Séance 3 – *Modifiers* et bonnes pratiques Compose

1. Objectifs de la séance

Dans la séance précédente, vous avez construit un écran Compose simple à partir de plusieurs composables : un écran de **démonstration**, une **carte** produit, un **titre**, un **prix**, un **état** de disponibilité et un bouton **favori**.

Vous avez déjà vu qu'un composable reçoit des données en paramètres et décrit l'interface correspondante. Maintenant, nous allons améliorer la manière d'écrire ces composables.

À la fin de cette séance, vous devez être capables de :

- expliquer le rôle de ***Modifier*** ;
- chaîner plusieurs **modifiers** correctement ;
- comprendre pourquoi l'ordre des **modifiers** est important ;
- distinguer **padding externe** et **padding interne** ;
- écrire des composables plus réutilisables ;
- passer un ***Modifier*** en paramètre d'un composable ;
- ajouter de premières informations **d'accessibilité** ;
- éviter quelques mauvaises pratiques fréquentes.

L'objectif n'est pas encore d'apprendre tous les composants Compose. Les composants plus riches comme *Card*, *Image*, *Icon* ou *TextField* arriveront dans le bloc suivant. Nous travaillons surtout la qualité d'écriture des composables.

2. Rappel rapide : où nous en sommes

Dans le TP précédent, vous avez construit une structure proche de celle-ci :

```
@Composable
fun ProductDemoScreen(modifier: Modifier = Modifier) {
    ProductCard(
        name = "Sac à dos",
        price = 49.99,
        isAvailable = true,
        isFavorite = false,
        onFavoriteClick = {
            // Pour l'instant, nous ne modifions pas encore l'état.
            // Nous verrons cela dans la séance sur la gestion d'état.
        },
        modifier = modifier.padding(24.dp)
    )
}
```

L'idée importante était la suivante :

- ***ProductDemoScreen*** décide ce qui est affiché ;
- ***ProductCard*** décrit une fiche produit ;
- ***ProductTitle***, ***ProductPrice***, ***ProductAvailability*** et ***FavoriteButton*** sont des petits composants réutilisables ;
- les **données descendent** vers les composables ;
- les **événements remontent** via des callbacks.

Dans cet séance, nous allons améliorer cette base avec de meilleures pratiques autour de ***Modifier***.

3. À quoi sert *Modifier* ?

Modifier est un mécanisme central dans Jetpack Compose. Il permet de **modifier un composable sans changer directement sa logique interne**.

Avec un ***Modifier***, on peut par exemple :

- ajouter de **l'espace autour** ou **dans** un composant ;
- définir une **largeur** ou une **hauteur** ;
- rendre un élément **cliquable** ;
- ajouter une **couleur de fond** ;
- gérer une **position** ;
- transmettre des informations **d'accessibilité** ;
- modifier certains **comportements** d'interaction.

Par exemple, dans le code suivant, on ne modifie pas le composant ***Text*** lui-même. On lui ajoute un comportement de **mise en page** : une **marge intérieure** de *16.dp*.

```
Text(
    text = "Sac à dos",
    modifier = Modifier.padding(16.dp)
)
```

Autre exemple, ici, on demande au composant d'occuper **toute la largeur** disponible :

```
Text(
    text = "Sac à dos",
    modifier = Modifier.fillMaxWidth()
)
```

Modifier est donc une manière de dire : **Ce composant reste le même, mais je veux modifier sa taille, son espace, son comportement ou certaines informations associées**.

4. Pourquoi passer *modifier* en paramètre ?

Une bonne pratique fréquente en Compose consiste à **faire accepter un paramètre *modifier*** aux composables réutilisables. Par exemple :

```
@Composable
fun ProductTitle(
    name: String,
    modifier: Modifier = Modifier
) {
    Text(
        text = name,
        modifier = modifier
    )
}
```

Pourquoi est-ce intéressant ?

Parce que le composable ***ProductTitle*** ne décide pas tout seul où il doit être placé. Le parent peut écrire :

```
ProductTitle(
    name = "Sac à dos",
    modifier = Modifier.padding(16.dp)
)
```

ou bien :

```
ProductTitle(
    name = "Casque audio",
    modifier = Modifier.fillMaxWidth()
)
```

Le même composable peut donc être utilisé dans plusieurs contextes.

C'est une idée importante :
Un composable réutilisable doit éviter d'imposer trop fortement son positionnement externe.
Il doit décrire son contenu, mais laisser son parent décider comment l'intégrer dans l'écran.

5. *Padding* externe et *padding* interne

Il faut faire attention à une distinction importante.

Quand on écrit :

```
ProductCard(
    //...
    modifier = Modifier.padding(24.dp)
)
```

on ajoute généralement un **espace autour de la carte**. C'est un ***padding externe*** du point de vue de la carte.

Quand on écrit à **l'intérieur** de ***ProductCard*** :

```
Column(
    modifier = Modifier.padding(16.dp)
) {
    ProductTitle(name = name)
    ProductPrice(price = price)
    //...
}
```

on ajoute un **espace à l'intérieur** de la carte, **autour de son contenu**.

Ce sont deux responsabilités différentes :

- le **parent** décide de **l'espace autour** du composant ;
- le **composant** décide de **l'espace entre son bord et son contenu**.

Par exemple :

```
@Composable
fun ProductDemoScreen(modifier: Modifier = Modifier) {
    ProductCard(
        name = "Sac à dos",
        price = 49.99,
        modifier = modifier.padding(24.dp)
    )
}
```

Ici, **l'écran** décide que la carte doit être placée **avec un espace** de *24.dp*.

Puis :

```
@Composable
fun ProductCard(
    name: String,
    price: Double,
    modifier: Modifier = Modifier
) {
    Surface(modifier = modifier) {
        Column(
            modifier = Modifier.padding(16.dp)
        ) {
            ProductTitle(name = name)
            ProductPrice(price = price)
        }
    }
}
```

Ici, la **carte** décide que son contenu doit avoir un **espace interne** de *16.dp*.

Cette séparation rend le composant plus propre.

6. Chaîner les *modifiers*

Un *Modifier* peut être chaîné. Par exemple :

```
Modifier
    .fillMaxWidth()
    .padding(16.dp)
```

Cela signifie que l'on applique plusieurs transformations à la suite.

On peut par exemple écrire :

```
Surface(
    modifier = Modifier
        .fillMaxWidth()
        .padding(24.dp)
) {
    Text(text = "Produit")
}
```

Ici, on demande à la ***Surface*** :

1. de **prendre toute la largeur** disponible ;
2. d'avoir un espace de *24.dp*.

Le chaînage est très courant en Compose. Il permet d'écrire **progressivement** ce que l'on veut faire avec un composant.

7. L'ordre des *modifiers* est important

L'ordre des **modifiers** n'est pas **décoratif**. Il change parfois le résultat. Prenons deux exemples.

Premier ordre :

```
Modifier
    .padding(16.dp)
    .fillMaxWidth()
```

Deuxième ordre :

```
Modifier
    .fillMaxWidth()
    .padding(16.dp)
```

Ces deux chaînes ne signifient pas exactement la même chose.

Dans le premier cas, **on applique d'abord un espace**, puis on demande à l'élément de prendre la **largeur maximale** dans l'espace restant.

Dans le second cas, on demande d'abord à l'élément de **prendre toute la largeur** disponible, puis on **applique un espace**.

Le même principe existe avec ***clickable***. Par exemple :

```
Modifier
    .clickable(onClick = onClick)
    .padding(16.dp)
```

Dans ce cas, la zone cliquable peut inclure l'espace ajouté ensuite.

Alors que :

```
Modifier
    .padding(16.dp)
    .clickable(onClick = onClick)
```

peut produire une zone cliquable différente.

Ce qu'il faut retenir :
*En Compose, l'ordre des **modifiers** décrit explicitement l'ordre des **transformations**.*
*Cela rend le système puissant, mais cela demande de lire attentivement la **chaîne**.*

8. Réutiliser des composants, pas seulement du code

Réutiliser un composant ne veut pas simplement dire copier-coller une fonction. Un composant réutilisable doit être **suffisamment flexible**.

Exemple peu réutilisable :

```
@Composable
fun ProductTitle() {
    Text(
        text = "Sac à dos",
        modifier = Modifier.padding(16.dp)
    )
}
```

Ce composant pose deux problèmes :

1. il affiche toujours le **même texte** ;
2. il impose son propre **padding**.

Version plus réutilisable :

```
@Composable
fun ProductTitle(
    name: String,
    modifier: Modifier = Modifier
) {
    Text(
        text = name,
        modifier = modifier
    )
}
```

Maintenant, le parent peut décider :

```
ProductTitle(
    name = "Sac à dos",
    modifier = Modifier.padding(8.dp)
)
```

ou :

```
ProductTitle(
    name = "Casque audio",
    modifier = Modifier.fillMaxWidth()
)
```

Le composable devient plus **adaptable**.

9. Bonne pratique : le parent place, l'enfant affiche

Une règle simple peut vous aider :

- → Le **parent décide** du placement externe.
- → **L'enfant décrit** son contenu.

Par exemple, ***ProductDemoScreen*** peut décider que la carte doit avoir un **padding externe** :

```
ProductCard(
    name = "Sac à dos",
    price = 49.99,
    modifier = Modifier.padding(24.dp)
)
```

Mais ***ProductCard*** peut décider que son **contenu interne** a besoin d'un **padding** :

```
Surface(modifier = modifier) {
    Column(
        modifier = Modifier.padding(16.dp)
    ) {
        ProductTitle(name = name)
        ProductPrice(price = price)
    }
}
```

Cette **règle** évite que les composants deviennent difficiles à réutiliser. Pae exemple, si un composant impose trop de marges externes, il devient compliqué à intégrer ailleurs.

10. Accessibilité de base

L'accessibilité consiste à rendre une application utilisable par **davantage de personnes**, notamment celles qui utilisent des **lecteurs d'écran**, des tailles de **police plus grandes** ou d'autres outils **d'assistance**.

Dans Compose, certains composants **Material** fournissent déjà des informations utiles. Par exemple, un ***Button*** expose naturellement un rôle de bouton.

Mais dès que vous créez vos propres composants cliquables, vous devez faire attention ! Exemple :

```
Surface(
    modifier = Modifier.clickable(onClick = onClick)
) {
    Text(text = "Voir le produit")
}
```

Visuellement, cela peut ressembler à un **élément cliquable**. Mais pour un service **d'accessibilité**, il peut **manquer des informations** : *quel est le **rôle de cet élément** ? Que **signifie** le clic ?*

On peut enrichir cela avec des informations sémantiques. Exemple :

```
Surface(
    modifier = Modifier
        .clickable(
            role = Role.Button,
            onClick = onClick
        )
        .semantics {
            contentDescription = "Ouvrir le détail du produit Sac à dos"
        }
) {
    Text(text = "Sac à dos")
}
```

Ici, on indique que l'élément se comporte **comme un bouton**, et on fournit une **description compréhensible**.

*Il ne faut **pas abuser des descriptions** non plus. Un texte déjà visible n'a pas toujours besoin d'être répété. Mais lorsqu'une action n'est pas évidente, ou lorsqu'un composant personnalisé est cliquable, il faut penser à ce qui sera compris sans l'image ou sans le contexte visuel.*

11. Quelques règles pratiques à retenir

Pour cette séance, retenez surtout ces règles :

1. Un composable réutilisable accepte généralement un *modifier **: Modifier = Modifier***.
2. Le ***modifier*** reçu doit être transmis au premier composant qui émet réellement l'UI.
3. Le **parent** décide de l'espace **externe**.
4. Le **composant** décide éventuellement de son espace **interne**.
5. L'ordre des ***modifiers*** compte.
6. Un composant **interactif** doit **exposer** clairement ses **actions**.
7. **L'accessibilité** doit être pensée dès les petits composants, **pas seulement à la fin du projet**.
8. Il vaut mieux créer **plusieurs petits composables** clairs qu'un grand composable difficile à lire.