KamiLimu

22nd June 2024

Introduction to Jetpack Compose

Introduction to the Kotlin programming language with the Jetpack Compose Framework

What we will cover

- What is Jetpack Compose
- Declarative UI
- Composable Functions
- Tooling and Debugging
- Modifiers
- State Management
- Modularisation & Reusability
- Integration with XML-based Android code
- Material Design
- Theming and Styling

KAMI LIMU

Jetpack Compose

What is Jetpack Compose?

Jetpack Compose is a modern toolkit for building native Android user interfaces

Jetpack Compose addresses several key issues developers face when using traditional UI frameworks

Here are the main problems that are being addressed

Jetpack Compose

Boilerplate Code

Traditional UI development often involves writing boilerplate code, such as defining XML layouts and manually handling view binding

Compose allows developers to build UIs by simply describing the desired UI state and its transformations in Kotlin code

Performance Overheads

The traditional View system can have performance overheads due to deep view hierarchies and complex view-rendering processes

Compose optimizes performance by minimizing view hierarchies and using efficient recomposition strategies

Jetpack Compose

Complex State Management

Managing UI state changes in the Android View system can be cumbersome and error-prone

In the traditional Android view system, state management often involves managing view states manually through a combination of XML layouts, View classes, and ViewModel or other state-holder classes

The process generally involves:

- Defining views in XML
- Referencing and manipulating these views in Activities or Fragments
- Using ViewModels to hold and manage UI-related data, ensuring it survives configuration changes

KAMI LIMU

Jetpack Compose - State Management

Example of state management in the traditional Android view system with a simple counter when the button is pressed

```xml
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="0"
        android:textSize="30sp"/>

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Increment"/>

</LinearLayout>
```

```kotlin
class MainActivity : AppCompatActivity() {
    private lateinit var textView: TextView
    private lateinit var button: Button
    private var count = 0

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        textView = findViewById(R.id.textView)
        button = findViewById(R.id.button)

        button.setOnClickListener {
            count++
            textView.text = count.toString()
        }
    }
}
```

KamiLimu

Jetpack Compose - State Management

Jetpack Compose leverages a declarative approach, where the UI automatically updates in response to state changes

State management is directly integrated into the composable functions

Example of the counter in Jetpack Compose

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

    Column(
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center,
        modifier = Modifier.fillMaxSize()
    ) {
        Text(
            text = count.toString(),
            fontSize = 30.sp
        )
        Button(onClick = { count++ }) {
            Text(text = "Increment")
        }
    }
}
```

Jetpack Compose

Fragmented UI Development

In the traditional approach, UI development often involves working with XML for layout definitions and Java/Kotlin for behavior.

Jetpack Compose unifies these into a single Kotlin codebase, streamlining the development process and reducing context-switching

Difficulty in Creating Custom Views

Creating custom views in the Android View system can be complex and requires extending existing classes and handling various lifecycle methods. Jetpack Compose makes it easier to create custom UI components by using composable functions

KAMI LIMU

Jetpack Compose - Custom Views

Example in Android view system
Creating a custom view by extending a View class

```java
// MyCustomView.java
public class MyCustomView extends View {
    public MyCustomView(Context context) {
        super(context);
        init();
    }

    public MyCustomView(Context context, AttributeSet attrs) {
        super(context, attrs);
        init();
    }

    public MyCustomView(Context context, AttributeSet attrs, int defStyleAttr) {
        super(context, attrs, defStyleAttr);
        init();
    }

    private void init() {
        // Initialize your custom view here
    }

    @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
        // Draw your custom view here
    }
}

// Usage in XML layout
<com.example.MyCustomView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
```

Example in Compose
Creating a custom UI component using composable functions

```kotlin
// MyCustomComposable.kt
import androidx.compose.foundation.Canvas
import androidx.compose.foundation.layout.size
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.graphics.drawscope.DrawScope
import androidx.compose.ui.unit.dp

@Composable
fun MyCustomComposable() {
    Canvas(modifier = Modifier.size(100.dp)) {
        drawCustomShape(this)
    }
}

fun drawCustomShape(drawScope: DrawScope) {
    with(drawScope) {
        drawCircle(
            color = Color.Red,
            radius = size.minDimension / 2
        )
    }
}

// Usage in a Compose function
@Composable
fun MyScreen() {
    MyCustomComposable()
}
```

Jetpack Compose

Slow Iteration and Development Cycles

The traditional Android UI toolkit often requires rebuilding and redeploying the app to see changes, slowing down development cycles

Compose supports features like live previews and live edits, allowing developers to see changes instantly and iterate more quickly

Testing and Maintainability

Writing and maintaining UI tests can be difficult with the traditional framework due to the complexity of the view hierarchy and asynchronous UI updates

Compose improves testability by providing a more predictable and straightforward way to describe UI, making it easier to write and maintain tests

Key Differences

Declarative vs Imperative

- Traditional View System: Imperative, requires manual UI updates
- Jetpack Compose: Declarative, UI updates automatically with state changes

State Management

- Traditional View System: Uses ViewModel, LiveData, and manual view binding
- Jetpack Compose: Uses State, remember, and other state management APIs within composable functions

UI Definition

- Traditional View System: XML layouts separate from logic
- Jetpack Compose: UI defined directly in Kotlin code with composable functions

Reactivity

- Traditional View System: Requires explicit observers for state changes
- Jetpack Compose: Automatically re-renders when state changes

Jetpack Compose - Getting Started

Step 1: Create a New Project

- Open Android Studio
- Click on "Start a new Android Studio project"
- Select "Empty Activity" under the "Phone and Tablet" tab
- Click "Next"

Step 2: Configure Your Project

- Name: ComposeApp
- Package name: com.example.composeapp (Replace com.example with your package name or use org.kamilimu)
- Save location: Choose your preferred location
- Language: Kotlin
- Minimum API level: API 24 (“Nougat”; Android 7.0)
- Click "Finish"

KAMI LIMU

Jetpack Compose - Understanding the code

Greeting Composable

This is a simple Composable that takes a name parameter and displays a greeting message

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

Preview

A "preview" is a feature that allows you to see how your Composable functions (UI components) will look like without running the entire application

The @Preview annotation is used to preview the Composable in the Android Studio design view

You can customize the preview by providing parameters to your Composable function within the @Preview annotation

```
@Preview(name = "Light Theme", showBackground = true)
@Composable
fun PreviewMyComposableLightTheme() {
    MyComposable(theme = Theme.Light)
}

@Preview(name = "Dark Theme", showBackground = true)
@Composable
fun PreviewMyComposableDarkTheme() {
    MyComposable(theme = Theme.Dark)
}
```

Jetpack Compose - Understanding the code

Preview Parameters

- name: Specifies the name of the preview. Helps distinguish different previews
- showBackground: Boolean flag indicating whether to show a background behind the preview (useful for seeing how your Composable looks on different backgrounds)

Preview Parameters

To run previews:

- Navigate to the file containing your Composables
- Click on the preview pane next to your code in Android Studio
- Select the preview you want to see

KamiLimu

Jetpack Compose - Modifiers

What are Modifiers?

Modifiers are a fundamental concept used to change the behavior, appearance, or layout of UI elements (Composables)

They allow you to specify how a Composable should be sized, positioned, styled, or interact with other Composables

Modifiers are immutable and can be chained together to create complex layouts and behaviors efficiently

Jetpack Compose - Modifiers

The Modifier class is the base class for all modifiers in Jetpack Compose

It provides a set of functions that can be used to modify the behavior or appearance of a Composable

Common Modifiers

- Padding: Adds padding around a Composable `Modifier.padding(16.dp)`
- Background Color: Sets the background color of a Composable `Modifier.background(Color.Blue)`
- Size: Sets the size of a Composable `Modifier.size(width = 200.dp, height = 100.dp)`
- Fill Max Size: Fills the maximum available size within its parent `Modifier.fillMaxSize()`
- Click Listener: Adds a click listener to a Composable `Modifier.clickable(onClick = { /* Handle click event */ })`

KAMI LIMU

Jetpack Compose - Modifiers

Example Usage

You can chain multiple modifiers together to achieve complex layouts and styles

By chaining multiple modifiers together, you can create rich and responsive user interfaces efficiently

Understanding modifiers is key to mastering Jetpack Compose's declarative UI development paradigm

```kotlin
@Composable
fun ClickableText(onClick: () -> Unit) {
    Text(
        text = "Click me!",
        color = Color.White,
        modifier = Modifier
            .padding(16.dp)
            .background(Color.Blue)
            .clickable(onClick = onClick)
            .padding(8.dp)
            .align(Alignment.CenterHorizontally)
    )
}
```

```kotlin
@Composable
fun MyComponent() {
    Text(
        text = "Hello, Jetpack Compose!",
        modifier = Modifier
            .padding(16.dp)
            .background(Color.Blue)
            .fillMaxWidth()
            .padding(vertical = 8.dp, horizontal = 16.dp)
    )
}
```

KAMI LIMU

Jetpack Compose - Take Home

Bull’s Eye Game App

22nd June 2024 -Take Home

Build a basic bull’s eye game ap

Objective

Using a slider, a player should closely estimate the value displayed on the screen

Requirements

Display a number on the screen that is randomly generated

Display a slider with a value range of 1 - 100

Display a button with the text “Hit”

The button should submit the value that the player estimates on the slider

Show a view that updates the score

Show a button to reset the scores

Test the app

Write unit tests to test your core functions

Thank you.
Any Question?