Lesson 1: Android Basics

Kotlin, Coroutines, Android Framework & Jetpack Compose

Adrián Catalán

adriancatalan@galileo.edu

Agenda

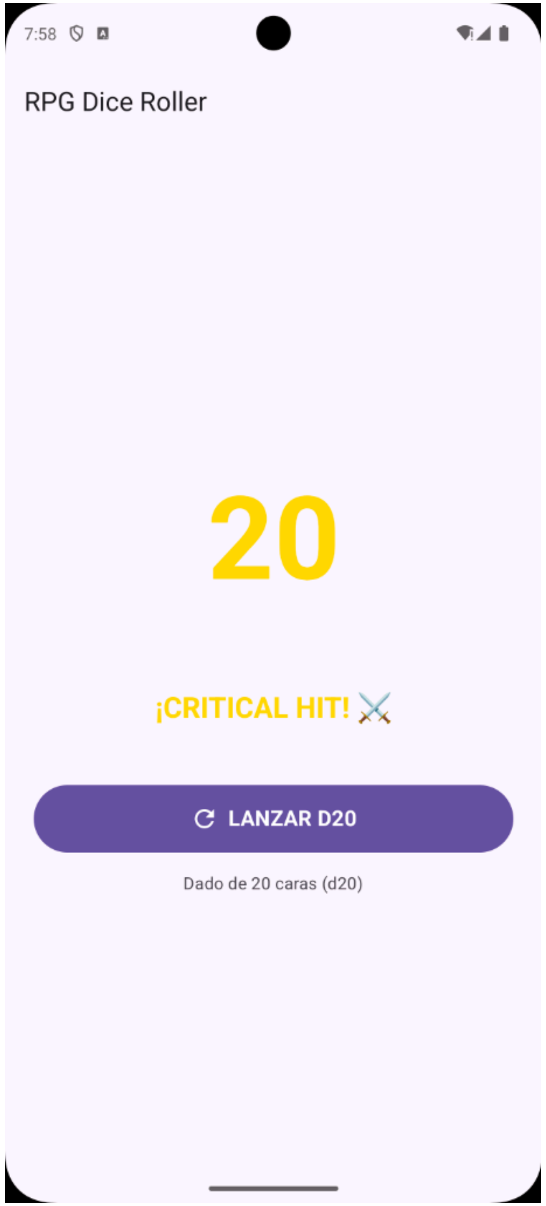
1. Module App
2. Kotlin & Coroutines
3. Android Activities & Logcat
4. Compose UI Basics
5. Deep Dive
6. Challenge Lab

Dice Roller App

We are building a Digital D20 for RPG games.

Core Requirements:

1. **Visual Feedback**: Show current dice value (1-20).
2. **Interaction**: "Roll" button that animates the numbers.
3. **Feedback**: Critical Hit (20) and Critical Miss (1) alerts.
4. **Performance**: Non-blocking animation.



RPG Dice Roller

20

¡CRITICAL HIT! ⚔️

Dado de 20 caras (d20)

1. Kotlin & Coroutines

Kotlin Language Highlights

Kotlin the preferred language for Android

1. **Null Safety**:

```kotlin
var name: String = "Dice" // Can NEVER be null
var label: String? = null  // Can be null
// label.length // Compiler Error! Must check for null first
```

2. **Immutability First**:

```kotlin
val MAX = 20 // 'val' (Value) vs 'var' (Variable)
```

3. **Syntactic Convenience**:
    - **Data Classes**: Automatic `toString`, `equals`.
    - **Smart Casts**: Automatic type casting checks.
    - **Extensions**: Add functions to existing classes.

Kotlin: Higher-Order Functions & Lambdas

Functions are "First Class Citizens". We can pass them as variables.

The Lambda Syntax `{ }`:

```
// A variable that holds a function
val onClick = { println("Clicked!") }

// Passing a function to a Button
Button(onClick = { rollDice() }) { ... }
```

This allows us to easily define "what happens next" (Callbacks) without creating anonymous interface classes like in Java (`new OnClickListener...`).

Applied Syntax in `MainActivity.kt`

We use these features directly in our app logic:

```kotlin
// Compile-time constant (Static)
private const val MAX_DICE_VALUE = 20

// Mutable state variable (var) that holds an Integer
var diceValue by remember { mutableIntStateOf(1) }

// Range generation
val randomValue = (1..20).random()
```

Theory: The Main Thread & Blocking

The Golden Rule of Android: Do not block the Main Thread (UI Thread).

- **Main Thread Responsibilities**: Drawing frames (60fps), handling touches, executing UI code.
- **ANR**: If blocked for >5s, Android shows "Application Not Responding".

Analogy: The Blocking Barista

Scenario (1 Thread):

1. Customer A orders.
2. **Barista makes coffee (waits 3 mins).**
3. *Queue stops completely*.
4. Barista serves A.
5. Barista takes Customer B's order.

> **Result**: The "App" (Coffee Shop) freezes. Customers (Users) leave.

Analogy: The Non-Blocking Barista

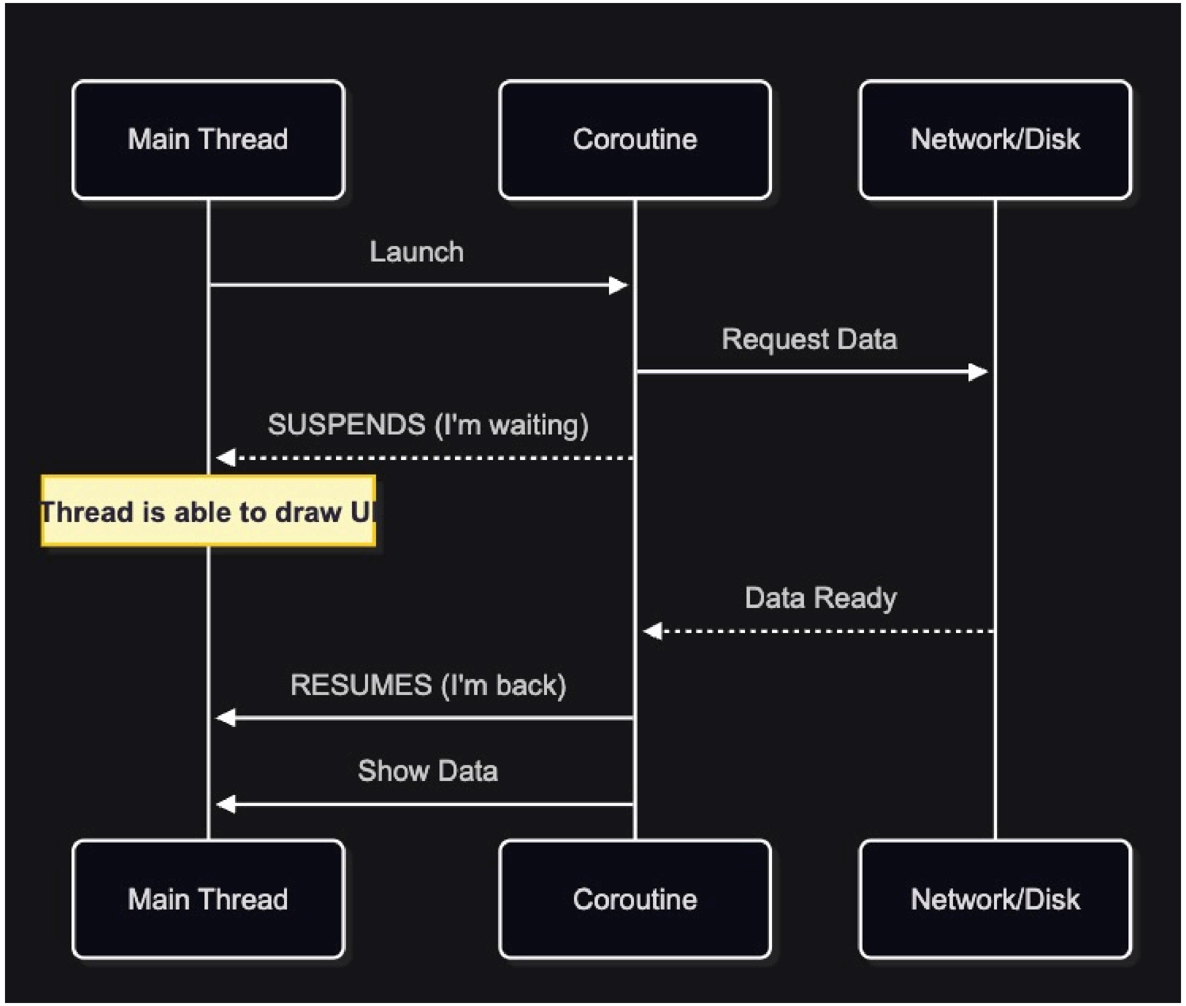
Scenario (Coroutines):

1. Customer A orders.
2. **Barista gives order to machine (Async).**
3. *Barista immediately takes Customer B's order.*
4. Machine beeps (Callback/Resume).
5. Barista serves A.

> **Result**: Valid UI. Happy Users.

Visualizing "Suspend"

The magic happens when the thread is **freed**.



```
// BLOCKING (Bad)
Thread.sleep(1000)
// The entire app freezes for 1 second. No clicks, no draws.

// NON-BLOCKING (Good - Coroutines)
delay(1000)
// The function pauses, but the Main Thread goes back to work.
// Users can click buttons while we wait.
```

Coroutines

Coroutines allow us to write asynchronous code that *looks* sequential but is **non-blocking**.

- `suspend` **functions**: Functions that can "pause" execution without blocking the thread.
- `delay()`: The coroutine equivalent of sleep.

Generic Suspend Example (Network):

```kotlin
// "suspend" marker tells Kotlin this function might pause
suspend fun fetchUserData(): User {
    // Allows Main Thread to continue while we wait for network
    val json = networkClient.get("users/1")
    return parse(json)
}
```

Dispatchers: Who does the work?

Coroutines need a thread to run on. **Dispatchers** decide which thread.

1. `Dispatchers.Main`:
   - **Use for**: UI updates, animations, handling clicks.
   - *Rule*: Fast operations only.
2. `Dispatchers.IO`:
   - **Use for**: Network requests, Database, Reading files.
   - *Rule*: Anything that waits.
3. `Dispatchers.Default`:
   - **Use for**: Heavy CPU calculations (Image processing, algorithms).

Coroutine Scopes: Android Lifecycle

A Scope controls **how long** a coroutine lives.

`lifecycleScope` (Activity/Fragment):

- Bound to the screen's lifecycle.
- **Automatic Cancellation**: If the user closes the app, the download stops.
- **Use Case**: One-off background tasks like "Compressing Image" or "Saving to Disk".

Coroutine Scopes: Compose

`rememberCoroutineScope`:

- Bound to the Composable function in the UI tree.
- **Crucial for Callbacks**: You CANNOT call `suspend` functions inside a normal `onClick` lambda. You need this scope to "Launch" them.

```
val scope = rememberCoroutineScope()
Button(onClick = {
    scope.launch { rollDice() } // Bridge from Sync to Async
}) { ... }
```

Applied Coroutines in `rollDice`

```kotlin
// Scope: Survives recompositions, cancelled on screen exit
val scope = rememberCoroutineScope()

fun rollDice() {
    scope.launch { // Fire and Forget
        isRolling = true
        repeat(15) {
            diceValue = (1..20).random()

            // Pauses this coroutine logic, lets UI redraw
            delay(80)
        }
        isRolling = false
    }
}
```

Real World Patterns

1. **API Calls (Network)**

```
suspend fun login(user: User) = withContext(Dispatchers.IO) {
    api.post("/login", user)
}
```

2. **Database (Room)**

```
// Room generates suspend functions automatically
@Dao interface UserDao {
    @Insert suspend fun add(user: User)
}
```

3. **Simple Timers**

```
LaunchedEffect(Unit) {
    delay(3000) // Wait 3s then close screen
    navController.popBackStack()
}
```

Resources: Kotlin & Coroutines

Want to learn more?

- Kotlin Docs: Coroutines Guide - The official bible.
- Android Developers: Kotlin Coroutines - Best practices for Android.
- Kotlin Playground - Try code in your browser.
- Codelab: Use Coroutines - Hands-on practice.
- Flow Documentation - Reactive streams.

2. Android

Intro to Android Framework

Android is a Linux-based OS with a Java/Kotlin framework layer on top.

Project Structure (`app/src/main`):

```
src/main/
├── AndroidManifest.xml      <-- App Identity (Permissions, Activities)
├── java/
│   └── com/example/dice/
│       └── MainActivity.kt  <-- Kotlin Source Code
└── res/
    ├── drawable/            <-- Images/Icons
    └── values/              <-- Strings, Colors, Themes (XML)
```

The Build System: Gradle

Android apps use **Gradle** to compile and manage dependencies.

`build.gradle.kts` (Kotlin Script):

```kotlin
plugins {
    alias(libs.plugins.android.application)
}

android {
    namespace = "com.curso.android.dice"
    compileSdk = 34 // Android 14
}

dependencies {
    // External Libraries
    implementation(libs.androidx.core.ktx)
    implementation(libs.androidx.compose.material3)
}
```

MainActivity: The Entry Point

In modern Compose apps, the `Activity` is simply the **Container** for your Composables.

From Code Comments:

> `ComponentActivity`: Modern AndroidX base Activity that supports Jetpack Compose and is lighter than `AppCompatActivity`.

```kotlin
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge() // Draw behind status bars
        setContent {
            MaterialTheme { DiceRollerScreen() }
        }
    }
}
```

Lifecycle: The Life of an App

The OS manages the app process. It's not just "Open" or "Closed".

Visualizing the Lifecycle:

```
onCreate() → onStart() → onResume() → [RUNNING] → onPause() → onStop() → onDestroy()
    ↑                                                              ↓
    └──────────────────────────────────────────────────────────────┘
```

- **onCreate**: Called ONCE. Setup UI.
- **onStart/onResume**: App becomes visible and interactive.
- **onPause/onStop**: App goes to background (user pressed Home).
- **onDestroy**: Cleanup.

Logcat: Verification

Logcat is your window into the app's soul.

Log Levels:

- `Log.e` (Error)
- `Log.w` (Warning)
- `Log.i` (Info)
- `Log.d` (Debug) -> **Most used for development**

```kotlin
// Code Sample
private const val TAG = "MainActivity"
Log.d(TAG, "onCreate: Edge-to-Edge enabled")
```

Resources: Android Framework

- The Activity Lifecycle - Detailed diagrams.
- Analyze with Logcat - Master the logs.
- Guide to App Architecture - How to structure apps.
- Android Studio Debugging - Breakpoints and inspectors.
- Codelab: Android Basics - Full course.

3. Compose UI Basics

Imperative vs. Declarative UI

Imperative (The Old Way - XML):

- We manually manipulate the view hierarchy.
- "Find the TextView, then set its text."
- *Risk*: State mismatch. The UI might not show the real data if we forget to update it.

Declarative (The New Way - Compose):

- We describe the UI based on the current *State*.
- "The UI *is* a representation of the data."
- *Benefit*: Single Source of Truth.

Declarative Structure in Compose

We build UI by nesting functions.

```
Scaffold { padding ->
    Column( // Vertical Layout
        modifier = Modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        // Child 1: The Dice
        Box(contentAlignment = Alignment.Center) {
             Text(text = diceValue.toString(), fontSize = 96.sp)
        }
    }
}
```

> `Modifiers` are chained sequentially.

State & Recomposition

What is Recomposition?

> "Recomposition is the process of re-executing a @Composable function when its state changes. It's like an automatic UI 'refresh'."

State breakdown from Code:

```
// 1. remember: Keeps value between recompositions
// 2. mutableStateOf: Observable State object
var diceValue by remember { mutableIntStateOf(1) }

// Usage:
Text(text = diceValue.toString())
```

The Big Three: Layouts

How do we arrange elements?

1. Column (Vertical)

```
[ Element 1 ]
[ Element 2 ]
[ Element 3 ]
```

2. Row (Horizontal)

```
[ A ] [ B ] [ C ]
```

3. Box (Stack/Overlay)

```
[ Back  ]
[ Front ]
```

Alignment & Arrangement

- **Arrangement**: How children are distributed along the main axis.
  - `Arrangement.Center`, `Arrangement.SpaceBetween`.
- **Alignment**: How children are positioned on the cross axis.
  - `Alignment.CenterHorizontally` (in Column).

Example:

```
Column(
    modifier = Modifier.fillMaxSize(), // Take full screen
    verticalArrangement = Arrangement.Center, // Center vertically
    horizontalAlignment = Alignment.CenterHorizontally // Center horizontally
) { ... }
```

Modifiers

Modifiers tell a UI element **how** to lay out, draw, or behave. Order matters!

```
Box(
    modifier = Modifier
        .size(100.dp)
        .background(Color.Red)   // Red background
        .padding(16.dp)          // Padding INSIDE the red box
        .background(Color.Blue)  // Blue box INSIDE the padding
)
```

Common Modifiers:

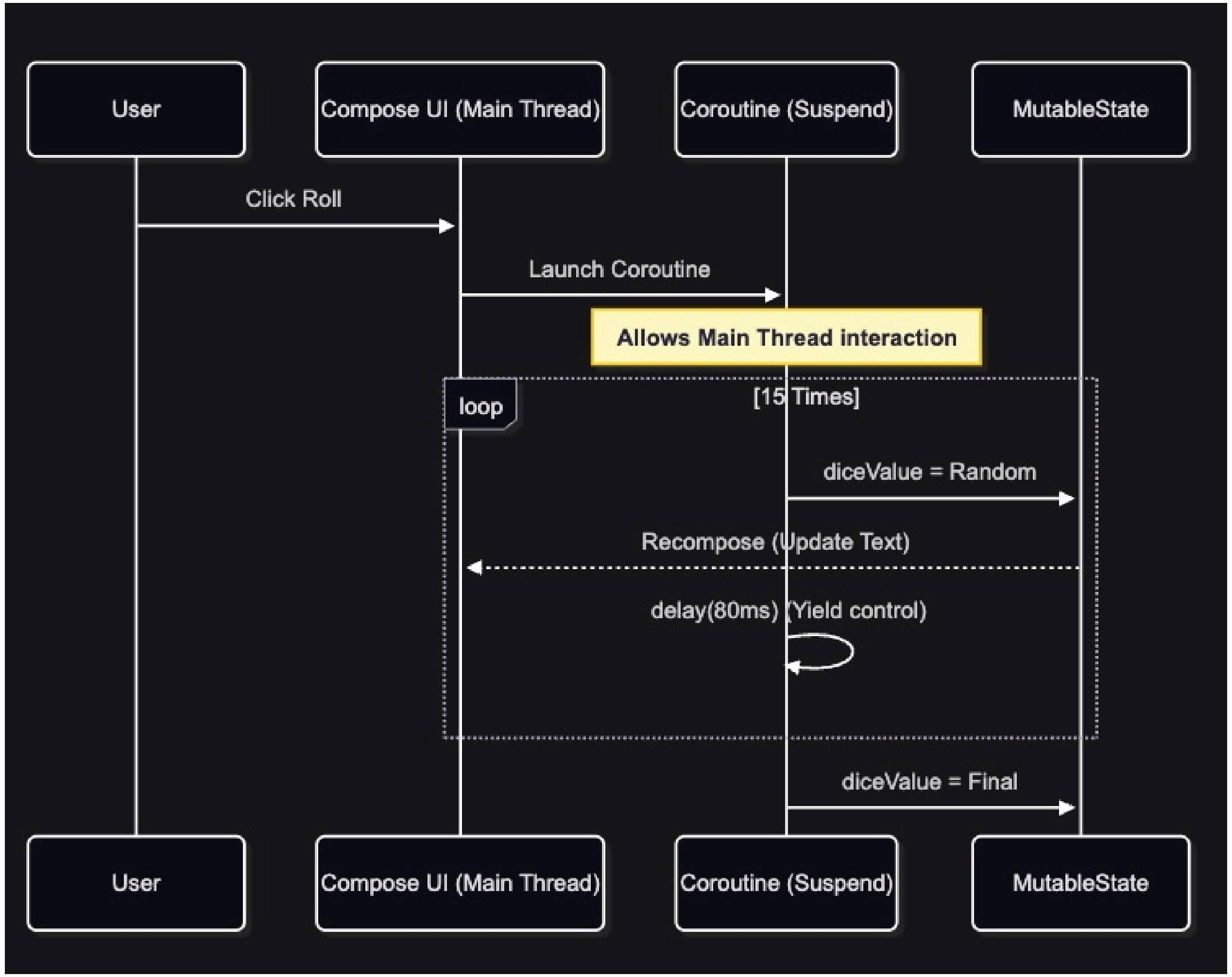
- `.padding(8.dp)`
- `.fillMaxWidth()` / `.fillMaxSize()`
- `.clickable { ... }`

Resources: Jetpack Compose

- Compose Layouts Basics - Columns, Rows, Boxes.
- Thinking in Compose - Shift your mindset.
- List of Compose Modifiers - Cheatsheet.
- State in Compose - Deep dive on `remember`.
- Codelab: Jetpack Compose Basics - *Start here* Codelab

4. Deep Dive

1. The "Roll" Flow Diagram



2. Threading Model

The secret is separating **Work** from **Drawing**.

- **UI Thread (Main)**:
    - Responsibilities: Drawing pixels, listening for touches.
    - Status: **BUSY** only when Recomposing (few ms). **IDLE** while waiting for delay.
- **Coroutine**:
    - Status: **SUSPENDED** during `delay(80)`. It does NOT occupy the thread.
- **Result**:
    - The Main Thread is free 99% of the time, even during the animation.
    - Frames render at 60fps.

3. Unidirectional Data Flow (UDF)

The data only moves one way, making the app predictable.

Concept:

- **State flows DOWN**: From helper variables -> `Text()` composables.
- **Events flow UP**: From `Button` clicks -> lambda functions -> Logic.

Applied Code:

```
     [ DiceRollerScreen ]
         /         \
(State: 5)         (Event: onClick)
    |                   ^
    v                   |
 [ Text ]           [ Button ]
```

We never modify the widget directly. We modify the State, and the widget updates itself.

4. Recomposition & Stability

Compose is optimized to do the minimum work possible.

- **Smart Recomposition**: When `diceValue` changes, Compose skips the `Button`. Why? Because `isRolling` didn't change, only the number did.
- **Stability**: Compose knows `Int` and `String` are immutable. It can safely skip checking them if the reference is the same.
- **Lifecycle Awareness**: Our `rememberCoroutineScope` is tied to the UI. If the user rotates the screen (destroying the Activity), the scope cancels the animation automatically, preventing crashes.

5. Under the Hood: The State Machine

How does `suspend` work if Java doesn't have it?

The **Kotlin Compiler** rewrites your code.

1. Creates a `Continuation` class (State wrapper).
2. Turns your function into a giant `switch(state)` statement.
3. **Label 0**: Execute until first delay. Return `SUSPEND`.
4. **Label 1**: (Called when delay finishes) Jump back here with existing variables restored.

> It's not magic, it's efficient code generation.

6. Challenge Lab

Lab: RPG Character Sheet

Goal: Transform the simple Dice Roller into a **Character Creation Screen**.

Scenario:
You are building the "New Character" screen for a D&D app. Players need to roll for their base stats: **Strength**, **Dexterity**, and **Intelligence**.

Requirements

1. **Three Stat Rows**:
   - Instead of one big number, create 3 rows (Str, Dex, Int).
   - Each row has a Label ("STR"), a Value ("14"), and a "Roll" button.
2. **Total Score**:
   - Display the **Sum** of all 3 stats at the bottom.
3. **Validation Rule (Logic)**:
   - If the Total < 30, show a "Re-roll recommended!" message in Red.
   - If Total >= 50, show "Godlike!" in Gold.
4. **Visual Polish**:
   - Use `Card` or `Row` to make the stat lines look professional.
   - Add logical padding.

Step Playbook

Step 1: Refactor (Applied Layouts)

- Create a reusable Composable function:

```
@Composable
fun StatRow(name: String, value: Int, onRoll: () -> Unit) { ... }
```

Step 2: State Management (Hoisting)

- Lift the state up to the parent screen.

```
var str by remember { mutableIntStateOf(10) }
var dex by remember { mutableIntStateOf(10) }
...
```

Step 3: Logic

- Calculate `val total = str + dex + int` inside parent and pass it to a footer text.

Part 2: The Traffic Light

Goal: Create a **New Project** to master State and Effects.

Requirements:

1. **State**: Define `enum class Light { Red, Yellow, Green }`.
2. **UI**: Draw 3 Circles (`Box` with `clip(CircleShape)`).
   - Active light = Bright Color / Inactive = Gray.

3. **Logic (The Brain)**:

```
LaunchedEffect(Unit) { // Runs ONCE when app starts
    while(true) { // Infinite Loop
        state = Red
        delay(2000)
        state = Green
        delay(2000)
        state = Yellow
        delay(1000)
    }
}
```