New Languages, Old Friends

— and Declarative UI (User Interface) Foundations

Kotlin · Dart · Swift (read-only) · UI = f(state)

War Story: The API (Application Programming Interface) That Didn't Exist

The year is 2024. A team ships an Android feature in three days using an AI assistant.

- The AI confidently used `LazyColumn.scrollToItem()` — a method that does not exist
- The code compiled (it was in a library the AI hallucinated); the crash hit production
- Three days of debugging to find a method call no one had looked up

The root cause: the developer trusted unfamiliar-language output without being able to read it critically.

The fix for you: today you learn to *read* the language before you trust the tool.

Syntax You'll Meet Today

Five constructs, each anchored to something you already know.

New syntax	What it means	Anchor
Kotlin `val` / `var`	immutable / mutable binding	Java `final x` / `x`
Kotlin `?.` and `?:`	null-safe call / default if null	Java `if (x != null) x.f()`
Kotlin `data class`	value object with equals/hashCode/copy	Java POJO (Plain Old Java Object) + boilerplate
Kotlin trailing lambda	last parameter can live outside `()`	Java SAM (Single Abstract Method) / lambda
Dart `final` / `const`	runtime immutable / compile-time constant	JS `let` / `const`

These five unlock 80% of the code you will see this semester.

Part 1: Kotlin for Java Developers

Java you know → Kotlin you will write

val / var and Type Inference

Java

```java
// Java: type always explicit
final String name = "UniEats";   // immutable
String mutableName = "UniEats";  // mutable

// Java: verbose generics
final List<FoodPlace> places =
    new ArrayList<FoodPlace>();
```

Kotlin

```kotlin
// val = final (immutable binding)
val name: String = "UniEats"
val name = "UniEats"   // type inferred

// var = mutable binding
var mutableName = "UniEats"

// type inferred from constructor
val places = mutableListOf<FoodPlace>()
```

> **Rule of thumb:** always reach for `val` first; only use `var` when you know mutation is needed.

Null Safety: `?.` and `?:`

Java (null-unsafe)

```java
// NullPointerException waiting to happen
String city = user.getAddress().getCity();

// Safe version — verbose
String city = "Unknown";
if (user != null
    && user.getAddress() != null) {
  city = user.getAddress().getCity();
}
```

Kotlin (null-safe by design)

```kotlin
// ?: won't compile — city is non-null String
// val city: String = user.address.city

// ? marks a type as nullable
val city: String? = user?.address?.city

// ?: is the Elvis operator — default if null
val city: String = user?.address?.city ?: "Unknown"

// !! asserts non-null — throws if null (avoid!)
val city: String = user!!.address!!.city
```

`?.` ≈ Java's `if (x != null) x.f()` · `?:` ≈ Java's ternary `x != null ? x : default`

Swift Aside — Notice It's Nearly the Same

Kotlin

```kotlin
val name: String = "UniEats"
val city: String? = user?.address?.city
val city = user?.address?.city ?: "Unknown"
```

Swift (read-only)

```swift
let name: String = "UniEats"
let city: String? = user?.address?.city
let city = user?.address?.city ?? "Unknown"
```

- `val` → `let`
- `var` → `var` (same!)
- `?:` (Elvis) → `??` (nil coalescing)
- Null safety: identical concept, slightly different spelling

> Swift is the third language introduced this semester. You will read it, not write it. Every time you see it, ask: "what is the Kotlin equivalent?"

Data Classes

Java POJO (Plain Old Java Object) — 40 lines of boilerplate

```java
public class FoodPlace {
    private final String id;
    private final String name;
    private final String cuisine;

    public FoodPlace(String id,
                     String name,
                     String cuisine) {
        this.id = id;
        this.name = name;
        this.cuisine = cuisine;
    }

    public String getId() { return id; }
    public String getName() { return name; }
    // + getCuisine(), equals(), hashCode(),
    // toString(), copy()... ~35 more lines
}
```

Kotlin data class (1 line)

```kotlin
data class FoodPlace(
    val id: String,
    val name: String,
    val cuisine: String,
    val rating: Double = 0.0,   // default value
)

// You get for free:
// equals() / hashCode() based on all fields
// toString()  →  FoodPlace(id=1, name=...)
// copy()      →  place.copy(rating = 4.5)
// destructuring → val (id, name) = place
```

Lambdas and Trailing Lambda Syntax

Java lambda

```
// Java: lambda must stay inside ()
button.setOnClickListener(view -> {
    savePlace(place);
});

list.stream()
    .filter(p -> p.rating > 4.0)
    .map(p -> p.name)
    .collect(toList());
```

Kotlin lambda + trailing syntax

```
// If the last parameter is a lambda,
// move it outside the parentheses
button.setOnClickListener {
    savePlace(place)
}

// Higher-order functions on collections
val topNames = places
    .filter { it.rating > 4.0 }
    .map { it.name }

// 'it' is the implicit single parameter
```

Trailing lambda = last parameter is a function → move it after `)`. This is how ALL Composables work.

Kotlin in 60 Seconds — Full Example

```kotlin
data class FoodPlace(
    val id: String,
    val name: String,
    val cuisine: String,
    val rating: Double = 0.0,
)

fun topPlaces(places: List<FoodPlace>, minRating: Double = 4.0): List<String> {
    return places
        .filter { it.rating >= minRating }  // trailing lambda, 'it' = FoodPlace
        .sortedByDescending { it.rating }
        .map { "${it.name} (${it.rating})" } // string template
}

// Extension function — adds a method to an existing type without subclassing
fun FoodPlace.shortLabel(): String = "$name · $cuisine" // single-expression function

fun main() {
    val place: FoodPlace? = findPlace("42")  // nullable return
    val label = place?.shortLabel() ?: "Not found"  // safe call + Elvis
    println(label)
}
```

Your First Test — Proving a Function Works

A test is just code that calls your code and checks the result. No device, no framework magic.

```
import kotlin.test.Test
import kotlin.test.assertEquals

class FoodPlaceTest {
    @Test
    fun `shortLabel combines name and cuisine`() {
        val place = FoodPlace(id = "1", name = "Sora", cuisine = "Vietnamese")
        assertEquals("Sora · Vietnamese", place.shortLabel())
    }
}
```

- `assertEquals(expected, actual)` — the whole idea: it *fails loudly* when the value is wrong. (Same as JUnit `assertEquals` / Jest `expect()` you've seen on the web.)
- It runs in milliseconds because `shortLabel()` is a **pure function** — same input, same output, no UI, no network.
- **Remember this all semester:** the more logic you push into pure functions like this, the more of your app you can test this fast. We add tests for state (L03), async code (L06), and the full pyramid + UI tests (L14).

Part 2: Dart for JavaScript Developers

JS you know → Dart you will write (for Flutter)

Types, Variables, and final / const

JavaScript / TypeScript

```
// JS: dynamic by default, TS adds types
let name: string = "UniEats";
const id: number = 42;

// No compile-time constants in JS
const PI = 3.14159;  // only runtime-const

// Null: any reference can be null/undefined
let city: string | null = null;
```

Dart

```
// Dart: statically typed, always
String name = "UniEats";
int id = 42;
var name = "UniEats";  // type inferred, like JS

// final = set once at runtime (like JS const)
final String appName = "UniEats";

// const = compile-time constant (true immutable)
const double pi = 3.14159;
const padding = EdgeInsets.all(16);  // reused!

// Null safety (like Kotlin — ? makes nullable)
String? city = null;  // nullable
String city = "Cluj"; // guaranteed non-null
```

Classes and Constructors

JavaScript / TypeScript

```
class FoodPlace {
  constructor(
    public readonly id: string,
    public readonly name: string,
    public readonly cuisine: string,
    public readonly rating: number = 0,
  ) {}

  shortLabel(): string {
    return `${this.name} · ${this.cuisine}`;
  }
}

const place = new FoodPlace("1", "Sushi", "Japanese");
```

Dart

```
class FoodPlace {
  // Named parameters with required keyword
  const FoodPlace({
    required this.id,
    required this.name,
    required this.cuisine,
    this.rating = 0.0,
  });

  final String id;
  final String name;
  final String cuisine;
  final double rating;

  // Arrow function (=> is single expression)
  String shortLabel() => '$name · $cuisine';
}

final place = FoodPlace(id: '1', name: 'Sushi', cuisine: 'Japanese');
```

Async / Await — You Already Know This

JavaScript

```javascript
// async/await over Promises
async function fetchPlaces(city) {
  try {
    const response = await fetch(
      `/api/places?city=${city}`
    );
    const data = await response.json();
    return data.places;
  } catch (error) {
    console.error("Failed:", error);
    return [];
  }
}

// Calling it
const places = await fetchPlaces("Cluj");
```

Dart

```dart
// async/await over Futures — same idea
Future<List<FoodPlace>> fetchPlaces(String city) async {
  try {
    final response = await http.get(
      Uri.parse('/api/places?city=$city'),
    );
    final data = jsonDecode(response.body);
    return (data['places'] as List)
        .map((j) => FoodPlace.fromJson(j))
        .toList();
  } catch (e) {
    debugPrint('Failed: $e');
    return [];
  }
}

// Calling it
final places = await fetchPlaces('Cluj');
```

What Is a Flutter Widget?

Think of a Widget like a **Java object whose only job is to describe how it looks** — it does not draw anything itself.

- In Flutter, **every piece of UI is a Widget**: a button, a text label, a row, even invisible padding.
- A `StatelessWidget` is like a Java class with only `final` fields: it receives data in its constructor and cannot change it.
- You override `build(context)` — Flutter calls it whenever it needs to (re)draw this part of the screen.
- Widgets nest inside each other to form a **widget tree**, the same way XML tags nest in an Android layout file.

```
// The shell of every StatelessWidget — build() returns the UI description.
class MyCard extends StatelessWidget {
  const MyCard({super.key, required this.title});
  final String title;  // immutable: final from the Dart ramp ✓

  @override
  Widget build(BuildContext context) {
    return Text(title); // return a UI description — Flutter draws it
  }
}
```

> Rule of thumb: if a UI element never changes its own data, make it a `StatelessWidget`. (Widgets that *can* change — `StatefulWidget` — come in L03.)

Dart in 60 Seconds — Full Flutter Widget

```dart
import 'package:flutter/material.dart';

class PlaceCard extends StatelessWidget {
  const PlaceCard({           // const constructor = widget can be cached
    super.key,
    required this.place,
    required this.onTap,
  });

  final FoodPlace place;
  final VoidCallback onTap;   // VoidCallback = () => void in TypeScript

  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: onTap,           // passing a function — same as JS onClick={onTap}
      child: Card(
        child: ListTile(
          leading: CircleAvatar(
            backgroundImage: NetworkImage(place.photoUrl),
          ),
          title: Text(place.name),
          subtitle: Text('${place.cuisine} · ★ ${place.rating}'),  // string template
        ),
      ),
    );
  }
}
```

Swift Aside #2 — Same Story as Kotlin

Dart

```dart
// Null safety
String? city = null;
final label = city ?? "Unknown";

// Async
Future<List<FoodPlace>> load() async {
  return await repo.fetch();
}

// Arrow function
String label() => '$name · $cuisine';
```

Swift (read-only)

```swift
// Nil safety — identical concept
var city: String? = nil
let label = city ?? "Unknown"

// Async — identical keyword
func load() async -> [FoodPlace] {
  return await repo.fetch()
}

// Computed property (similar intent)
var label: String { "\(name) · \(cuisine)" }
```

> Pattern: once you know Kotlin or Dart, Swift is a 10-minute read. Same ideas, different brackets.

Part 3: UI (User Interface) = f(state)

The mental model behind all four frameworks

What Is a @Composable?

In Jetpack Compose (Android), a `@Composable` **function** is the building block of every screen.

- Think of it as a **Java method that returns a description of a piece of UI** — not a View object, but a description that Compose turns into real pixels.
- You annotate it with `@Composable` so the Compose compiler knows it can be called repeatedly to (re)draw the screen.
- Composables can call other Composables, nesting them like method calls — no XML layout file needed.

```
// The simplest Composable: a function that describes a text label.
@Composable
fun Greeting(name: String) {     // input: plain Kotlin data (val/var from the ramp ✓)
    Text("Hello, $name!")        // Text() is a built-in Composable
}
// Call it inside another Composable, just like a regular function:
@Composable
fun Screen() {
    Greeting(name = "UniEats")
}
```

> **Anchor:** `@Composable` ≈ a Java method that is **called by the framework** every time it needs to know what your UI looks like. You describe it; Compose draws it.

The Imperative Problem — A Quick Reminder

Imperative (XML (Extensible Markup Language) / findViewById era)

```
// You manage every view manually
val tv = findViewById<TextView>(R.id.name)
tv.text = user.name
tv.visibility = if (user.isLoaded)
    View.VISIBLE else View.GONE
val btn = findViewById<Button>(R.id.save)
btn.isEnabled = user.isValid
```

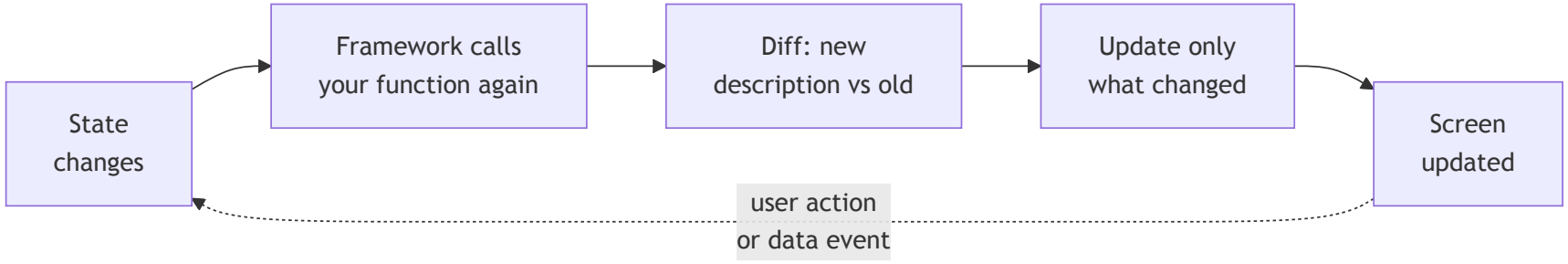
Miss one call → state and UI (User Interface) drift apart → bug.

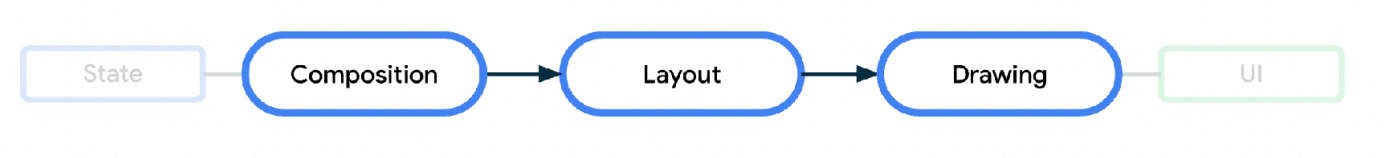
Declarative (Compose / Flutter / SwiftUI / RN)

```
@Composable
fun UserCard(user: User) {
    if (user.isLoaded) {
        Text(user.name)        // Kotlin from ramp ✓
        Button(
            onClick = { save() },
            enabled = user.isValid
        ) { Text("Save") }     // trailing lambda ✓
    }
}
```

UI is *always* consistent with `user`.

`UI = f(state)` — The Render Loop





Compose pipeline: State → Composition → Layout → Drawing → UI (User Interface)

`UI = f(state)` — Across All Four Stacks

Platform	Term for "f"	Term for "call f again"	State trigger
Jetpack Compose	`@Composable` function	**Recomposition**	`State<T>` / `StateFlow`
SwiftUI	`View.body` computed property	**Rebuild**	`@State`, `@Observable`
Flutter	`Widget.build()` method	**Rebuild**	`setState()`, Riverpod
React Native (RN)	Function component	**Re-render**	`useState` setter

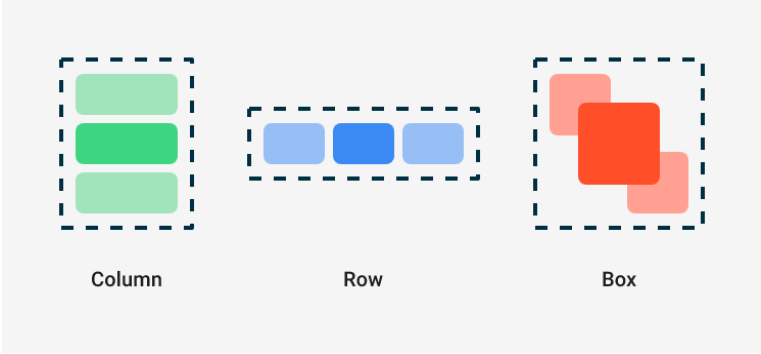
Common principle: you describe *what* the UI looks like; the framework decides *when and how* to update it.

> The "State trigger" column names (`StateFlow` , `@State` , `useState` ,...) are just labels here — we cover how to declare and use state correctly in **L03**.

Part 4: Your First Layouts

Column / Row ↔ Flutter widgets ↔ Flexbox ↔ SwiftUI stacks

Layout Primitives — The Same Idea in Four Names



Direction	Compose	SwiftUI	Flutter	React Native (RN)
Vertical	`Column`	`VStack`	`Column`	`View` (default)
Horizontal	`Row`	`HStack`	`Row`	`View` + `flexDirection:'row'`
Z-axis overlay	`Box`	`ZStack`	`Stack`	`View` + `position:'absolute'`

Vertical + Horizontal Layout — Android and iOS

First time seeing `Modifier`? In Compose a `Modifier` is a chain of instructions you attach to a UI element — padding, size, click handling. Read `Modifier.padding(16.dp)` as "this element, with 16dp of padding around it." Flutter wraps a widget (a `Padding` around its child) to do the same job; SwiftUI calls them view modifiers (`.padding()`). Full slide on this in L03.

Jetpack Compose (Kotlin)

```kotlin
@Composable
fun PlaceCard(place: FoodPlace) {
    Row(                                    // horizontal
        verticalAlignment = Alignment.CenterVertically,
        modifier = Modifier.padding(16.dp)
    ) {
        AsyncImage(
            model = place.photoUrl,
            contentDescription = place.name,
            modifier = Modifier.size(64.dp)
        )
        Column(                             // vertical
            modifier = Modifier.padding(start = 12.dp)
        ) {
            Text(place.name)
            Text(place.cuisine)
        }
    }
}
```

`Row` = horizontal · `Column` = vertical

SwiftUI (Swift — read-only)

```swift
struct PlaceCard: View {
    let place: FoodPlace

    var body: some View {
        HStack(alignment: .center, spacing: 12) { // HStack = Row
            AsyncImage(url: URL(string: place.photoUrl))
                .frame(width: 64, height: 64)
                .clipShape(RoundedRectangle(cornerRadius: 8))
            VStack(alignment: .leading) { // VStack = Column
                Text(place.name)
                    .font(.headline)
                Text(place.cuisine)
                    .font(.caption)
            }
        }
        .padding()
    }
}
```

Translation key: `HStack` = `Row`, `VStack` = `Column`

Vertical + Horizontal Layout — Flutter and React Native

Flutter (Dart)

```dart
class PlaceCard extends StatelessWidget {
  const PlaceCard({super.key, required this.place});
  final FoodPlace place;

  @override
  Widget build(BuildContext context) {
    return Padding(
      padding: const EdgeInsets.all(16),
      child: Row(                          // horizontal
        children: [
          Image.network(place.photoUrl,
              width: 64, height: 64),
          const SizedBox(width: 12),     // spacer
          Column(                          // vertical
            crossAxisAlignment:
                CrossAxisAlignment.start,
            children: [
              Text(place.name),
              Text(place.cuisine),
            ],
          ),
        ],
      ),
    );
  }
}
```

React Native (RN) — TypeScript (TSX (TypeScript XML))

```tsx
// RN layout IS CSS (Cascading Style Sheets) Flexbox —
// you already know this
import { View, Text, Image, StyleSheet } from 'react-native';

export function PlaceCard({ place }: { place: FoodPlace }) {
  return (
    <View style={styles.row}>    {/* flexDirection:'row' = horizontal */}
      <Image
        source={{ uri: place.photoUrl }}
        style={styles.image}
      />
      <View style={styles.col}>  {/* default column = vertical */}
        <Text style={styles.name}>{place.name}</Text>
        <Text style={styles.cuisine}>{place.cuisine}</Text>
      </View>
    </View>
  );
}

const styles = StyleSheet.create({
  row: { flexDirection: 'row', alignItems: 'center', padding: 16 },
  image: { width: 64, height: 64, borderRadius: 8 },
  col: { marginStart: 12 },
  name: { fontSize: 16, fontWeight: '600' },
  cuisine: { fontSize: 12, color: '#888' },
});
```

Part 5: Your First List

LazyColumn (Compose) · List/LazyVStack (SwiftUI) · ListView.builder (Flutter) · FlatList (RN)

Lists — Android (Compose) and iOS (SwiftUI)

Jetpack Compose — `LazyColumn`

```
@Composable
fun PlaceList(
    places: List<FoodPlace>,          // val from ramp ✓
    onPlaceClick: (FoodPlace) -> Unit // lambda type ✓
) {
    LazyColumn(
        contentPadding = PaddingValues(16.dp),
        verticalArrangement = Arrangement.spacedBy(8.dp),
    ) {
        items(
            items = places,
            key = { it.id },   // stable id = correct animations
        ) { place ->           // trailing lambda ✓
            PlaceCard(
                place = place,
                onClick = { onPlaceClick(place) }
            )
        }
    }
}
```

SwiftUI — `List` / `LazyVStack` (read-only)

```
struct PlaceList: View {
    let places: [FoodPlace]
    var onPlaceClick: (FoodPlace) -> Void

    var body: some View {
        // List is the virtualized default in SwiftUI
        List(places, id: \.id) { place in
            PlaceCard(place: place)
                .onTapGesture { onPlaceClick(place) }
        }
        // Or, for custom styling:
        // LazyVStack { ForEach(places, id: \.id) { ... } }
    }
}
```

- `List` = virtualized by default
- `LazyVStack` = manual lazy stack (like LazyColumn)
- `id: \.id` = the key equivalent; or make the model `Identifiable`, as UniEats' `Spot` is: `List(sampleSpots) { spot in … }`

Lists — Flutter and React Native

Flutter — `ListView.builder`

```dart
class PlaceList extends StatelessWidget {
  const PlaceList({
    super.key,
    required this.places,
    required this.onPlaceClick,
  });

  final List<FoodPlace> places;
  final ValueChanged<FoodPlace> onPlaceClick;

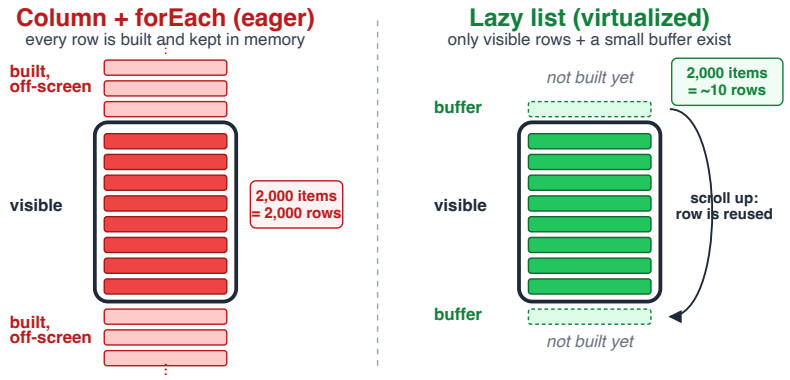
  @override
  Widget build(BuildContext context) {
    return ListView.builder(
      padding: const EdgeInsets.all(16),
      itemCount: places.length,
      itemBuilder: (context, index) {
        final place = places[index];  // val ✓
        return PlaceCard(
          place: place,
          onTap: () => onPlaceClick(place), // => arrow ✓
        );
      },
    );
  }
}
```

React Native (RN) — `FlatList`

```tsx
import { FlatList } from 'react-native';

export function PlaceList({
  places,
  onPlaceClick,
}: {
  places: FoodPlace[];
  onPlaceClick: (place: FoodPlace) => void;
}) {
  return (
    <FlatList
      data={places}
      keyExtractor={(item) => item.id}  // key equivalent
      contentContainerStyle={{ padding: 16 }}
      ItemSeparatorComponent={() => (
        <View style={{ height: 8 }} />
      )}
      renderItem={({ item: place }) => (
        <PlaceCard
          place={place}
          onPress={() => onPlaceClick(place)}
        />
      )}
    />
  );
}
```

List Virtualization — Why It Matters



Scenario	`Column + forEach`	Lazy list
20 items	Fine	Fine
200 items	Slow startup	Fast
2,000 items	OOM (Out Of Memory) crash	Fast
Infinite scroll	Not possible	Fine

Never use for unbounded data:

- Compose: `Column { places.forEach { PlaceCard(it) } }`
- Flutter: `Column(children: places.map((p) => PlaceCard(p)).toList())`
- RN: `ScrollView` + `.map()`

Always use: `LazyColumn` · `List` / `LazyVStack` · `ListView.builder` · `FlatList`

Part 6: Legacy — The World You'll Meet at Work

Legacy Android: XML (Extensible Markup Language) + RecyclerView

```xml
<!-- res/layout/item_place.xml — you'll see this in real codebases -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="wrap_content"
    android:orientation="horizontal" android:padding="16dp">
    <ImageView android:id="@+id/imgPlace"
        android:layout_width="64dp" android:layout_height="64dp" />
    <TextView android:id="@+id/txtName"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_marginStart="12dp" android:textSize="16sp" />
</LinearLayout>
```

```
class PlaceViewHolder(view: View) : RecyclerView.ViewHolder(view) {
    private val img   = view.findViewById<ImageView>(R.id.imgPlace)
    private val name  = view.findViewById<TextView>(R.id.txtName)

    fun bind(place: FoodPlace) {
        name.text = place.name          // manual state sync — the problem we solved
        Glide.with(img).load(place.photoUrl).into(img)
    }
}
```

Rule: When you touch a screen in an existing XML codebase, migrate it to Compose. Do not write new XML (Extensible Markup Language).

Spot the AI Bug 🐞

Can you find the flaw before the next slide?

Spot the AI Bug 🐞 — Find the Flaw

An AI generated this Compose screen to show a list of ~500 campus food spots:

```
@Composable
fun SpotListScreen(spots: List<FoodSpot>) {
    Column {                          // ← look closely here
        spots.forEach { spot ->
            SpotCard(spot = spot)
        }
    }
}
```

Hint: Think back to what we just saw about how `Column` and `LazyColumn` differ.

Spot the AI Bug 🐞 — The Answer

```
@Composable
fun SpotListScreen(spots: List<FoodSpot>) {
    // ❌ BUG: Column builds ALL 500 cards immediately — even the ones off screen.
    //         On a long list this is slow, janky, and can run out of memory.
    // Column { spots.forEach { SpotCard(it) } }

    // ✅ FIX: LazyColumn builds only the items currently on screen (virtualization).
    //         items(...) is how you feed a list into it.
    LazyColumn {
        items(spots, key = { it.id }) { spot ->   // key = a stable identity per row
            SpotCard(spot = spot)
        }
    }
}
```

- `Column` is **eager** — every child is built now. `LazyColumn` is **lazy** — only visible items.
- The `key` keeps each row's identity stable when the list changes; without it, animations and per-row state can attach to the wrong row.

Interview Corner 🎤

Real questions from mobile engineering interviews

Interview Corner 🎤

Q1. What is the difference between `val` and `var` in Kotlin? What is the Java equivalent of each?

Q2. In Kotlin, when would you use `?.` and when would you use `!!`? What is the risk of `!!`?

Q3. Explain `UI = f(state)` in one sentence. What does a declarative framework do when the state changes?

Q4. What is the difference between a `Column` and a `LazyColumn` in Compose, and when does it matter?

Q5. What is the difference between `final` and `const` in Dart? Why does Flutter use `const` constructors so often?

Q6. React Native and Flutter both let you build the UI (User Interface) without writing Kotlin or Swift. What is the key difference in *how* each one actually puts pixels on the screen?

🖥️ Demo — UniEats Static List + Detail (all four stacks)

> The audience sees the 8 campus spots in a scrollable list and taps into a detail screen: hard-coded data, no network, no server. The focus is the declarative list pattern and what the `key` buys you.

Repo: `github.com/dancojocar/ma` @ tag `l02-ui` (`git switch --detach l02-ui`), files under `unieats/`

Stack	Do this, see this	File
Android	Tap **Sort** in the top bar: rows slide to new places. Delete `key = { it.id }` , re-run, tap Sort: rows just flash	`android/app/src/main/java/com/unieats/app/ui/spotlist/SpotListScreen.kt`
Flutter	`ListView.builder` of `SpotCard` s (`const` constructor, `key: ValueKey(spot.id)` , `SpotPhoto` = `Image.network` + placeholder); edit the `AppBar` title, save: hot reload	`flutter/lib/ui/spot_list/spot_list_screen.dart`
iOS (read)	`List(sampleSpots) { … }` , `Spot: Identifiable` is the key; `@State selectedSpot` opens the detail sheet	`ios/UniEats/Views/SpotListView.swift`
React Native	`FlatList` + `keyExtractor` ; open a spot, change `backText` in `StyleSheet.create` , save: Fast Refresh, detail stays open	`rn/app/(tabs)/spots/index.tsx`

What this lecture adds: `git diff l01-hello l02-ui -- unieats/` — the static list + detail UI in all four stacks (`l01-hello` is just the hello screen).

Two Things to Watch For Next Lecture

Modifier order matters (preview of L03):

```
// clip BEFORE clickable → ripple stays inside the card boundary
Modifier
    .clip(RoundedCornerShape(12.dp))
    .clickable { onPlaceClick(place) }
    .padding(16.dp)
```

Adaptive layout — one codebase, every screen size (preview of L04):

```
// WindowSizeClass drives which layout is shown
@Composable
fun AdaptivePane(showOnePane: Boolean) {
    if (showOnePane) { OnePane() } else { TwoPane() }
}
```

> Same pattern in SwiftUI's `NavigationSplitView`, Flutter's `AdaptiveScaffold`, and RN's `useWindowDimensions`.

Recap — If You Only Remember 3 Things

1. Kotlin = Java, fixed

- `val` / `var` · `?.` / `?:` · data class · trailing lambda
- Every new construct has a Java equivalent
- `!!` is a smell — it can throw

2. Dart = JS with types

- `final` / `const` · nullable `?` · `async` / `await` you know
- `const` constructors = Flutter performance
- `=>` is the same arrow function as JS

3. UI (User Interface) = f(state) — everywhere

- Describe what the UI *looks like*, not what to *do*
- State changes → framework re-runs your function → diffs → updates
- Use lazy lists for any unbounded data:
  - `LazyColumn` / `List` / `ListView.builder` / `FlatList`
- A `@Composable` / `Widget` / component is just a **function that returns a UI description** — the framework calls it; you don't

What's Next: L03 — State & Architecture Foundations

- `remember` / `mutableStateOf` — how Compose state is declared (today's **Sort** toggle uses it)
- `ViewModel` + `StateFlow` — why state must survive recomposition and rotation
- State hoisting — lifting state up to make composables reusable
- `@State` / `@Observable` in SwiftUI · `setState()` / Riverpod in Flutter · `useState` in React Native (RN)
- Why AI-generated state code often breaks: stale closures, recomposition loops

Lab this week (Lab 2): your project's first screens. Start the **UI only — Native** assignment (due 8 Nov): a list screen and separate create/edit screens for your own entity, with in-memory data — exactly today's `LazyColumn` + card pattern. Let Gradle sync once at home, not on the lab wifi.

Questions?