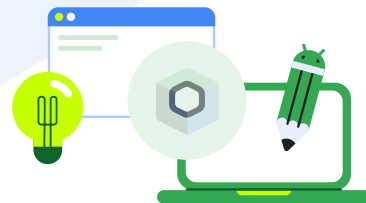


Développement Android avec **Jetpack Compose**





1

Android moderne et Jetpack Compose

Développement Android Jetpack Compose



Thinking in Compose

- **Prérequis** : Vous avez déjà travaillé sur **Android avec Java et XML**. Vous avez également déjà étudié **Kotlin**.
- **L'objectif** de cette séance n'est donc pas de reprendre Android depuis zéro, mais de comprendre le **changement de logique** introduit par **Jetpack Compose**.
- **Compose** ne change pas seulement la syntaxe. Ce n'est pas simplement « *du XML remplacé par du Kotlin* ».
- C'est **une autre manière de penser l'interface** : au lieu de **manipuler des vues** déjà créées, **on décrit l'interface** à afficher à partir des données disponibles.



Thinking in Compose

Pourquoi parle-t-on d'Android moderne ?

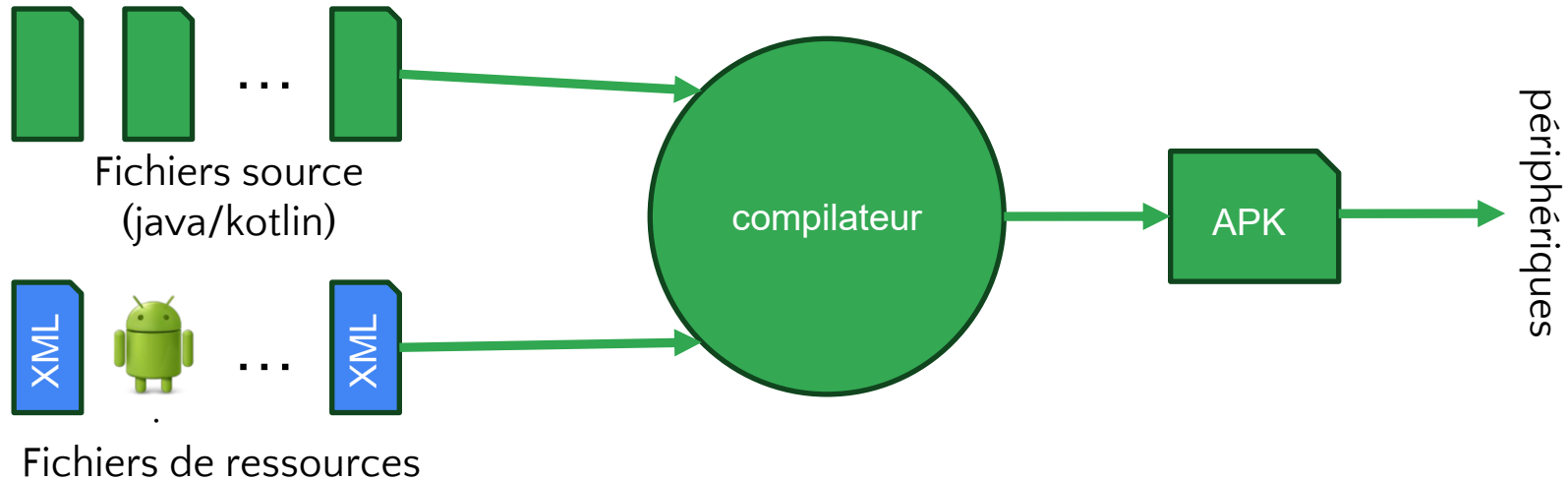
Le développement Android a beaucoup évolué. Pendant longtemps, une **application Android classique** reposait principalement sur :

- des **fichiers XML** pour décrire les **interfaces** ;
- des **classes Java ou Kotlin** pour **gérer la logique** ;
- des **Activity** et parfois des **Fragment** pour organiser **les écrans** ;
- des appels comme **findViewById()** pour récupérer des composants graphiques ;
- des **listeners** pour réagir aux **actions** de l'utilisateur.



Thinking in Compose

● Anatomie d'un Projet classique : Schéma de développement XLM+





Thinking in **Compose**

XML

(Fichier `activity_main.xml`)

```
<Button
    android:id="@+id/btnDay"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Jour"
    app:layout_constraintBottom_toTopOf="@+id/textView"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

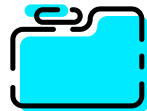


Java

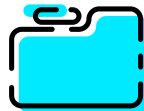
(Fichier `MainActivity.java`)

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    Button btnDay = (Button) findViewById(R.id.btnDay);
    btnDay.setText("Texte modifié depuis Java");
}
```



res



Layout



activity_main.xml



Thinking in Compose

Cela fonctionne, mais plus une **interface devient riche**, plus cette approche peut **devenir lourde** :

- beaucoup de fichiers différents ;
- une **séparation parfois confuse** entre XML et code ;
- des **mises à jour manuelles** de l'interface ;
- des **risques d'incohérence** entre l'état réel de l'application et ce qui est affiché à l'écran...

Jetpack Compose propose une autre manière de penser.



Thinking in Compose

Qu'est-ce que Jetpack Compose ?

Jetpack Compose est le **toolkit moderne** recommandé par Android pour construire des **interfaces natives**. L'UI est écrite en Kotlin avec des **fonctions** particulières appelées **composables**.

```
@Composable
fun WelcomeMessage() {
    Text(text = "Bienvenue dans Shop Explorer")
}
```

@Composable

Indique au compilateur que la fonction participe à la construction de l'interface utilisateur.

- Elle **décrit** ce qui doit être affiché, pas **comment modifier** une vue existante.



Thinking in Compose

Ce que dit l'écosystème Android actuel

- L'écosystème Android actuel est orienté **Compose-first** pour l'UI **moderne**, même si de nombreux projets existants utilisent encore XML.
- Compose s'intègre dans l'écosystème Android : **Android Studio**, **Material3**, **Navigation**, **ViewModel**, **StateFlow**, **Room** et **Retrofit**.

Compose pour l'UI native

Android présente Compose comme son **toolkit** moderne recommandé pour construire des interfaces **natives**.

Moins de code, plus direct

L'interface est décrite en Kotlin, avec des API orientées composants et **Preview** intégrée à Android Studio.

Material 3 + écrans adaptatifs

Compose s'intègre avec Material 3, le **thème** sombre, les **animations** et les interfaces **adaptatives**.



Thinking in Compose

XML vs Compose : même intention, autre modèle

Ancienne approche

```
<TextView  
    android:id="@+id/title"  
    android:text="Shop Explorer" />
```

```
val title = findViewById<TextView>(R.id.title)  
title.text = "Nouveau titre"
```



Nouvelle approche

```
@Composable  
fun ShopTitle() {  
    Text(text = "Shop Explorer")  
}
```

- 🕒 **Le vrai changement** : l'interface devient une **description** en fonction des données, pas une **suite d'instructions** qui modifient des **vues**.



Thinking in **Compose**

UI impérative

"comment modifier l'écran ?"

```
<TextView
    android:id="@+id/title"
    android:text="Produit ajouté" />

<Button
    android:id="@+id/button"
    android:enabled="false" />

<ProgressBar
    android:id="@+id/progressBar"
    android:visibility="gone" />
```

change ce texte → `title.text = "Produit ajouté"`
désactive ce bouton → `button.isEnabled = false`
cache cette vue → `progressBar.visibility = View.GONE`

Approche directe, mais vite fragile quand plusieurs états doivent être synchronisés.



UI déclarative

"que doit afficher l'écran ?"

```
@Composable
fun AddToCartButton(isAdded: Boolean) {
    Button(onClick = { }) {
        Text(
            text = if (isAdded) "Produit ajouté"
            else "Ajouter au panier"
        )
    }
}
```

isAdded = false : "Ajouter au panier"

isAdded = true : "Produit ajouté"

L'interface est une conséquence de l'état.



Thinking in Compose

Réflexe historique vs Réflexe Compose

Réflexe historique

Je récupère une vue.

Je change ses propriétés.

Je pense en modifications successives.



Réflexe Compose

Je décris l'écran attendu.

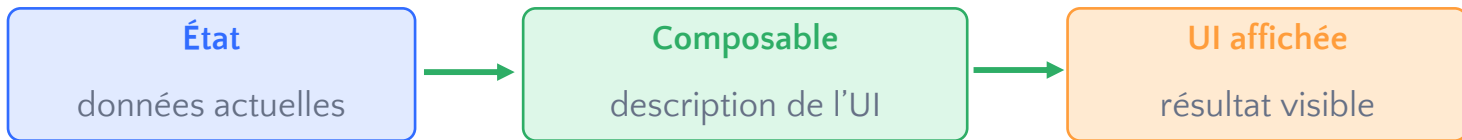
Je fais dépendre l'UI de données.

Je pense en états et événements.



Thinking in Compose

Principe central : état → interface



- Si l'état change, Compose peut réexécuter les composables concernés : c'est la **recomposition**.
- La **recomposition** est le mécanisme par lequel Compose met à jour l'interface quand les données utilisées par un composable changent.

```
@Composable
fun ProductCard(name: String, price: Double) {
    Text(text = "$name - $price €")
}
```

← Si name ou price change...
...l'affichage peut être *recalculé automatiquement* pour refléter le nouvel état.



Thinking in Compose

Exemple : une fiche-produit réutilisable

```
@Composable
fun ProductCard(
    name: String,
    price: Double
) {
    Text(text = "$name - $price €")
}

//...

ProductCard(
    name = "Sac à dos",
    price = 49.99
)
```

Sac à dos

49.99 €

Bonne pratique : le composable reçoit ses données. Il ne va pas les chercher lui-même.



2

Lecture du **projet généré** par Android Studio

Développement Android Jetpack Compose



Thinking in **Compose**

Les premiers éléments d'un projet Compose

🕒 Les premiers éléments d'un projet Compose

```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        enableEdgeToEdge()  
        setContent {  
            ShopExplorerTheme {  
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->  
                    WelcomeScreen(  
                        modifier = Modifier.padding(innerPadding)  
                    )  
                }  
            }  
        }  
    }  
}
```

Activity principale

Affichage edge-to-edge

Point d'entrée Compose

Thème Material

Structure d'écran

Marges calculées



Thinking in **Compose**

```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        //...  
    }  
}
```

Les premiers éléments d'un projet Compose

MainActivity et onCreate

- **MainActivity** : Point d'entrée visuel principal de l'application. Même avec Compose, on reste dans une application Android.
- **onCreate()** : Méthode appelée lors de la création de l'activité. C'est ici que l'on initialise le contenu principal.

*Compose modernise la création de l'interface, mais ne supprime pas les concepts Android : **cycle de vie**, **manifeste**, **permissions**, **ressources**.*



Thinking in **Compose**

```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        //...  
    }  
}
```

Les premiers éléments d'un projet Compose

Dans un projet d'application de base, les principales parties à connaître sont :

- 🕒 **Activité (Activity)** : Un écran avec une interface utilisateur et un contexte ;
- 🕒 **Ressources** : fichiers de mise en page (layout, images, audio, thèmes et couleurs) ;
- 🕒 **Le Manifeste** : AndroidManifest.xml ;
- 🕒 **Fichiers Gradle**



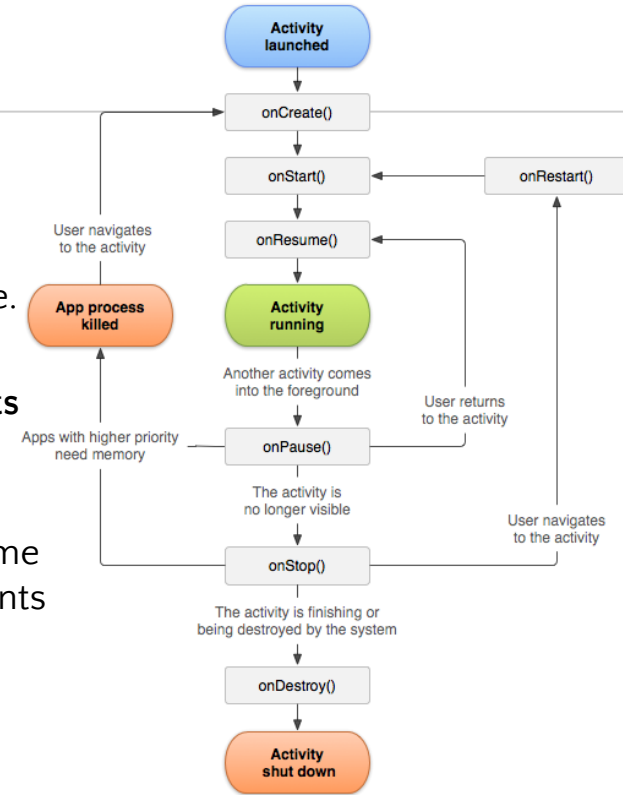
Thinking in Compose

- Une **Activity** représente un **point d'entrée visuel** d'une application Android.
- Elle permet d'afficher une interface utilisateur et de gérer une partie du **cycle de vie** de l'application.
- Habituellement, une **Activity** est présentée comme une **fenêtre plein écran**.
- Contrairement aux applications Android basées sur XML, dans une application Compose l'interface est généralement définie dans **une seule Activity principale**, souvent appelée « **MainActivity.kt** », contenant plusieurs écrans composables.
- La navigation entre ces écrans est généralement gérée avec **Navigation Compose**.



Thinking in Compose

- Dans le développement Android, la gestion des transitions entre les étapes du **cycle de vie** d'une activité est importante.
- La classe **Activity** possède plusieurs **méthodes principales**, appelées **callbacks**, qui permettent de gérer les **changements d'état de l'activité** : **onCreate()**, **onStart()**, **onResume()**, **onPause()**, **onStop()**, **onRestart()** et **onDestroy()**.
- Ces méthodes sont appelées **automatiquement** par le système Android pour permettre à l'**Activity** de réagir aux changements d'état : **création, affichage, passage au premier plan, pause, arrêt ou destruction**.





Thinking in Compose

Callback	Description simple
<code>onCreate()</code>	Appelée lors de la création de l' Activity , pour initialiser l'application et afficher l'interface avec <code>setContent { ... }</code> en Compose.
<code>onStart()</code>	Appelée quand l' Activity devient visible à l'utilisateur.
<code>onResume()</code>	Appelée quand l' Activity passe au premier plan et peut interagir avec l'utilisateur.
<code>onPause()</code>	Appelée quand l' Activity perd le focus , par exemple lorsqu'un autre écran ou une boîte de dialogue passe devant.
<code>onStop()</code>	Appelée quand l' Activity n'est plus visible à l'écran.
<code>onRestart()</code>	Appelée quand une Activity arrêtée revient à l'écran, juste avant <code>onStart()</code> .
<code>onDestroy()</code>	Appelée avant la destruction de l' Activity , pour libérer les ressources si nécessaire.



Thinking in Compose

Le Manifeste : AndroidManifest.xml

- Contiens la description de l'application
 - **Icône** de l'application ;
 - **Nom** affiché de l'application ;
 - **Thème** appliqué à l'application ou à certaines Activity ;
 - Déclare certains composants de l'application, comme les **Activity**, **services** ou **receivers**.
 - Liste des **caractéristiques matérielles** nécessaires, comme caméra, GPS ou Bluetooth
 - Ces déclarations peuvent influencer la visibilité de l'application sur **Google Play** selon l'appareil.
 - **Liste des permissions** : déclare les autorisations nécessaires, comme **Internet**, **caméra** ou **localisation** ;
 - **Métadonnées** : Peut contenir des informations supplémentaires, par exemple une clé API Google Maps.
- Il est créé **automatiquement** au démarrage du projet, mais peut être **modifié par le développeur**.

Exemple : AndroidManifest.xml



```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportRtl="true"
        android:theme="@style/Theme.GreetingCard">
        <activity
            android:name=".MainActivity"
            android:exported="true"
            android:label="@string/app_name"
            android:theme="@style/Theme.GreetingCard"
            android:windowSoftInputMode="adjustResize">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```



Thinking in Compose

Les Ressources

Ressources = toutes les données autres que le code, utilisées par l'application. Elles sont rangées dans le dossier « **res** », puis incluses dans l'**APK**.

Exemples :

- 🕒 **res/drawable** et **res/mipmap** : images, icônes et ressources graphiques ;
- 🕒 **res/values** : constantes définies en XML, comme les **chaînes**, **couleurs**, **dimensions**, **styles** ou **tableaux** ;
- 🕒 **res/font** : polices personnalisées ;
- 🕒 **res/raw** : fichiers bruts comme sons, vidéos ou fichiers JSON ;
- 🕒 **res/xml** : fichiers XML de configuration.



Thinking in Compose

Les Ressources : exemple – strings.xml

- Fichier ressources, contenant toutes les chaînes constantes

Type de la constante

```
<resources>
  <string name="app_name">GreetingCard</string>
  <string name="hello_world">Hello world!</string>
  <string name="action_settings">Settings</string>
</resources>
```

Nom de la constante (permet l'appel depuis l'application ou un autre fichier XML)

Valeur de la constante



Thinking in **Compose**

Les Ressources : La classe « R »

- Classe générée automatiquement par **Android Studio / Gradle**.
- Elle permet d'accéder aux ressources de l'application.
- Elle est créée à partir de **l'arborescence** présente dans le dossier « **res** ».
- Elle contient des **classes internes** correspondant aux différents types de ressources : *drawable, mipmap, string, color, font, xml, etc...*
- Elle contient des **identifiants** permettant de représenter les ressources de l'application.
Utilisation en Kotlin / Compose : **R.type.identificateur**.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="black">#FF000000</color>
  <color name="white">#FFFFFFFF</color>
</resources>
```

R.color.black

R.color.white



Thinking in **Compose**

```
enableEdgeToEdge()
setContent {
    ShopExplorerTheme {
        //... Contenu de l'application ...
    }
}
```

Les premiers éléments d'un projet Compose

enableEdgeToEdge() :

- Permet un rendu plus moderne : L'application peut dessiner **derrière les barres** système.
- Il faut ensuite protéger les contenus importants avec les **insets / padding**.

setContent { arbre de composables } :

- C'est la **racine de l'arbre** Compose. Tout ce qui se trouve à l'intérieur du bloc **setContent** correspond à l'interface Compose de l'écran.
- Avec XML, on associait une activité à un fichier XML de **layout**. Avec Compose, on associe une activité à un arbre de fonctions composables.



Thinking in **Compose**

```
enableEdgeToEdge()  
setContent {  
    ShopExplorerTheme {  
        //... Contenu de l'application ...  
    }  
}
```

Les premiers éléments d'un projet Compose

ShopExplorerTheme :

- Le **thème** enveloppe le contenu de l'application : il fournira plus tard **couleurs, typographies, formes, thème clair et sombre**.
- Pour l'instant, le thème généré suffit. On le modifiera dans la séance Material Design 3 pour donner une identité à notre appli.

Activity : point d'entrée Android



setContent : racine de l'arbre Compose



Theme : couleurs, typo, formes



Screen : contenu affiché



Thinking in **Compose**

```
Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
    WelcomeScreen(
        modifier = Modifier.padding(innerPadding)
    )
}
```

Les premiers éléments d'un projet Compose

Scaffold (échafaudage) :

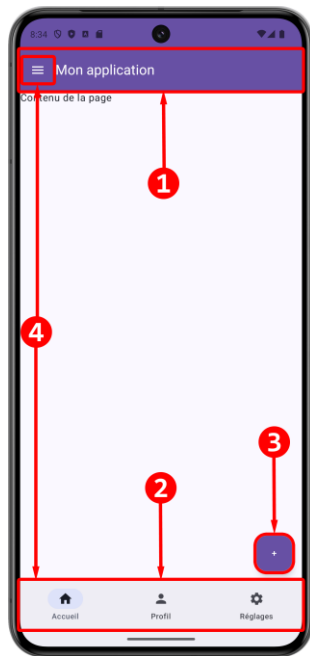
- C'est le **squelette** d'un écran. Il ne sert pas à afficher un composant en lui-même, mais à organiser les **grandes zones** de l'interface.
- **Exemples** : **topBar** pour le titre, **bottomBar** pour la navigation, **floatingActionButton** pour une action importante, **contenu principal** pour l'écran.
- Le paramètre **innerPadding** représente les **marges internes** calculées par **Scaffold**.
- **Modifier.fillMaxSize()** indique que le **Scaffold** doit occuper tout l'espace disponible.

```
Scaffold(
    modifier = Modifier.fillMaxSize(),
    topBar = { /* TopAppBar */ },
    floatingActionButton = { /* FAB */ },
    bottomBar = { /* NavigationBar */ }
) { innerPadding ->
    Content(modifier = Modifier.padding(innerPadding))
}
```

EXEMPLE



Thinking in **Compose**



Exemple Scaffold

1. **topBar** affiche la barre supérieure ;
2. **bottomBar** affiche la barre inférieure ;
3. **floatingActionButton** affiche un bouton flottant ;
4. **navigationIcon** dans **topBar** affiche l'Icon de navigation supérieur et **NavigationBar** dans **bottomBar** affiche un menu de navigation avec 3 entrées ;
5. le bloc final affiche le contenu principal.

```
Scaffold(  
    modifier = Modifier.fillMaxSize(),  
  
    topBar = {  
        TopAppBar(  
            title = {  
                Text("Mon application")  
            },  
            navigationIcon = {...},  
            colors = TopAppBarDefaults.topAppBarColors(...)   
        )  
    },  
  
    floatingActionButton = {...},  
  
    bottomBar = {  
        NavigationBar {...}  
    }  
  
) { innerPadding ->  
    Text(  
        text = "Contenu de la page",  
        modifier = Modifier.padding(innerPadding)  
    )  
}
```



Thinking in Compose

Fonctions composables

- Une **fonction composable** est une fonction Kotlin annotée avec **@Composable**, utilisée pour décrire une partie de l'interface utilisateur.
- Elle **peut appeler** d'autres composables, ne **retourne généralement rien**, doit rester **rapide** et éviter les effets de bord non contrôlés.
- Le paramètre **Modifier** permet de personnaliser l'**apparence**, la **disposition** ou le **comportement** d'un composant, par exemple avec **padding()**, **fillMaxWidth()** ou d'autres **modifiers** chaînés.
- **Surface** est un conteneur **Material Design 3** qui permet de créer une **zone visuelle** avec une **couleur**, une **forme**, une **élévation** ou un **modifier**.
- Les **Preview** permettent **d'afficher rapidement** un composant dans Android Studio **sans relancer** toute l'application, ce qui facilite le **test** et le **développement** de l'interface.

```
@Composable
fun ProductCard(name: String, modifier: Modifier = Modifier)
{
    Surface(modifier = modifier.padding(16.dp)) {
        Text(text = name, modifier = Modifier.padding(16.dp))
    }
}
@Preview(showBackground = true)
@Composable
fun ProductCardPreview() {
    ProductCard(name = "Produit")
}
```



Thinking in Compose

Ce qu'il faut retenir avant le TP

Avant de passer à la pratique, **retenez ces idées** :

- **Compose** permet d'écrire l'interface Android en Kotlin.
- Une fonction annotée **@Composable** décrit une partie de l'interface.
- **L'interface est déclarative** : elle dépend des données qu'on lui donne.
- **setContent** permet de définir l'interface **Compose** d'une **Activity**.
- **Text** permet d'afficher du texte.
- **Surface** permet de créer une zone visuelle.
- **Modifier** permet de modifier l'apparence, la taille, la position ou le comportement d'un composant.
- **@Preview** permet de visualiser un composant dans Android Studio.



Thinking in Compose

Passons maintenant aux **Travaux Pratiques** :

- 🕒 **TP 1** : Créer votre première application Android Compose
- 🕒 **TP 2** : Créer un écran dynamique avec Compose



Merci !

Des *questions* ?

Vous pouvez me contacter au :

- 🕒 **Email** : a.vardanyan@itds.fr
- 🕒 **Teams** : avardanyan2@myges.fr



Sources de référence et suite du cours

- **Android Developers – Jetpack Compose overview** : toolkit moderne recommandé pour l'UI native.
- **Android Developers – Scaffold** : structure d'écran Material avec topBar, bottomBar, FAB et padding interne.
- **Android Developers – insets / edge-to-edge** : gestion des zones système et contenus non masqués.
- **Prochaine séance** : arbre de composition, recomposition et flux de données unidirectionnel.