<TeachMeSkills/>

курс

Android разработчик

Jetpack Compose. 2 часть

Агенда занятия:

Создание своих компонентов

Работа со списками

Навигация

- **Создание своих компонентов**

  Работа со списками

  Навигация

Создание собственных компонентов

Jetpack Compose позволяет определять свои собственные UI-компоненты в виде функций с аннотацией @Composable. Такие Composable-функции можно многократно переиспользовать, как строительные блоки интерфейса. Вместо создания кастомных View-классов (как в классической Android-разработке) в Compose мы разбиваем интерфейс на мелкие переиспользуемые composable-функции, которые инкапсулируют определённый элемент UI или логику. Это повышает модульность и читаемость кода приложения.

Переиспользуемые composable-функции

Переиспользуемые composable-функции – это обычные функции Kotlin с аннотацией @Composable, которые формируют интерфейс и могут принимать параметры для настройки.

Их можно вызывать внутри других composable-функций, комбинируя сложный UI из простых компонентов. Хорошая практика – проектировать UI в виде набора мелких composable-функций, каждая из которых отвечает за один понятный элемент интерфейса или поведение. Такие функции легко повторно использовать в разных экранах приложения.

Переиспользуемые composable-функции

Например, можно создать composable-функцию MyButton, принимающую параметры текста, цвета фона, и лямбды клика, и внутри себя реализующую кнопку с заданным стилем. Затем MyButton можно использовать многократно на разных экранах, передавая разные тексты и обработчики событий. Так достигается реюзабельность компонентов – один раз реализовав UI-элемент, мы можем применять его где угодно, изменяя лишь входные данные.

Переиспользуемые composable-функции

```
@Composable
fun MyButton(
    text: String,
    onClick: () -> Unit,
    backgroundColor: Color = MaterialTheme.colorScheme.primary,
    textColor: Color = Color.White,
    modifier: Modifier = Modifier
) {
    Button(
        onClick = onClick,
        colors = ButtonDefaults.buttonColors(containerColor = backgroundColor),
        shape = RoundedCornerShape(16.dp),
        modifier = modifier.padding(4.dp)
    ) {
        Text(text = text, color = textColor)
    }
}
```

```
// Пример использования кастомной кнопки на экране
@Composable
fun ExampleUsage() {
    Column {
        MyButton(
            text = "Подтвердить",
            onClick = { /* обработка нажатия */ }
        )
        MyButton(
            text = "Удалить",
            backgroundColor = Color.Red,
            textColor = Color.White,
            onClick = { /* другое действие */ }
        )
    }
}
```

Подъём состояния (State Hoisting)

При создании компонентов важно решить, как они будут управлять состоянием (state).
Подъём состояния – это рекомендуемый шаблон, при котором состояние выносится из composable-функции наружу, к её вызывающему коду. Вместо того, чтобы компонент хранил и изменял своё состояние сам, он получает текущее значение через параметр (например, value) и функцию-колбэк для изменения этого значения (например, onValueChange).
Таким образом компонент становится stateless (без собственного состояния), а все данные приходят извне и сообщаются обратно через события.

Подъём состояния (State Hoisting)

Преимущества подъёма состояния:

1. Соблюдается единственный источник правды – состояние не дублируется в нескольких местах, а хранится наверху, у родительского компонента.
2. Компонент становится более переиспользуемым и тестируемым, т.к. не зависит от конкретного способа хранения данных – их можно передавать из ViewModel, из другого composable или любого источника.

Также реализуется принцип однонаправленного потока данных: данные (state) текут сверху вниз по дереву UI, а события (events) поднимаются снизу вверх. Благодаря этому UI-компоненты отделены от логики изменения данных.

Подъём состояния (State Hoisting)

```
// Stateless composable — не хранит состояние, получает все извне
@Composable
fun Counter(value: Int, onIncrement: () -> Unit, modifier: Modifier = Modifier) {
    Button(onClick = onIncrement, modifier = modifier) {
        Text(text = "Счёт: $value")
    }
}

// Stateful composable — хранит состояние count внутри и использует stateless Counter
@Composable
fun CounterStateful(modifier: Modifier = Modifier) {
    var count by remember { mutableStateOf(0) }
    Counter(value = count, onIncrement = { count++ }, modifier = modifier)
}
```

Параметр modifier: Modifier = Modifier

При создании своего composable-компонента обязательно добавляйте параметр modifier: Modifier = Modifier в его сигнатуру. Modifier в Compose используется для задания внешнего вида, отступов, расположения, обработчиков ввода и прочих аспектов компоновки. Передавая Modifier снаружи, вы позволяете пользователю вашего компонента настраивать его оформление и позиционирование.

Как это всё работает

Смотрим проект

Создание своих компонентов

▶ **Работа со списками**

Навигация

Работа со списками

Современные списки в Jetpack Compose строятся с помощью Lazy-компонентов – таких как **LazyColumn** и **LazyRow**. Эти компоненты предназначены для эффективного отображения прокручиваемых списков (аналог RecyclerView в View-системе) и реализуют ленивую подгрузку элементов: только те элементы, которые видны на экране, реально составляются и рисуются.

Работа со списками

LazyColumn vs LazyRow.

Как следует из названия, LazyColumn организует элементы столбцом, т.е. вертикально, а LazyRow – строкой, горизонтально. Оба работают схожим образом, различаясь лишь направлением прокрутки (вертикальный скролл для LazyColumn, горизонтальный – для LazyRow).

Работа со списками

Чтобы отобразить список, достаточно вызвать, например, LazyColumn и передать в его DSL-блок элементы с помощью функций item { ... } или items { ... }.

item используется для добавления одиночного элемента в список. Например, можно добавить заголовок списка или рекламный баннер единично.

```
LazyColumn {
    item {
        Text("Заголовок списка", style = MaterialTheme.typography.headlineMedium)
    }
    // ... далее элементы списка ...
}
```

Работа со списками

items используется для списка неизвестной заранее длины – она перебирает коллекцию данных. Например, если у нас есть List<Message> messages, мы можем сделать:

```
LazyColumn {
    items(messages) { message ->
        MessageRow(message)
    }
}
```

Работа со списками

Мы можем комбинировать item и items в одном списке. Compose позволяет в одном LazyColumn чередовать разные типы контента. Например, реализуя список товаров, можно вставить специальный элемент-рекламу между товарными позициями:

```
LazyColumn {
    items(products) { product ->
        ProductCard(product)
    }
    item {
        AdvertisementBanner()
    }
    items(moreProducts) { product ->
        ProductCard(product)
    }
}
```

Оптимизация списков: key

Списки в Compose умеют автоматически переиспользовать composable-элементы при прокрутке, но для максимальной эффективности recomposition стоит предоставлять уникальные ключи для элементов. Параметр key в функциях items/item позволяет задать стабильный идентификатор для каждого элемента списка.

Оптимизация списков: key

Зачем это нужно? Если изменяется порядок элементов (например, элемент переместился с нижней позиции вверх при сортировке по времени), Compose по умолчанию не знает, что это тот же самый элемент, просто на новом месте.
Без ключей система решит, что прошлый элемент удален, а на его месте появился новый, и перерисует большую часть списка. Если же у каждого элемента есть уникальный ключ (например, ID объекта), Compose сможет отследить, что элемент с этим ключом просто сменил позицию, и не будет лишний раз его перерисовывать

Оптимизация списков: key

Использование ключей очень простое: достаточно передать параметр key в функцию items.

Например:

```
LazyColumn {
    items(
        items = notes,
        key = { note -> note.id }  // используем уникальный идентификатор заметки
    ) { note ->
        NoteRow(note)
    }
}
```

Управление прокруткой

Для программного управления прокруткой используется состояние списка – **объект LazyListState**. Его можно получить с помощью функции **rememberLazyListState()**.
Этот объект передаётся в LazyColumn/LazyRow через параметр state и предоставляет информацию о текущей прокрутке (например, индекс первого видимого элемента, смещение) и методы для управления скроллом.
Два основных метода: **scrollToItem()** – мгновенно прокручивает список к заданному индексу (без анимации), и **animateScrollToItem()** – плавно прокручивает (с анимацией).

Управление прокруткой

Оба метода являются suspend-функциями, то есть их необходимо вызывать из корутины. Обычно для этого используют LaunchedEffect или rememberCoroutineScope + launch. Например, можно сделать кнопку "Наверх", при нажатии на которую список прокрутится к первому элементу.

Управление прокруткой

```
val listState = rememberLazyListState()
val coroutineScope = rememberCoroutineScope()

LazyColumn(state = listState) {
    items(messages) { message ->
        MessageRow(message)
    }
}

// Кнопка прокрутки списка наверх
Button(onClick = {
    coroutineScope.launch {
        listState.animateScrollToItem(index = 0)  // плавно прокручиваем к индексу 0 (пер
    }
}) {
    Text("Наверх")
}
```

Управление прокруткой

Помимо ручного управления, LazyListState позволяет наблюдать за прокруткой. Например, можно показать кнопку "В начало" только когда список пролистан вниз: для этого отслеживают listState.firstVisibleItemIndex через snapshotFlow или derivedStateOf. Но такие сложные кейсы выходят за рамки базового материала.

Как это всё работает

Смотрим проект

Создание своих компонентов

Работа со списками

▶ **Навигация**

Навигация

Для многокранового приложения одной декларации экранов недостаточно – нужно уметь переходить между экранами. В Jetpack Compose роль навигации выполняет библиотека Navigation Compose (модуль androidx.navigation.compose). Она предоставляет удобные средства для описания навигационного графа приложения и управления переходами при помощи NavController.

Основные компоненты Navigation Compose

NavController – контроллер навигации, управляющий стеком экранов. Он отслеживает, какие composable-экраны находятся сейчас на экране и на каком экране находится пользователь. По сути, NavController хранит состояние навигации и предоставляет методы для перемещения.

Функция composable – используется внутри NavHost для объявления экрана. Ей задаётся уникальный route (маршрут) и содержимое – lambda с UI данного экрана.

Основные компоненты Navigation Compose

NavHost – специальный composable-контейнер, внутри которого определяется граф навигации. NavHost связывает NavController с конкретными пунктами назначения (destination) – composable-экранами, на которые можно навигировать. При навигации NavHost автоматически меняет отображаемый контент на нужный экран. Каждый экран в графе идентифицируется строковым маршрутом (route).

Основные компоненты Navigation Compose

```
val navController = rememberNavController()
NavHost(navController = navController, startDestination = "list") {
    composable("list") { backStackEntry ->
        // Экран списка
        TaskListScreen(onTaskSelected = { taskId ->
            navController.navigate("detail/$taskId")
        })
    }
    composable("detail/{taskId}") { backStackEntry ->
        // Экран деталей, получаем аргумент из пути
        val taskId = backStackEntry.arguments?.getString("taskId")
        TaskDetailScreen(
            taskId = taskId,
            onBack = { navController.popBackStack() }
        )
    }
}
```

Передача аргументов между экранами

Navigation Compose позволяет передавать данные при переходе. В примере выше мы передали идентификатор задачи через часть пути (маршрут). Шаблон {taskId} в route экрана деталей означает, что NavController ожидает соответствующий аргумент при навигации. Мы сформировали строку "detail/42", и значение "42" стало доступно на экране деталей.

Передача аргументов между экранами

По умолчанию все параметры в маршруте трактуются как строки. Однако библиотека поддерживает явное указание типов аргументов через функцию navArgument. В случае, если taskId – число, можно описать экран так:

```
composable(
    route = "detail/{taskId}",
    arguments = listOf(navArgument("taskId") { type = NavType.IntType })
) { ... }
```

Как это всё работает

Смотрим проект

Практика

Задача 1. Создание собственного компонента

Реализуйте Composable-функцию LikeButton, отображающую иконку "лайк”, которая может быть либо заполненной, либо контурной.
Сделайте stateless-реализацию.
Затем создайте stateful-версию.

Задача 2. Работа со списком

Используя LazyColumn, реализуйте экран со списком, например, список задач.

Каждая задача может содержать название и чекбокс "выполнено".

Создайте data class Task(val id: Int, val title: String, val completed: Boolean) и список тестовых данных.

Отобразите задачи в LazyColumn с помощью items.

Задача 3. Навигация между экранами

Разработайте простое Compose-приложение с двумя экранами и навигацией между ними.

Первый экран – список (можете использовать список задач из задания 2 или простой список имен).

Второй экран – детальная информация (например, детали выбранного элемента списка).

Домашнее задание

Задача 1. Приложение "Список дел"

Приложение должно иметь как минимум два экрана – список задач и экран деталей/редактирования задачи
Список задач: используйте LazyColumn для отображения списка.
Пользовательские компоненты: создайте как минимум один свой компонент для UI.
Навигация и аргументы: экран деталей задачи должен получать ID (или другой параметр) выбранной задачи и отображать подробную информацию.

Дополнительно (по желанию):

Реализуйте добавление новой задачи. Для этого может быть отдельный экран "Новая задача" или диалог. Можно также добавить возможность удаления задачи на экране деталей с возвратом назад.

Q&A

Ваши вопросы

Спасибо

<TeachMeSkills/>