



PROF. PEDRO HENRIQUE

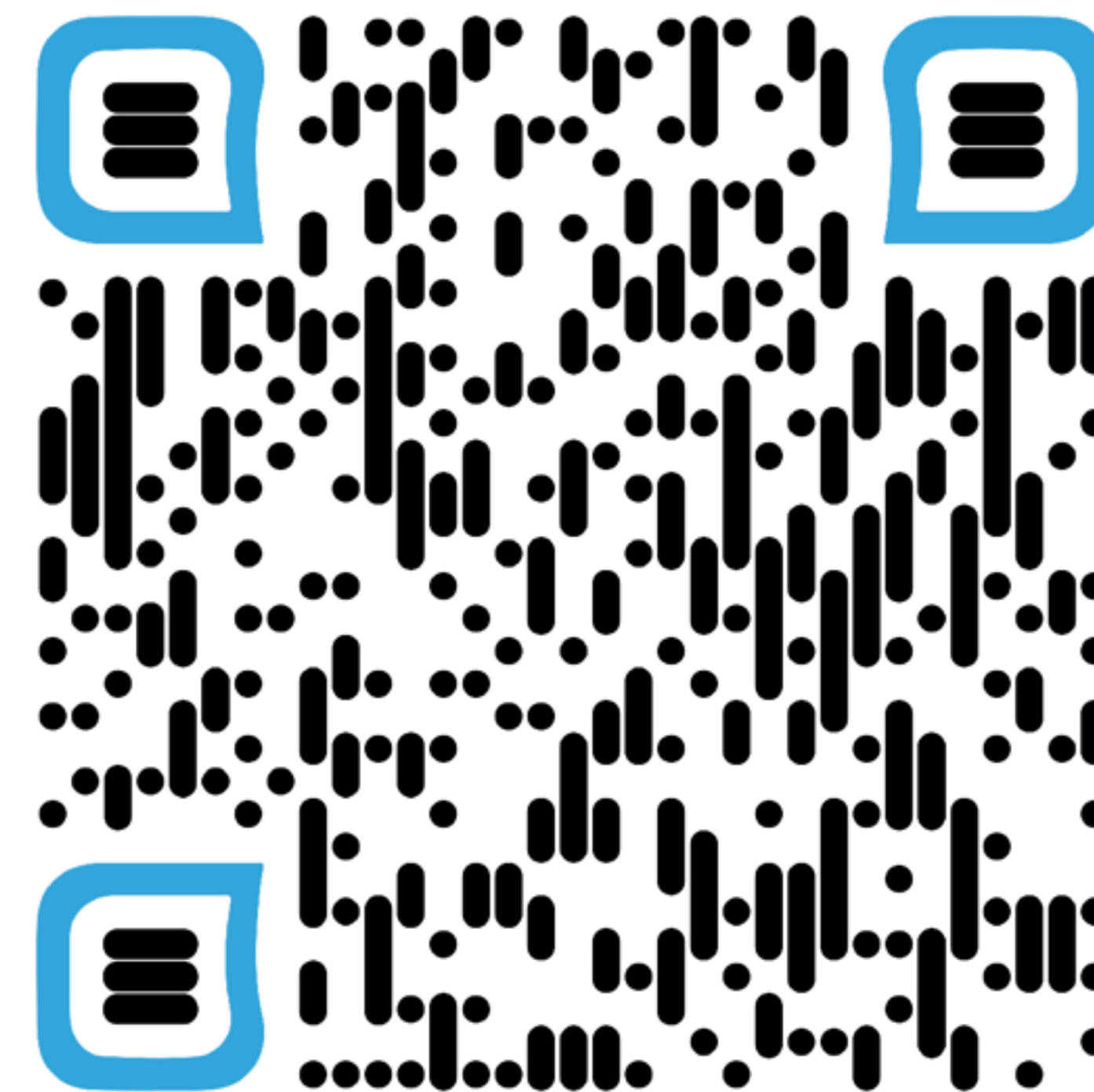
DESENVOLVIMENTO PARA IOS 11 COM SWIFT 4

ONDE ENCONTRAR O MATERIAL?

LEIA O QR CODE

ALÉM DO TRADICIONAL BLACKBOARD DO IESB

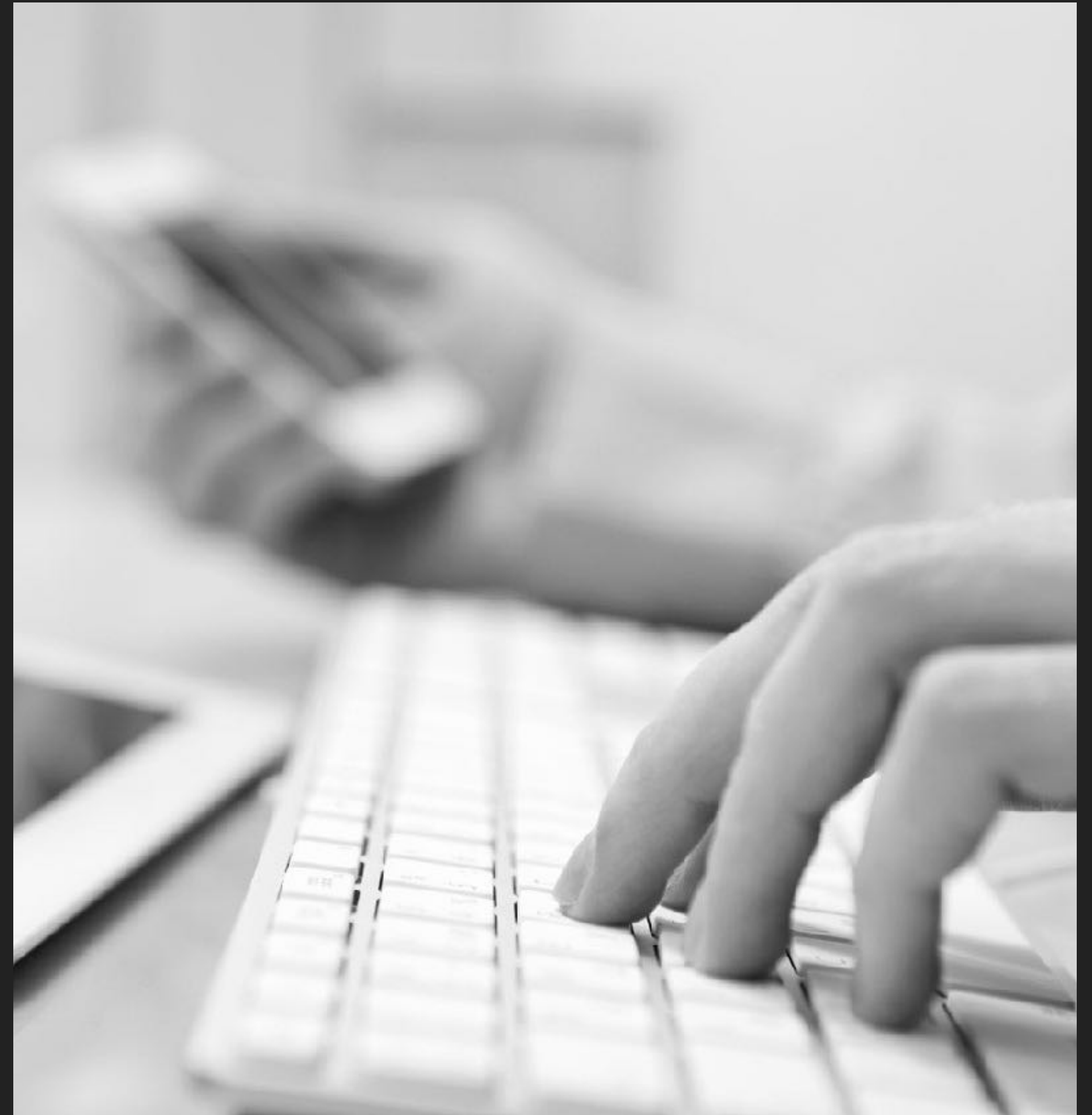
GITHUB DA TURMA




[HTTPS://GIT.IO/VF50W](https://git.io/vf50w)

AGENDA

- ▶ Múltiplos MVCs com SplitViewController (adaptando para iPad e iPhone)
- ▶ Ciclo de vida do UIViewController
- ▶ Introdução ao gerenciamento de memória
 - ▶ Cuidados com as closures



A black and white photograph of several wind turbines against a cloudy sky. The largest turbine is in the foreground on the left, with its blades extending towards the top left. Two smaller turbines are visible in the background, one in the center and one further back on the right.

MÚLTIPLOS MVCS COM SPLIT VIEW CONTROLLER: TORNANDO O NOSSO APP ADAPTADO PARA IPAD E IPHONE

Múltiplos Dispositivos!!!

CICLO DE VIDA DO UVIEWCONTROLLER

- ▶ Os ViewControllers possuem um ciclo de vida, ou seja, uma sequência 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)

DEPOIS DA INSTANCIACÃO E DO PREENCHIMENTO DOS OUTLETS...

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

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

DEPOIS DA INSTANCIACÃO 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.

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 (o ciclo só passa uma vez pelo **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
- ▶ Pode ser usado também para otimizar a performance

IMEDIATAMENTE ANTES DA VIEW APARECER NA TELA, VOCÊ É AVISADO

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



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

VOCÊ TAMBÉM VAI SER NOTIFICADO QUANDO SUA VIEW FOR DESAPARECER

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

- Naturalmente, existe também a versão "did" deste método.

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 frequência (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**.

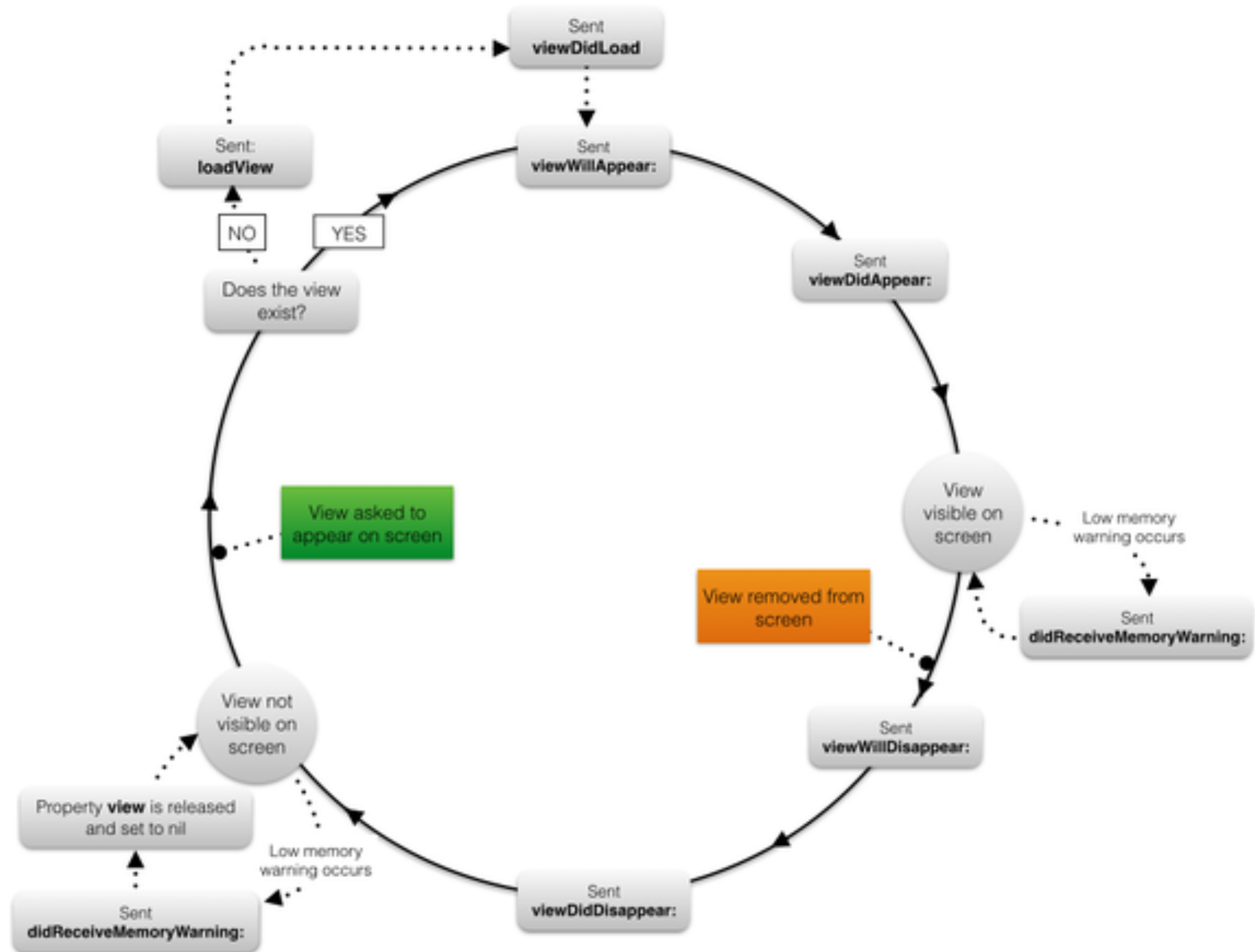
BOM DIA! HORA DE ACORDAR!!!

- ▶ O método **awakeFromNib** é chamado para todos os objetos que vem do storyboard;
- ▶ Esta chamada acontece antes do preenchimento dos outlets;
- ▶ Não é uma boa prática sobrescrever este método. Você deve colocar qualquer código em outro local (**viewDidLoad** ou **viewWillAppear**, por exemplo).

LINK DO 

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

CICLO DE VIDA





VAMOS BISBILHOTAR O CICLO
DE VIDA DO VIEW CONTROLLER

HORA DA PRÁTICA



AUTOMATIC REFERENCE COUNTING

GERENCIAMENTO DE MEMÓRIA

LINKS DO



<https://pt.stackoverflow.com/questions/3797/o-que-são-e-onde-estão-o-stack-e-heap>

<https://krakendev.io/blog/weak-and-unowned-references-in-swift>

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) e **não é a mesma coisa** que garbage collector.

INFLUENCIANDO O ARC

- ▶ Podemos influenciar a maneira como o ARC funciona usando algumas palavras reservadas junto com a declaração da variável de um tipo de referência:
- ▶ **strong**
- ▶ **weak**
- ▶ **unowned**

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 nil 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**.

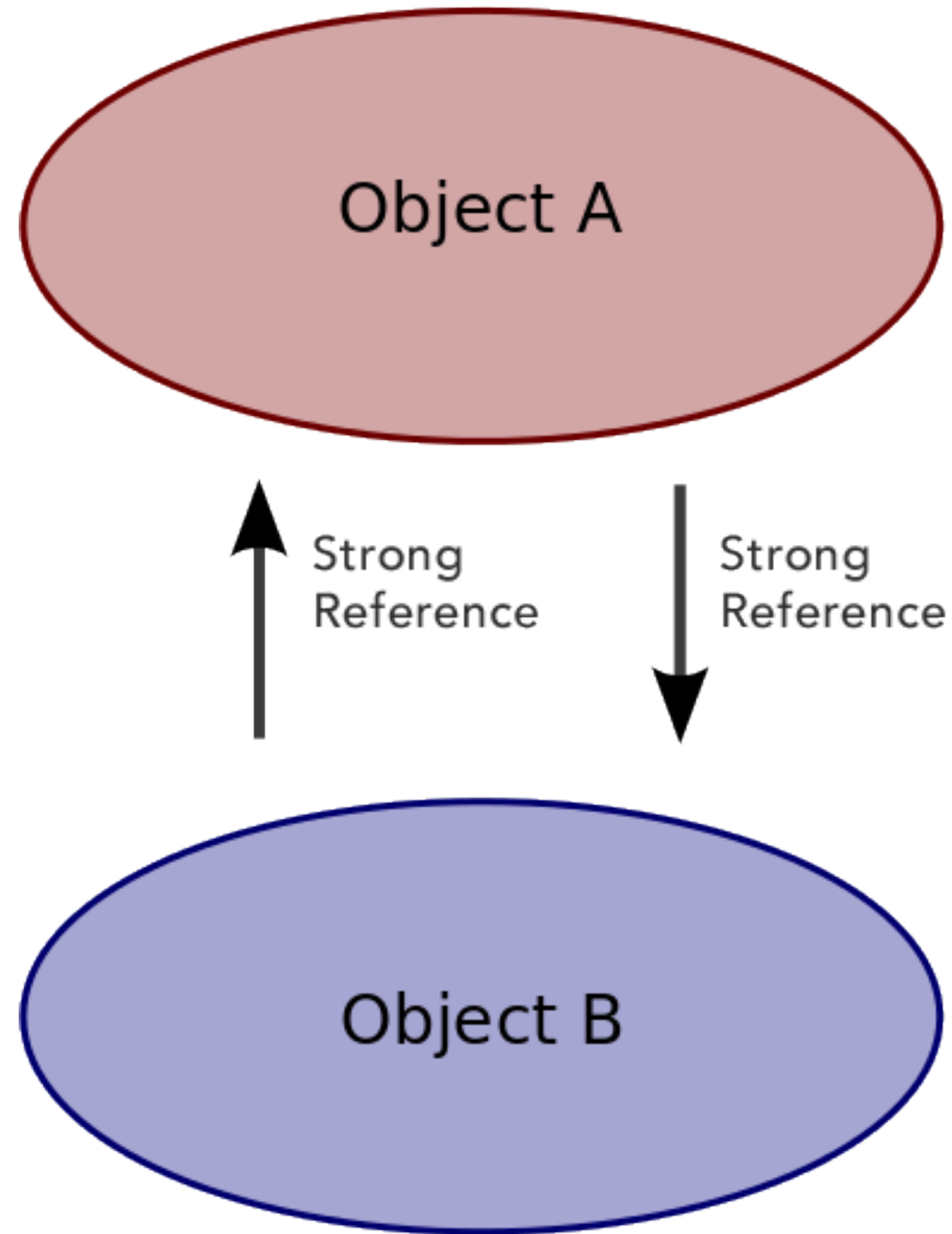
► **unowned**

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.

CLOSURES E A CAPTURA DE CONTEXTO

- ▶ As closures são armazenadas na memória heap também, elas são tipos-referência. Podem ser colocadas em arrays, dicionários e etc. São exatamente como classes no Swift;
- ▶ Além disso, elas “capturam” o contexto em que estão inseridas, ou seja, variáveis externas à closure que são usadas dentro dela, são capturadas e armazenadas no heap também.
- ▶ Estas variáveis capturadas precisam ficar no heap pelo tempo que a closure ficar e esta situação pode gerar o temido retain-cycle

RETAIN CYCLE



EXEMPLO PRÁTICO DE RETAIN CYCLE

- ▶ Na nossa calculadora, imagine que adicionamos uma API pública para permitir adicionar uma operação unária no CalculatorBrain

`func addUnaryOperation(symbol: String, operation: (Double) -> Double)`

Este método não faria nada além de adicionar a operação no nosso dicionário de enum

- ▶ Agora imagine um ViewController usando este método para adicionar uma operação de "raiz quadrada verde":

```
addUnaryOperation(symbol: "✓", operation: { (x: Double) -> Double in
    display.textColor = UIColor.green
    return sqrt(x)
})
```

EXEMPLO PRÁTICO DE RETAIN CYCLE

```
addUnaryOperation("✅") { (x: Double) -> Double in  
    display.textColor = UIColor.green  
    return sqrt(x)  
}
```

```
addUnaryOperation("✅") {  
    display.textColor = UIColor.green  
    return sqrt($0)  
}
```

EXEMPLO DE RETAIN CYCLE

```
addUnaryOperation("✅") {  
    self.display.textColor = UIColor.green  
    return sqrt($0)  
}
```

O Swift nos força a incluir a declaração self aqui para nos lembrar que self vai ser capturado! Agora, a Model e o Controller apontam um para o outro através desta closure! Nenhum dos dois objetos irá jamais deixar a memória! O nome disso é retain cycle (ou memory cycle ou ainda memory leak)

COMO QUEBRAMOS ESSE CICLO?

- ▶ O swift nos permite controlar o comportamento de captura...

```
addUnaryOperation("✅") { [/*declaração especial*/] in
    self.display.textColor = UIColor.green
    return sqrt($0)
}
```


COMO QUEBRAMOS ESSE CICLO?

- ▶ O swift nos permite controlar o comportamento de captura...

```
addUnaryOperation("✅") { [ me = self ] in  
    me.display.textColor = UIColor.green  
    return sqrt($0)  
}
```


COMO QUEBRAMOS ESSE CICLO?

- ▶ O swift nos permite controlar o comportamento de captura...

```
addUnaryOperation("✓") { [ unowned self = self ] in
    self.display.textColor = UIColor.green
    return sqrt($0)
}
```

COMO QUEBRAMOS ESSE CICLO?

- ▶ O swift nos permite controlar o comportamento de captura...

```
addUnaryOperation("✅") { [ unowned self ] in  
    self.display.textColor = UIColor.green  
    return sqrt($0)  
}
```

COMO QUEBRAMOS ESSE CICLO?

- ▶ O swift nos permite controlar o comportamento de captura...

```
addUnaryOperation("✅") { [ weak self ] in  
    self?.display.textColor = UIColor.green  
    return sqrt($0)  
}
```

COMO QUEBRAMOS ESSE CICLO?

- ▶ O swift nos permite controlar o comportamento de captura...

```
addUnaryOperation("✅") { [ weak weakSelf = self ] in  
    weakSelf?.display.textColor = UIColor.green  
    return sqrt($0)  
}
```

ESTA É A FORMA MAIS USADA PELA COMUNIDADE IOS.

**HORA DA PRÁTICA! VAMOS FAZER O QUE
ACABAMOS DE FALAR NA NOSSA
CALCULADORA**

Professor Pedro



PROJETO DA DISCIPLINA

**Descrição completa e prazo de e
entrega no Blackboard**