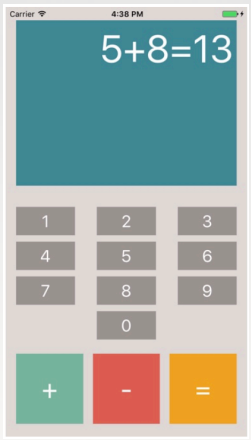
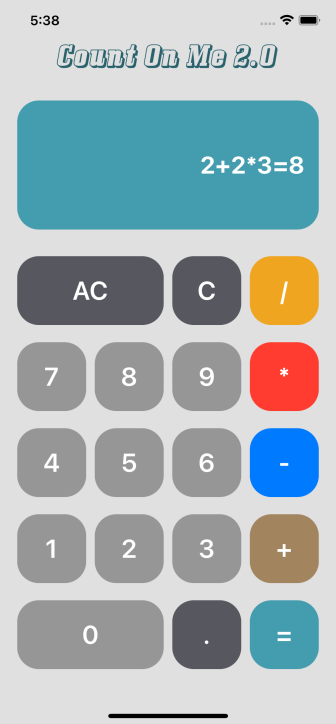
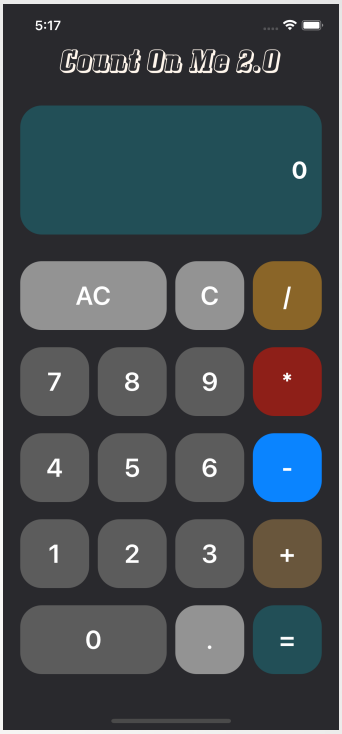


CountOnMe 2.0
SwiftUI

Présentation de l'application:

CountOnMe est une Application iOS pour iPhone réalisée dans le cadre du parcours de développeur iOS OpenClassRooms.
Le présent document fait référence à la version SwiftUI de l'application développée dans le cadre du projet final de mon parcours d'apprentissage.





- L'application est une calculatrice qui s'affiche uniquement en mode portrait.
- La priorité des opérations est prise en compte, les multiplications et divisions sont prioritaires sur les additions et soustractions.
- L'expression complète reste affichée à l'écran avec le résultat après appui sur la touche égal.
- Si l'utilisateur commence par appuyer sur le symbole decimal, un zero est automatiquement ajouté devant.
- Lorsqu'un résultat est affiché à l'écran:
    - l'appui sur un opérateur récupère le précédent résultat.
    - l'appui sur un numéro efface l'expression précédente .

Architecture générale:

L'architecture est simple et consiste en une vue (ContentView) qui affiche les boutons de la calculatrice. Nous avons extrait l'écran (DisplayView) dans l'idée de décomposer au maximum notre interface et d'optimiser la lisibilité du code, mais compte tenu de la simplicité de l'application, tout notre code aurait pu être écrit dans Content View.

La logique de la calculatrice est en revanche déportée dans un fichier Calculator.swift qui constitue donc le modèle de la vue.

CONTENT VIEW:

```
            /// DISPLAY ////////////////////////////////////////////////////
            DisplayView(calculator: calculator)
            Spacer()
            /// BUTTONS ////////////////////////////////////////////////////
            LazyVGrid(columns: gridColumns, spacing: 20) {
                ForEach(0..<keys.count, id: \.self) { id in
                    Button(action: { self.buttonAction(key: keys[id]) }) {
                        Text(keys[id])
                            .font(.system(size: 30))
                            .foregroundColor(.white).bold()
                    }
                    .frame(width: id == 0 || id == 15 ? 170 : 80, height: 80)
                    .background(colorButton(id: id))
                    .cornerRadius(25)
                    if id == 0  || id == 15 { Color.clear }
                }
```

Fonctionnement général de la calculatrice:

Lorsque l'utilisateur appuie sur une touche, la valeur est ajoutée à un tableau nommé « elements » dans notre modèle Calculator.
Le modèle contient différentes propriétés calculées qui s'assurent que l'expression mathématique reste correcte à chaque saisie de l'utilisateur.
Le calcul du résultat est effectué par un objet de type NSExpression qui permet d'utiliser les éléments String du tableau et de les convertir en calcul mathématique grâce à la méthode expressionValue().

DisplayView:

L'écran dispose d'une simple variable de type String qui récupère le contenu du tableau « elements » et le formate pour l'affichage.
Dans la mesure où NSExpression fonctionne avec des variables de type Double, nous ajoutons .0 à chaque nombre non décimal ajouté au tableau « elements », puis pour un affichage plus agréable, l'écran enlève ces .0.
Si le tableau est vide, l'écran affiche 0 comme une calculatrice classique.

Notes relatives à SwiftUI:

Grille de boutons:

Nous avons utilisé une LazyGrid pour dessiner la grille de boutons. Il s'agit tout d'abord de créer un tableau de GridItem qui va définir le nombre et la taille d'éléments présents dans chaque colonne.

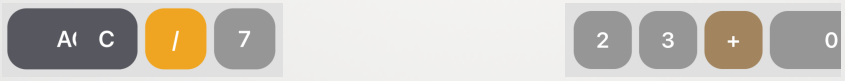
```
let gridColumns = [GridItem(.fixed(80), spacing: 10, alignment: .leading),
                   GridItem(.fixed(80), spacing: 10, alignment: .leading),
                   GridItem(.fixed(80), spacing: 10, alignment: .leading),
                   GridItem(.fixed(80), spacing: 10, alignment: .leading)]
```

Nous fournissons ensuite un tableau d'élements à disposer en grille, dans notre cas, les différentes touches de la calculatrice:

```
var keys = ["AC", "C", "/", "7", "8", "9", "*", "4", "5", "6","-", "1", "2", "3", "+", "0", ".", "="]
```

Puis en bouclant sur le tableau de « keys » avec ForEach, la LazyVGrid va disposer les éléments en grille qui s'étend verticalement en respectant les contraintes demandées pour les colonnes via les GridItem.
Nous avons ensuite demandé une taille plus importante pour les boutons AC et 0 ce qui entraine une superposition de boutons sur les 1 et 4:

```
.frame(width: id == 0 || id == 15 ? 170 : 80, height: 80)
```



Il suffit ensuite d'ajouter 2 items transparents pour respecter la contrainte de 4 éléments par ligne afin d'obtenir la disposition souhaitée, à noter que Color est une View en SwiftUI.

```
if id == 0  || id == 15 { Color.clear }
```

Il y a donc finalement bien 18 boutons dans notre calculatrice, mais 20 items dans notre grille, dont 2 plus larges et 2 transparents.

DarkMode:

La mise en place du darkMode en SwiftUI est très simple, il suffit d'ajouter nos propres couleurs à notre asset catalog:

Il est également possible de détecter si le dark mode est actif sur le Device à l'aide d'une variable d'environmment ce qui permet de programmer d'autres comportements selon le mode actif:

```
@Environment(\.colorScheme) var colorScheme
```

On peut par exemple modifier la couleur d'un texte selon le mode actif:

```
Text(colorScheme == .dark ? "In dark mode" : "In light mode")
```

Previews:

Le Preview dans SwiftUI est défini par une structure qui renvoie une View à laquelle on peut passer des valeurs constantes et ainsi visualiser différentes versions de nos vues simultanément.

Ainsi, nous pouvons par exemple afficher en permanence un preview en mode normal et en dark mode en passant la variable d'environnement colorScheme:

```swift
///////////////////////////////////////////////////////////
// MARK: Previews
///////////////////////////////////////////////////////////
struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView(calculator: Calculator())
            .environment(\.colorScheme, .light)
    }
}



/////////////////// DARK MODE PREVIEW ////////////////////////////////////////////////
struct ContentViewDark_Previews: PreviewProvider {
    static var previews: some View {
        ContentView(calculator: Calculator())
            .environment(\.colorScheme, .dark)
    }
}
```