Unit 8 — Views and Compose

Mobile Application Development — Lecture 8

Masoud Hamad

State University of Zanzibar (SUZA)

Semester II, 2025/2026

Outline

Two UI Toolkits on Android

Views (legacy)

- XML layouts + Kotlin/Java
- `Activity`, `Fragment`, `RecyclerView`
- Mature, huge ecosystem of libraries
- Imperative: `findViewById`, `setText`

Jetpack Compose (modern)

- 100% Kotlin, no XML
- `@Composable` functions
- Declarative: UI = f(state)
- Recommended for new apps

Real-world apps rarely get a clean rewrite — they **mix both** during migration.

When Do You Need Interop?

- Existing app with thousands of View-based screens — migrating gradually
- Third-party library only offers a View (e.g., specialized charts, maps, video player)
- Platform widgets without Compose equivalents (e.g., older ad SDKs)
- Team expertise split between Views and Compose

Key insight

Compose and Views can coexist in the **same Activity, same screen, same ViewGroup**. Migration can be per-screen or per-widget.

ComposeView in an XML Layout

```
<!-- activity_main.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" ...>

    <TextView android:text="This is a View" ... />

    <androidx.compose.ui.platform.ComposeView
        android:id="@+id/compose_view"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

</LinearLayout>
```

Wiring it up in Kotlin

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

        findViewById<ComposeView>(R.id.compose_view).apply {
            setViewCompositionStrategy(
                ViewCompositionStrategy.DisposeOnViewTreeLifecycleDestroyed
            )
            setContent {
                MaterialTheme {
                    Greeting("Compose inside Views!")
                }
            }
        }
    }
}
```

Compose in a Fragment

```
class MyFragment : Fragment() {
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        return ComposeView(requireContext()).apply {
            setViewCompositionStrategy(
                ViewCompositionStrategy.DisposeOnViewTreeLifecycleDestroyed
            )
            setContent {
                MaterialTheme { MyScreen() }
            }
        }
    }
}
```

AndroidView

Embed a traditional View inside a Composable:

```
@Composable
fun MyScreen() {
    Column {
        Text("Compose above")

        AndroidView(
            factory = { ctx ->
                TextView(ctx).apply {
                    text = "Legacy TextView"
                    textSize = 20f
                }
            },
            update = { tv ->
                tv.text = "Updated from Compose"
            },
            modifier = Modifier.fillMaxWidth()
        )

        Button(onClick = { }) { Text("Compose below") }
    }
}
```

AndroidView: factory vs update

- `factory` — called **once** to create the View
- `update` — called on every recomposition that reads the state the View depends on

Typical pattern: hoist state in Compose, push it into the View via `update`.

Inflating an XML Layout

```
@Composable
fun MapScreen() {
    AndroidView(
        factory = { ctx ->
            LayoutInflater.from(ctx)
                .inflate(R.layout.map_view, null, false)
        },
        modifier = Modifier.fillMaxSize()
    )
}
```

Useful for reusing existing XML layouts or integrating third-party widgets.

ViewModel Shared Between Views and Compose

Both sides can observe the **same ViewModel**:

```
// Compose side
val count by viewModel.count.collectAsState()
Text("Count: $count")

// View side (inside an Activity)
lifecycleScope.launch {
    viewModel.count.collect { c ->
        textView.text = "Count: $c"
    }
}
```

This is the **safest migration path**: keep business logic in ViewModels and migrate one screen at a time.

CompositionLocals & Theming

- A Compose subtree inside a View uses Compose's `MaterialTheme`, not the AppCompat theme
- Use `MdcTheme` (Material 3 Components interop) to inherit theme colors from `themes.xml`
- Fonts, ripple effects, and typography can all be shared via interop themes

Incremental Migration Playbook

1. Add Compose dependencies to an existing View-based app
2. Migrate **one leaf screen** (no children) first — usually a settings screen
3. Replace `RecyclerView` adapters with `LazyColumn`
4. Wrap Compose screens behind Navigation Compose or keep them as Fragments
5. Keep ViewModels / repositories unchanged — they are UI-agnostic
6. Leave complex legacy screens for last

Pitfalls

- **Theming mismatch** — forgetting `MaterialTheme` wrapper leads to unstyled UI
- **Composition strategy** — always set `ViewCompositionStrategy` to avoid leaks
- **Click listeners across boundaries** — prefer ViewModel events over direct cross-toolkit calls
- **Double lifecycle** — Views and Compose both have lifecycles; avoid duplicated work
- **Performance** — nesting many `AndroidViews` inside `LazyColumn` is expensive

Summary

- Compose and Views interoperate freely — critical for real-world migrations
- `ComposeView` hosts Compose inside an XML layout
- `AndroidView` hosts a View inside Compose
- Share state via **ViewModels** — the safest migration anchor
- Migrate **incrementally**, screen by screen

End of the course. Next steps: WorkManager & Hilt in production, Kotlin Multiplatform, Jetpack Glance widgets, publishing to Play Store.

Thank you!