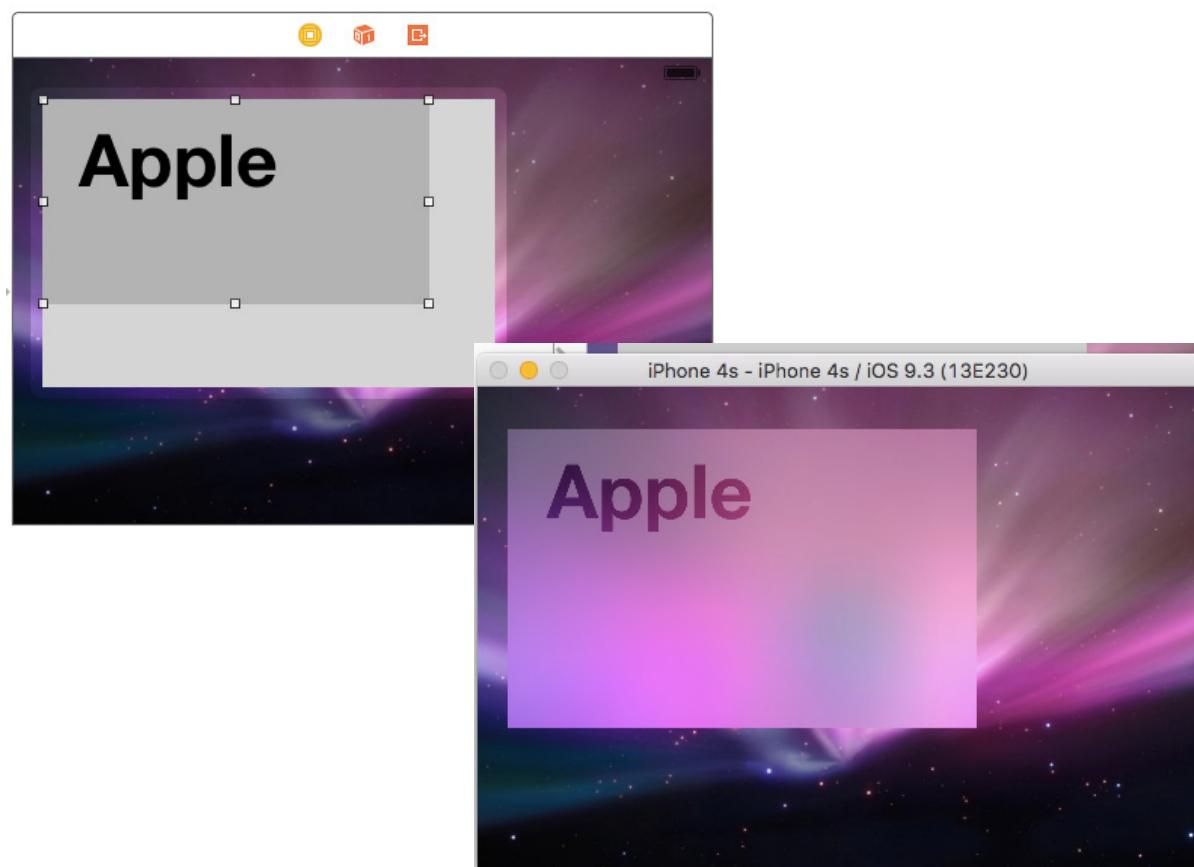


Trabalhando com efeitos Blur e Vibrancy em conjunto



Podemos trabalhar os efeitos Blur e Vibrancy de forma simultânea em um mesmo conjunto de objetos quando for necessário.

O procedimento é semelhante aos indicados anteriormente, com a diferença que o objeto trará consigo dois objetos view, um dentro do outro, e devem ser dimensionados conforme a necessidade:



Capítulo 20



Eventos de toque e UITouch



Vimos a alguns capítulos atrás, a classe **UIResponder**, responsável pela interação entre os objetos e o usuário final. Uma das interações mais comuns entre o usuário e a aplicação, são os eventos de toque.

Quando interagimos com toques na tela, existem alguns métodos de **UIResponder** que são disparados no momento dessas interações.

Vamos abordar os métodos que interagem com eventos de toque pela tela do dispositivo:

```
//Método disparado quando um evento de toque é iniciado:  
func touchesBegan(_ touches: Set<UITouch>,  
with event: UIEvent?)
```

```
//Método disparado quando o toque se mantém arrastando:  
func touchesMoved(_ touches: Set<UITouch>,  
with event: UIEvent?)
```

```
//Método disparado quando o toque é encerrado:  
func touchesEnded(_ touches: Set<UITouch>,  
with event: UIEvent?)
```

```
//Método disparado quando o toque é cancelado por outra  
tela, como uma tela de recebimento de ligação:  
func touchesCancelled(_ touches: Set<UITouch>,  
with event: UIEvent?)
```

Seção 20-2

UITouch



Herança: NSObject > UITouch

Um objeto **UITouch** representa a localização, tamanho, movimento e força de um evento de toque na tela. A força de um toque está disponível a partir de iOS 9 em dispositivos que suportam 3D touch ou Apple Pencil.

Entre os recursos de um objeto **UITouch** estão a posição do toque, o objeto gráfico tocado e a contagem de toques.

Principais propriedades

Propriedade	Tipo	Descrição
view	UIView?	Retorna o objeto gráfico que foi tocado.
tapCount	Int	Mostra a quantidade de toques dados na tela
type	UITouchType	Define o tipo de toque, dedo / stylus
force	CGFloat	Mostra a força do toque (disponível apenas para alguns dispositivos)

Principais métodos

//Método que retorna a posição atual do toque na view definida:
.location(in view: UIView?) -> CGPoint

//Método que retorna a posição anterior do toque na view definida:
.previousLocation(in view: UIView?) -> CGPoint

Capítulo 21



Trabalhando com multitelas

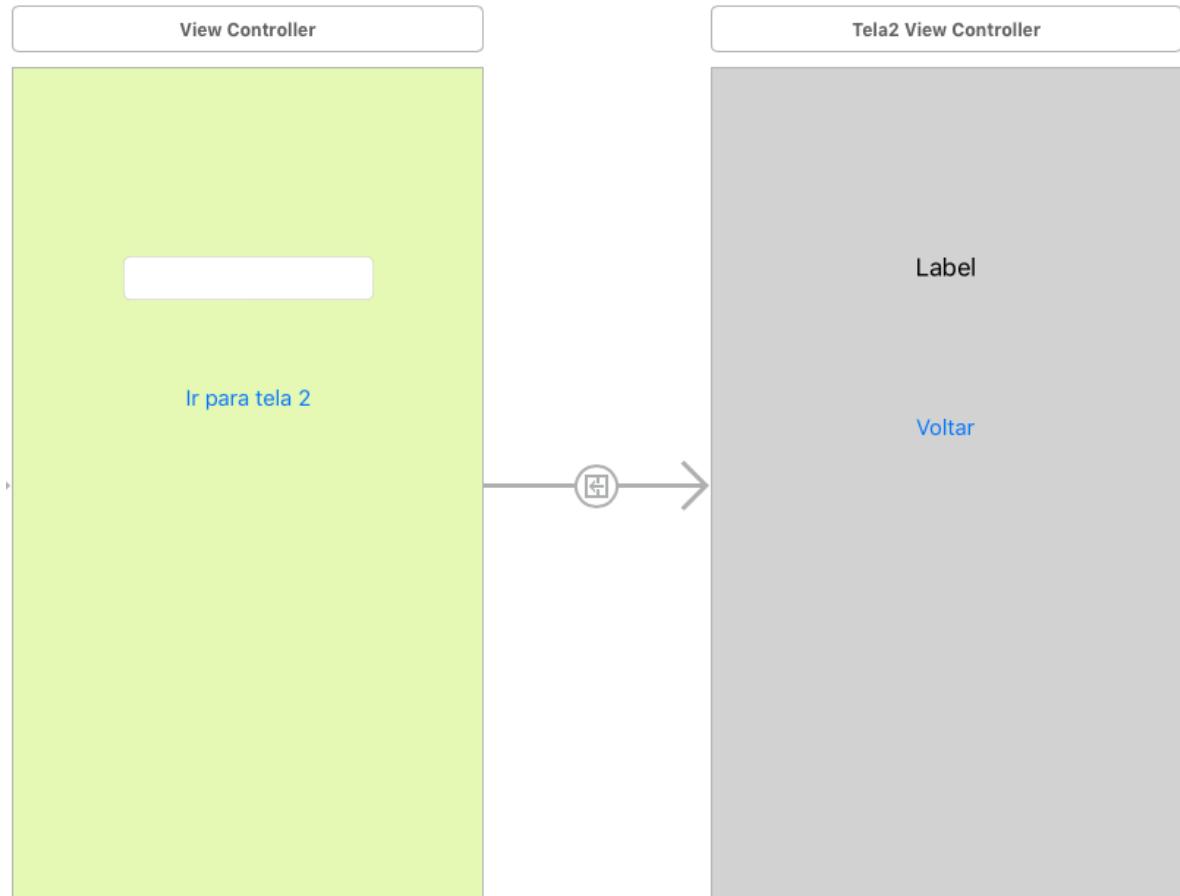
Seção 21-1

UIStoryboard



Herança: NSObject > UIStoryboard

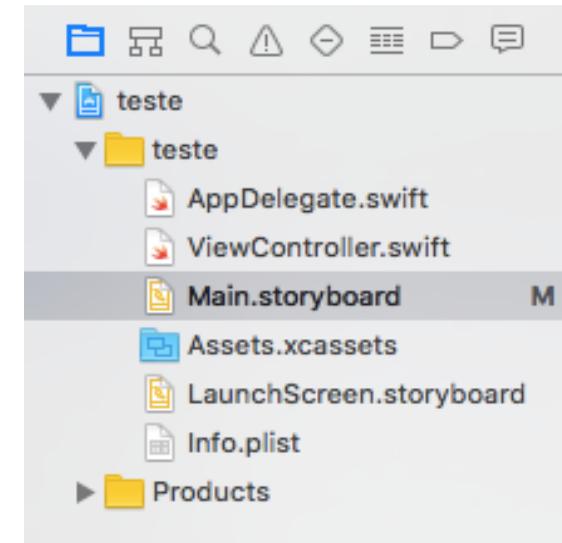
Storyboard é um conceito que foi implementado a partir do iOS 5 e nos permite trabalhar de forma mais simples com múltiplas telas para um aplicativo. Podemos chamar de Storyboard, a área onde se concentram as Views Controllers que irão compor as telas da nossa aplicação.



Quando iniciamos um aplicativo Single View App, temos a disposição **dois Storyboards** previamente criados.

O primeiro deles, **LaunchScreen.storyboard** é responsável por implementar uma tela de inicialização para o aplicativo.

O segundo Storyboard, chamado **Main.storyboard** irá concentrar os itens que farão parte da nossa aplicação, podendo ter multiplas Views Controllers conforme a necessidade de cada projeto.



Com intuito de organizar nosso projeto, podemos distribuir nossas views controllers em storyboards diversos, e interligar esses storyboards através de objetos de referência.

É uma prática comum a refatoração de um projeto depois que temos todas as view controllers construídas em um storyboard, e posterior redistribuição conforme as necessidades.

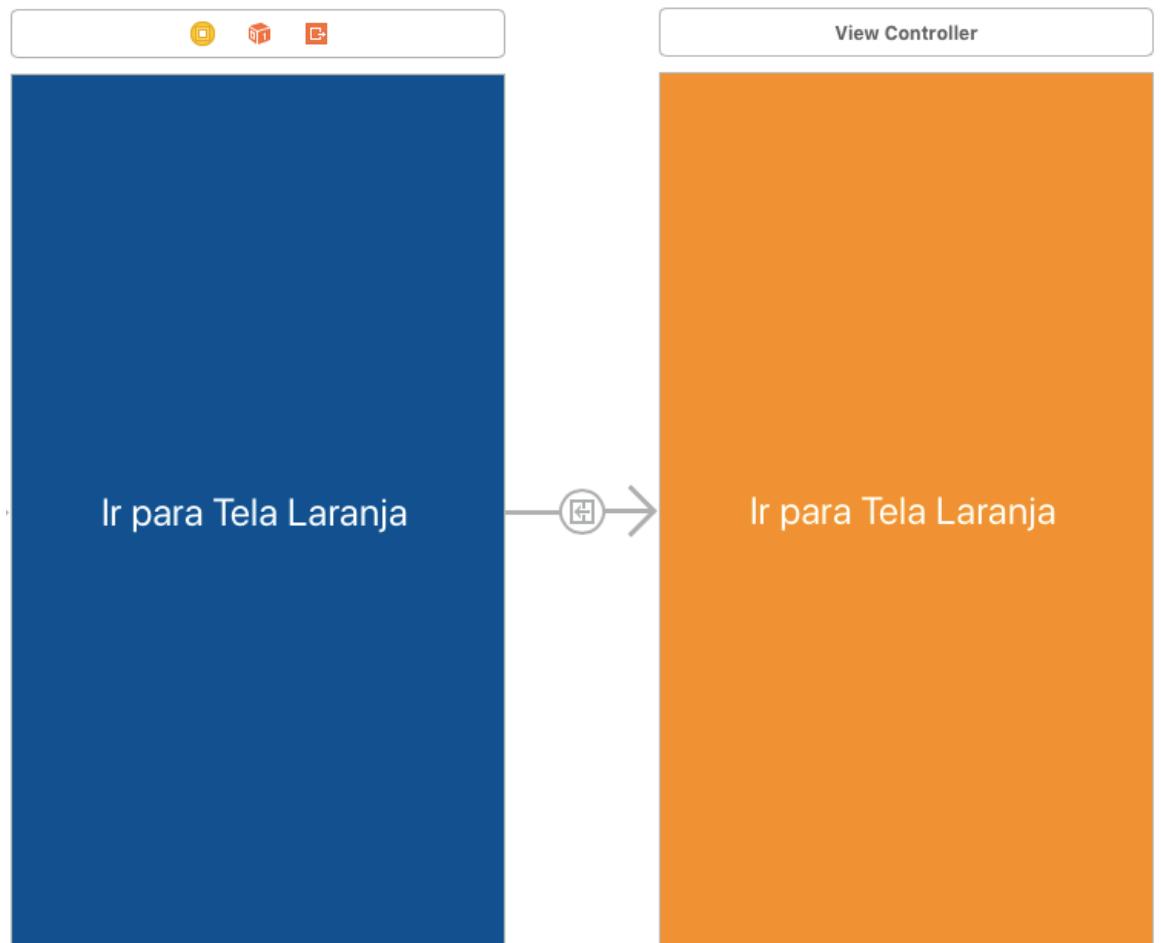
Seção 21-2

UIStoryboardSegue



Herança: NSObject > UIStoryboardSegue

Um objeto **UIStoryboardSegue** é responsável por executar a transição visual entre dois view controllers. Os **Segues** são identificados os objetos gráficos no formato de setas que interligam duas View Controllers.



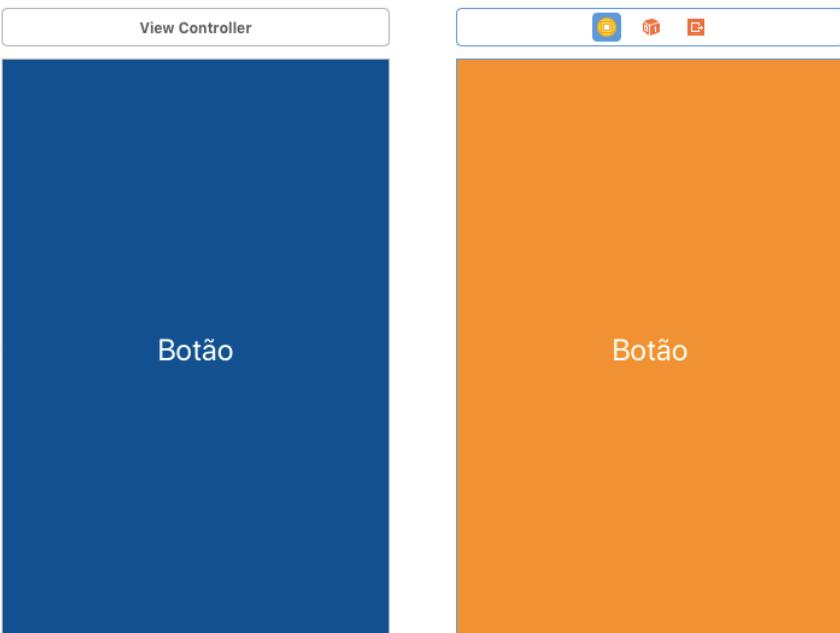
Os objetos **Segue** são utilizados para preparar a transição de um controlador de exibição para outro.

Quando um **segue** é acionado, antes que ocorra a transição visual, o tempo de execução de storyboard chama dentro da view controller atual o método `.prepare(for segue: UIStoryboardSegue, sender: Any?)` para que possa passar todos os dados necessários para o view controller que está prestes a ser exibido.

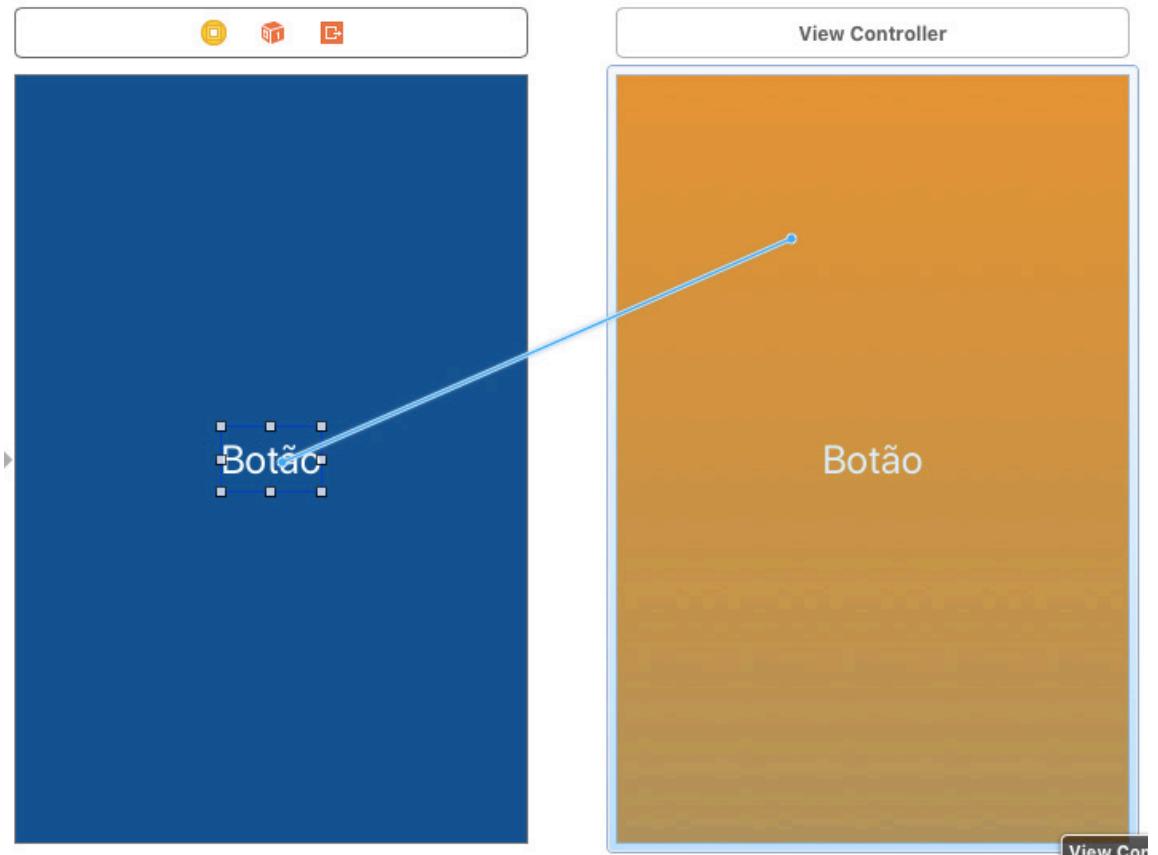
Criando Segue

Podemos criar um segue a partir de um objeto que ao disparar uma ação, faça em conjunto, a transição para uma próxima view controller.

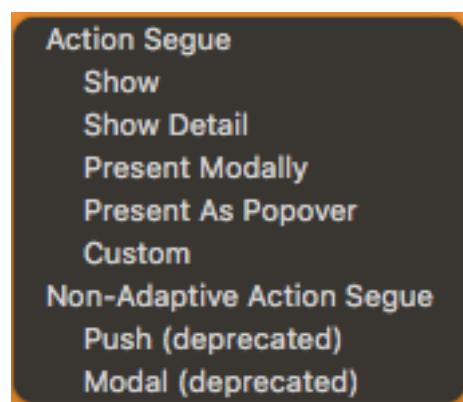
Podemos usar como exemplo um botão que ao ser pressionado permita a transferência para outra view controller. É importante montarmos uma estrutura com duas view controllers e um botão em cada uma delas que nos orientará:



Em seguida, com o botão da view controller inicial selecionado, vamos arrastá-lo com o botão direito do mouse e soltá-lo em cima da segunda view controller:



Ao soltar o botão, um menu suspenso é exibido. Nele escolhemos a opção **Show**:



Pronto! Temos o **Segue** criado e totalmente funcional. basta rodar a aplicação e fazer os testes.

Seção 21-3

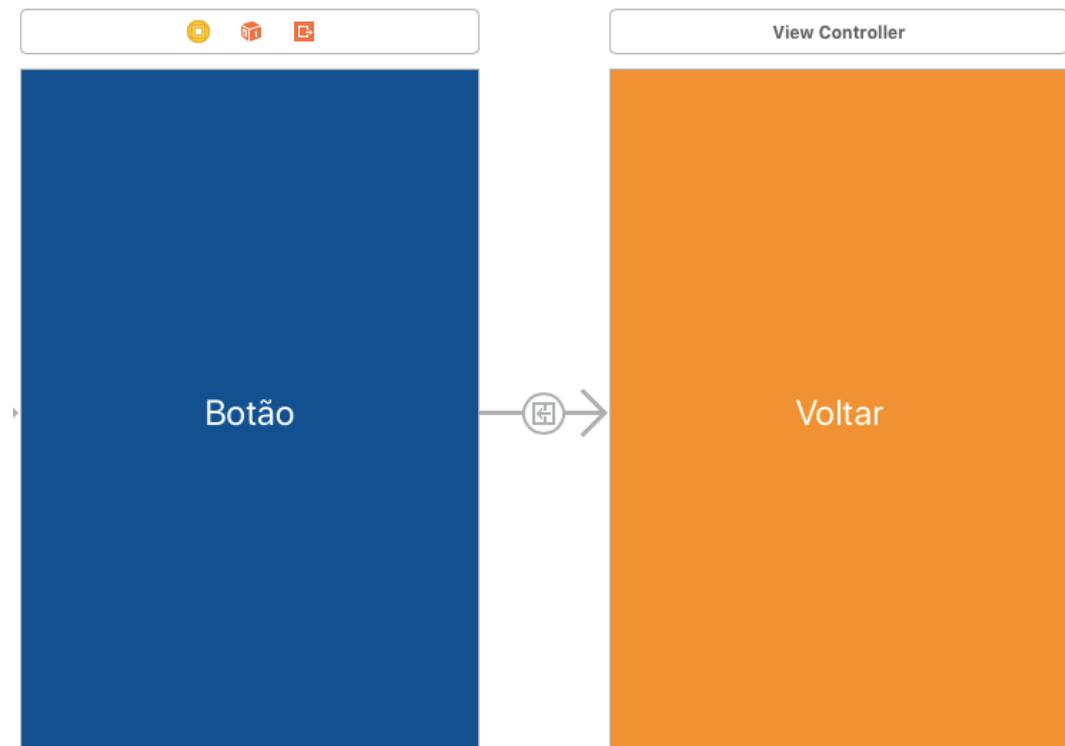
Dispensando View Controllers - Dismiss



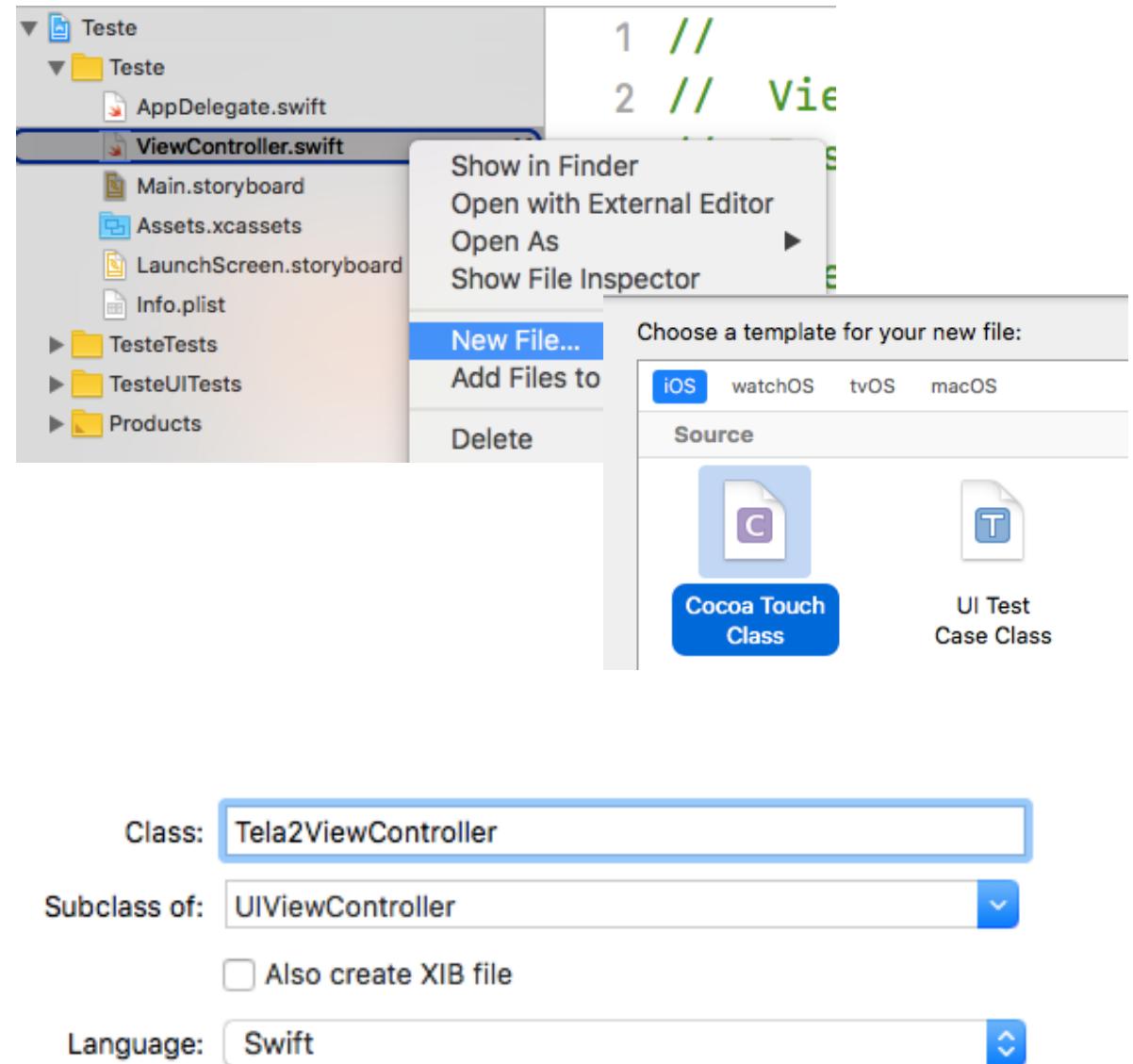
Quando um Segue é realizado, a nova view controller exibida passa a fazer parte da pilha de objetos que estão em memória. Não é recomendado realizar outro segue para voltar a uma view controller anterior.

Quando temos a necessidade de dispensar uma view controller, voltar para uma view controller anterior, e consequentemente retirar a view dispensada da memória, realizamos um **Dismiss**.

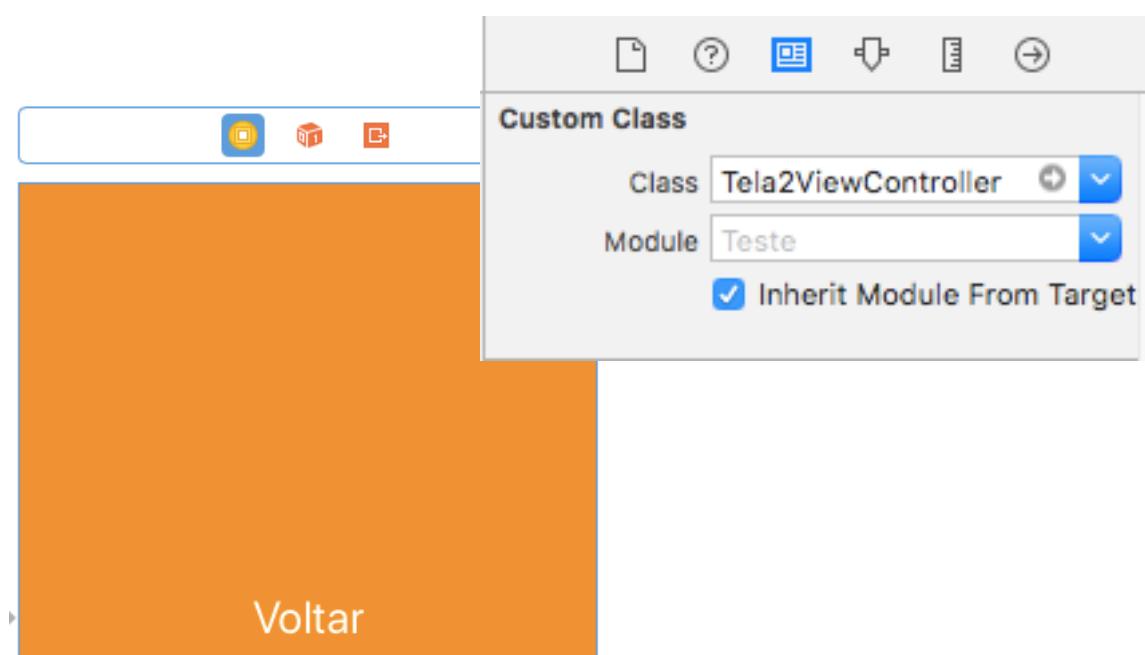
Vamos trabalhar com o exemplo de duas view controllers conforme a imagem, onde um segue foi criado entre Botão, e a segunda View Controller:



Para a segunda tela, vamos criar um arquivo de classe chamado Tela2ViewController, que seja subclasse de UIViewController:



Vamos indicar a classe criada para a segunda View Controller em **Identity Inspector**:



Dentro da classe Tela2ViewController, vamos criar uma **@IBAction** do botão voltar com as seguintes instruções:

```
class Tela2ViewController: UIViewController {  
  
    @IBAction func voltar(_ sender: UIButton) {  
        self.dismiss(animated: true, completion: nil)  
    }  
}
```

Rode a aplicação e faça os devidos testes.

#Dica!

O método **.dismiss** possui uma closure que pode ser utilizar para executar um bloco complementar a sua ação.

Seção 21-4

Criando novos Storyboards

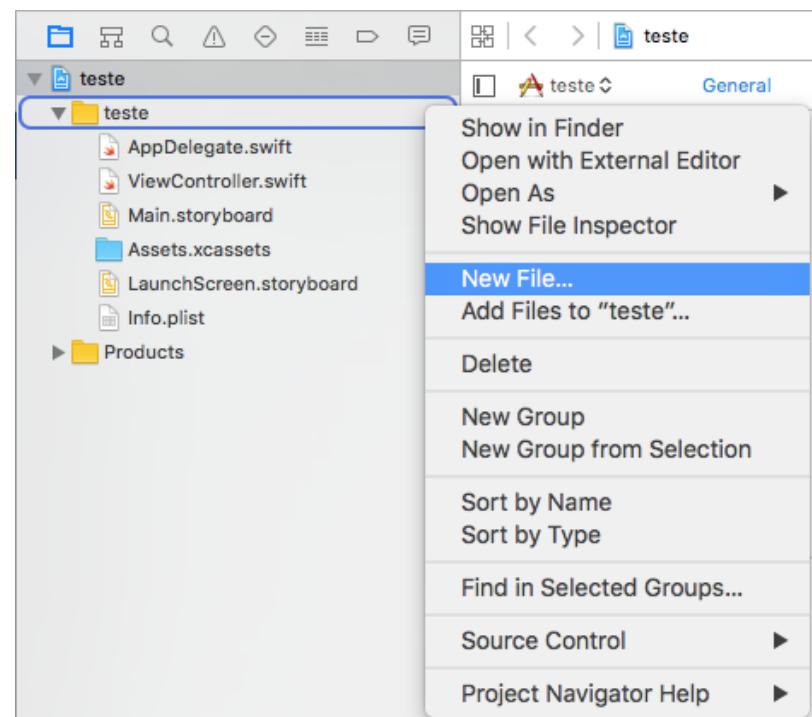


Podemos criar novos storyboards com intuito de dividir nosso projeto em diferentes áreas, para uma melhor organização e dinâmica.

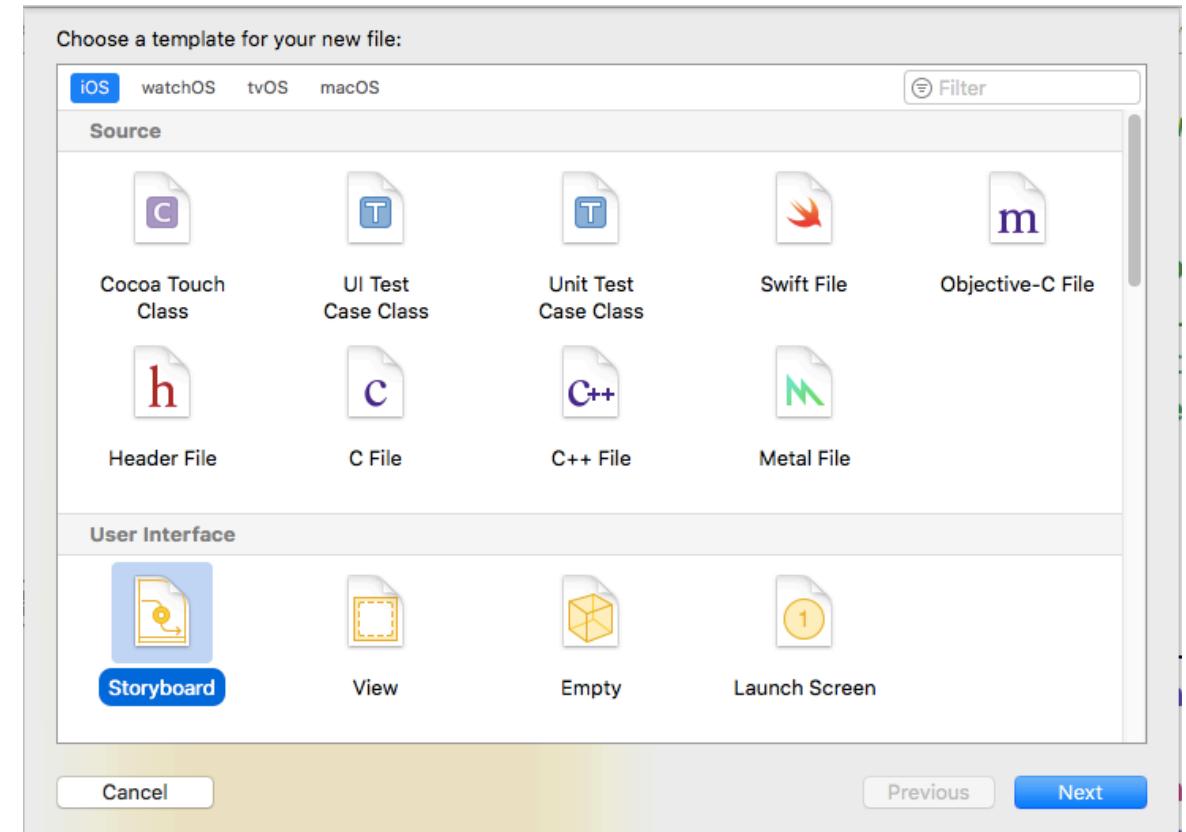
As áreas de storyboard são arquivos que fazem parte do pacote da aplicação com a extensão **.Storyboard**.

Para criarmos um novo storyboard vamos aos procedimentos:

- Na área **Navigator** do XCode, vamos clicar com o botão direito do mouse sobre a pasta do projeto, e escolher a opção **New File...**:



- Em seguida, na caixa exibida, vamos escolher a opção **iOS/User Interface**, e vamos escolher um arquivo **Storyboard**, e clicar no botão **Next**.

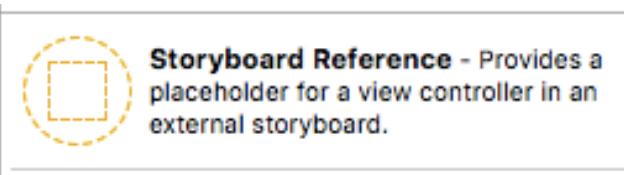


- Para finalizar, vamos dar **um nome** para nosso novo storyboard, e clicar no botão **Create**.

A partir daí, temos a disposição um novo storyboard que pode ser utilizado, assim como o Main.storyboard, para criarmos novas vistas para nosso projeto.

Seção 21-5

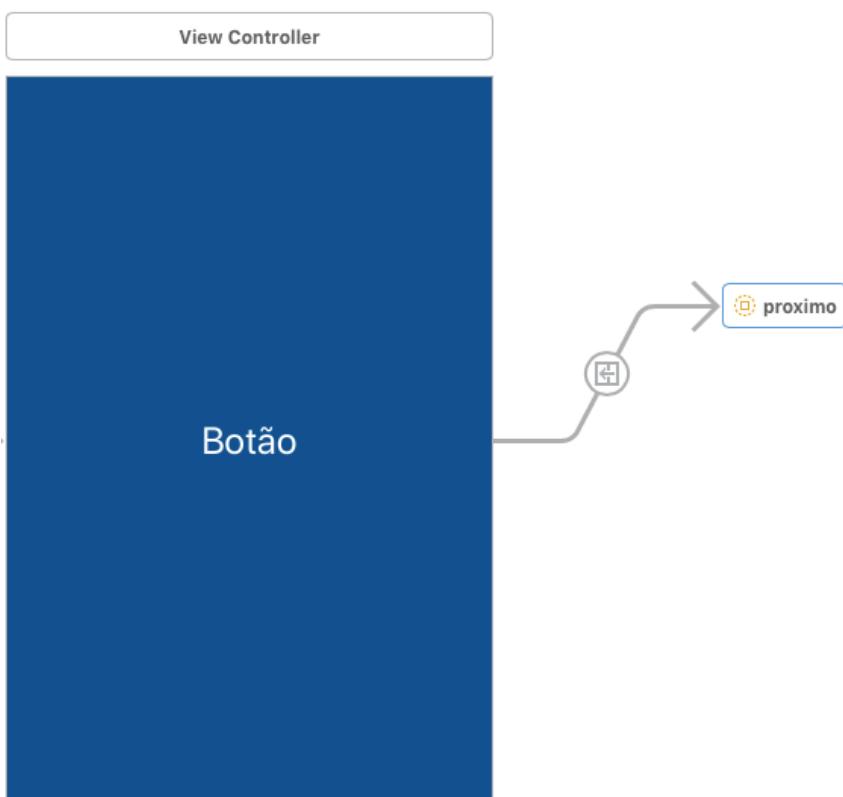
Storyboard Reference



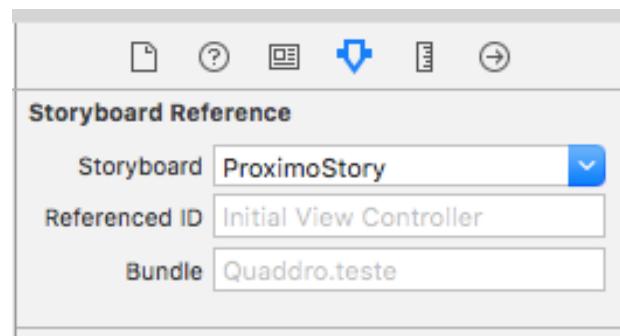
Storyboard Reference - Provides a placeholder for a view controller in an external storyboard.

Quando dividimos nosso projeto em storyboards distintos, podemos criar referências entre esses storyboards, o que torna a organização do trabalho mais funcional e dinâmica.

O objeto que fará o link entre dois storyboards é o **Storyboard Reference**.



Para criar o link entre os storyboards, devemos colocar um objeto **Storyboard Reference** no storyboard de **origem** e apontar esse objeto para o storyboard de **destino**. Para tal podemos utilizar a opção **Storyboard** do painel **Attributes**:



Capítulo 22



Container View

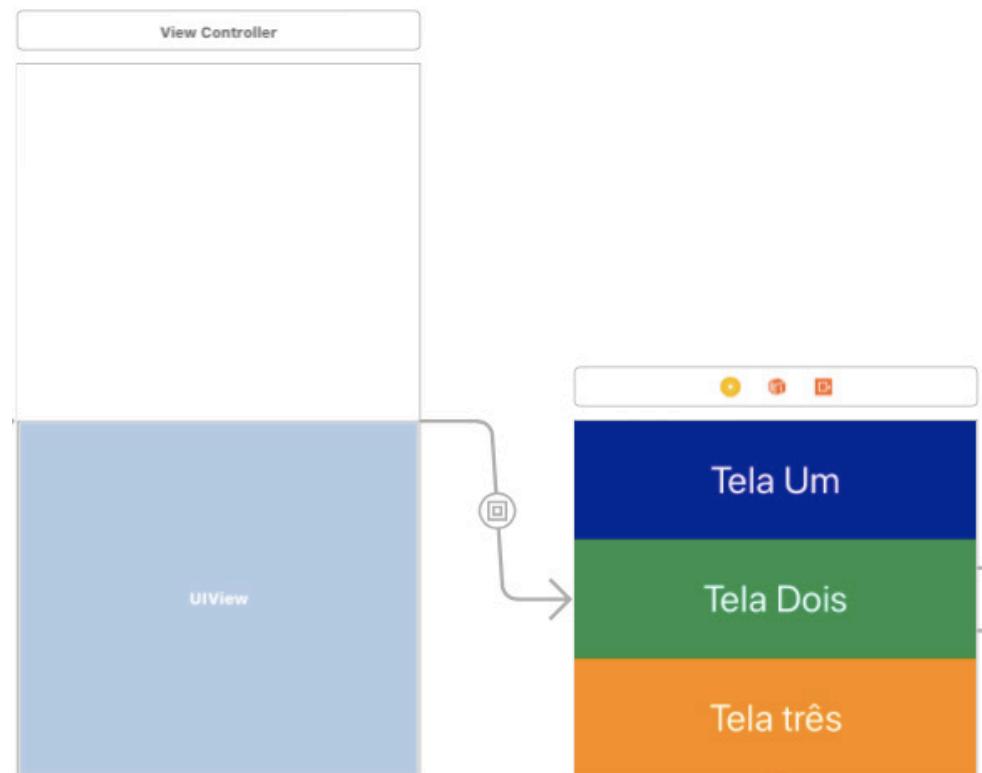
Seção 22-1

Container View



Com objetos do tipo **Container View** podemos combinar o conteúdo de vários views controllers em uma única interface.

O recurso é frequentemente utilizado para facilitar a navegação e para criar novos tipos de interface com base em um conteúdo existente.



Quando colocamos um objeto **Container View** em nossa interface, automaticamente uma segunda view controller é criada para ser exibida dentro da vista a ser utilizada como **container**.

É importante salientar que a classe que comanda essa nova view controller não é criada de forma automática, ficando a cargo do programador a sua implementação.



Capítulo 23



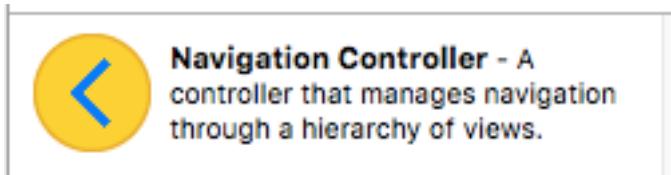
UINavigationController

Seção 23-1

UINavigationController

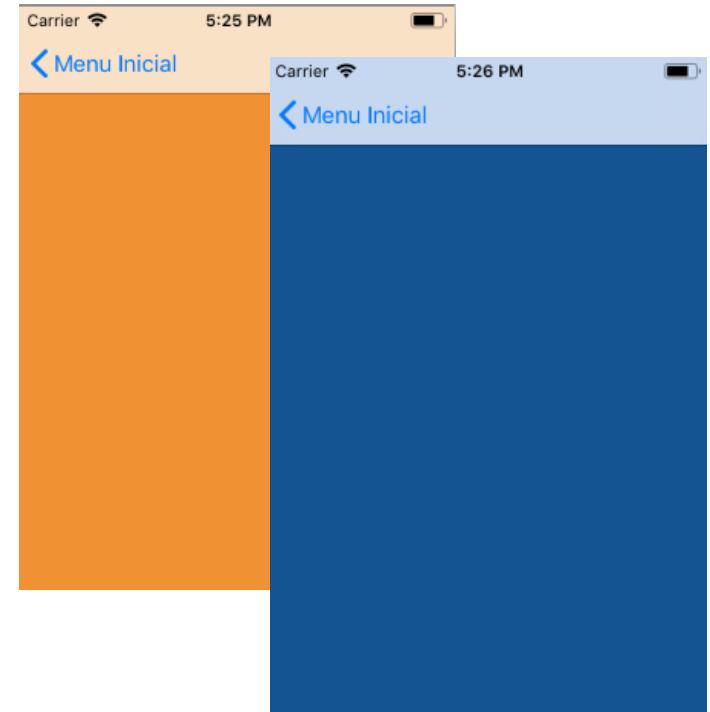
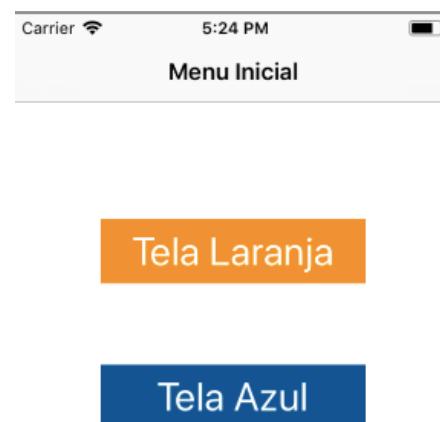
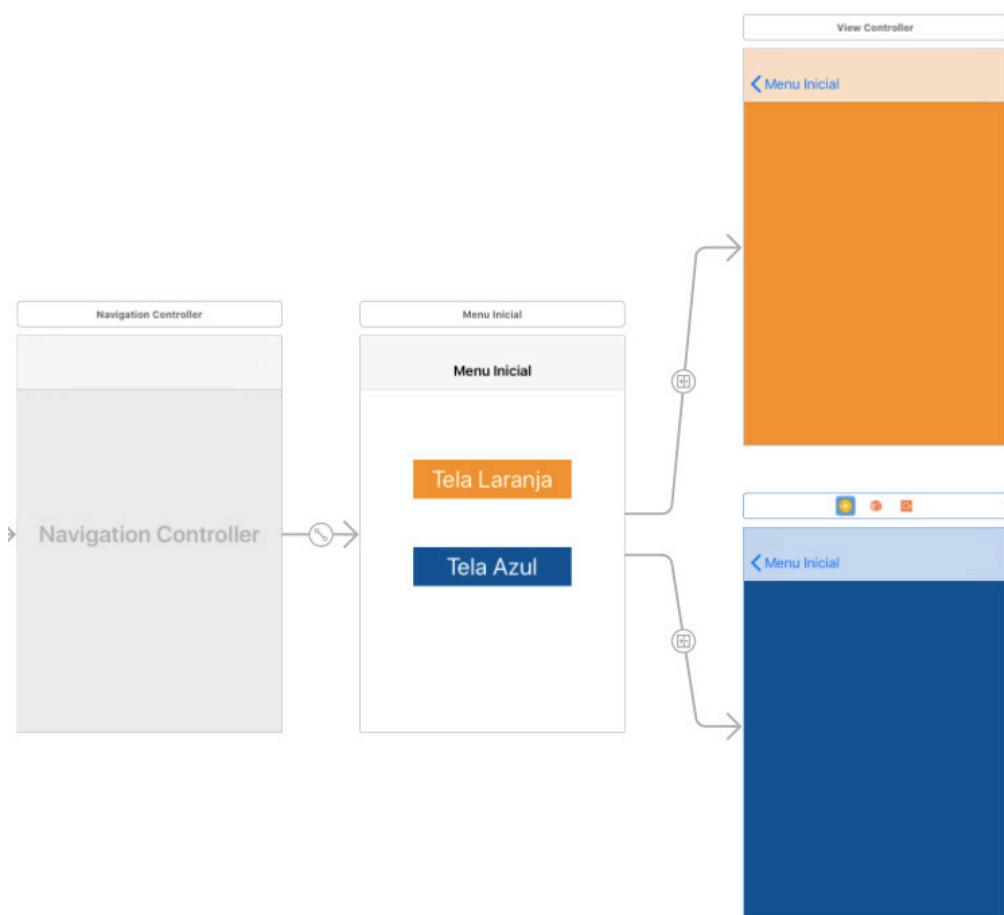


Herança: NSObject > UIResponder > UIViewController > UINavigationController



A classe **UINavigationController** implementa um **ViewController** especializado em gerenciar a navegação por hierarquia de conteúdo.

O uso de um **NavigationController** está associado a navegação sequencial, que tende a avançar os conteúdos por derivação, com a possibilidade de retornar ao conteúdo anterior.



Sempre quando avançarmos pela hierarquia, um botão **voltar** ficará disponível para retornarmos a tela anterior.

Sequenciando novas telas

O sequenciamento de telas dentro de hierarquia é feito de forma totalmente automática a partir de um simples **Segue** com a opção **Show**.

A partir do momento que lincamos dois objetos através de um segue, a nova view controller lincada passará a fazer parte da sequência hierárquica, ficando a cargo do programador a implementação de uma classe para comandar a nova view controller.

Seção 23-2

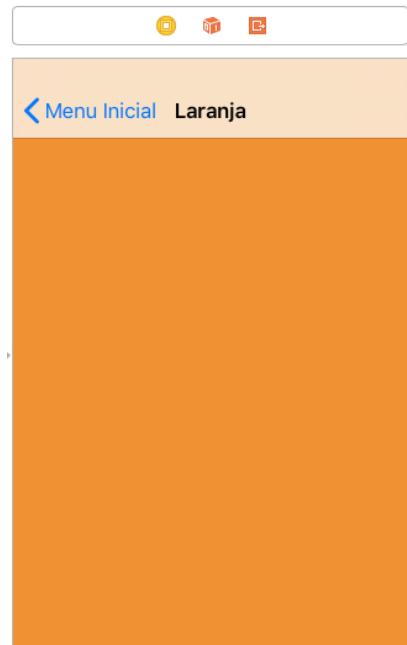
UINavigationItem



Herança: NSObject > UINavigationItem



Item responsável por colocar um **título** em uma View Controller que está atuando dentro da hierarquia de um **NavigationBar**.



Quando um título é definido para uma view controller, o botão **voltar** da próxima tela na hierarquia **apresentará esse título**, caso contrario, o botão voltar apresentará a palavra **Back**.

Capítulo 24



UITabBarController

Seção 24-1

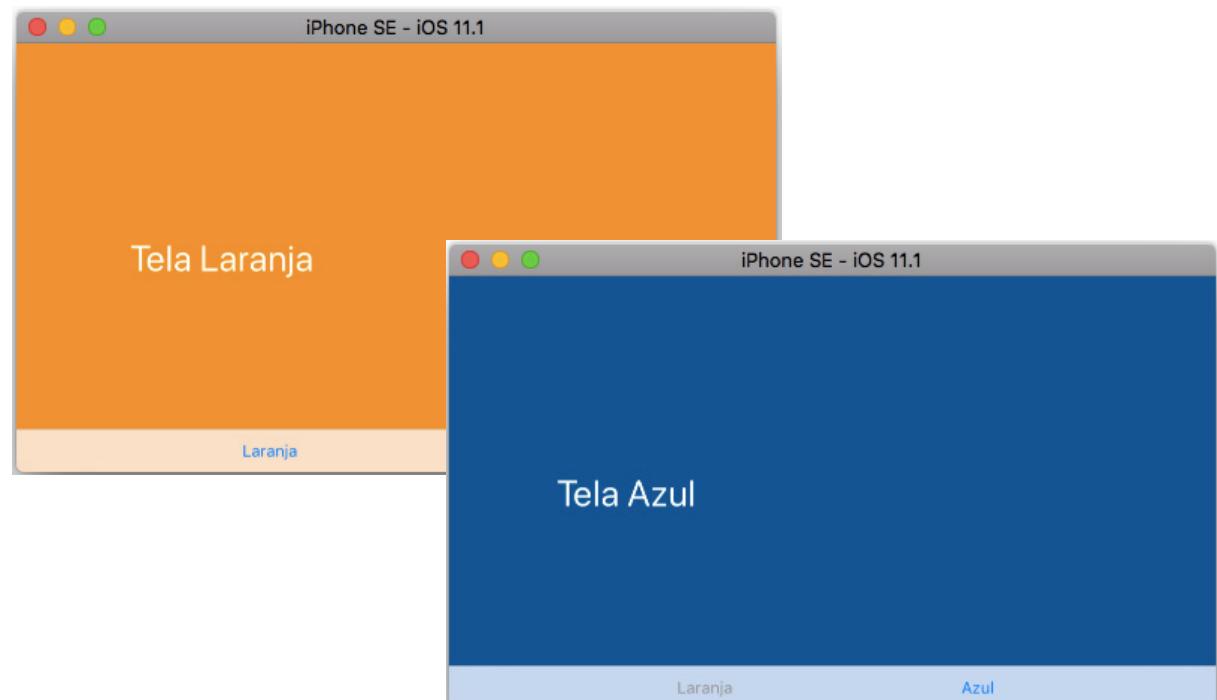
UITabBarController



Herança: NSObject > UIResponder > UIViewController > UITabBarController

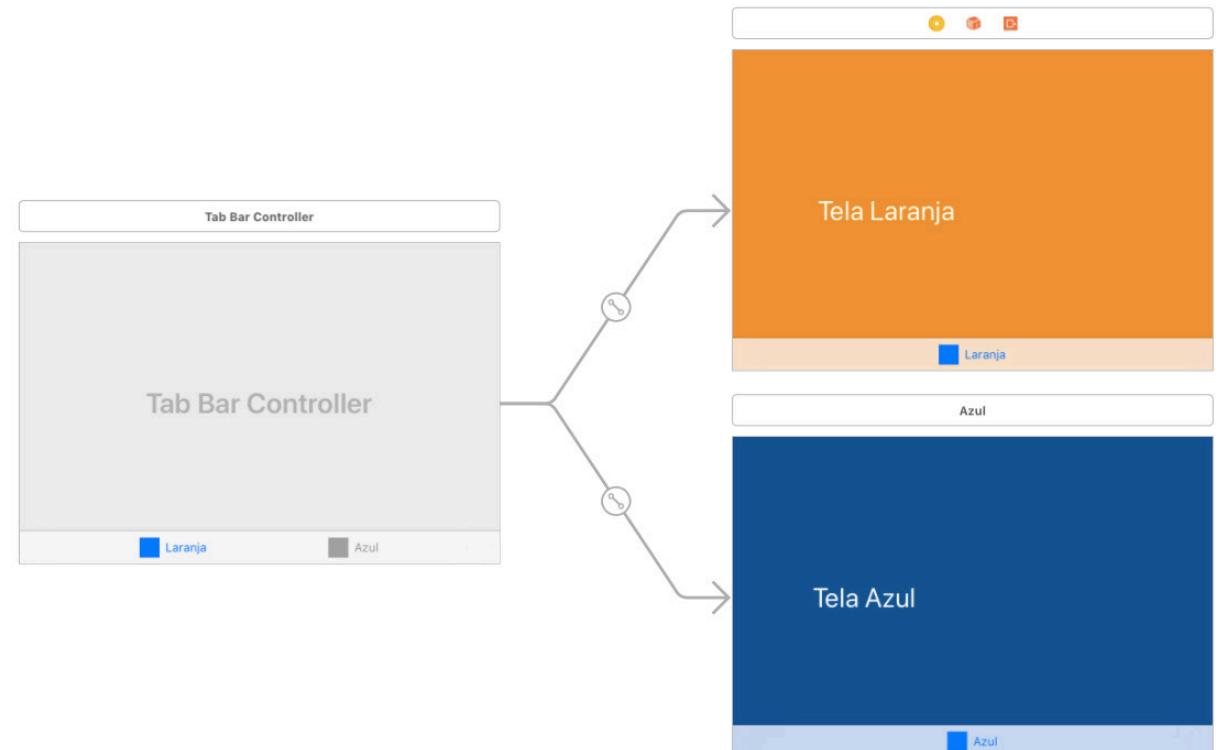


O **UITabBarController** é o controller responsável pelo gerenciamento de uma **TabBar** e suas telas através de uma barra localizada na parte inferior da tela.



As telas incluídas em uma **TabBar** podem variar em tipos (tabelas, view controllers, etc) e isso possibilita uma boa flexibilidade na produção.

Podemos desmembrar os elementos em UITabBar, que seria a barra propriamente dita, e UITabBarItem, que seria um dos itens da barra. A imagem a seguir ilustra um **UITabBarController** com duas telas:

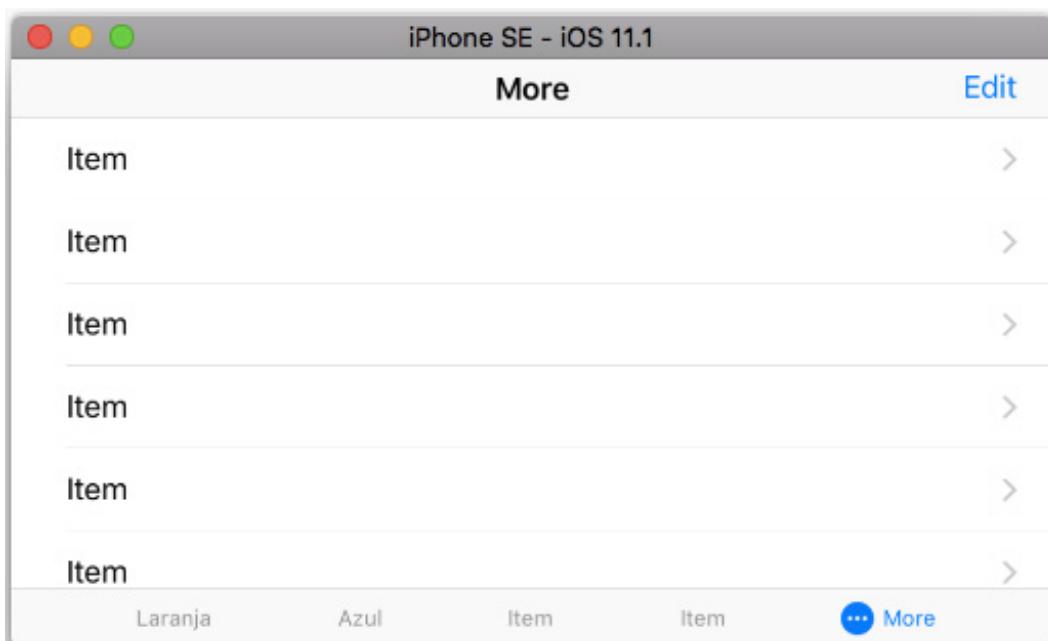
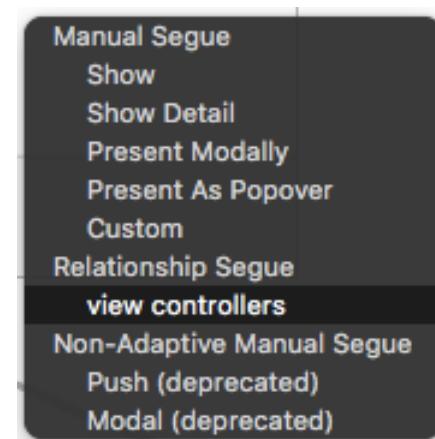


Em uma tabbar podemos ter uma infinidade de views, conforme as necessidades de cada projeto. Caso a quantidade de botões exceda a largura da tela, um botão **More...** será criado, agrupando cada um desses botões em um menu.

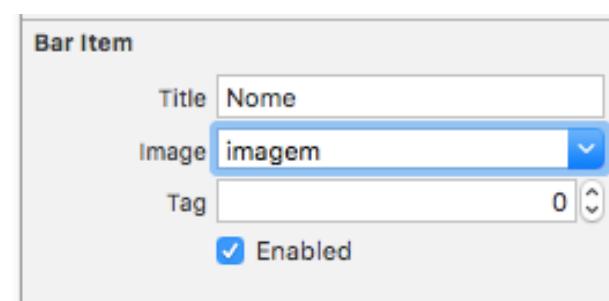


Relationship Segue

Para vincular uma nova view controller ao nosso sistema de tabbar, devemos no momento da realização do segue, utilizar a opção, **Relationship Segue - view controller**, para relacionar os dois objetos.



A partir daí, um novo botão será criado na tabbar, que pode ter seu título e aparência trabalhados através das opção de **Bar Item**, no painel **Attributes**.





Encapsulamento e Subscripts

Seção 25-1

Encapsulamento



Através do **encapsulamento** podemos definir diferentes níveis de acesso para as classes, propriedades e métodos.

Utilizamos este conceito quando queremos definir como nossas classes, propriedades e métodos serão acessados por outras classes ou objetos dentro da aplicação. Este conceito traz mais segurança e controle durante o desenvolvimento.

Dentre os níveis de encapsulamento existentes, três níveis se destacam e são importantes de serem abordados:

- **public** - Permite acesso aos elementos da classe por qualquer outra classe que tenha importado a fonte para a sua utilização. Normalmente, esse tipo de acesso é utilizado para especificar os itens públicos a serem utilizados em um framework.

- **internal** - Permite acesso apenas dentro do próprio bundle (pacote) do projeto. Esse é o padrão dos elementos criados dentro da Swift;

- **private** - Permite acesso apenas dentro da classe onde foi declarada. Caso seja necessário modificar os dados, é necessário métodos para alteração e acesso.

Um exemplo de encapsulamento é a variável **saldoBancário** de um cliente de banco.

Essa variável não pode ter um acesso público senão qualquer parte do programa poderia mudar o seu valor. Nesse caso definimos a variável como privada para que seu valor seja alterado usando métodos que irão conter mecanismos de travas dependendo da operação.

No exemplo aditante teremos dois arquivos **.swift** dentro de uma mesma aplicação. Teremos um arquivo que armazenará nossa classe, e posteriormente utilizaremos essa classe no arquivo principal (**main**)

```
//Arquivo da Classe:  
class Pessoa {  
    //Variáveis do objeto, sendo a idade privada  
    var nome : String  
    private var idade : Int  
  
    //Método que permite o acesso e altera a idade  
    func mudarIdade(novaIdade: Int){  
        idade = novaIdade  
    }  
  
    //Função que imprime o nome  
    func imprimeNome(){  
        print("Nome: \(nome)")  
    }  
}
```

Seção 25-2

Subscripts - Sintaxe



Subscript é mais uma grande novidade da linguagem de programação Swift. Com **subscripts** podemos acessar informações de tipos tais como, classes, estruturas e enumerações.

Um subscript é uma chave que utilizamos para extrair uma informação de uma coleção, podemos utilizá-lo para definir e recuperar valores de índice sem a necessidade de métodos distintos para configuração e recuperação.

Sua sintaxe é bastante simples:

```
subscript(index: Int) -> Int {  
    get {  
        // retorne algum valor.  
    }  
  
    set(newValue) {  
        // execute alguma ação  
    }  
}
```

Este acesso será realizado através de posicionamento de índices. Para facilitar a compreensão de subscripts veja o exemplos adiante, primeiro vamos preparar o arquivo com a classe a ser utilizada:

```
//Arquivo com a Classe Pessoa  
class Pessoa {  
    var nome = String()  
    var idade = Int()  
    private var cargos = [String]()  
  
    func definirCargo(comArray array : [String]) {  
        self.cargos = array  
    }  
  
    func perfil(){  
        print("Meu nome é \(nome) e tenho \(idade)")  
    }  
  
    subscript(index : Int) -> String{  
        get{  
            return self.cargos[index]  
        }  
  
        set(novoValor){  
            self.cargos += [novoValor]  
        }  
    }  
}
```

Agora vamos trabalhar com o arquivo principal da aplicação:

```
//Arquivo principal da aplicação

//Instanciando a classe Pessoa
var jose = Pessoa()

//Criar array de cargos
var arrayCargos = [String]()

//Definição dos elementos da classe instanciada
jose.nome = "José"
jose.idade = 40
jose.definirCargo(comArray : arrayCargos)
jose[0] = "Gerente"

//Imprimindo informações em console
jose.perfil()
print(jose[0])
```

Note que conseguimos fazer uma atribuição a um índice do array **cargo**, por conta da definição do **subscript** na classe **Pessoa**.

#Dica!

Como estamos trabalhando com **arrays**, é interessante tratarmos o acesso a **índices ainda nulos**, caso contrário é possível que a aplicação quebre.



Tratamento de erros



Tratamento de erros é o processo de responder e recuperar as condições de erro em sua aplicação. A Swift fornece suporte para lançar, capturar, propagar e manipular erros recuperáveis em tempo de execução.

Algumas operações garantidas com relação a sua execução de serem. Em alguns casos utilizamos opcionais para representar valores nulos, mas quando uma operação falha, muitas vezes é importante entender o que causou essa falha, para que o código possa responder adequadamente.

Representando e lançando erros

É possível representar e lançar erros para relatar que algo inesperado ocorreu e houve a necessidade de alterar o fluxo natural do sistema. A representação de erros pode ser feita através de um **enum**:

Vamos tomar como exemplo, os erros possíveis em tentativas de acesso com logins e senhas, e a partir daí criarmos uma estrutura de erros:

```
//Criando representações de erros
enum ErroDeAcesso : Error {
    case loginESenhaErrados
    case senhaErrada
    case loginErrado
}
```

Na Swift, os erros são representados por valores de tipos que estão em conformidade com o protocolo de **Error**, que foi adotado pelo enum que representa o conjunto de erros. Esse protocolo vazio indica que um tipo pode ser usado para manipulação de erros.

Agora vamos criar duas variáveis que representarão uma tentativa de conexão:

```
//Tentativa de conexão
var login = "Admin"
var senha = "admin"
```

Em seguida, vamos criar a função que efetivará um login ou então lançará um erro:

```
//Função para efetuar o login ou lançar um erro
func efetuarLogin(umLogin : String, umaSenha : String) throws {
    if umLogin != login && umaSenha != senha{
        throw ErroDeAcesso.loginESenhaErrados
    } else if umLogin == login && umaSenha != senha{
        throw ErroDeAcesso.senhaErrada
    }
}
```

```

} else if umLogin != login && umaSenha == senha {

    throw ErroDeAcesso.loginErrado
}

print("Login Efetuado")
}

```

Utilizando o do, try e catch para tratar os erros

Por ultimo temos que tratar os erros quando eles ocorrem na tentativa de acesso, como é o caso da nossa aplicação. Para tal, utilizaremos a estrutura composta pelos comandos **do, try e catch**.

Com o **do**, vamos obrigar a aplicação a fazer uma tentativa de acesso com a função que criamos para tal:

```

do{

    //Diretriz de acesso
    try efetuarLogin(umLogin: "Admin", umaSenha:
        "admin")

}

```

Já com o **catch**, vamos pegar os erros lançados e tratá-los com uma resposta em console:

```

catch ErroDeAcesso.loginESenhaErrados{

    print("Login e senha errados")
}

catch ErroDeAcesso.loginErrado{

    print("Login digitado está errado")
}

catch ErroDeAcesso.senhaErrada{

    print("Senha digitada está errada")
}

```

Apêndice 1



Caderno de Exercícios

Exercícios relacionado ao Capítulo 1

Simulação e debugging



Exercício 1

Faça a simulação de alguma aplicação desenvolvida anteriormente em um dispositivo físico.

Exercício 2

Dado o código adiante, faça a solução necessária:

```
class ViewController: UIViewController {  
    var nome : String  
}
```

! Class 'ViewController' has no initializers ×

Exercícios relacionados ao Capítulo 2 Protocols



Exercício 1:

Crie um protocolo para ser utilizado como base para a criação de diferentes classes de carros, onde tenha como propriedades o fabricante, modelo, cor e ano de fabricação.

Utilize esse protocolo para a criação de ao menos três diferentes objetos.

Exercícios relacionados ao Capítulo 3

Extensões



Exercício 1:

Aproveite uma das classes para carro criadas no ultimo capítulo, e faça uma extensão. Adicione a essa extensão um método que imprima as suas propriedades.

Exercícios relacionados aos Capítulos 4 e 5

UIResponder / Trabalhando com textos



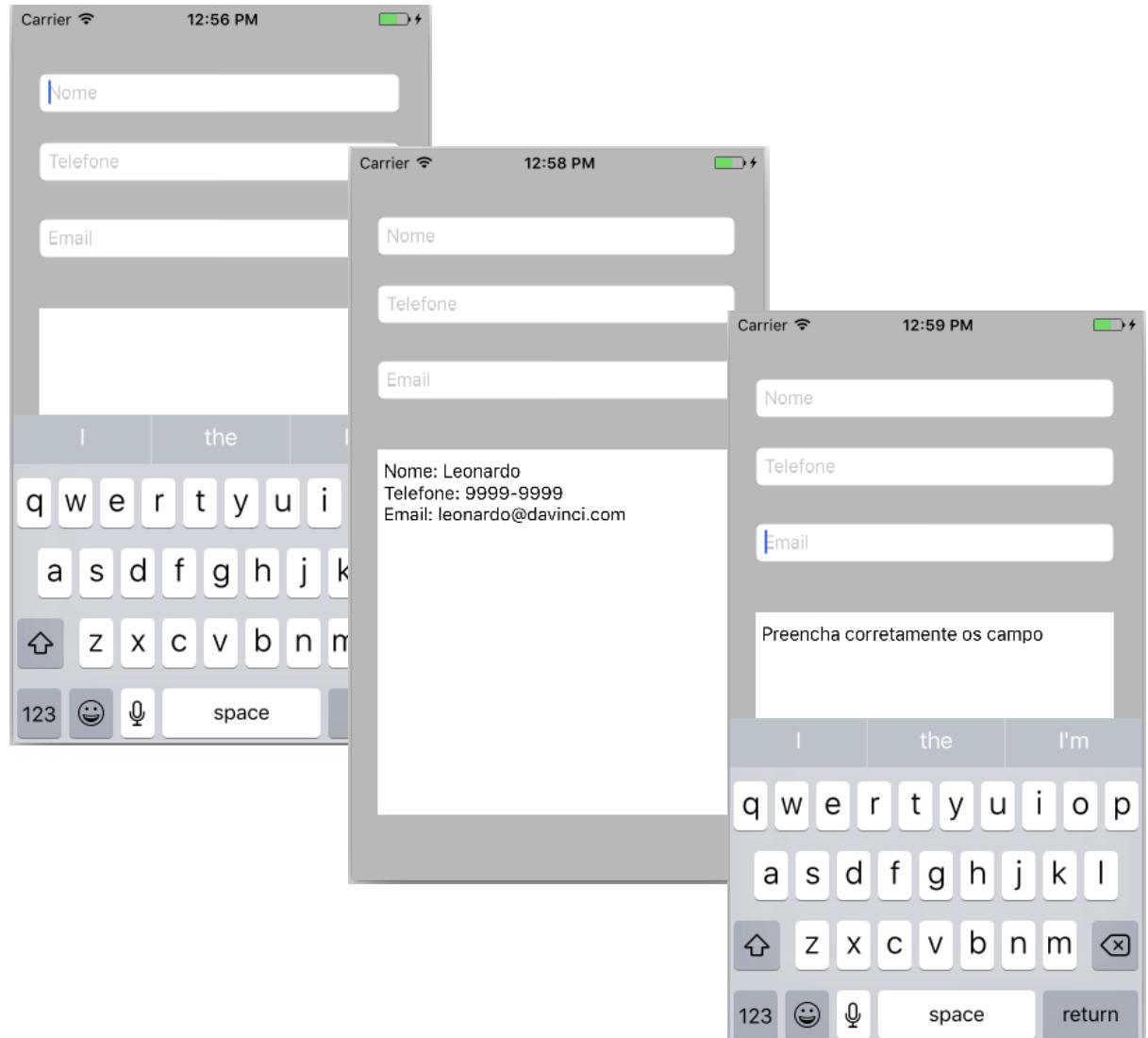
Exercício 1

Crie uma aplicação que contenha três **textFields** e uma **textView**, onde usuário deverá preencher todas as textFields e clicar em **return** no teclado para passar para o próximo campo.

É importante que, assim que a aplicação entrar em ação, o primeiro campo já tenha o cursor posicionado para a digitação.

Caso todos os campos estejam preenchidos, deverá ser montada uma espécie de **ficha com as informações**, o texto deve ser exibido na **textView**, e as textFields devem ser limpas e o teclado deve ser retirado da tela.

Caso haja campos vazios, a **textView** deverá receber o seguinte texto:
Preencha corretamente os campos.



Exercícios relacionados ao Capítulo 6

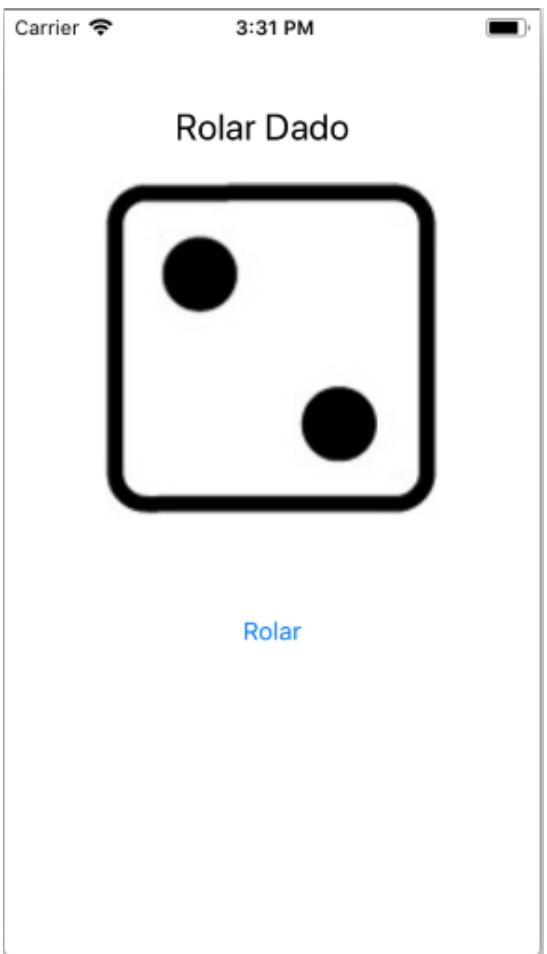
Trabalhando com imagens



Exercício 1

Criar uma aplicação que contenha uma **label** (servirá de título), uma **imageView** e um **button**.

A clicar no button, será exibida uma imagem aleatória referente a um dos lados de um dado D6.



Exercícios relacionados ao Capítulo 7

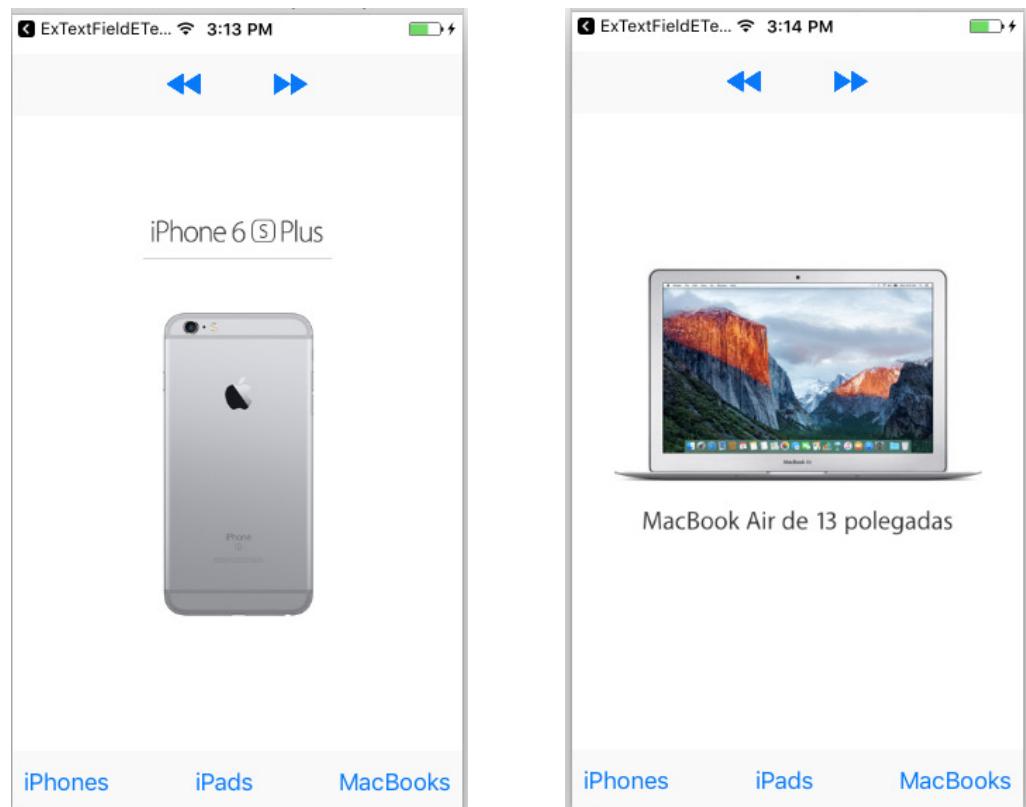
Barras de ferramentas - Toolbar



Exercício 1

Criar uma aplicação que contenha duas **toolbars**, uma localizada na parte superior da tela do aplicativo, e outra na parte inferior. Ao centro, vamos locar uma **imageView**.

A toolbar acima contém dois **buttons** que criam uma espécie de barra de navegação entre as imagens existentes para cada produto. A toolbar abaixo varia os produtos entre **iPhone, iPads e Macbooks**.



Exercícios relacionados aos Capítulos 8 e 9

UIProgressView / UIActivityIndicatorView



Exercício 1

Crie uma aplicação que contenha os seguintes itens:

- 2 text fields;
- 1 progress view;
- 1 activity indicator;
- 1 button;
- 1 label.

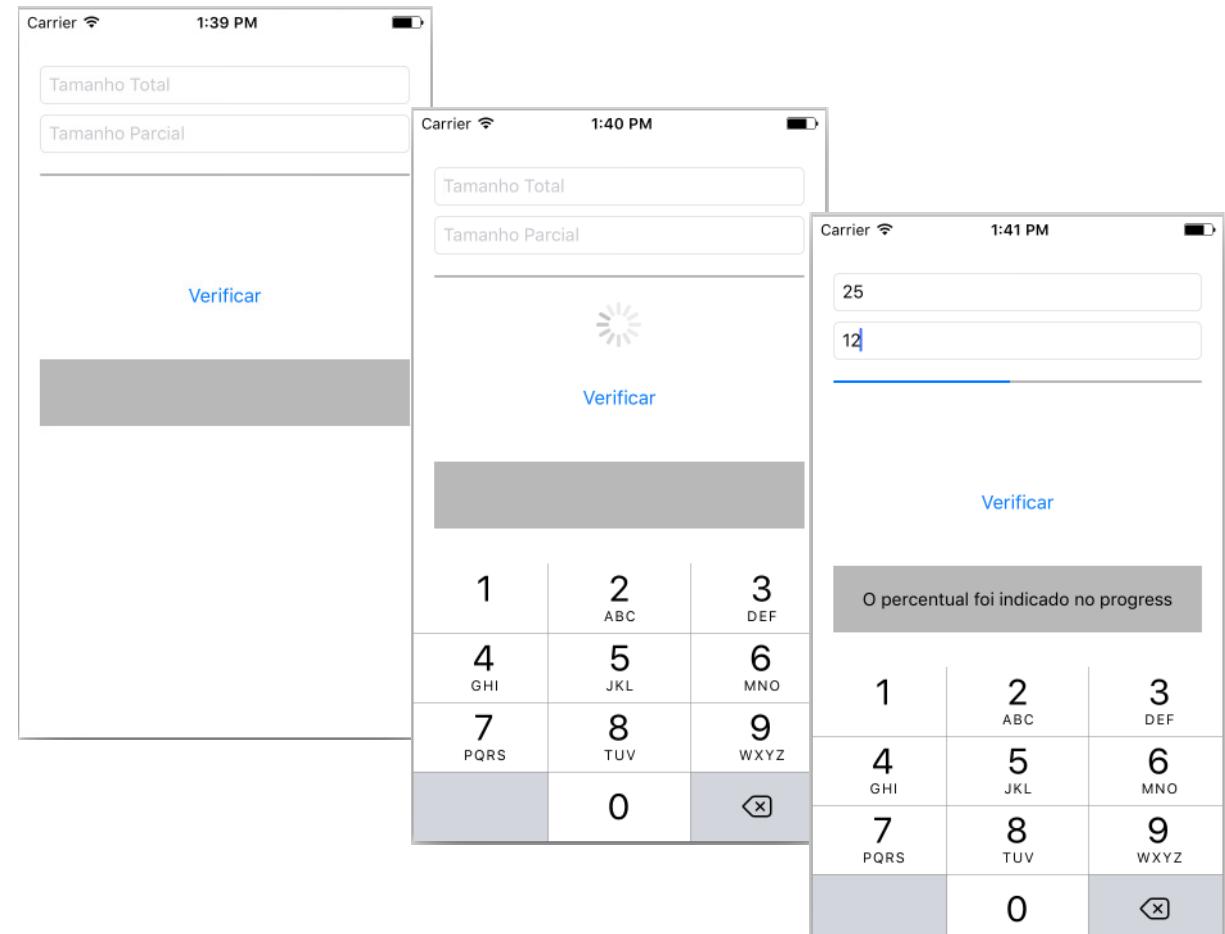
As textFields trabalharão o teclado numérico e receberão dois valores, um valor total e um valor parcial.

Enquanto o usuário não finalizar o processo de preenchimento dos campos, o activityIndicator de permanecerá ativo.

Quando o usuário inserir os valores e clicar no botão, o percentual de relação entre o valor parcial e o valor total será exibido no progressView e uma mensagem exibida na label avisando do ocorrido.

- Caso o valor parcial seja maior que o total, uma mensagem também deve ser exibida informando o usuário e o progress não terá seu valor alterado.

- Caso o usuário não preencha os campos, uma mensagem também deve ser exibida.



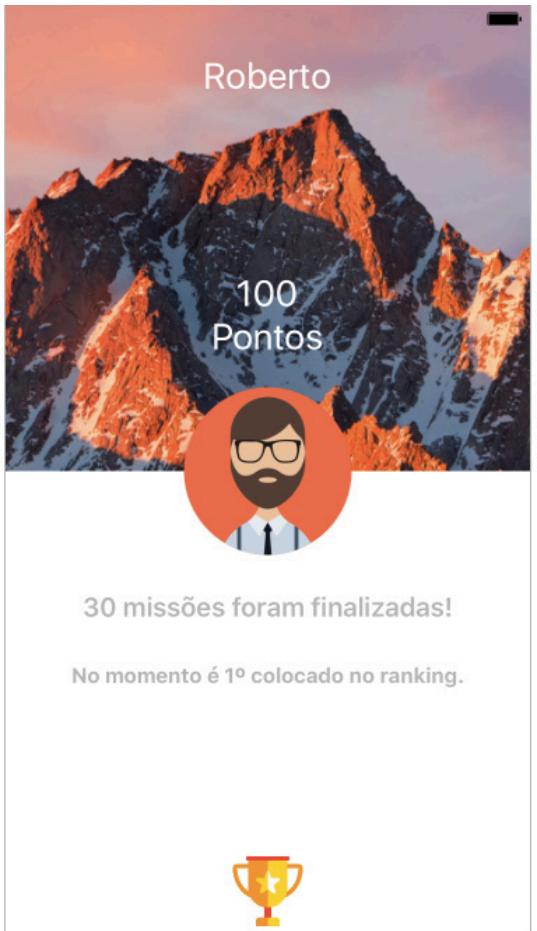
Exercícios relacionados ao Capítulo 10

Autolayout



Exercício 1

Aplicando os conceitos de **AutoLayout** desenvolver uma aplicação que se adeque aos diversos tamanhos de tela em **portrait** conforme o layout indicado, utilizando seguintes imagens.



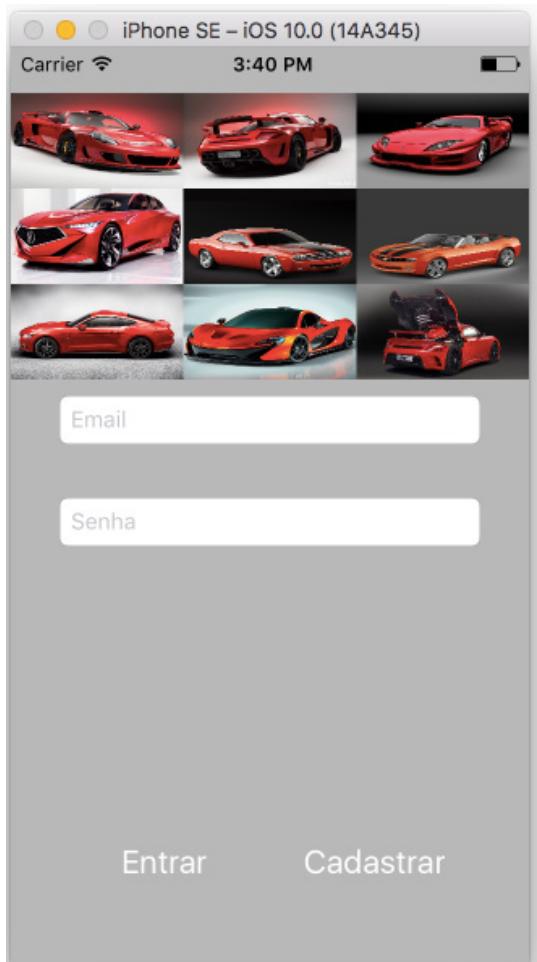
Exercícios relacionados ao Capítulo 11

UIStackView



Exercício 1

Utilizando os conceitos de **StackView** e **Autolayout** criar uma interface conforme exibida no exercício.



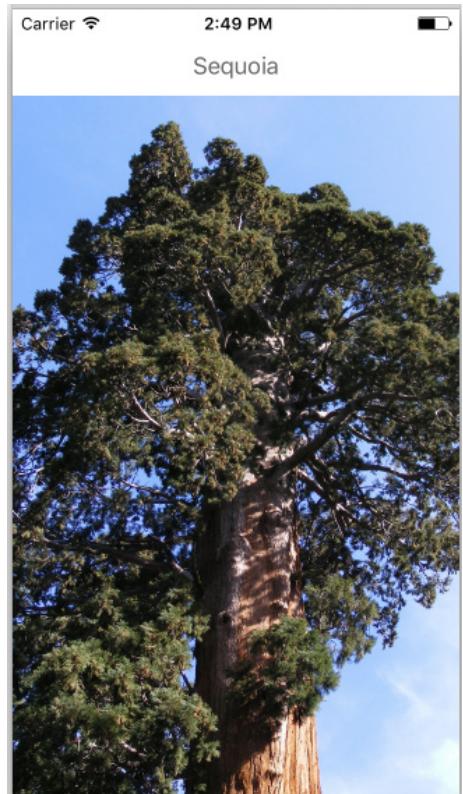
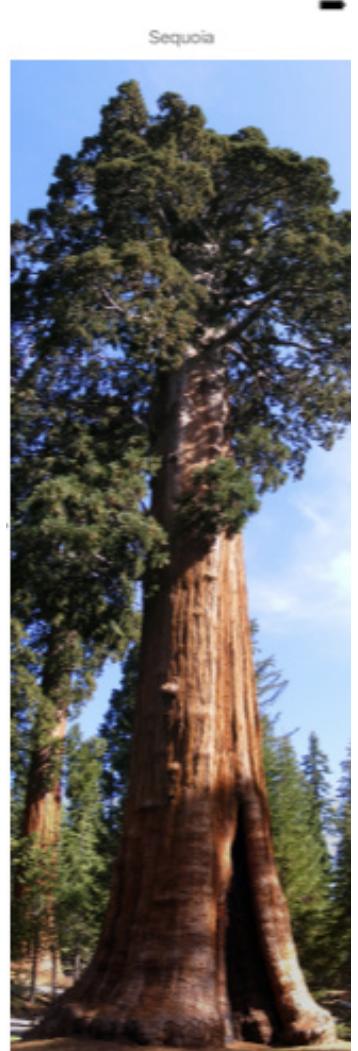
Exercícios relacionados ao Capítulo 12

UIScrollView



Exercício 1

Crie uma interface gráfica que exiba uma imagem completa partindo do princípio que ela seja verticalmente maior do que o dispositivo e o que obrigue a utilização de um scroll na tela.



Exercícios relacionados ao Capítulo 13

Closures



Exercício 1:

Crie uma closure que imprima em console o valor de dois números e a soma de ambos:

******* Closure para soma de dois valores *******

Primeiro valor informado: 10

Segundo valor informado: 20

A soma dos dois valores é: 30

Exercício 2:

Crie uma closure que imprima em console o valor múltiplos números e a soma de todos eles:

******* Closure para soma de múltiplos valores *******

Os números indicados foram: [1, 2, 3, 10]

O resultado da soma dos valores é 16

Exercícios relacionados ao Capítulo 14

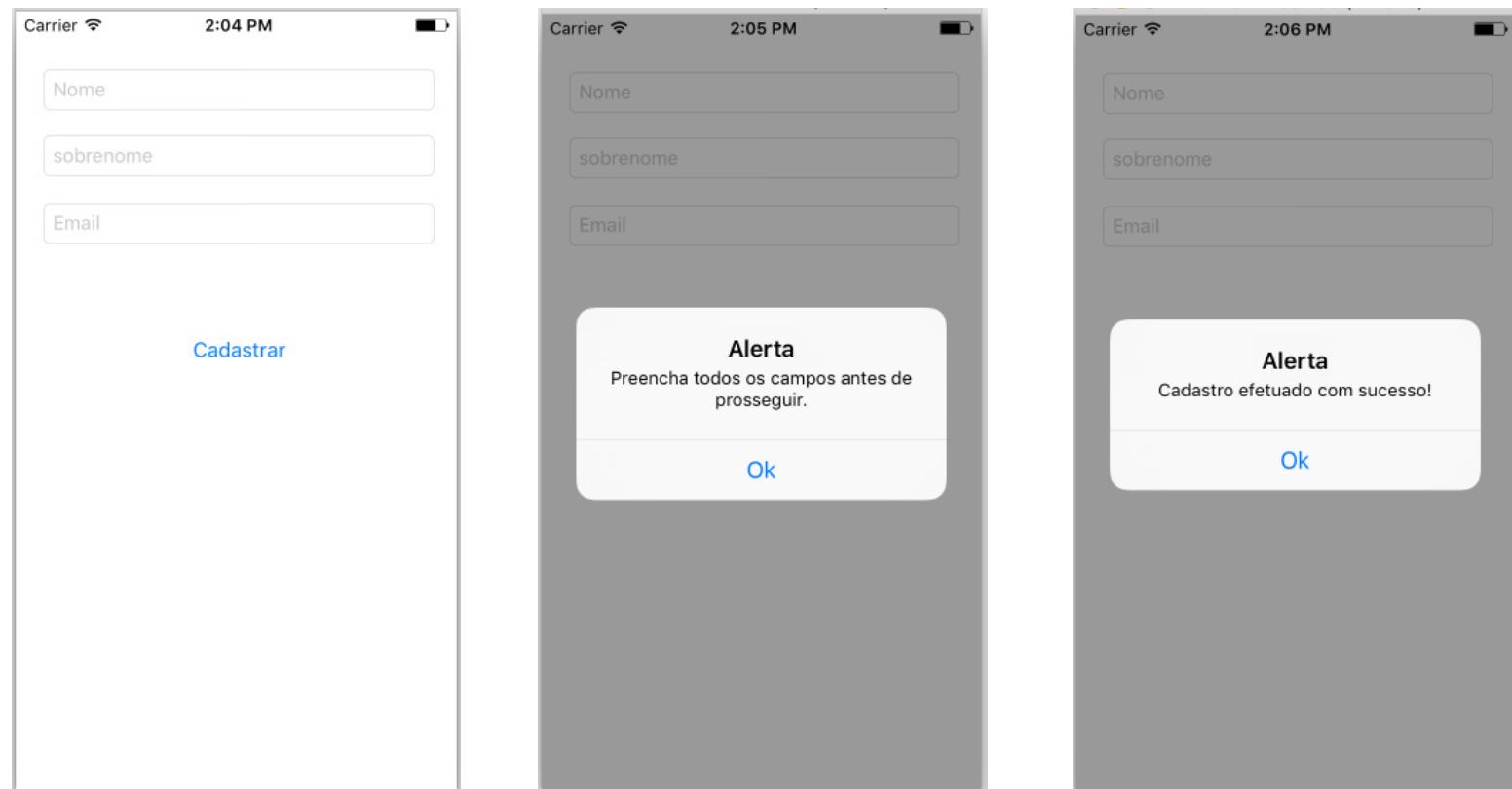
Trabalhando com Alertas



Exercício 1

Criar uma aplicação que contenha três **textFields** e um **button** de cadastro. Caso o usuário clique em **cadastrar** antes de preencher todos os campos, deverá ser exibido um **alerta** avisando o ocorrido.

Caso os campos tenham sido preenchidos corretamente deverá ser exibido um **alerta** avisando que o cadastro foi efetuado e os campos de texto devem ser limpos.



Exercícios relacionados ao Capítulo 15

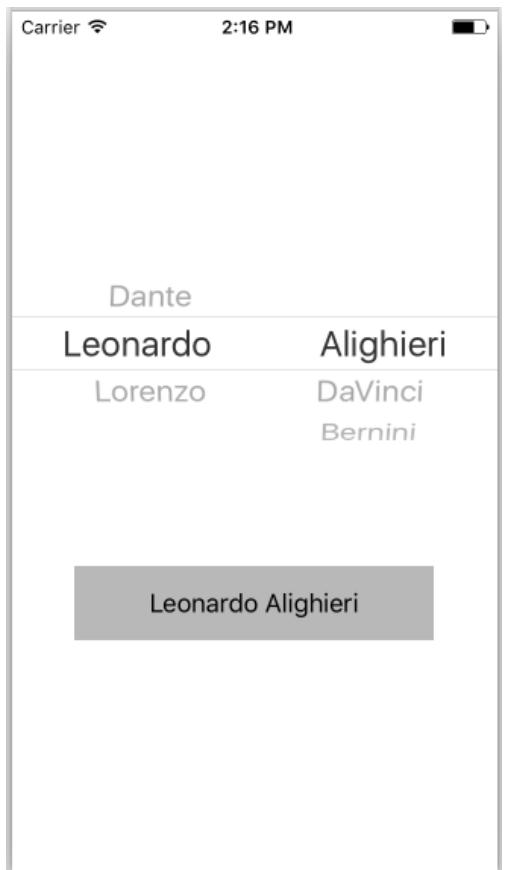
UIPickerView



Exercício 1

Criar uma interface que contenha um **pickerview** com **duas colunas** e uma **label**. Na coluna da esquerda o pickerView deve exibir nomes. Já na da direita teremos sobrenomes.

A cada seleção feita, o objeto deverá exibir o nome e o sobrenome escolhido na label.



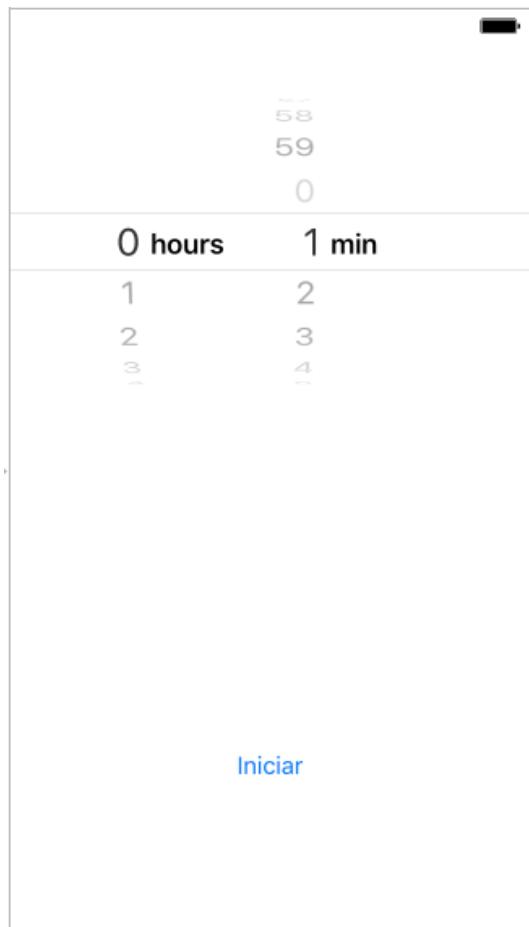
Exercícios relacionados ao Capítulo 16

UIDatePicker, Date e Timer



Exercício 1

Criar uma aplicação que a cada segundo troque a cor do fundo da tela aleatoriamente por tempo determinado. A escolha do tempo deve ser feita por meio de um **DatePicker**.



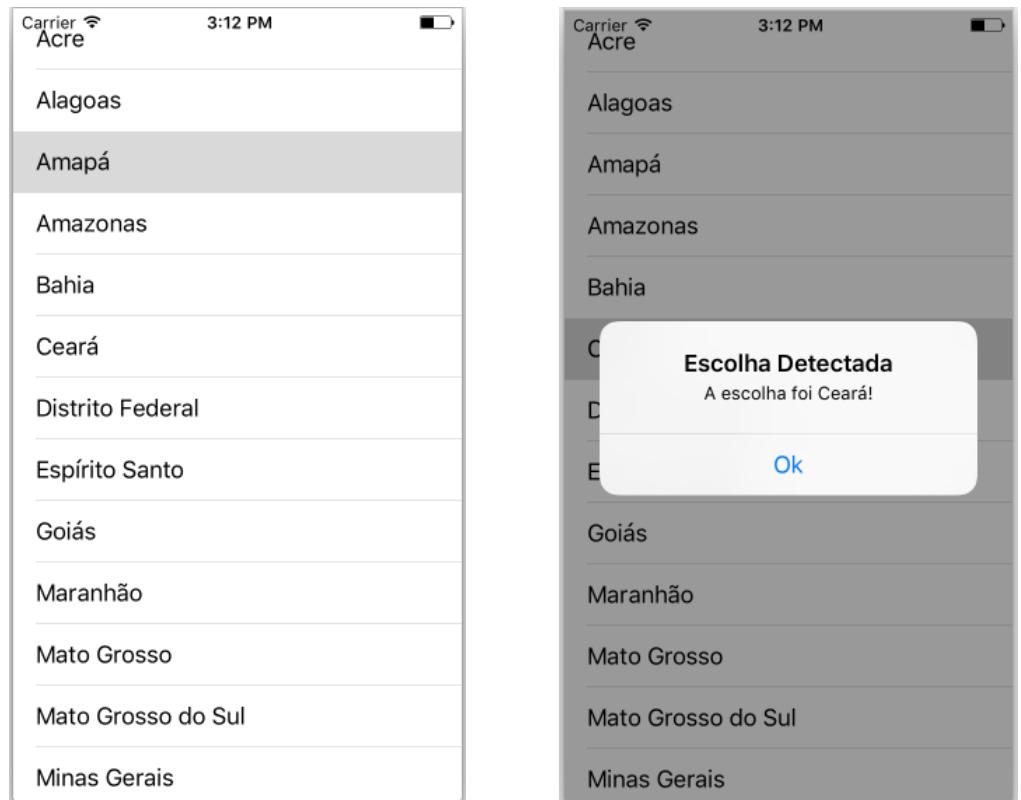
Exercícios relacionados ao Capítulo 17

UITableView e UITableViewController



Exercício 1

Criar uma **tableView** que exiba os estados do Brasil. Caso haja uma seleção de um dos estados, deverá ser exibido um alerta indicando o estado escolhido.



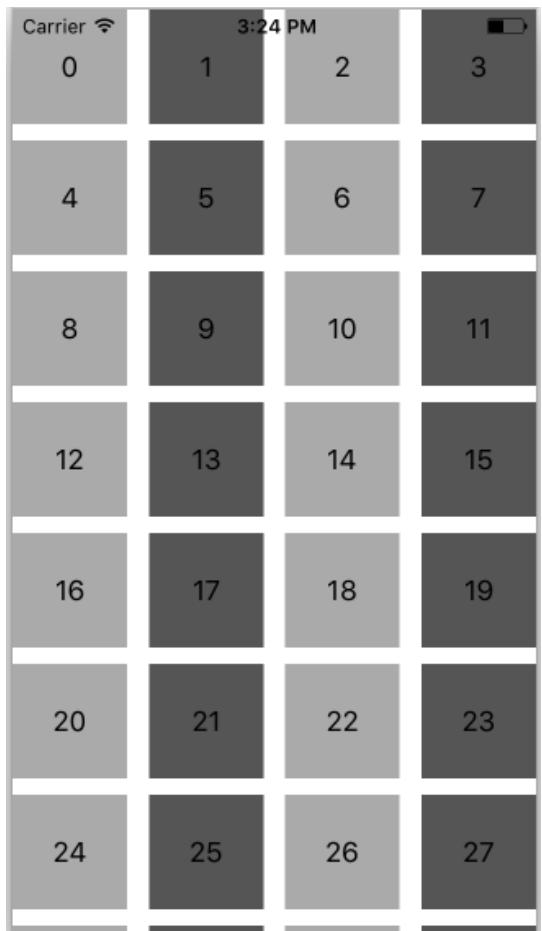
Exercícios relacionados ao Capítulo 18

UICollectionView



Exercício 1

Criar uma **CollectionView** onde o background de cada célula se intercale entre cinza claro e cinza escuro de forma automática, e o número do itens (de 0 a 50) seja exibido no centro do elemento.



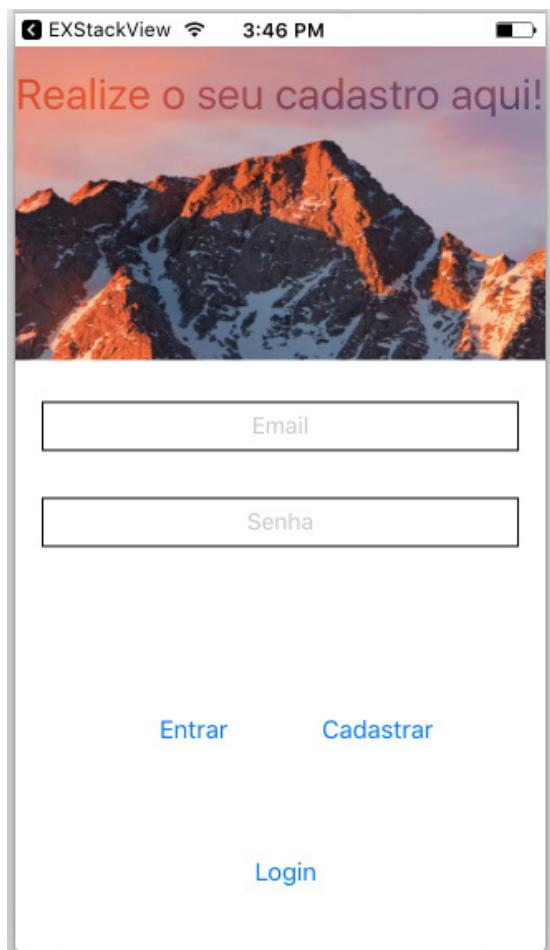
Exercícios relacionados ao Capítulo 19

UIVisualEffects



Exercício 1

Crie uma interface conforme demonstrado utilizando os conceitos de **AutoLayout** e **UIVisualEffects**.



Exercícios relacionados ao Capítulo 20

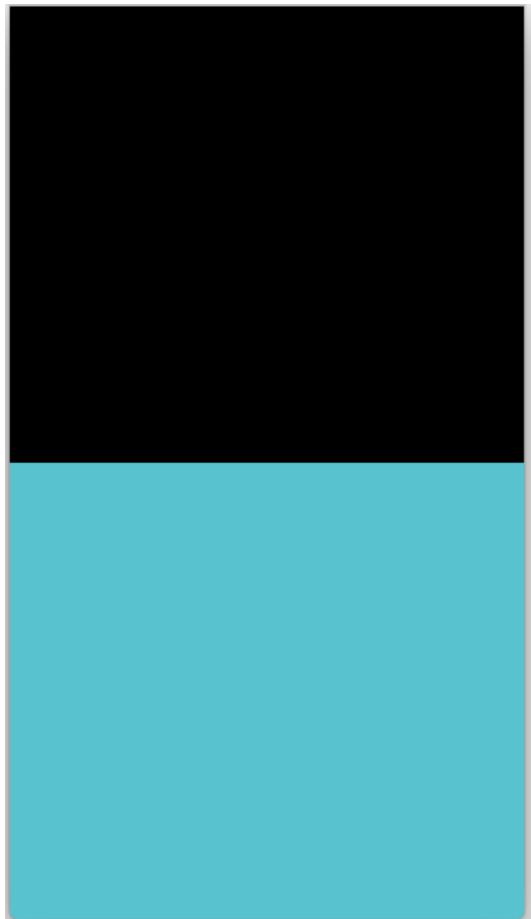
Eventos de toque e UITapGestureRecognizer



Exercício 1

Crie uma interface que contenha duas **views** que ocupem metade do tamanho da tela cada uma. Quando a view localizada acima for tocada, deverá ficar branca e quando o toque for finalizado ela voltará a ser preta.

Já a view na parte inferior terá a cor de background alterada aleatoriamente a cada inicio de toque.



Exercícios relacionados ao Capítulo 21

Trabalhando com multitelas



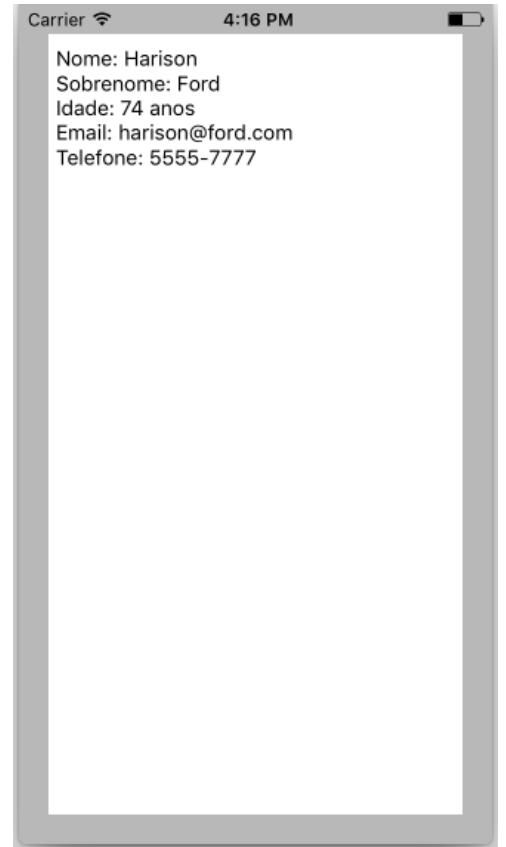
Exercício 1

Crie uma aplicação que contenha duas telas. Na primeira tela teremos um **button** e cinco **texfields** que resgatam os seguintes valores inseridos pelo usuários:

- nome
- sobrenome
- idade
- email
- telefone

Caso o usuário preencha todos os campos e clique no botão, deverá acontecer o direcionamento para a outra tela que conterá um **texview** que representará uma ficha com os valores preenchidos na tela anterior.

Caso os campos não estejam preenchidos o usuário permanecerá na mesma tela.



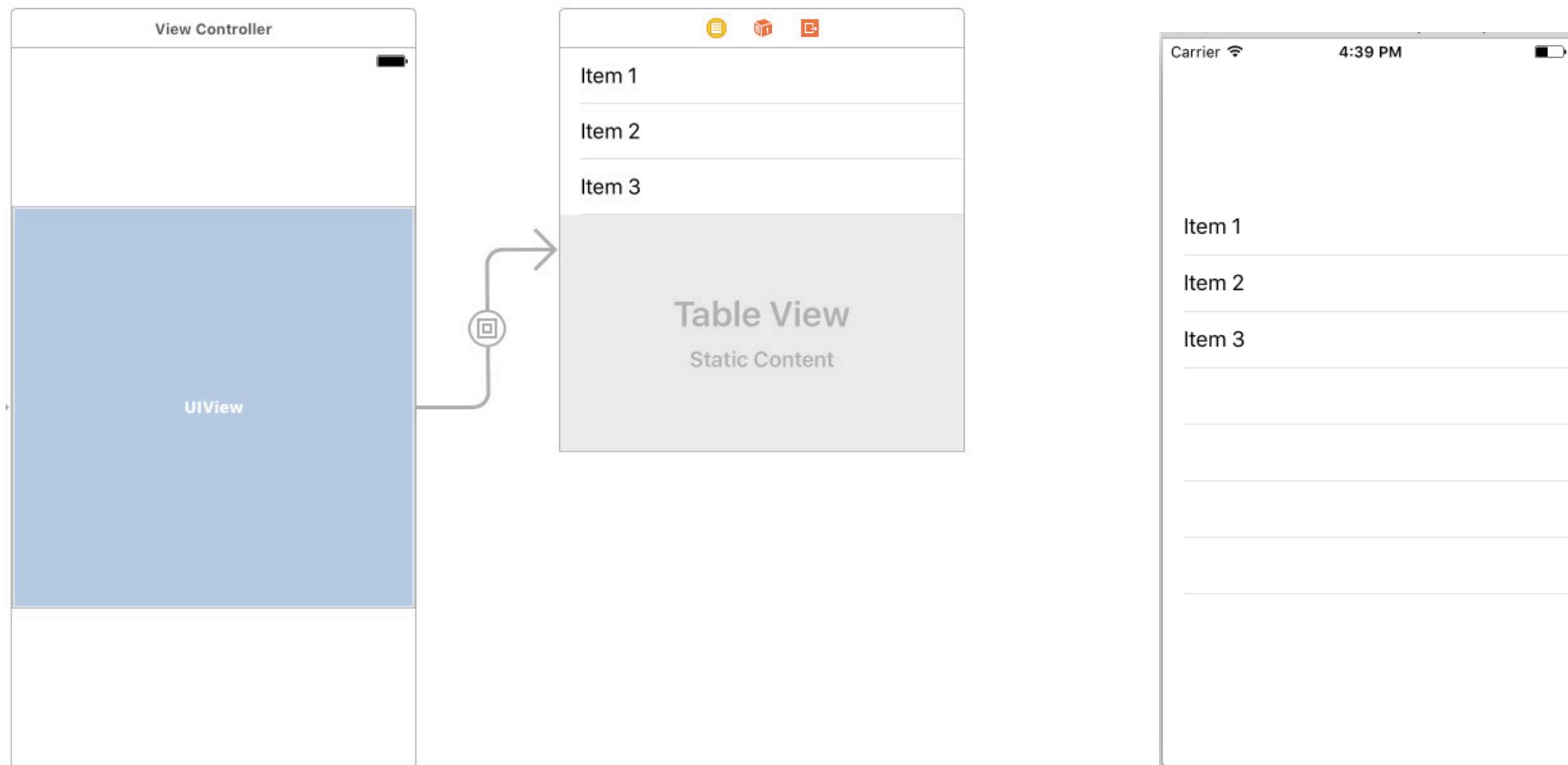
Exercícios relacionados ao Capítulo 22

Container View



Exercício 1

Crie uma aplicação que contenha um **Container View**. Essa container view deverá exibir uma **tableView** estática centralizada conforme o exemplo:



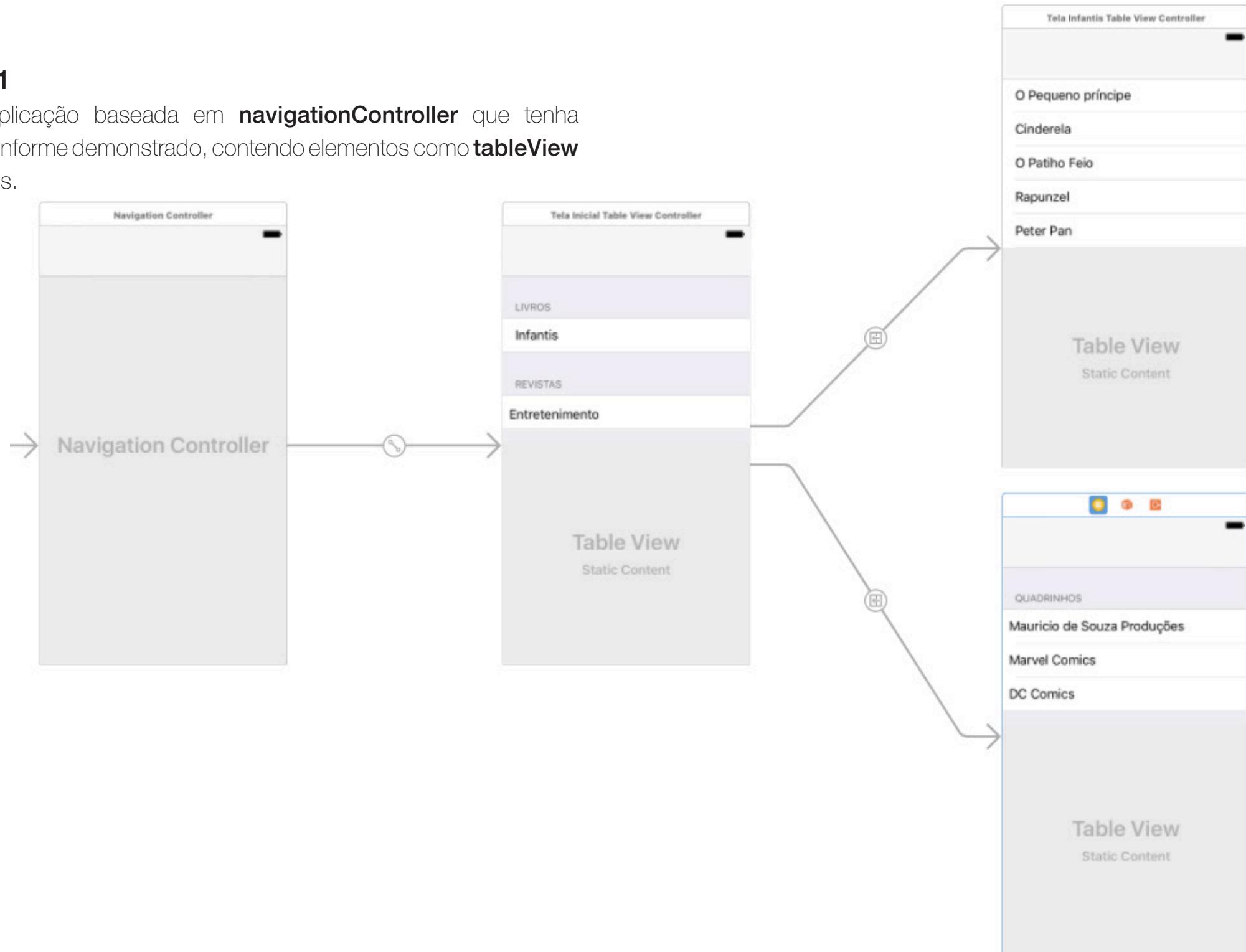
Exercícios relacionados ao Capítulo 23

UINavigationController



Exercício 1

Crie uma aplicação baseada em **navigationController** que tenha funcionamento conforme demonstrado, contendo elementos como **tableView** em telas diferentes.



Exercícios relacionados ao Capítulo 24

UITabBarController

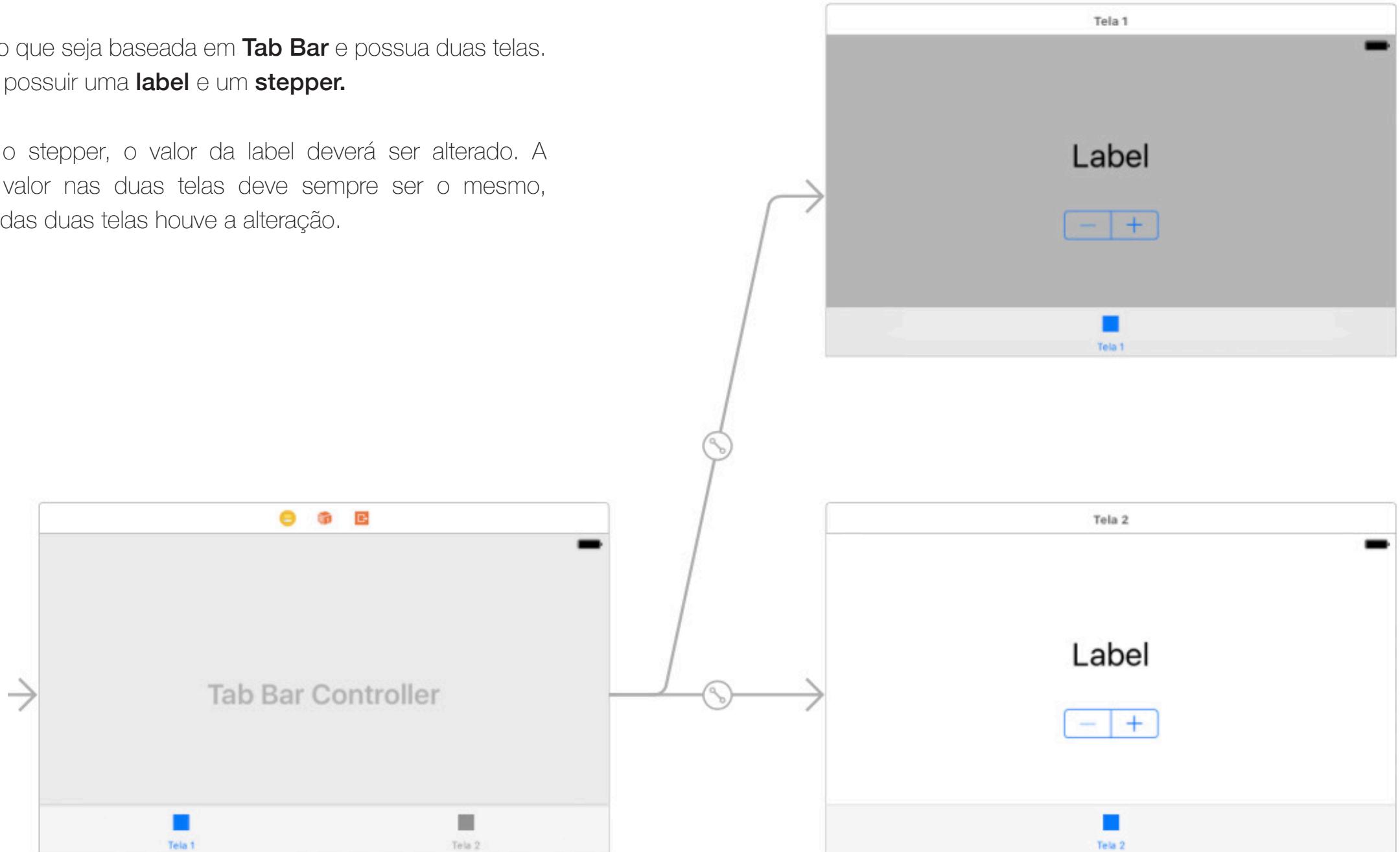


Exercício 1

Crie uma aplicação que seja baseada em **Tab Bar** e possua duas telas.

Ambas às telas devem possuir uma **label** e um **stepper**.

Ao interagir com o stepper, o valor da label deverá ser alterado. A observação é que o valor nas duas telas deve sempre ser o mesmo, independente em qual das duas telas houve a alteração.



Exercícios relacionados ao Capítulo 25

Encapsulamento e Subscripts



Exercício 1:

Dada a escalação da seleção da copa de 70: Felix, Brito, Piazza, Carlos Alberto, Clodoaldo, Jairzinho, Gérson, Tostão, Pelé, Rivelino e Everaldo, crie:

- Uma classe para o time;
- Dentro da classe, determine que a escalação titular é privada;
- Crie funções e subscripts para possibilitar a impressão em console do time e a alteração de jogadores;
- Como sugestão de alteração, troque o goleiro Felix pelo goleiro Leão.