SwiftUI

Réalisé Par
EL HADDAJ Aya
AKJIJ Salma

Plan

- INTRODUCTION A SWIFTUI
- LES PROPERTY WRAPPERS
- LES FUNCTION BUILDERS
- L'ASYNCHRONE AVEC ASYNC/AWAIT
- LE MAIN ACTOR

Introduction à SwiftUI

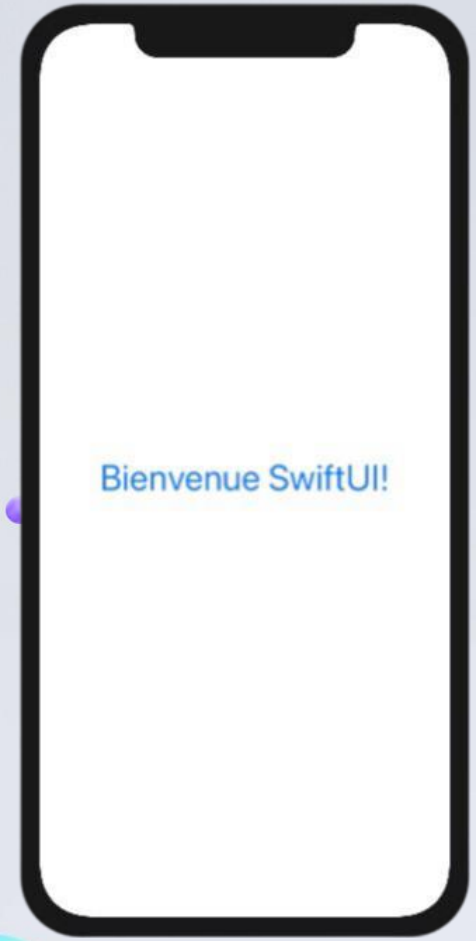
Qu'est-ce que SwiftUI ?

- SwiftUI est un framework déclaratif introduit par Apple en 2019.
- Vise a créer des interfaces utilisateur de manière plus intuitive et efficace.
- SwiftUI permet de décrire ce que l'interface doit faire
- Fonctionne sur toutes les plateformes Apple.
- S'intègre pleinement avec Swift.

Pourquoi SwiftUI ?

- **Simplification** : SwiftUI réduit la quantité de code nécessaire pour créer des interfaces utilisateur complexes.
- **Approche déclarative** : Au lieu de dire **comment** l'interface doit se comporter (comme avec UIKit), tu décris **ce qu'elle doit faire**.
- **Multiplateforme** : SwiftUI est conçu pour fonctionner sur toutes les plateformes Apple (iOS, iPadOS, macOS, watchOS, tvOS, et visionOS) avec un seul codebase.
- **Intégration avec Swift** : SwiftUI tire pleinement parti des fonctionnalités modernes de Swift, comme les **Property Wrappers** et les **Opaque Return Types**.

Exemple de Code



Ici, SwiftUI permet de créer une interface simple avec seulement quelques lignes de code. La vue est automatiquement mise à jour lorsque l'état change.

```swift
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Bienvenue SwiftUI!")
        .font(.largeTitle)
        .foregroundColor(.blue)
        .padding()
    }
}
```

Différence entre SwiftUI et UIKit

Aspect	SwiftUI	UIKit
Approche	Déclarative : Tu décris **ce que l'interface doit faire.**	Impérative : Tu dis **comment l'interface doit se comporter.**
Code	Moins de code, plus lisible.	Plus de code, souvent verbeux.
Gestion de l'état	Automatique avec **@State**, **@Binding**, etc.	Manuel avec des méthodes comme setNeedsLayout() ou setNeedsDisplay().

Différence entre SwiftUI et UIKit

Aspect	SwiftUI	UIKit
Multiplateforme	**Un seul codebase** pour iOS, iPadOS, macOS, watchOS, tvOS, et visionOS.	Principalement conçu pour iOS.
Intégration Swift	Utilise les fonctionnalités modernes de Swift (**Property Wrappers**, etc.).	Basé sur Objective-C, moins intégré avec les fonctionnalités modernes.

Exemple de Comparaison

- Ici avec UIKit

```swift
// UIKit
let label = UILabel()
label.text = "Hello, UIKit!"
label.font = UIFont.systemFont(ofSize: 20)
label.textColor = UIColor.black

```

- Ici avec SwiftUI

```swift
// SwiftUI
Text("Hello, SwiftUI!")
    .font(.system(size: 20))
    .foregroundColor(.black)


```

SwiftUI et les Plateformes Apple

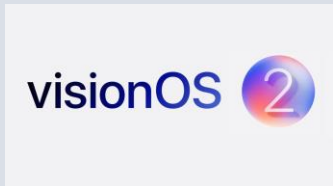
SwiftUI et les Plateformes Apple

SwiftUI est conçu pour être universel.

On peux utiliser le **même code** pour développer des applications sur toutes les plateformes Apple.

Cela réduit le temps de développement et facilite la maintenance.

- iOS & iPadOS
- macOS
- watchOS
- tvOS
- visionOS



Fonctionnalités Utiles de Swift

Les Opaques Return Types

- Les **Opaque Return Types** (types de retour opaques) permettent de retourner un type spécifique **sans révéler ce type à l'extérieur** de la fonction. Cela est particulièrement utile dans SwiftUI pour retourner des vues sans avoir à spécifier exactement quel type de vue est retourné. Le mot-clé **some** est utilisé pour indiquer un type de retour opaque.

Exemple de Code

```swift
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
            .padding()
    }
}
```

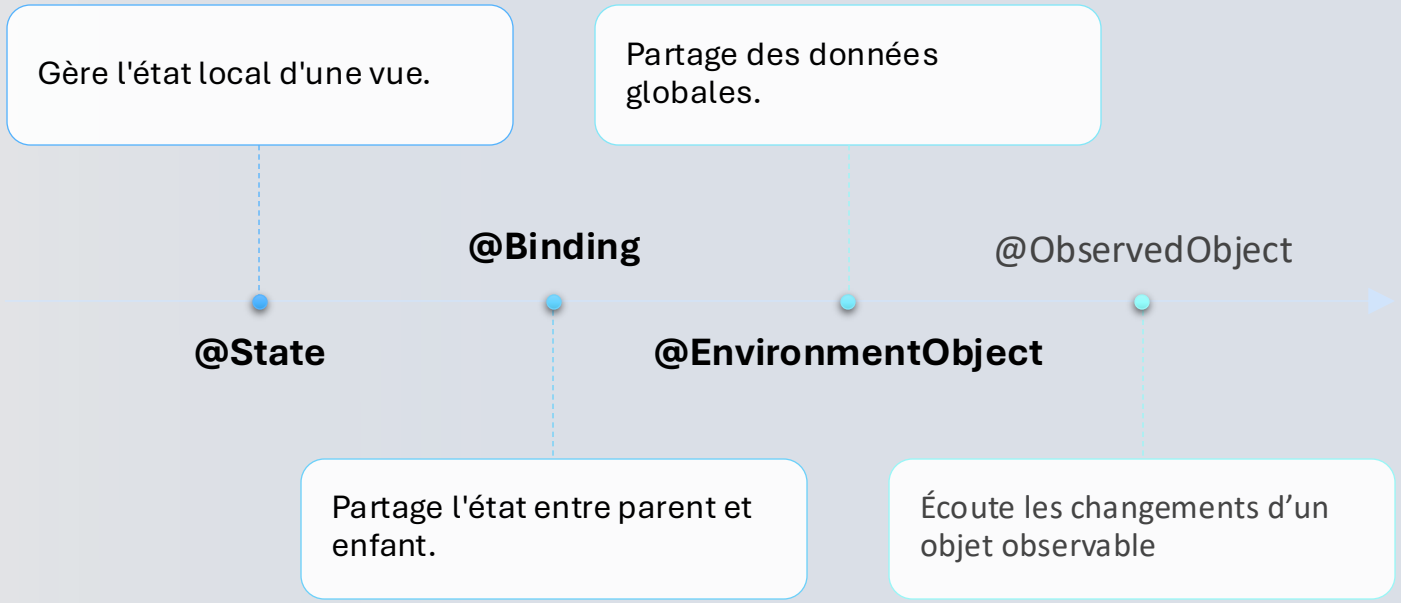
- Ici, body retourne un type opaque (**some View**), ce qui signifie que le type exact de la vue est caché, mais on sait qu'il s'agit d'une vue conforme au protocole **View**.

Les Property Wrappers

Les Property Wrappers

(Enveloppeurs de propriétés)

- Ceux sont une fonctionnalité de Swift qui permet de définir un comportement spécifique pour une propriété.
- Dans SwiftUI, ils sont largement utilisés pour gérer l'état et les données. Voici les principaux property wrappers



@State

- **@State** est utilisé pour gérer l'état local d'une vue.
- Lorsque la valeur d'une propriété **@State** change, la vue est automatiquement mise à jour.
- C'est idéal pour stocker des données simples qui ne sont utilisées que dans une seule vue.

Exemple de Code

Ici, **@State** est utilisé pour gérer l'état de **isOn**. Lorsque l'utilisateur interagit avec le **Toggle**, la vue est automatiquement mise à jour pour afficher "Activé" ou "Désactivé".

```swift
import SwiftUI

struct ContentView: View {
    @State private var isOn: Bool = false

    var body: some View {
        VStack {
            Toggle("Activer", isOn: $isOn)
                .padding()

            Text(isOn ? "Activé" : "Désactivé")
                .foregroundColor(isOn ? .green : .red)
                .font(.title)
                .padding()
        }
    }
}
```

@Binding

- **@Binding** permet de partager l'état entre une vue parent et une vue enfant.
- La vue enfant peut modifier la valeur, mais la source de vérité reste dans la vue parent.

Exemple de Code

```swift
import SwiftUI

struct ChildView: View {
    @Binding var count: Int

    var body: some View {
        Button("Incrémenter") {
            count += 1
        }
    }
}

struct ContentView: View {
    @State private var count = 0

    var body: some View {
        VStack {
            Text("Valeur : \(count)")
            ChildView(count: $count) // Liaison via @Binding
        }
    }
}
```

- **@State** dans **ContentView** garde la valeur centrale de **count.**
- **@Binding** dans **ChildView** permet de modifier count depuis le bouton enfant.
- Toute modification dans **ChildView** met à jour automatiquement **ContentView**.

@ObservedObject

- **@ObservedObject** permet à une vue d'écouter les changements d'un objet observable.
- Cet objet doit adopter **ObservableObject** et marquer ses propriétés avec **@Published** pour déclencher des mises à jour automatiques de l'interface.

Exemple de Code

```swift
import SwiftUI

class Counter: ObservableObject {
    @Published var value: Int = 0
}

struct ContentView: View {
    @ObservedObject var counter = Counter()

    var body: some View {
        VStack {
            Text("Valeur : \(counter.value)")
            Button("Incrémenter") {
                counter.value += 1
            }
        }
    }
}
```

- **Counter** est un objet observable avec une propriété value marquée **@Published** (qui permet de notifier les changements de value).
- **ContentView** écoute les changements de **Counter** grâce à **@ObservedObject**.
- Lorsque le bouton est cliqué, **counter.value** est mis à jour, et la vue se rafraîchit automatiquement.

@EnvironmentObject

- **@EnvironmentObject** permet de partager des données entre plusieurs vues sans avoir à les passer manuellement.
- Il est pratique pour gérer des paramètres globaux comme le mode sombre ou les préférences utilisateur.

Exemple de Code

```swift
import SwiftUI

class Settings: ObservableObject {
    @Published var isDarkMode = false
}

struct ContentView: View {
    @EnvironmentObject var settings: Settings

    var body: some View {
        VStack {
            Text(settings.isDarkMode ? "🌙 Sombre" : "🌞 Clair")
            Button("Changer") { settings.isDarkMode.toggle() }
        }
    }
}

@main
struct MyApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView().environmentObject(Settings())
        }
    }
}
```

- **Settings** stocke un état global (**isDarkMode**).
- **@EnvironmentObject** permet d'accéder à **Settings** sans le passer en paramètre.
- **.environmentObject(Settings())** rend **Settings** accessible partout.
- **ContentView** affiche "Sombre" ou " Clair" selon **isDarkMode**.
- Le bouton "**Changer**" bascule le mode et met à jour l'affichage.

Résumé des Property Wrappers

Property Wrappers	Utilisation
@State	Pour gérer l'état local d'une vue
@Binding	Pour partager l'état entre une vue parent et une vue enfant
@ObservedObject	Pour écouter les changements d'un objet observable
@EnvironmentObject	Pour partager des données globales dans toute l'application

Les Function Builders

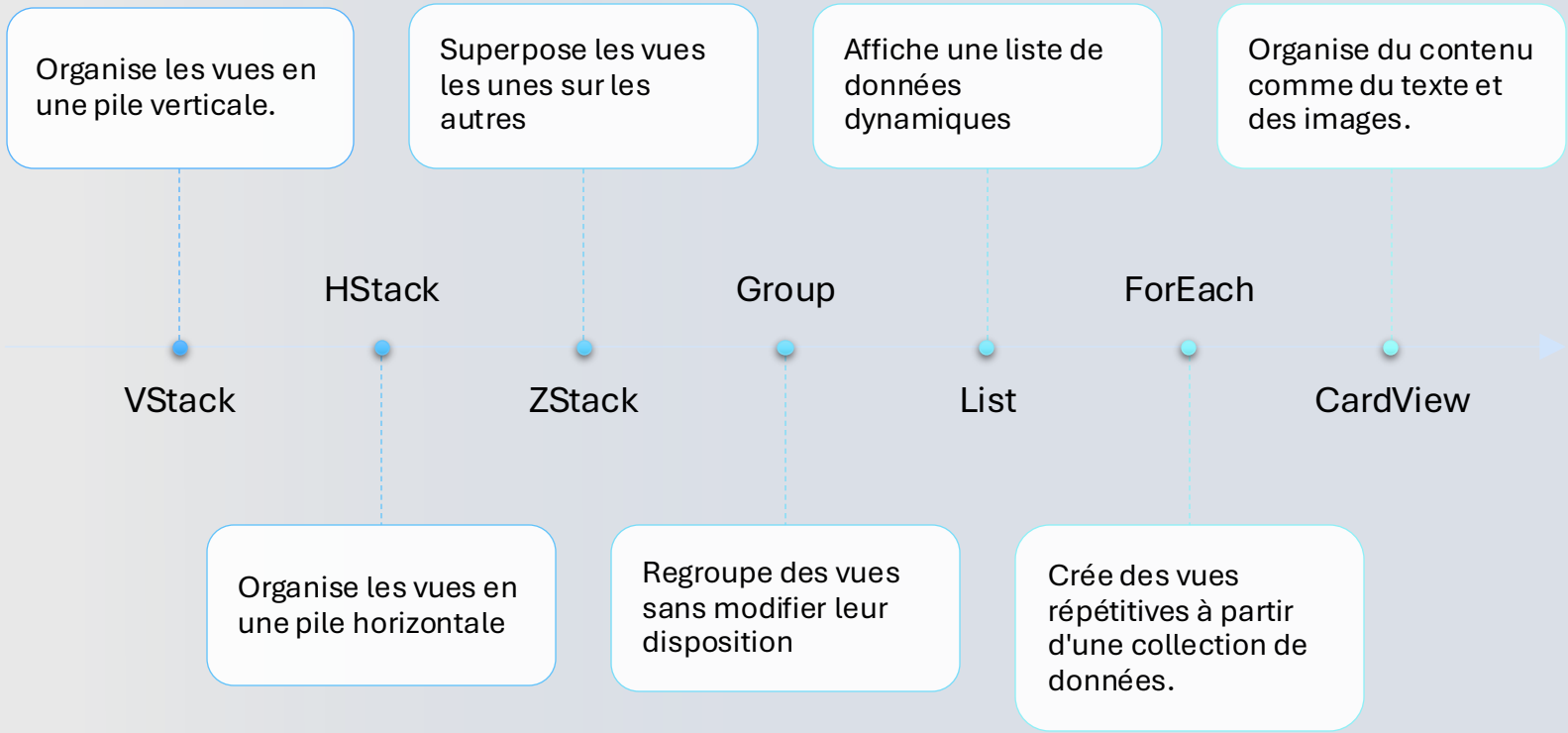
Les Function Builders

(Constructeurs de fonctions)

Les **Function Builders** (constructeurs de fonctions) sont une fonctionnalité de Swift qui permet de construire des structures de données complexes de manière **déclarative** et **intuitive**. Ils sont principalement utilisés dans SwiftUI pour organiser les vues en piles (**VStack**, **HStack**, **ZStack**), regrouper des vues (**Group**), ou créer des listes (**List**).

Les Function Builders

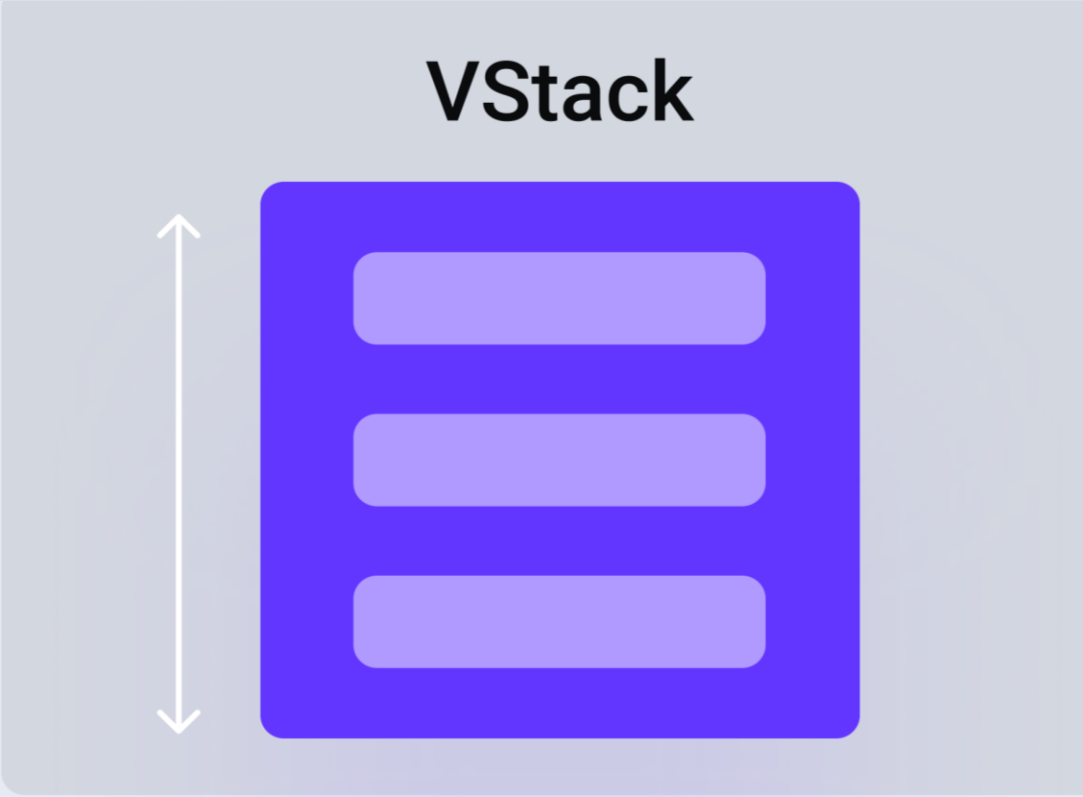
(Constructeurs de fonctions)



VStack (Vertical Stack)

VStack organise les vues en une pile verticale.

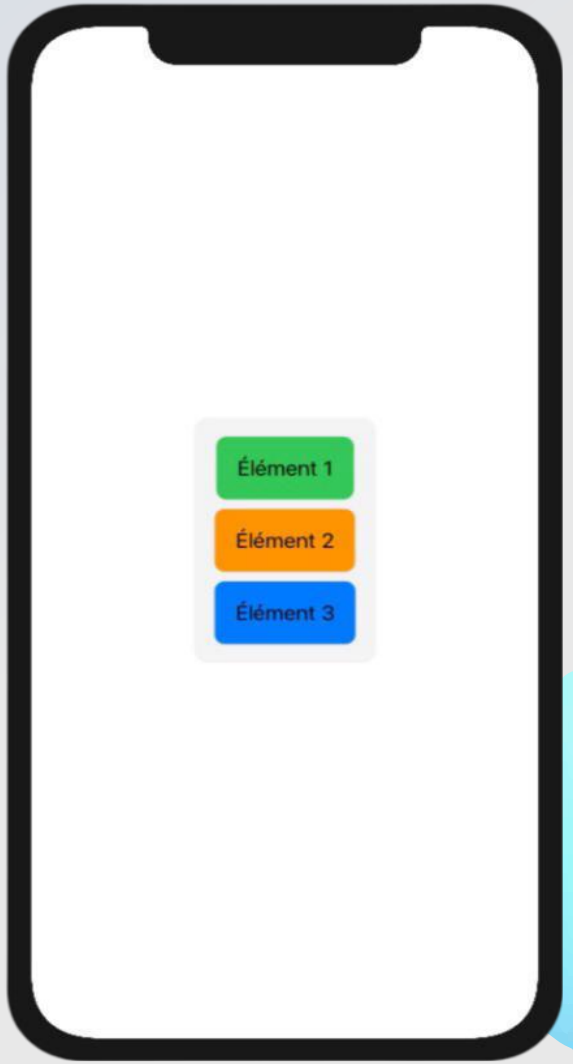
Les vues sont disposées de haut en bas.



Exemple de Code

Ici, **VStack** utilise un function builder pour organiser les trois **Text** en une pile verticale.

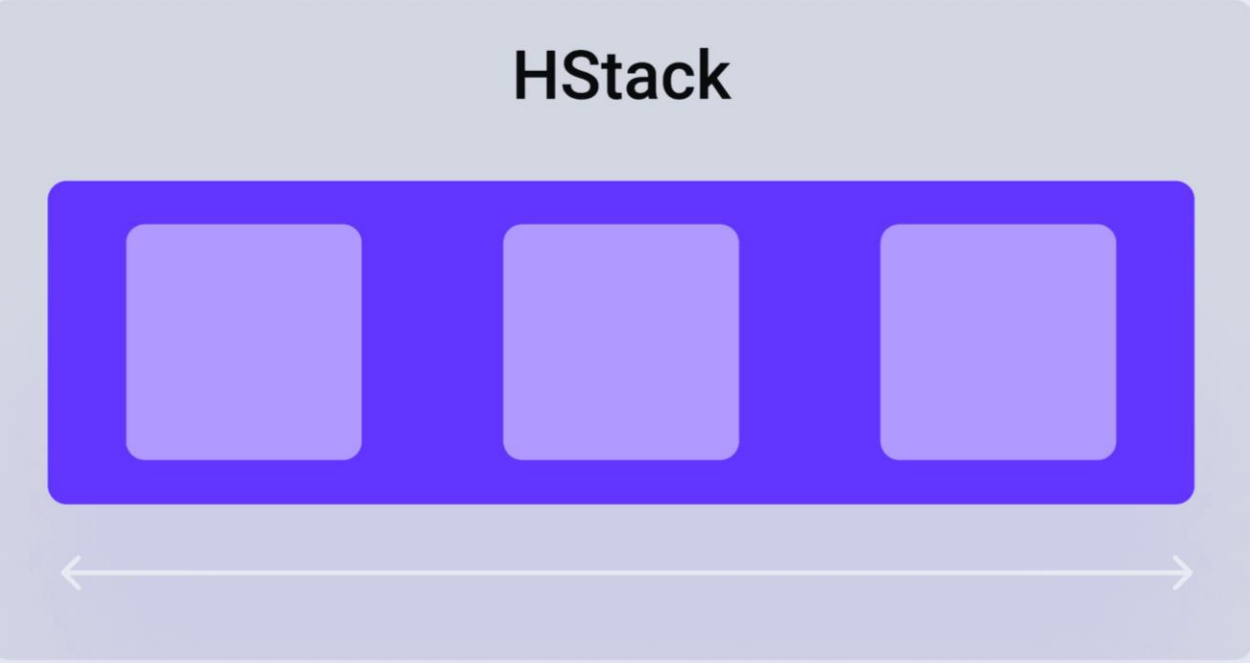
```swift
import SwiftUI

struct ContentView: View {
    @State private var isOn: Bool = false

    var body: some View {
        VStack {
            Text("Élément 1")
                .padding()
                .background(Color.green)
            Text("Élément 2")
                .padding()
                .background(Color.orange)
            Text("Élément 3")
                .padding()
                .background(Color.blue)
        }
        .padding()
        .background(Color.gray.opacity(0.1)) // Fond de la VStack
    }
}
```



HStack (Horizontal Stack)

HStack organise les vues en une pile horizontale.

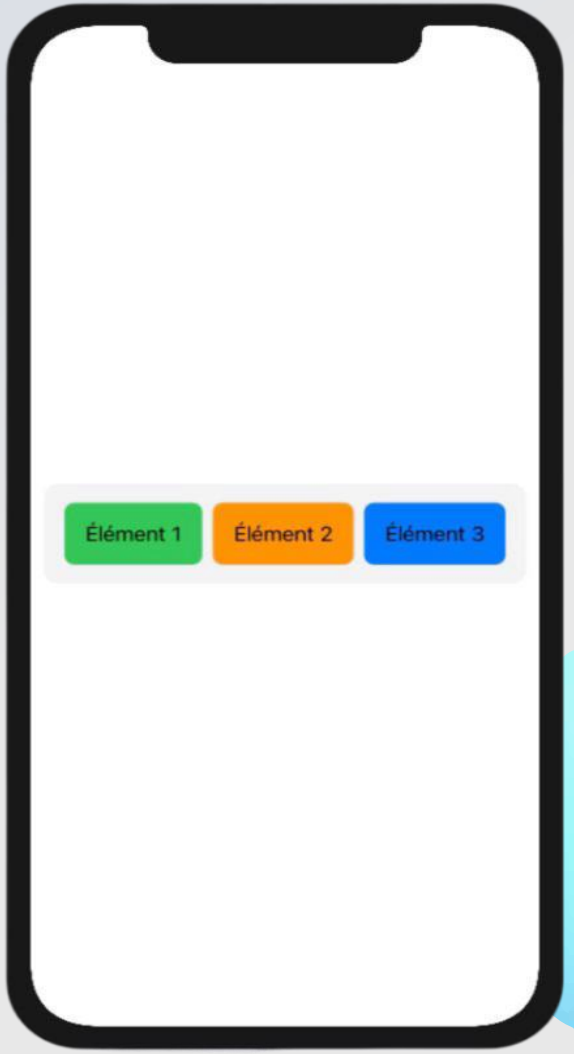
Les vues sont disposées de gauche à droite.



Exemple de Code

Ici, **HStack** utilise un function builder pour organiser les trois **Text** en une pile horizontale.

```swift
import SwiftUI

struct ContentView: View {
    @State private var isOn: Bool = false

    var body: some View {
        HStack {
            Text("Élément 1")
                .padding()
                .background(Color.green)
            Text("Élément 2")
                .padding()
                .background(Color.orange)
            Text("Élément 3")
                .padding()
                .background(Color.blue)
        }
        .padding()
        .background(Color.gray.opacity(0.1)) // Fond de la HStack
    }
}
```



ZStack (Z-axis Stack)

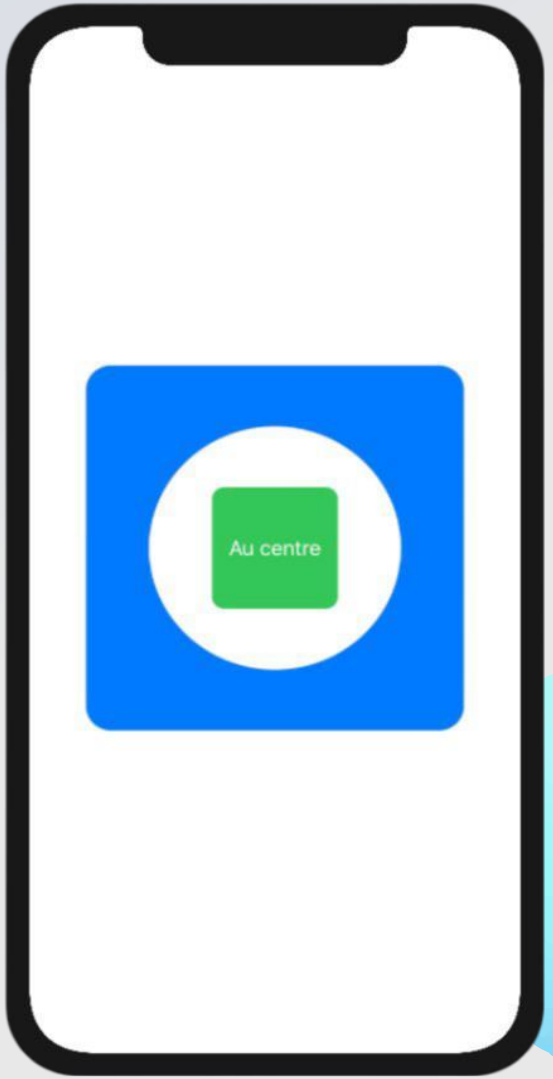
ZStack superpose les vues les unes sur les autres, en les alignant sur l'axe Z (profondeur).

La première vue est au **fond**, et la dernière vue est au **premier plan**.

Exemple de Code

Ici, **ZStack** utilise un function builder pour superposer un **Text**, un **Rectangle**, un **Circle** et un **Rectangle**.

```swift
import SwiftUI

struct ContentView: View {
    @State private var isOn: Bool = false

    var body: some View {
        ZStack {
            // Un carré de fond
            Rectangle()
                .fill(Color.blue)
                .frame(width: 300, height: 300)

            // Un cercle à l'intérieur du carré
            Circle()
                .fill(Color.white)
                .frame(width: 200, height: 200)

            // Un autre rectangle plus petit à l'intérieur du cercle
            Rectangle()
                .fill(Color.green)
                .frame(width: 100, height: 100)

            // Un texte au centre du plus petit rectangle
            Text("Au centre")
                .font(.largeTitle)
                .foregroundColor(.white)
        }
    }
}
```



Group

Group en SwiftUI est un conteneur de mise en page qui permet de regrouper plusieurs vues sans modifier leur disposition ou leur comportement dans la hiérarchie. Il est utile pour **appliquer des modifications globales** (comme un style) à un ensemble d'éléments sans créer de nouveaux conteneurs visibles, contrairement à **VStack** ou **HStack**.

Exemple de Code

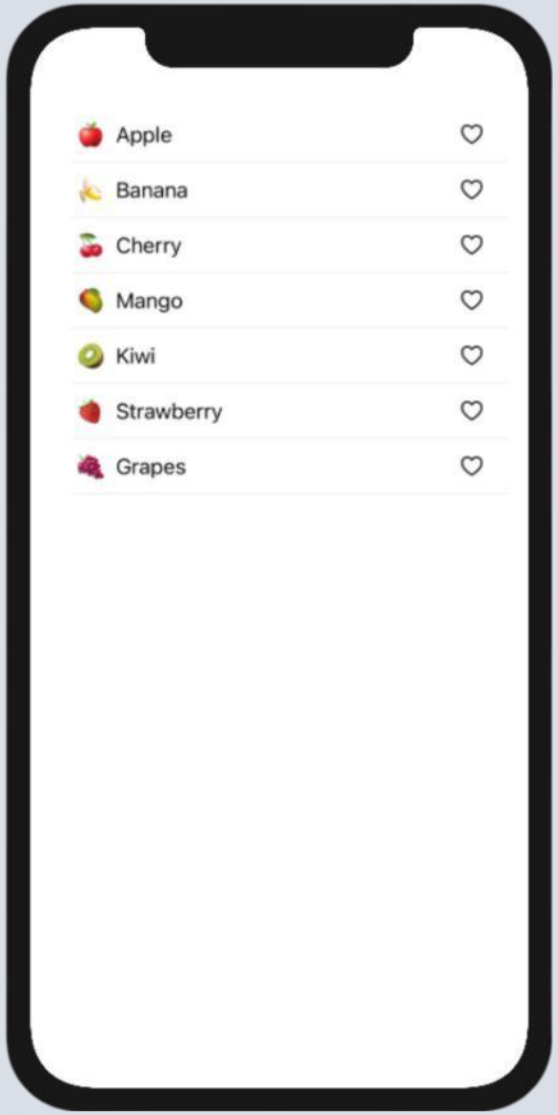
Ici, **Group** est utilisé pour regrouper plusieurs vues (**Tex**t) et leur appliquer des modificateurs communs (comme **.font** et **.foregroundColor**) sans avoir à répéter ces modificateurs pour chaque vue individuellement.

```swift
import SwiftUI

struct ContentView: View {
    var body: some View {
        VStack {
            Group {
                Text("Ligne 1")
                Text("Ligne 2")
                Text("Ligne 3")
            }
            .font(.headline)
            .foregroundColor(.blue)

            Group {
                Text("Ligne 4")
                Text("Ligne 5")
                Text("Ligne 6")
            }
            . font(.subheadline)
            .foregroundColor(.green)

        }
    }
}
```

List

List est utilisé pour afficher une liste de vues.

Elle est souvent utilisée pour afficher des données dynamiques.



Exemple de Code

- Ici, **List** utilise un function builder pour créer une liste dynamique à partir d'un tableau de tuple (String, String).

```swift
import SwiftUI

struct ContentView: View {
    let fruits = [
        ("🍎", "Apple"),
        ("🍌", "Banana"),
        ("🍒", "Cherry"),
        ("🍋", "Mango"),
        ("🥝", "Kiwi"),
        ("🍓", "Strawberry"),
        ("🍇", "Grapes")
    ]

    var body: some View {
        List(fruits, id: \.1) { fruit in
            HStack {
                Text(fruit.0) // Emoji
                Text(fruit.1) // Nom du fruit
            }
        }
    }
}
```

ForEach

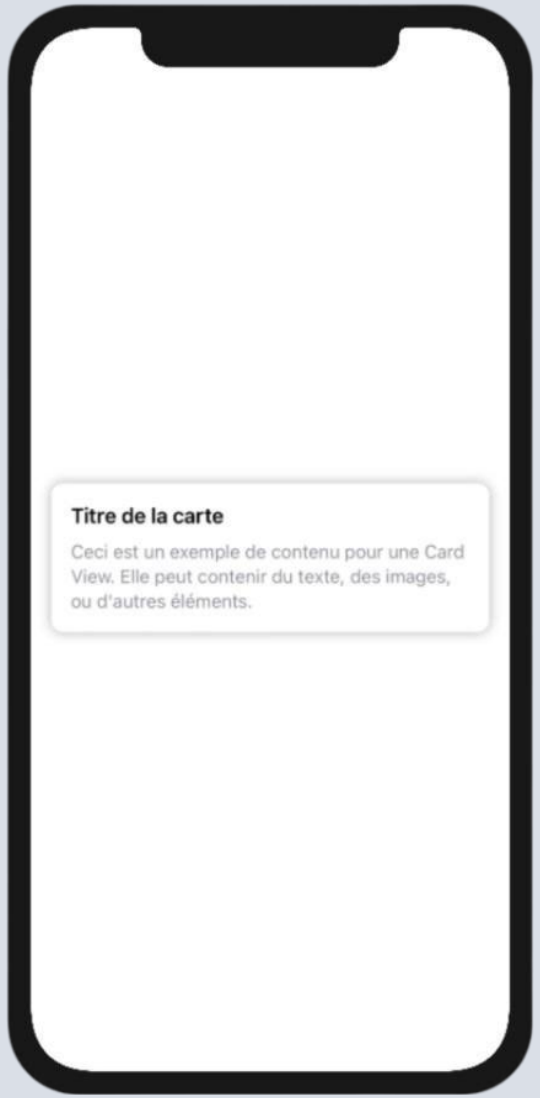
ForEach est utilisé pour créer des vues répétitives à partir d'une collection de données. Il est souvent utilisé à l'intérieur d'autres conteneurs
comme **VStack**, **HStack**, ou **List**.

- Ici, **ForEach** est utilisé pour créer une **VStack** contenant un **Text** pour chaque élément du tableau items.

```swift
import SwiftUI

struct ContentView: View {
    let items = ["Pomme", "Banane", "Orange"]

    var body: some View {
        VStack {
            ForEach(items, id: \.self) { item in
                Text(item)
            }
        }
        .padding()
    }
}
```

CardView

Une **Card View** est un conteneur visuel qui regroupe des informations connexes (comme du texte, des images, ou des boutons) dans un cadre stylisé. Elle est souvent utilisée pour afficher des éléments de manière organisée, comme dans une liste ou une grille.



Exemple de Code

- Ce code définit une **Card View**, affichant un titre et une description stylisés avec des coins arrondis et une ombre. **ContentView** affiche simplement cette carte.

```swift
import SwiftUI

struct CardView: View {
    var body: some View {
        VStack(alignment: .leading, spacing: 10) {
            Text("Titre de la carte")
                .font(.headline)

            Text("Ceci est un exemple de contenu pour une Card View.
                Elle peut contenir du texte, des images,
                ou d'autres elements.")
                .font(.subheadline)
                .foregroundColor(.gray)

        }
        .padding()
        .background(Color.white)
        .cornerRadius(10)
        .shadow(radius: 5)
    }
}
```

Résumé des Function Builders

Property Wrappers	Utilisation
VStack	Organise les vues en une pile verticale.
HStack	Organise les vues en une pile horizontale.
ZStack	Superpose les vues les unes sur les autres.
Group	Regroupe des vues sans les organiser dans une pile spécifique

Résumé des Function Builders

Property Wrappers	Utilisation
List	Affiche une liste de vues, souvent utilisée pour des données dynamiques
CardView	Conteneur visuel stylisé pour regrouper des informations connexes (texte, images, boutons).
ForEach	Crée des vues répétitives à partir d'une collection de données.

L'Asynchrone avec Async / Await

L'Asynchrone avec Async / Await

- L'asynchrone avec async et await est une fonctionnalité introduite dans Swift 5.5 pour simplifier la gestion des tâches asynchrones.
- Cela permet d'écrire du code asynchrone de manière plus lisible et moins sujette aux erreurs.
- Au lieu d'utiliser des closures ou des callbacks, tu peux simplement utiliser await pour attendre le résultat d'une tâche asynchrone.

Exemple de code

```swift
import SwiftUI

struct ContentView: View {
    @State private var data: String = "Chargement..."

    var body: some View {
        Text(data)
            .task {
                await loadData()
            }
    }

    func loadData() async {
        // Simule une tâche asynchrone (par exemple, une requête réseau)
        try? await Task.sleep(nanoseconds: 2_000_000_000) // 2 secondes
        data = "Données chargées !"
    }
}

```

- Ici, **async** et **await** sont utilisées pour exécuter une tâche asynchrone simulée avec **Task.sleep()**, retardant l'affichage des données pendant 2 secondes avant de mettre à jour l'interface avec "Données chargées !".

Le MainActor

Le MainActor

Le **MainActor** est un concept introduit pour garantir que certaines tâches sont exécutées sur le **thread principal**. C'est particulièrement important pour les mises à jour de l'interface utilisateur, qui doivent toujours se faire sur le thread principal. Le mot-clé **@MainActor** peut être utilisé pour marquer une fonction ou une classe qui doit s'exécuter sur le thread principal.

Exemple de Code

```swift
import SwiftUI

@MainActor
class ViewModel: ObservableObject {
    @Published var text: String = "Bonjour"

    func updateText() {
        text = "Texte mis à jour !"
    }
}

struct ContentView: View {
    @StateObject private var viewModel = ViewModel()

    var body: some View {
        VStack {
            Text(viewModel.text)
            Button("Mettre à jour") {
                viewModel.updateText()
            }
        }
        .padding()
    }
}
```

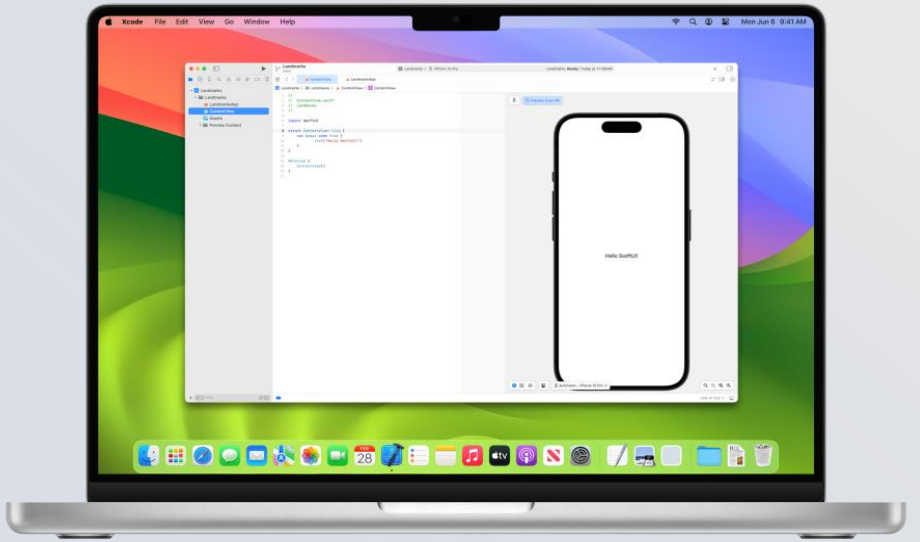
- Ici, **@MainActor** assure que **ViewModel** met à jour l'UI sur le thread principal. **ViewModel** utilise **@Published** pour réagir aux changements de texte, et **ContentView** gère **ViewModel** via **@StateObject**, affichant un texte réactif et un bouton permettant de le modifier.

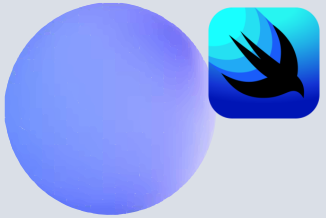
Tutoriel Recommandé

Pour Aller Plus Loin

Pour approfondir votre compréhension, consultez le tutoriel officiel d'Apple :

Créer et Combiner des Vues





Conclusion Générale

SwiftUI est un framework moderne et intuitif qui simplifie le développement d'interfaces utilisateur sur les plateformes Apple. Grâce à son approche déclarative, il réduit la quantité de code nécessaire et permet une gestion efficace de l'état avec des outils comme **@State**, **@Binding**, et **async/await**. Multiplateforme, il permet de créer des applications pour iOS, iPadOS, macOS, watchOS, tvOS, et visionOS avec un seul codebase. Que ce soit pour des composants réutilisables comme les **Card Views** ou des listes dynamiques avec **List**, SwiftUI offre une solution puissante et flexible pour les développeurs.

Ressources & Réferences

Ressources & Références

- *https://blog.ippon.fr/2022/02/02/introduction-a-swiftui/*
- *https://developer.apple.com/tutorials/swiftui/creating-and-combining-views*
- *https://www.appcoda.com/learnswiftui/swiftui-basics.html*

ippon POSITIVE TECHNOLOGY

Developer

APPCODA

Merci