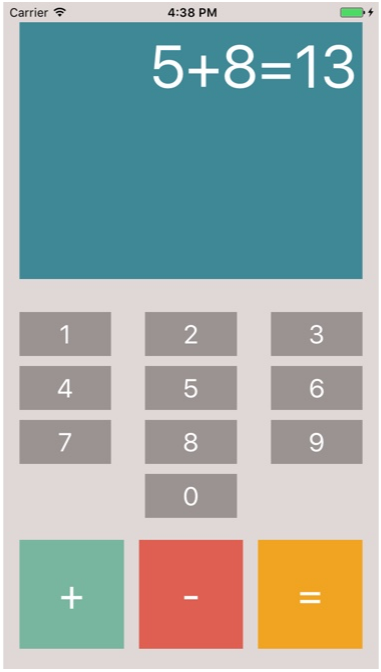
PAYAN Mickaël
Projet 5 : Améliorez une application existante.

Introduction

Un client m'a contacté afin que j'améliore et finalise son projet. Celui-ci est CountOnMe, une calculatrice.

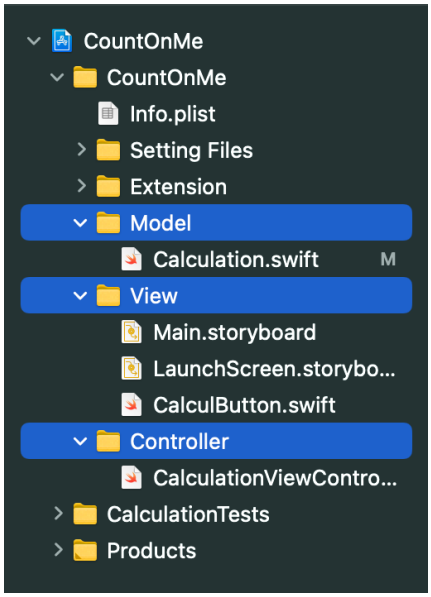


Calculatrice CountOnMe

- Le design de l'application n'a pas été finalisé dans le projet Xcode : rien n'est responsive.
- L'architecture du projet ne respecte pas encore les bonnes pratiques de développement (pas de MVC).
- L'ensemble de l'application n'est pas testé.
- Seul la soustraction et l'addition ont été faits, il manque la division et la multiplication.

Architecture MVC

Tout d'abord, avant de s'attaquer au design de l'application, aux tests ainsi qu'aux fonctions manquantes afin de finaliser le projet, j'instaure l'architecture MVC à celui-ci.



- MODEL
  ⇒ Contient les données à afficher.
- VIEW
  ⇒ Contient la présentation de l'interface graphique.
- CONTROLLER
  ⇒ Contient la logique concernant les actions effectuées par l'utilisateur.

Modèle de données : Calculation

Le modèle de données contient un fichier « Calculation ». C'est dans celui-ci que seront contenues les données à afficher tel que :

- **Extension privée et propriétés avec fermetures**

```swift
private extension Calculation {
    var elements: [String] {
        return displayCalculText.split(separator: " ").map { "\($0)" }
    }

    var expressionIsCorrect: Bool {
        return elements.last != "+" && elements.last != "-" && elements.last != "x" && elements.last != "/"
    }

    var expressionHaveEnoughElement: Bool {
        return elements.count >= 3
    }

    var canAddOperator: Bool {
        return elements.last != "+" && elements.last != "-" && elements.last != "x" && elements.last != "/"
    }
}
```

Tout d'abord voici quatre propriétés :

⇒ elements : propriété de type tableau de String qui retourne l'affichage du texte (« displayCalculText »). La méthode .split(separator : _) permet de fractionner les caractères.
⇒ expressionIsCorrect, canAddOperator : propriétés du type Booléen. Retournent et vérifient que les derniers éléments ne soient pas identiques.
⇒ expressionHaveEnoughElement : propriété du type Booléen qui retourne et vérifie que les éléments soient supérieurs ou égaux à 3.

- **Affichage et communication indirect**

```
13      var displayResultHundler: (_ result: String) -> Void = {_ in }
14
15      private var displayCalculText = "" {
16          didSet {
17              displayResultHundler(displayCalculText)
18          }
19      }
```

La propriété « displayCalculText », implémentée d'un observateur de propriétés didSet va agir directement lorsque la valeur est modifiée.

La propriété « displayResultHandler » qui est une fermeture, va permettre une communication indirecte entre le modèle et contrôleur. Et permettra donc de renvoyer les valeurs inscrites dans « displayCalculText ».

L'action de l'affichage des valeurs se fera dans le contrôleur.

- **Les basiques : effacer, ajouter un chiffre/opérateur**

```
21      func clearAll() {
22          displayCalculText = "0"
23      }
24
25      func addNumber(_ number: String) {
26          if displayCalculText == "0" || displayCalculText.contains("=")  {
27              displayCalculText = ""
28          }
29          displayCalculText = displayCalculText + String(number)
30      }
31
32      func addOperator(_ operator: String) {
33          replaceOperator()
34          displayCalculText.append(`operator`)
35          if displayCalculText.contains("=") {
36              resolveOperation()
37          }
38      }
```

La fonction « clearAll » va permettre simplement d'effacer le calcul et de le remplacer par 0.

La fonction « addNumber », comme sa désignation l'indique permet d'ajouter les chiffres. Celle-ci est implémentée d'une condition if afin que, si la valeur actuelle de « displayCalculText » est strictement égale à 0 ou contient un « = », alors la valeur sera supprimée. Cela permet tout simplement de débuter le calcul en supprimant le 0, ou si l'opération est résolue, de le commencer directement. Après quoi, le chiffre s'ajoute (l.29).

Enfin la fonction « addOperator » permet d'ajouter les opérateurs (l.34). Plutôt que l'application retourne une erreur fatale (crash), car « displayCalculText » contient déjà un opérateur, j'ai préféré que celui remplacé grâce à la méthode suivante.

```
40      func replaceOperator() {
41          if !canAddOperator {
42              displayCalculText.removeLast(3)
43          }
44      }
```

⇒ removeLast( ) : supprime le nombre spécifié d'éléments de la fin de la collection.

Enfin la condition if qui suit, si le texte affiché contient un « = », alors l'opération peut être résolue.

- **Résoudre l'opération**

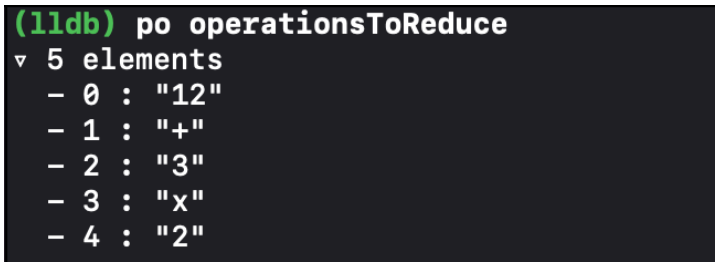
```
46      func resolveOperation() {
47          guard expressionIsCorrect, expressionHaveEnoughElement else {
48              return replaceOperator()
49          }
50          var operationsToReduce = elements
51
52          while operationsToReduce.count > 1 {
53              var operandIndex = 1
54              if let index = operationsToReduce.firstIndex(where: { $0.contains("x") || $0.contains("/")}) {
55                  operandIndex = index
56              }
57              guard let left = Double(operationsToReduce[operandIndex-1]),
58                    let right = Double(operationsToReduce[operandIndex+1]) else {
59                  return
60              }
61              let operand = operationsToReduce[operandIndex]
62              var result: Double
63              switch operand {
64              case "+": result = left + right
65              case "-": result = left - right
66              case "x": result = left * right
67              case "/": result = left / right
68                  if left == 0 || right == 0 {
69                      result = 0
70                  }
71              default: return
72              }
73              operationsToReduce.remove(at: operandIndex+1)
74              operationsToReduce.remove(at: operandIndex-1)
75              operationsToReduce.insert("\(result.removeZeroFromEnd())", at: operandIndex)
76              operationsToReduce.remove(at: operandIndex-1)
77          }
78          displayCalculText.append(" = \(operationsToReduce.first!)")
79      }
80  }
```

Premièrement, vérifier si l'expression est correcte et si l'expression a assez d'éléments. Si la condition est respectée, le code se poursuit ligne 50, sinon elle retourne la méthode « replaceOperator ».

Imaginons que tout se passe bien. Tout d'abord une instance « operationsToReduce » pour récupérer tous les éléments.

Ensuite, une boucle while vérifie le nombre d'éléments et doit être strictement supérieur à 1. Le premier élément étant à l'index 0 dans un tableau, qui correspond au premier chiffre/nombre, l'index 1 lui correspondra à un opérateur et ainsi de suite.

Exemple :

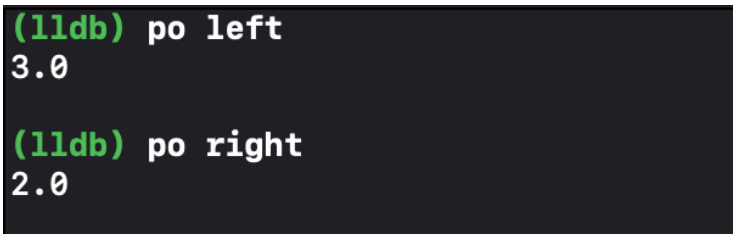


Index : valeur

Une fois à l'intérieur de la boucle while, une propriété « operandIndex » pour valeur un Int égal à 1.

L'instruction if let qui suit permet de réaliser en un seul coup le teste de nullité ainsi que de déballer l'optionnel (unwrapping). Dans ce cas-là, « index » n'est pas égal à nil, et va permettre de réaliser la priorité de calcul lorsqu'un opérateur multiplier ou diviser est présent dans les éléments. Après quoi, « operandIndex » est égal à la valeur « index ». Dans l'exemple ci-dessus, la valeur « index » est 3.

Guard let permet de déballer les optionnels « right » and « left », les deux permettant d'identifier le chiffre ou nombre aux extrémités de l'opérateur.



Left a pour index « operandToReduce[operandIndex-1] et Right operandToReduce[operandIndex+1]. Ça veut dire quoi ? Simplement que l'opérateur multiplier a pour index 3(operandIndex), donc left 2(operandIndex-1) et right 4(operandIndex+1).

```
  - 2 : "3"
  - 3 : "x"
  - 4 : "2"
```

Index : valeur

Par la suite, la constante « operand » permet simplement de récupérer l'opérateur inscrit. La constante « result » du type Double permettra d'obtenir, comme sa désignation l'indique, le résultat. Comment ?

L'instruction switch va permettre d'inspecter la valeur de l'opérateur(operand) et de la faire correspondre à un cas. Le cas identifié permettra d'obtenir un résultat (result) juste.

Enfin qu'est-ce qu'il se passe ici ?

```
73              operationsToReduce.remove(at: operandIndex+1)
74              operationsToReduce.remove(at: operandIndex-1)
75              operationsToReduce.insert("\(result.removeZeroFromEnd())", at: operandIndex)
76              operationsToReduce.remove(at: operandIndex-1)
```

Tout d'abord, l'élément à l'extrémité droite de l'opérateur multiplié est supprimé, soit l'index 4 :

```
(lldb) po operationsToReduce
▿ 4 elements
  - 0 : "12"
  - 1 : "+"
  - 2 : "3"
  - 3 : "x"
```

Ensuite, l'élément à l'extrémité gauche de l'opérateur multiplier est supprimer à son tour, soit l'index 2 :

```
(lldb) po operationsToReduce
▿ 3 elements
  - 0 : "12"
  - 1 : "+"
  - 2 : "x"
```

Maintenant, il est temps d'ajouter le résultat, et celui-ci aura pour index 3 :

```
(lldb) po operationsToReduce
▿ 4 elements
  - 0 : "12"
  - 1 : "+"
  - 2 : "x"
  - 3 : "6"
```

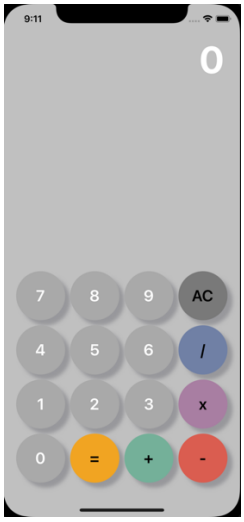
Enfin, le dernier élément à supprimer est l'opérateur lui-même qui a pour index 2 :

```
(lldb) po operationsToReduce
▿ 3 elements
  - 0 : "12"
  - 1 : "+"
  - 2 : "6"
```

Et c'est fini ? Enfin le résultat final ? Non, avant d'arriver à la ligne 78, le procédé se reproduit pour le calcul restant : 12 + 6.

Design et interface graphique

Le design de l'application a été modifier et finaliser. L'application est maintenant responsive et s'adapte à toutes les tailles d'écrans.

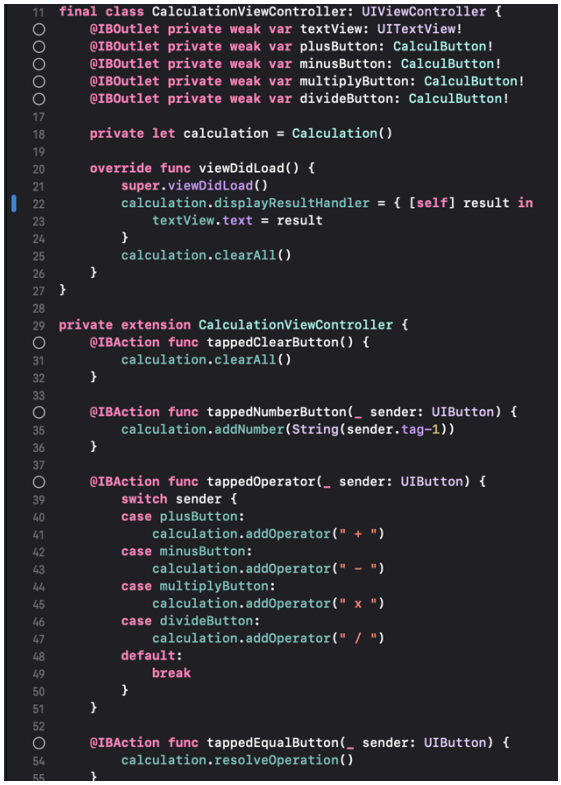



```swift
@IBDesignable final class CalculButton: UIButton {
    override func layoutSubviews() {
        super.layoutSubviews()
        layer.cornerRadius = frame.height / 2
        layer.shadowColor = UIColor.systemGray.cgColor
        layer.shadowOpacity = 0.8
        layer.shadowOffset = CGSize(width: 8, height: 8)
        titleLabel?.font = UIFont.boldSystemFont(ofSize: 25)
    }
}
```

Un fichier « CalculButton » a été ajouter dans la partie View et c'est normal. Cette classe vient modifier et obtenir un rendu direct sur l'interface graphique grâce au « IBDesignable ». Les méthodes suivantes permettent d'arrondir, d'ombrer et de définir la taille de la police de caractère des boutons.

Les actions de l'utilisateur

Le modèle de données est réalisé, l'interface graphique aussi, cependant l'application ne fonctionne toujours pas. Du moins l'utilisateur ne peut pas interagir avec l'application.



```swift
final class CalculationViewController: UIViewController {
    @IBOutlet private weak var textView: UITextView!
    @IBOutlet private weak var plusButton: CalculButton!
    @IBOutlet private weak var minusButton: CalculButton!
    @IBOutlet private weak var multiplyButton: CalculButton!
    @IBOutlet private weak var divideButton: CalculButton!

    private let calculation = Calculation()

    override func viewDidLoad() {
        super.viewDidLoad()
        calculation.displayResultHandler = { [self] result in
            textView.text = result
        }
        calculation.clearAll()
    }
}

private extension CalculationViewController {
    @IBAction func tappedClearButton() {
        calculation.clearAll()
    }

    @IBAction func tappedNumberButton(_ sender: UIButton) {
        calculation.addNumber(String(sender.tag-1))
    }

    @IBAction func tappedOperator(_ sender: UIButton) {
        switch sender {
        case plusButton:
            calculation.addOperator(" + ")
        case minusButton:
            calculation.addOperator(" - ")
        case multiplyButton:
            calculation.addOperator(" x ")
        case divideButton:
            calculation.addOperator(" / ")
        default:
            break
        }
    }

    @IBAction func tappedEqualButton(_ sender: UIButton) {
        calculation.resolveOperation()
    }
```

Une instance du modèle « Calculation » est créée afin d'avoir accès à ses méthodes.

Nous pouvons constater, que l'affichage est réalisé par la fermeture « displayResultHandler » dans lequel y est implémenté un IBOutlet du type UITextView. Le fait de passer dans le « ViewDidLoad » permet que le code soit lancé une fois que le mainView a été chargé (dès le lancement de l'application).

Enfin les actions sont minimalistes car les données sont contenues dans le modèle « Calculation », soit ajouter un chiffre, un opérateur et résoudre l'opération.

Conclusion

La multiplication et division ont été ajoutées. L'application respecte l'architecture MVC, s'adapte à toutes les tailles d'écrans (responsive) en mode portrait, est exempte de toute erreur et est fonctionnelle à partir d'iOS 11.

L'application CountOnMe est maintenant finalisée et prête à être livré.