Jetpack Compose : l'interface déclarative

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-11-01

Table des matières

1 Objectifs 2
2 1. Pourquoi une interface déclarative 3
2.1 1.1 Le modèle hérité : décrire une scène, puis la muter à la main 3
2.2 1.2 Le modèle Compose : décrire une fonction, laisser le framework réconcilier 4
3 2. Le modèle d'exécution : composition, recomposition, mise en page, dessin . 5
3.1 2.1 Les quatre phases 5
3.2 2.2 `@Composable` : une convention de contexte, pas un changement de sémantique 6
3.3 2.3 Pureté : la règle d'or d'une fonction de composition 6
3.4 2.4 `@Preview` : itérer sans appareil 7
4 3. Écrire des composables : conventions et briques de base 7
4.1 3.1 Les conventions officielles 7
4.2 3.2 Les briques de base 8
4.3 3.3 `MaterialTheme` et `Scaffold` : la charpente d'un écran 9
5 4. État et recomposition : `mutableStateOf` et `remember` 10
5.1 4.1 Une variable Kotlin ordinaire ne déclenche rien 10
5.2 4.2 Rendre un état observable : `mutableStateOf` 10
5.3 4.3 États composés et collections 11
6 5. La remontée d'état et le flux unidirectionnel 12
6.1 5.1 Définition 12
6.2 5.2 La bonne hauteur de remontée 13
7 6. `Modifier` : la chaîne importe — appui du TP 14
7.1 6.1 Un modificateur est une description d'habillage immuable 14
8 7. Mise en page : `Column`, `Row`, `Box` — appui du TP 15
8.1 7.1 Les deux axes linéaires 15
8.2 7.2 `weight` : le partage d'espace proportionnel 16
8.3 7.3 `Box` : la superposition 16

9 8. Listes et défilement : de `verticalScroll` à `LazyColumn` — appui du TP 17
9.1 8.1 Le problème : la paresse n'est pas le défaut 17
9.2 8.2 `LazyColumn` : composer ce qui est visible 17
10 9. Material Design 3 : le thème comme paramètre d'architecture — appui du TP 18
10.1 9.1 Un thème, trois familles de valeurs 18
10.2 9.2 Couleur dynamique : mention, pas mode d'emploi 19
11 10. Limites de cette leçon et suite du cours 20
11.1 10.1 Ce qui est explicitement différé 20
11.2 10.2 La limite structurelle, celle qui nous occupe tout de suite 20
11.3 10.3 Renvoi au travail pratique 21
12 Bibliographie optionnelle 21

1 Objectifs

À l'issue de cette leçon, vous serez capable de :

1. expliquer pourquoi Jetpack Compose remplace la manipulation impérative des vues par une description déclarative de l'interface, et situer ce changement par rapport à la leçon 06 (Activities, XML, `findViewById`) ;
2. décrire le modèle d'exécution de Compose : composition, recomposition, mise en page (layout) et dessin (draw), et énoncer les règles de pureté d'une fonction annotée `@Composable` ;
3. écrire des composables respectant les conventions officielles, les organiser avec `MaterialTheme` et `Scaffold`, et les composer avec les briques de base (`Text`, `Button`, `Icon`, champs de saisie) ;
4. manipuler l'état observable (`mutableStateOf`) et sa conservation entre recompositions (`remember`), en rattachant la syntaxe `by` aux délégués de propriété vus en leçon 02 ;
5. appliquer la remontée d'état (state hoisting) et le flux de données unidirectionnel — état qui descend, événements qui montent — et identifier le `ViewModel` comme cible de remontée d'un écran entier ;
6. construire des interfaces avec `Modifier` (en maîtrisant le piège de l'ordre des appels), `Column`, `Row`, `Box` et la pondération `weight` ;
7. afficher des listes performantes avec `LazyColumn` et comprendre le rôle des clés stables ;
8. personnaliser un thème Material Design 3 (jeu de couleurs, typographie, formes, clair/sombre) ;
9. identifier ce que cette leçon **ne couvre pas** (navigation, effets, animations, performance fine) et pourquoi la persistance de l'état devient le chapitre suivant.

Organisation : les sections 6 à 9 sont des sections d'appui du TP — gardez-les ouvertes pendant la pratique et consultez-les en aller-retour avec les exercices E3 et E5.

Cette leçon est la contrepartie théorique du travail pratique `07_mobiledev_101_P_JetPackCompose` (« Découvrir Jetpack Compose ») : les objectifs que vous y poursuivez — comprendre les concepts fondamentaux, créer une interface moderne, manipuler l'état et les événements, utiliser les composants Material Design — sont traités systématiquement ci-dessous. Les exemples d'application suivent le fil du TP : un outil de suivi d'entraînement (catalogue d'exercices, session active, historique, statistiques).

Prérequis : leçon 02 (Kotlin : classes, propriétés et délégués, null-safety, lambdas) et leçon 06 (composants Android principaux : Activity, cycle de vie, ViewModel).

---

2 1. Pourquoi une interface déclarative

2.1 1.1 Le modèle hérité : décrire une scène, puis la muter à la main

En leçon 06, une interface Android se construit en deux temps. D'abord une **description statique** : un fichier XML instancie une hiérarchie de vues (`TextView`, `Button`, `LinearLayout`…) au moment du `setContentView`. Ensuite une **synchronisation manuelle** à chaque changement de données : on retrouve la vue par son identifiant et on réapplique soi-même la mise à jour.

```
// Approche impérative (leçon 06) : la vue est un objet mutable qu'on corrige à la main.
class CounterActivity : AppCompatActivity() {
    private var count = 0
    private lateinit var label: TextView

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_counter)
        label = findViewById(R.id.countLabel)
        label.text = count.toString()          // affichage initial
        findViewById<Button>(R.id.increment).setOnClickListener {
            count += 1
            label.text = count.toString()      // ET SI J'OUBLIE CETTE LIGNE ?
        }
    }
}
```

Rien n'oblige le compilateur à ce que chaque chemin qui modifie `count` pense à réécrire `label.text`. L'invariant « l'écran reflète les données » est une **convention humaine**, qui se viole silencieusement et sans message d'erreur : c'est l'une des sources historiques de bogues d'interface les plus fréquents sur Android.

2.2 1.2 Le modèle Compose : décrire une fonction, laisser le framework réconcilier

Jetpack Compose inverse la charge de la preuve. On n'écrit plus comment transformer l'interface d'un état à l'autre, mais à quoi elle ressemble **pour un état donné** :

```
// Approche déclarative : le « quoi », pas le « comment mettre à jour ».
@Composable
fun CounterLabel(count: Int) {
    Text(text = count.toString())
}
```

`CounterLabel` n'est pas une fonction qui fabrique un `TextView` : c'est une **description de l'interface pour une valeur de `count`**. Quand `count` change, Compose réexécute la description et ajuste lui-même ce qui doit l'être. Le développeur n'a plus d'invalidation à mémoriser : elle est structurellement impossible à oublier, puisqu'il n'y a plus d'invalidation manuelle du tout.

```
flowchart TB
    subgraph IMP["Impératif (Views + XML)"]
           D1[Donnée modifiée] ⟶ D2["Le développeur doit appeler setText / setVisible / ..."]
        D2 ⟶ D3[Vue à l'écran]
        D2 -. oubli possible .→ D4["Écran incohérent<br/>(bogue classique)"]
    end
    subgraph DEC["Déclaratif (Compose)"]
         E1[Donnée modifiée] ⟶ E2["Recomposition automatique<br/>des parties qui lisent cette donnée"]
        E2 ⟶ E3[Interface à l'écran]
    end
```

Pour un développeur venu de Java et d'Android classique (le profil visé par ce cours), trois ruptures méritent d'être nommées dès maintenant :

- **plus de XML** : la mise en page s'écrit en Kotlin, ce qui supprime la frontière fragile entre le fichier de description et le code de liaison ;
- **plus de `findViewById`** : il n'y a pas d'objet-vue à retrouver, seulement des appels de fonctions ;

- **plus de cycle de vie des vues** : il y a un cycle de vie de la description, piloté par l'état — et c'est précisément l'objet de la section suivante.

Compose n'est pas une abstraction « au-dessus » des vues : c'est un moteur de rendu propre, une bibliothèque Kotlin (multiplateforme sur JVM), sans dépendance au toolkit `android.view`. Une application peut mélanger les deux mondes, mais ce cours traite Compose comme l'outil principal d'interface, conformément au fil du TP.

---

3 2. Le modèle d'exécution : composition, recomposition, mise en page, dessin

3.1 2.1 Les quatre phases

Comprendre Compose, c'est comprendre le pipeline que parcourt chaque frame potentielle :

```
flowchart LR
    A["Exécution des @Composable<br/>(construction de l'arbre)"] ⟶ B["Mise en page<br/>(tailles et positions)"]
    B ⟶ C["Dessin<br/>(rendu à l'écran)"]
    S[(État observable)] -.→|lit| A
    S -.→|"mutation"| R[Recomposition]
    R ⟶ A
```

1. **La composition.** Compose exécute vos fonctions annotées `@Composable` et produit une description arborescente de l'interface (une structure de données légère, pas des objets-vues). Cette phase se déclenche à l'affichage initial et à chaque **recomposition**.
2. **La mise en page** (layout). Le système attribue à chaque nœud une taille et une position — c'est là que travaillent les conteneurs `Column`, `Row`, `Box` et le `Modifier` (sections 6 et 7).
3. **Le dessin** (draw). Les nœuds sont peints à l'écran.
4. **La recomposition.** Lorsqu'un **état observable** lu pendant la composition change, Compose réexécute **uniquement les fonctions qui ont lu cet état**, puis fait suivre mise en page et dessin pour les parties concernées. C'est le mécanisme qui remplace l'invalidation manuelle de la section 1.

Deux conséquences pratiques, à retenir avant même d'écrire du code :

- Une fonction `@Composable` **peut être exécutée très souvent** — à chaque interaction qui touche l'état qu'elle lit. Elle doit donc rester rapide et sans effet de bord (section 2.3).

- Compose est capable de **sauter** la recomposition d'un composable dont personne ne lit l'état muté et dont les paramètres sont inchangés. Cette optimisation n'est pas un sujet de cette leçon (voir §10), mais elle explique la discipline des paramètres : passez des données aux composables, ne les faites pas lire « de loin ».

3.2 2.2 `@Composable` : une convention de contexte, pas un changement de sémantique

L'annotation `@Composable` indique au compilateur que la fonction ne peut être appelée **que depuis un contexte de composition** — une autre fonction `@Composable`, ou un point d'entrée comme `setContent`. Elle n'introduit ni thread, ni magie d'exécution : les règles Kotlin habituelles (visibilité, types, null-safety) s'appliquent inchangées.

```
@Composable
fun WorkoutGreeting(name: String) {
    Text(
        text = "Bonjour $name, prêts pour la séance ?",
        style = MaterialTheme.typography.titleLarge
    )
}

// Interdit à la compilation :
// WorkoutGreeting("Alice")   // hors contexte de composition — erreur attendue
```

Le nom de la convention vient de la théorie des composants fonctionnels : un composable est une **unité de description réutilisable**, l'équivalent conceptuel d'un composant, mais réalisé par un simple appel de fonction — pas par une classe, pas par un fichier.

3.3 2.3 Pureté : la règle d'or d'une fonction de composition

Puisqu'une fonction `@Composable` peut être réexécutée à tout moment et autant de fois que nécessaire, elle doit être **sans effet de bord observable** : ne pas écrire dans un fichier, ne pas lancer de requête réseau, ne pas incrémenter une variable extérieure, ne pas dépendre d'une horloge globale. Son résultat visible ne doit dépendre que de deux choses : ses **paramètres** et les **valeurs d'état** qu'elle lit.

```
// MAUVAIS exemple — deux violations typiques.
private var renderCount = 0                                  // effet de bord sur
variable globale

@Composable
fun BadBanner() {
    renderCount += 1                              // exécuté à CHAQUE recomposition
```

```
    Text(text = "Affichage n° $renderCount",
         style = MaterialTheme.typography.bodyMedium)
}
```

Ici `renderCount` augmente au gré des recompositions, dont le nombre n'est **pas** une grandeur contrôlée par le développeur : le comportement devient indéterministe. Les besoins légitimes d'effets (démarrer une animation, consigner un événement, planifier une action) existent, mais relèvent d'API dédiées (`LaunchedEffect` et consorts) que cette leçon explicitement **n'aborde pas** — voyez §10.

3.4 2.4 `@Preview` : itérer sans appareil

Pendant le développement, l'annotation `@Preview` demande à Android Studio de rendre la fonction dans l'éditeur, sans émulateur ni téléphone :

```
@Preview(showBackground = true)
@Composable
private fun WorkoutGreetingPreview() {
    MaterialTheme {                // enveloppe : fournit couleurs et typographie
        WorkoutGreeting("Alice")
    }
}
```

`@Preview` accepte plusieurs annotations sur la même fonction (clair/sombre, langues, tailles de police) ; l'exploration fine relève des environnements de développement, pas de ce cours. Retenez-en l'essentiel : **un composable se lit comme il s'exécute**, et cette propriété est exactement ce que le mode déclaratif met au premier plan.

---

4 3. Écrire des composables : conventions et briques de base

4.1 3.1 Les conventions officielles

- **PascalCase** pour les fonctions `@Composable` (`WorkoutGreeting`, `ExerciseCard`) : la typographie même du code signale « ceci décrit de l'interface ».
- **Aucune valeur de retour** : un composable dessine, il ne calcule pas.
- **Dernier paramètre lambda** (idiom trailing lambda) : le contenu « imbriqué » passe en bloc `{ ... }`, ce qui donne aux conteneurs une allure de structure de contrôle.
- **Sans état par défaut** : un composable qui reçoit tout par paramètres et ne retient rien est testable et réutilisable — c'est la matière de la section 5.
- **Un fichier par écran ou famille logique** : pas de hiérarchie de classes, de simples fonctions organisées comme du Kotlin ordinaire.

4.2 3.2 Les briques de base

```
@Composable
fun SessionHeader(title: String, isActive: Boolean) {
    Column {
        Text(
            text = title,
            style = MaterialTheme.typography.headlineSmall
        )
        Row(verticalAlignment = Alignment.CenterVertically) {
            Icon(
                imageVector = if (isActive) Icons.Default.PlayArrow
                              else Icons.Default.Pause,
               contentDescription = if (isActive) "Séance en cours" else "Séance
en pause"
            )
            Spacer(modifier = Modifier.width(8.dp))
            Text(
                text = if (isActive) "En cours" else "En pause",
                style = MaterialTheme.typography.bodyMedium
            )
        }
    }
}
```

Quatre enseignements dans ce fragment :

1. `Text` et `Icon` sont de simples fonctions : le style vient du **thème** (`MaterialTheme.typography`), pas de copies collées de valeurs littérales.
2. `contentDescription` est un paramètre d'accessibilité **obligatoire dans la conception**, pas une option décorative — `null` n'est licite que pour une icône purement ornementale.
3. Les conteneurs (`Column`, `Row`) sont présentés en section 7 ; ici, seule leur existence est nécessaire.
4. La logique conditionnelle s'exprime en **Kotlin ordinaire** (`if`/`else`). Il n'y a pas de directives de template : c'est le langage qui fait le tri, ce qui rend le code d'interface inspectable comme n'importe quel code.

Les boutons et la saisie suivent la même logique : un `Button` reçoit un lambda `onClick` — retour sur les fonctions d'ordre supérieur de la leçon 02 — et son contenu s'écrit en bloc trailing lambda.

```
@Composable
fun StartButton(onStart: () → Unit) {
```

```
    Button(onClick = onStart) {
        Text("Démarrer la séance")        // le bloc {} est le contenu du bouton
    }
}
```

Notez la signature : `StartButton` ne sait **pas** ce que « démarrer » signifie. Il reçoit le comportement en paramètre. Nous généraliserons ce principe en section 5.

4.3 3.3 `MaterialTheme` et `Scaffold` : la charpente d'un écran

```
@Composable
fun FitTrackApp() {
     MaterialTheme {                          // fournit couleurs, typographie,
formes (§9)
        Scaffold(
            topBar = {
                TopAppBar(title = { Text("FitTrack") })
            },
            floatingActionButton = {
                 FloatingActionButton(onClick = { /* ouvrir un ajout d'exercice
*/ }) {
                    Icon(Icons.Default.Add, contentDescription = "Ajouter")
                }
            }
          ) { innerPadding →                  // lambda de contenu, paramètre =
marges réservées
            Box(modifier = Modifier.padding(innerPadding)) {
                ExerciseCatalog(exercises = exerciseSamples)
            }
        }
    }
}
```

`Scaffold` implante la **grille Material** : emplacements réservés (`topBar`, barre inférieure, FAB, snackbars) et marges calculées pour que le contenu ne passe pas sous ces barres — le paramètre `innerPadding` est là pour cela. `MaterialTheme` est le point d'accès aux valeurs du thème ; nous le personnalisons en section 9.

---

5 4. État et recomposition : `mutableStateOf` et `remember`

5.1 4.1 Une variable Kotlin ordinaire ne déclenche rien

Le moteur de recomposition ne surveille que l'**état observable**, c'est-à-dire des conteneurs spéciaux dont Compose connaît les lectures et les écritures. La version naive

```
@Composable
fun BrokenCounter() {
    var count = 0                    // variable locale, effet de vie = un appel
de fonction
    Text("Compteur : $count")
    Button(onClick = { count += 1 }) { Text("+1") }
}
```

ne fonctionne pas, pour une raison que la section 2 rend limpide : chaque recomposition **réexécute la fonction depuis sa première ligne**. `count` est recréée à `0` à chaque appel ; même si Compose daignait se réveiller, la valeur serait perdue. Double cause d'échec : (a) rien n'est observable, donc aucune recomposition n'est demandée ; (b) rien n'est conservé, donc la valeur ne survivrait pas.

5.2 4.2 Rendre un état observable : `mutableStateOf`

```
@Composable
fun Counter() {
    var count by remember { mutableStateOf(0) }
     // Variante idiomatique pour un Int : mutableIntStateOf(0) — comportement
identique
    Column {
         Text("Séries terminées : $count")           // LECTURE ⇒ inscription
au suivi
          Button(onClick = { count += 1 }) {          // ÉCRITURE ⇒ demande
de recomposition
            Text("+1 série")
        }
    }
}
```

Décomposons l'idiom, car chaque pièce a un rôle précis :

- `mutableStateOf(0)` crée un conteneur d'état observable. Quand `Counter` **lit** `.value`, Compose note la dépendance ; quand `onClick` **écrit**, Compose recompose les lecteurs. C'est le contrat de la section 2.1.

- `remember { ... }` **mémoïse** le résultat du bloc tant que le composable reste dans la même position de l'arbre : sans lui, `mutableStateOf(0)` recréerait un conteneur neuf à chaque recomposition — même problème que `var count`, seulement plus cher.
- `by` est un **délégué de propriété** (leçon 02) : `count` se lit et s'écrit comme un `Int`, mais l'accès transite par le conteneur observable. Écrire `val count = remember { mutableStateOf(0) }` puis `count.value` est strictement équivalent — l'idiom `by` est simplement plus discret.
- Pour un `Int`, `mutableIntStateOf(0)` est la **spécialisation idiomatique** de `mutableStateOf(0)` : comportement identique (lecture suivie, écriture déclenchant la recomposition) — vous le rencontrerez dans le quiz 2 du TP 07.

Deux remarques de typage héritées de la leçon 02 : l'état est **parfaitement compatible avec le système de null-safety** — `mutableStateOf<Exercise?>(null)` décrit un exercice « non sélectionné » sans recourir à un flag booléen annexe ; et comme tout `var`, un état ne doit muter que depuis le **thread de composition**.

5.3 4.3 États composés et collections

Pour un objet entier, on rend observable soit l'objet (recréé à chaque écriture — cf. §6 sur l'immutabilité), soit ses champs :

```
data class SessionState(val setsDone: Int = 0, val target: Int = 5)

@Composable
fun SessionProgress(state: SessionState) {
    Text("${state.setsDone} / ${state.target} séries")
}

@Composable
fun SetLogger() {
    val log = remember { mutableStateListOf<String>() }     // liste observable
    Column {
        Button(onClick = { log += "Série ${log.size + 1} bouclée" }) {
            Text("Marquer une série")
        }
                        log.forEach   {   entry   →   Text(entry,   style   =
MaterialTheme.typography.bodySmall) }
    }
}
```

`remember` sans paramètre conserve la valeur tant que la **position** dans l'arbre est stable. Dès que la liste des éléments eux-mêmes varie (section 8), la notion de position devient fragile — c'est exactement ce que règlent les clés de `items`.

---

6 5. La remontée d'état et le flux unidirectionnel

6.1 5.1 Définition

Nous fixons ici le vocabulaire pour tout le reste du cours :

> **Remontée d'état** (state hoisting) : pratique qui consiste à sortir un état de son composable d'usage pour le confier à un **appelant** — le composable concerné devenant alors **sans état** (stateless) : il reçoit ses valeurs en paramètres et signale ses intentions par des lambdas.

Le bénéfice n'est pas stylistique : un composable sans état est **réutilisable** (il n'impose rien sur la provenance de ses données), **testable** (on injecte les valeurs, on observe les appels) et **prévisible** (une seule source de vérité par donnée).

```
// Sans état : reçoit la valeur, signale l'intention.
@Composable
fun StepCounter(
    count: Int,
    onIncrement: () -> Unit,
    modifier: Modifier = Modifier
) {
    Column(modifier = modifier) {
        Text("Séries : $count")
        Button(onClick = onIncrement) { Text("+1") }
    }
}

// Avec état : le détenteur, minimal.
@Composable
fun CounterScreen() {
    var count by remember { mutableStateOf(0) }
    StepCounter(count = count, onIncrement = { count += 1 })
}
```

Le graphe de données a toujours la même forme :

```
flowchart TD
    H["Détenteur de l'état<br/>CounterScreen (ou, à terme, ViewModel)"]
     H -- "état (descend par les paramètres)" ⟶ A["Composable sans état<br/
>StepCounter"]
    A -- "événement (remonte par lambda)" ⟶ H
    H -- "mutation ⇒ recomposition ciblée" ⟶ A
```

C'est le **flux de données unidirectionnel** (UDF, pour unidirectional data flow) : l'état circule vers le bas, les événements vers le haut, la mutation repart du détenteur. Un écran n'a pas deux sources de vérité ; il en a une, et des vues.

6.2 5.2 La bonne hauteur de remontée

Remonter jusqu'où ? Trois paliers, du plus local au plus structurel :

1. **État local** (`remember` dans le composable) : tout ce qui est purement d'affichage — un champ de texte en cours de frappe, un panneau déroulant ouvert.
2. **État remonté à l'écran** (le composable-racine de l'écran, ou mieux son `ViewModel`) : toute donnée qui doit survivre à la recomposition et à la reconfiguration de l'Activity — la rotation d'écran, la fermeture du clavier.
3. **État de domaine** (couches Repository / données) : ce qui doit survivre au processus lui-même — et c'est le sujet de la leçon 08.

Le palier 2 mérite un rappel de la leçon 06. Le `ViewModel` y était présenté comme le conteneur d'état **survivant à la reconfiguration** (rotation), contrairement à l'Activity. Compose ne change rien à cette architecture : il lui offre simplement un point d'ancrage naturel.

```
class SessionViewModel : ViewModel() {
    var state by mutableStateOf(SessionState())
        private set

    fun addSet() {
        state = state.copy(setsDone = state.setsDone + 1)
    }
}

@Composable
fun SessionScreen(viewModel: SessionViewModel = viewModel()) {
    SessionProgress(state = viewModel.state)          // état descend
    Button(onClick = { viewModel.addSet() }) {        // événement monte
        Text("Terminer une série")
    }
}
```

Dans une application Android réelle, on obtient le `ViewModel` via `by viewModels()` côté Activity ou `viewModel()` (ou `hiltViewModel()`) côté Compose ; le détail d'injection de dépendances n'est pas au programme de cette leçon. Retenez le principe : **le ViewModel est la cible de remontée d'état d'un écran**, et les composables de cet écran sont sans état.

---

7 6. `Modifier` : la chaîne importe — appui du TP

7.1 6.1 Un modificateur est une description d'habillage immuable

Presque tout composable visible accepte un paramètre `modifier: Modifier = Modifier` — `Modifier`, la valeur vide. Chaque appel (`padding`, `background`, `size`, `clickable`…) renvoie un **nouvel objet** décrivant un habillage supplémentaire : rien ne mute, exactement comme les fonctions `copy` des data classes de la leçon 02.

L'ordre des maillons **est sémantique**, parce que chaque maillon habille le résultat du précédent. Comparons deux chaînes composées des mêmes appels :

```
@Composable
fun ModifierOrderDemo() {
    Column {
        // (a) LE BON ORDRE : on réduit d'abord la place, on peint ensuite.
        Text(
            "padding puis background",
            modifier = Modifier
              .padding(16.dp)       // réserve 16 dp AUTOUR de la surface peinte
                .background(Color.LightGray)
                .padding(8.dp)        // puis 8 dp ENTRE la peinture et le texte
        )
        // (b) ORDRE INVERSÉ : le fond est peint TROP tôt, puis rogné.
        Text(
            "background puis padding",
            modifier = Modifier
               .background(Color.LightGray)   // peint tout l'espace disponible
              .padding(16.dp)              // puis le texte se décale : l'espace
       )                                     // réservé EST compris dans le fond
    }
}
```

Le piège classique — `padding` placé **après** `background` alors qu'on voulait une marge extérieure — produit un fond qui « bave » dans les marges intérieures. Le symptôme est visuel, la cause est strictement algébrique : une chaîne de modificateurs est un produit **non commutatif**. La règle mnémonique que nous appliquerons dans tout le TP : **taille et marge extérieures d'abord, peinture ensuite, contenu en dernier**.

Autres constantes de discipline :

- un composable qui se veut habillable **expose** `modifier` en paramètre nommé et l'applique **au premier nœud qu'il émet** — jamais il ne compose son propre habillage en dur ;

- les habillages d'interaction (`clickable`, `selectable`) s'appliquent **après** `background`/`clip`, pour que la zone cliquable coïncide avec la zone peinte.

---

8 7. Mise en page : `Column`, `Row`, `Box` — appui du TP

8.1 7.1 Les deux axes linéaires

Compose ne transpose pas les `LinearLayout`/`ConstraintLayout` XML : la mise en page est elle-même déclarative. Trois conteneurs suffisent à décrire presque tout.

```
@Composable
fun ExerciseCard(exercise: Exercise) {
    Row(
        modifier = Modifier.fillMaxWidth().padding(12.dp),
       verticalAlignment = Alignment.CenterVertically,         // axe transverse
       horizontalArrangement = Arrangement.spacedBy(12.dp)      // axe principal
    ) {
        Icon(Icons.Default.FitnessCenter, contentDescription = null)
          Column(modifier = Modifier.weight(1f)) {                    // prend
l'espace restant
            Text(exercise.name, style = MaterialTheme.typography.titleMedium)
           Text("${exercise.sets} × ${exercise.reps} à ${exercise.weightKg} kg",
                 style = MaterialTheme.typography.bodySmall)
        }
        Text("OK", color = MaterialTheme.colorScheme.primary)
    }
}
```

Vocabulaire à fixer (chaque conteneur expose **arrangement** le long de son axe principal, **alignement** sur son axe transverse) :

Conteneur	Axe principal	Arrangement	Alignement
`Column`	vertical	`verticalArrangement`	`horizontalAlignment`
`Row`	horizontal	`horizontalArrangement`	`verticalAlignment`
`Box`	superposition	—	`contentAlignment` (position des enfants)

`Arrangement.spacedBy(12.dp)`, `Arrangement.SpaceBetween`, `Alignment.Bottom`… sont des valeurs typées — le compilateur refuse les combinaisons d'axe incohérentes, ce que le XML ne savait pas faire.

8.2 7.2 `weight` : le partage d'espace proportionnel

`Modifier.weight(1f)` (dans une `Row`/`Column`) attribue à un enfant une **part de l'espace restant** : tous les enfants pondérés se répartissent le solde proportionnellement à leur poids. C'est l'équivalent conceptuel du `layout_weight` XML, sans fichier de description.

```
@Composable
fun SessionStatsBar(active: Int, done: Int, paused: Int) {
    Row(modifier = Modifier.fillMaxWidth()) {
        StatCell("$active", "En cours",   modifier = Modifier.weight(1f))
        StatCell("$done",   "Terminées",  modifier = Modifier.weight(1f))
        StatCell("$paused", "En pause",   modifier = Modifier.weight(1f))
    }
}

@Composable
private fun StatCell(value: String, label: String, modifier: Modifier = Modifier)
{
                Column(modifier    =    modifier,    horizontalAlignment    =
Alignment.CenterHorizontally) {
        Text(value, style = MaterialTheme.typography.headlineMedium)
        Text(label, style = MaterialTheme.typography.labelMedium)
    }
}
```

8.3 7.3 `Box` : la superposition

`Box` empile ses enfants **du premier appelé au dernier appelé** — le dernier est « au dessus ». C'est le conteneur des badges, des images avec légende en surimpression, des états de chargement par-dessus un contenu.

```
@Composable
fun OverlaidCard() {
    Box(modifier = Modifier.fillMaxWidth().height(120.dp)) {
        Image(painter = /* ressource de la semaine */,
              contentDescription = null,
              modifier = Modifier.fillMaxSize())
        Text("Cette semaine : 5 séances",
             modifier = Modifier.align(Alignment.BottomStart).padding(8.dp),
             color = Color.White)
    }
}
```

Retenir : dans l'ordre d'appel se joue l'ordre de peinture. Un « overlay » qui ne se voile pas est presque toujours un enfant appelé trop tôt.

---

9 8. Listes et défilement : de `verticalScroll` à `LazyColumn` — appui du TP

9.1 8.1 Le problème : la paresse n'est pas le défaut

Une `Column` avec `Modifier.verticalScroll(state)` défile — mais **compose ses enfants d'avance**, tous, y compris la 500e ligne d'un historique de séances. Pour un catalogue d'exercices de dix lignes, c'est acceptable et c'est même la solution la plus simple ; pour des listes longues ou de hauteurs variables, le coût de composition initiale et de mise en page explose.

9.2 8.2 `LazyColumn` : composer ce qui est visible

`LazyColumn` (et son jumeau horizontal `LazyRow`) ne compose et ne met en page que les éléments visibles, plus une petite marge de part et d'autre — l'équivalent fonctionnel du `RecyclerView`, sans adaptateur ni ViewHolder :

```
data class Exercise(
    val id: Long,
    val name: String,
    val sets: Int,
    val reps: Int,
    val weightKg: Double,
)

// Catalogue statique d'exemple — mêmes champs que le modèle du TP 07 (E3)
val exerciseSamples = listOf(
    Exercise(id = 1, name = "Squat", sets = 5, reps = 5, weightKg = 60.0),
     Exercise(id = 2, name = "Développé couché", sets = 4, reps = 8, weightKg
= 60.0),
     Exercise(id = 3, name = "Soulevé de terre", sets = 5, reps = 3, weightKg
= 100.0),
)

@Composable
fun ExerciseCatalog(exercises: List<Exercise>) {
    LazyColumn(
        modifier = Modifier.fillMaxSize(),
        verticalArrangement = Arrangement.spacedBy(8.dp),
        contentPadding = PaddingValues(16.dp),
    ) {
```

```
        item { Text("Catalogue", style = MaterialTheme.typography.titleLarge) }
        items(items = exercises, key = { it.id }) { exercise →
            Card(modifier = Modifier.fillMaxWidth()) {
                ExerciseCard(exercise)
            }
        }
    }
}
```

Points de maîtrise :

- le bloc de `LazyColumn` est un **DSL de contenu** : `item { }` pour un élément ponctuel (un titre), `items(liste) { }` pour un mappage, et les en-têtes groupés existent via `stickyHeader`. On n'y met pas n'importe quel code : seulement des déclarations d'éléments.
- **`key = { it.id }` n'est pas cosmétique.** Les clés stabilisent l'identité d'un élément quand la liste se réordonne : sans clé, un état retenu par `remember` dans une ligne (par exemple un panneau déroulé) reste attaché à la **position**, pas à l'exercice, et « saute » de ligne au réordonnancement. Avec clé, il suit le bon élément. Toute liste dont l'ordre peut varier — un historique de séances trié par date, par exemple — en a besoin.
- le vocabulaire du fil rouge s'installe ici : un **exercice** (nom, séries, répétitions, poids), une **séance** qui enchaîne des exercices, un **historique** qui capitalise des séances. C'est ce que le TP 07 vous fera construire écran par écran.

---

10 9. Material Design 3 : le thème comme paramètre d'architecture — appui du TP

10.1 9.1 Un thème, trois familles de valeurs

Material Design 3 (MD3) structure le style en trois objets accessibles partout via `MaterialTheme` :

- `colorScheme` — le **jeu de couleurs** sémantiques : `primary`, `secondary`, `surface`, `error`, et leurs conteneurs. On consomme des **rôles**, pas des hexadécimaux.
- `typography` — l'**échelle typographique** : de `displayLarge` à `labelSmall`, déjà cohérente en tailles et graisses.
- `shapes` — les **rayons** (`small`, `medium`, `large`) appliqués aux coins.

La discipline qui en découle : dans les sections 3 à 8, chaque `style =` et chaque couleur citaient le thème, jamais une valeur brute. Un composant qui peint `Color(0xFF6750A4)` en dur est un composant hors thème.

```
private val LightColors = lightColorScheme(
    primary = Color(0xFF0061A4),
    secondary = Color(0xFF535F70),
    surface = Color(0xFFFCFCFF),
)

private val DarkColors = darkColorScheme(
    primary = Color(0xFF9ECBFF),
    secondary = Color(0xFFBDC7DC),
    surface = Color(0xFF1A1C1E),
)

@Composable
fun FitnessTheme(content: @Composable () -> Unit) {
    val colors = when (isSystemInDarkTheme()) {       // suit le réglage système
        true -> DarkColors
        false -> LightColors
    }
    MaterialTheme(
        colorScheme = colors,
        typography = FitnessTypography,               // objet Typography maison
        shapes = FitnessShapes,                         // objet Shapes maison
        content = content,
    )
}
```

Un écran s'enveloppe alors de `FitnessTheme { … }`, et **tout** le contenu hérite du jeu. Le point d'entrée de l'application (`setContent` de l'Activity, vu en leçon 06, ou `ComponentActivity` + `setContent` pour une application entièrement Compose) est le lieu naturel de cette enveloppe.

10.2 9.2 Couleur dynamique : mention, pas mode d'emploi

Android 12+ permet de **dériver** le jeu de couleurs du fond d'écran de l'utilisateur (dynamic color, API `DynamicColors`). Une application MD3 l'active typiquement en essayant `dynamicLightColorScheme(context)`/`dynamicDarkColorScheme(context)` avant de retomber sur ses couleurs maison. Ce cours se limite à cette mention : le dynamic color est une couche optionnelle **par-dessus** l'architecture `colorScheme`, et l'essentiel — consommer des rôles et non des constantes — reste vrai avec ou sans.

---

11 10. Limites de cette leçon et suite du cours

11.1 10.1 Ce qui est explicitement différé

Cette leçon couvre le **socle** : modèle d'exécution, écriture de composables, état, remontée d'état, modificateurs, mise en page, listes, thème. Quatre chantiers majeurs de Compose **ne sont pas** au programme ici et ne doivent pas être déduits des exemples :

1. **La navigation** entre écrans (`NavController`, routes, arguments typés) : nos exemples tiennent sur un écran parce que le TP 07 est un premier écran.
2. **Les effets** — la famille `LaunchedEffect`, `DisposableEffect`, `SideEffect` : ils encadrent les besoins d'effets de bord évoqués en §2.3, avec des règles précises de redémarrage et d'annulation.
3. **Les animations** — transitions de valeurs, `AnimatedVisibility`, mouvements partagés entre écrans.
4. **La performance fine** : stabilisation des lambdas, `derivedStateOf`, clés de recomposition, outils de profilage.

Chacun de ces points est un chapitre en puissance ; les ignorer dans un premier écran est sans danger, les ignorer dans une application réelle est une dette.

11.2 10.2 La limite structurelle, celle qui nous occupe tout de suite

Il reste une catégorie d'événements que Compose ne peut pas couvrir, quelle que soit la maîtrise de ses API : **la mort du processus**. Le `remember` survit à la recomposition, le `ViewModel` survit à la reconfiguration — aucun des deux ne survit à l'arrêt du système, à la mise à jour de l'application, ou simplement au téléphone éteint et rallumé.

> L'état Compose — comme celui du `ViewModel` — **ne survit pas à la mort du processus**. La **persistance** est le chapitre 08 : préférences (`SharedPreferences` / DataStore), base locale (Room), fichiers, cloud.

C'est la raison pour laquelle le TP 07 vous fait délibérément construire une interface dont les données sont **fabriquées en mémoire** : catalogue et historique sont des structures vivantes, prêtes à être remplacées par des sources durables au chapitre suivant. Pré-annonçons la forme que prendra cette transition. La leçon 08 et son TP structureront la séance d'entraînement comme une entité persistée :

```
// Forme future (leçon 08, entité Room) — reprise verbatim du modèle posé en E7
du TP 07.
data class Workout(
    val id: Long = 0L,                  // 0 = non persisté (Room générera)
```

```
    val exerciseName: String,           // nom de l'exercice
    val sets: Int,                      // séries
    val reps: Int,                      // répétitions
    val weightKg: Double,               // poids
    val timestamp: Long,                // date (epoch millis)
    val durationMinutes: Int,           // durée
    val caloriesBurned: Int             // calories brûlées
)
```

Tout ce que vous manipulez dans cette leçon — listes d'exercices, compteurs de séries, états de session — se retrouvera **tel quel** dans les champs de cette structure, avec une garantie de plus : la survivance aux redémarrages. Le fil rouge n'est pas interrompu, il change de couche.

11.3 10.3 Renvoi au travail pratique

`07_mobiledev_101_P_JetPackCompose` (« Découvrir Jetpack Compose ») met en œuvre les objectifs de cette leçon : concepts fondamentaux (§1–§2), interface moderne avec les composants Material Design (§3, §9), état et événements (§4–§5), et la structure d'écrans du suivi d'entraînement — catalogue, session active, historique, statistiques (§7–§8). Le stockage y est volontairement absent : ce sera le cœur du TP 08.

---

12 Bibliographie optionnelle

- Documentation officielle Android — section « Jetpack Compose » (`developer.android.com/compose`) : référence normative du modèle d'exécution, de l'API `Modifier` et des composants Material ; lecture conseillée après les §2, §6 et §9.
- Guide « Material Design 3 in Compose » (`https://developer.android.com/develop/ui/compose/designsystems/material3`) : l'intégration des designs Material (couleurs, typographie, formes) dans Compose, pour l'exploration visuelle.

Ces références sont fournies à titre d'approfondissement : tout le contenu évalué dans le TP est développé dans la présente leçon, sans renvoi obligatoire à l'extérieur.