SwiftUI &UIKit

Integration

Hubert Kiełkowski
iOS developer

What is a Coordinator?

In simple words, a Coordinator is like a middleman that helps SwiftUI and UIKit talk to each other. When we use UIKit components in SwiftUI, the Coordinator handles the communication between the two frameworks, managing events and updates.

What is the Delegate Pattern?

The delegate pattern is a way for one object to communicate with another object when certain events happen. Think of it like having an assistant (the delegate) who takes care of specific tasks for you (the main object). In our case, UIKit components often use delegates to handle user interactions and updates, and the Coordinator acts as this assistant, passing information back to SwiftUI.

Hubert Kiełkowski
iOS developer

How UIViewRepresentable Works

1. **Purpose:**

   UIViewRepresentable is a protocol in SwiftUI that allows you to integrate UIKit components (which are based on UIView from UIKit) into SwiftUI views.

2. **Wrapper for UIKit Views:**

   When you create a struct that conforms to UIViewRepresentable, you're essentially creating a bridge between SwiftUI and UIKit. SwiftUI doesn't directly support UIKit views, so this protocol acts as a translator.

3. **Two Main Methods:**

   makeUIView(context:): This method is where you create and configure your UIKit view (UIView, UIButton, UITextField, etc.). It's like setting up the view in UIKit's world.
   updateUIView(_:context:): This method is called whenever SwiftUI needs to update the UIKit view based on changes in SwiftUI's state or data bindings. It's how SwiftUI keeps the UIKit view in sync with its own state.

4. **State Management:**

   You can use SwiftUI's @State, @Binding, or other state management techniques to pass data back and forth between SwiftUI and your UIKit view. For example, updating a label's text in a UILabel from a SwiftUI @State variable.

5. **Coordinator (Optional):**

   Sometimes you need to handle interactions or delegate methods of UIKit views. You use a nested Coordinator class inside your UIViewRepresentable struct to manage these interactions. The coordinator acts as a delegate or data source for the UIKit view.

6. **Integration with SwiftUI Views:**

   Once you've defined your UIViewRepresentable struct, you can use it just like any other SwiftUI view in your SwiftUI hierarchy. SwiftUI handles the layout and positioning of your UIKit-based view seamlessly.

Hubert Kiełkowski
iOS developer

```swift
import SwiftUI
import UIKit

// Step 1: Create a struct that conforms to UIViewRepresentable
struct ActivityIndicatorView: UIViewRepresentable {
    // Step 2: Define the Coordinator to handle interactions
    //(optional in this case)
    class Coordinator: NSObject {
        var parent: ActivityIndicatorView

        init(parent: ActivityIndicatorView) {
            self.parent = parent
        }

        // No additional methods needed in this example
    }

    // Step 3: Implement required methods

    // 3.1: Create and configure the UIActivityIndicatorView
    func makeUIView(context: Context) -> UIActivityIndicatorView {
        let indicatorView = UIActivityIndicatorView(style: .large)
        indicatorView.color = .blue
        indicatorView.startAnimating() // Start animating when view is created
        return indicatorView
    }

    // 3.2: Update the UIActivityIndicatorView (if needed)
    func updateUIView(_ uiView: UIActivityIndicatorView, context: Context) {
        // No update needed in this example,
        //since we don't change the indicator state dynamically
    }

    // Step 4: Optionally implement a Coordinator if needed for interaction handling
    func makeCoordinator() -> Coordinator {
        return Coordinator(parent: self)
    }
}

// Step 5: Use the ActivityIndicatorView in a SwiftUI view
struct ContentExampleView: View {
    @State private var isLoading = false

    var body: some View {
        VStack {
            ActivityIndicatorView()
                .frame(width: 50, height: 50)

            Button(action: {
                isLoading.toggle()
            }) {
                Text(isLoading ? "Stop Loading" : "Start Loading")
                    .padding()
                    .foregroundColor(.white)
                    .background(Color.blue)
                    .cornerRadius(8)
            }
        }
        .padding()
    }
}
```

Let's explore a more interesting example using UIViewRepresentable to wrap a UIActivityIndicatorView in SwiftUI. This component will show how to integrate a UIKit-based activity indicator, which is commonly used for indicating progress or loading states, into a SwiftUI view hierarchy.

Hubert Kiełkowski
iOS developer

Explanation:

1. **ActivityIndicatorView (UIViewRepresentable):**
   - **Purpose:** Wraps a UIActivityIndicatorView (UIKit) to show a loading indicator in SwiftUI.
   - **Steps:**
     - **Step 1:** Define ActivityIndicatorView struct conforming to UIViewRepresentable.
     - **Step 2:** Optionally define a Coordinator to manage interactions (not used in this example).
     - **Step 3:** Implement makeUIView to create and configure the UIActivityIndicatorView.
     - **Step 4:** Implement updateUIView to update the view based on SwiftUI state (not needed in this example).
     - **Step 5:** Optionally implement makeCoordinator if interaction handling is required.
2. **ContentView (SwiftUI View):**
   - **Purpose:** Demonstrates the usage of ActivityIndicatorView within a SwiftUI view hierarchy.
   - **Usage:**
     - Includes an ActivityIndicatorView to show loading state.
     - A Button toggles the loading state (isLoading) to start or stop the activity indicator.

Summary:

This example showcases how UIViewRepresentable allows seamless integration of UIKit's UIActivityIndicatorViewinto SwiftUI. It demonstrates starting and stopping the activity indicator based on SwiftUI's state (isLoading). This approach enables using UIKit's specialized UI components within SwiftUI applications for enhanced customization and functionality not available natively in SwiftUI.

Hubert Kiełkowski
iOS developer

Custom Slider Integration

1. Define the Custom Slider Struct:

```swift
struct CustomSlider: UIViewRepresentable {
    @Binding var value: Double
    var range: ClosedRange<Double>
    var step: Double

    func makeUIView(context: Context) -> UISlider {
        let slider = UISlider(frame: .zero)
        slider.minimumValue = Float(range.lowerBound)
        slider.maximumValue = Float(range.upperBound)

        // Custom thumb image
        let thumbImage = UIImage(systemName: "circle.fill")
        slider.setThumbImage(thumbImage, for: .normal)

        // Custom track images
        slider.setMinimumTrackImage(UIImage(named: "minTrack"), for: .normal)
        slider.setMaximumTrackImage(UIImage(named: "maxTrack"), for: .normal)

        slider.addTarget(context.coordinator,
                         action: #selector(Coordinator.valueChanged(_:)),
                         for: .valueChanged)
        return slider
    }

    func updateUIView(_ uiView: UISlider, context: Context) {
        uiView.value = Float(value)
    }

    func makeCoordinator() -> Coordinator {
        Coordinator(self)
    }

    class Coordinator: NSObject {
        var parent: CustomSlider

        init(_ parent: CustomSlider) {
            self.parent = parent
        }

        @objc func valueChanged(_ sender: UISlider) {
            parent.value = Double(sender.value)
        }
    }
}
```

Hubert Kiełkowski
iOS developer

2. Using the Custom Slider in SwiftUI:

```swift
struct SliderView: View {
    @State private var sliderValue: Double = 0.5

    var body: some View {
        VStack {
            CustomSlider(value: $sliderValue, range: 0...1, step: 0.01)
                .padding()

            Text("Slider Value: \(sliderValue)")
        }
    }
}
```

(CustomSlider vs Slider)

- **Advanced Customization:** CustomSlider allows for extensive customization such as custom thumb images and track visuals, which is more challenging to achieve with SwiftUI's Slider.
- **Complex Interactions:** It supports complex behaviors like multi-step sliders and custom animations directly through UIKit's delegate methods, offering more flexibility compared to SwiftUI's simpler interaction patterns.
- **Legacy Code Integration:** Ideal for integrating existing UIKit slider implementations seamlessly into SwiftUI projects, preserving legacy functionalities without complete reimplementation

Hubert Kiełkowski
iOS developer

Custom Image Integration

1. Define the Custom ImageView Struct:

```
struct CustomImageView: UIViewRepresentable {
    @Binding var image: UIImage

    func makeUIView(context: Context) -> UIImageView {
        return UIImageView()
    }

    func updateUIView(_ uiView: UIImageView, context: Context) {
        uiView.image = image
    }
}
```

2. Custom Actions for ImageView:

```
extension UIImage {
    func cropped(to rect: CGRect) -> UIImage? {
        guard let cgImage = self.cgImage?.cropping(to: rect) else { return nil }
        return UIImage(cgImage: cgImage)
    }

    func applyingGrayscale() -> UIImage? {
        let context = CIContext()
        guard let currentFilter = CIFilter(name: "CIPhotoEffectMono") else {
            return nil
        }
        currentFilter.setValue(CIImage(image: self), forKey: kCIInputImageKey)
        guard let output = currentFilter.outputImage else { return nil }
        guard let cgImage = context.createCGImage(output, from: output.extent) else {
            return nil
        }
        return UIImage(cgImage: cgImage)
    }
}
```

Hubert Kiełkowski
iOS developer

3. Using the Custom ImageView in SwiftUI:

```swift
struct ImageViewContainer: View {
    @State var image = UIImage(named: "example")!

    var body: some View {
        VStack {
            CustomImageView(image: $image)
                .frame(width: 200, height: 200)
                .border(Color.black, width: 1)

            HStack {
                Button(action: cropImage) {
                    Text("Crop")
                }
                Button(action: applyGrayscale) {
                    Text("Grayscale")
                }
            }
        }
    }

    func cropImage() {
        let rect = CGRect(x: 0, y: 0, width: 100, height: 100)
        if let croppedImage = image.cropped(to: rect) {
            image = croppedImage
        }
    }

    func applyGrayscale() {
        if let grayscaleImage = image.applyingGrayscale() {
            image = grayscaleImage
        }
    }
}
```

Custom ImageView (CustomImageView vs Image)

- **Customized Appearance:** CustomImageView offers advanced customization options such as custom overlays, blend modes, and direct manipulation of image data, features not as readily accessible with SwiftUI's Image.
- **Advanced Image Manipulation:** Enables operations like cropping, applying filters, and more sophisticated image effects using UIKit's Core Image integration, which may require more effort to achieve in SwiftUI.

Hubert Kiełkowski
iOS developer

Custom UITextView Integration

1. Define the Custom TextView Struct:

```
struct RichTextView: UIViewRepresentable {
    @Binding var content: NSMutableAttributedString

    func makeUIView(context: Context) -> UITextView {
        return UITextView()
    }

    func updateUIView(_ uiView: UITextView, context: Context) {
        uiView.attributedText = content
    }
}
```

2. Using the Custom Text View in SwiftUI:

```
struct MainTextView: View {
    @State var content = NSMutableAttributedString(string: "")

    var body: some View {
        VStack {
            RichTextView(content: $content)
                .frame(minWidth: 0, maxWidth: .infinity, minHeight: 0, maxHeight: .infinity)

            HStack {
                Button(action: applyBold) {
                    Text("Bold")
                }
                Button(action: applyItalic) {
                    Text("Italic")
                }
            }
            .padding()
        }
    }

    func applyBold() {
        let boldAttribute = [NSAttributedString.Key.font: UIFont.boldSystemFont(ofSize: 14)]
        content.addAttributes(boldAttribute, range: selectedRange())
    }

    func applyItalic() {
        let italicAttribute = [NSAttributedString.Key.font: UIFont.italicSystemFont(ofSize: 14)]
        content.addAttributes(italicAttribute, range: selectedRange())
    }

    func selectedRange() -> NSRange {
        return NSRange(location: 0, length: content.length)
    }
}
```

Hubert Kiełkowski
iOS developer

(RichTextView vs TextEditor)

- **Text Styling and Formatting:** RichTextView supports detailed text styling through attributed strings, including different fonts, colors, and inline image attachments, features not fully supported by SwiftUI's TextEditor.
- **Integration with Text Libraries:** Seamless integration with UIKit-based text manipulation libraries and frameworks, extending capabilities beyond SwiftUI's native text editing functionalities.
- **Flexibility in Text Handling:** Offers greater flexibility in handling complex text layouts and formatting requirements, leveraging UIKit's mature text

Benefits of Integrating UIKit Components with SwiftUI

1. **Advanced Customization:**
    - **Appearance:** UIKit components like UISlider and UIImageView can be customized more extensively compared to their SwiftUI counterparts. Custom thumb images, track images, and more detailed styling options are available.
    - **Behavior:** You can implement complex behaviors and interactions that are more difficult to achieve with SwiftUI's native components.
2. **Access to Delegate Methods:**
    - UIKit components provide delegate methods, allowing for more granular control and detailed event handling.
3. **Legacy Code Integration:**
    - Integrating UIKit with SwiftUI allows you to reuse existing UIKit components and legacy code without major refactoring.

Hubert Kiełkowski
iOS developer

Hope You enjoy it!

Hubert Kiełkowski
iOS developer