Développement Android avec Kotlin

Cours - 03 - Introduction à Jetpack Compose

Jordan Hiertz

Contact
hiertzjordan@gmail.com
jordan.hiertz@al-enterprise.com

Présentation de Jetpack Compose

- **Framework moderne d'UI déclarative pour Android**
  - Développé par Google pour remplacer les layouts basés sur XML
  - Plus simple, plus flexible et réactif que l'approche traditionnelle

- **Basé sur Kotlin**
  - Langage officiel d'Android, avec une syntaxe expressive et concise

- **Construction d'UI réactive**
  - UI qui réagit automatiquement aux changements de données via la gestion de l'état

- **Interopérabilité avec les composants Android existants**
  - Peut être utilisé avec ou à côté des vues XML traditionnelles

- **Moins de code, moins de complexité**
  - Les composants UI sont directement dans le code Kotlin, facilitant la maintenance

D’où vient-on ? Le système de Views

- **Le modèle classique basé sur XML**
  - Les interfaces utilisateurs sont définies dans des fichiers XML séparés.
  - Les composants UI (TextView, Button, ImageView etc.) sont hiérarchisés sous forme de vues.
  - La logique de l'application est définie dans les classes Kotlin, séparée de la définition de l'UI.

- **Couplage entre XML et code**
  - Les vues XML doivent être référencées dans le code à l'aide de la méthode `findViewById()`.
  - Peut vite devenir complexe avec de grandes hiérarchies de vues.

- **Layouts statiques et moins flexibles**
  - Les changements d'UI nécessitent souvent des ajustements complexes des fichiers XML.

- **Performance et maintenance**
  - Des hiérarchies de vues profondes peuvent affecter les performances

Views vs Compose

```xml
<LinearLayout
    xmlns:android="http://schemas.android.com/apk
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, World!" />

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click me" />
</LinearLayout>
```


```kotlin
@Composable
fun Greeting() {
    Column {
        Text(text = "Hello, World!")
        Button(onClick = { /* Action */ }) {
            Text("Click me")
        }
    }
}
```

Raisonnement dans Compose

- **Les éléments de l'UI sont des fonctions et non des objets.**
  - On ne peut pas les trouver par référence ou appeler des méthodes pour les muter.
  - Ils sont contrôlés par les états/arguments passés en paramètre.

```
1 @Composable
2 fun Answer(answer: Answer) {
3     Row {
4         Image(answer.image)
5         Text(answer.text)
6         RadioButton(false, onClick = { /* ... */ })
7     }
8 }
```

Permet de définir un composant

Chaque éléments UI est une fonction

Pour afficher l'état, on passe simplement la propriété à la fonction `Image`

Puis `Text`

Puis `RadioButton`

Raisonnement dans Compose

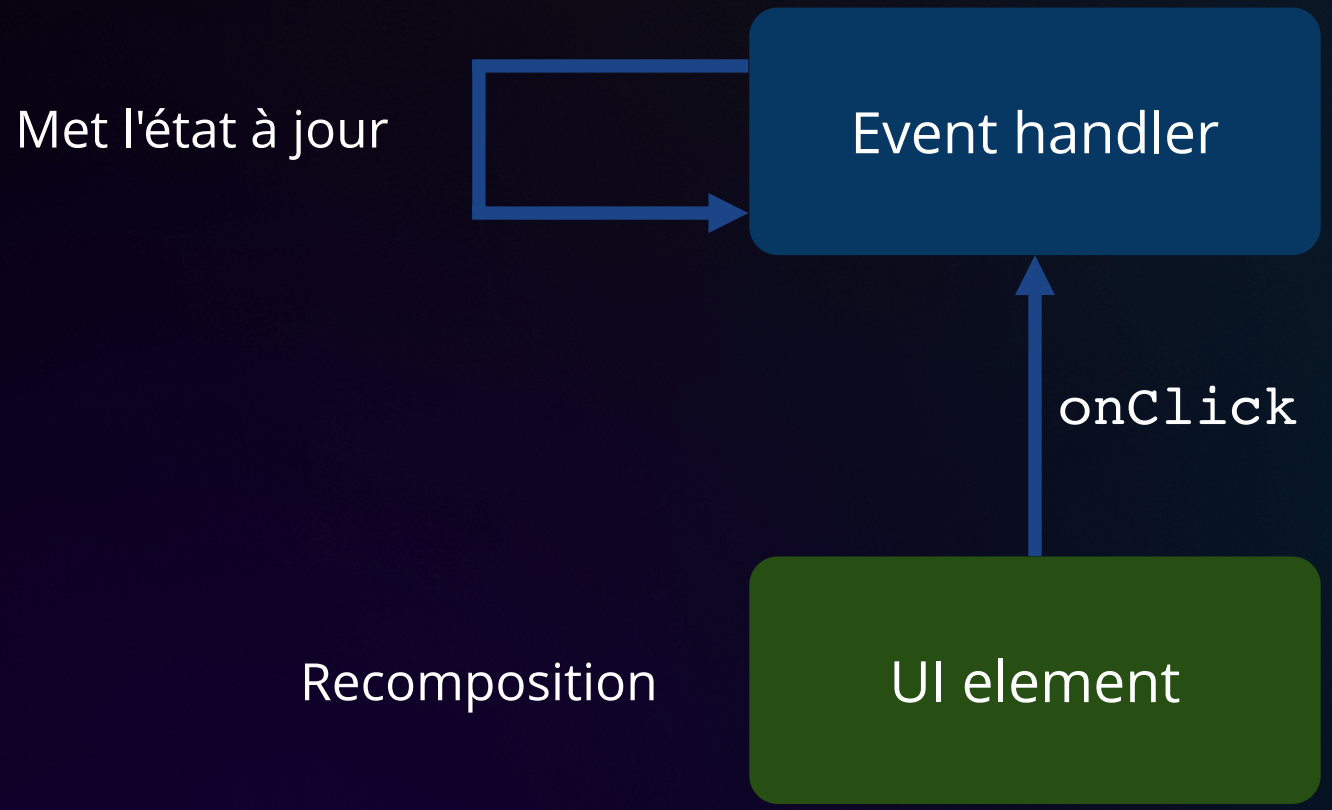
Construire l'interface utilisateur en décrivant le quoi et non le comment

- **Déclaratif** : En Jetpack Compose, on se concentre sur ce que l'UI doit afficher, pas sur les étapes détaillées pour y parvenir.
    - **Définir l'UI** : Nous déclarons à quoi doit ressembler notre interface en fonction de l'état.
    - **Compose gère le rendu** : Nous n'indiquons pas à Compose comment il doit dessiner ou rendre chaque composant.

Cette approche simplifie la gestion des interfaces complexes en évitant de manipuler directement le cycle de vie ou l'affichage.

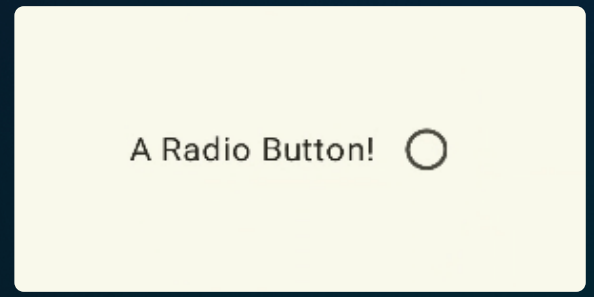
Si l'état contrôle l'interface utilisateur, comment pouvons-nous mettre à jour l'état pour déclencher une mise à jour de l'interface ?

Raisonnement dans Compose



Raisonnement dans Compose

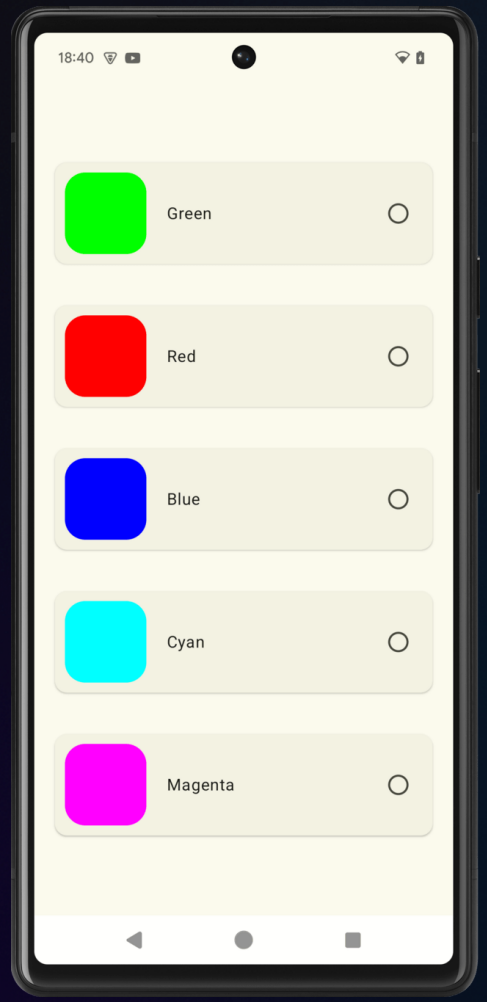
```
@Composable
fun Answer(answer: Answer) {
    Row {
        /** ... **/

        var isSelected by remember { mutableStateOf(false) }

        RadioButton(
            selected = isSelected,
            onClick = { isSelected = !isSelected }
        )
    }
}
```



En résumé

1. **Décrire "quoi" et non "comment"**
    - On spécifie l'apparence et le comportement sans manipuler directement des vues.
2. **Les éléments de l'interface utilisateur sont des fonctions**
    - Chaque composant est une fonction qui génère des éléments graphiques en fonction des paramètres.
3. **L'état contrôle l'interface utilisateur**
    - Le rendu de l’interface dépend entièrement de l’état fourni aux composants.
4. **Les événements contrôlent l'état**
    - Les interactions de l'utilisateur ou les événements modifient l'état, ce qui déclenche la mise à jour de l’UI.

Les fonctions composables



Les fonctions composables

```
1 @Composable
2 fun Answer(answer: Answer) {
3     Row {
4         Image(answer.image)
5         Text(answer.text)
6         RadioButton(selected = false, onClick = { /** ... **/})
7     }
8 }
```

- Une fonction **@Composable** crée un composant UI.
- Cette annotation indique au compilateur que la fonction convertit des données en UI.
- Les fonctions composables sont les **briques de base** de l'interface dans Compose.
- Elles encouragent à **décomposer l'UI** en éléments réutilisables.

Les fonctions composables

```
@Composable
fun Answer(answer: Answer) { /* .... */ }

@Composable
fun SingleChoiceQuestion(answers: List<Answer>) {
    Column {
        answers.forEach { answer ->
            Answer(answer = answer)
        }
    }
}
```

- **Paramètres configurables** : La fonction prend un paramètre `List<Answer>`, permettant à l'UI de s'adapter dynamiquement en fonction des données fournies.
- **Aucune valeur de retour** : Une fonction composable n'a pas besoin de renvoyer une valeur, elle **émet directement** l'interface utilisateur.

Les fonctions composables

```
1 @Composable
2 fun SingleChoiceQuestion(answers: List<Answer>) {
3     Column {
4         answers.forEach { answer ->
5             // Erreur : On ne peut pas stocker un composable !
6             val answer = Answer(answer = answer)
7         }
8     }
9 }
```

- **Immutabilité des composables** : Les composables sont **immutables**, on ne peut pas les stocker pour mise à jour ultérieure.
- **Pas de stockage de référence** : Impossible de conserver un composable dans une variable pour le modifier.
- **Mise à jour via paramètres** : Toute mise à jour se fait en **repassant les paramètres** lors de l'appel.

Les fonctions composables

```
1 @Composable
2 fun SingleChoiceQuestion(answers: List<Answer>) {
3     Column {
4         answers.forEach { answer ->
5             Answer(answer = answer)
6         }
7     }
8 }
```

- **Utilisation de la syntaxe Kotlin** : En tant que fonction Kotlin, on peut utiliser des **structures et fonctions du langage** (comme `forEach`) pour construire l'UI.

Les fonctions composables

```
1  @Composable
2  fun SingleChoiceQuestion(answers: List<Answer>) {
3      Column {
4          if (answers.isEmpty()) {
5              Text("There are no answers !!")
6          } else {
7            answers.forEach { answer ->
8                Answer(answer = answer)
9            }
10         }
11     }
12 }
```

- **Affichage conditionnel** : Utiliser un simple **if statement** pour contrôler l'affichage. Pas besoin de manipuler `view.visibility = gone` comme dans l'approche traditionnelle Android.
- On appelle simplement le composable s'il doit être visible.

Les fonctions composables

```
@Composable
fun SingleChoiceQuestion(answers: List<Answer>) {
    // Ne pas faire ça, pas d'effets secondaires !
    SomeSingleton.meaningfulVariable = true

    Column {
        /* ... */
    }
}
```

- **Pas d'effets secondaires** : Une fonction composable doit être **pure**, sans effets secondaires. Elle ne doit pas modifier de variables globales ou des objets extérieurs.
- **Idempotence :** Elle doit se comporter **de la même façon** si elle est appelée plusieurs fois avec les mêmes arguments.

Les fonctions composables

```
1 @Composable
2 fun SingleChoiceQuestion(answers: List<Answer>) {
3     Column {
4         answers.forEach { answer ->
5             Answer(answer = answer)
6         }
7     }
8 }
```

- **L'UI contrôlée par les paramètres** : Les paramètres fournis contrôlent entièrement l'UI.
- **Transformation d'état en UI** : La fonction transforme **l'état** (comme la liste `answers`) en une interface utilisateur.
- **Pas de désynchronisation** : Si l'état (la liste) change, **une nouvelle UI** est automatiquement générée.
- **Recomposition** : Ce processus de mise à jour automatique est appelé **recomposition**.

Recomposition

La **recomposition** survient quand un composable est **ré-invoqué** avec des paramètres différents ou quand **l'état interne** d'un composable change.

Recomposition

```
1 @Composable
2 fun SingleChoiceQuestion(answers: List<Answer>) {
3   answers.forEach { answer ->
4     Answer(
5         answer = answer,
6         isSelected = false,
7     )
8   }
9 }
```

Le composable `Answer` prend un paramètre `isSelected` (boolean) pour indiquer si la réponse est sélectionnée.

- Dans le système classique des `Vues`, un clic mettrait à jour l'UI automatiquement.
- Ici, chaque `Answer` est toujours passé avec `isSelected = false`, donc ils restent **désélectionnés**, même après interaction.

Recomposition

```
1  @Composable
2  fun SingleChoiceQuestion(answers: List<Answer>) {
3      var selectedAnswer: Answer? = null
4      answers.forEach { answer ->
5        Answer(
6            answer = answer,
7            isSelected = false,
8        )
9      }
10 }
```

On a besoin que le composable recompose dès que l'utilisateur interagit avec l'UI.

Pour cela, nous avons besoin d'une variable `selectedAnswer` qui mémorise la réponse choisie.

Recomposition

```
1  @Composable
2  fun SingleChoiceQuestion(answers: List<Answer>) {
3      var selectedAnswer: MutableState<Answer?> = mutableStateOf(null)
4      answers.forEach { answer ->
5        Answer(
6            answer = answer,
7            isSelected = selectedAnswer.value == answer,
8        )
9      }
10 }
```

- **On encapsule l'objet `Answer` dans un `MutableState`.**
- Qu'est-ce qu'un `MutableState` ?
  - C'est un observable intégré à Compose.
  - Chaque changement à ce state déclenche automatiquement la recomposition de tous les composables qui l'utilisent.

Recomposition

```
1  @Composable
2  fun SingleChoiceQuestion(answers: List<Answer>) {
3      var selectedAnswer: MutableState<Answer?> =
4          remember { mutableStateOf(null) }
5
6      answers.forEach { answer ->
7        Answer(
8            answer = answer,
9            isSelected = selectedAnswer.value == answer,
10       )
11     }
12 }
```

- **Créer un objet state nécessite d'utiliser `remember { }`.**

  Cela garantit que la valeur ne sera pas réinitialisée entre deux recompositions.

- Avec ce qu'on a ici, l'UI survit à la recomposition, mais pas au changement de configuration.

Recomposition

```
1  @Composable
2  fun SingleChoiceQuestion(answers: List<Answer>) {
3      var selectedAnswer: MutableState<Answer?> =
4          rememberSaveable { mutableStateOf(null) }
5
6      answers.forEach { answer ->
7        Answer(
8            answer = answer,
9            isSelected = selectedAnswer.value == answer,
10       )
11     }
12 }
```

- **Créer un objet state nécessite d'utiliser `remember { }`.**

  Cela garantit que la valeur ne sera pas réinitialisée entre deux recompositions.

- Avec ce qu'on a ici, l'UI survit à la recomposition, mais pas au changement de configuration.

  Pour cela, on peut utiliser `rememberSaveable`.

Recomposition

```
1  @Composable
2  fun SingleChoiceQuestion(answers: List<Answer>) {
3      var selectedAnswer: MutableState<Answer?> =
4          rememberSaveable { mutableStateOf(null) }
5
6      answers.forEach { answer ->
7        Answer(
8            answer = answer,
9            isSelected = selectedAnswer.value == answer,
10       )
11     }
12 }
```

Avec ce code, `selectedAnswer` est un `MutableState`.

Recomposition

```kotlin
@Composable
fun SingleChoiceQuestion(answers: List<Answer>) {
    var selectedAnswer: Answer? by rememberSaveable { mutableStateOf(null) }

    answers.forEach { answer ->
      Answer(
          answer = answer,
          isSelected = selectedAnswer == answer,
      )
    }
}
```

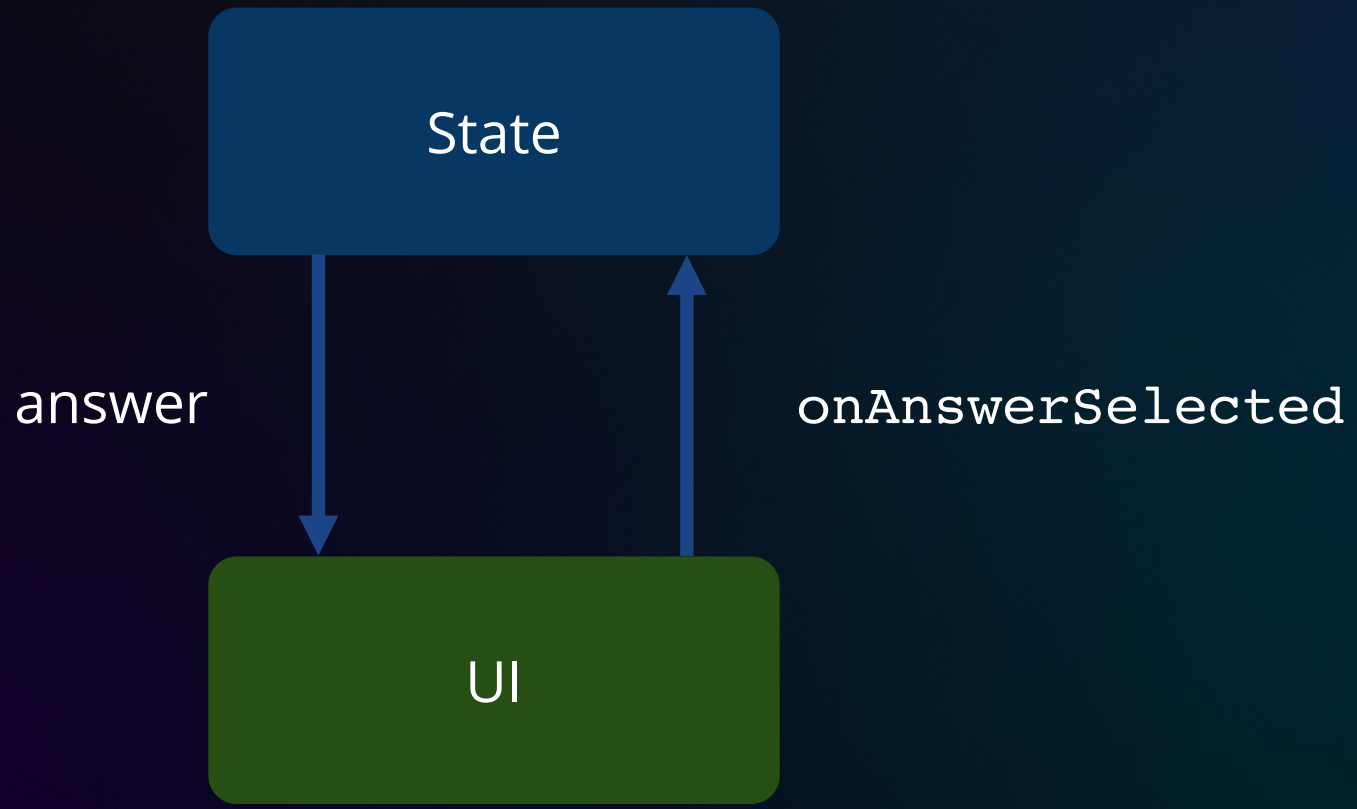
- Nous pouvons utiliser la syntaxe des propriétés déléguées de Kotlin avec le mot-clé `by`.
- Notre variable `selectedAnswer` devient directement de type `Answer?`, ce qui simplifie son utilisation.
- Cette syntaxe permet de travailler directement avec `selectedAnswer` sans avoir à utiliser la propriété `value`, rendant le code plus clair et plus concis.

Recomposition

```
1  @Composable
2  fun SingleChoiceQuestion(answers: List<Answer>) {
3      var selectedAnswer: Answer? by rememberSaveable { mutableStateOf(null) }
4
5      answers.forEach { answer ->
6        Answer(
7            answer = answer,
8            isSelected = selectedAnswer == answer,
9            onAnswerSelected = { answer -> selectedAnwser = answer }
10       )
11     }
12 }
```

- Grâce à notre nouvel état, nous pouvons passer une lambda en paramètre à notre composable `Answer`. Cela permet d'effectuer une action lorsque l'utilisateur clique sur un élément.
- Dans cette lambda, nous mettons à jour la valeur de `selectedAnswer` avec la réponse sélectionnée.

Recomposition



Comportements dans Jetpack Compose

Les fonctions composables doivent être **prédictibles** et **sans effets secondaires**. Elles doivent toujours **produire le même résultat pour les mêmes entrées** (principe de pureté fonctionnelle). Cela garantit une interface utilisateur stable et réactive à l'état.

Comportements : Ordre d'exécution des fonctions composables

- Les fonctions composables peuvent **s'exécuter dans un ordre non séquentiel.**
- Bien qu'on puisse penser que le code s'exécute dans l'ordre où il est écrit, **Compose optimise l'exécution** en fonction des besoins de performance et des priorités d'affichage.
- **Les éléments les plus prioritaires**, comme ceux liés aux interactions utilisateur, **peuvent être dessinés en premier**, même s'ils apparaissent plus tard dans le code.

Comportements : Exécution parallèle des fonctions composables

- Les fonctions composables peuvent **s'exécuter en parallèle**.
- Compose utilise les architectures multi-core pour **exécuter plusieurs tâches simultanément**, améliorant ainsi les performances de rendu d'un écran.
- **Pour cette raison, il est crucial d'éviter les effets secondaires**, comme l'écriture dans des variables globales, afin de prévenir des comportements inattendus dans un environnement parallèle.

Comportements : Optimisation de la recomposition

- **La recomposition évite autant d'étapes que possible**.
- Compose **recompose uniquement les parties de l'UI** qui nécessitent une mise à jour.
- Si un composable **n'utilise pas l'état** qui a déclenché la recomposition, il est **ignoré**.
- Compose peut recomposer uniquement les sous-parties pertinentes d'un composable pour améliorer les performances.
- Bien structurer ses états permet de minimiser les recompositions, ce qui réduit les calculs inutiles et améliore l'efficacité.

Comportements : Recomposition optimiste

- La recomposition est **optimiste**.
- Compose **prévoit de terminer la recomposition** avant que les paramètres ne changent.
- Si les paramètres changent avant la fin, Compose peut **annuler la recomposition** en cours et la recommencer avec les nouveaux paramètres.
- Cette technique permet à Compose de gérer les interfaces dynamiques de manière fluide, même lorsque les états évoluent rapidement.

Comportements : Fréquence de la recomposition

- La recomposition peut se produire **fréquemment**, notamment lorsqu'un composable **joue une animation**.
- Les recompositions peuvent avoir lieu à chaque **frame** de l'animation.
- Il est essentiel de rendre le composable **rapide et efficace** pour éviter de perdre des frames.
- Il est recommandé de **minimiser les opérations lourdes** dans les composables, et d'utiliser des outils comme `remember` pour stocker les états non nécessaires à la recomposition.

En résumé

1. **@Composable :** Une fonction devient un composable en étant annotée avec `@Composable`.

2. **Facile à créer :** La création d'un composable est simple, ce qui encourage une architecture modulaire basée sur des composants réutilisables.

3. **Paramétrables** : Les composables acceptent des paramètres et **doivent** en prendre pour configurer leur comportement.

4. **Gestion de l'état : MutableState et remember :** Ils permettent de conserver l'état d'un composable et d'assurer que Compose réagira aux changements d'état

5. **Sans effets secondaires :** Les composables ne doivent pas provoquer d'effets secondaires pour garantir la prévisibilité et éviter des comportements imprévisibles.

En résumé

- Les composables peuvent :
  - **S'exécuter dans n'importe quel ordre** : Compose ne suit pas forcément l'ordre d'apparition du code.
  - **S'exécuter en parallèle** : Compose peut profiter des architectures multi-core pour améliorer les performances.
  - **Être ignorés** : Les composables non affectés par un changement d'état peuvent être ignorés pour optimiser les performances.
  - **S'exécuter fréquemment** : En particulier lors d'animations, où chaque frame peut nécessiter une recomposition.

Conclusion

- **Compose** : Un framework moderne pour créer des interfaces utilisateur déclaratives.
- **L'état dans Compose** : La gestion des états réactifs avec `remember` et `MutableState`
- **Les fonctions composables** : La création de composants UI flexibles et réutilisables avec `@Composable`.
- **La recomposition** : Le mécanisme qui met à jour l'interface utilisateur en fonction des changements d'état.
- **Les comportements dans Compose** : Exécution parallèle, ordre d'exécution flexible, optimisation des performances.