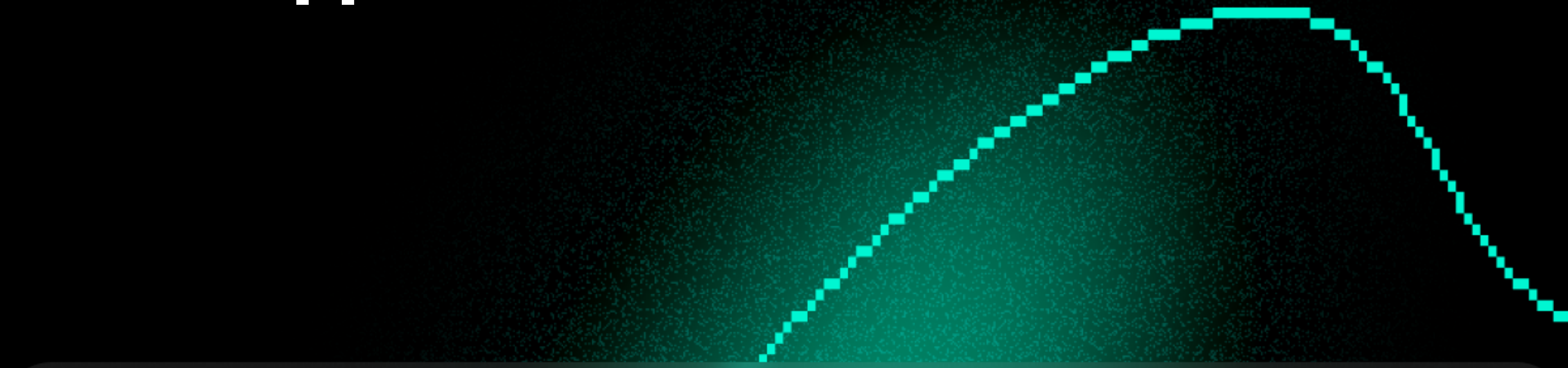
Compose и SwiftUI: Найди 10 отличий



Контур

Алексей Панов

Ведущий инженер-программист

Обо мне

- Живу в Екатеринбурге
- Техлид инфраструктуры мобильной разработки в Контуре

ajiekcx

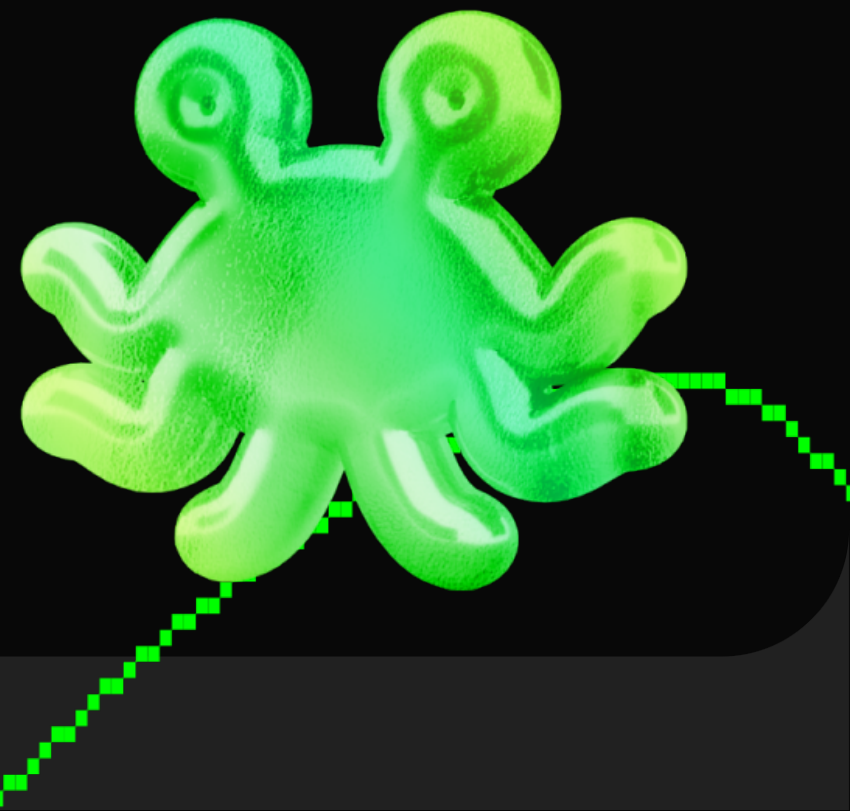
Алексей Панов

Kotlin Adept Notes

Интересные посты:

- Что выбрать для навигации в Compose
- Декларативный Bottom Sheet
- Compose анимации через стейт
- Flutter vs Compose
- Корутинные рецепты
- Как вкатиться в KMP без MacOS

UI фреймворки под капотом



SwiftUI и Compose

```swift
struct Counter: View {
  @State private var count = 0

  var body: some View {
    VStack {
      Text("Clicked: \(count)")
      Button(action: { count += 1 }) {
        Text("Increment")
      }
    }
  }
}
```

```kotlin
@Composable
fun Counter() {
    var count by remember { mutableStateOf(0) }

    Column {
        Text("Clicked: $count")
        Button(onClick = { count++ }) {
            Text("Increment")
        }
    }
}
```

SwiftUI и Compose

```swift
struct Counter: View {
  @State private var count = 0

  var body: some View {
    VStack {
      Text("Clicked: \(count)")
      Button(action: { count += 1 }) {
        Text("Increment")
      }
    }
  }
}
```

```kotlin
@Composable
fun Counter() {
    var count by remember { mutableStateOf(0) }

    Column {
        Text("Clicked: $count")
        Button(onClick = { count++ }) {
            Text("Increment")
        }
    }
}
```

SwiftUI и Compose

```swift
struct Counter: View {
  @State private var count = 0

  var body: some View {
    VStack {
      Text("Clicked: \(count)")
      Button(action: { count += 1 }) {
        Text("Increment")
      }
    }
  }
}
```

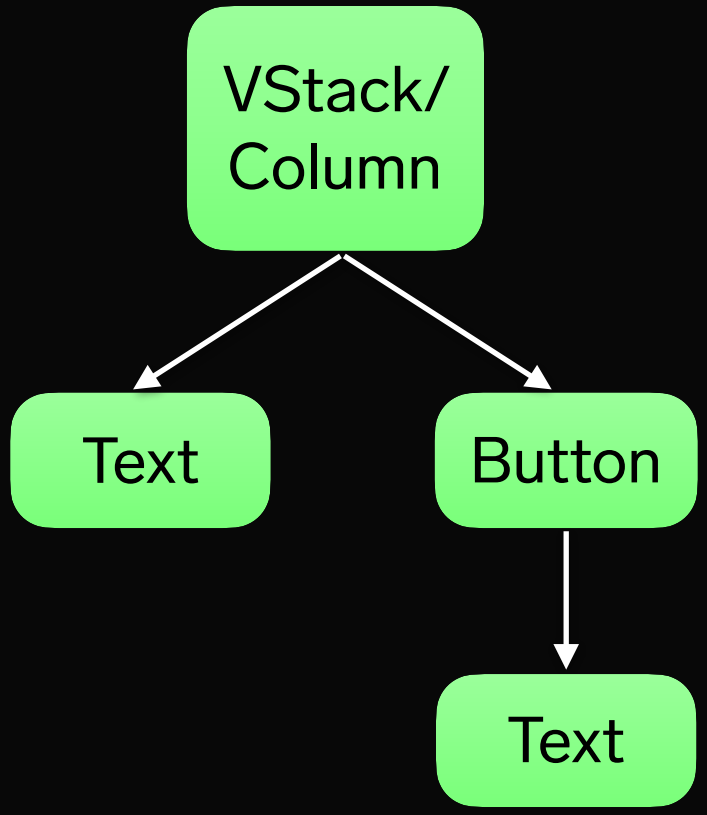
```kotlin
@Composable
fun Counter() {
    var count by remember { mutableStateOf(0) }

    Column {
        Text("Clicked: $count")
        Button(onClick = { count++ }) {
            Text("Increment")
        }
    }
}
```

SwiftUI и Compose

```swift
struct Counter: View {
  @State private var count = 0

  var body: some View {
    VStack {
      Text("Clicked: \(count)")
      Button(action: { count += 1 }) {
        Text("Increment")
      }
    }
  }
}
```

```kotlin
@Composable
fun Counter() {
    var count by remember { mutableStateOf(0) }

    Column {
        Text("Clicked: $count")
        Button(onClick = { count++ }) {
            Text("Increment")
        }
    }
}
```

SwiftUI и Compose



SwiftUI

```swift
struct Counter: View {
  @State private var count = 0

  var body: some View {
    VStack {
      Text("Clicked: \(count)")
      Button(action: { count += 1 }) {
        Text("Increment")
      }
    }
  }
}
```

SwiftUI

```swift
struct Counter: View {
  @State private var count = 0

  var body: some View {
    return VStack {
      Text("Clicked: \(count)")
      Button(action: { count += 1 }) {
        Text("Increment")
      }
    }
  }
}
```

SwiftUI

```
struct Counter: View {
  @State private var count = 0

  var body: some View {
    return VStack {
      Text("Clicked: \(count)")
      button()
    }
  }

  @ViewBuilder func button() -> some View {
    Button(action: { count += 1}) {
      Text("Increment")
    }
  }
}
```

SwiftUI

```swift
struct Counter: View {
  @State private var count = 0

  var body: some View {
    return VStack {
      Text("Clicked: \(count)")
      button()
    }
  }

  @ViewBuilder func button() -> some View {
    Button(action: { count += 1}) {
      Text("Increment")
    }
  }
}
```

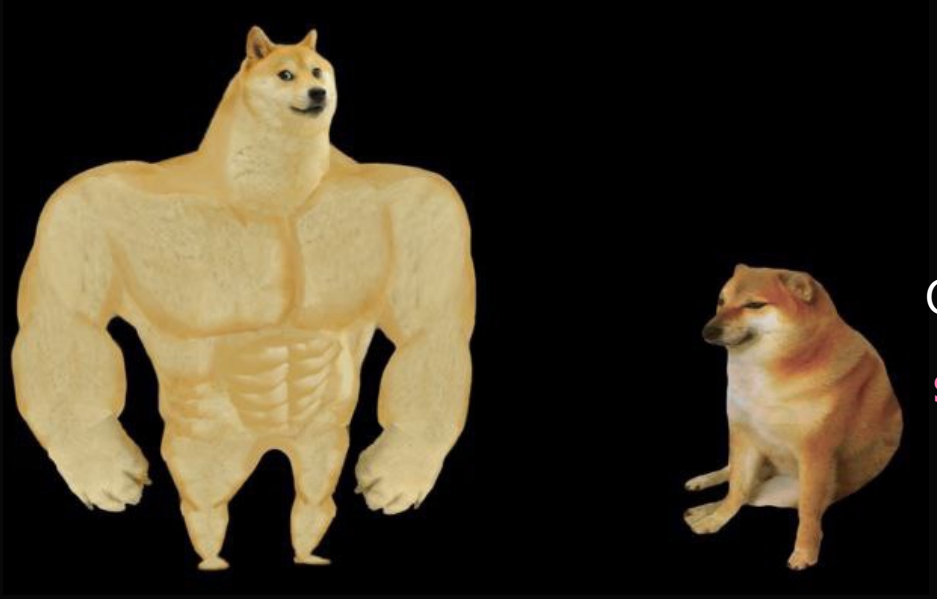
```swift
@resultBuilder public struct ViewBuilder {
…
    public static func
buildBlock<Content>(_ content: Content) ->
Content where Content : View
…
}
```

SwiftUI

```
struct Counter: View {
  @State private var count = 0

  var body: VStack<TupleView<(Text, Button<Text>)>> {
    return VStack {
      Text("Clicked: \(count)")
      button()
    }
  }

  @ViewBuilder func button() -> some View {
    Button(action: { count += 1}) {
      Text("Increment")
    }
  }
}
```

SwiftUI дженерик

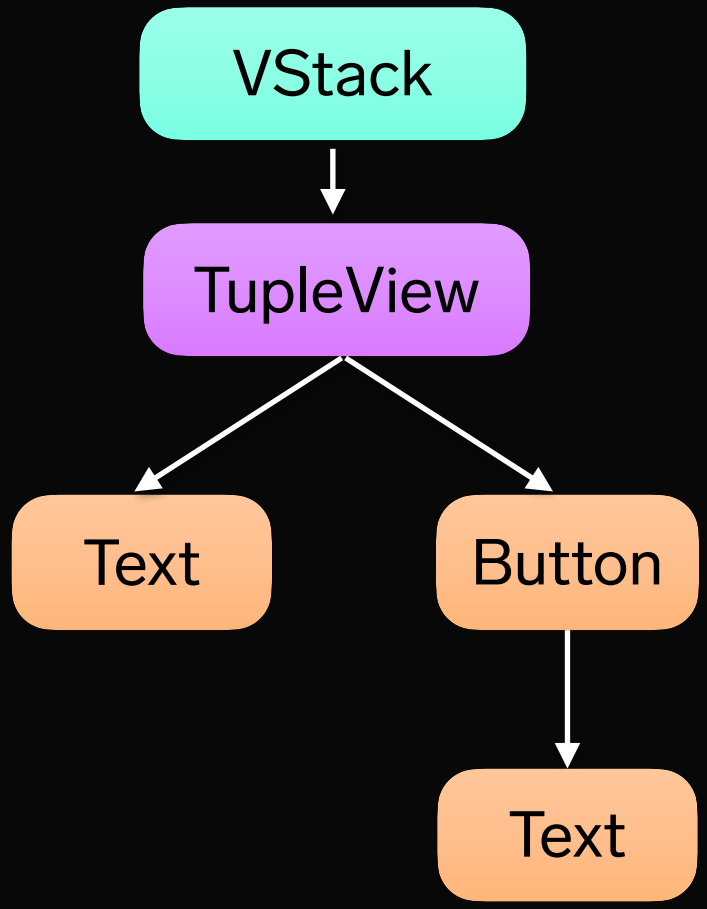
```
ModifiedContent<VStack<TupleView<(SummaryTopBar, Spacer,
_ConditionalContent<_ConditionalContent<Spinner, ErrorView>,
_ConditionalContent<ModifiedContent<_ConditionalContent<ModifiedContent<ModifiedCon
tent<ListContentView<TupleView<(ModifiedContent<SummaryRevenueWidget,
_PaddingLayout>, ModifiedContent<SummarySalesWidget, _PaddingLayout>, … >>
```

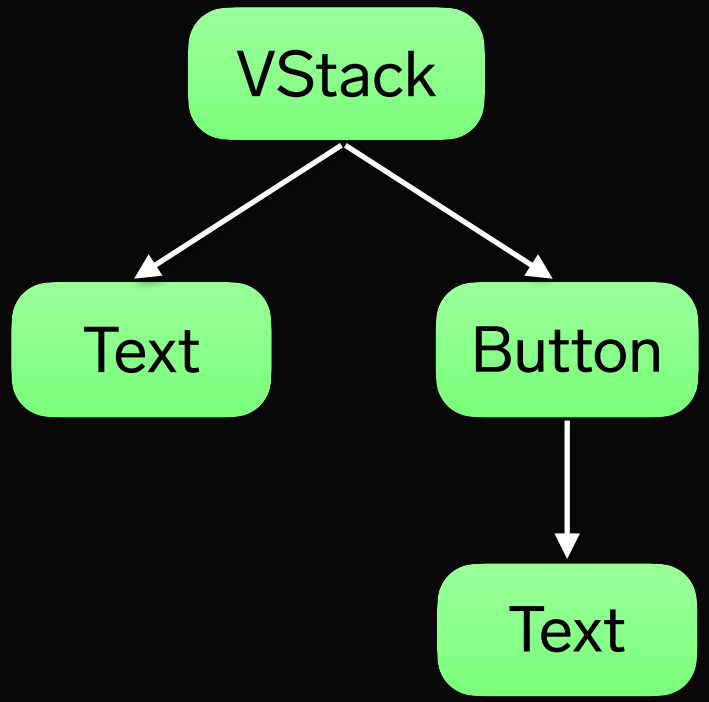


Обычный дженерик

```
struct Inject<T> { }
```

View graph vs Layout tree





Алгоритм работы SwfitUI

- Инициализация View и построение графа
- Подписка на стейт и вызов body
- Рендеринг layout tree

Compose под капотом

```
@Composable
fun Counter() {
    var count by remember { mutableStateOf(0) }

    Column {
        Text("Clicked: $count")
        Button(onClick = { count++ }) {
            Text("Increment")
        }
    }
}
```

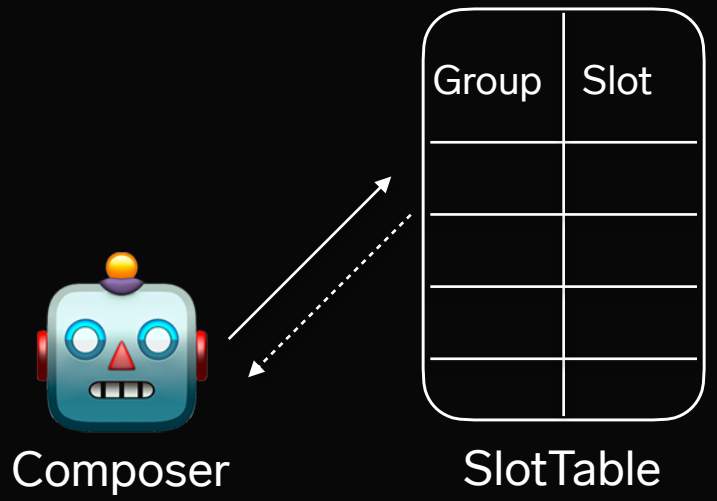
Compose под капотом

```
fun Counter($composer: Composer) {
    $composer.startRestartGroup(123)
    var count by remember($composer) { mutableStateOf(0) }

    Column($composer) {
        Text("Clicked: $count", $composer)
        Button(onClick = { count++ }, $composer) {
            Text("Increment", $composer)
        }
    }
    $composer.endRestartGroup()
}
```

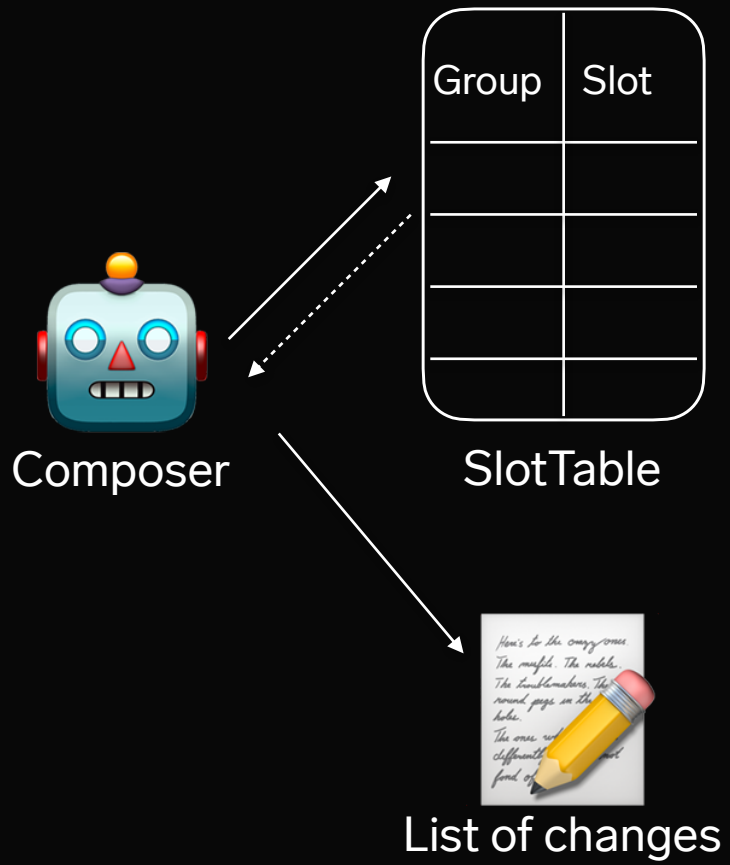
Compose под капотом

Composer

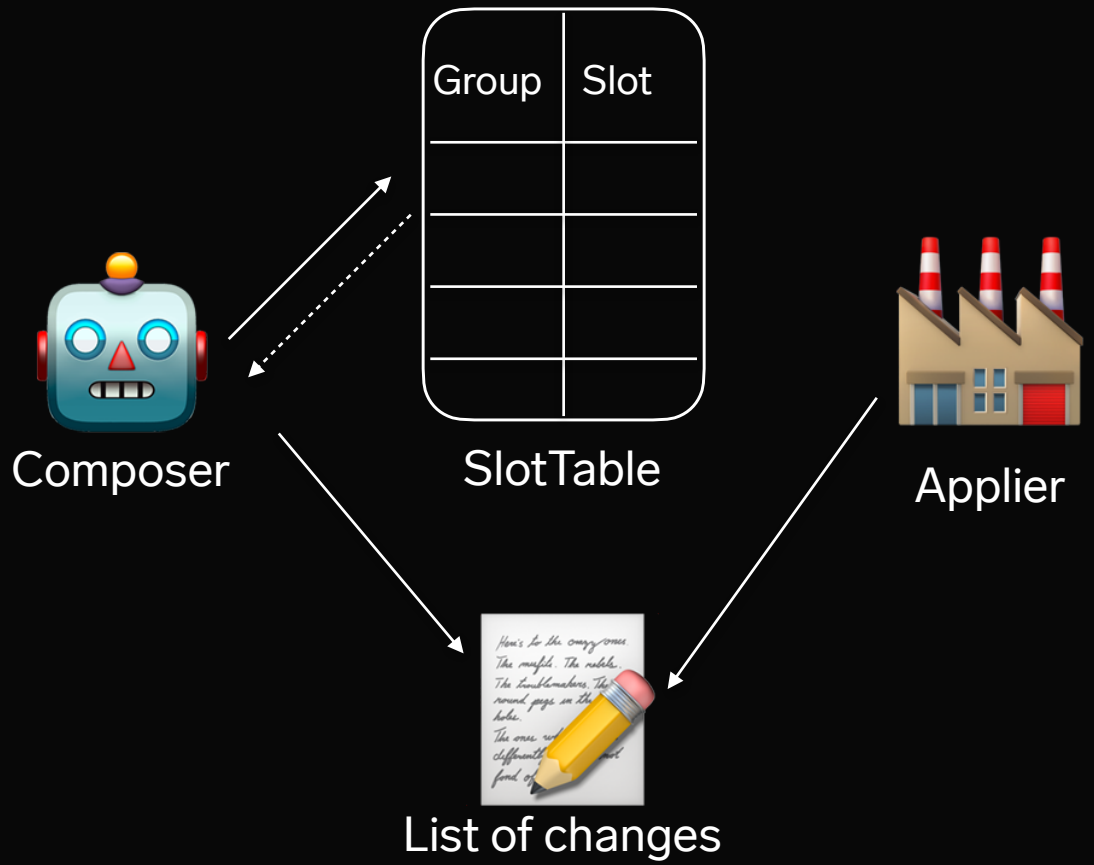
Compose под капотом



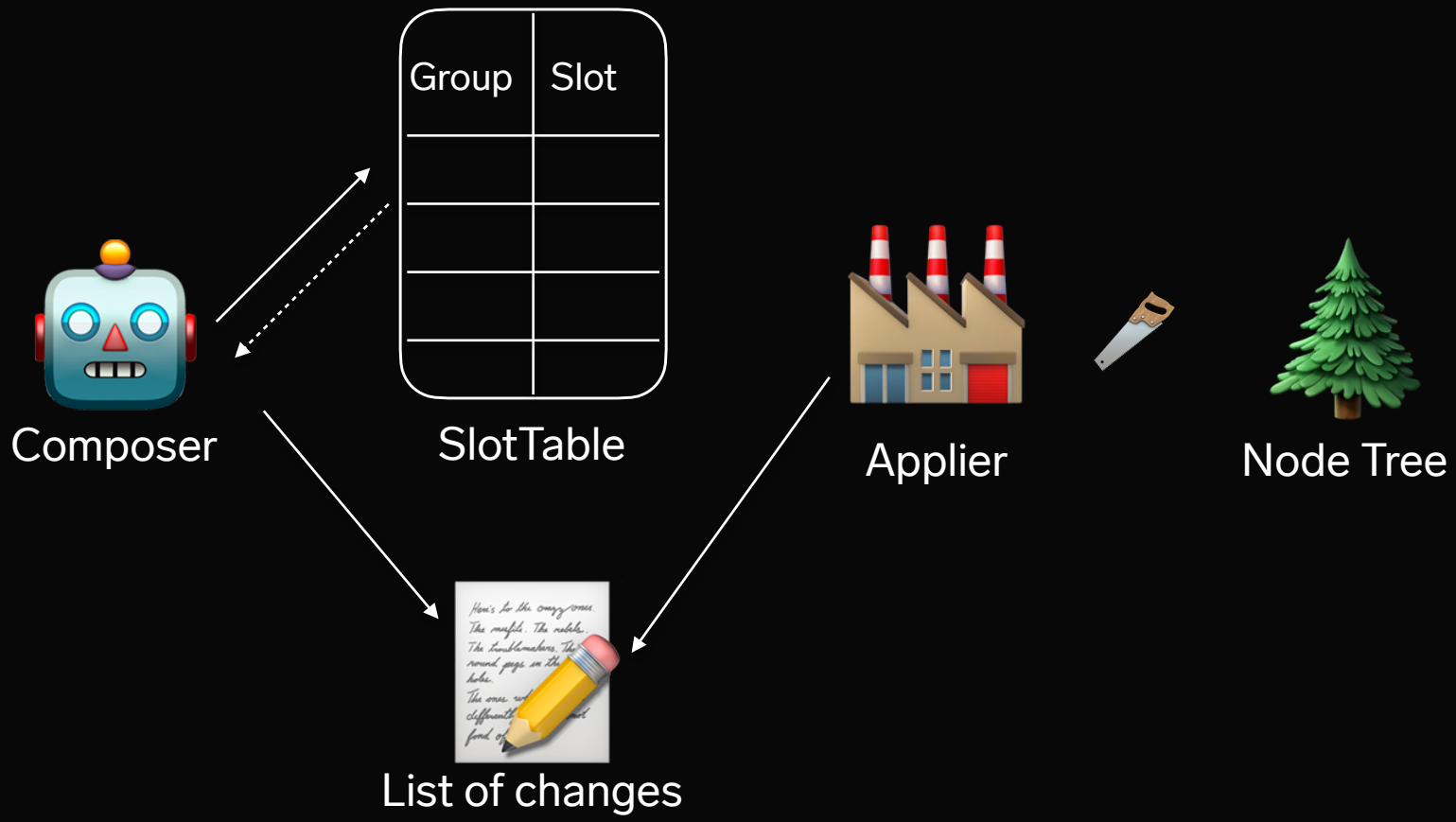
Compose под капотом



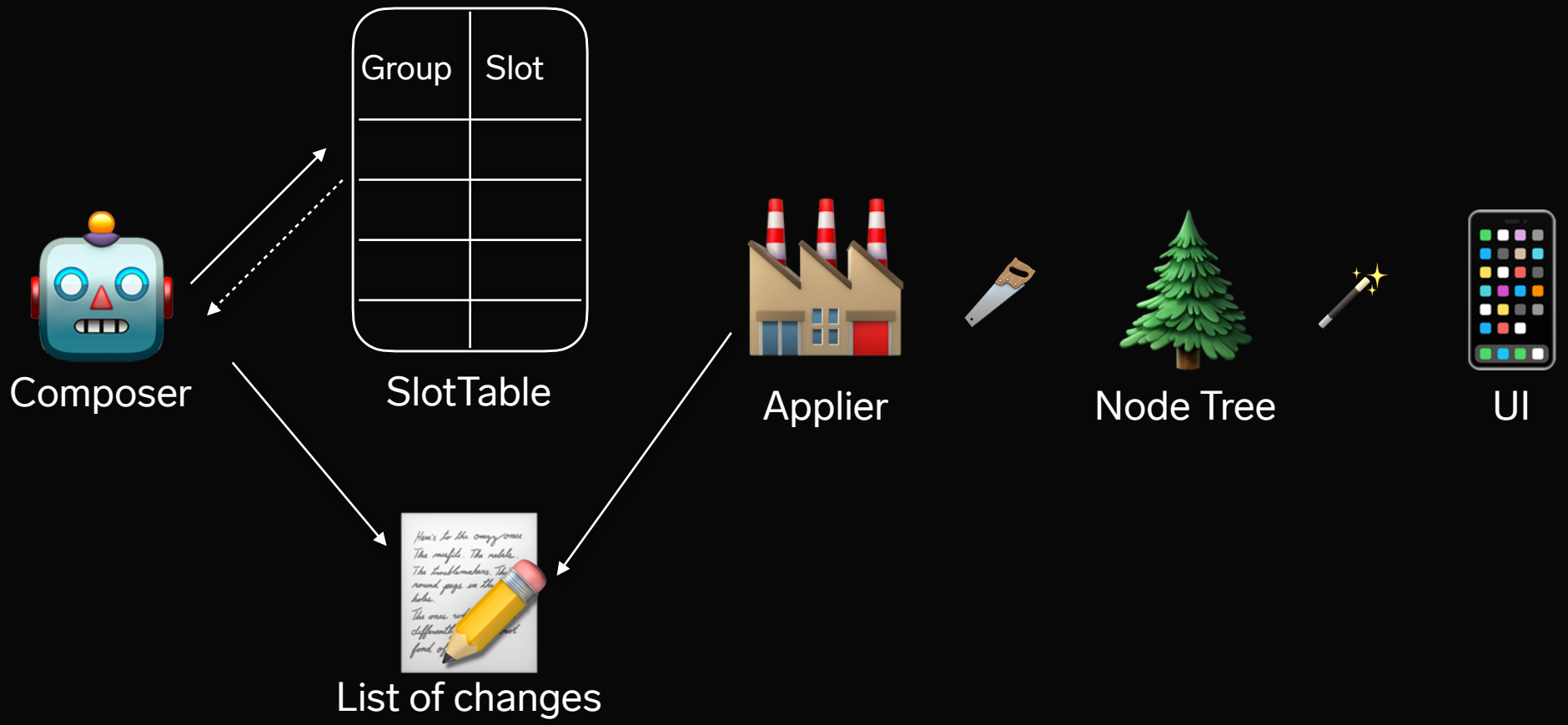
Compose под капотом



Compose под капотом



Compose под капотом



Ключевые отличия

SwiftUI

- Использует структуры
- Строго типизирован
- Конструирует UI

Compose

- Использует функции
- Нет типизации
- Эмитит UI

Состояние и жизненный цикл

State в Compose

```
var count by remember { mutableStateOf(0) }
```

- by — ключевое слово Котлин для реализации делегатов
- remember — сохраняет значение между вызовами функции

State и MutableState

```
@Stable
interface State<out T> {
    val value: T
}
```

```
@Stable
interface MutableState<T> : State<T> {
    override var value: T
    operator fun component1(): T
    operator fun component2(): (T) → Unit
}
```

Mobius
2023 Autumn

Что скрывает State в Compose

Алексей Панов

Контур

SwiftUI State

```
@State private var count = 0
```

SwiftUI State

```
@State private var count = 0
```

```
@frozen @propertyWrapper public struct State<Value> : DynamicProperty {
    public init(wrappedValue value: Value)
    public init(initialValue value: Value)
    public var wrappedValue: Value { get nonmutating set }
    public var projectedValue: Binding<Value> { get }
}
```

SwiftUI State

```
@State private var text = ""

TextField("Placeholder", text: $text)

TextField("Placeholder", text: Binding(get: { text }, set: { text = $0 }))
```

SwiftUI StateObject

```swift
final class ViewModel: ObservableObject {
  @Published private(set) var count = 0

  func inc() { count += 1 }
}

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

  var body: some View {
    return VStack {
      Text("Clicked: \(viewModel.count)")

      Button(action: viewModel.inc) {
        Text("Increment")
      }
    }
  }
}
```

SwiftUI ObservedObject

```swift
final class ViewModel: ObservableObject {
  @Published private(set) var count = 0

  func inc() { count += 1 }
}

struct Counter: View {
  @ObservedObject private var viewModel: ViewModel = ViewModel()

  var body: some View {
    return VStack {
      Text("Clicked: \(viewModel.count)")

      Button(action: viewModel.inc) {
        Text("Increment")
      }
    }
  }
}
```

SwiftUI ObservedObject

```swift
final class ViewModel: ObservableObject {
  @Published private(set) var count = 0

  func inc() { count += 1 }
}

struct Counter: View {
  @ObservedObject private var viewModel: ViewModel = ViewModel()

  var body: some View {
    return VStack {
      Text("Clicked: \(viewModel.count)")

      Button(action: viewModel.inc) {
        Text("Increment")
      }
    }
  }
}
```

ObservedObject vs StateObject

```
struct Counter: View {
  @State private var count = 0

  var body: some View {
    return VStack {
      ObservedCounter() // Внутренний счетчик обнулится при изменении count
      StateCounter()
      Text("Clicked: \(count)")
      Button(action: { count += 1}) {
        Text("Increment")
      }
    }
  }
}
```

Проблемы ObservableObject

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

  var body: some View {
    Self._printChanges()
    return VStack {
      Button(action: viewModel.inc) {
        Text("Increment")
      }
    }
  }
}
```

```
UnusedCounter: _viewModel changed.
```

Observation framework

```swift
@Observable final class ObservableViewModel {
  private(set) var count = 0

  func inc() { count += 1 }
}

struct ObservableCounter: View {
  @State private var viewModel = ObservableViewModel()

  var body: some View {
    return VStack {
      Text("Clicked: \(viewModel.count)")
      Button(action: viewModel.inc) {
        Text("Increment")
      }
    }
  }
}
```

Observation framework

```swift
@Observable final class ObservableViewModel {
  private(set) var count = 0

  func inc() { count += 1 }
}

struct ObservableCounter: View {
  @State private var viewModel = ObservableViewModel()

  var body: some View {
    return VStack {
      Text("Clicked: \(viewModel.count)")
      Button(action: viewModel.inc) {
        Text("Increment")
      }
    }
  }
}
```

Observation бэкпорт

Perception

CI passing | Slack chat | Swift 5.10 | 5.9 | Platforms iOS | macOS | visionOS | tvOS | watchOS | Linux

Observation tools for platforms that do not officially support observation.

Learn More

This library was created by Brandon Williams and Stephen Celis, who host the Point-Free video series which explores advanced Swift language concepts.

Environment Object

```
@main
struct sampleApp: App {

  var body: some Scene {
    WindowGroup {
      Counter()
        .environmentObject(ViewModel())
    }
  }
}

struct Counter: View {
  @EnvironmentObject private var viewModel: ViewModel

  var body: some View { … }
}
```

Compose Composition locals

```
val LocalViewModel = compositionLocalOf<ViewModel> { error("no instance provided") }

CompositionLocalProvider(LocalViewModel provides ViewModel()) {
    Counter()
}

@Composable
fun Counter() {
    val viewModel = LocalViewModel.current
    Column {
        Text("Count: ${viewModel.count}")
        Button(onClick = viewModel::inc) {
            Text("Increment")
        }
    }
}
```

Системные параметры

```
@Environment(\.locale) var locale: Locale
```

SwiftUI

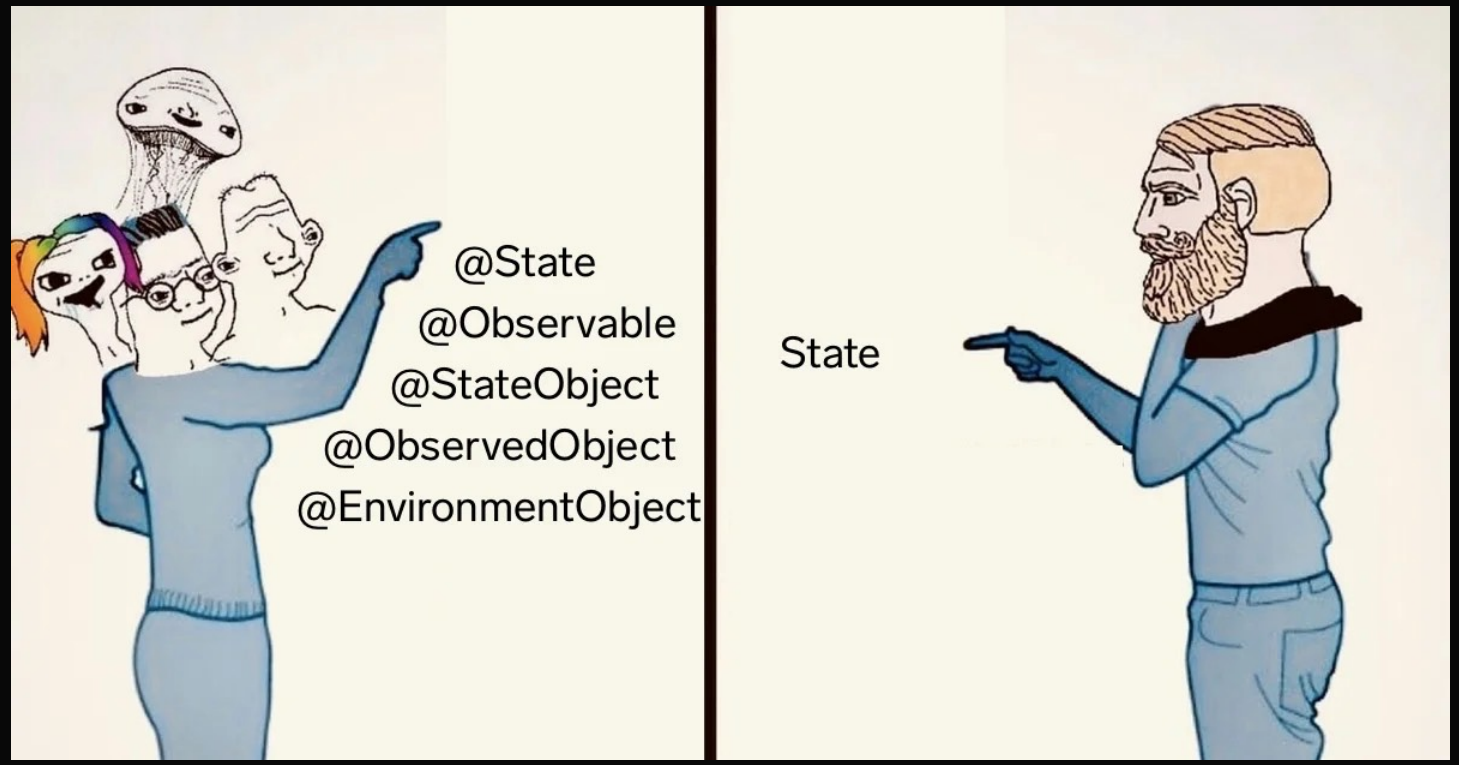
```
LocalConfiguration.current.locales
```

Compose

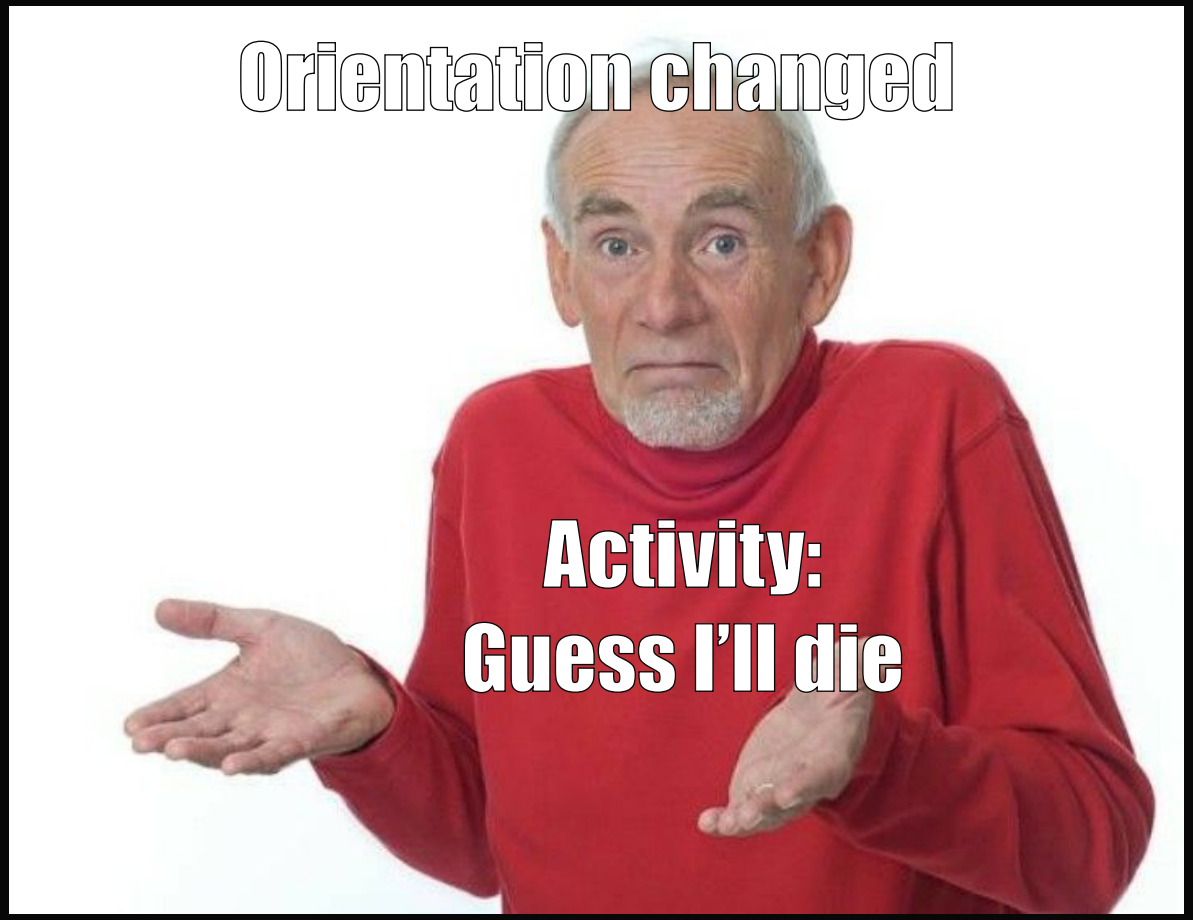
Composition locals под капотом

```
internal class DynamicProvidableCompositionLocal<T> constructor(
    private val policy: SnapshotMutationPolicy<T>,
    defaultFactory: () → T
) : ProvidableCompositionLocal<T>(defaultFactory) {

    @Composable
    override fun provided(value: T): State<T> = remember
{ mutableStateOf(value, policy) }.apply {
        this.value = value
    }
}
```

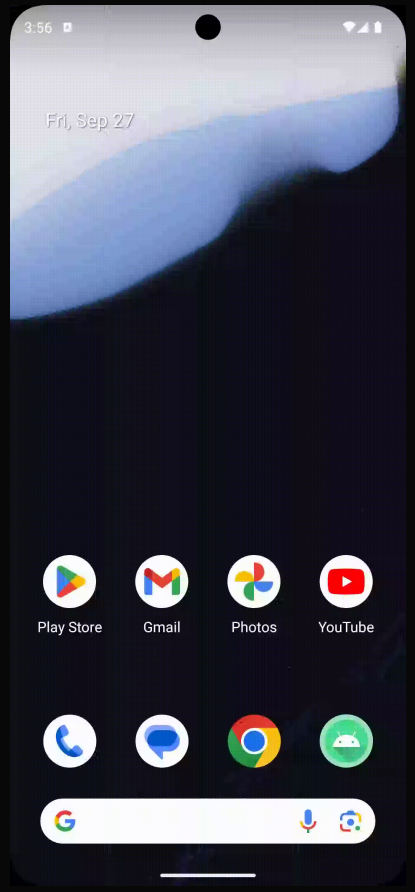
SwiftUI vs Compose



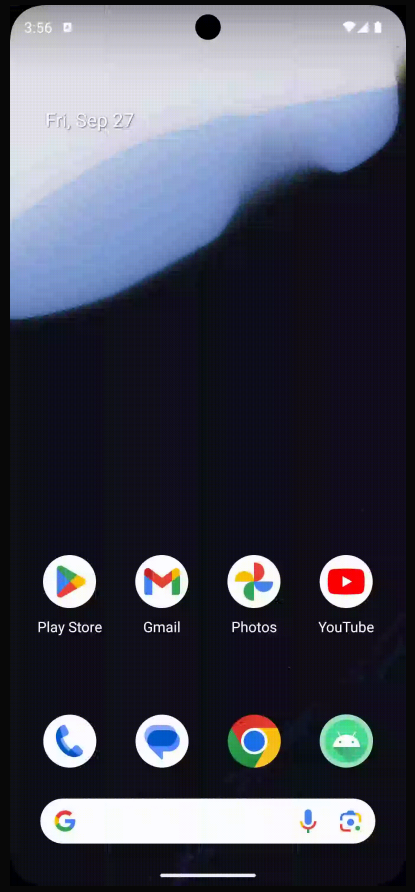
Жизненный цикл



Жизненный цикл в Compose



Жизненный цикл в Compose



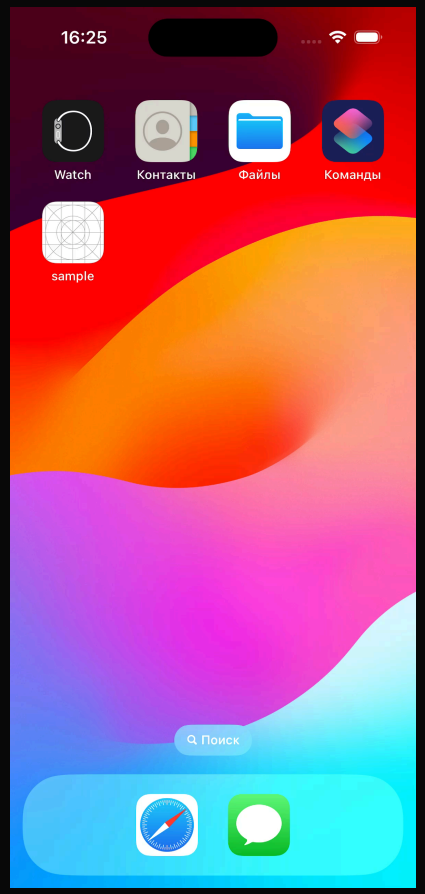
- OnCreate
- OnStart
- OnResume

- OnPause
- OnStop

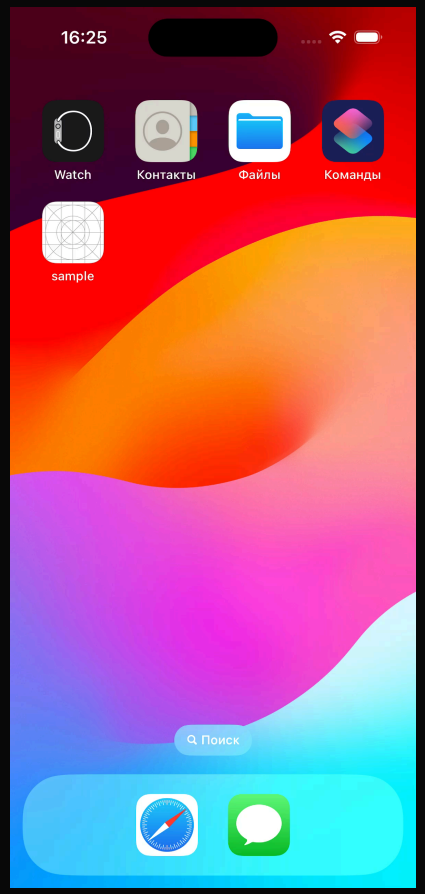
- OnStart
- OnResume

- OnPause
- OnStop
- OnDestroy

Жизненный цикл в SwiftUI



Жизненный цикл в SwiftUI



- Active
- Inactive
- Background
- Inactive
- Active
- Inactive
- Background

Соответствие ЖЦ

Compose	SwiftUI
OnCreate	init
OnStart	—
OnResume	Active
OnPause	Inactive
OnStop	Background
OnDestroy	deinit

Соответствие ЖЦ

Compose	SwiftUI	UIKit
OnCreate	init	init
OnStart	—	viewWillAppear viewWillEnterForeground
OnResume	Active	didBecomeActive
OnPause	Inactive	willResignActive
OnStop	Background	viewDidDisappear didEnterBackground
OnDestroy	deinit	didLeaveWindowHierarchy

SwiftUI ScenePhase

```
@main
struct sampleApp: App {
  @Environment(\.scenePhase) var scenePhase: ScenePhase

  var body: some Scene {
    WindowGroup {
      Text("Hello SwiftUI")
        .onChange(of: scenePhase) { oldPhase, newPhase in
          switch newPhase {
          case .active: print("Active")
          case .inactive: print("Inactive")
          case .background: print("Background")
          @unknown default: break
          }
        }
    }
  }
}
```

Compose LifecycleEffect

```
@Composable
fun LifecycleEventEffect(
    event: Lifecycle.Event,
    lifecycleOwner: LifecycleOwner = LocalLifecycleOwner.current,
    onEvent: () → Unit
) { ... }
```

SwiftUI lifecycle

```
struct LifecycleView: View {
  init() {
    // Может вызываться сколько угодно раз
  }

  var body: some View {
    Text("Hello") // Может вызываться сколько угодно раз
  }
}
```

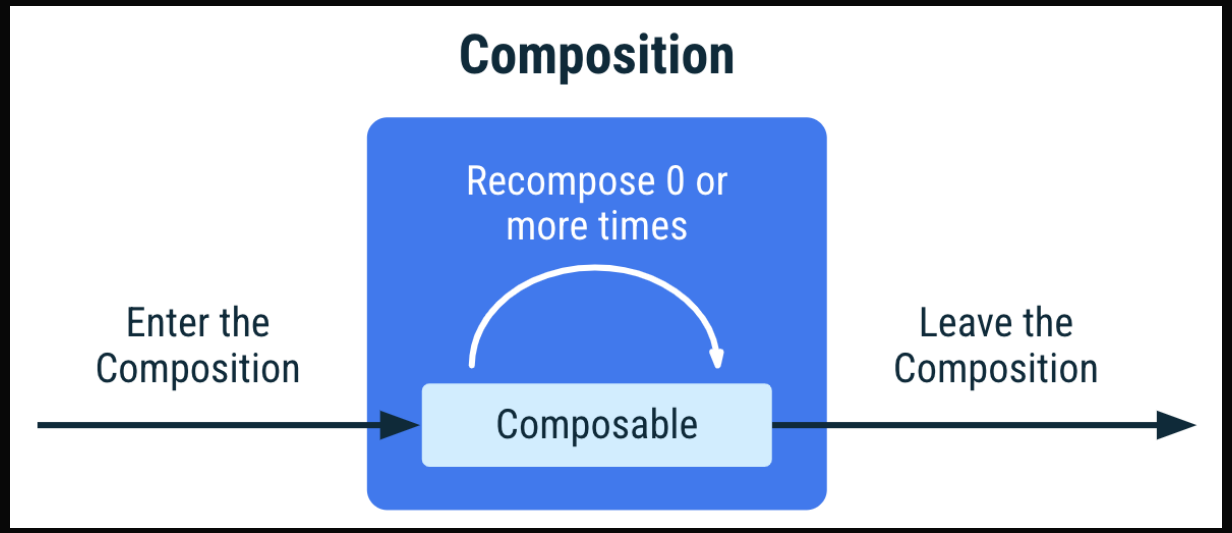
SwiftUI lifecycle

```
final class LifecycleMonitor {
  init() {
    print(#function)
  }

  deinit {
    print(#function)
  }
}

struct LifecycleView: View {
  let lifecycleMonitor = LifecycleMonitor()

  var body: some View {
    Text("Hello")
  }
}
```

SwiftUI lifecycle

```
struct LifecycleView: View {
  var body: some View {
    Text("Hello")
      .onAppear {
        print("onAppear")
      }
      .onDisappear {
        print("onDisappear")
      }
  }
}
```

Compose lifecycle



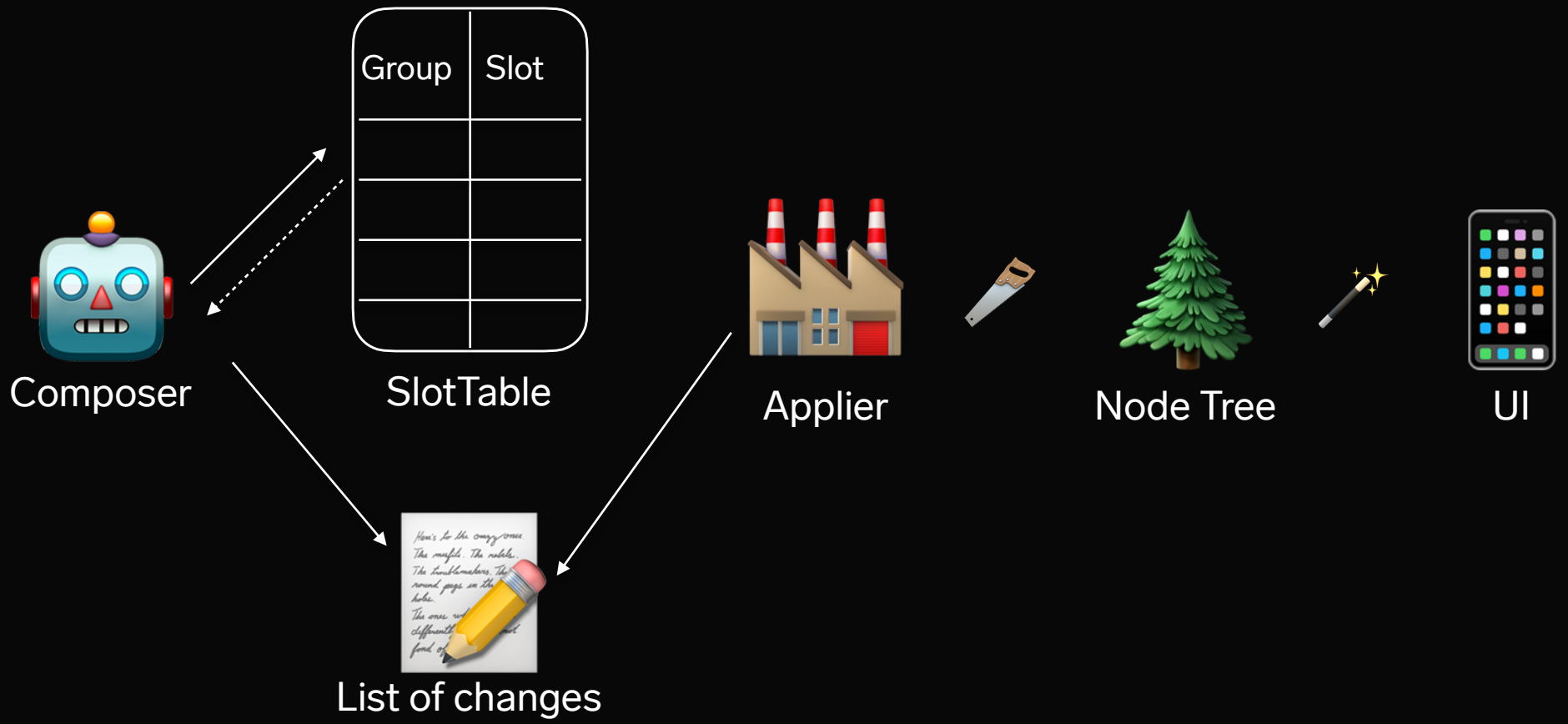
Compose lifecycle

```
@Composable
fun ComposeLifecycle() {
    DisposableEffect(Unit) {
        println("init")
        onDispose {
            println("onDispose")
        }
    }
}
```

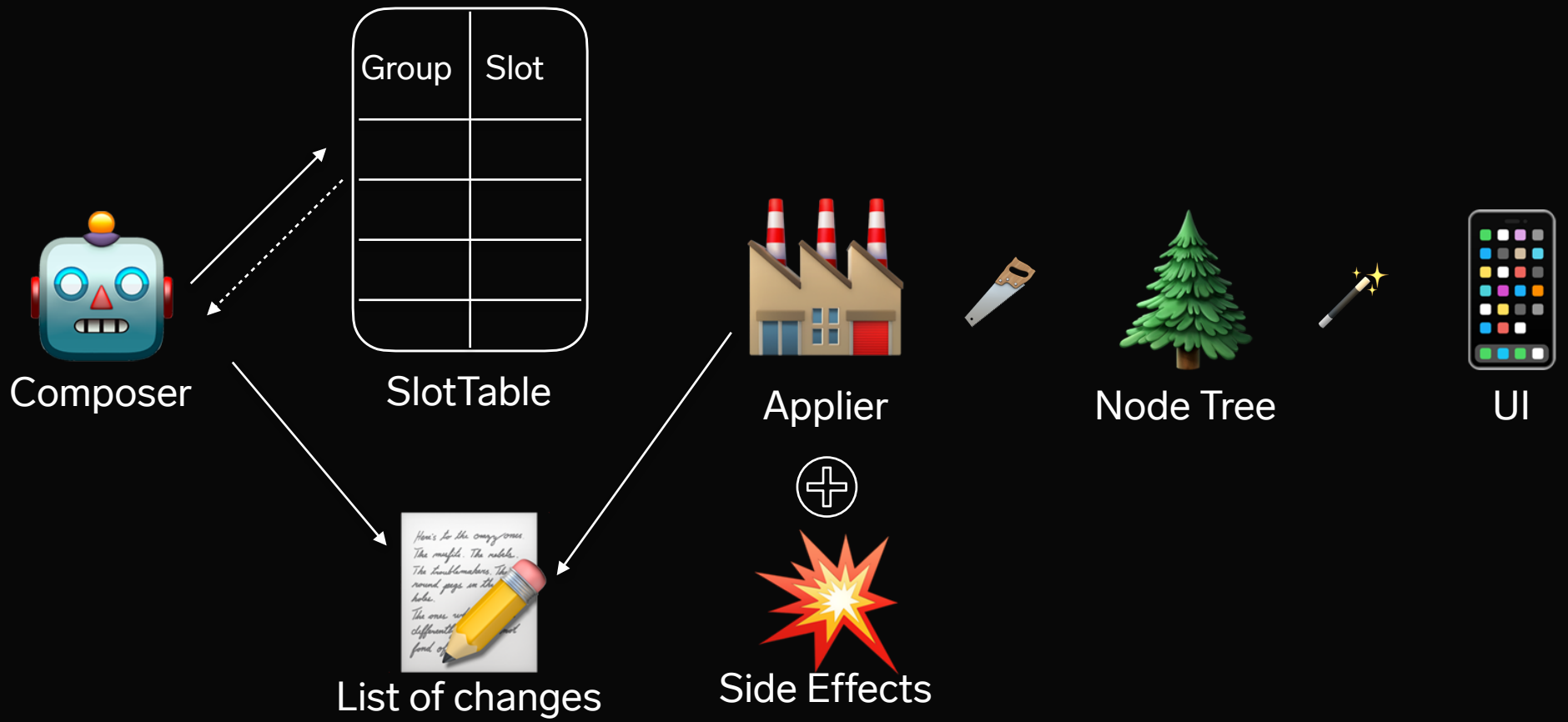
DisposableEffect

```
@Composable
@NonRestartableComposable
fun DisposableEffect(
    key1: Any?,
    effect: DisposableEffectScope.() → DisposableEffectResult
) {
    remember(key1) { DisposableEffectImpl(effect) }
}
```

Compose под капотом



Compose под капотом



Apply changes

```
private fun applyChangesInLocked(changes: ChangeList) {
    val manager = RememberEventDispatcher(abandonSet)
    try {
        if (changes.isEmpty()) return
        applier.onBeginChanges()
        slotTable.write { slots →
            changes.executeAndFlushAllPendingChanges(applier, slots, manager)
        }
        applier.onEndChanges()

        manager.dispatchRememberObservers()
        manager.dispatchSideEffects()
    } finally {
        if (this.lateChanges.isEmpty())
            manager.dispatchAbandons()
    }
}
```

DisposableEffect

```
private class DisposableEffectImpl(
    private val effect: DisposableEffectScope.() → DisposableEffectResult
) : RememberObserver {
    private var onDispose: DisposableEffectResult? = null

    override fun onRemembered() {
        onDispose = InternalDisposableEffectScope.effect()
    }

    override fun onForgotten() {
        onDispose?.dispose()
        onDispose = null
    }

    override fun onAbandoned() {}
}
```

BackHandler

```
@Composable
fun BackHandler(onBack: () → Unit) {
    val backCallback = remember {
        object : OnBackPressedCallback(enabled = true) {
            override fun handleOnBackPressed() {
                onBack()
            }
        }
    }
    val backDispatcher = LocalOnBackPressedDispatcherOwner.current!!.onBackPressedDispatcher
    val lifecycleOwner = LocalLifecycleOwner.current
    DisposableEffect(lifecycleOwner, backDispatcher) {
        backDispatcher.addCallback(lifecycleOwner, backCallback)
        onDispose { backCallback.remove() }
    }
}
```

BackHandler

```
@Composable
fun BackHandler(onBack: () → Unit) {
    val backCallback = remember {
        object : OnBackPressedCallback(enabled = true) {
            override fun handleOnBackPressed() {
                onBack()
            }
        }
    }
    val backDispatcher = LocalOnBackPressedDispatcherOwner.current!!.onBackPressedDispatcher
    val lifecycleOwner = LocalLifecycleOwner.current
    DisposableEffect(lifecycleOwner, backDispatcher) {
        backDispatcher.addCallback(lifecycleOwner, backCallback)
        onDispose { backCallback.remove() }
    }
}
```

BackHandler

```
@Composable
fun BackHandler(onBack: () → Unit) {
    val backCallback = remember {
        object : OnBackPressedCallback(enabled = true) {
            override fun handleOnBackPressed() {
                onBack()
            }
        }
    }
    val backDispatcher = LocalOnBackPressedDispatcherOwner.current!!.onBackPressedDispatcher
    val lifecycleOwner = LocalLifecycleOwner.current
    DisposableEffect(lifecycleOwner, backDispatcher) {
        backDispatcher.addCallback(lifecycleOwner, backCallback)
        onDispose { backCallback.remove() }
    }
}
```

BackHandler

```
@Composable
fun BackHandler(onBack: () → Unit) {
    val backCallback = remember {
        object : OnBackPressedCallback(enabled = true) {
            override fun handleOnBackPressed() {
                onBack()
            }
        }
    }
    val backDispatcher = LocalOnBackPressedDispatcherOwner.current!!.onBackPressedDispatcher
    val lifecycleOwner = LocalLifecycleOwner.current
    DisposableEffect(lifecycleOwner, backDispatcher) {
        backDispatcher.addCallback(lifecycleOwner, backCallback)
        onDispose { backCallback.remove() }
    }
}
```

BackHandler

```
@Composable
fun BackHandler(onBack: () → Unit) {
    val currentOnBack by rememberUpdatedState(onBack)
    val backCallback = remember {
        object : OnBackPressedCallback(enabled = true) {
            override fun handleOnBackPressed() {
                currentOnBack()
            }
        }
    }
    val backDispatcher = LocalOnBackPressedDispatcherOwner.current!!.onBackPressedDispatcher
    val lifecycleOwner = LocalLifecycleOwner.current
    DisposableEffect(lifecycleOwner, backDispatcher) {
        backDispatcher.addCallback(lifecycleOwner, backCallback)
        onDispose { backCallback.remove() }
    }
}
```

BackHandler

```
@Composable
fun BackHandler(onBack: () → Unit) {
    val currentOnBack by rememberUpdatedState(onBack)
    val backCallback = remember {
        object : OnBackPressedCallback(enabled = true) {
            override fun handleOnBackPressed() {
                currentOnBack()
            }
        }
    }
    val backDispatcher = LocalOnBackPressedDispatcherOwner.current!!.onBackPressedDispatcher
    val lifecycleOwner = LocalLifecycleOwner.current
    DisposableEffect(lifecycleOwner, backDispatcher) {
        backDispatcher.addCallback(lifecycleOwner, backCallback)
        onDispose { backCallback.remove() }
    }
}
```

BackHandler

```
@Composable
fun BackHandler(onBack: () → Unit, enabled: Boolean) {
    val currentOnBack by rememberUpdatedState(onBack)
    val backCallback = remember {
        object : OnBackPressedCallback(enabled = enabled) {
            override fun handleOnBackPressed() {
                currentOnBack()
            }
        }
    }
    val backDispatcher = LocalOnBackPressedDispatcherOwner.current!!.onBackPressedDispatcher
    val lifecycleOwner = LocalLifecycleOwner.current
    DisposableEffect(lifecycleOwner, backDispatcher) {
        backDispatcher.addCallback(lifecycleOwner, backCallback)
        onDispose { backCallback.remove() }
    }
}
```

BackHandler

```
@Composable
fun BackHandler(onBack: () → Unit, enabled: Boolean) {
    val currentOnBack by rememberUpdatedState(onBack)
    val backCallback = remember {
        object : OnBackPressedCallback(enabled = enabled) {
            override fun handleOnBackPressed() {
                currentOnBack()
            }
        }
    }
    SideEffect {
        backCallback.isEnabled = enabled
    }
    val backDispatcher = LocalOnBackPressedDispatcherOwner.current!!.onBackPressedDispatcher
    val lifecycleOwner = LocalLifecycleOwner.current
    DisposableEffect(lifecycleOwner, backDispatcher) { … }
}
```

Ключевые отличия в ЖЦ

SwiftUI

- ЖЦ можно отследить через фазы сцены
- Используются модификаторы onAppear/onDisappear для реагирования на события ЖЦ вьюшки

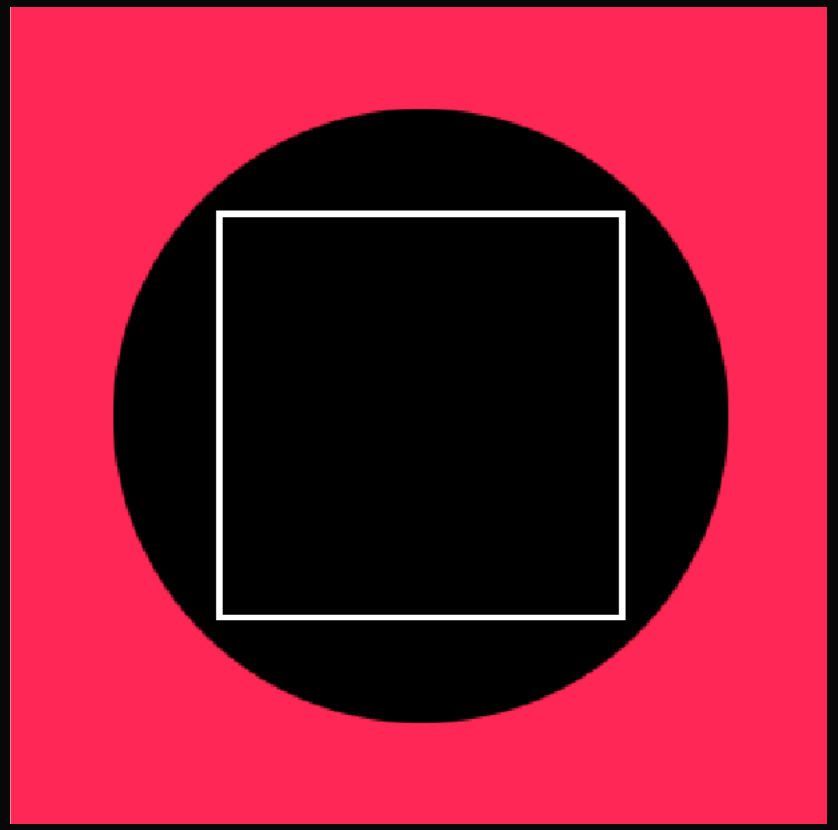
Compose

- ЖЦ можно отследить через колбэки Activity и LifecycleEventEffect
- Используется SideEffect API для реагирования на события ЖЦ Composable функции

Система модификаторов

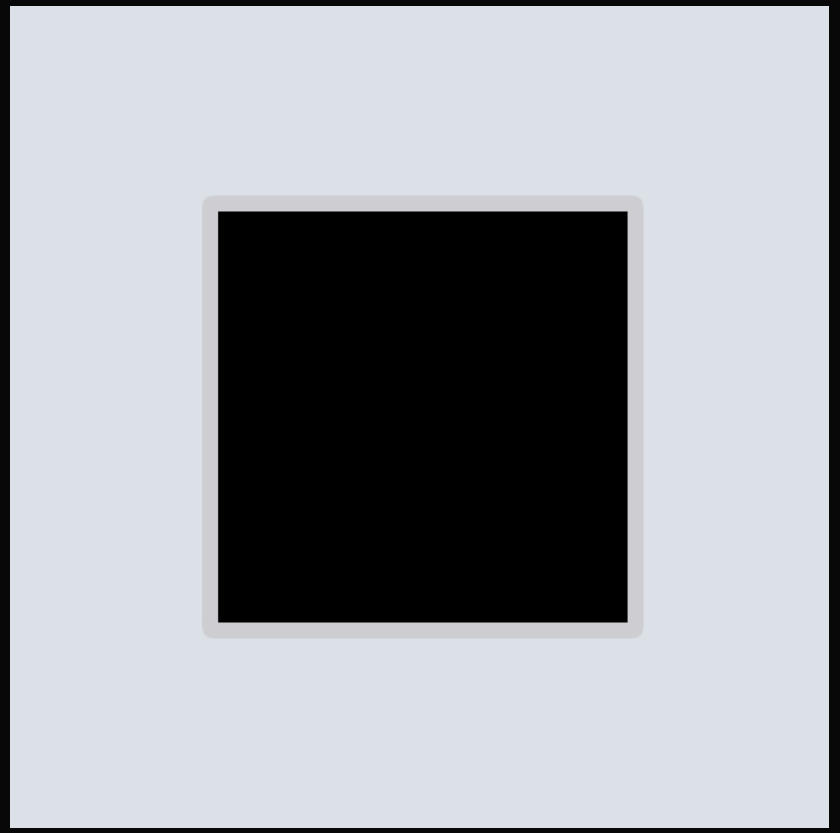
Модификаторы SwiftUI

```
struct SquidPlayer: View {
  var body: some View {
    // TODO
  }
}
```



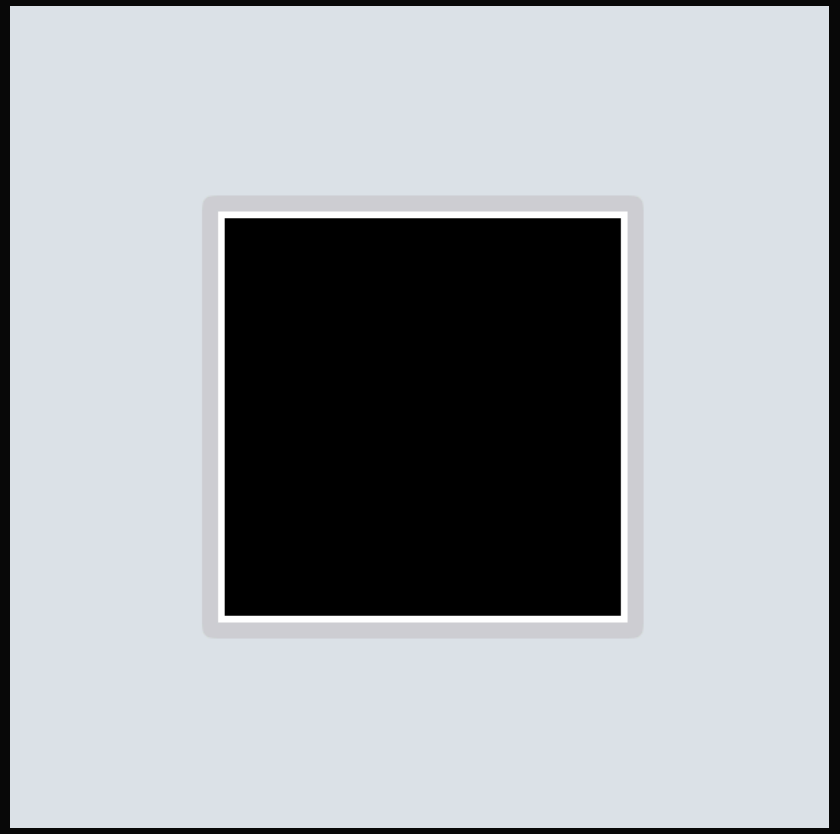
Модификаторы SwiftUI

```
struct SquidPlayer: View {
  var body: some View {
    Rectangle()
      .frame(width: 64, height: 64)
  }
}
```



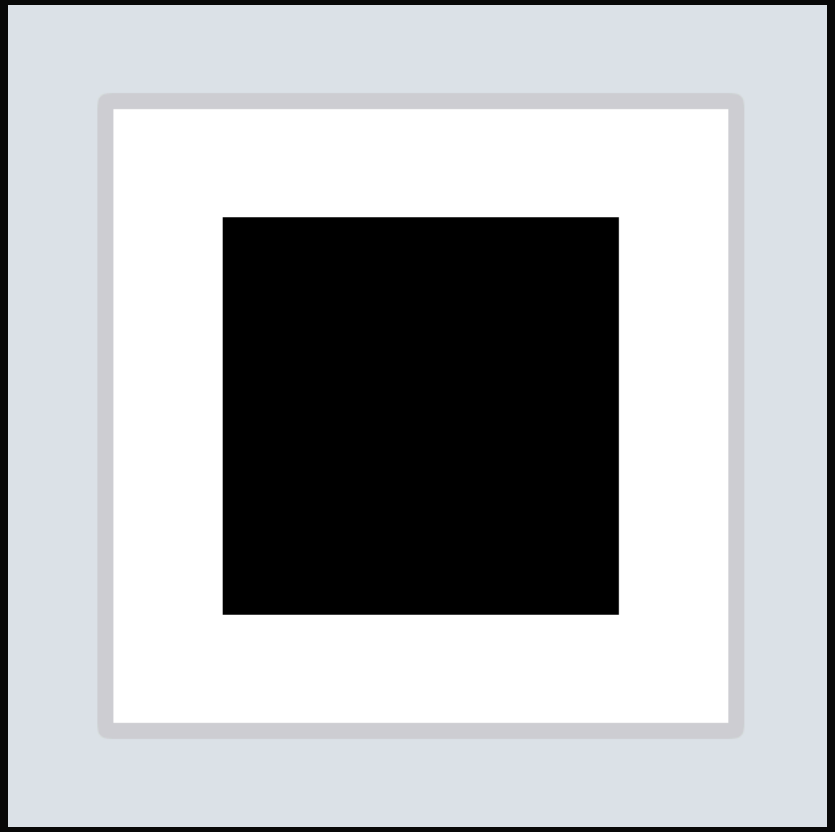
Модификаторы SwiftUI

```
struct SquidPlayer: View {
  var body: some View {
    Rectangle()
      .frame(width: 64, height: 64)
      .border(.white, width: 1)
  }
}
```



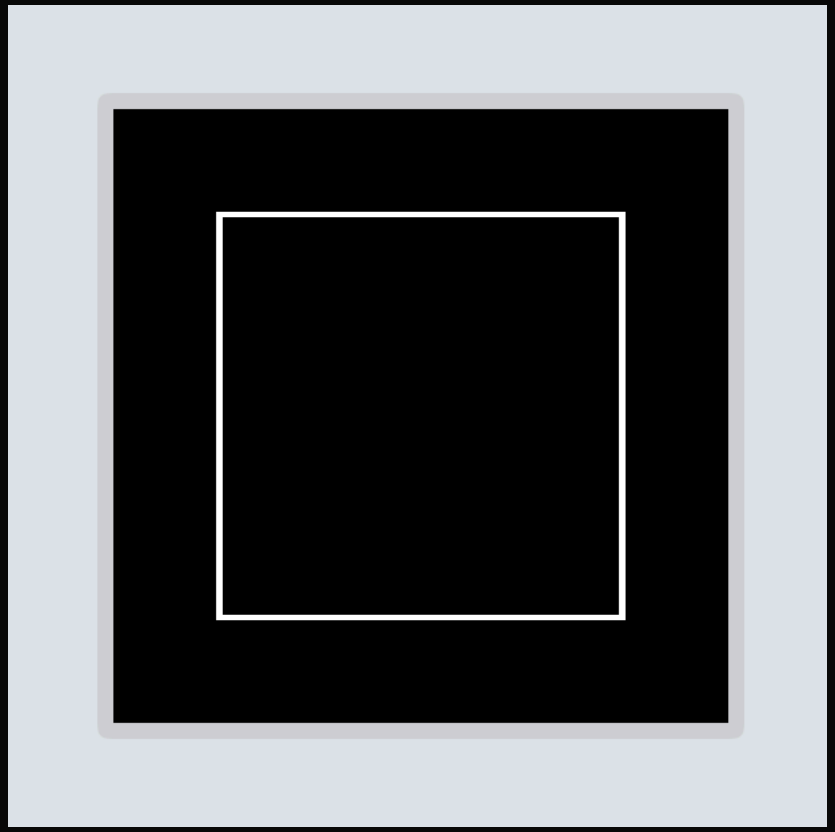
Модификаторы SwiftUI

```
struct SquidPlayer: View {
  var body: some View {
    Rectangle()
      .frame(width: 64, height: 64)
      .border(.white, width: 1)
      .padding(16)
  }
}
```



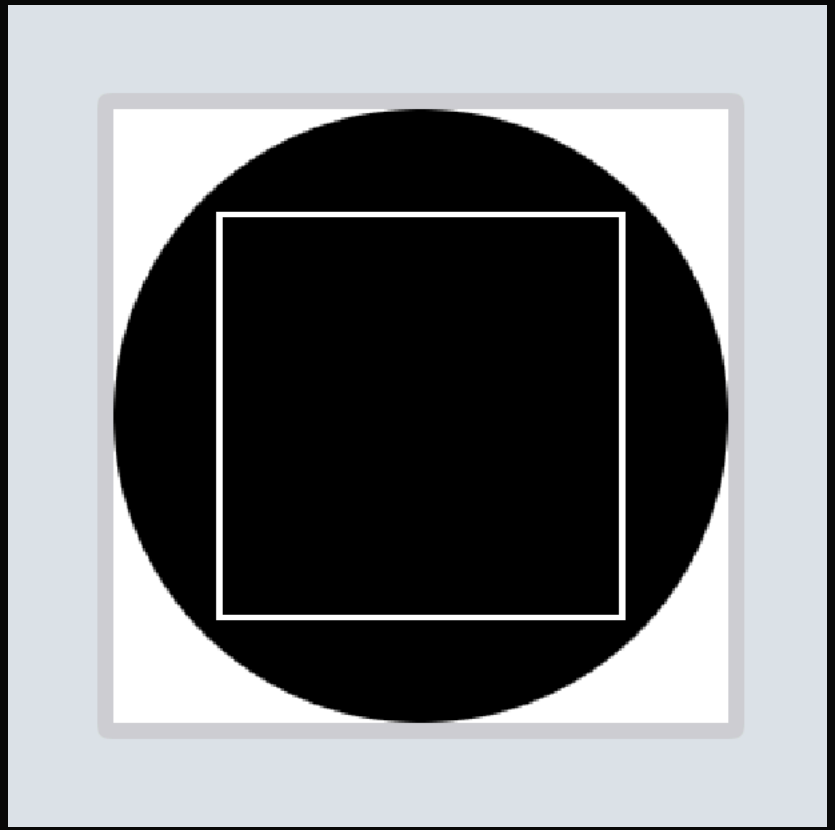
Модификаторы SwiftUI

```
struct SquidPlayer: View {
  var body: some View {
    Rectangle()
      .frame(width: 64, height: 64)
      .border(.white, width: 1)
      .padding(16)
      .background(.black)
  }
}
```



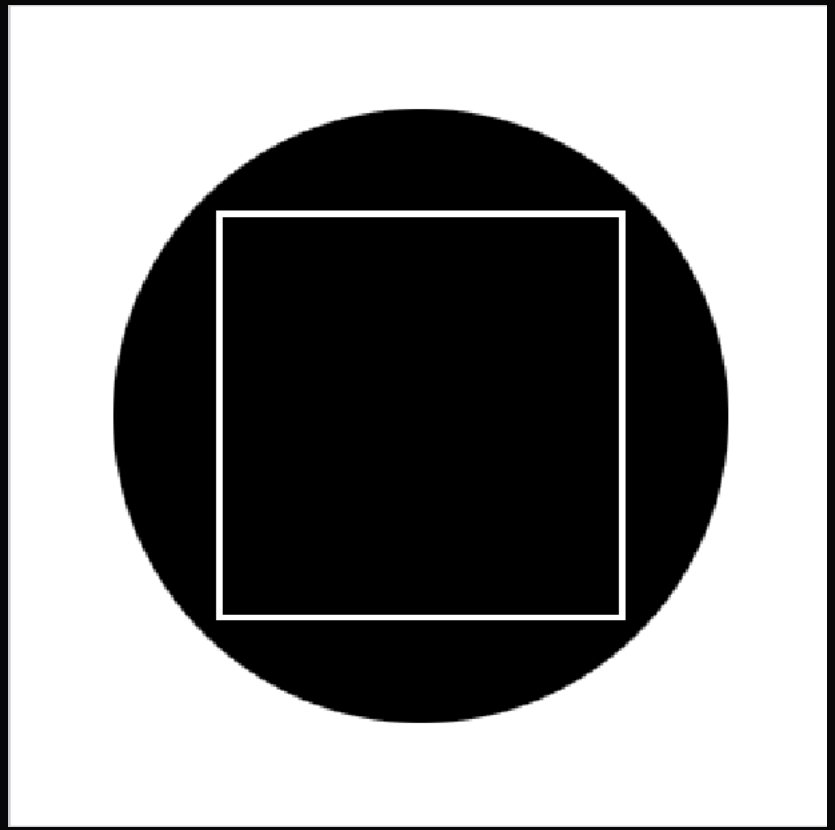
Модификаторы SwiftUI

```
struct SquidPlayer: View {
  var body: some View {
    Rectangle()
      .frame(width: 64, height: 64)
      .border(.white, width: 1)
      .padding(16)
      .background(.black)
      .clipShape(Circle())
  }
}
```



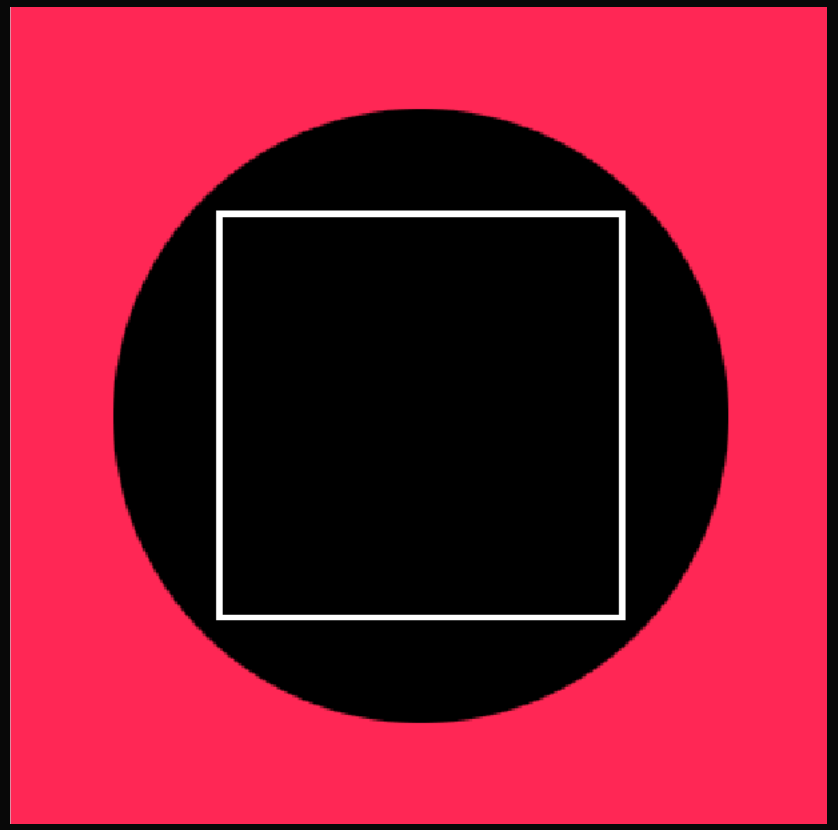
Модификаторы SwiftUI

```
struct SquidPlayer: View {
  var body: some View {
    Rectangle()
      .frame(width: 64, height: 64)
      .border(.white, width: 1)
      .padding(16)
      .background(.black)
      .clipShape(Circle())
      .padding(16)
  }
}
```



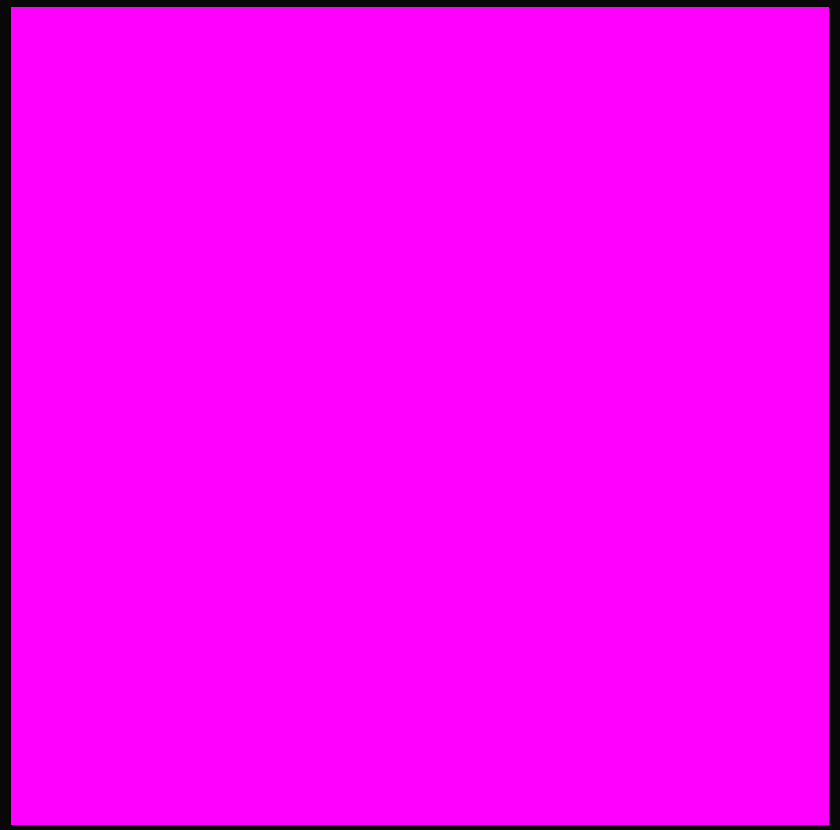
Модификаторы SwiftUI

```
struct SquidPlayer: View {
  var body: some View {
    Rectangle()
      .frame(width: 64, height: 64)
      .border(.white, width: 1)
      .padding(16)
      .background(.black)
      .clipShape(Circle())
      .padding(16)
      .background(.pink)
  }
}
```



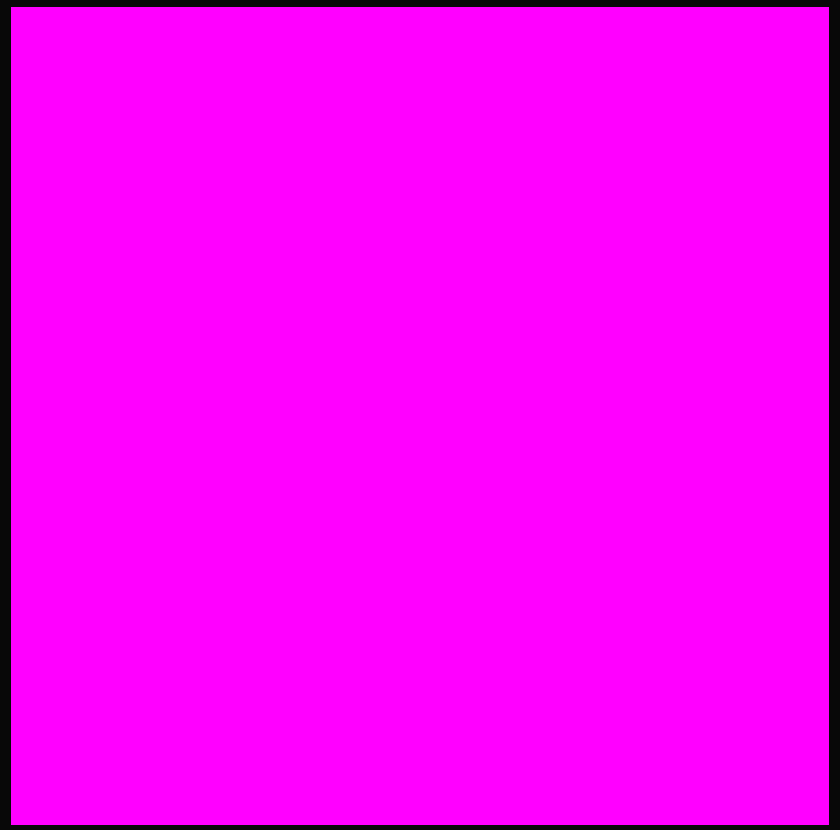
Модификаторы Compose

```
@Composable
fun SquidPlayer() {
    Box(
        Modifier
            .size(128.dp)
            .background(Color.Magenta)
    )
}
```



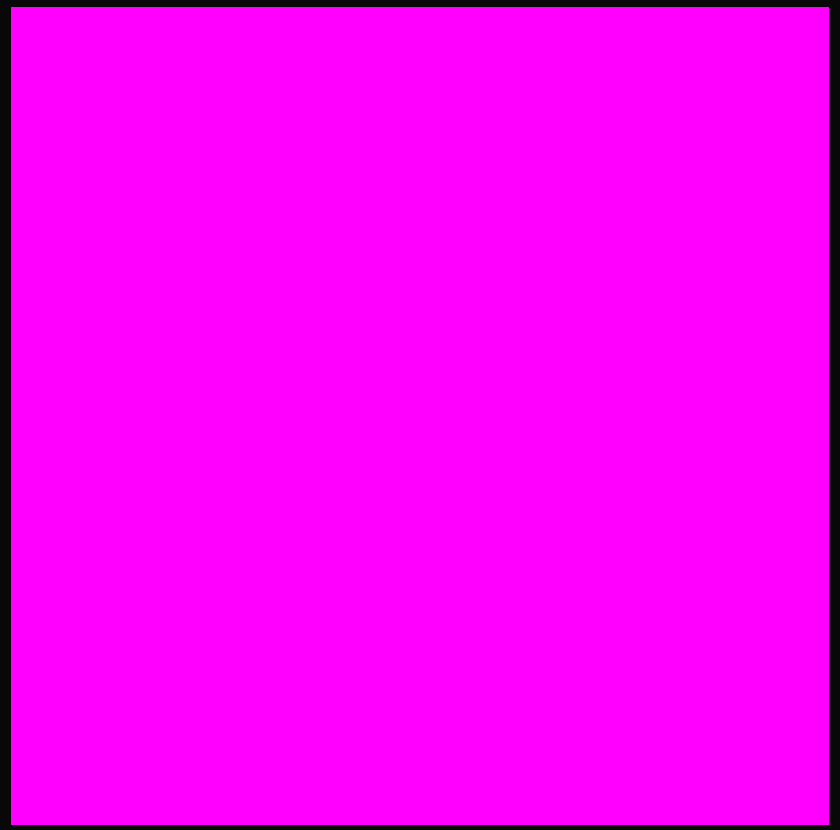
Модификаторы Compose

```
@Composable
fun SquidPlayer() {
    Box(
        Modifier
            .size(128.dp)
            .background(Color.Magenta)
            .padding(16.dp)
    )
}
```



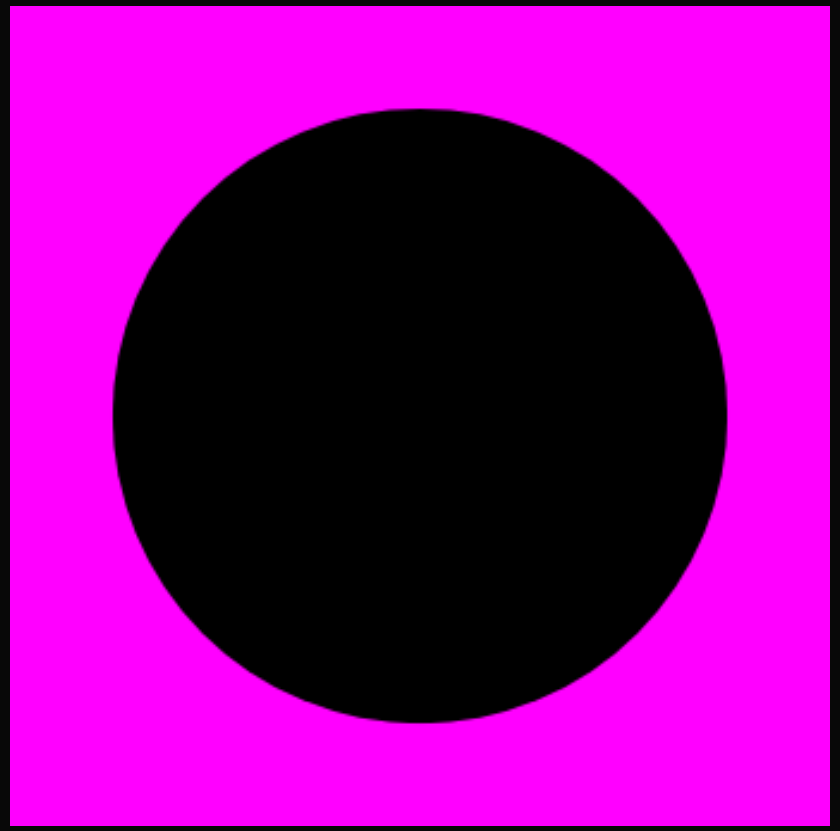
Модификаторы Compose

```
@Composable
fun SquidPlayer() {
    Box(
        Modifier
            .size(128.dp)
            .background(Color.Magenta)
            .padding(16.dp)
            .clip(CircleShape)
    )
}
```



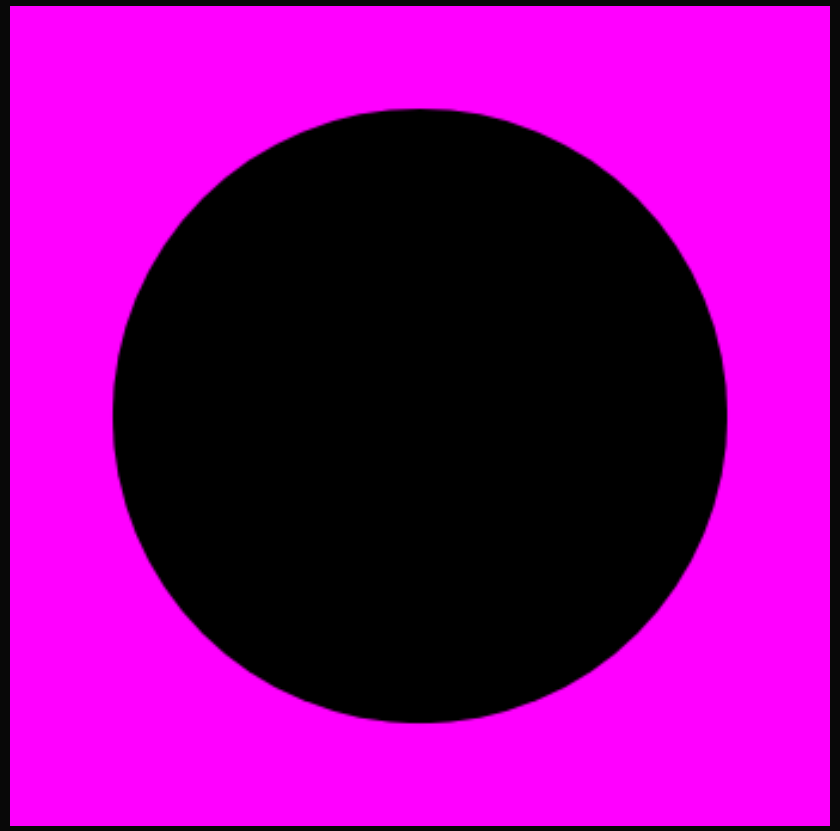
Модификаторы Compose

```
@Composable
fun SquidPlayer() {
    Box(
        Modifier
            .size(128.dp)
            .background(Color.Magenta)
            .padding(16.dp)
            .clip(CircleShape)
            .background(Color.Black)
    )
}
```



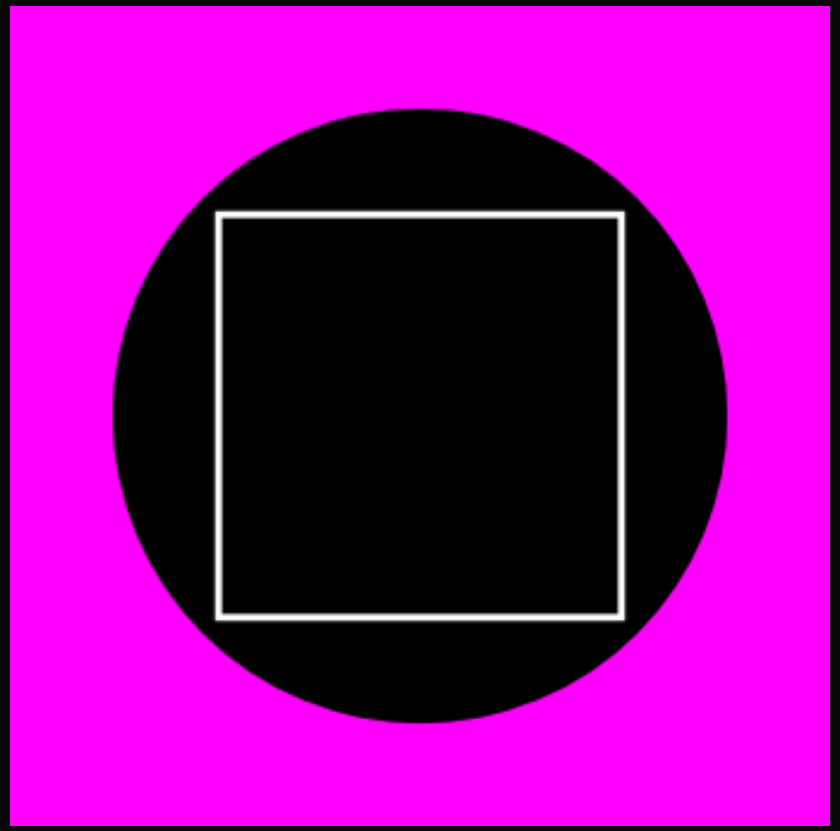
Модификаторы Compose

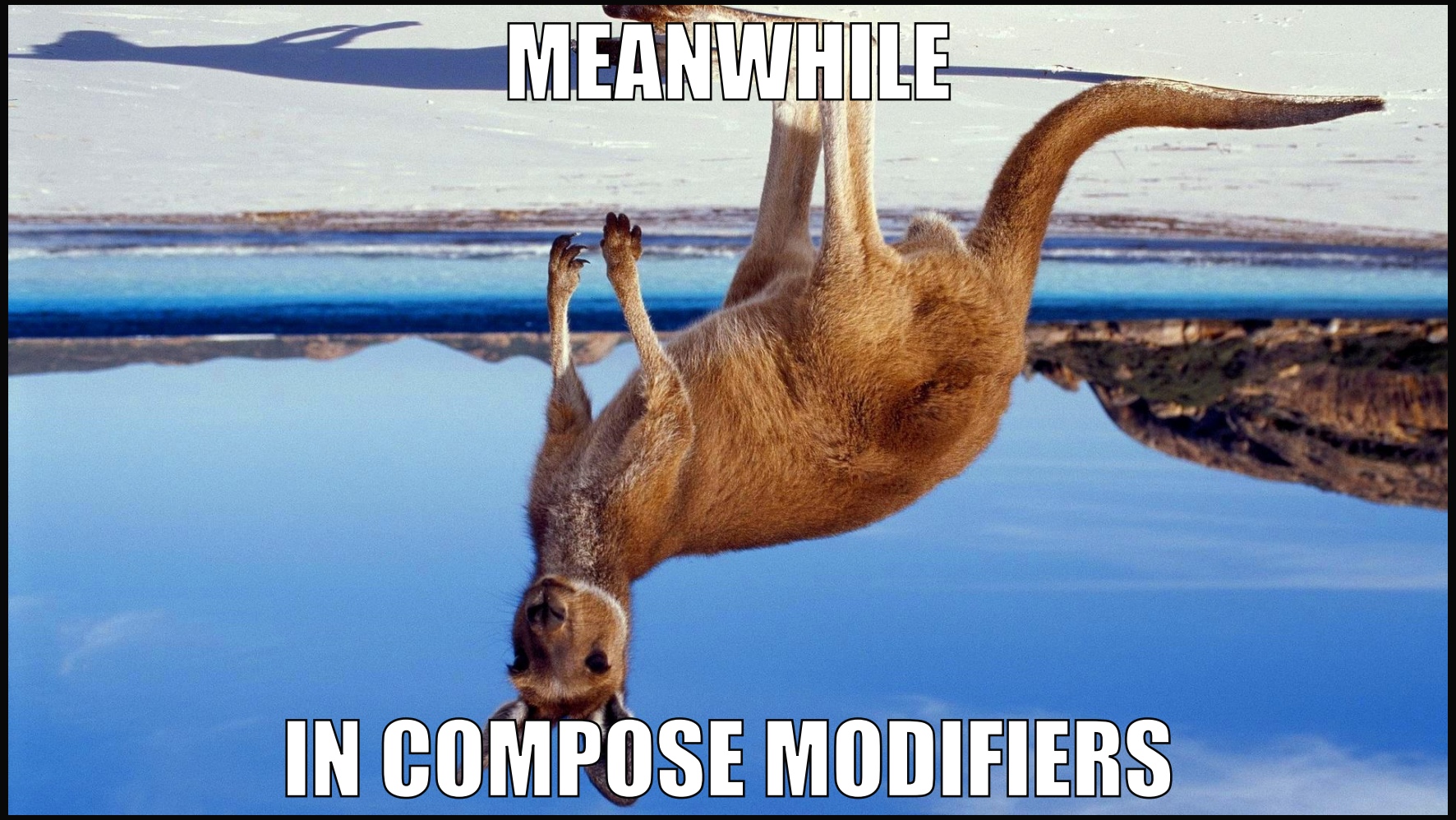
```
@Composable
fun SquidPlayer() {
    Box(
        Modifier
            .size(128.dp)
            .background(Color.Magenta)
            .padding(16.dp)
            .clip(CircleShape)
            .background(Color.Black)
            .padding(16.dp)
    )
}
```



Модификаторы Compose

```
@Composable
fun SquidPlayer() {
    Box(
        Modifier
            .size(128.dp)
            .background(Color.Magenta)
            .padding(16.dp)
            .clip(CircleShape)
            .background(Color.Black)
            .padding(16.dp)
            .border(1.dp, Color.White)
    )
}
```




MEANWHILE
IN COMPOSE MODIFIERS

Кастомные модификаторы

```
Text("Hello SwiftUI")
    .skeleton(.red)
```

Кастомные модификаторы

```
Text("Hello SwiftUI")
    .skeleton(.red)

extension View {
  func skeleton(_ color: Color) -> some View {
    self.modifier(SkeletonModifier(color: color))
  }
}
```

Кастомные модификаторы

```
Text("Hello SwiftUI")
    .skeleton(.red)

extension View {
  func skeleton(_ color: Color) -> some View {
    self.modifier(SkeletonModifier(color: color))
  }
}

private struct SkeletonModifier: ViewModifier {
  let color: Color

  func body(content: Content) -> some View {
    content
      .background(color)
      .foregroundColor(color)
  }
}
```

Кастомные модификаторы

```
Text("Hello Compose", Modifier.skeleton(Color.Red))
```

Кастомные модификаторы

```
Text("Hello Compose", Modifier.skeleton(Color.Red))

fun Modifier.skeleton(color: Color) = this then SkeletonElement(color)
```

Кастомные модификаторы

```
Text("Hello Compose", Modifier.skeleton(Color.Red))

fun Modifier.skeleton(color: Color) = this then SkeletonElement(color)

private data class SkeletonElement(val color: Color) : ModifierNodeElement<SkeletonNode>() {
    override fun create() = SkeletonNode(color)

    override fun update(node: SkeletonNode) {
        node.color = color
    }
}
```

Кастомные модификаторы

```
Text("Hello Compose", Modifier.skeleton(Color.Red))

fun Modifier.skeleton(color: Color) = this then SkeletonElement(color)

private data class SkeletonElement(val color: Color) : ModifierNodeElement<SkeletonNode>() {
    override fun create() = SkeletonNode(color)

    override fun update(node: SkeletonNode) {
        node.color = color
    }
}

private class SkeletonNode(var color: Color) : DrawModifierNode, Modifier.Node() {
    override fun ContentDrawScope.draw() {
        drawRect(color)
    }
}
```

Область видимости модификаторов

```swift
// Применяется к любой вьюшке
extension View {
  func skeleton(_ color: Color) -> some View {
    self.modifier(SkeletonModifier(color: color))
  }
}

// Применяется только к тексту
extension Text {
  func skeleton(_ color: Color) -> some View {
    self.modifier(SkeletonModifier(color: color))
  }
}
```

Область видимости модификаторов

```
Box(Modifier.fillMaxSize()) { // BoxScope
    SkeletonText(Modifier.align(Alignment.Center))
}

interface BoxScope {
    @Stable
    fun Modifier.align(alignment: Alignment): Modifier

    @Stable
    fun Modifier.matchParentSize(): Modifier
}
```

Ключевые отличия модификаторов

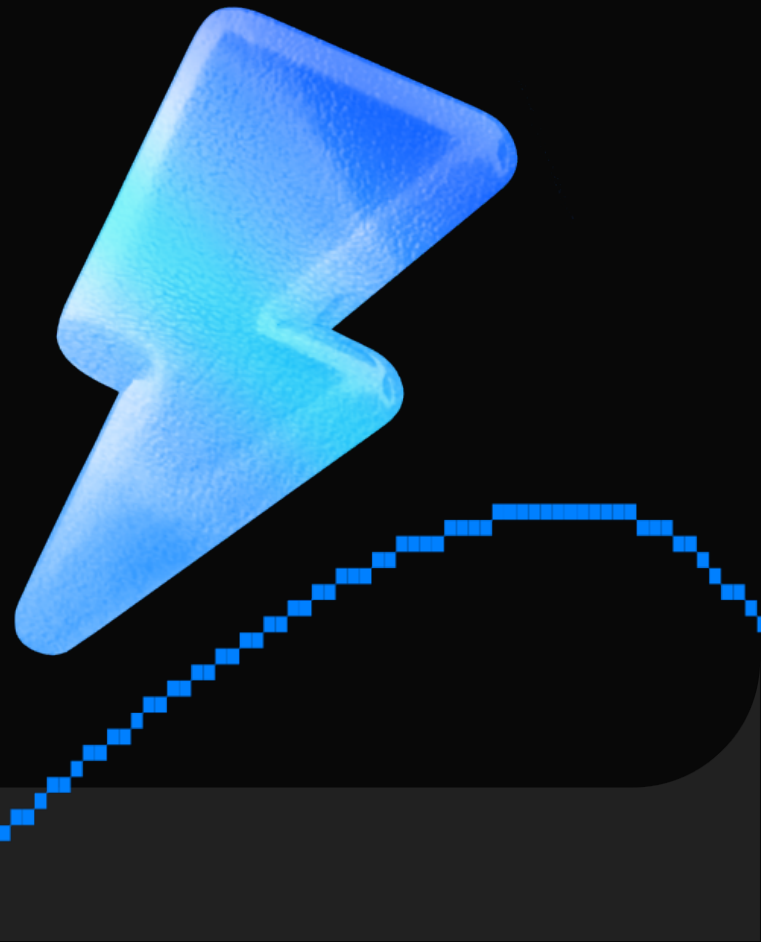
SwiftUI

- Модификаторы применяются прямо ко View
- Возвращают новую вьюшку
- Применяются «во вне»
- Ограничиваются типами

Compose

- Модификатор передается как отдельный параметр
- Возвращает новый модификатор
- Применяются «во внутрь»
- Ограничены скоупами

Корутины в Compose и SwiftUI



Асинхронные операции в Compose

- Анимации / жесты
- Показ snackbar / bottom sheet
- Искусственные задержки

Создание CoroutineScope

```
@Composable
fun CoroutineScopeSample() {
    val scope = rememberCoroutineScope()
    Button(onClick = {
        scope.launch {
            delay(1000)
            println("Hello coroutine scope")
        }
    }) {
        Text(text = "Print with delay")
    }
}
```

Создание CoroutineScope

```
@Composable
fun LaunchedEffectSample() {
    LaunchedEffect(Unit) {
        delay(1000)
        println("Hello coroutine scope")
    }
}
```

Асинхронные операции в SwiftUI

- Модификатор refreshable
- Искусственные задержки

Создание Task

```
struct TaskSample: View {
  var body: some View {
    EmptyView()
      .onAppear {
        Task {
          try! await Task.sleep(nanoseconds: 1_000_000_000)
          print("Hello Task")
        }
      }
  }
}
```

Создание Task

```
struct TaskSample: View {
  var body: some View {
    EmptyView()
      .task {
        try! await Task.sleep(nanoseconds: 1_000_000_000)
        print("Hello Task")
      }
  }
}
```

Корутины в Kotlin и Swift

Описание	Kotlin	Swift
Входная точка	CoroutineScope	Task

Корутины в Kotlin и Swift

Описание	Kotlin	Swift
Входная точка	CoroutineScope	Task
ЖЦ корутины	Job	Task

Корутины в Kotlin и Swift

Описание	Kotlin	Swift
Входная точка	CoroutineScope	Task
ЖЦ корутины	Job	Task
Асинхронная функция	suspend	async

Корутины в Kotlin и Swift

Описание	Kotlin	Swift
Входная точка	CoroutineScope	Task
ЖЦ корутины	Job	Task
Асинхронная функция	suspend	async
Контекст выполнения	Dispatcher	Actor

Корутины в Kotlin и Swift

Описание	Kotlin	Swift
Входная точка	CoroutineScope	Task
ЖЦ корутины	Job	Task
Асинхронная функция	suspend	async
Контекст выполнения	Dispatcher	Actor
Ожидание результата	Неявно	await

Корутины в Kotlin и Swift

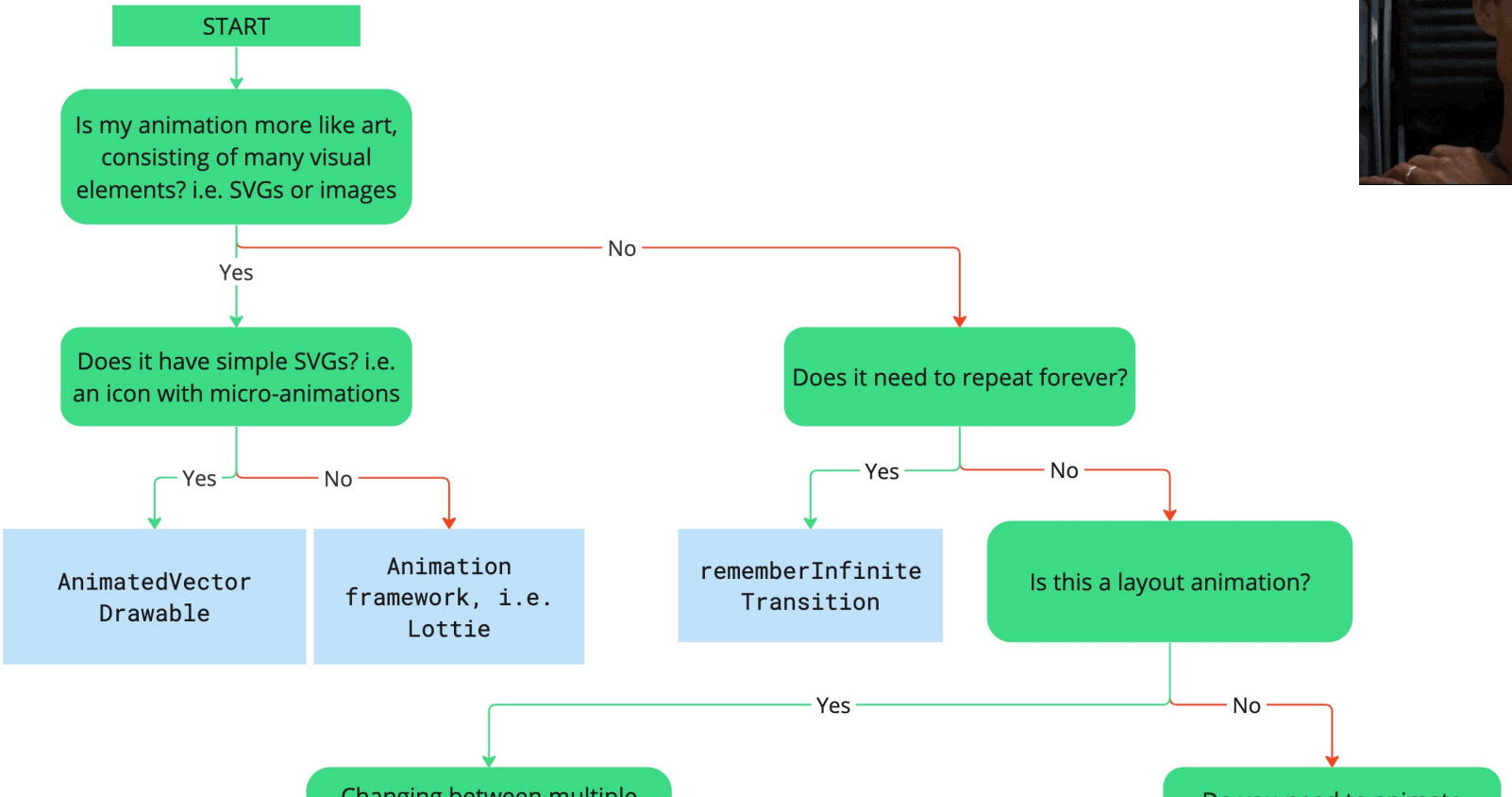
Описание	Kotlin	Swift
Входная точка	CoroutineScope	Task
ЖЦ корутины	Job	Task
Асинхронная функция	suspend	async
Контекст выполнения	Dispatcher	Actor
Ожидание результата	Неявно	await
Параллелизм	async/await	async/let

Устройство Анимаций

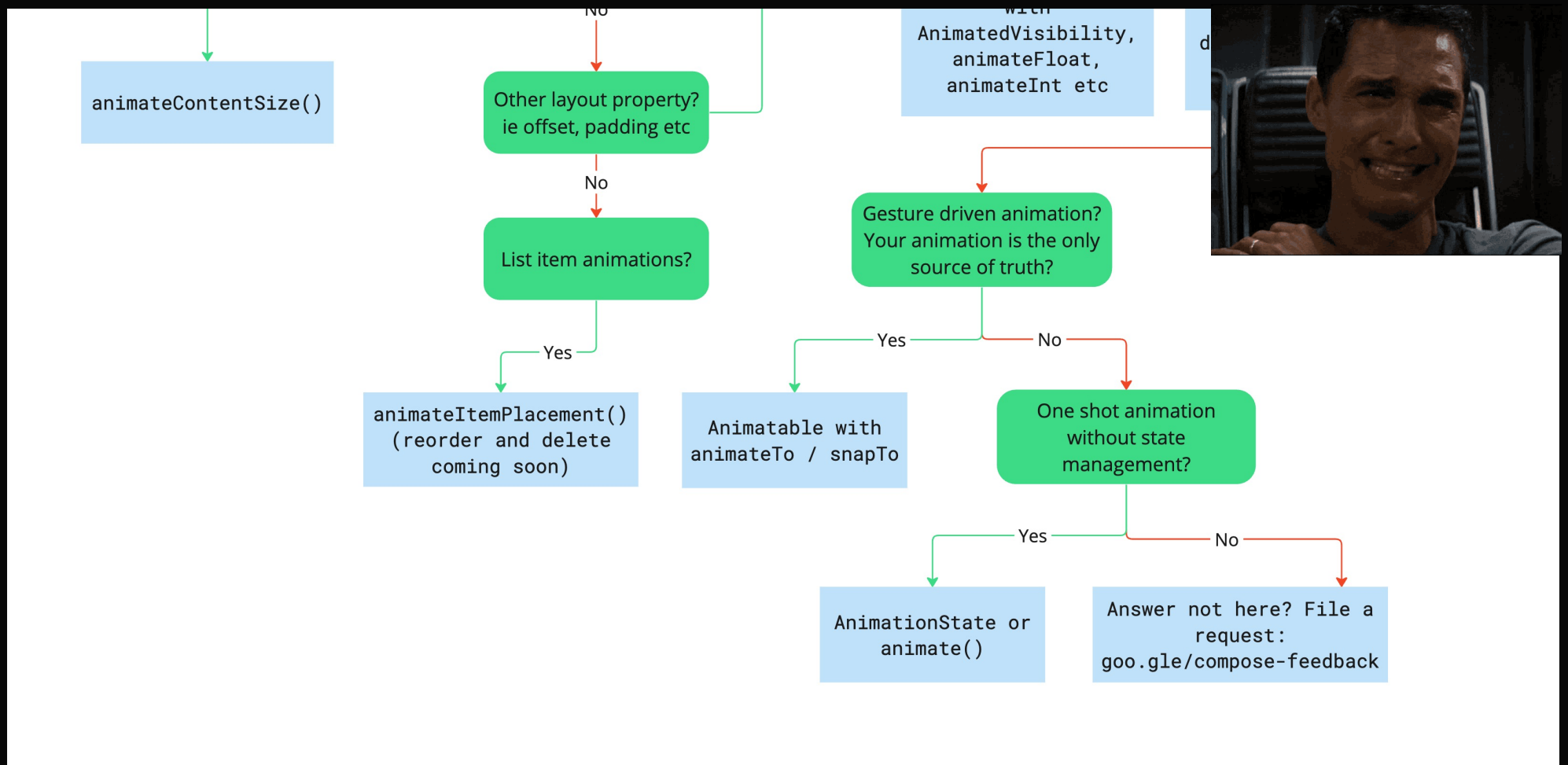
Анимации в SwiftUI

- Модификатор animation
- Функция withAnimation
- Модификатор transition

Анимации в Compose



Анимации в Compose



Анимация появления

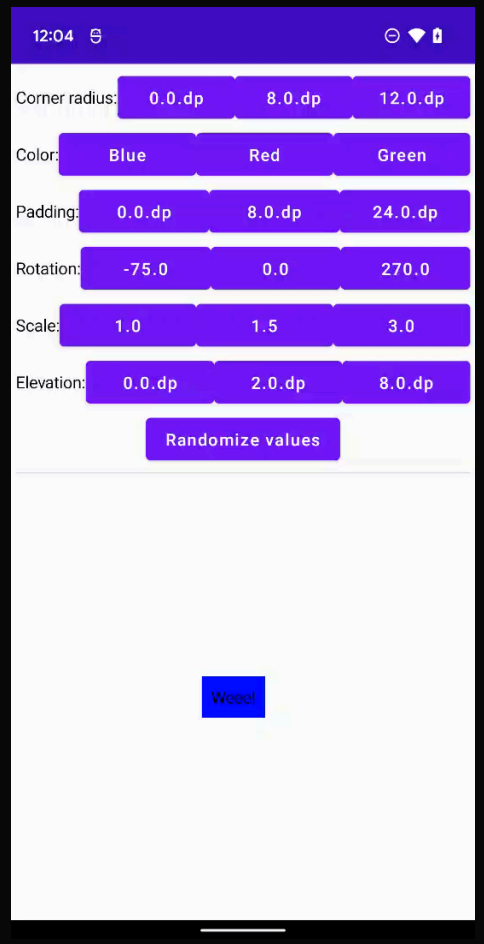
```
@Composable
fun VisibilitySample(modifier: Modifier = Modifier) {
    var showDetails by remember { mutableStateOf(false) }

    Column(modifier) {
        AnimatedVisibility(visible = showDetails) {
            Text("Details")
        }
        Button(onClick = { showDetails = showDetails.not() }) {
            Text("Show/Hide details")
        }
    }
}
```

Анимация появления

```
struct VisibilitySample: View {
  @State private var showDetails = false

  var body: some View {
    VStack {
      if showDetails {
        Text("Details")
      }
      Button("Show details") {
        withAnimation {
          showDetails.toggle()
        }
      }
    }
  }
}
```

Аналог withAnimation в Compose

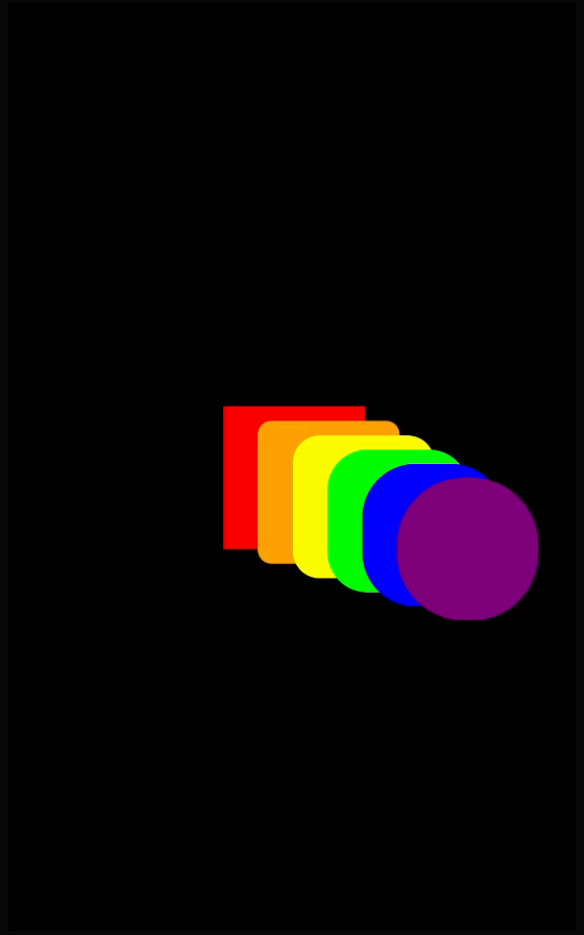
```
var scale by remember { mutableStateOf(1f) }
var elevation by remember { mutableStateOf(0.dp) }

with(rememberAutoTransition()) {
    Button(onClick = {
        withAnimation {
            scale = 0.5f
            elevation = 8.dp
        }
    }) { /* … */ }
}
```



https://github.com/zach-klippenstein/compose-autotransition

Бесконечная анимация

```
@Composable
fun StackedSpring() {
    val colors = listOf(Color.Red, Color(0xFFFFA500), Color.Yellow, Color.Green, Color.Blue, Color(0xFF800080))
    val animationSpec = rememberInfiniteTransition()

    val offsetX by animationSpec.animateFloat(
        initialValue = -25f,
        targetValue = 25f,
        animationSpec = infiniteRepeatable(
            animation = tween(durationMillis = 1000, easing = EaseInOut),
            repeatMode = RepeatMode.Reverse
        )
    )

    val cornerRadius by animationSpec.animateFloat(
        initialValue = 1f,
        targetValue = 25f,
        animationSpec = infiniteRepeatable(
            animation = tween(durationMillis = 1000, easing = EaseInOut),
            repeatMode = RepeatMode.Reverse
        )
    )

    Box(modifier = Modifier.fillMaxSize(), contentAlignment = Alignment.Center) {
        colors.forEachIndexed { index, color →
            Box(
                modifier = Modifier
                    .size(100.dp)
                    .offset(
                        x = Dp(offsetX * index),
                        y = Dp(10f * index)
                    )
                    .background(
                        color = color,
                        shape = RoundedCornerShape(cornerRadius * index)
                    )
            )
        }
    }
}
```

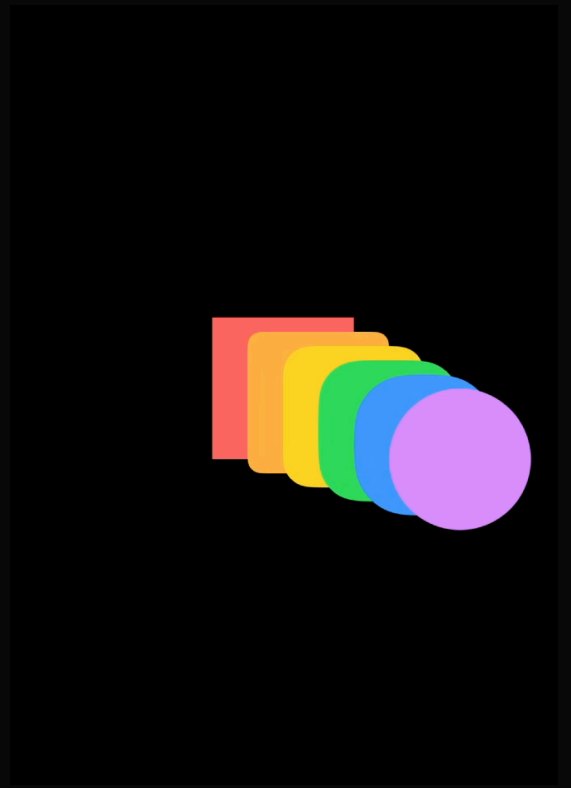


Бесконечная анимация

```
val animationSpec = rememberInfiniteTransition()

val offsetX by animationSpec.animateFloat(
    initialValue = -25f,
    targetValue = 25f,
    animationSpec = infiniteRepeatable(
        animation = tween(durationMillis = 1000, easing = EaseInOut),
        repeatMode = RepeatMode.Reverse
    )
)

val cornerRadius by animationSpec.animateFloat(
    initialValue = 1f,
    targetValue = 25f,
    animationSpec = infiniteRepeatable(
        animation = tween(durationMillis = 1000, easing = EaseInOut),
        repeatMode = RepeatMode.Reverse
    )
)
```

Бесконечная анимация

```
struct StackedSpring: View {
    let colors: [Color] =
    [.red, .orange, .yellow, .green, .blue, .purple]
    @State private var isMoving = false

    var body: some View {
        ZStack {
            ForEach(0..<colors.count, id: \.self) {
                Rectangle()
                    .fill(colors[$0])
                    .frame(width: 100, height: 100)
                    .cornerRadius(isMoving ? CGFloat($0) * 10.0 : CGFloat($0) * 1.0)
                    .offset(x: isMoving ? CGFloat($0) * 25.0 : CGFloat($0) * -25.0,
                            y: CGFloat($0) * 10.0)
            }
            .onAppear {
                withAnimation(.easeInOut(duration: 1).repeatForever(autoreverses: true)) {
                    isMoving.toggle()
                }
            }

        }
    }
}
```

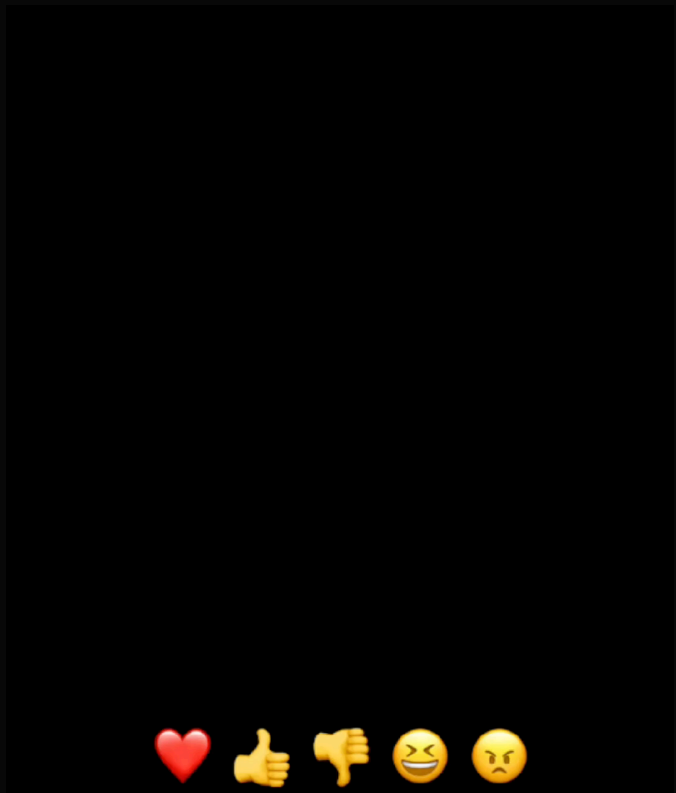


Бесконечная анимация

```
.onAppear {
    withAnimation(.easeInOut(duration: 1).repeatForever(autoreverses: true)) {
        isMoving.toggle()
    }
}
```

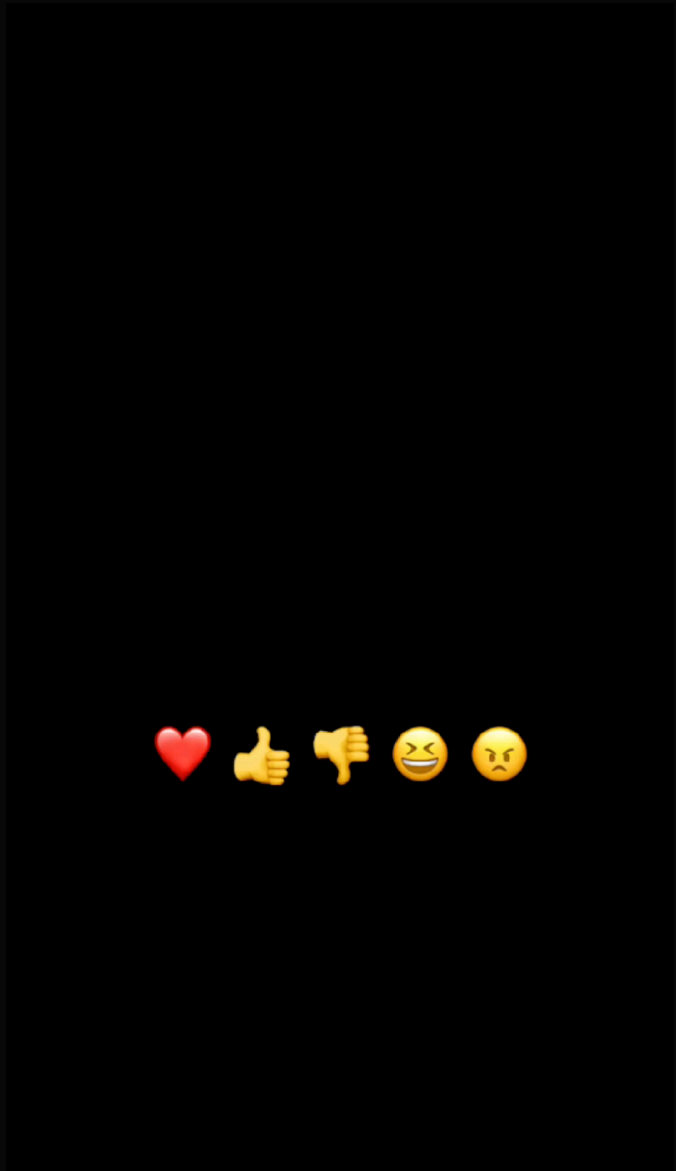
Keyframe в SwiftUI (iOS 17+)

```
content
  .keyframeAnimator(
    initialValue: EmojiReactionProps(),
    trigger: reactionCount) { content, value in
      content
        .rotationEffect(value.angle)
        .scaleEffect(value.scale)
        .scaleEffect(y: value.verticalStretch)
        .offset(y: value.verticalTranslation)
    } keyframes: { _ in
      KeyframeTrack(\.angle) {
        CubicKeyframe(.zero, duration: 0.58)
        CubicKeyframe(.degrees(16), duration: 0.125)
        CubicKeyframe(.degrees(-16), duration: 0.125)
        CubicKeyframe(.degrees(16), duration: 0.125)
        CubicKeyframe(.zero, duration: 0.125)
      }
      …
    }
```



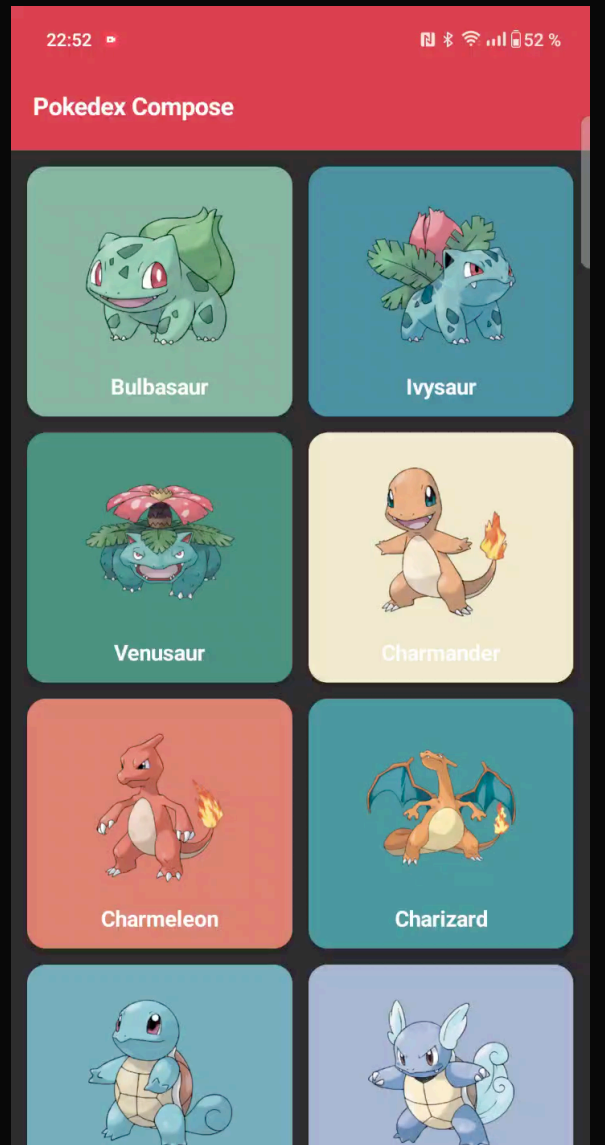
Keyframe в Compose

```
val coroutineScope = rememberCoroutineScope()
val translation = remember { Animatable(initialValue = 0f) }
val angle = remember { Animatable(initialValue = 0f) }
val scale = remember { Animatable(initialValue = 1f) }

Emoji(
    modifier = Modifier
        .graphicsLayer {
            rotationZ = angle.value
            translationY = translation.value.dp.toPx()
            scaleY = scale.value
            scaleX = scale.value
        }
        .clickable {
            coroutineScope.launch {
                launch {
                    translation.animateTo(0f, keyframes {
                        durationMillis = 1450
                        20f at 200 using EaseOutBounce
                        (-60f) at 300
                        (-60f) at 1300
                        0f at 1450
                    })
                }
                launch { … }
            }
        }
)
```



Hero анимация

1. Объявляем SharedTransitionLayout
2. Определяем модификатор sharedElement на каждое состояние



Hero анимация

```
struct MoviesListView: View {
    @Namespace var namespace

    var body: some View {
        NavigationView {
            ScrollView {
                ForEach(viewModel.movies, id: \.id) { movie in
                    Button(action: {
                        viewModel.didTapOnMovie(movie: movie)
                    }, label: {
                        movieCardView(movie: movie)
                            .matchedGeometryEffect(id: movie.id, in: namespace)
                    })
                }
            }

        }
        .modal(bindable: $viewModel.selectedMovie, destination: { movie in
            movieCardView(movie: movie)
                .matchedGeometryEffect(id: movie.id, in: namespace)
        })
    }
}
```

10 отличий SwiftUI и Compose

10 отличий SwiftUI и Compose

1. Структуры против функций

10 отличий SwiftUI и Compose

1. Структуры против функций
2. Строгая и нестрогая типизация

10 отличий SwiftUI и Compose

1. Структуры против функций
2. Строгая и нестрогая типизация
3. Один стейт против множества

10 отличий SwiftUI и Compose

1. Структуры против функций
2. Строгая и нестрогая типизация
3. Один стейт против множества
4. Колбэки против биндингов

10 отличий SwiftUI и Compose

1. Структуры против функций
2. Строгая и нестрогая типизация
3. Один стейт против множества
4. Колбэки против биндингов
5. Библиотека или часть ОС

10 отличий SwiftUI и Compose

1. Структуры против функций
2. Строгая и нестрогая типизация
3. Один стейт против множества
4. Колбэки против биндингов
5. Библиотека или часть ОС
6. Неявные и явные модификаторы

10 отличий SwiftUI и Compose

1. Структуры против функций
2. Строгая и нестрогая типизация
3. Один стейт против множества
4. Колбэки против биндингов
5. Библиотека или часть ОС
6. Неявные и явные модификаторы
7. Разный порядок модификаторов

10 отличий SwiftUI и Compose

1. Структуры против функций
2. Строгая и нестрогая типизация
3. Один стейт против множества
4. Колбэки против биндингов
5. Библиотека или часть ОС
6. Неявные и явные модификаторы
7. Разный порядок модификаторов
8. Видимость через типы или скоупы

10 отличий SwiftUI и Compose

1. Структуры против функций
2. Строгая и нестрогая типизация
3. Один стейт против множества
4. Колбэки против биндингов
5. Библиотека или часть ОС
6. Неявные и явные модификаторы
7. Разный порядок модификаторов
8. Видимость через типы или скоупы
9. Сложность корутин (9 абстракций против 4)

10 отличий SwiftUI и Compose

1. Структуры против функций
2. Строгая и нестрогая типизация
3. Один стейт против множества
4. Колбэки против биндингов
5. Библиотека или часть ОС
6. Неявные и явные модификаторы
7. Разный порядок модификаторов
8. Видимость через типы или скоупы
9. Сложность корутин (9 абстракций против 4)
10. Анимации через стейт и отдельный API на каждый чих

Вопросы?

Контакты

ajiekcx

kotlin_adept

Контур

Алексей Панов

Ведущий инженер-программист