<TeachMeSkills/>

курс

Android разработчик

Jetpack Compose. 1 часть

Агенда занятия:

Базовые контейнеры

Базовые компоненты

Modifier

- **Базовые контейнеры**
- Базовые компоненты
- Modifier

Что такое Jetpack Compose

Jetpack Compose – это современный декларативный UI–фреймворк для Android, позволяющий описывать интерфейс полностью на Kotlin без использования XML-разметки.

В отличие от классического подхода с View и XML, Compose использует компонуемые функции вместо макетов и реактивное обновление UI вместо ручного изменения View.

Отличия от XML/View

Код вместо XML.

Интерфейс задаётся кодом Kotlin (@Composable функции) вместо декларативных XML-файлов. Отпадает необходимость вызывать findViewById() и привязывать виджет к коду – UI генерируется функцией напрямую.

Декларативность vs Императивность.

Во View-системе разработчик вручную изменяет состояние виджетов (вызывает textView.setText(), добавляет/удаляет view), а в Compose описывает, как UI должен выглядеть при данном состоянии, и система обновляет только необходимые элементы при изменении данных.

Отличия от XML/View

Меньше шаблонного кода.

Compose устраняет большую часть шаблонного кода (boilerplate), такого как создание адаптеров для списков или использование привязок. Данные и UI связываются напрямую через состояние, и изменения данных автоматически отражаются на экране.

Отличия от XML/View

Реактивность.

В Compose встроена реактивная модель: если изменяется наблюдаемое состояние, соответствующие компоненты UI перерисовываются автоматически, не требуя вызова методов обновления вручную. Это снижает вероятность ошибок, когда старый подход заставлял разработчика помнить об обновлении всех связанных View.

Интеграция с View.

Compose можно постепенно внедрять в существующие приложения – он совместим с традиционными View. Можно в одном экране использовать и Compose-элементы, и старые View (через специальные контейнеры), упрощая миграцию.

Отличия от XML/View

xml

```xml
<TextView
    android:id="@+id/message"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello, World!" />
```

```
@Composable
fun GreetingMessage(name: String) {
    Text(text = "Hello, $name!")
}
```

Декларативный подход к UI: UI как функция от состояния

Compose следует декларативной парадигме: вы описываете что должно быть показано при данном состоянии, вместо того чтобы пошагово прописывать как изменить UI при событиях.
Интерфейс становится чистой функцией UI = f(state):

Декларативный подход к UI: UI как функция от состояния

Императивный стиль (View/XML): разработчик меняет существующее UI, вызывая методы виджетов при событиях.
Например, при нажатии кнопки нужно найти текстовое поле и обновить его текст. Код управления состоянием разбросан по обработчикам событий, легко допустить ошибку (не обновить какой-то виджет, обновить несуществующий элемент и т.п.).

Декларативный подход к UI: UI как функция от состояния

Декларативный стиль (Compose): разработчик описывает привязку данных к UI. Например, функция принимает name: String и отображает Text("Hello, $name"). При изменении name функция просто вызывается снова (рекомпозируется) с новым значением и перерисовывает текст.

Нет необходимости вручную отслеживать и синхронизировать изменения – фреймворк делает это сам, “пересобирая” экран.

Модель состояния и рекомпозиция в Compose

Состояние (State) – это источник правды для UI в Compose. Когда меняется состояние, фреймворк запускает процесс рекомпозиции, то есть повторного выполнения соответствующих компонуемых функций, чтобы обновить UI.

Модель состояния и рекомпозиция в Compose

Composition и Recomposition.

Композиция – начальное построение UI, когда компонуемые функции вызываются впервые и формируют Composition (специальную структуру данных, описывающую UI-дерево).

Рекомпозиция – повторный вызов части компонуемых функций при изменении их входных данных или наблюдаемого состояния, для обновления Composition. Изначально функция выполняется (композится) один раз, затем при изменениях данных она может перекомпозироваться (recompose) множество раз.

Модель состояния и рекомпозиция в Compose

Обновление через параметры.

Compose – декларативная система, поэтому единственный способ обновить UI – вызвать ту же функцию с новыми аргументами (состоянием).

Например, у вас есть @Composable fun MessageCard(text: String). Если text изменился, нужно заново вызвать MessageCard с новым значением. На практике Compose сам это делает, отслеживая состояние: вы изменяете переменную состояния, и система помечает, какие функции зависят от нее, и перевызывает их.

Модель состояния и рекомпозиция в Compose

mutableStateOf и автоматическая подписка.

В Compose есть специальный тип состояния MutableState<T> – наблюдаемое состояние. Его проще всего получить через функцию mutableStateOf(initial). Значение, обернутое в mutableStateOf, при изменении уведомляет Compose runtime, и все компонуемые, которые во время последней композиции прочитали это значение, ставятся в очередь на рекомпозицию.

Модель состояния и рекомпозиция в Compose

Как только выполнен count.value++, Compose узнает, что count обновился, и автоматически перерисовывает (рекомпозирует) ту часть UI, где использовался count, т. е. текст с счётчиком увеличится на единицу.

```
val count = mutableStateOf(0)     // создание состояния
Text("Count: ${count.value}")     // использование состояния в UI
// ... позже
count.value++                     // изменение состояния
```

Модель состояния и рекомпозиция в Compose

remember для хранения между рекомпозициями.

При каждой рекомпозиции компонуемые функции могут вызываться заново, создавая новые локальные переменные. Чтобы сохранить какое-то значение между вызовами, используется API remember.

Например:

```
@Composable
fun Counter() {
    var count by remember { mutableStateOf(0) }
    Button(onClick = { count++ }) {
        Text("Clicked $count times")
    }
}
```

Модель состояния и рекомпозиция в Compose

Здесь remember { mutableStateOf(0) } создаст один раз объект состояния при первой композиции, и при следующих рекомпозициях будет возвращать тот же объект (с сохранившимся значением). Без remember переменная count сбрасывалась бы каждый раз к initial.

Что запускает рекомпозицию

Любое изменение обозреваемого состояния, прочитанного в композиции, помечает соответствующую composable функцию “неактуальной”. Когда наступает следующий кадр (или цикл диспетчера UI), Compose инвалидирует (invalidation) эту область и запускает ее рекомпозицию.

Что запускает рекомпозицию

Изменения могут происходить:

- **В локальном состоянии** (через mutableStateOf внутри composable + remember).
- **В внешнем состоянии**, например, Flow/LiveData из ViewModel, которые интегрируются в Compose через collectAsState() – тоже возвращает State<T> объект. При эмиссии нового значения Flow происходит рекомпозиция.
- **В параметрах функции.** Если родительский composable вызвал дочерний с другим аргументом, дочерний будет рекомпозирован.

Granularity

Granularity – гранулярность обновлений.
Compose очень точечно обновляет UI. Не весь экран перерисовывается, а только те функции, чьи данные изменились. Например, если у вас список элементов и изменился только один элемент, при правильном подходе Compose обновит (рекомпозирует) только соответствующий Composable-элемент списка. Это большое отличие от invalidate() в View, который часто заново прорисовывал весь View или большой участок

Базовые контейнеры компоновки

Jetpack Compose предоставляет ряд стандартных контейнеров для компоновки элементов.

Три основных из них: **Column, Row и Box**. С их помощью можно расположить элементы по вертикали, горизонтали или накладывать друг на друга соответственно. Эти контейнеры аналогичны привычным макетам Android (LinearLayout вертикальный, LinearLayout горизонтальный и FrameLayout).

Column (вертикальный столбец)

Column располагает дочерние элементы вертикально друг под другом. Если поместить несколько компонентов внутрь Column, они будут выстроены в колонку (по одной на каждой "строке"). По умолчанию элементы прижаты к верхней границе контейнера и выравниваются слева. Column – это эквивалент вертикального LinearLayout.

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

Column (вертикальный столбец)

Для настройки положения элементов внутри Column можно использовать параметры **verticalArrangement** (например, Arrangement.Center для центровки по вертикали или Arrangement.SpaceBetween для распределения свободного места) и **horizontalAlignment** (например, Alignment.CenterHorizontally для центровки по горизонтали).

Если требуется сделать колонку прокручиваемой, в дальнейшем можно будет использовать модификатор **verticalScroll** или специализированные списки (LazyColumn), но это выходит за рамки текущего занятия.

Row (горизонтальный ряд)

Row размещает дочерние элементы горизонтально в одну строку. Элементы располагаются слева направо (если не указано иное) и по умолчанию выравниваются по верхней границе контейнера.

Row аналогичен горизонтальному LinearLayout.

Пример: строка, содержащая изображение аватара и рядом колонку из двух текстов
(имя пользователя и статус).

```
@Composable
fun ProfileRow() {
    Row(verticalAlignment = Alignment.CenterVertically) {
        Image(
            painter = painterResource(R.drawable.avatar),
            contentDescription = "Аватар",
            modifier = Modifier.size(40.dp)
        )
        Column {
            Text("Иван Петров")
            Text("онлайн", fontSize = 12.sp)
        }
    }
}
```

Box (слоящийся контейнер)

Box накладывает дочерние элементы друг на друга (поверх друг друга) в порядке их объявления. Это похоже на FrameLayout в классическом Android. Box полезен, когда нужно, чтобы один элемент перекрывал другой (например, значок поверх изображения). По умолчанию все дочерние элементы Box позиционируются в его верхнем левом углу, но Box позволяет задавать выравнивание содержимого.

Box (слоящийся контейнер)

Пример: изображение аватара с наложенным поверх значком статуса (галочка).

```
@Composable
fun AvatarBox() {
    Box {
        Image(
            painter = painterResource(R.drawable.avatar),
            contentDescription = "Аватар",
            modifier = Modifier.size(80.dp)
        )
        Icon(
            imageVector = Icons.Filled.Check,
            contentDescription = "Статус онлайн",
            tint = Color.Green,
            modifier = Modifier.align(Alignment.BottomEnd)
        )
    }
}
```

Базовые контейнеры

▶ **Базовые компоненты**

Modifier

Базовые UI-компоненты: Text, Button, Image, Icon

Теперь рассмотрим основные визуальные компоненты Jetpack Compose, из которых строится интерфейс.
Они аналогичны стандартным View-элементам Android, но используются внутри Composable-функций.

Text (текст)

Text – компонент для отображения текста. Это аналог TextView, но в Compose он задаётся декларативно. Достаточно вызвать Text("Ваш текст") внутри @Composable-функции, чтобы вывести строку на экран.

Text поддерживает множество параметров для стилизации: цвет (color), размер шрифта (fontSize), начертание (fontWeight), добавление отступов через modifier и др. Например, можно сделать текст крупнее и цветным.

```
Text(text = "Привет, Compose!")
```

Button (кнопка)

Button – компонент для создания интерактивной кнопки. В Compose кнопка представляет собой функцию с параметром onClick (обработчик нажатия) и слотом для содержимого (чаще всего Text). Эквивалент старого Button/MaterialButton в View.

```
Button(
    onClick = { /* действие при нажатии */ },
    modifier = Modifier.padding(8.dp)
) {
    Text("Нажми меня")
}
```

Image (изображение)

Image – компонент для отображения изображения (Bitmap или VectorDrawable). В Compose, чтобы отобразить картинку из ресурсов, используют вспомогательную функцию painterResource. Она загружает рисунок и возвращает объект Painter, который понимает Image.

```
Image(
    painter = painterResource(R.drawable.my_image),
    contentDescription = "Описание изображения"
)
```

Icon (иконка)

Icon – специальный компонент для отображения значков (иконографики). Обычно используется с библиотекой Material Icons, встроенной в Compose. Для доступа к иконкам используется объект Icons. В Compose доступны несколько наборов иконок (Filled, Outlined, Rounded, TwoTone, Sharp). Например, Icons.Filled.Home – это иконка "дом" в залитом стиле.

```
Icon(
    imageVector = Icons.Filled.Star,
    contentDescription = "Избранное"
)
```

Базовые контейнеры

Базовые компоненты

- **Modifier**

Modifier: назначение и применение

В Compose модификаторы (Modifier) используются для декорирования и изменения поведения компонентов. Modifier можно представить как набор инструкций, который "навешивается" на компонент и описывает его внешний вид, размер, отступы, выравнивание и интерактивность.

Modifier: назначение и применение

Модификаторы позволяют:

- **Изменять размер и расположение элемента** (padding для отступов, fillMaxWidth для расширения на всю ширину родителя, width/height для фиксированных размеров);
- **Менять внешний вид:** добавлять фон (background), рамку (border), прозрачность (alpha);
- **Добавлять поведение:** делает элемент кликабельным (clickable), прокручиваемым (verticalScroll/horizontalScroll), перетаскиваемым (draggable) и др.;
- **Вносить дополнительные сведения:** например, помечать элемент меткой для доступности (через semantics или тот же contentDescription).

Modifier: назначение и применение

Модификаторы задаются через параметр modifier почти у каждого компонуемого элемента. Они объединяются цепочкой вызовов: Modifier.padding(...).background(...).clickable {...} и т.д. – последовательность, которую можно читать слева направо.

Например:

```
Text(
    text = "Hello, Compose!",
    modifier = Modifier
        .padding(16.dp)
        .background(Color.Cyan)
)
```

Modifier: назначение и применение

Порядок модификаторов имеет значение. Каждый последующий модификатор применяется к результату предыдущего, поэтому меняя порядок, можно получить разный эффект.

Например, если поменять местами padding и background в первом примере, фон будет окрашивать только область текста без учета отступов, или наоборот – захватывать разный размер области.

Modifier: назначение и применение

Всегда **следите за тем**, в каком порядке вы вызываете модификаторы, и понимайте, как этот порядок влияет на итоговую разметку элемента.
Еще пример: если сначала вызвать .padding(), а потом .clickable(), то зона, реагирующая на нажатие, не будет включать паддинги. Нажатие сработает только по исходной области элемента. Если же сначала .clickable(), а затем .padding(), то кликабельная область расширится вместе с отступами.
Таким образом, располагайте модификаторы осознанно, исходя из требуемого результата.

Modifier: назначение и применение

Смотрим проект

Практика

Задача 1.

Сверстайте карточку товара.

- Картинка товара (иконка или Image)
- Название и цена (Text)
- Кнопка “В корзину”

Оформление: отступы, фон, закругления

Задача 2.

Сверстайте простую карточку профиля с использованием Jetpack Compose, применяя изученные контейнеры, компоненты и модификаторы.

Ваша карточка профиля должна включать следующие элементы:

- Аватар
- Имя пользователя
- Статус или описание
- Кнопка действия – Button (например, "Добавить в друзья" или "Написать сообщение").

Задача 3. Счётчик с кнопками

Создайте экран с числом и двумя кнопками.
Отображается count.
Кнопки: + и -
При нажатии изменяется значение счетчика.

Домашнее задание

Задача 1. Экран профиля

Разработайте страницу профиля пользователя.

Верхняя часть – фото пользователя. Ниже – имя, информация о пользователе, и ряд кнопок (например "Позвонить", "Написать", "Добавить").

Примените Modifier.padding для отступов и Modifier.fillMaxWidth() для ширины изображений/кнопок где нужно.

Q&A

Ваши вопросы

Спасибо

<TeachMeSkills/>