PROF. PEDRO HENRIQUE

DESENVOLVIMENTO PARA IOS 11 COM SWIFT 4

GITHUB DA TURMA

HTTPS://GIT.IO/VF50W

ONDE ENCONTRAR O MATERIAL?

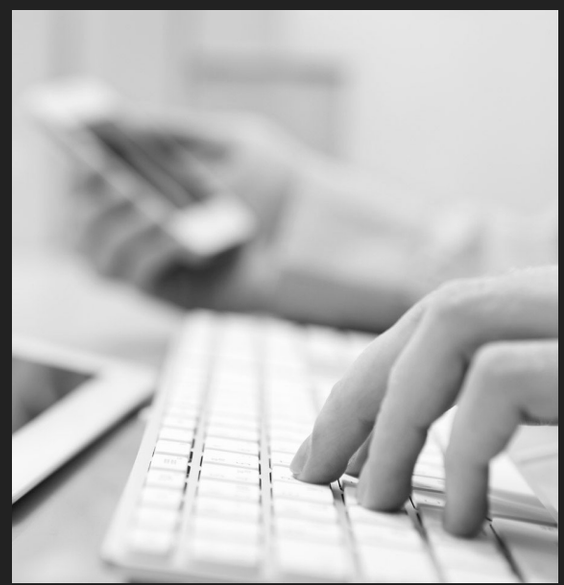
LEIA O QRCODE

ALÉM DO TRADICIONAL BLACKBOARD DO IESB

O QUE VAMOS FAZER HOJE?

AGENDA

- Review
- Retain Cycle - Pergunta de entrevista
- Navegação (closures e Notification)
- Ciclo de vida



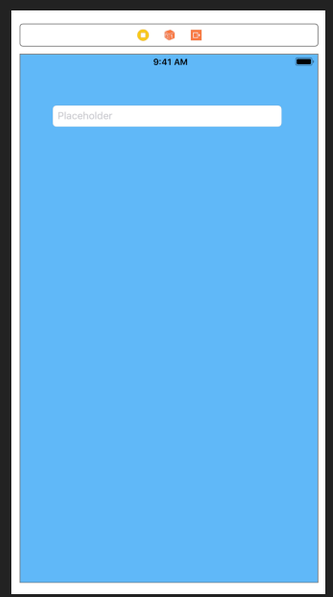
REVIEW

REVIEW - RECEBENDO INPUT DO USUÁRIO

- Cria outlet do Text Field
- Utilizar a property .text

```
@IBOutlet weak var textoUsuario: UITextField!
```

```
@IBAction func escreverTextoDigitado(_ sender: Any) {
    print(textoUsuario.text ?? "nil")
}
```



REVIEW – PASSANDO INFORMAÇÕES ENTRE TELAS

- Para a próxima tela (não conectado por Segue - Programático)
- Para a próxima tela (com Segue - Storyboard)

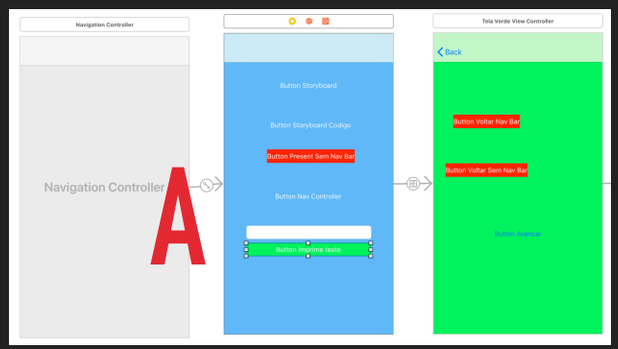
MAIS UTILIZADOS

- Para a tela anterior (Segue Unwind)

POUQUISSIMO UTILIZADO

- Utilizando o Padrão de Projeto Delegate

IMPORTANTÍSSIMO DOMINAR



A

B

PARA PRÓXIMA TELA

A -> B

REVIEW – PASSANDO PARA PRÓXIMA TELA COM/SEM SEGUE

TELA DESTINO (B)

- Criamos a property que vamos receber o valor
- Essa property pode ser de qualquer tipo (Int, String, Enum, UIImage...)

```
class TelaVerdeViewController: UIViewController {
    var texto: String?
```

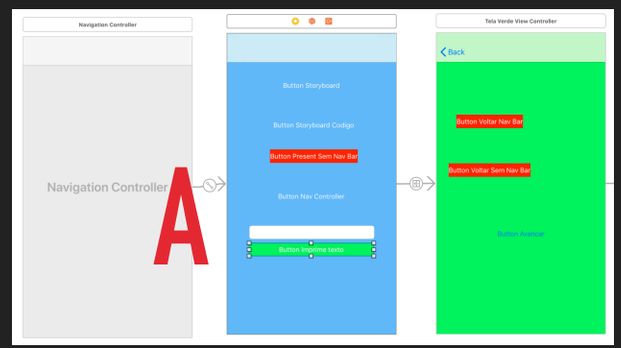
Sem interrogação, ele não pode ser nulo então espera um valor

REVIEW – PASSANDO PARA PRÓXIMA TELA COM/SEM SEGUE

TELA ORIGEM (A)

- Definimos o tipo da ViewController (TelaVerdeViewController)
- Preenchemos a property da variável (telaVerdeViewController.texto minúsculo pois é a variável)

```
@IBAction func buttonNavControllerPressed(_ sender: Any) {
    let storyboardMain = UIStoryboard(name: "Main", bundle: nil)
    if let telaVerdeViewController = storyboardMain.instantiateViewController(withIdentifier:
        "telaVerdeStoryboardID") as? TelaVerdeViewController {
        telaVerdeViewController.texto = "oi"
        self.navigationController?.pushViewController(telaVerdeViewController, animated: true)
    }
}
```



A

B

PARA TELA ANTERIOR

B -> A

REVIEW – PARA TELA ANTERIOR DELEGATE

TELA ORIGEM (B)

- Criamos um protocolo
- Criamos uma variável delegate (é uma referência para a classe que implementa esse protocolo)
- Chamamos o método da variável delegate que temos, então ele irá executar na tela anterior

TELA DESTINO (A)

- Informamos que nossa classe implementa o protocolo da outra classe
- Implementamos os métodos do protocolo

Protocol é como uma interface do Java

REVIEW - PARA TELA ANTERIOR DELEGATE

TELA ORIGEM (B)

```
11  protocol TelaVerdeViewControllerDelegate: AnyObject {
12      func voltarPassandoArgumentos(_ texto:String)
13  }
14
15  class TelaVerdeViewController: UIViewController {
16      weak var delegate: TelaVerdeViewControllerDelegate?
```

REVIEW – PARA TELA ANTERIOR DELEGATE

TELA ORIGEM (B)

- No momento que queremos podemos chamar o método do delegate.

```
    @IBAction func buttonVoltarPressed(_ sender: Any) {
        delegate?.voltarPassandoArgumentos("texto de retorno")
//        quando se utiliza a nav bar, para voltar a tela, usa o pop
        self.navigationController?.popViewController(animated: true)
    }
```

REVIEW - PARA TELA ANTERIOR DELEGATE

TELA DESTINO (A)

```
if let telaVerdeViewController = storyboardMain.instantiateViewController(withIdentifier:
    "telaVerdeStoryboardID") as? TelaVerdeViewController {
    telaVerdeViewController.texto = textoUsuario.text

    // informa a tela seguinte que essa tela executara o metodo voltarPassandoArgumentos
    telaVerdeViewController.delegate = self
    self.navigationController?.pushViewController(telaVerdeViewController, animated: true)
}
```

```
11  class TelaInicialViewController: UIViewController, TelaVerdeViewControllerDelegate {
12      func voltarPassandoArgumentos(_ texto: String) {
13          print(texto)
14      }
```


GERENCIAMENTO DE MEMÓRIA
RETAIN CYCLE

CONTAGEM AUTOMÁTICA DE REFERENCIAS

- Tipos de referências (objetos de classes) são armazenados na memória heap;
- Como o iOS sabe a hora de recuperar a memória consumida por estes objetos?
- O sistema faz uma contagem de referências para cada um dos objetos armazenados no heap. Quando um determinado objeto passa a ter zero referências, ele é eliminado da memória;
- Isto acontece de forma automática e é conhecido pelo nome de ARC (Automatic Reference Counting).

RETAIN CYCLE

Retain Cycle é quando a contagem do ARC se perde por algo que criamos

- Isso ocorre quando temos um objeto apontando para outro e vice-versa formando um ciclo
- O sistema contabiliza que sempre aqueles objetos vão ter uma referência.
- Ainda que coloquemos os dois como nil (nulo), eles estarão na memória e não conseguiremos mais aquele espaço de memória.

INFLUENCIANDO O ARC

- **strong**
  É a forma padrão de contagem de referencias. Enquanto alguém, em algum lugar mantiver um ponteiro **strong** para uma determinada instância, ela permanecerá na memória heap.

- **weak**
  Significa algo como: "se ninguém tiver interessado nisso, eu também não estou", ou seja, o ponteiro **weak** poderá ter o valor setado para mil quando não houver mais outros ponteiros **strong.** Em decorrência disso, **weak** só pode ser usado junto com tipos opcionais. O melhor exemplo disto são os outlets, cujos ponteiros **strong** estão na hierarquia de views, então os outlets no ViewController podem ser **weak.**

- **unowned**
  Perigoso! Significa: "não conte esta referência". Irá provocar crash do aplicativo caso usado de forma incorreta. Por isso, o uso dele é extremamente limitado, geralmente usado apenas com a intenção de quebrar referências cíclicas entre objetos.

GERENCIAMENTO DE MEMÓRIA

A GENTE JÁ VIU ALGO ASSIM?

```
11  protocol TelaVerdeViewControllerDelegate: AnyObject {
12      func voltarPassandoArgumentos(_ texto:String)
13  }
14
15  class TelaVerdeViewController: UIViewController {
16      weak var delegate: TelaVerdeViewControllerDelegate?
```

Retain cycle

VAMOS ENTENDER A RAZÃO DO WEAK

Um exemplo de Retain cycle.
Criar um novo arquivo playground ou alterar o do blackboard

VAMOS CRIAR DUAS CLASSES

```
class Pessoa {
    let nome: String
    var casa: Apartamento?
}
```

```
class Apartamento {
    let numero: Int
    var dono: Pessoa?
}
```

- Let indica uma constante, então deveria ter um valor
- Vamos criar um init (construtor) para receber esse valor.
- Init é o primeiro método a ser chamado em uma classe.

```
class Pessoa {
    let nome: String
    var casa: Apartamento?

    init(nome: String) {
        self.nome = nome
    }
}
```

```
class Apartamento {
    let numero: Int
    var dono: Pessoa?

    init(numero: Int) {
        self.numero = numero
    }
}
```

- Vamos criar variáveis que criem esses objetos
- E imprimir os valores

```
var rene:Pessoa? = Pessoa(nome: "rene")
var casa:Apartamento? = Apartamento(numero: 106)
print(rene?.nome)
print(rene?.casa)
```

- O código está assim até agora:
- E está indicando que estamos sem casa e a casa está sem dono:

```
Optional("rene")
nil
Optional(106)
nil
```

```swift
class Pessoa {
    let nome: String
    var casa: Apartamento?

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

class Apartamento {
    let numero: Int
    var dono: Pessoa?

    init(numero: Int) {
        self.numero = numero
    }
}

var rene:Pessoa? = Pessoa(nome: "rene")
var casa:Apartamento? = Apartamento(numero: 106)

print(rene?.nome)
print(rene?.casa)

print(casa?.numero)
print(casa?.dono)
```

GERENCIAMENTO DE MEMÓRIA

- Utilizando o método deinit sabemos quando o objeto é retirado da memória

```swift
class Pessoa {
    let nome: String
    var casa: Apartamento?

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

    deinit {
        print("\(nome) retirado da memoria")
    }
}
```

```swift
class Apartamento {
    let numero: Int
    var dono: Pessoa?

    init(numero: Int) {
        self.numero = numero
    }

    deinit {
        print("Ap \(numero) retirado da memoria")
    }
}
```

- Após os prints, vamos definir esses objetos como nil (nulos)

```
casa = nil
rene = nil
```

- Vemos que o deinit foi chamado

```
Ap 106 retirado da memoria
rene retirado da memoria
```

AINDA NÃO TEMOS UM RETAIN CYCLE

- Logo após o fim dos prints, mas antes de setar como nulo, vamos nos colocar como donos da casa:

```
print(casa?.dono)
casa?.dono = rene
casa = nil
```

- Vemos que ainda está tudo ok

```
Ap 106 retirado da memoria
rene retirado da memoria
```

AINDA NÃO TEMOS UM RETAIN CYCLE

- Para ser um ciclo, um objeto deve estar apontando para o outro
- Nesse momento só temos uma mão
- Vamos colocar a nossa casa, como essa:

```
print(casa?.dono)
casa?.dono = rene
rene?.casa = casa
casa = nil
```

AGORA TEMOS UM RETAIN CYCLE

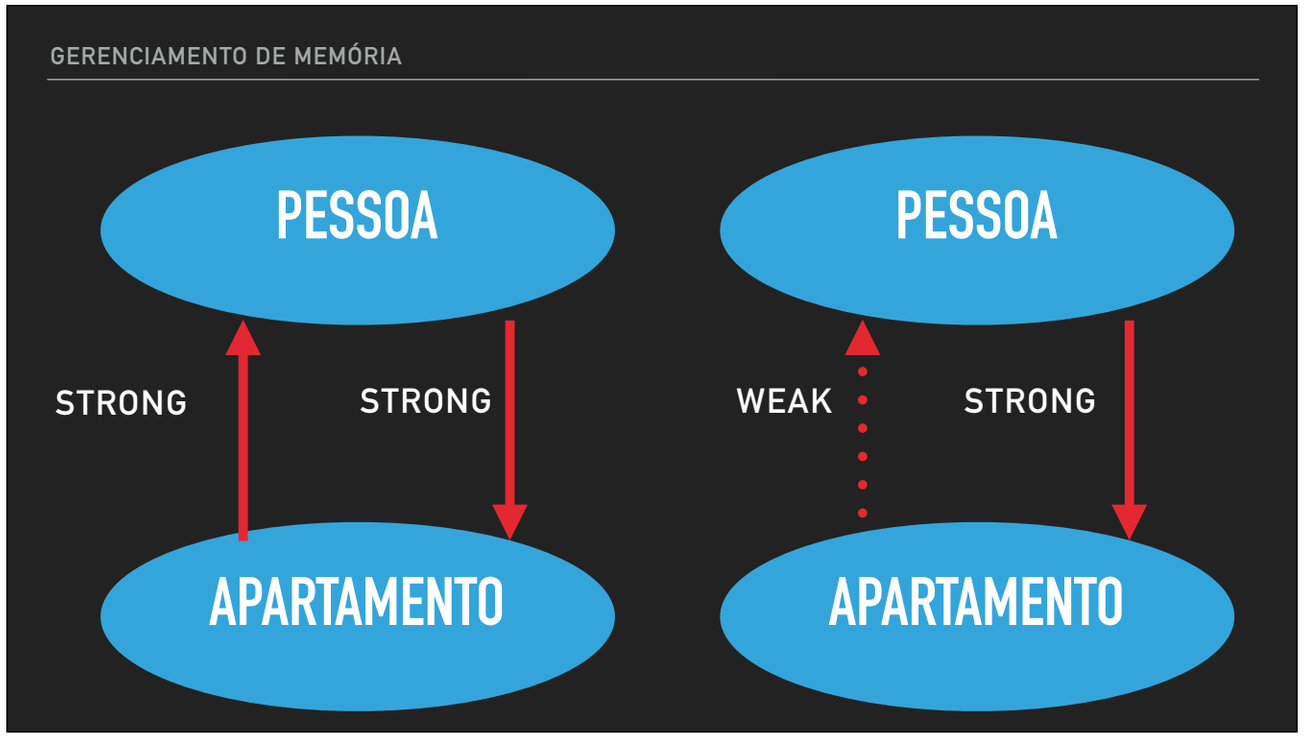
- Um objeto esta apontando para o outro e vice versa
- O console não exibe mais o deinit, ele está como antes:

```
Optional("rene")
nil
Optional(106)
nil
```

```
var rene:Pessoa? = Pessoa(nome: "rene")
var casa:Apartamento? = Apartamento(numero: 106)

print(rene?.nome)
print(rene?.casa)

print(casa?.numero)
print(casa?.dono)
casa?.dono = rene
rene?.casa = casa
casa = nil
rene = nil
```

COMO CORRIGIR?

WEAK


GERENCIAMENTO DE MEMÓRIA
PESSOA
PESSOA
STRONG
STRONG
WEAK
STRONG
APARTAMENTO
APARTAMENTO

- Ao colocar o weak no dono ou na casa (referências que eram strong para outra classe)

```
weak var dono: Pessoa?
```

- Vemos que o deinit voltou a ser chamado

```
Ap 106 retirado da memoria
rene retirado da memoria
```

PASSANDO DADOS

CLOSURES

CLOSURES

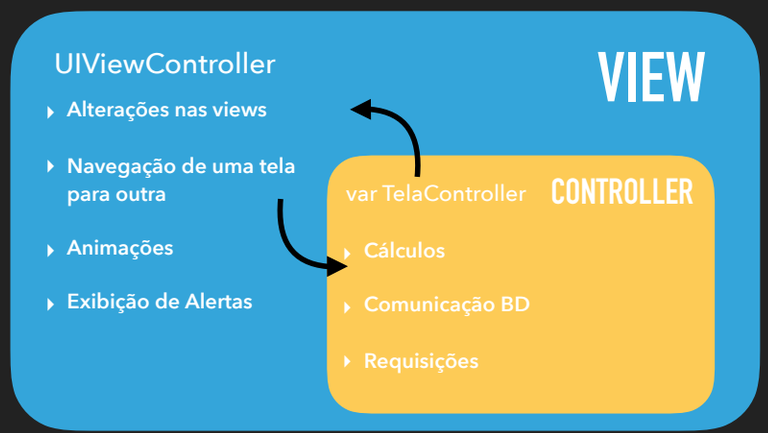
- São pequenos trechos de código a serem executados
- É como uma função em linha (na mesma linha)
- Caracterizados por { }
- Para acessar Outlets e variáveis, use o self.nomeDaVariável

LEMBRAM DO COMPLETION?

```
//nao utiliza a Nav Bar, por isso nao exibe o botao voltar
present(telaVerdeViewController, animated: true, completion: { () in
    print("Trocou de tela 😄")
})
```

CLOSURES

- Boa forma de conversar entre as camadas do MVC
- Quando necessário a camada mais abaixo chama a execução dessa closure
- Então, é uma boa para requisições Http (veremos mais a frente)

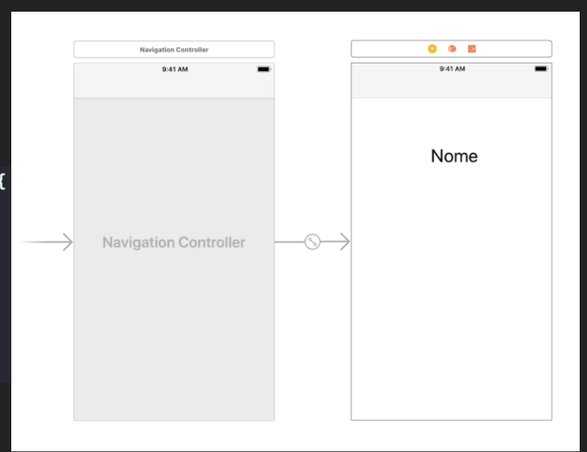


CLOSURES

CAMADA VIEW

- Para esse exemplo, vamos criar uma nova ViewController
- Criar uma classe para ela - ClosureExemploViewController
- Colocar uma Label
- Criar a Outlet dessa Label

```
11  class ClosureExemploViewController: UIViewController {
12
13      @IBOutlet weak var nomeUsuario: UILabel!
14
15      override func viewDidLoad() {
16          super.viewDidLoad()
17      }
18  }
```



CLOSURES

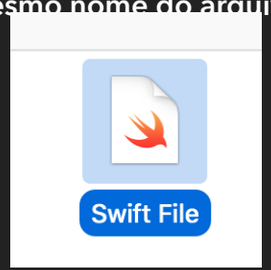
CAMADA CONTROLLER

- Vamos criar outra classe, porém será

  um Swift File - ClosureExemploController.swift

- (Não há diferença entre um CocoaTouch Class e o Swift File, se você copiar o conteúdo de uma CocoaTouch Class, ele será reconhecido no Storyboard. O Storyboard reconhece arquivos .swift que extendem a classe específica

- Vamos criar uma classe de mesmo nome do arquivo, que será nossa Controller



CLOSURES

CAMADA CONTROLLER

- ClosureExemploController - Nossa classe controller
- recuperarNomeDoUsuario - Nossa função que irá recuperar o nome do usuario
- _ handlerNomeRecuperado - parâmetro recebido pela função, o _ (underscore) indica que ao chamar a função não temos que escrever handlerNomeRecuperado
- @escaping (String?) -> () Tipo do parâmetro a ser enviado. Uma função que recebe uma string e não retorna nada

```
import Foundation

class ClosureExemploController: NSObject {
    func recuperarNomeDoUsuario(_ handlerNomeRecuperado: @escaping (String?) -> ()) {
        sleep(10)
        handlerNomeRecuperado("Rene")
    }
}
```

CLOSURES

CAMADA CONTROLLER

- E o @escaping ?
- Mantém essa closure na memória até o fim da execução
- Ao executar algo em uma thread de background, a variável que contém a closure (handlerNomeRecuperado) seria perdido. O @escaping faz com que não seja perdido.
- Agora não fará diferença, pois não estamos usando uma thread de background, tanto que por 10 segundos qualquer ação na tela será bloqueada. Veremos isso na aula de Network

```
import Foundation

class ClosureExemploController: NSObject {
    func recuperarNomeDoUsuario(_ handlerNomeRecuperado: @escaping (String?) -> ()) {
        sleep(10)
        handlerNomeRecuperado("Rene")
    }
}
```

CLOSURES

CAMADA VIEW

Voltando para a Camada View:

- Criar uma variável com nossa controller
- Chamar a função que criamos

```
11  class ClosureExemploViewController: UIViewController {
12
13      @IBOutlet weak var nomeUsuario: UILabel!
14
15      let controller = ClosureExemploController()
16
17      override func viewDidLoad() {
18          super.viewDidLoad()
19
20          controller.recuperarNomeDoUsuario { (nome) in
21              self.nomeUsuario.text = nome ?? ""
22          }
23      }
24  }
```

CLOSURES

CAMADA VIEW

- controller.recuperarNomeDoUsuario - Estamos falando para ir na

Instância da nossa controller executar nossa função

- { (nome) in é a closure que estamos enviando. O parâmetro nome é do tipo String?, pois é como foi declarado na Camada Controller.
- Dentro das { } temos o que será executado ao fim da recuperação do nome do usuário, geralmente a tela é atualizada com os dados recebidos do servidor.

```
controller.recuperarNomeDoUsuario { (nome) in
    self.nomeUsuario.text = nome ?? ""
}
```

LINK DO ❤️

http://fuckingclosuresyntax.com/

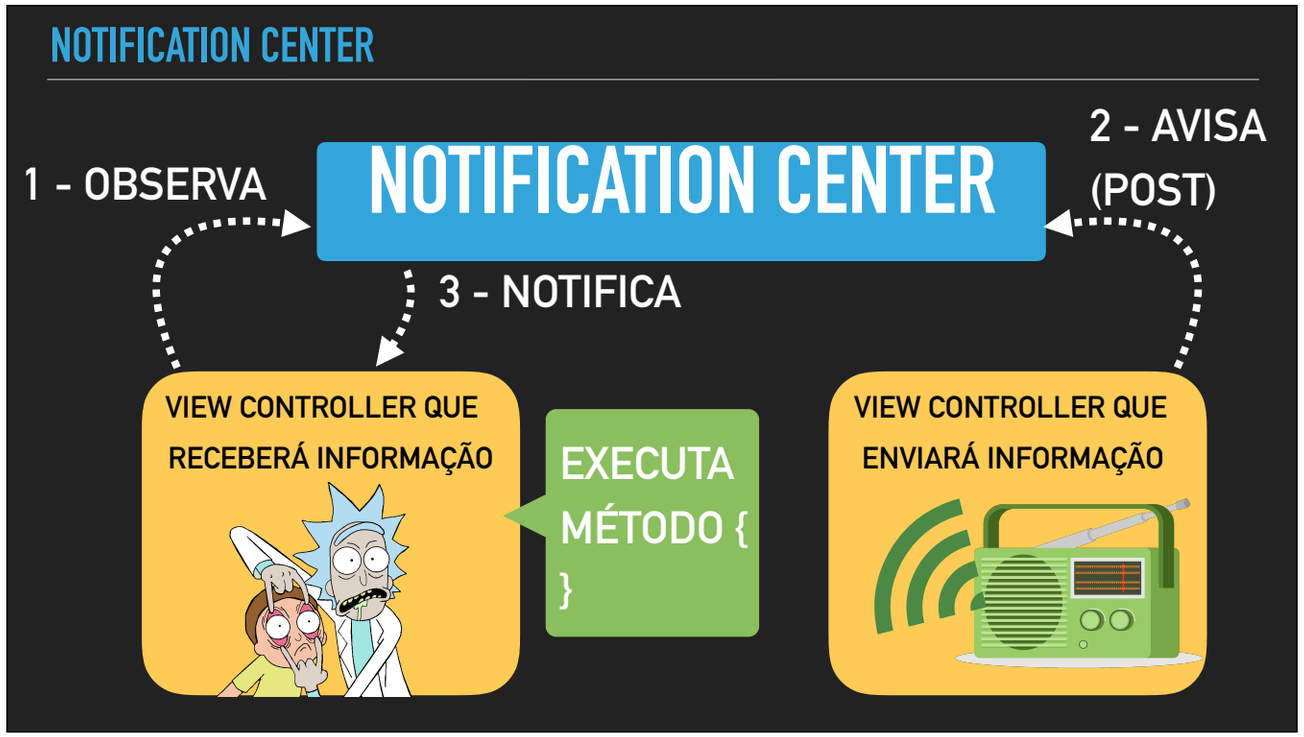
PASSANDO DADOS

NOTIFICATION CENTER

NOTIFICATION CENTER

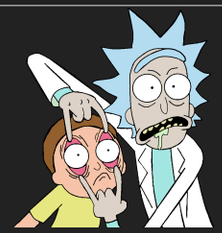
- Design Pattern - Observer - Observable
- Comunica sem ter uma relação direta - Não necessita de uma variável, protocolo, método, nem delegate
- É usado somente em situações específicas para não dificultar o desenvolvimento por outros devs
- Usado quando se quer ter algo com o valor mais atual e sempre está atualizando em um tempo indeterminado:
  - WhatsApp
  - Ações

NÃO É UMA NOTIFICAÇÃO PUSH


NOTIFICATION CENTER
1 - OBSERVA
NOTIFICATION CENTER
2 - AVISA
(POST)
3 - NOTIFICA
VIEW CONTROLLER QUE
RECEBERÁ INFORMAÇÃO
EXECUTA
MÉTODO {
}
VIEW CONTROLLER QUE
ENVIARÁ INFORMAÇÃO

NOTIFICATION CENTER

QUEM OBSERVA



- Notificação é definida por um nome - Notification.Name
- Coloca Observer e indica qual método vai retornar

1

```
let notificacao = Notification.Name("notificacaoDeTelaAberta")
NotificationCenter.default.addObserver(self, selector:
    #selector(escreverNotificacaoDeTelaAberta(notification:)), name: notificacao, object: nil)
```

- Cria o método que vai receber a notificação

```
@objc func escreverNotificacaoDeTelaAberta(notification:Notification) {
    print(notification.userInfo ?? "sem userInfo")
}
```

3

NOTIFICATION CENTER

QUEM ENVIA A INFORMAÇÃO

- Notificação é definida por um nome - Notification.Name
- Vai avisar que certa situação ocorreu - POST
- Dados enviados como um dicionário

2

```
let notificationName = Notification.Name("notificacaoDeTelaAberta")
NotificationCenter.default.post(name: notificationName, object: nil, userInfo:["mensagem": "Dado
    enviado", "abriuTela": true])
```

NOTIFICATION CENTER

QUEM OBSERVA

- Depois que não é mais necessário devemos retirar o Observer

```
let notificacao = Notification.Name("notificacaoDeTelaAberta")
NotificationCenter.default.removeObserver(self, name: notificacao, object: nil)
```

```
override func viewDidDisappear(_ animated: Bool) {
    let notificacao = Notification.Name("notificacaoDeTelaAberta")
    NotificationCenter.default.removeObserver(self, name: notificacao, object: nil)
}
```

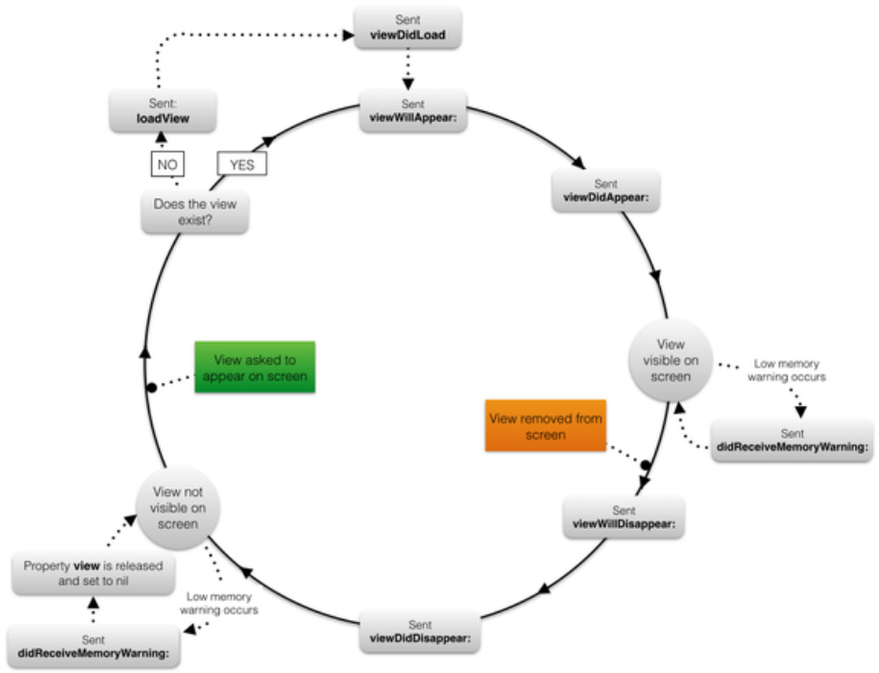

UIVIEWCONTROLLER
CICLO DE VIDA

CICLO DE VIDA DO UIVIEWCONTROLLER

- Os ViewControllers possuem um ciclo de vida, ou seja, uma sequencia de mensagens é enviada para o objeto conforme ele progride através do tempo de vida que ele tem no nosso software;
- Por que isso é importante!?
  É muito comum que sobrescrevamos estes métodos para fazer certas tarefas.
- Onde ele começa?
  Na criação do objeto. Os MVCs são instanciados, na maioria dos casos, através do Storyboard. Existem meios de fazer isso puramente através do código, mas falaremos disso no futuro.
- E o que acontece depois?
  Preparação para segue, se for o caso
  Preenchimento dos outlets
  Aparecer e desaparecer
  Mudanças de geometria
  Situações de escassez de recursos (falta de memória)

VIEW CONTROLLER

CICLO DE VIDA



CICLO DE VIDA DO VIEW CONTROLLER

DEPOIS DA INSTANCIAÇÃO E DO PREENCHIMENTO DOS OUTLETS...

- O método **viewDidLoad** é chamado. Este é um lugar excepcional para colocar código de incialização. É melhor do que um **init** porque aqui todos os seus outlets estão setados.
- Executado uma vez
- **Sempre** que você sobrescrever um método do ciclo de vida, dê uma chance à superclasse:

```
override func viewDidLoad() {
    super.viewDidLoad()

}
```

Outlets iniciados

DEPOIS DA INSTANCIAÇÃO E DO PREENCHIMENTO DOS OUTLETS...

```
override func viewDidLoad() {
    super.viewDidLoad()

}
```

- Uma boa coisa para fazer aqui é sincronizar sua View com sua Model, porque você pode ter a certeza de que seus outlets estão setados.
- **Tome cuidado!** Aqui as geometrias das Views ainda não estão preenchidas! Neste ponto não se pode ter certeza de que a tela é de um iPhone ou de um iPad, por exemplo.
- Então, não faça nada que dependa da geometria das views aqui.

CICLO DE VIDA DO VIEW CONTROLLER

IMEDIATAMENTE ANTES DA VIEW APARECER NA TELA, VOCÊ É AVISADO

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)
}
```

- Sua view é carregada apenas uma vez (**viewDidLoad**), mas a view pode aparecer e desaparecer várias vezes
- Então, tenha sempre o cuidado para não colocar aqui o código que deveria estar no **viewDidLoad**, sob a pena de ficar executando um trecho de código diversas vezes desnecessariamente.
- Aqui deve constar código que controla mudanças do MVC enquanto a View não está na tela
- As geometrias da sua View estão preenchidas aqui, então esse é o lugar

IMEDIATAMENTE APÓS A VIEW APARECER NA TELA

```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)
}
```

- Carregar animação que será exibida ao usuário
- Carregar dados externos (API)

VOCÊ TAMBÉM VAI SER NOTIFICADO QUANDO SUA VIEW FOR DESAPARECER

```
override func viewWillDisappear(_ animated: Bool) {
    super.viewWillDisappear(animated)
    //sempre chame super nos métodos will/did

    /* Aqui, faça uma eventual limpeza, tomando
    o cuidado para não fazer nada demasiadamente
    pesado, ou o aplicativo pode apresentar "lag"

     Você pode até mesmo iniciar uma thread aqui.
     Falaremos de threads depois.
    */
}
```

- Logo antes de outra ViewController tomar a tela

E QUANDO ELA JÁ DESAPARECEU

```
override func viewDidDisappear(_ animated: Bool) {
    let notificacao = Notification.Name("notificacaoDeTelaAberta")
    NotificationCenter.default.removeObserver(self, name: notificacao, object: nil)
}
```

- Como vimos, muito usado para remover Observers

MUDANÇAS DE GEOMETRIA

- A vasta maioria do trabalho relacionado é feito com o Autolayout
- Mas se você quiser, pode se envolver diretamente nas mudanças de geometria através dos métodos **viewWillLayoutSubviews** e **viewDidLayoutSubviews**;
- Eles são evocados quando o frame da view raiz muda e provoca o re-layout das subviews. Isto ocorre, por exemplo, quando rotacionamos o dispositivo
- Entre o **will** e o **did**, o autolayout acontece
- Estes métodos são chamados com muita freqüência (estados antes e depois de uma animação, por exemplo e etc), então, não coloque aqui código que não seja eficiente para ser executado repetidamente

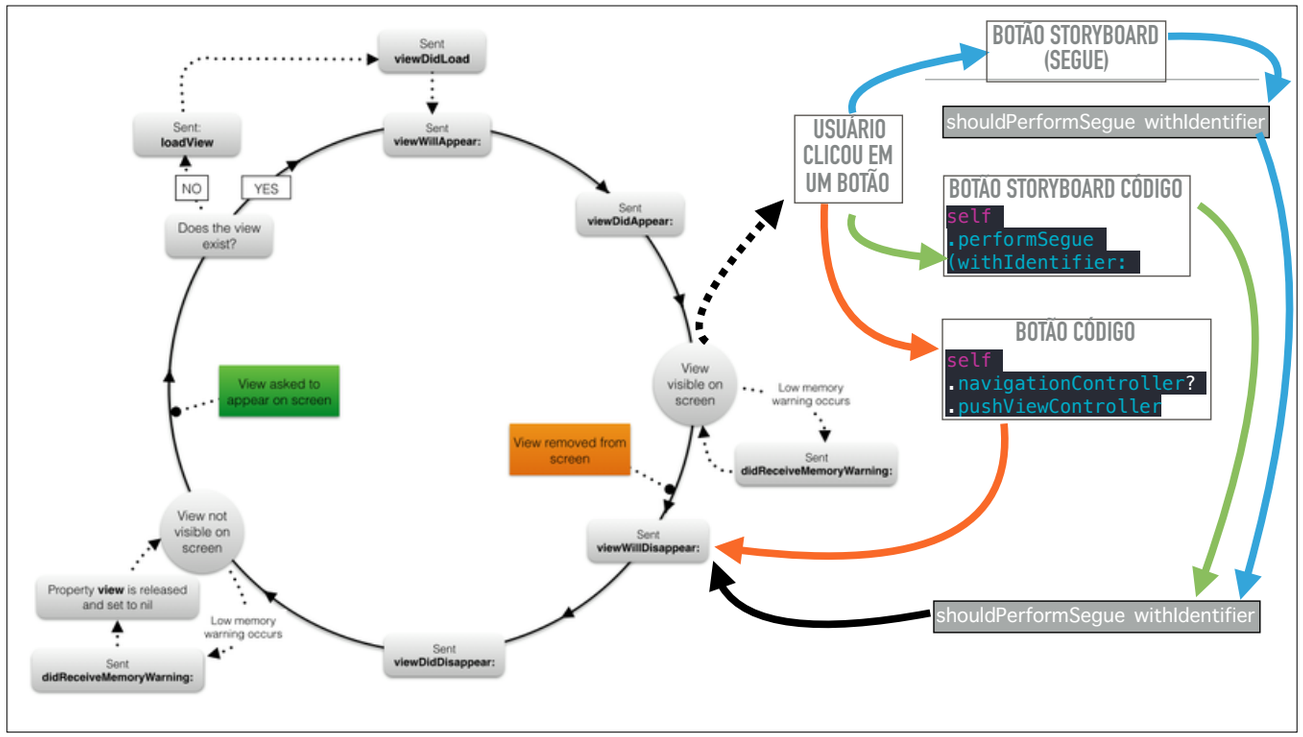
ROTAÇÃO DO DISPOSITIVO

- Normalmente, as mudanças de formato acontecem quando da rotação do dispositivo;
- É possível controlar quais orientações o aplicativo suporta através das configurações do projeto no Xcode;
- Caso você queira “participar" da animação de rotação, você pode usar este método:

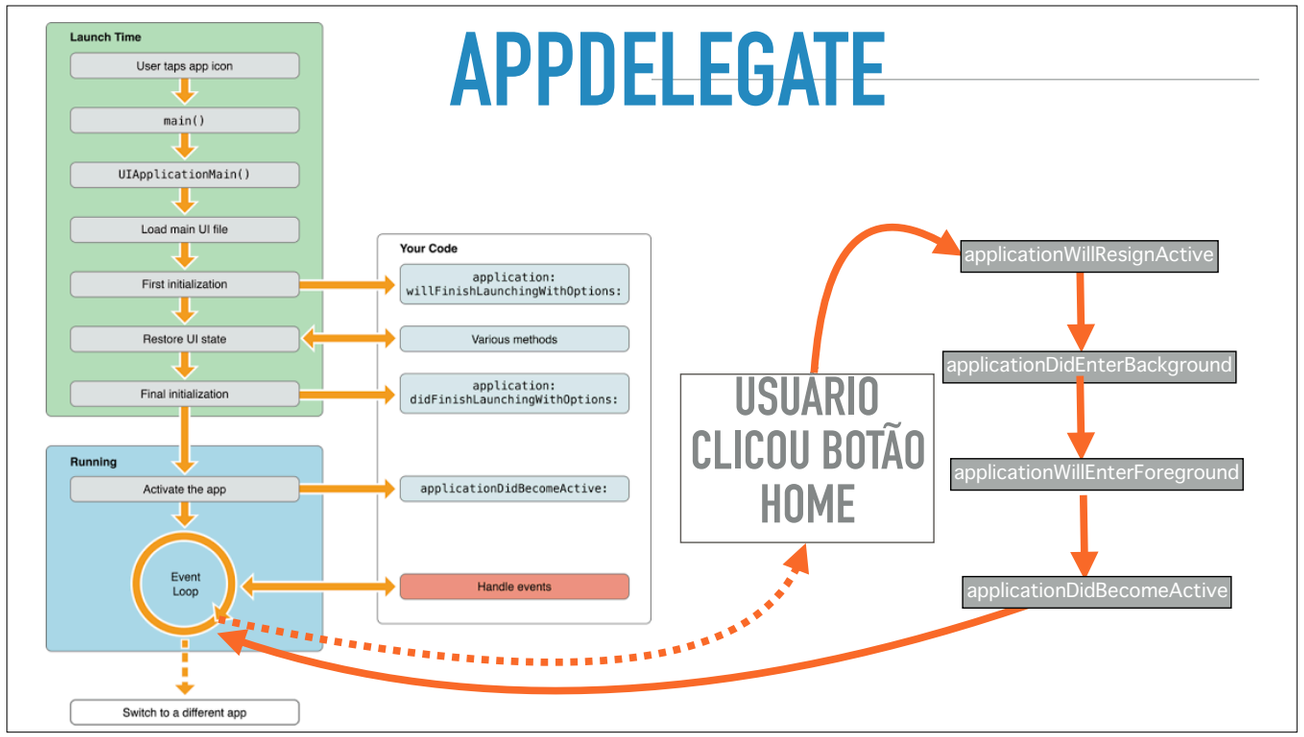
```
override func viewWillTransition(to size: CGSize,
                                 with coordinator: UIViewControllerTransitionCoordinator) {

}
```

BAIXA DE MEMÓRIA

- Em situações de baixa de memória, o método **didReceiveMemoryWarning** será chamado;
- Raramente isso acontece, mas aplicações que usam recursos que consomem grandes quantidades de memória devem se antecipar e implementar este método. Um exemplo é uma aplicação que faz uso de imagens e sons;
- Na implementação, quaisquer recursos (aqueles que consomem muita memória) que podem ser facilmente recriados devem ser descartados. Isso se dá simplesmente setando o valor **nil.**


Sent viewDidLoad
Sent: loadView
Sent viewWillAppear:
NO
YES
Does the view exist?
Sent viewDidAppear:
View asked to appear on screen
View visible on screen
Low memory warning occurs
Sent didReceiveMemoryWarning:
View removed from screen
View not visible on screen
Sent viewWillDisappear:
Property view is released and set to nil
Low memory warning occurs
Sent didReceiveMemoryWarning:
Sent viewDidDisappear:
USUÁRIO CLICOU EM UM BOTÃO
BOTÃO STORYBOARD (SEGUE)
shouldPerformSegue withIdentifier
BOTÃO STORYBOARD CÓDIGO
self .performSegue (withIdentifier:
BOTÃO CÓDIGO
self .navigationController? .pushViewController
shouldPerformSegue withIdentifier

APPDELEGATE



LINK DO ❤️

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

VAMOS BISBILHOTAR O CICLO
DE VIDA DO VIEW CONTROLLER

HORA DA PRÁTICA