

Introdução ao desenvolvimento iOS (aula 3)

Curso de desenvolvimento iOS em Swift
Professores: Ráfagan Abreu e Lucas Paim
CINQ Technologies & Scopus

Swift: Hands on

- Actions:
 - Tipos de actions para UIButton.
- ViewController;
- Funções Swift:
 - Sintaxe com influência do Objective-C;
 - Underline em nomes de funções.
- Imprimindo valores no debugger ao pressionar botão;
- Linkando actions do storyboard para código e código para storyboard;

Swift: Hands on

- Analisando o autocomplete;
- Navegando com command: Hierarquias de UIButton;
- Help com option: Documentação do UIButton;
- Imprimindo o texto do botão no console;
- Var e Let;
- Optionals e unwrap (*):
 - Dica: Muito cuidado com unwraps!

Swift: Hands on

- Outlet do label;
- Criar action único para concatenar string ao display da calculadora;
- Cuidado ao renomear actions e outlets;
- Criar lógica com propriedade booleana para desconsiderar zero à esquerda;
- Implementar lógica do AC para limpar display (*);

Swift: Hands on

- Refatorar operações para suportar método de execução de operação:
 - Utilizar if let e switch;
 - Reparar que Swift é UNICODE.
 - Verificar utilizando breakpoints cada case do switch.
- Analisar Stack Trace do debugger;
- Configurar Auto Shrink do label no Storyboard (*);

Swift: Hands on

- Implementar troca de sinal:
 - Optional Double() VS String();
- Trocar layout para iPhone 5 e criar operação de raiz quadrada;
- Implementar operação de raiz quadrada;
- Implementar computed property para o texto do display para Double (*).

Exercício 4: Swift

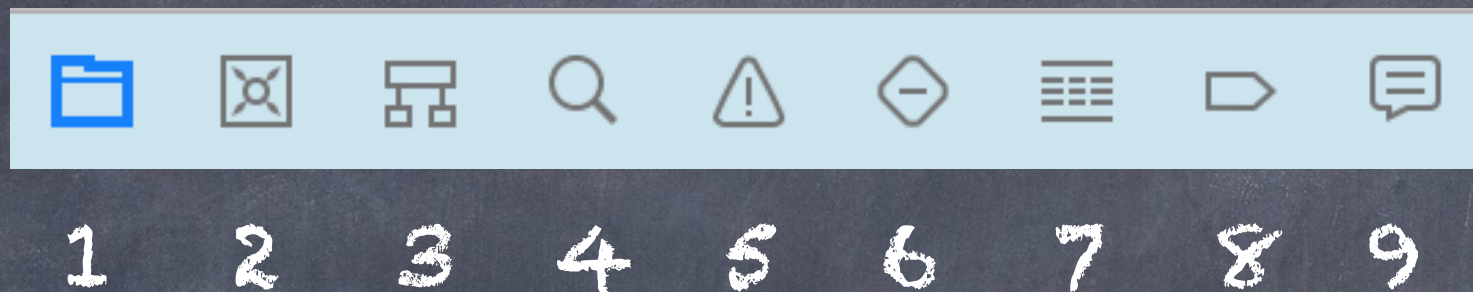
- Implemente as operações binárias: +, −, × e ÷;
- Desafio 1: Implemente % e =;
- Desafio 2: Varie AC também para C;

			0
AC	±	%	÷
7	8	9	×
4	5	6	−
1	2	3	+
0		,	=
			√

Introdução ao desenvolvimento iOS (aula 4)

Curso de desenvolvimento iOS em Swift
Professores: Ráfagan Abreu e Lucas Paim
CINQ Technologies & Scopus

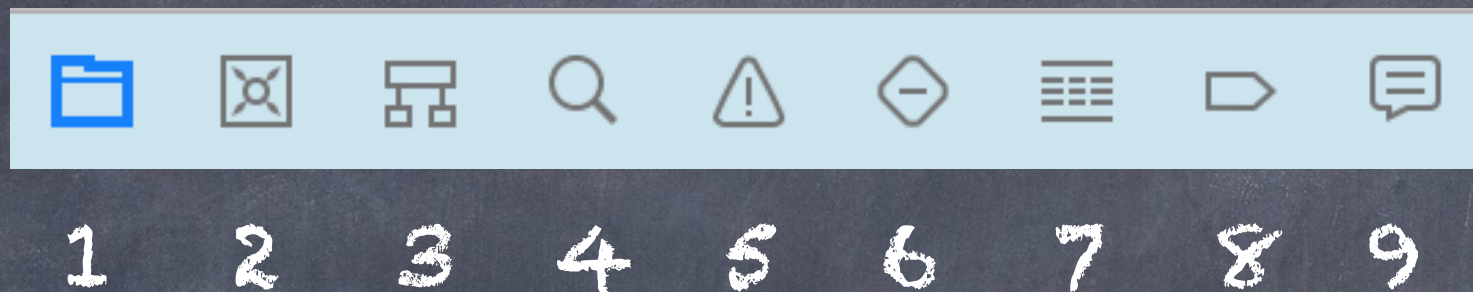
Xcode 9: Hands On



- Navigator:

1. Project: Arquivos, pastas e grupos;
2. Source Control: Git;
3. Symbol: Hierarquias, métodos;
4. Find: Busca de texto, declarações, regex, ...;
5. Issue: Erros e Warnings;

Xcode 9: Hands On



- Navigator:

- 6. Test: Testes que passaram e reprovaram;
- 7. Debug: Um pequeno profiler da aplicação;
- 8. Breakpoint: Todos os breakpoints setados;
- 9. Report: Histórico de ações.

Xcode 9: Hands On



1 2 3 4

- Library:

1. File Template: criação de arquivos;
2. Code Snippet: pequenos trechos de código do projeto nas diversas linguagens utilizadas pelo ecossistema da Apple;
3. Object: elementos de UI, gestures, etc...;
4. Media: assets.

Xcode 9: Hands On

- Adicione a sua conta do GitHub ao Xcode nas preferências;
- Clone o repositório da aula:
 - <https://github.com/Guizion/calculadora>.
- Analise o Source Control Navigator;
- Verifique as views do git (comparison, blame e log);
- Analise as opções do menu "Source Control".

Exercício 5: Atalhos

- Experimente alguns atalhos do Xcode:
- Legenda: Command(⌘), Shift(⇧), Option(⌥), Control(^)
- Abrir arquivo: ⌘ + ⇧ + O
- Commit: ⌘ + ⌥ + C
- Buscar na página: ⌘ + F
- Buscar no projeto: ⌘ + ⇧ + F
- Mover linha para cima ou para baixo: ⌘ + ⌥ + [ou]
- Corrigir indentação: ^ + I
- Comentar linha: ⌘ + /
- Limpar console: ⌘ + K
- Revelar arquivo no navigator: ⌘ + ⇧ + J
- Actions: ⌘ + Click Esquerdo
- Split de dois scripts: ⌥ + Click Esquerdo
- Rodar projeto: ⌘ + R
- Build projeto: ⌘ + B
- Limpar projeto: ⌘ + ⇧ + K
- Options sobre variáveis

MVC

- Todo o ecossistema Cocoa Touch é pensado sobre o padrão Model View Controller (MVC);
- **Model:** O que sua aplicação é (regras de negócio, entidades);
- **Controller:** Como o seu Model é apresentado ao usuário (lógica da camada de UI);
- **View:** Componentes de UI (genéricos).
- No iOS, o Controller é completamente acoplado à View (por Outlets, Actions). Por isso, o chamaremos de ViewController:
 - Pelo fato do Controller assumir muitas responsabilidades, alguns brincam definindo a sigla MVC para iOS como Massive View Controller.

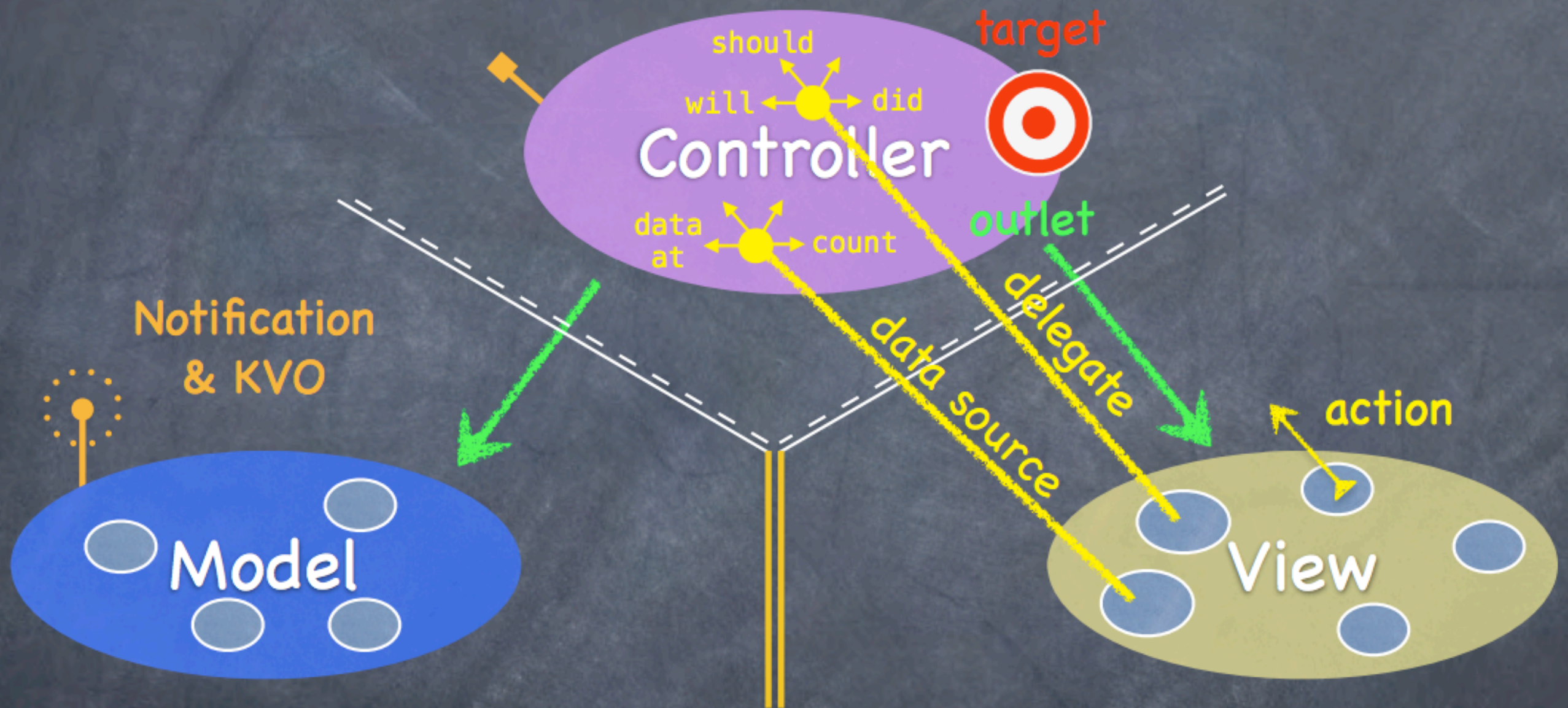
MVC

- Controllers sempre podem falar diretamente com Models;
- Controllers também podem falar com as Views, por meio de **Outlets**;
- Model e View nunca falam entre si;
- O Controller escuta os eventos (**Actions**) da View;
- As vezes, a View precisa sincronizar com o Controller, dando dicas sobre o que vai fazer (should, will, did). Ela, portanto, delega (**Delegate**) ao Controller as decisões;

MVC

- Não existe acoplamento entre a View e os dados que ela exibe:
 - Exemplo: Qual é o objeto do índice i da table (data at) ou quantos elementos existem na collection (count);
 - O Controller assume o papel de fornecer esses dados (**Data Source**), formatando Models para exibir esses dados adequadamente.
- Se o Model não se comunica com a View, o que acontece se ele mudar?
 - Resposta: Observers (Notification, Key Value Observer).

MVC – Stanford



MVC: Hands On

- Vamos adaptar o nosso exercício da calculadora para trabalhar segundo o padrão MVC;
- Crie um novo arquivo Swift chamado CalculatorManager: Ele será a nossa camada de modelo;
- Crie uma struct "CalculatorManager". Por que struct?
 - Objetos de class vivem na heap e são sempre passados como referência;
 - Já objetos de struct são passados sempre como cópia: Isso é bom, porque valores são thread-safe, mais previsíveis (stateless),
 - Outra vantagem é que não precisamos inicializar os atributos da classe.

MVC: Hands On

- Crie métodos para setar e executar operadores;
- Crie uma variável privada que armazenará o valor exibido no display da calculadora;
- Crie uma property readonly que retorne o valor acumulado no display (*);

MVC: Hands On

```
9  import Foundation
10
11  struct CalculatorManager {
12      private var accumulator: Double?
13
14      var result: Double? {
15          get {
16              return accumulator
17          }
18      }
19
20      func performOperation(_ symbol: String) {
21
22      }
23
24      func setOperation(_ operation: Double) {
25
26      }
27  }
```


MVC: Hands On

- Implemente a função `setOperator`:
 - Perceba que temos um erro (`self is immutable`);
 - Como structs são sempre cópia, ao alterarmos um valor, é necessário criar uma nova struct;
 - O Swift faz isso automaticamente, anotando o método com **mutating**;
 - Detalhe: não poderemos criar objetos de struct com `let` caso algum método esteja anotado como `mutating` (*).

MVC: Hands On

- Agora, vamos refatorar o nosso ViewController para trabalhar em conjunto com o Model:
 - Crie a instância do manager no ViewController;
 - Em performOperation, realize as chamadas do manager;
 - Mova o código apropriadamente para o CalculatorManager (*).

```
43  @IBAction func performOperation(_ sender: UIButton) {  
44      if userIsTyping {  
45          userIsTyping = false  
46          manager.setOperation(displayValue)  
47      }  
48      if let mathSymbol = sender.currentTitle {  
49          manager.performOperation(mathSymbol)  
50      }  
51      if let result = manager.result {  
52          displayValue = result  
53      }  
54  }  
55  }
```


MVC: Hands On

- Vamos agora observar o switch criado:
 - O Switch é considerado um "Bad Smell", porque:
 - Normalmente causam duplicação de código;
 - Todos os cases são testados no pior caso.
 - Podemos substituí-lo por:
 - Polimorfismo;
 - Dicionários.

MVC: Hands On

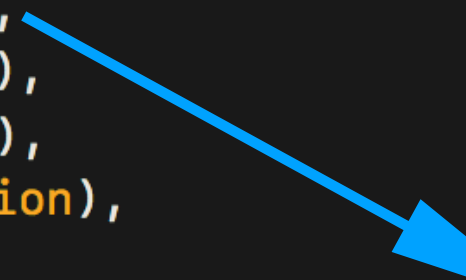
- Vamos substituir o switch por um dicionário:
 - A key será o caractere da operação;
 - O value será a operação;
 - A classe utiliza generics e possui syntax sugar [].
- A representação das operações será feita utilizando enums:
 - Enums, assim como structs, são recursos bem mais poderosos no Swift do que no Objective-C.
- Implementaremos as funções das operações. (*)

MVC: Hands On

```
42     private enum Operation {  
43         case unaryOperation((Double) -> Double)  
44         case binaryOperation((Double, Double) -> Double)  
45     }
```

Enum

```
47     private let operations: Dictionary<String, Operation> = [  
48         "√": Operation.unaryOperation(sqrt),  
49         "±": Operation.unaryOperation(changeSign),  
50         "+": Operation.binaryOperation(plus),  
51         "-": Operation.binaryOperation(minus),  
52         "×": Operation.binaryOperation(times),  
53         "÷": Operation.binaryOperation(division),  
54     ]
```



Dicionário

```
15     func plus(_ v1: Double, _ v2: Double) -> Double {  
16         return v1 + v2  
17     }
```


MVC: Hands On

```
56     mutating func performOperation(_ symbol: String) {  
57         if let operation = operations[symbol] {  
58             switch operation {  
59                 case .unaryOperation(let op):  
60                     accumulator = (accumulator != nil ? op(accumulator!) : accumulator)  
61                 default:  
62                     break  
63             }  
64         }  
65     }
```

Exemplo de utilização

- O próximo passo é implementar o enum das operações binárias:
 - Precisamos armazenar o valor anterior e a operação para executar quando o novo valor e o botão igual forem pressionados.
- Perceba que setOperation não possui um bom nome: Vamos refatorá-lo (refactor > rename) para setOperand.

MVC: Hands On

```
42     private struct PreviousBinaryOperation {  
43         let function: (Double, Double) -> Double  
44         let firstOperand: Double  
45  
46         func perform(with secondOperand: Double) -> Double {  
47             return function(firstOperand, secondOperand)  
48         }  
49     }
```

Struct do histórico

```
case .binaryOperation(let op):  
    if accumulator != nil {  
        binaryOperationMemory = PreviousBinaryOperation(function: op, firstOperand: accumulator!)  
        accumulator = nil  
    }
```

Case do performOperation

MVC: Hands On

- Agora, o Equals:
 - Insira-o no enum Operation sem argumentos;
 - Adicione-o ao dicionário para "=";
 - Crie uma função para definir o accumulator de acordo com valor e a operação em memória;
 - No case de performOperation, chame a função criada a partir de um case equals. (*)

```
86     mutating func doPreviousBinaryOperation() {  
87         if binaryOperationMemory != nil && accumulator != nil {  
88             accumulator = binaryOperationMemory!.perform(with: accumulator!)  
89             binaryOperationMemory = nil  
90         }  
91     }
```


MVC: Hands On

- As funções do dicionário em sua maioria estão sendo criadas, mas já que vamos utilizá-las somente em um lugar, isso é realmente necessário?
 - Vamos substituí-las por lambdas!
- Por exemplo, na soma, você pode: (*)
 - `"-": Operation.binaryOperation({ (v1: Double, v2: Double) -> Double in return v1 - v2 })`
 - Ou:
`"+": Operation.binaryOperation({ (v1, v2) in v1 + v2 })`
 - Ou simplesmente:
`"+": Operation.binaryOperation({ $0 + $1 })`

Exercício 6: Melhorando a calculadora

1. Não permita 0 negativo e múltiplos 0's;
2. Ao pressionar AC, limpar memória;
3. Desenvolva a operação de porcentagem e C na nova arquitetura;
4. Permitir encadeamento de operações binárias e do igual;
5. Permita operações de ponto flutuante (.), mostrando valores .0 apenas quando existirem dados;
6. Ainda temos espaço para mais 3 botões. Implemente uma das seguintes operações: $\sqrt[n]{}$ (raiz de n), \log_x (logaritmo na base x), utilização de constantes (e, π), sin ou cos.