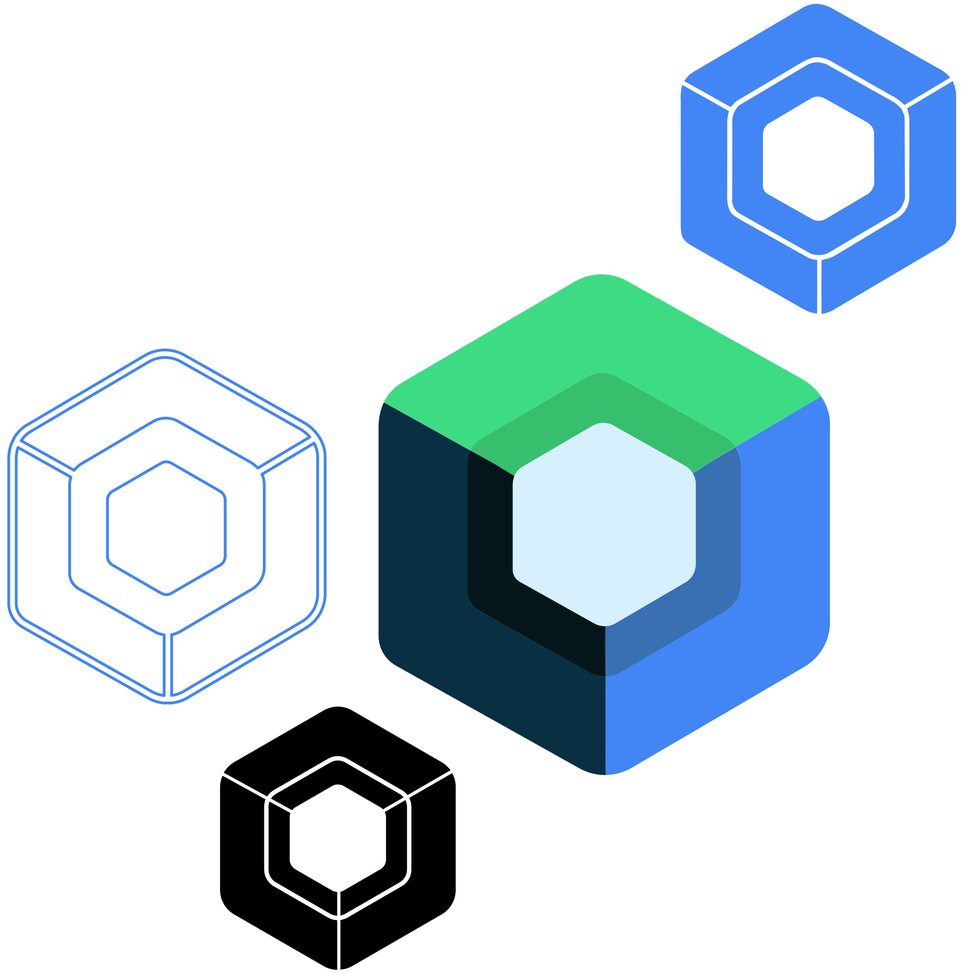
Thinking in Compose

state, stateless composables and Modifier



ДВФУ · ИМКТ · Департамент МиКМ · Android-разработка · осень-2025

Agenda

Recap: Composable, remember/remember, “UI=f(state)”

Thinking in Compose

Lifecycle

State hoisting

State holders

Modifiers

Live coding: Anime Browser Update

Outcome

- Понимание state -> recomposition -> UI
- stateless/stateful компоненты
- Понимание навигации

Thinking in Compose

Compose - how it works?

- Ui = f(state)
- Изменился state -> recomposition
- Нет “сеттера на Text”, есть новый вызов функции с новыми аргументами

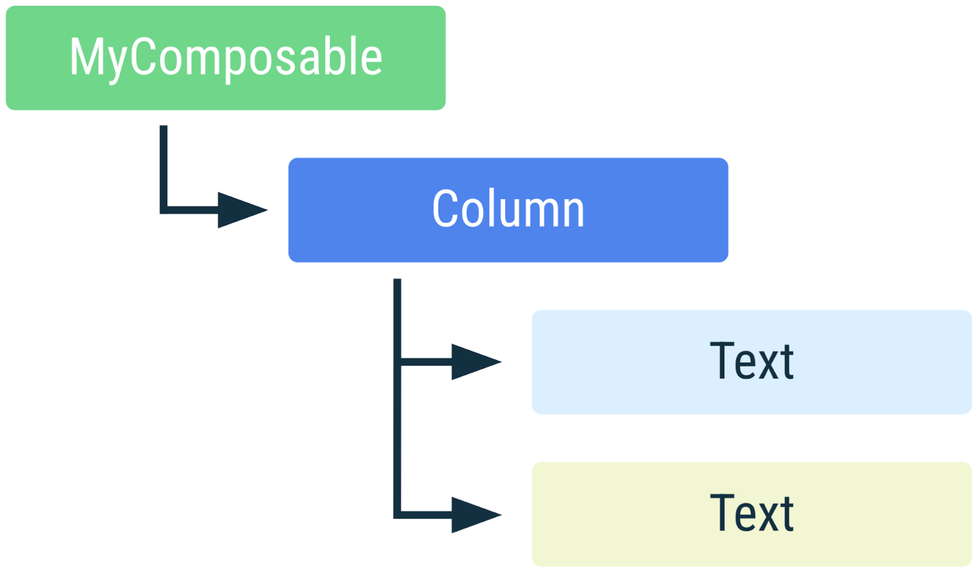
Thinking in Compose

UI = функция от state

- Мы не говорим UI “сделай так и так”
- Мы описываем: “при таком state UI выглядит так”
- При изменении observable state: Compose пересобирает нужные composable (recomposition)

Anatomy Compose

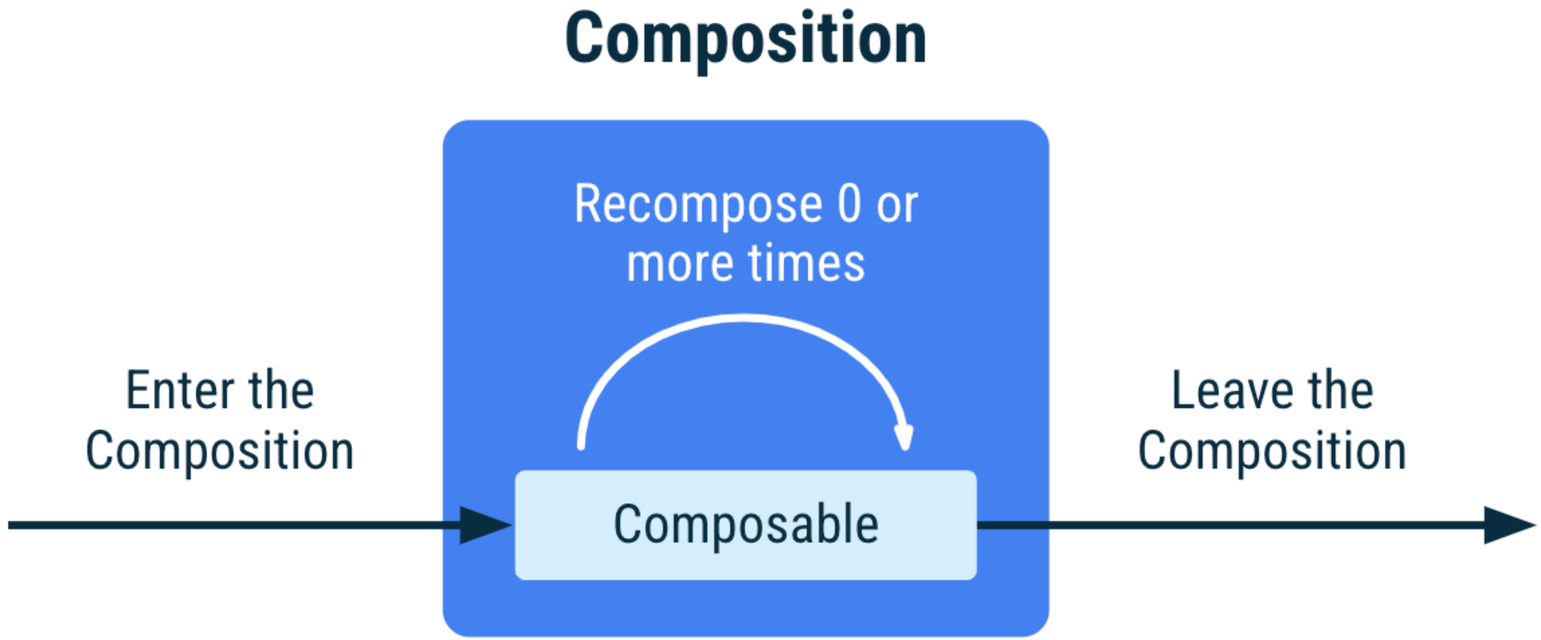
```
@Composable
fun MyComposable() {
    Column {
        Text("Hello")
        Text("World")
    }
}
```



Lifecycle

Lifecycle of composables

- Recomposition
- Phases:
  - Composition
  - Recomposition
  - Disposal



«Recomposition - когда Compose заново выполняет composable-функции в ответ на изменения состояния».

Lifecycle of composables

Enter -> (Recompose… Recompose… Recompose…) -> Leave

Важно помнить:

- Нельзя делать сайд-эффекты прямо в теле composable (например: запускать корутину, писать в лог постоянно, дергать сеть)
- Всё, что “нужно запомнить” между вызовами - через remember / rememberSaveable
- “Жизненный цикл” сайд-эффектов делается через Effect API

Composable fun

Она может держать состояние, но не “внутри функции” как в ООР, а через механизмы Compose:

- Локального “памятного var” внутри функции не существует - локальные переменные пересоздаются на каждом вызове.
- Всё, что должно пережить перевызовы, идёт в remember / rememberSaveable.
- Изменение данных обычно происходит в обработчиках событий (onClick / onValueChange), а не “само по себе” во время отрисовки.

Composable fun

- Composable - это не объект и не “экран со внутренними полями”.
- Это функция, которая описывает UI на основе текущего состояния. Когда состояние меняется, Compose просто заново вызывает нужные composable (recomposition)
- Composable должен быть максимально “чистым”: не хранить состояние в обычных переменных и не делать сайд-эффекты в теле. UI меняется только через изменение state.

State

State - что это вообще?

- State - любое значение, которое может меняться во времени
- Чтобы UI реагировал, state должен быть observable (mutableStateOf, remember, rememberSaveable)
- У state есть владелец (owner)
- Владелец отвечает за:
  - где и как хранить
  - когда и как менять

Где хранить состояние?

State можно хранить:

- Внутри composable (remember)
- Выше по дереву (родитель)

Single source of truth: одно место, где state “настоящий”

State

```
@Composable
fun Counter() {
    var count by remember { mutableStateOf(0) } // <- состояние

    Button(onClick = { count++ }) {              // <- запись -> триггерит recomposition
        Text("Count: $count")                    // <- чтение -> подписка на изменения
    }
}
```

State

- **Обычный var -** не триггерит перерисовку UI.
- mutableStateOf - “var с сигнализацией”: когда меняется, UI перерисовывается.
- remember - “память Compose”: твой observable-var не сбрасывается на каждом шаге.

Определения

- remember — это @Composable-функция, которая “запоминает” (кэширует) результат вычисления и возвращает тот же объект/значение при последующих рекомпозициях в рамках того же места в композиции, пока это место не исчезнет из дерева и/или не изменятся ключи.
- mutableStateOf(value) - это функция Jetpack Compose, которая создаёт наблюдаемое (observable) изменяемое состояние типа MutableState<T>.
  - Она хранит текущее значение в state.value (или через делегат var x by mutableStateOf(...)).
  - Когда value меняется, Compose помечает Composable, которые читали это состояние, и при следующем кадре выполняет recomposition (перерисовку) этих участков UI.

UDF

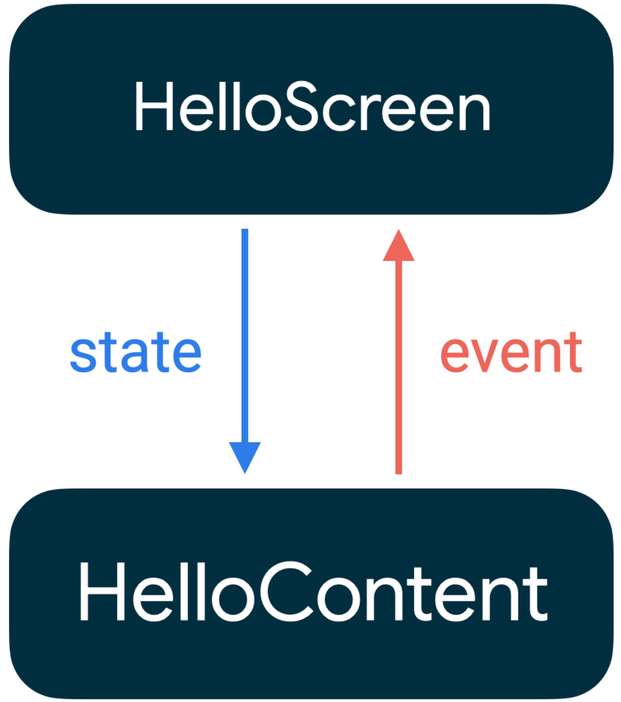
Data flows **down**, events flow **up.**

UDF

- **Unidirectional Data Flow:**
  - State flows down: состояние передаём вниз как параметры
  - Events flow up: действия пользователя поднимаем вверх через callbacks
- Цель: отделить UI (рисуем) от логики (меняем state)

https://developer.android.com/develop/ui/compose/architecture

UDF



- Event: UI генерирует событие (клик/ввод) -> вверх
- Update state: обработчик события меняет состояние
- Display state: новое состояние спускается вниз -> UI перерисовывается

State hoisting

State hoisting

Подъём состояния (state hoisting) в Compose - это паттерн, при котором состояние переносят к вызывающему composable, чтобы сделать сам composable без состояния (stateless).

Общий подход к подъёму состояния в Jetpack Compose заключается в том, чтобы заменить переменную состояния двумя параметрами:

- **value: T -** текущее значение, которое нужно отображать
- **onValueChange: (T) -> Unit -** событие, запрашивающее изменение значения, где T - предлагающееся новое значение

Stateful vs stateless composables

- Stateful composable:
  - хранит state внутри
  - сам решает, когда его менять
- Stateless composable:
  - получает **value** и **onValueChange** извне
  - сам state не хранит

State hoisting

Идея

- Держим state в **наименьшем общем предке** всех, кто его читает и пишет
- Вниз передаём:
  - immutable state (value)
  - callbacks (events)
- Вверх поднимается только событие (например, клик, изменение текста)

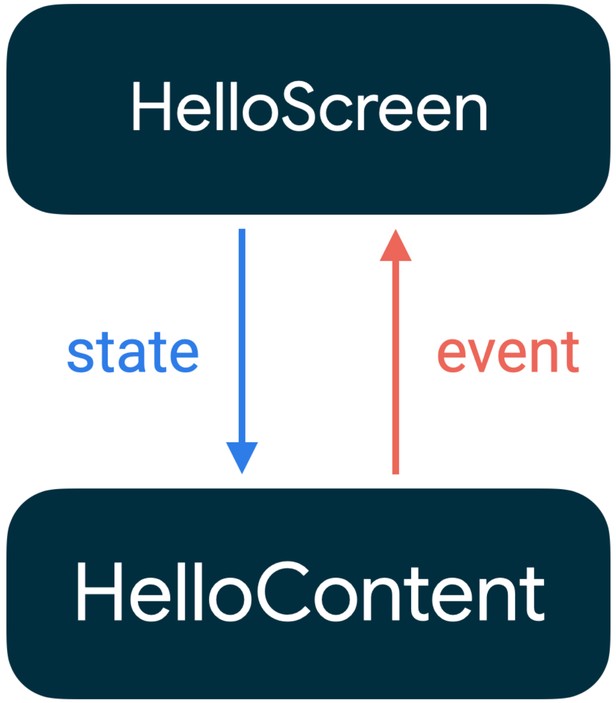
State hoisting

```
@Composable
fun Parent() {
    var value by remember { mutableStateOf("") }
    Child(
        value = value,
        onValueChange = { value = it }
    )
}
```

- Child - stateless
- Parent - stateful

“Hoist state to the lowest common ancestor”

State hoisting




```
@Composable
fun HelloScreen() {
    var name by rememberSaveable { mutableStateOf("") }

    HelloContent(name = name, onNameChange = { name = it })
}

@Composable
fun HelloContent(name: String, onNameChange: (String) -> Unit) {
    Column(modifier = Modifier.padding(16.dp)) {
        Text(
            text = "Hello, $name",
            modifier = Modifier.padding(bottom = 8.dp),
            style = MaterialTheme.typography.bodyMedium
        )
        OutlinedTextField(value = name, onValueChange = onNameChange, label = { Text("Name") })
    }
}
```

Stateful vs stateless composables

	Stateful	Stateless
Что делает	Хранит и меняет state	Только рисует по параметрам
Где использовать	На уровне экрана / крупных блоков	Внутри, для переиспользуемых компонентов

UI-компоненты по возможности делать stateless, а state держать выше.

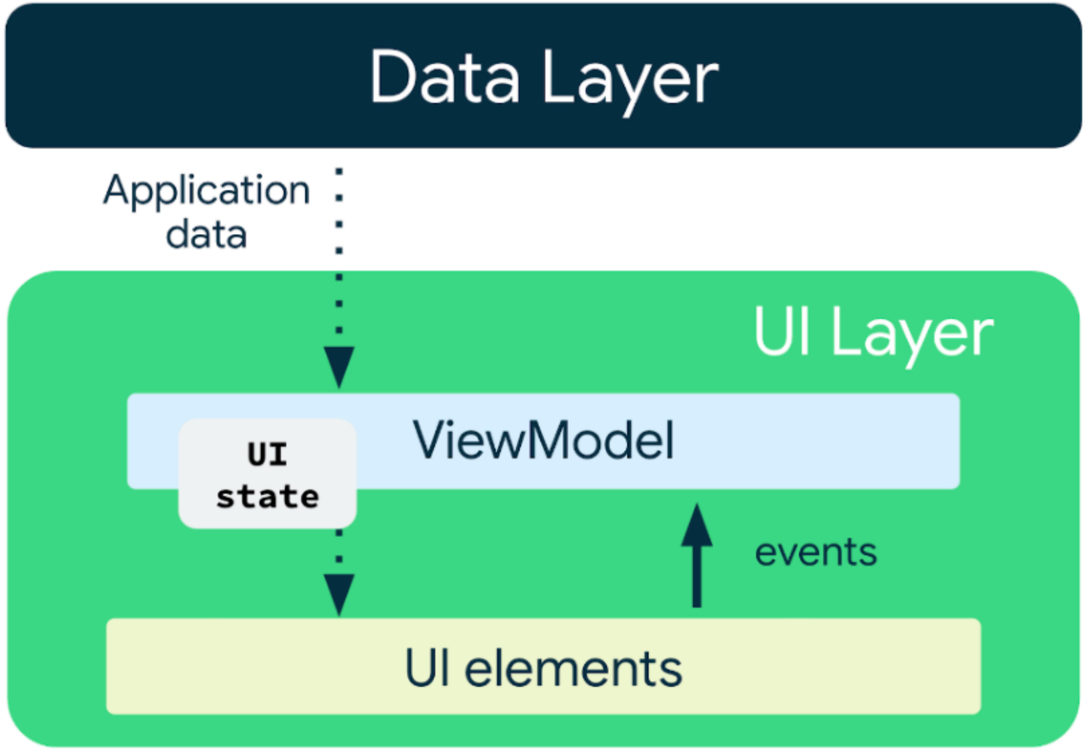
State holder

State holder

- State holder = класс, который:
  - хранит/производит Screen UiState
  - принимает events
  - применяет логику и обновляет state
- Важно: state holder можно сделать через ViewModel или обычный класс - сегодня делаем обычный класс

https://developer.android.com/topic/architecture/ui-layer/stateholders

UDF - State holder



Screen structure

UiState + Events + Screen()

- Шаблон (“контракт” экрана):
  - UiState - всё, что нужно, чтобы нарисовать экран
  - Events - все действия пользователя с экрана
  - Screen(state, events) - чистая отрисовка по входным данным
- Почему удобно:
  - UI становится stateless, проще превью/тесты/рефакторинг
  - состояние и логика - в одном месте (holder)

https://developer.android.com/develop/ui/compose/architecture

Частые грабли

- Разбросанный state: по remember в карточках -> потом невозможно понять, “кто владеет правдой”
- Mutable внутри UiState (например, MutableList) -> неожиданные баги; лучше immutable + copy()
- Смешивание UI logic и business logic -> держите “что показываем” отдельно от “что делаем”

Modifier

Modifier — это специальный объект, который позволяет «декорировать» (изменить) UI-элемент (composable): его размер, поведение, расположение, оформление.

Modifier

Modifier - конструктор поведения

Modifier - это способ изменить или дополнить поведение и внешний вид composable-элемента.

Через него вы говорите «этот элемент должен быть таким-то»:

- как он выглядит (size, padding, background)
- как он располагается (align, fillMaxWidth)
- как он реагирует на жесты/клики (clickable)
- как он обрабатывает ввод (scroll, draggable)
- какие эффекты получает (shadow, border)

Фактически это цепочка инструкций, применяемая к UI-элементу.

Modifier

```
modifier = Modifier
    .background(Color.Red)
    .padding(16.dp)

// фон на всю область + "внутренний" отступ
```

```
modifier = Modifier
    .padding(16.dp)
    .background(Color.Red)

// фон только вокруг содержимого, "отступ" вокруг карточки прозрачный
```

Полный список Modifier’s

https://developer.android.com/develop/ui/compose/modifiers-list