Explanations and own thoughts

Essential Android development concepts: Activity, Composable, and Modifier

Understanding three fundamental concepts unlocks modern Android development: **Activities** serve as your app's entry points and lifecycle managers, **Composables** form the building blocks of declarative UI, and **Modifiers** shape how those UI elements look and behave. Together, these concepts represent the foundation of Android Studio development with Jetpack Compose, replacing much of the complexity that previously came with XML layouts and manual view management.

Activity: your app's window into the Android system

An **Activity** is a core Android component that represents a single screen with a user interface. Unlike traditional desktop applications that start from a `main()` method, Android apps are initiated through Activity instances managed by the Android system through callback methods. Every Activity provides the window in which your app draws its UI, typically filling the entire screen though it can be smaller and float above other windows.

In modern Android development with Jetpack Compose, Activities extend `ComponentActivity` and use the `setContent { }` function to define their UI declaratively. This differs from traditional development where Activities extended `AppCompatActivity` and used `setContentView(R.layout.activity_main)` to inflate XML layouts.

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MyAppTheme {
                Surface(modifier = Modifier.fillMaxSize()) {
                    Greeting("Android Developer")
                }
            }
        }
    }
}
```

The Activity lifecycle governs how your app responds to user behavior

The Activity class provides **six core lifecycle callbacks** that manage transitions between states: `onCreate()`, `onStart()`, `onResume()`, `onPause()`, `onStop()`, and `onDestroy()`, plus `onRestart()` for returning activities.

onCreate() fires when the system first creates the Activity. This is where you perform one-time initialization: set up your UI with `setContent { }`, instantiate class-scope variables, associate ViewModels, and recover any saved instance state. This callback always triggers `onStart()` immediately afterward.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // Recover saved state if available
    count = savedInstanceState?.getInt(KEY_COUNT, 0) ?: 0

    setContent {
        CounterScreen(count = count, onIncrement = { count++ })
    }
}
```

onStart() executes when the Activity becomes visible to the user. Use this callback for final preparations before coming to the foreground: initialize UI-related code, register broadcast receivers for UI changes, and start animations. This callback may fire multiple times as users navigate away and return.

onResume() signals that the Activity has come to the foreground and is now interactive. This is where your app's core functionality runs—start camera previews, resume media playback, and refresh data. The Activity remains in this state until something takes focus away, such as a phone call, another activity launching, or the screen turning off.

onPause() provides the first indication that the user is leaving your Activity. This fires when the Activity loses focus—another activity comes to the foreground, a dialog appears, or the app enters multi-window mode. Pause operations that shouldn't continue while partially visible, like releasing sensors that affect battery life. Importantly, avoid heavy operations here since this callback is brief.

onStop() indicates the Activity is completely hidden from the user. This is the appropriate place for CPU-intensive shutdown operations and database writes. The Activity object remains in memory with its state preserved, ready to be restored if the user returns.

onDestroy() executes before the Activity is destroyed, either because the user dismissed it or the system needs to reclaim resources (such as during a configuration change like screen rotation). Release any resources not already released by earlier callbacks and perform final cleanup.

```
class MainActivity : ComponentActivity() {
    private var count = 0

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        count = savedInstanceState?.getInt("count_key", 0) ?: 0
        setContent { CounterScreen(count) }
    }

    override fun onStart() {
        super.onStart()
        Log.d("Lifecycle", "Activity is now visible")
    }

    override fun onResume() {
        super.onResume()
        Log.d("Lifecycle", "Activity is interactive")
    }

    override fun onPause() {
        super.onPause()
        Log.d("Lifecycle", "Activity losing focus")
    }

    override fun onStop() {
        super.onStop()
        Log.d("Lifecycle", "Activity no longer visible")
    }

    override fun onDestroy() {
        super.onDestroy()
        Log.d("Lifecycle", "Activity being destroyed")
    }

    override fun onSaveInstanceState(outState: Bundle) {
        super.onSaveInstanceState(outState)
        outState.putInt("count_key", count)
    }
}
```

Modern apps favor single-activity architecture

Google officially recommends that new Android applications use **single-activity architecture** with the Navigation component. Rather than creating multiple Activities for different screens, you use one Activity that hosts multiple Composable screens managed by a NavController. This approach simplifies navigation, eliminates status bar flickering during transitions, and provides better backstack management with built-in deep linking support.

```
@Composable
fun MyAppNavigation() {
    val navController = rememberNavController()

    NavHost(navController = navController, startDestination = "home") {
        composable("home") {
            HomeScreen(onNavigateToDetails = { id ->
                navController.navigate("details/$id")
            })
        }
        composable("details/{itemId}") { backStackEntry ->
            DetailsScreen(itemId = 
backStackEntry.arguments?.getString("itemId"))
        }
    }
}
```

Composable functions transform data into UI declaratively

A **Composable function** is a Kotlin function annotated with `@Composable` that defines part of your UI. These functions are the fundamental building blocks of Jetpack Compose, describing *what* the UI should look like rather than *how* to construct it step by step. Composables that emit UI don't return values—they describe the current screen state.

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

```
@Composable
fun MessageCard(message: Message) {
    Row(modifier = Modifier.padding(8.dp)) {
        Image(
            painter = painterResource(R.drawable.profile),
            contentDescription = null
        )
        Column {
            Text(text = message.author)
            Text(text = message.body)
        }
    }
}
```

The `@Composable` annotation informs the Compose compiler that this function converts data into UI. Composables can only be called from other Composable functions, creating a tree of UI components that Compose manages automatically.

Declarative UI inverts the traditional Android paradigm

Traditional Android development used an **imperative** approach: you found views by ID, then called methods to modify their internal state. This grew increasingly complex as more views needed coordination.

```
// Traditional imperative approach
val textView = findViewById<TextView>(R.id.greeting)
textView.setText("Hello, $name")  // Mutating widget state
```

Jetpack Compose uses a **declarative** approach: you describe what the UI should look like for a given state, and Compose handles updates automatically. Widgets are stateless—you call the same function with new arguments when state changes.

```
// Compose declarative approach
@Composable
fun Greeting(name: String) {
    Text(text = "Hello, $name")  // UI reflects current state
}
```

This eliminates entire categories of bugs caused by forgetting to update views or creating conflicting states. Compose intelligently determines what changed and updates only the affected portions of the UI.

Recomposition updates your UI when state changes

Recomposition is the process of calling Composable functions again when their inputs change. Compose tracks state through special constructs like `mutableStateOf` and `remember`, triggering recomposition when that state changes.

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

    Button(onClick = { count++ }) {
        Text("Clicked $count times")
    }
}
```

When `count` changes, Compose schedules recomposition for only the affected Composables. This **smart recomposition** skips functions whose inputs haven't changed, potentially runs in parallel across multiple threads, and can be canceled and restarted if parameters change during execution.

```
@Composable
fun NameList(header: String, names: List<String>) {
    Column {
        Text(header)  // Only recomposes when header changes
        HorizontalDivider()
        LazyColumn {
            items(names) { name ->
                NameItem(name)  // Each item recomposes independently
            }
        }
    }
}
```

Composables must follow three critical rules

Composable functions must be **fast** because they might execute every frame during animations. Move expensive operations to background coroutines and pass computed values as parameters.

They must be **idempotent**, producing the same output for the same inputs. Avoid using `random()` or relying on execution timing within Composables.

They must be **free of side effects**—don't write to shared properties, update ViewModel observables directly, or modify global variables inside a Composable body.

```
// ❌ BAD - Side effect inside composable
@Composable
fun BadCounter(items: List<String>) {
    var count = 0
    for (item in items) {
        Text("Item: $item")
        count++  // Side effect!
    }
}

// ✅ GOOD - No side effects
@Composable
fun GoodCounter(items: List<String>) {
    Column {
        items.forEach { item -> Text("Item: $item") }
        Text("Count: ${items.size}")
    }
}
```

State hoisting—lifting state up to a caller and passing it down as parameters—creates reusable, testable Composables:

```
// Stateless composable (more reusable)
@Composable
fun HelloContent(name: String, onNameChange: (String) -> Unit) {
    Column(modifier = Modifier.padding(16.dp)) {
        Text("Hello, $name")
        OutlinedTextField(
            value = name,
            onValueChange = onNameChange,
            label = { Text("Name") }
        )
    }
}

// Stateful wrapper
@Composable
fun HelloScreen() {
    var name by rememberSaveable { mutableStateOf("") }
    HelloContent(name = name, onNameChange = { name = it })
}
```

Modifiers decorate and configure Composables

A **Modifier** is an object that decorates or augments a Composable, controlling its size, layout, behavior, appearance, and interactivity. Modifiers are standard Kotlin objects that you chain together using a fluent API, with each modifier affecting what comes after it.

```
@Composable
fun StyledGreeting(name: String) {
    Text(
        text = "Hello, $name",
        modifier = Modifier
            .fillMaxWidth()
            .padding(16.dp)
            .background(Color.LightGray, RoundedCornerShape(8.dp))
    )
}
```

Layout modifiers control size and spacing

Size modifiers constrain how large a Composable can be:

```
Modifier.size(100.dp)                       // Fixed square
Modifier.size(width = 200.dp, height = 100.dp)  // Fixed rectangle
Modifier.fillMaxWidth()                     // Fill available width
Modifier.fillMaxSize()                      // Fill all available space
Modifier.fillMaxWidth(0.5f)                 // Fill 50% of width
Modifier.widthIn(min = 50.dp, max = 200.dp)  // Constrained range
```

Padding adds space inside a Composable's bounds. Unlike traditional Android, Compose has no separate margin concept—you achieve margins by ordering padding before background modifiers:

```
Modifier.padding(16.dp)                                 // All sides
Modifier.padding(horizontal = 16.dp, vertical = 8.dp)  // Symmetric
Modifier.padding(start = 8.dp, top = 16.dp, end = 8.dp, bottom = 16.dp)
```

Appearance modifiers shape visual presentation

Background applies colors, gradients, or shapes behind content:

```
Modifier.background(Color.Blue)
Modifier.background(Color.Blue, shape = RoundedCornerShape(8.dp))
Modifier.background(
    brush = Brush.horizontalGradient(listOf(Color.Red, Color.Blue)),
    shape = CircleShape
)
```

Border and **clip** define outlines and content boundaries:

```
Modifier.border(2.dp, Color.Red, RoundedCornerShape(8.dp))
Modifier.clip(RoundedCornerShape(8.dp))
Modifier.clip(CircleShape)
```

Shadow and **alpha** add depth and transparency:

```
Modifier.shadow(elevation = 4.dp, shape = RoundedCornerShape(8.dp))
Modifier.alpha(0.5f)  // 50% transparent
```

Interaction modifiers respond to user input

Clickable makes any Composable respond to taps:

```
Modifier.clickable { viewModel.onItemSelected() }
Modifier.combinedClickable(
    onClick = { /* single tap */ },
    onLongClick = { /* long press */ },
    onDoubleClick = { /* double tap */ }
)
```

Scrollable enables content to scroll:

```
Modifier.verticalScroll(rememberScrollState())
Modifier.horizontalScroll(rememberScrollState())
```

Modifier order determines behavior

The sequence of chained modifiers critically affects the result. Each modifier wraps around those that follow, creating layers of behavior:

```
// Entire area including padding is clickable
Modifier
    .clickable { onClick() }
    .padding(16.dp)

// Only inner content is clickable, padding area is NOT
Modifier
    .padding(16.dp)
    .clickable { onClick() }
```

Think of modifiers as wrapping paper: the first modifier wraps everything, the second wraps what's inside that, and so on. This enables creating margin-like effects:

```
Box(
    modifier = Modifier
        .padding(8.dp)           // Outer spacing (like margin)
        .background(Color.White, RoundedCornerShape(8.dp))
        .border(1.dp, Color.Gray, RoundedCornerShape(8.dp))
        .padding(16.dp)          // Inner spacing (like padding)
        .fillMaxWidth()
) {
    Text("Card content")
}
```

Best practices maximize modifier effectiveness

Always accept a **modifier parameter** in custom Composables, defaulting to an empty `Modifier`:

```
@Composable
fun CustomCard(
    title: String,
    modifier: Modifier = Modifier  // Always include this
) {
    Card(modifier = modifier) {
        Text(title, modifier = Modifier.padding(16.dp))
    }
}
```

Reuse modifiers for performance in frequently updating UIs, especially within lazy lists:

```
// Define once outside composition
val itemModifier = Modifier
    .padding(bottom = 12.dp)
    .size(216.dp)
    .clip(CircleShape)

@Composable
fun AuthorList(authors: List<Author>) {
    LazyColumn {
        items(authors) { author ->
            AsyncImage(
                model = author.imageUrl,
                modifier = itemModifier  // Reused, no reallocation
            )
        }
    }
}
```

Some modifiers only work within specific scopes. `weight()` is available only in `RowScope` or `ColumnScope`, while `matchParentSize()` works only in `BoxScope`:

```
Row {
    Text("Left", modifier = Modifier.weight(1f))
    Text("Right", modifier = Modifier.weight(1f))
}
```

Conclusion

These three concepts form the backbone of modern Android development. **Activities** manage your app's lifecycle and serve as the container for your UI, though modern apps typically use just one Activity with navigation handled by Composables. **Composable functions** replace XML layouts with a declarative Kotlin-based approach that automatically updates when state changes, eliminating manual view management. **Modifiers** provide a chainable, ordered system for styling and configuring those Composables without inheritance or complex XML attributes.

The shift from imperative to declarative UI fundamentally changes how you think about Android development. Rather than manipulating widget state step by step, you describe the UI as a function of your app's state and let Compose handle the updates. Combined with single-activity architecture and the Navigation component, this creates apps that are easier to build, test, and maintain.

Regarding the code file:

The @Preview annotation in Android Studio allows you to visualize your Jetpack Compose functions (composables) directly within the IDE without needing to build and run your app on a physical device or emulator.

Key features include:

•Instant Feedback: You can see UI changes in real-time as you modify your code.

•Multiple Configurations: You can create multiple preview functions to see how your UI looks in different states (e.g., light/dark mode, different screen sizes, or different languages).

•Interactivity: The Interactive Mode allows you to click and interact with the UI elements directly in the preview.

•Deployment: You can quickly deploy just that specific composable to a device using the "Run Preview" button.

In the code, the GreetingPreview function uses @Preview(showBackground = true) to show you what the Greeting composable looks like with a default background in the Design or Split view of your editor.

Column composable has been used inside the code for building simple layout and spacer layout for padding between elements.

•fillMaxSize(): Makes the Column take up the entire screen.

•padding(): Added both outer padding and internal padding for better spacing.

•border(): Added a gray border with rounded corners around the main content.

•background(): Applied a light yellow background color specifically to the "Hello" text.

•fillMaxWidth(): Applied to the Button so it spans the full width of its container.

These changes show how you can chain modifiers to control layout, spacing, and styling.

•Scaffold: Used as the top-level structure in MainScreen to provide a TopAppBar.

•TextField: Added to Greeting to allow user input (with state managed by remember).

•Row: Used to align text items and a button horizontally.

•Card: Wraps the greeting text to give it elevation and a defined background.

•Box: Demonstrates layering and custom alignment (the cyan box with "Inside a Box").

I have been helped by the **Gemini-3-Flash-Preview and Claude Opus 4.5 AI models** for completing this 1st week assignment.

Thank you for your time.