SwiftUI: Новый взгляд на разработку интерфейсов в iOS

Введение в SwiftUI. Основные отличия от UIKit

Сегодня мы с вами начинаем изучение **SwiftUI** — относительно нового инструмента Apple, который позволяет разрабатывать пользовательские интерфейсы декларативным способом.

SwiftUI был представлен на конференции **WWDC 2019** и с тех пор активно развивается, постепенно заменяя традиционный **UIKit**. Но почему Apple вообще решила создать новый фреймворк для интерфейсов? Почему не оставить все как есть? Давайте разберемся.

1. Почему появился SwiftUI?

Если мы посмотрим на традиционный **UIKit**, то увидим, что он строится по **императивному принципу**. Это означает, что мы **пошагово** указываем системе, какие элементы интерфейса создавать, какие свойства им устанавливать, какие методы вызвать. Пример на UIKit:

```
let label = UILabel()
label.text = "Привет, мир!"
label.textColor = .black
label.font = UIFont.systemFont(ofSize: 24)
view.addSubview(label)
```

Каждый объект нужно создавать вручную, изменять его свойства, размещать на экране. Такой подход работал годами, но имел ряд недостатков:

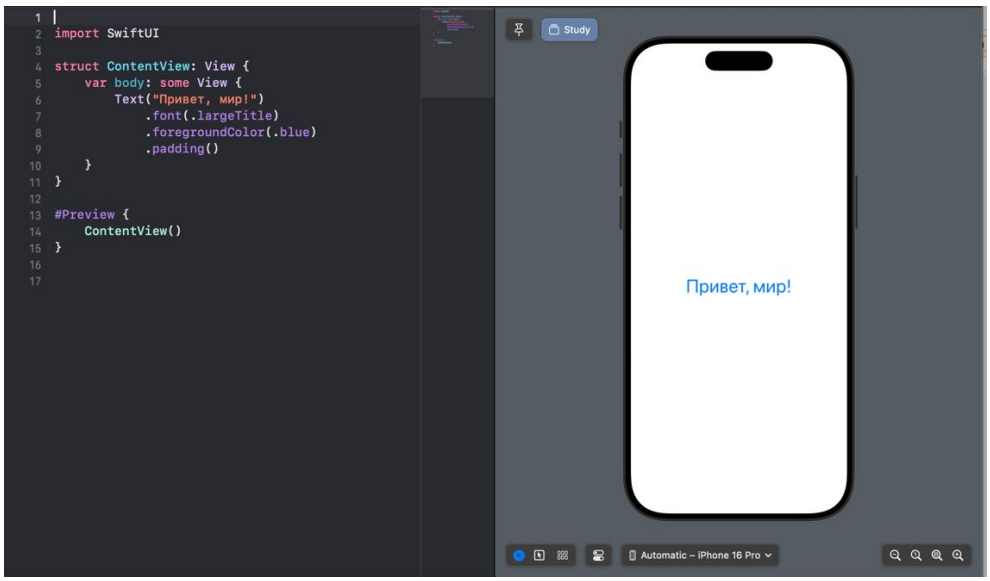
1. **Большое количество кода** даже для простых экранов.
2. **Сложность в обновлении UI**, особенно при работе с асинхронными данными.
3. **Фрагментированность кода**: разметка в Storyboard, логика в коде, данные отдельно.

Apple решила сделать разработку **проще, удобнее и быстрее**, и представила **SwiftUI**.

2. Декларативный подход в SwiftUI

SwiftUI использует **декларативный** стиль программирования. Это означает, что мы **не описываем, как** создать и расположить элементы на экране, а просто **говорим системе, какой интерфейс мы хотим получить**.

Обратите внимание: нигде нет вызовов addSubview, frame, constraints. Мы просто описали, что хотим видеть текст, задали его свойства — и всё! Система сама позаботится о его размещении. Также отличительной особенностью верстки на SwiftUI является то, что в Xcode вы можете сразу видеть результат того, что написали в коде, это сильно упрощает процесс верстки компонентов.



3. Основные отличия SwiftUI от UIKit

Давайте разберем ключевые различия между UIKit и SwiftUI в таблице:

Функция	**UIKit**	**SwiftUI**
Подход	Императивный	Декларативный
Обновление UI	Через UIView и UIViewController	Автоматически с @State, @Binding
Макет	AutoLayout, фреймы, Storyboard	Стековые контейнеры (HStack, VStack, ZStack)
Анимации	UIView.animate() и Core Animation	Встроенные модификаторы
Совместимость	Поддерживается со старых iOS	Требует iOS 13+
Количество кода	Много даже для простых интерфейсов	Минимальное

Как видите, SwiftUI призван **упростить** разработку, сделать её **более гибкой и читаемой**. Однако, поскольку UIKit разрабатывался более **10 лет**, он до сих пор остается основным инструментом для сложных интерфейсов.

4. Создание первого проекта на SwiftUI

Теперь давайте создадим **первый проект** и посмотрим, как устроена структура приложения на SwiftUI.

Шаг 1. Создание проекта в Xcode

1. **Открываем Xcode** и выбираем **“Create a new Xcode project”**.
2. Выбираем шаблон **App** и нажимаем **Next**.
3. В поле **Interface** выбираем **SwiftUI** (по умолчанию Xcode предлагает UIKit).
4. Вводим **имя проекта**, например “SwiftUIExample”.
5. Нажимаем **Next**, выбираем папку и создаем проект.

После создания проекта откроется файл **ContentView.swift** — это главный экран приложения.

5. Структура проекта в SwiftUI

В отличие от UIKit, в SwiftUI **нет** AppDelegate и SceneDelegate, вместо них используется структура App.

Пример SwiftUIExampleApp.swift:

```swift
import SwiftUI

@main
struct SwiftUIExampleApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}

```

Здесь WindowGroup — это контейнер для основного экрана приложения (ContentView).

Файл ContentView.swift:

```swift
import SwiftUI

struct ContentView: View {
    var body: some View {
        VStack {
            Text("Привет, SwiftUI!")
                .font(.title)
                .padding()
        }
    }
}

#Preview {
    ContentView()
}

```

Мы описали Text, добавили модификаторы (font, padding) — и получили готовый экран.

Итак

- SwiftUI — это декларативный фреймворк, который упрощает разработку интерфейсов.
- UIKit использует императивный подход, который требует больше кода.
- SwiftUI легче поддерживать, но он требует iOS 13+ и пока уступает UIKit в возможностях.

Далее мы подробнее разберем **основные компоненты SwiftUI**: Text, Button, Image, HStack/VStack/ZStack, List, NavigationView и многое другое.

Alert. Action sheet. Toggle (UISwitch)

SwiftUI делает работу с UI намного проще, чем UIKit, и сегодня разберем три важных элемента: **Alert**, **Action Sheet** и **Toggle**. Они нужны для общения приложения с пользователем: либо мы его о чем-то предупреждаем, либо даем выбор, либо просто даем возможность включить/выключить что-то.

Alert – всплывающее уведомление

Когда нам нужно показать пользователю какое-то важное сообщение, например, «Вы уверены, что хотите удалить этот файл?» — мы используем Alert. В UIKit для этого был UIAlertController, но теперь все проще.
Вот так можно сделать алерт в SwiftUI:

```swift
import SwiftUI

struct AlertExampleView: View {
    @State private var showAlert = false

    var body: some View {
        VStack {
            Button("Показать Alert") {
                showAlert = true
            }
            .alert("Внимание!", isPresented: $showAlert) {
                Button("OK", role: .cancel) {}
                Button("Удалить", role: .destructive) {
                    print("Удалено")
                }
            } message: {
                Text("Вы уверены, что хотите удалить этот файл?")
            }
        }
    }
}
```

Как это работает?

• У нас есть @State private var showAlert = false — он отслеживает, показывать алерт или нет.

• Кнопка меняет showAlert на true, и срабатывает .alert().

• Мы передаем заголовок, текст и несколько кнопок: «ОК» (ничего не делает) и «Удалить» (красная, потому что destructive).

Простой способ показать пользователю, что его действие требует подтверждения.

Action Sheet – меню выбора

Бывает, что алерт — это слишком, а нам просто нужно дать пользователю несколько вариантов. Например, «Удалить», «Сохранить», «Отменить». В UIKit это был UIAlertController с .actionSheet, а в SwiftUI теперь есть confirmationDialog().

```swift
struct ActionSheetExampleView: View {
    @State private var showActionSheet = false

    var body: some View {
        VStack {
            Button("Показать Action Sheet") {
                showActionSheet = true
            }
            .confirmationDialog("Выберите действие", isPresented: $showActionSheet, titleVisibility:
                .visible) {
                Button("Удалить", role: .destructive) {
                    print("Удалено")
                }
                Button("Сохранить") {
                    print("Сохранено")
                }
                Button("Отмена", role: .cancel) {}
            }
        }
    }
}
```

Что тут происходит?

• Опять же, @State private var showActionSheet = false, чтобы отслеживать, показывать ли меню.
• Кнопка активирует confirmationDialog(), который показывает варианты.
• .destructive делает кнопку красной (важно для критичных действий).
• .cancel отменяет действие (по стандарту он всегда внизу).

Такой подход удобен, когда нужно дать пользователю выбор, но не нагружать его интерфейс лишними кнопками.

Toggle – переключатель

Это аналог UISwitch из UIKit. Например, в настройках мы хотим включить или выключить «Режим полета».

```
struct ToggleExampleView: View {
    @State private var isOn = false

    var body: some View {
        VStack {
            Toggle("Режим полета", isOn: $isOn)
                .padding()
        }
    }
}
```

Простая штука:

• @State private var isOn = false — отслеживает состояние.
• Toggle("Режим полета", isOn: $isOn) привязывается к этому @State, чтобы менять его при переключении.

Приятный бонус – SwiftUI автоматически запоминает последнее состояние Toggle, если использовать его в @AppStorage.

Что в итоге?

• **Alert** — для важной информации и подтверждений.
• **Action Sheet** — когда нужно дать несколько вариантов выбора.
• **Toggle** — удобный способ включать и выключать настройки.

В SwiftUI все это выглядит максимально лаконично и понятно.

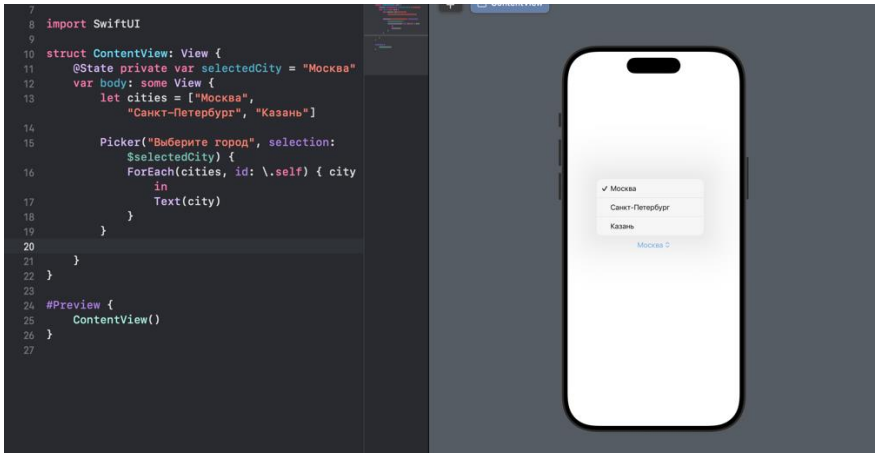
Picker, Form и NavigationView

В SwiftUI компоненты интерфейса, такие как **Picker**, **Form** и **NavigationView**, предоставляют удобные способы взаимодействия с пользователем, при этом освобождая разработчика от множества традиционных задач, связанных с управлением состоянием и жизненным циклом компонентов. Рассмотрим, как эти элементы могут быть использованы для создания простых и понятных интерфейсов.

Picker – простой и элегантный выбор значений

Picker в SwiftUI — это инструмент, который позволяет пользователю выбрать одно из нескольких предложенных значений. Он может выглядеть как простое колесо выбора или как список, в зависимости от контекста.

Допустим, мы создаем экран с настройками города. В UIKit вам нужно было бы использовать UIPickerView с делегатами для отображения и обработки выбора. В SwiftUI это гораздо проще:

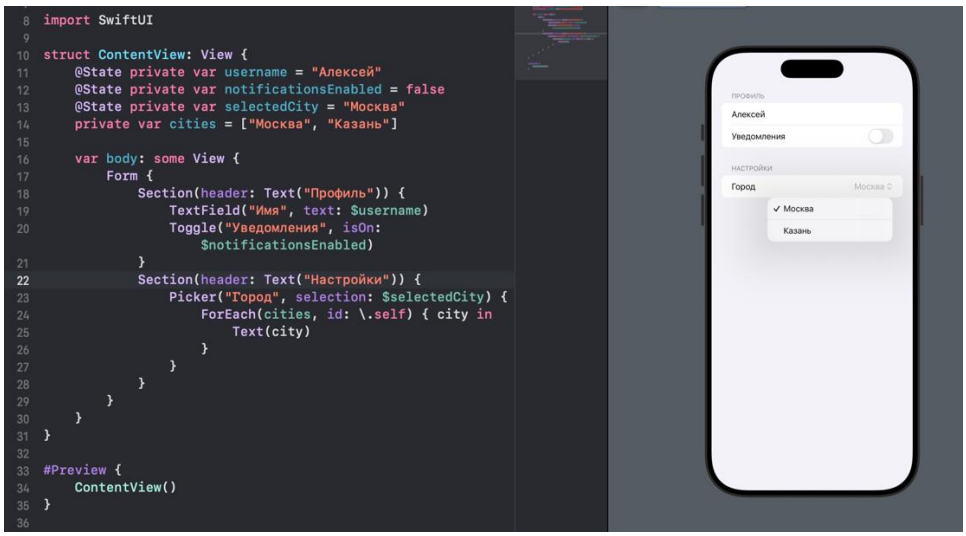


Здесь мы создаем Picker с помощью простого списка городов. Главное отличие — это привязка состояния через @State, благодаря которой при изменении значения города интерфейс обновляется автоматически.

Form – упрощенная работа с формами

Когда дело касается создания форм, SwiftUI снова упрощает задачу. Вместо того чтобы вручную управлять каждым элементом ввода, мы можем использовать Form, который автоматически адаптируется к платформе и формирует интерфейс.

Предположим, у нас есть форма, в которой нужно ввести имя и выбрать город. В UIKit это обычно делается через UITableView, где каждый элемент — это отдельная ячейка с текстовым полем или переключателем. В SwiftUI все, что нужно, — это использовать Form:

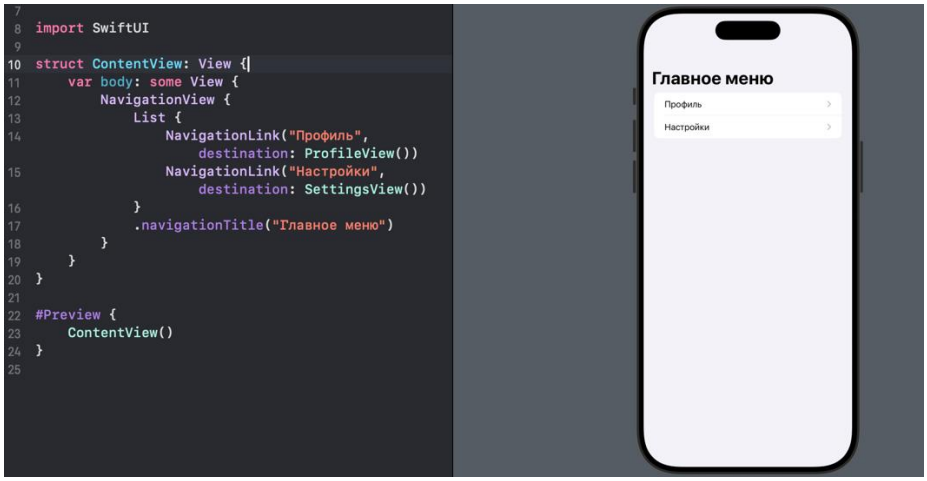


Здесь Form автоматически создает группу элементов с разделами, а также обрабатывает адаптацию под платформу: на iOS это будет UITableView, на macOS — стандартная форма.

NavigationView – упрощение навигации

SwiftUI избавляет от необходимости работать с UINavigationController и предоставляет NavigationView, который позволяет управлять навигацией между экранами. С помощью этого компонента можно легко организовать переходы между представлениями, например, через ссылки в списке.

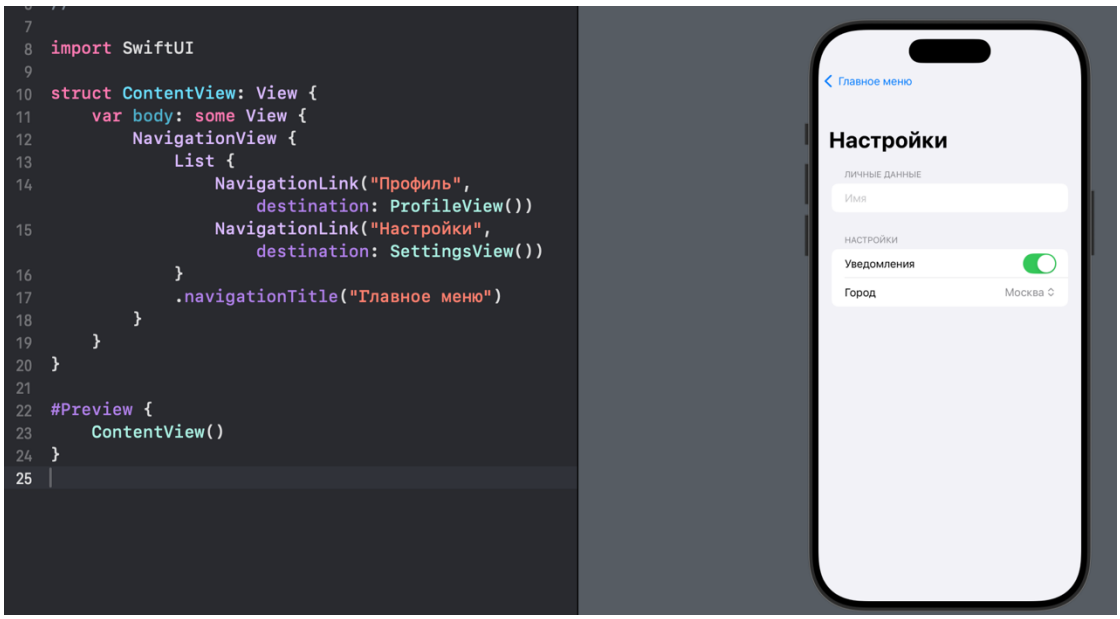
Пример с переходом на экраны профиля и настроек:



Здесь мы оборачиваем список в NavigationView, и все элементы списка автоматически становятся кликабельными, вызывающими переходы на другие экраны. NavigationLink позволяет создать плавный переход, а navigationTitle задает заголовок для текущего экрана.

Что происходит, когда мы комбинируем все это?

Теперь давайте рассмотрим, как все эти компоненты могут работать вместе. Представим, что нам нужно создать экран с настройками, который включает форму для ввода имени пользователя, переключатель для уведомлений и выбор города. Все это мы можем организовать с помощью Form, Picker, и Toggle внутри NavigationView:



Этот пример показывает, как в одном экране можно объединить формы для ввода данных, переключатели и выбор из списка, и все это с поддержкой плавной навигации между экранами.

Таким образом, SwiftUI позволяет создавать интерфейсы, которые намного проще в реализации по сравнению с UIKit, при этом сохраняя гибкость и адаптивность.

Text, TextField и ViewModifier

Продолжаем знакомиться с возможностями SwiftUI, и теперь давайте разберем такие важные компоненты, как **Text**, **TextField** и **ViewModifier**. Эти элементы интерфейса широко используются для отображения текста и ввода данных, а также для изменения внешнего вида представлений с помощью модификаторов.

Text – отображение текста

Text в SwiftUI — это базовый элемент для отображения текста. Он поддерживает форматирование, такие как стили шрифтов, выравнивание, цвет и многое другое. В SwiftUI работать с текстом проще, чем в UIKit, где нужно было настраивать UILabel и его различные атрибуты.

Простой пример использования **Text**:

```
Text("Привет, мир!")
   .font(.largeTitle)
   .foregroundColor(.blue)
```

Здесь мы создаем текст с фразой “Привет, мир!”, устанавливаем размер шрифта через .font(.largeTitle) и изменяем цвет через .foregroundColor(.blue).

Text поддерживает и другие свойства, такие как выравнивание текста (.multilineTextAlignment(.center)) или добавление подчеркивания, жирного шрифта и т. д. В общем, это очень мощный и гибкий инструмент для отображения текста.

TextField – для ввода текста

TextField — это компонент для получения данных от пользователя, и в SwiftUI его использование значительно проще по сравнению с UIKit, где для этого нужно было работать с UITextField и его делегатами. В SwiftUI достаточно просто создать привязку к состоянию и указать, что пользователь будет вводить текст.

Пример с **TextField**:

```
@State private var username: String = ""

var body: some View {
   TextField("Введите имя", text: $username)
      .padding()
      .textFieldStyle(RoundedBorderTextFieldStyle())
      .autocapitalization(.none)
}
```

В данном примере:

- **@State** используется для хранения текста, который вводит пользователь.
- **TextField** позволяет задать подсказку (в данном случае “Введите имя”).
- .textFieldStyle(RoundedBorderTextFieldStyle()) задает стиль поля ввода с округленными углами.
- .autocapitalization(.none) отключает автокапитализацию.

При изменении текста в **TextField** значение автоматически сохраняется в переменной **username** благодаря двусторонней привязке ($username).

ViewModifier – модификация представлений

ViewModifier в SwiftUI — это мощный инструмент для изменения внешнего вида представлений. Он позволяет модифицировать и кастомизировать компоненты через reusable модификаторы, что делает код более читаемым и переиспользуемым.

Например, вместо того чтобы каждый раз вручную настраивать padding, фон, и границу, можно создать собственный

```swift
import SwiftUI

struct MyCustomModifier: ViewModifier {
    func body(content: Content) -> some View {
        content
            .padding()
            .background(Color.blue)
            .cornerRadius(10)
            .foregroundColor(.white)
    }
}

extension View {
    func myCustomStyle() -> some View {
        self.modifier(MyCustomModifier())
    }
}
```

Здесь мы создаем кастомный **ViewModifier** MyCustomModifier, который добавляет отступы, фон и округляет углы. Далее мы расширяем **View**, добавляя новый метод myCustomStyle(), который применяет этот модификатор к любому представлению.

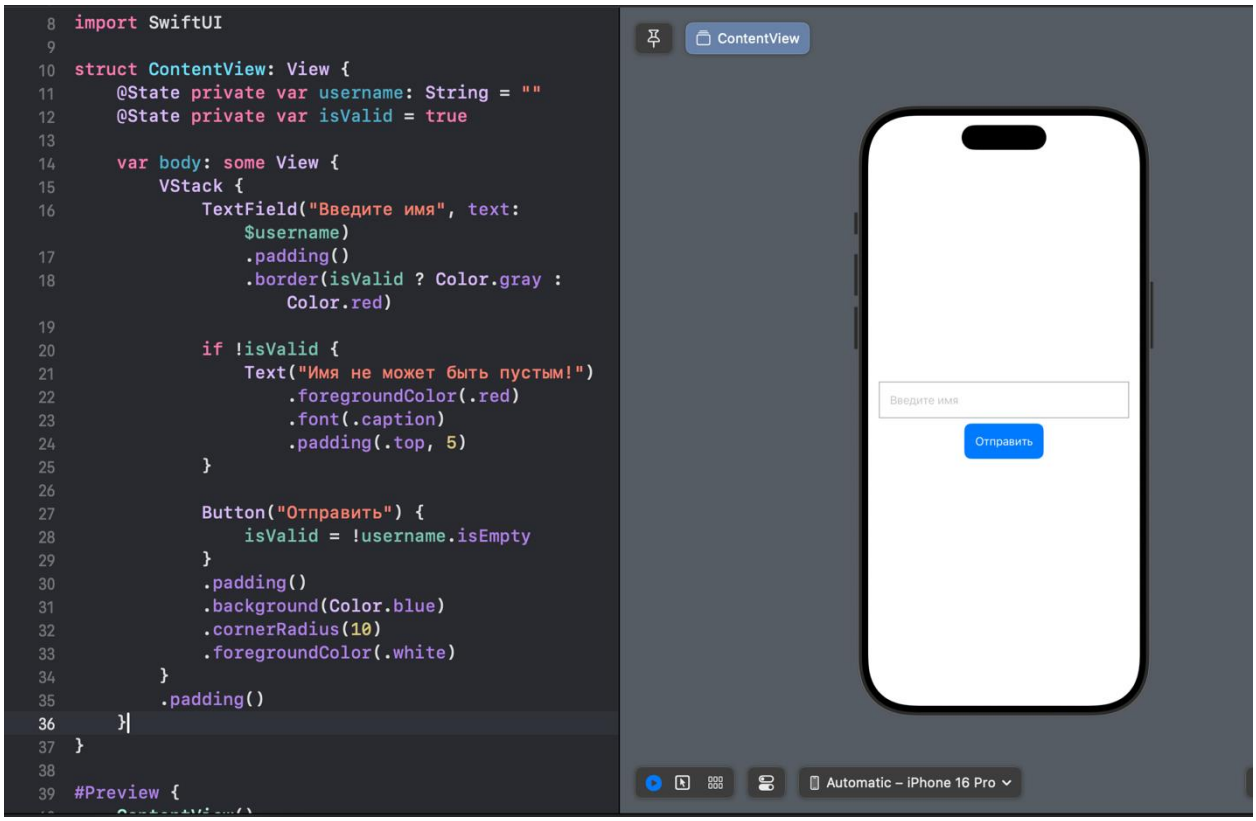
Теперь, чтобы применить этот стиль к элементу, достаточно вызвать .myCustomStyle():

```
Text("Пример текста с кастомным стилем")
    .myCustomStyle()
```

ViewModifier помогает избежать дублирования кода и улучшить структуру проекта, особенно если одно и то же поведение нужно применить к нескольким элементам.

Комбинирование Text, TextField и ViewModifier

Часто вам придется комбинировать **Text**, **TextField** и **ViewModifier** для создания сложных интерфейсов. Например, если вы хотите создать форму с текстовыми полями и отображать подсказки или ошибки:



В этом примере:

- Мы используем **TextField** для ввода имени.
- Если имя пустое, то отображаем текст ошибки **Text**.
- Также используем модификаторы для создания кнопки и для добавления отступов и округления углов.

TextField и **Text** с **ViewModifier** позволяют гибко управлять внешним видом, поведением и обработкой ввода.

Используя эти компоненты, вы можете создавать гибкие и красивые интерфейсы с минимальными усилиями.

Slider, ObservableObject и Segment

Перейдем к трем важным компонентам в SwiftUI, которые активно используются при создании интерактивных интерфейсов: **Slider**, **ObservableObject** и **Segment**. Эти элементы позволяют управлять пользовательским вводом, состоянием приложения и выбором между вариантами. Они часто встречаются в настройках и взаимодействиях с данными, а также обеспечивают плавный и удобный UX.

Slider – Регулировка значений в диапазоне

Slider — это компонент, который дает пользователю возможность выбрать значение в пределах заданного диапазона. Он идеально подходит для таких задач, как настройка громкости, яркости, и других параметров. Важно отметить, что слайдер автоматически связывается с состоянием и обновляет интерфейс при изменении значения.

Вот как можно применить **Slider** для выбора значения:

```
@State private var sliderValue: Double = 50

var body: some View {
   VStack {
      Text("Значение: \(Int(sliderValue))")
         .font(.title)

      Slider(value: $sliderValue, in: 0...100, step: 1)
         .padding()
   }
   .padding()
}
```

• Значение слайдера находится в диапазоне от 0 до 100, и при изменении ползунка автоматически обновляется отображаемый текст.

• Привязка данных через @State позволяет легко отслеживать изменения состояния компонента и обновлять UI.

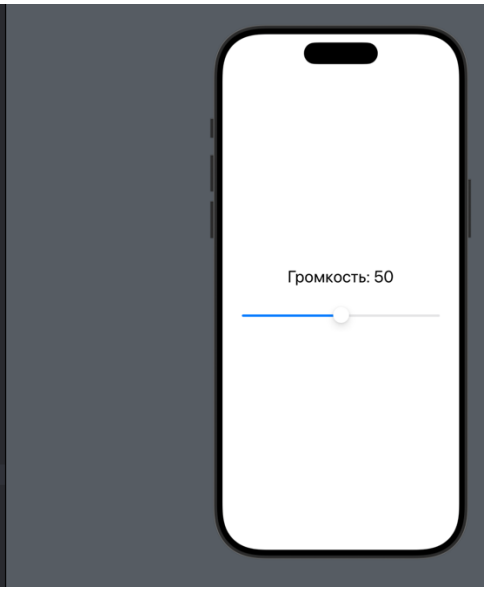
Этот компонент подходит для любого сценария, когда необходимо предоставить пользователю простой способ выбора числового значения в определенном интервале.

ObservableObject – Управление состоянием и привязка данных

Для работы с более сложными состояниями в SwiftUI мы часто используем **ObservableObject**. Это протокол, который позволяет создавать объекты, состояния которых могут быть отслежены интерфейсом. Когда объект, реализующий этот протокол, изменяет свои данные, все связанные представления автоматически обновляются. Таким образом, **ObservableObject** предоставляет эффективный способ управления состоянием на уровне приложения.

Пример реализации:

```swift
import SwiftUI

class UserSettings: ObservableObject {
    @Published var volume: Double = 50
}

struct ContentView: View {
    @StateObject private var settings =
        UserSettings()

    var body: some View {
        VStack {
            Text("Громкость:
                \(Int(settings.volume))")
                .font(.title)

            Slider(value: $settings.volume, in:
                0...100, step: 1)
                .padding()
        }
        .padding()
    }
}

#Preview {
    ContentView()
}
```



• В этом примере класс **UserSettings** отслеживает настройку громкости, а благодаря **@Published** все изменения в значении громкости автоматически отражаются в интерфейсе.

• Использование **@StateObject** для инициализации состояния гарантирует, что объект будет правильно храниться и обновляться в представлении.

Этот подход помогает легко управлять состоянием в приложении, делая его динамичным и отзывчивым.

SegmentedControl (Picker) – Переключение между опциями

Когда нужно предоставить пользователю выбор из нескольких вариантов, удобнее всего использовать **SegmentedControl**. В SwiftUI его реализация осуществляется через **Picker**, который с помощью определенного стиля превращается в сегментированный выбор.

Для примера использования этого компонента можно обратить внимание на следующий код:

```swift
import SwiftUI

struct ContentView: View {
    @State private var selectedOption: Int = 0

    var body: some View {
        VStack {
            Picker("Выберите опцию", selection:
                $selectedOption) {
                Text("Опция 1").tag(0)
                Text("Опция 2").tag(1)
                Text("Опция 3").tag(2)
            }
            .pickerStyle(SegmentedPickerStyle())
            .padding()

            Text("Вы выбрали опцию
                \(selectedOption + 1)")
                .font(.title)
                .padding()
        }
        .padding()
    }
}

#Preview {
    ContentView()
}
```

• **Picker** с **SegmentedPickerStyle()** превращается в **SegmentedControl**, давая возможность выбрать одну из нескольких заранее заданных опций.
• При изменении выбора пользователя отображается информация о текущем выборе.

Сегментированные элементы идеально подходят для случаев, когда необходимо переключаться между несколькими четко определенными состояниями или режимами.

• **Slider** является отличным инструментом для выбора числовых значений в заданном диапазоне и часто используется для настроек, таких как громкость или яркость.
• **ObservableObject** позволяет нам управлять состоянием приложения, автоматически обновляя представления при изменении данных. Это основной принцип, стоящий за привязкой данных в SwiftUI.
• **SegmentedControl**, реализованный через **Picker**, идеально подходит для случаев, когда нужно выбрать один вариант из нескольких.

Все эти компоненты становятся основой динамичного и отзывчивого интерфейса, который так важен для удобства пользователя. Благодаря SwiftUI эти инструменты становятся не только мощными, но и простыми в реализации.

UIActivityView. NavigationView. TabView

Когда мы говорим о создании удобных и функциональных интерфейсов в SwiftUI, важно упомянуть несколько ключевых компонентов, которые помогают решать основные задачи взаимодействия с пользователем. **UIActivityView**, **NavigationView** и **TabView** — это те элементы, которые обеспечивают гибкость и простоту в разработке приложений, в которых требуется обмен контентом, навигация между экранами и организация контента в несколько разделов.

UIActivityView — это элемент, который позволяет пользователям делиться контентом с другими приложениями. Это универсальный компонент для обмена данными, который поддерживает множество приложений, от социальных сетей до мессенджеров. Например, с его помощью можно поделиться текстом, изображениями или ссылками. В SwiftUI нет прямого аналога для **UIActivityView**, однако его легко интегрировать с помощью **UIViewControllerRepresentable**, что позволяет использовать стандартный механизм обмена контентом iOS без написания собственного интерфейса для каждого приложения. Когда пользователь нажимает кнопку “Поделиться”, открывается системное окно обмена, в котором он может выбрать приложение, с помощью которого будет осуществлен обмен.

NavigationView играет ключевую роль в навигации между экранами приложения. Этот элемент помогает создавать многослойную структуру интерфейса, где пользователи могут переходить с одного экрана на другой, и все эти переходы будут интуитивно понятными. **NavigationView** автоматически добавляет элементы управления, такие как кнопка “Назад” или заголовок на панели навигации, что упрощает создание структурированных приложений. Внутри **NavigationView** используется **NavigationLink**, который позволяет переключаться между экранами, что автоматически обеспечивает правильную работу навигации. Такой подход значительно упрощает задачу, так как система сама заботится о создании переходов и правильном отображении элементов управления.

TabView — это компонент для организации контента в виде вкладок, где каждая вкладка может содержать свой экран с уникальным контентом. Он идеально подходит для приложений, где необходимо разделить контент на несколько разделов, каждый из которых

будет доступен с помощью вкладки. Например, классическим примером использования **TabView** является приложение с вкладками “Главная”, “Любимые” и “Настройки”. В SwiftUI добавление вкладок происходит просто: для каждой вкладки создается свой экран, который привязывается к конкретной вкладке через компонент **tabItem**. Это решение значительно упрощает организацию интерфейса, когда необходимо иметь несколько разделов, доступных для переключения с помощью вкладок.

Все эти компоненты играют важную роль в создании удобных и функциональных приложений, позволяя разработчикам сэкономить время на реализации типовых функций и фокусироваться на уникальных аспектах проекта. Они обеспечивают стандартное поведение, которое знакомо пользователям, и делают приложение интуитивно понятным и удобным в использовании.

Предположим, что вы разрабатываете приложение, в котором пользователи могут делиться контентом, перемещаться между несколькими экранами и управлять своим контентом через вкладки. Как это можно реализовать с использованием **UIActivityView**, **NavigationView** и **TabView** в SwiftUI?

Шаг 1: Использование UIActivityView для обмена контентом

Начнем с того, что нам нужно добавить возможность делиться контентом через приложение. Мы создадим кнопку, которая будет открывать системное меню для обмена.

```swift
import SwiftUI

struct ShareContentView: View {
    @State private var showShareSheet = false
    let contentToShare = "This is the content I want to share!"

    var body: some View {
        Button("Share") {
            showShareSheet.toggle()
        }
        .sheet(isPresented: $showShareSheet) {
            ActivityViewController(activityItems: [contentToShare])
        }
    }
}

struct ActivityViewController: UIViewControllerRepresentable {
    var activityItems: [Any]

    func makeUIViewController(context: Context) -> UIActivityViewController {
        return UIActivityViewController(activityItems: activityItems, applicationActivities: nil)
    }

    func updateUIViewController(_ uiViewController: UIActivityViewController, context: Context) {}
}

```

В этом примере, когда пользователь нажимает кнопку “Share”, открывается системное меню для обмена контентом. Мы используем **UIViewControllerRepresentable** для внедрения **UIActivityViewController** в SwiftUI.

Шаг 2: Добавление навигации с помощью NavigationView

Теперь представьте, что ваше приложение состоит из нескольких экранов, и вам нужно организовать переходы между ними. Для этого используем **NavigationView** и **NavigationLink**.

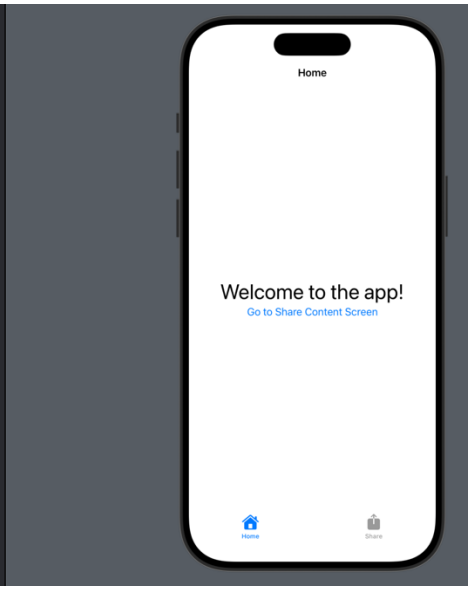
```swift
import SwiftUI

struct ContentView: View {
    var body: some View {
        NavigationView {
            VStack {
                Text("Welcome to the app!")
                    .font(.largeTitle)
                NavigationLink(destination:
                    ShareContentView()) {
                    Text("Go to Share Content Screen")
                        .foregroundColor(.blue)
                }
            }
            .navigationBarTitle("Home", displayMode:
                .inline)
        }
    }
}

#Preview {
    ContentView()
}
```

В этом примере, когда пользователь нажимает на текст “Go to Share Content Screen”, происходит переход на экран с возможностью поделиться контентом. Все это происходит внутри **NavigationView**, который автоматически добавляет панель навигации с кнопкой “Назад” и заголовком.

Шаг 3: Организация контента с помощью TabView

Теперь предположим, что ваше приложение состоит из нескольких разделов, и вам нужно организовать их с помощью вкладок. Это можно сделать с помощью **TabView**.

```swift
//
//  Created by Алексей Смицкий on 16.02.2025.
//

import SwiftUI

struct MainTabView: View {
    var body: some View {
        TabView {
            ContentView()
                .tabItem {
                    Label("Home", systemImage:
                        "house.fill")
                }

            ShareContentView()
                .tabItem {
                    Label("Share", systemImage:
                        "square.and.arrow.up.fill")
                }
        }
    }
}

struct ContentView: View {
    var body: some View {
        NavigationView {
            VStack {
                Text("Welcome to the app!")
```



В этом примере мы создаем два экрана в одном приложении: главный экран с приветствием и экран для обмена контентом. Они помещены в **TabView**, и пользователь может легко переключаться между ними, используя вкладки в нижней части экрана.

В результате мы получаем приложение, которое:

1. Позволяет пользователям делиться контентом через системное меню с помощью **UIActivityView**.
2. Обеспечивает плавную навигацию между экранами с помощью **NavigationView**.
3. Организует контент в виде вкладок, что дает пользователям возможность легко переключаться между различными разделами с помощью **TabView**.

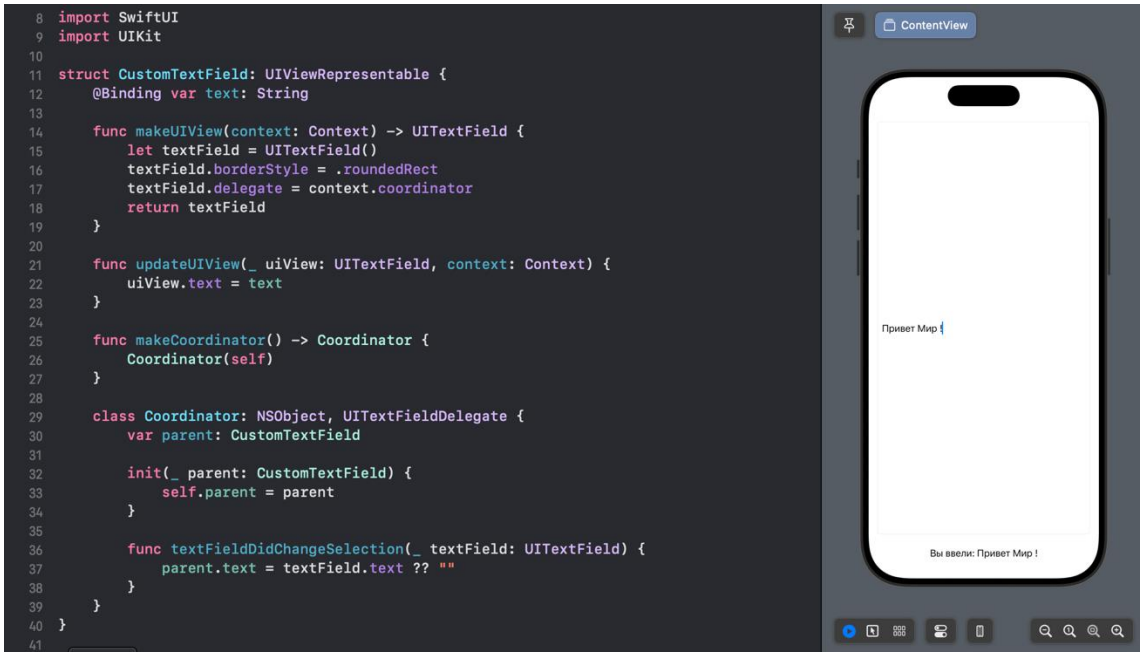
SwiftUI и UIKit: Как связать два мира

Хотя **SwiftUI** значительно упростил разработку интерфейсов, многие проекты по-прежнему используют **UIKit**. Важно понимать, как можно взаимодействовать между ними, чтобы, например, использовать мощные возможности **UIKit** в SwiftUI или, наоборот, добавить SwiftUI-компоненты в старые проекты.

Использование UIKit в SwiftUI

В SwiftUI можно интегрировать любые **UIKit-компоненты** с помощью протокола UIViewControllerRepresentable (для UIViewController) или UIViewRepresentable (для UIView). Например, давайте добавим **UITextField** в SwiftUI.

Пример: UITextField внутри SwiftUI



Использование SwiftUI в UIKit

Если вам нужно добавить **SwiftUI View** в существующий UIKit-проект, используйте UIHostingController.

Пример: Встраивание SwiftUI в UIKit

Представьте, что у вас есть экран на **UIKit**, и вы хотите добавить туда SwiftUI-вью. Это можно сделать так:

```swift
import SwiftUI
import UIKit

struct SwiftUIView: View {
    var body: some View {
        Text("Привет из SwiftUI!")
            .font(.title)
            .padding()
    }
}

class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

        let swiftUIView = SwiftUIView()
        let hostingController = UIHostingController(rootView: swiftUIView)

        addChild(hostingController)
        hostingController.view.frame = view.bounds
        view.addSubview(hostingController.view)
        hostingController.didMove(toParent: self)
    }
}
```

Здесь UIHostingController создает мост между UIKit и SwiftUI. Мы добавляем его в UIViewController, и SwiftUI-интерфейс рендерится в UIKit.

Когда использовать интеграцию?

- **Нужно использовать UIKit в SwiftUI**, если у вас есть сложные кастомные элементы, которые еще не реализованы в SwiftUI (например, UICollectionView).
- **Нужно использовать SwiftUI в UIKit**, если вы постепенно переводите приложение на SwiftUI и хотите использовать его преимущества (быстрое создание UI, декларативный подход).

В заключении хотелось бы отметить, что

SwiftUI — это **современный декларативный фреймворк** для создания пользовательского интерфейса на устройствах Apple. Он позволяет разработчикам писать **менее громоздкий, понятный и читаемый код**, сокращая количество ошибок и улучшая поддержку интерфейсов на разных устройствах.

Одним из важных преимуществ SwiftUI является его **гибкость**: он отлично интегрируется с UIKit, позволяя использовать старый код и постепенно мигрировать приложения.

Хотя SwiftUI еще развивается и уступает UIKit в возможностях глубокой кастомизации, его **будущее очевидно** — Apple активно продвигает этот фреймворк как **основной инструмент** для разработки UI. Освоение SwiftUI уже сейчас дает возможность **разрабатывать быстрее и эффективнее**, что особенно важно для новых проектов.

Чтобы углубить знания, стоит изучить **анимации, работу со списками, обработку жестов и управление состоянием**. SwiftUI — это **не просто новый фреймворк, а принципиально иной взгляд на разработку интерфейсов**, и его изучение — важный шаг для любого iOS-разработчика.