



PROF. PEDRO HENRIQUE

DESENVOLVIMENTO PARA IOS 10 COM SWIFT 3



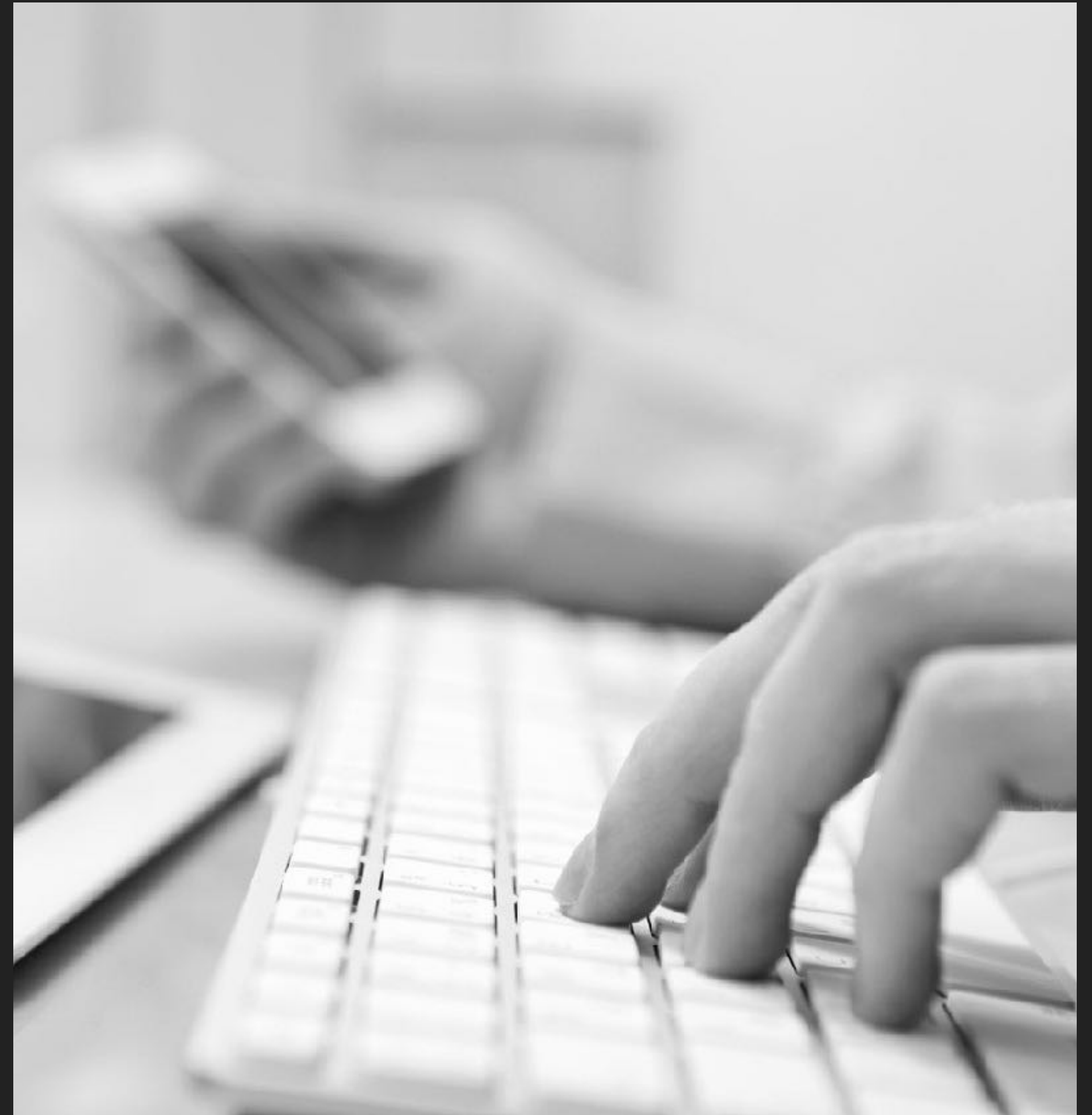
ONDE ENCONTRAR O MATERIAL?

LEIA O QR CODE

ALÉM DO TRADICIONAL BLACKBOARD DO IESB

AGENDA

- ▶ Mergulhando de cabeça no Swift;
- ▶ Detalhes sobre o tipo Optional;
- ▶ Tuplas;
- ▶ Range<T>;
- ▶ Estruturas de Dados, Métodos e Propriedades;
- ▶ Inicialização;
- ▶ O coringa AnyObject; Introspecção e typecast;
- ▶ UserDefaults;
- ▶ Uso de assertivas



OPTIONAL

- ▶ Conforme já falamos anteriormente, o Optional é apenas um enum;
- ▶ Em outras palavras...

```
enum Optional<T> {  
    case none  
    case some(T)  
}
```

T é um tipo genérico (tal qual acontece no Java).

O TIPO OPTIONAL É APENAS UM ENUM...

```
enum Optional<T> {  
    case none  
    case some(T)  
}
```

```
let x: String? = nil  
// ↑ é a mesma coisa que ↓  
let x = Optional<String>.none
```

```
let x: String? = "hello"  
// ↑ é a mesma coisa que ↓  
let x = Optional<String>.some("hello")
```

AGORA TUDO FAZ SENTIDO!

ENTÃO ISSO QUER DIZER QUE...

```
enum Optional<T> {  
    case none  
    case some(T)  
}
```

```
let x: String? = nil  
// ↑ é a mesma coisa que ↓  
let x = Optional<String>.none
```

```
let x: String? = "hello"  
// ↑ é a mesma coisa que ↓  
let x = Optional<String>.some("hello")
```

```
let y = x!  
// ↑ é a mesma coisa que ↓  
switch x {  
    case some(let value): y = value  
    case none: // lança uma exceção  
}
```

AGORA TUDO FAZ SENTIDO!

ENTÃO ISSO QUER DIZER QUE...

```
let x: String? = ...
if let y = x {
    // fazer alguma coisa com y
}
// ↑ é a mesma coisa que ↓
switch x {
    case .some(let y):
        //fazer alguma coisa com y
    case .none:
        break
}
```

```
enum Optional<T> {
    case none
    case some(T)
}
```

```
let x: String? = nil
// ↑ é a mesma coisa que ↓
let x = Optional<String>.none
```

```
let x: String? = "hello"
// ↑ é a mesma coisa que ↓
let x = Optional<String>.some("hello")
```

```
let y = x!
// ↑ é a mesma coisa que ↓
switch x {
    case some(let value): y = value
    case none: // lança uma exceção
}
```

AND AS WE WIND ON DOWN THE ROAD...

PODEMOS “ENCADEAR” OS OPTIONAL'S

- ▶ Por exemplo, **hashValue** é um **var** em String;
- ▶ Se quisermos obter o **hashValue** de uma variável do tipo **String**? ?
- ▶ E se essa variável for, na verdade, o **text** de um **UILabel**? ?

HOW EVERYTHING STILL TURNS TO GOLD

```
// Imagine que este é um @IBOutlet sem o desencapsulamento automático!
var display: UILabel?

//Sem o encadeamento
if let temp1 = display {
    if let temp2 = temp1.text {
        let x = temp2.hashValue
        //agora sim eu tenho o valor!
    }
}

//Usando o encadeamento, aquilo vira isso:
if let x = display?.text?.hashValue{
    //Aqui o x é um Int
}

let x = display?.text?.hashValue //Aqui o x é um Int?
```

CALMA AÍ PESSOAL!



AINDA NÃO ACABOU...

OPTIONAL??

- ▶ Existe ainda um operador para expressar valores padrão para quando o desencapsulamento de opcional falhar;
- ▶ Por exemplo... Queremos colocar uma **String** em um **UILabel**, mas se a variável estiver sem valor (**nil**), queremos colocar um " " (espaço em branco).
- ▶ Daí usamos o operador **??**

```
//Aquele @IBOutlet imaginário
let display: UILabel?;

let s: String? = nil // pode ser nula
if s != nil {
    display.text = s!
} else {
    display.text = " "
}

/*
    Isso pode ser muito mais facilmente
    expressado através do comando ??
    exemplificado abaixo:
*/

display.text = s ?? " "
```

TUPLAS

- ▶ O que é isso!?
- ▶ Uma forma de agrupar mais de um valor;
- ▶ Você pode usar uma dupla em qualquer lugar que você pode usar um tipo qualquer;
- ▶ Exemplo...


```
let x: (String, Int, Double) = ("hello", 5, 0.85)
// O tipo de x é TUPLA

// Isso serve para dar nome aos elementos da tupla
let (word, number, value) = x

print(word) //imprimie hello
print(number) // imprime 5
print(value) // imprime 0.85
```

```
/* Ou os elementos da tupla podem ser nomeados
na declaração, de forma parecida com o que
fazemos nos métodos. Esta é a forma que é
considerada uma boa prática e será a nossa
preferida para as aulas e para as vossas
entregas.
```

```
*/
```

```
let x: (w: String, i: Int, v: Double) = ("hello", 5, 0.85)
```

```
print(x.w) //imprimie hello
```

```
print(x.i) // imprime 5
```

```
print(x.v) // imprime 0.85
```

```
// Isso também pode acontecer.
```

```
let (wrd, num, val) = x //Serve para renomear os elementos
```

TUPLAS

- ▶ As duplas também podem ser usadas como tipos de retorno;
- ▶ Então isso quer dizer que você pode usar tuplas para criar métodos que retornam múltiplos valores.

```
func getSize() -> (weight: Double, height: Double) {  
    return (250, 80)  
}  
  
let x = getSize()  
print("O peso é \"(x.weight)\"") // 250  
//ou  
print("A altura é \"(getSize().height)\"") // 80
```




RANGE

- ▶ O **Range** do Swift é simplesmente uma representação de dois pontos;
- ▶ Ele pode representar coisas tais quais a seleção de um texto, uma parte de um array, etc;
- ▶ **Range** é uma struct genérica (**Range<T>**), mas nesse caso o **T** é restrito a coisas que podem ser comparadas entre si (protocolo **Comparable**);
- ▶ Então, um **Range<Int>** seria uma boa maneira de especificar um pedaço de um array;
- ▶ Existem outros tipos mais especializados, como um chamado **CountableRange**, que pode conseguir valores consecutivos e tem poderes de iteração e acesso por índice

RANGE: A SINTAXE

- ▶ Existe uma sintaxe especial para criar um **Range**;
- ▶ Seja com `..` (para um range exclusivo) ou com `...` (para um range inclusivo) - sempre falando das duas partes.

```
let array = ["a", "b", "c", "d"]
let a = array[2...3] // a vai ser um pedaço do array contendo ["c", "d"]
let b = array[2..<3] // b vai ser um pedaço do array contendo ["c"]
let c = array[6...8] // erro de execução (array index out of bounds)
let d = array[4...1] // erro de execução (o inicial é maior do que o final)

//Um Range de String não é Range<Int>. É Range<String.Index>
let e = "hello"[2..<4] // isso é != "ll", na verdade nem compila
let f = "hello"[start..
```

RANGE

- ▶ Quando o tipo das partes for um `Int`, então `..<` gera um **CountableRange**;
- ▶ Na verdade, depende se a parte inicial/final é passiva de conversão para **Int**;
- ▶ Um **CountableRange** é enumerável usando o operador **for in**;
- ▶ É assim que se faz, por exemplo, algo como **for (i=0; i<20; i++)**, mas em Swift...

```
for i in 0..<20 {  
    //blabla  
}
```

RANGE

- ▶ E se eu quiser fazer algo do tipo **for (i=0.5; i<15.25; i+=0.3) ???**
- ▶ Os números de ponto flutuante não são conversíveis para **Int**;
- ▶ Então isso quer dizer que **0.5...15.25** não gera um **CountableRange**, que é o que precisamos;
- ▶ Felizmente, no Swift existe uma função global que cria um **CountableRange** à partir de números de ponto flutuante:
for i in stride(from: 0.5, through: 15.25, by: 0.5) { //blablabla }
- ▶ O tipo de retorno da função **stride** na verdade é um **ClosedCountableRange**

ESTRUTURAS DE DADOS EM SWIFT

- ▶ Classes, estruturas e enumerações. Estas são 3 das 4 estruturas de dados fundamentais do Swift;
- ▶ Similaridades na sintaxe

```
class ViewController: UIViewController {  
  
}  
  
struct CalculatorBrain {  
  
}  
  
enum Operations {  
  
}
```

ESTRUTURAS DE DADOS

- Similaridades na sintaxe... Propriedades e funções/métodos:

```
func doIt(withArg arg: Int) -> Double {  
    }  
  
var propriedadeArmazenada: Int = 10  
  
var propriedadeComputada: Int {  
    get {  
        return propriedadeArmazenada * 3  
    }  
    set {  
        propriedadeArmazenada *= newValue  
    }  
}
```

Aqui não entra
um tipo Enum

ESTRUTURAS DE DADOS

- ▶ Similaridades na sintaxe... Inicializadores (mais uma vez, este exemplo não se aplica a enums)

```
init(comValor valor: Double, naRubrica rubrica: String, paraPessoa idPessoa: Int) {  
}
```

AGORA AS DIFERENÇAS...

- ▶ Herança
 - ▶ Somente as classes podem brincar nesse playground
- ▶ Coisas que são do tipo Valor (value type)
 - ▶ Struct
 - ▶ Enum
- ▶ Coisas que são do tipo Referência
 - ▶ Class

VALOR X REFERÊNCIA

- ▶ Valor (value) - **struct** e **enum**
 - ▶ Copiados ao serem passados como argumento para uma função;
 - ▶ Copiados quando atribuídos a uma variável diferente;
 - ▶ Imutáveis quando atribuídos a uma constante (**let**). Argumentos de funções são deste tipo;
 - ▶ **Lembre-se** que qualquer função interna pode modificar a struct/enum, desde que tenha a palavra chave **mutating**

VALOR X REFERÊNCIA

▶ Referência - class

- ▶ São armazenadas na memória heap e passam por contagem automática de referências;
- ▶ Ponteiros para um **class** (**let**) ainda assim podem sofrer mutação via métodos ou propriedades;
- ▶ Quando passadas como argumentos, as classes são passadas por referência e não cópia (o que é passado é o ponteiro para aquela área da memória)

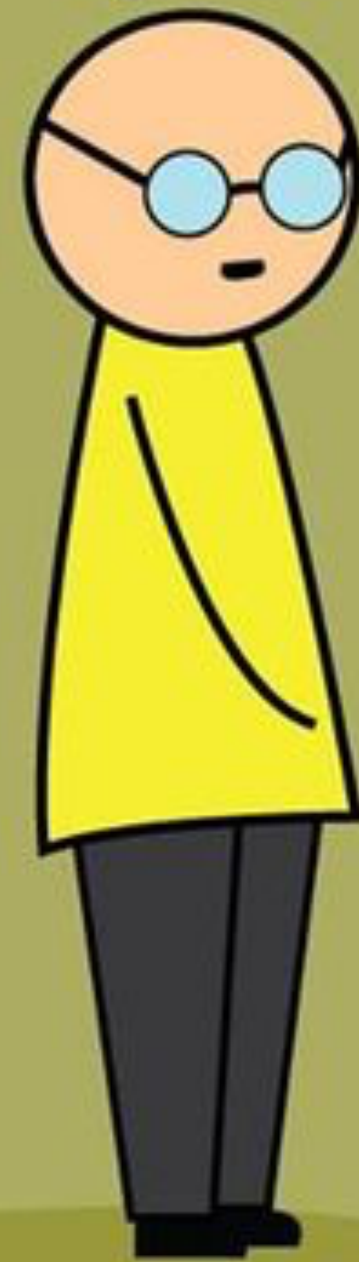
VALOR X REFERÊNCIA

▶ Qual escolher???

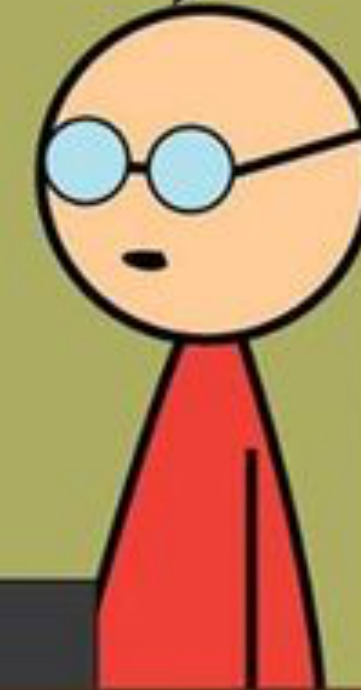
- ▶ Já discutimos essa relação entre **class** e **struct** na aula passada;
- ▶ O uso do **enum** depende da situação. Costumamos usar quando você tem um tipo de dado com valores discretos.
- ▶ O uso da **class** é encorajado quando você precisa escrever uma API que precisa de extensibilidade via herança ou quando você julgar que a passagem por referência faz mais sentido do que a tática copy-on-write que é aplicada a **struct** e **enum**.



How do functions
break up?



They stop calling
each other!



MÉTODOS

- ▶ Vamos falar dos parâmetros...
- ▶ Todos os parâmetros de uma função possuem um nome **interno** e outro **externo**;
- ▶ O nome interno é aquele usado como nome da variável local ao método;
- ▶ O nome externo é aquele que “quem chama” vai usar na chamada;
- ▶ Exemplo...

MÉTODOS

INTERNO

```
func foo(nomeExterno interno: Int, outroNomeExterno outroInterno: Double) {  
    var soma = 0.0  
    for _ in 0..  
        soma += outroInterno  
}  
  
func bar() {  
    let result = foo(nomeExterno: 123, outroNomeExterno: 5.5)  
}
```

EXTERNO

MÉTODO

- ▶ Você ainda pode colocar um `_` se você não quiser que “quem chama” precise usar um nome externo para aquele dado parâmetro;
- ▶ Isso, na imensa maioria das vezes, é feito somente com o primeiro argumento e com nada além dele.

```
func somar(_ n1: Int, com n2: Int) -> Int {  
    return n1 + n2;  
}  
  
func bar() {  
    let result = somar(10, com: 20)  
    print(result)  
}
```

MÉTODO

- ▶ Se você colocar apenas um nome no argumento, ele vai passara ser o nome usado tanto internamente quanto externamente

```
func somar(umNumero: Int, outroNumero: Int) -> Int {  
    return umNumero + outroNumero  
}  
  
func bar() {  
    let result = somar(umNumero: 10, outroNumero: 20)  
    print(result)  
}
```


MÉTODO

- ▶ Você pode sobrescrever métodos e propriedades da sua superclasse;
- ▶ Nos métodos, basta colocar a palavra reservada **override** antes de **func** ou **var**
- ▶ Um método pode ser precedido da palavra **final** para indicar que ele não pode ser sobrescrito;
- ▶ A mesma coisa pode acontecer com uma classe inteira, o que impede a herança.

MÉTODOS

- ▶ Os métodos podem ser de classe ou de instância, assim como as propriedades;
- ▶ Os métodos e propriedades de classe são marcados com a palavra reservada **static**;
- ▶ Por exemplo, na struct **Double** existe um vasto número de propriedades e métodos que não são de instância;
- ▶ Não existe: **53.2.algumMetodo()**
- ▶ Em vez disso, você acessa estes métodos e propriedades usando a classe Double por si só. Exemplo: **Double.pi**



PROGRAMMING FACT

**Its easier to stay awake till 5 AM,
than to wake up at 5 AM.**


```
var umaPropriedadeArmazenada: Int = 42 {
    willSet {
        print("O novo valor é \(newValue)")
    }
    didSet {
        print("O valor antigo é \(oldValue)")
    }
}

override var propriedadeHerdada: String {
    willSet {
        print("O novo valor é \(newValue)")
    }
    didSet {
        print("O valor antigo é \(oldValue)")
    }
}

var operations: Dictionary<String, Operation> = [ ... ] {
    willSet {
        /* Vai ser executado quando uma
           operação for adicionada/removida
        */
    }
    didSet {
        /* Vai ser executado quando uma
           operação for adicionada/removida
        */
    }
}
```


PROPRIEDADES

- ▶ As propriedades também podem se tornar preguiçosas, as vezes;
- ▶ Uma propriedade preguiçosa (chamamos de **lazy**) não é inicializada até que alguém a acesse;
- ▶ No trecho de código usado para inicializar a variável você pode fazer o que bem entender;
- ▶ Isso ainda entra na regra “você tem que inicializar todas as suas propriedades”;
- ▶ As propriedades preguiçosas jamais podem ser constantes (let), podendo ser exclusivamente **var**.

```
/* Seria interessante usar isso, caso o nosso  
CalculatorBrain usasse muitos recursos para  
ser inicializado, tomando muito tempo.  
*/
```

```
lazy var brain = CalculatorBrain()
```

```
lazy var someProperty: UIView = {
```

```
    let view = UIView()  
    return view;
```

```
}()
```

```
lazy var myProperty = self.umMetodoQueCriaUmaVariavel()
```

ARRAYS

Esta é a forma mais usada.

```
var a = Array<String>()  
// ↑ é a mesma coisa de ↓  
var a = [String]()  
  
// O tipo de animais é inferido como Array<String>  
let animais = ["Girafa", "Vaca", "Cachorro", "Pássaro"]  
animais.append("Gato") // Não compila, animais é imutável (porque é let)  
let animal = animais[4] // Erro de execução! (array index out of bounds)
```


ARRAYS

- ▶ Existem alguns métodos interessantes dos Arrays que podem receber **closures** como parâmetros;

```
// Podemos usar para filtrar
let numeros: [Int] = [20, 3, 119, 6, 12, 33]
let numerosGrandes: [Int] = numeros.filter({$0 > 20})

// Também podemos usar para transformar
let comoString: [String] = numeros.map({String($0)})

//Podemos ainda fazer uma redução
let soma = numeros.reduce(0, {$0 + $1})

//Aproveitando a oportunidade...
let soma = numeros.reduce(0, +) //+ é uma função
```

DICTIONARY ATTACK!



DICIONÁRIOS!!!

```
var tabelaBrasileirao = Dictionary<String,Int>()  
//↑ é a mesma coisa que ↓  
var tabelaBrasileirao = [String:Int]() // mais usado  
  
tabelaBrasileirao = ["Vasco":1, "Cruzeiro":11]  
  
// ranking é um Int? (seria nil)  
let ranking = tabelaBrasileirao["São Paulo"]  
  
tabelaBrasileirao["Fluminense"] = 12  
  
// Podemos usar uma tupla para iterar um dicionário  
for (key, value) in tabelaBrasileirao {  
    print("O time \ \(key) está na posição \ \(value)")  
}
```


STRING

- ▶ Uma **String** é composta de caracteres Unicode. Mas existe também o conceito de **Character**;
- ▶ Um Character é o que um humano chama de caracter léxico simples;
- ▶ Isso é verdade mesmo quando a coisa é composta de múltiplos Unicodes;
- ▶ Exemplo: "café" é composto de 5 unicodes, mas tem 4 caracteres;
- ▶ Você pode acessar qualquer caracter de uma String usando a notação **[]**;
- ▶ Mas os índices não são Int. São **String.Index**. Vejamos...

```
// hmm, se a gente quisesse obter o s[0] (a letra "h")?  
let s: String = "hello"  
  
//o tipo de firstIndex não é um Int  
let firstIndex: String.Index = s.startIndex  
  
// firstChar = a letra h  
let firstChar: Character = s[firstIndex]  
let secondIndex: String.Index = s.index(after: firstIndex)  
  
// secondChar = letra e  
let secondChar: Character = s[secondIndex]  
  
// fifthChar = o (a quinta letra)  
let fifthChar: Character = s[s.index(firstIndex, offsetBy: 4)]  
  
// substring = "he"  
let substring = s[firstIndex...secondIndex]
```

"HELLÔ0000"

STRING

- ▶ Mesmo que a String seja indexável, ela não é uma coleção (como é um Array);
- ▶ Apenas coleções podem fazer coisas do tipo **for in** e **indexOf(of:)**;
- ▶ Para a nossa sorte, a String sabe retornar os caracteres que a compõe na forma de um Array que podemos iterar


```
// hmm, se a gente quisesse obter o s[0] (a letra "h")?  
let s: String = "hello"  
  
for c: Character in s.characters {  
    // itera sobre todos os caracteres em s  
}  
  
// quantos caracteres existem em s?  
let count = s.characters.count  
  
// qual é o índice do primeiro espaço?  
let firstSpace: String.Index = s.characters.index(of: " ")
```

STRING

- ▶ É importante observar que a String é um value type (é uma struct);
- ▶ Então, isso quer dizer que quando quer que você a modifique, isso vai criar uma nova cópia;
- ▶ Isso quer dizer também que a mutabilidade dela pode ser controlada com var e let.

OUTRAS CLASSES LEGAIS

▶ **NSObject**

- ▶ É a classe base de todas as classes no Objective-C
- ▶ Alguns recursos avançados do iOS dependem de você herdar de NSObject

▶ **NSNumber**

- ▶ Tratamento genérico para vários tipos de números (é um tipo de referência)

▶ **Date**

- ▶ Não é difícil imaginar que ela serve para guardar datas. Ela também guarda o horário e datas no futuro e no passado;
- ▶ Veja também as classes **Calendar**, **DateFormatter** e **DateComponents** (voltaremos a falar disso quando formos falar de iOS)

▶ **Data**

- ▶ Um tipo que guarda um pacote de bits. É usado para salvar, restaurar e transmitir dados brutos

INICIALIZAÇÃO – QUANDO EU VOU PRECISAR DE UM?

- ▶ Os métodos **init** não são muito comuns no Swift por que as propriedades podem ser inicializadas de outras formas (como fizemos nas duas aulas passadas);
- ▶ As propriedades podem ser opcionais, que, por conseguinte, não vão precisar de um inicializador;
- ▶ Você também pode inicializar as propriedades com closures;
- ▶ Ou usar as propriedades lazy;
- ▶ Você pode ter quantos métodos inicializadores numa classe ou struct quantos você desejar;
- ▶ Cada um desses métodos, naturalmente, precisa ter diferentes argumentos;
- ▶ Para “chamar” seu método **init**, basta escrever o nome do tipo e colocar os parênteses.

INICIALIZADORES - EXEMPLOS

```
var brain = CalculatorBrain()  
var pbo = PendingBinaryOperation(function: +, firstOperand: 23)  
let textNumber = String(45.2)  
let emptyString = String()
```

INICIALIZADORES

- ▶ Você recebe alguns métodos **init** “de graça”;
- ▶ Um deles é um que não recebe qualquer argumento e ele existe para qualquer estrutura que seja uma classe;
- ▶ A título de curiosidade, no Swift não existe uma classe base para tudo, tal qual no Objective-C e no Java, por exemplo;
- ▶ A struct, como já vimos na aula passada, ganha o inicializador de brinde com todas as suas propriedades;

INICIALIZADORES

- ▶ O que eu posso fazer dentro de um método **init**?
- ▶ Você pode "setar" toda e qualquer propriedade, mesmo aquelas que já tem um valor padrão;
- ▶ Você pode "setar " constantes (let);
- ▶ Você pode chamar outros métodos **init**;
- ▶ Numa classe, você pode evocar o inicializador da superclasse, se houver;
- ▶ Mas existem algumas regras que temos que seguir para usar os métodos **init** em classes...

INICIALIZADORES – O QUE VOCÊ PRECISA FAZER DENTRO DO INIT?

- ▶ Ao final do método **init**, você precisa garantir que todas as propriedades tiveram seus valores preenchidos (os opcionais podem ser iniciados com nil);
- ▶ Existem dois tipos de inicializadores em uma classe:
 - ▶ O designado (designated); e
 - ▶ O de conveniência (convenience)
- ▶ Um **init** designado deve apenas e somente evocar o **init** designado da superclasse;
- ▶ Você precisa inicializar todas as propriedades introduzidas na sua classe antes de chamar o inicializador da superclasse;
- ▶ Você precisa evocar o inicializador da superclasse antes de lidar com qualquer propriedade herdada;

INICIALIZADORES – O QUE VOCÊ PRECISA FAZER DENTRO DO INIT?

- ▶ Um inicializador de conveniência deve apenas e somente evocar um inicializador designado da sua própria classe;
- ▶ Um inicializador de conveniência precisa chamar um inicializador da classe antes de lidar com qualquer propriedade;
- ▶ A chamada a outros métodos **init** precisa terminar antes de você poder acessar propriedades ou evocar métodos
- ▶ Ufa!

HERANÇA DE INICIALIZADORES

- ▶ Se você não implementar nenhum **init** designado, você vai herdar todos os designados da superclasse;
- ▶ Se você sobrescrever **todos** os inicializadores designados da superclasse, você vai herdar todos os inicializadores de conveniência;
- ▶ Se você não implementar qualquer **init**, você vai herdar todos da superclasse;
- ▶ Qualquer um dos **init** citados aqui está sujeito às regras mencionadas no slide anterior.

INICIALIZADOR OBRIGATÓRIO

- ▶ Uma classe pode indicar um ou mais dos seus métodos **init** como sendo obrigatórios, pelo uso da palavra reservada **required**;
- ▶ Feito isto, qualquer eventual subclasse vai, necessariamente, precisar implementar este(s) método(s) **init**;
- ▶ Mais uma vez, os inicializadores mencionados aqui estão sujeitos àquelas mesmas regras.

INICIALIZADORES QUE FALHAM

- ▶ Eventualmente, um inicializador pode falhar ao criar a instância do objeto;
- ▶ Quando ocorrer uma situação deste tipo, o inicializador deve ser do tipo opcional - indicando que quando houver a falha, ele irá retornar **nil**;

```
init?(outro: OutroTipo) {  
    /*  
        Aqui posso retornar nil  
        caso a conversão falhe  
    */  
}
```




TIPOS CORINGA – ANY E ANYOBJECT

- ▶ Estes são tipos comumente usados para compatibilidade com as APIs do Objective-C;
- ▶ Mas, ultimamente, estes tipos coringa estão cada vez menos sendo usados, uma vez que a maioria das APIs do Objective-C foram atualizadas quando o Swift surgiu;
- ▶ Mas, precisamos conhecer, então lá vamos nós!
- ▶ Variáveis do tipo **Any** podem armazenar qualquer coisa;
- ▶ Variáveis do tipo **AnyObject** podem armazenar qualquer coisa que seja uma classe;
- ▶ É possível converter o tipo para um tipo conhecido;
- ▶ Como o Swift é fortemente tipado, o correto é evitar usar tipos Any e AnyObject

ANY E ANYOBJECT

- ▶ Já que isso ainda existe, onde você vai ver isso no iOS?
 - ▶ Algumas vezes (é raro), você vai usar em argumentos de métodos que podem receber diferentes tipos como parâmetro;
 - ▶ Existe um método do UIViewController que vamos usar na próxima aula que é um exemplo disso (é um método relacionado à mudança de tela)
- ▶ Onde mais?
 - ▶ Você pode usar para criar coleções que podem receber valores de diferentes tipos, mas muito provavelmente você vai acabar usando uma solução baseada em enums, como fizemos no CalculatorBrain;
 - ▶ Então, o que sobra é quando você quer escrever algum código que seja compatível com Objective-C
- ▶ Ahhh! você ainda pode usar para retornar um objeto sobre o qual você não deseja que quem chama saiba qual é.

ANY E ANYOBJECT

- ▶ Normalmente não se pode usar diretamente, uma vez que não se sabe qual é o tipo;
- ▶ Em vez disso, convertemos para algum tipo conhecido na hora de usar;
- ▶ A conversão é feita com o operador **as?**
 - ▶ Como é de se imaginar, a conversão nem sempre é possível, então o retorno da conversão é um opcional;
- ▶ Você pode verificar se a conversão é possível usando o operador **is**, que retorna true ou false

```
if let agoraEuSei as? TipoConhecido {  
    // A conversão funcionou  
}
```



TYPECAST

- ▶ Por falar nisso, conversão de tipos com **as?** não é uma exclusividade de Any e AnyObject;
- ▶ Você pode converter entre tipos com **as?**, desde que a conversão faça sentido;
- ▶ Majoritariamente, você pode converter um objeto para a superclasse;
- ▶ Mas também pode ser usado para converter para protocolos (veremos isso depois)
- ▶ Com o passar do curso, vamos ir entrando nos detalhes disso. Falaremos disso também na nossa aula do sábado!

PROMETO QUE

TÁ QUASE ACABANDO

USERDEFAULTS

- ▶ Um banco de dados minúsculo e ultra-leve;
- ▶ O **UserDefaults** é, essencialmente, um pequeno banco de dados que é perene entre os lançamentos do seu aplicativo;
- ▶ Ele é ótimo para armazenar pequenas configurações e coisas do tipo;
- ▶ Nunca use para coisas grandes!

AAAAAAHHHHHHHHHHHHHHH!

USERDEFAULTS – O QUE PODE?

- ▶ Lá só podem entrar coisas que podem entrar num objeto chamado **PropertyList**;
- ▶ Isso quer dizer que podem entrar quaisquer combinações de:
 - ▶ Array
 - ▶ Dictionary
 - ▶ String
 - ▶ Date
 - ▶ Data
 - ▶ Int, Double, etc (qualquer tipo numérico)

SERÁ?

AINDA BEM QUE ACABOU!

Aluno cansado



DEVER DE CASA!!!

LEITURA PARA A SEMANA

- ▶ Neste link: https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/index.html
- ▶ Lembre-se: Para chegar onde 1% chega, você precisa fazer o que 99% não faz.