

# Workshop: The Core Loop (State & Lists)

**Class 4: Interactive UI Time:** 75 Minutes

## The Goal

In previous classes, we built "Static" UI—screens that look good but don't *do* anything (like a painting). Today, we build "Dynamic" UI—apps that react to the user.

We will build the **Task List** from our Figma file.

## The "Core Loop" Theory

Before we open Android Studio, understand the loop that powers every modern app:

1. **Event:** The user clicks a checkbox.
  2. **Update State:** We change a variable (e.g., `isChecked = true`).
  3. **Recomposition:** Jetpack Compose notices the variable changed. It destroys the old UI and instantly "repaints" the screen to match the new data.
- 

## Phase 1: Project Creation

Let's start from zero.

1. Open **Android Studio**.
2. Click **New Project**.
3. **Crucial Step:** Select **Empty Activity**.
  - *Note:* Ensure the icon shows the Compose logo (green/blue triangle). Do **not** select "Empty Views Activity".
4. **Name:** `TaskManager`.
5. **Package Name:** `com.example.taskmanager` (or your custom domain).
6. **Language:** Kotlin.
7. **Build Configuration:** Recommended (Kotlin DSL) is fine.
8. Click **Finish**.

*Wait for Gradle to sync (the loading bars at the bottom right).*

---

## Phase 2: The Data Model

**Concept: Data Classes** UI is just a reflection of data. Before we draw a single pixel, we need to define *what* a Task is. In Kotlin, we use a `data class`. This is a container that holds pure information—no UI logic, just facts.

1. Open `MainActivity.kt`.
2. Scroll to the very bottom of the file (outside the `MainActivity` class).
3. Add this code:

```
// A blueprint for what a "Task" looks like in our memory
data class TaskItem(
    val id: Int,
    val label: String,
    val dueTime: String
)
```

---

## Phase 3: The List Container (LazyColumn)

**Concept: LazyColumn vs. Column**

- **Column:** Loads *everything* at once. If you have 1,000 tasks, it tries to draw 1,000 views immediately. Your phone will freeze and crash.
  - **LazyColumn:** The modern "RecyclerView". It only draws the items currently visible on the screen. As you scroll down, it recycles the memory from the top to draw the new items at the bottom. **Always use this for lists.**
1. Delete the default `Greeting` and `GreetingPreview` functions.
  2. Create a new Composable called `TaskScreen`.
  3. We will generate a fake list of data to test our UI.

```
@Composable
fun TaskScreen() {
    // 1. DUMMY DATA
    // We create a list of objects to represent our "Database"
    val tasks = listOf(
        TaskItem(1, "Complete Lab 4", "Today, 5:00 PM"),
        TaskItem(2, "Buy Groceries", "Tomorrow, 10:00 AM"),
        TaskItem(3, "Call Mom", "Sunday"),
        TaskItem(4, "Fix Android Bug", "ASAP"),
        TaskItem(5, "Go to the Gym", "Tonight")
    )
    // 2. THE LIST CONTAINER
```

```

LazyColumn(
    modifier = Modifier
        .fillMaxSize() // Take up the whole screen
        .background(Color(0xFFF5F5F5)), // Light Gray background
    contentPadding = PaddingValues(16.dp), // Add margin around the list itself
    verticalArrangement = Arrangement.spacedBy(8.dp) // Add 8dp gap between every item
) {
    // 3. THE MAGIC LOOP
    // 'items' takes our list and runs the code block for every single item
    items(tasks) { task ->
        // For now, just print the text to prove it works.
        // We will replace this with a fancy row later.
        Text(text = "Todo: ${task.label}")
    }
    /*if you are on an old version of Android, the code must change to:
    items(tasks.count()) { index ->
        // For now, just print the text to prove it works.
        // We will replace this with a fancy row later.
        Text(text = "Todo: ${tasks[index].label}")
    }
    */
}

```

4. **Important:** Scroll up to the `onCreate` method and call `TaskScreen()` inside `setContent`.

```

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContent {
        TaskScreen()
    }
}

```

*Run the app now. You should see a boring, text-only list.*

---

## Phase 4: The Visual Design (Material Surface)

**Concept: Surface & Elevation** In Material Design, we don't just use white rectangles. We use **Surface**. **Surface** handles the heavy lifting of:

- **Clipping:** Making rounded corners (`shape`).
  - **Shadows:** Making the card look 3D (`shadowElevation`).
  - **Colors:** Handling background colors properly.
1. Create a new Composable at the bottom of the file called `TaskRow`.
  2. We pass a `TaskItem` into it as a parameter.

```
@Composable
fun TaskRow(task: TaskItem) {
    Surface(
        modifier = Modifier.fillMaxWidth(), // Card takes full width
        shape = RoundedCornerShape(12.dp), // Smooth corners
        color = Color.White,
        shadowElevation = 2.dp // Subtle shadow effect
    ){
        // "Row" puts the checkbox and text side-by-side
        Row(
            modifier = Modifier.padding(16.dp), // Padding INSIDE the card
            verticalAlignment = Alignment.CenterVertically
        ){
            // Placeholder for Checkbox (we'll do logic next)
            Checkbox(checked = false, onCheckedChange = {})
            Spacer(modifier = Modifier.width(12.dp))
            // Text is stacked vertically (Title on top of Due Date)
            Column {
                Text(
                    text = task.label,
                    fontSize = 16.sp,
                    fontWeight = FontWeight.Bold
                )
                Text(
                    text = "Due: ${task.dueTime}",
                    fontSize = 12.sp,
                    color = Color.Gray
                )
            }
        }
    }
}
```

3. **Update the List:** Go back up to `TaskScreen` and swap the boring `Text` for your new `TaskRow`.

```

items(tasks) { task ->
    TaskRow(task = task) // Pass the data down
}
/*if you are on an old version of Android, the code must change to:
items(tasks.count()) { index ->
    // For now, just print the text to prove it works.
    // We will replace this with a fancy row later.
    TaskRow(task = tasks[index])
}
*/

```

*Run the app. It looks professional now, but the checkbox doesn't work yet.*

---

## Phase 5: State (The Brains)

**Concept: mutableStateOf** A normal variable in Kotlin (`var x = false`) is "blind" to the UI. If you change it, the UI doesn't know. We need a special wrapper: `mutableStateOf(false)`. This creates a variable that acts like a **Radar**. When the value changes, it sends a signal to Compose saying: *"Hey! I changed! Redraw the screen!"*

**Concept: remember** Composable functions are goldfish. Every time they redraw (recompose), they forget everything and reset.

- *Without remember:* The function runs -> sets `isChecked = false`. User clicks -> redraws -> sets `isChecked = false` again.
- *With remember:* The function runs -> sets `isChecked = false`. User clicks -> redraws -> `remember` says "Wait! I have a saved value in the bank. Use `true`, not `false`."

1. Update `TaskRow` to handle state.

@Composable

```

fun TaskRow(task: TaskItem) {
    // 1. DEFINE STATE
    // "remember" = Don't forget this when you redraw.
    // "mutableStateOf" = Watch this variable for changes.
    var isChecked by remember { mutableStateOf(false) }
    Surface(
        modifier = Modifier.fillMaxWidth(),

```

```

shape = RoundedCornerShape(12.dp),
// 2. REACTIVE COLOR
// If checked, turn purple. If not, white.
color = if (isChecked) Color(0xFFE8DEF8) else Color.White,
shadowElevation = 2.dp
){
    Row(
        modifier = Modifier.padding(16.dp),
        verticalAlignment = Alignment.CenterVertically
    ){
        // 3. THE INTERACTION
        Checkbox(
            checked = isChecked, // Read the state
            onCheckedChange = { newValue ->
                // Write the state.
                // This triggers RECOMPOSITION (The function runs again).
                isChecked = newValue
            }
        )

        Spacer(modifier = Modifier.width(12.dp))

        Column {
            Text(
                text = task.label,
                fontSize = 16.sp,
                fontWeight = FontWeight.Bold,
                // 4. REACTIVE TEXT STYLE
                // If checked, draw a line through it.
                textDecoration = if (isChecked) TextDecoration.LineThrough else null
            )

            Text(
                text = "Due: ${task.dueTime}",
                fontSize = 12.sp,
                color = Color.Gray
            )
        }
    }
}
}

```

*Run the app. Click the boxes. Watch the colors and text styles change instantly.*

---

## Phase 6: Student Exercise (Homework)

### The "Shopping Counter"

**Goal:** Managing Integer state instead of Boolean state.

**Scenario:** Instead of just a To-Do list, turn this into a **Shopping List**. Users need to decide *how many* of each item to buy.

#### Tasks:

1. **Change State:** Instead of `isChecked` (Boolean), create `var quantity` by `remember { mutableStateOf(1) }`.
2. **Update UI:** Replace the Checkbox with a customized Row containing:
  - A "Minus" Button (-).
  - A Text showing the ``${quantity}``.
  - A "Plus" Button (+).
3. **Logic:**
  - Clicking `+` increments the quantity.
  - Clicking `-` decrements the quantity (Bonus: Don't let it go below 0).
  - If quantity > 5, turn the text color **Red** (Panic buying!).

**Delivery (zip with code and screenshot):**

<https://www.dropbox.com/request/vuTBYsYGKBB27aNOpvDX>