Le TP consiste à créer une application Android Compose qui affiche une liste de données. Ce sera une liste de messages. Ces messages sont un texte accompagné d'un icône. Toute cette partie du TP s'appuie sur le TP4. Vous avez intérêt à bien le relire quand c'est demandé.

1. Affichage d'une liste de messages

👉 Relisez le début du TP4, §1.1 pour savoir comment on crée un projet Jetpack Compose.

👉 Créez le projet TP5 exactement comme le TP4, dépendance incluse.

👉 Vérifiez que la compilation se passe bien et corrigez les versions dans `libs.versions.toml` s'il y a un problème.

👉 Ajoutez trois sous-packages dans le projet : `messages.state`, `messages.controller` et `messages.ui`

1.1. État

👉 Créez `MessageListState.kt` dans le package `fr.iutlan.tp5.messages.state` et mettez ce contenu :

```
package fr.iutlan.tp5.messages.state

enum class Icone {
    ETOILE,
    APPEL,
    FAVORI
}
```

👉 Dans le même fichier, ajoutez une classe de données appelée `Message` définie par deux propriétés constantes `icone` de type `Icone` et `texte` de type `String`. Relisez le TP4 pour savoir comment on définit une telle classe.

👉 Ajoutez la propriété suivante :

```
val id: UUID = UUID.randomUUID(),
```

Cette propriété sert à identifier le message d'une manière unique.

NB: pour simplifier, on met tout l'état de l'application dans un seul fichier, mais évidemment, on devrait séparer les classes.

👉 Ajoutez une classe de données appelée `MessageListState` définie par une propriété constante `liste` de type `List<Message>` et initialisée par `listOf()`. C'est une fonction Kotlin qui crée une liste vide. Le type `List<>` crée une liste non modifiable en Kotlin. La différence entre des listes modifiables et des listes non modifiables est expliquée dans cette page .

👉 Ajoutez ces méthodes à la classe `MessageListState` :

```
    /**
     * retourne une copie de this dans laquelle on a rajouté le message
     * @param message : message à rajouter dans la copie de this
     */
```

```
    fun copyAddMessage(message: Message): MessageListState {
        return MessageListState(this.liste + message)
    }

    /**
     * retourne une copie de this dans laquelle on a enlevé le message
     * @param message : message à enlever de la copie de this
     */
    fun copyDeleteMessage(message: Message): MessageListState {
        return MessageListState(this.liste - message)
    }
```

Ces méthodes créent une copie de la liste en lui ajoutant/enlevant le message passé en paramètre. La notation `liste + element` ou `liste1 + liste2` est expliquée dans cette discussion . Il y a une différence entre la méthode `add` qui ne peut s'appliquer qu'à une liste modifiable et la méthode `plus` ou `+` qui recrée une nouvelle instance de liste avec les éléments de l'ancienne et les nouveaux, idem avec `minus` et `-`.

1.2. Contrôleur (Vue-Modèle)

Dans ce projet, c'est la vue qui est la plus complexe, et de loin. Alors on s'occupe d'abord du *ViewModel*. Son rôle est de gérer un singleton de l'état : effectuer toutes les modifications des données en tant que remplacement de cet état par un autre et abonner la vue aux changements.

☛ Ajoutez `MessageListViewModel.kt` dans `fr.iutlan.tp5.messages.controller` :

```
package fr.iutlan.tp5.messages.controller

import androidx.compose.runtime.mutableStateOf
import androidx.lifecycle.ViewModel
import fr.iutlan.tp5.messages.state.*

class MessageListViewModel : ViewModel() {

    // singleton contenant l'état, observable mais privé
    private val _state = mutableStateOf(MessageListState())

    // getter pour consulter cet état à l'extérieur, mais setter privé
    var state
        get() = _state.value            // _state.value = instance de MLState
        private set(newstate) {
            _state.value = newstate     // remplace l'état par le nouveau
        }

    /** constructeur */
    init {
        clearList()
    }
```

```
    /// méthodes pour modifier les données

    /**
     * supprime tous les messages
     */
    fun clearList() {
        state = MessageListState()
    }

    /**
     * ajoute le message à la liste, sauf si le message est vide
     * @param message : celui à ajouter
     */
    fun addNewMessage(message: Message) {
        // TODO
    }
}
```

Remarquez à quel point ce contrôleur ressemble à celui des feux tricolores du TP4.

👉 Complétez la fonction `addNewMessage`. Pour savoir si un message est vide, utilisez `message.texte.isBlank()`.

1.3. Vue

On attaque l'affichage de la liste des messages, ainsi que de quoi créer de nouveaux messages.

👉 Créez un fichier appelé `MessageListView.kt` dans `fr.iutlan.tp5.messages.ui`.

NB: vous avez remarqué qu'en Kotlin, un même fichier source peut contenir différentes choses et pas seulement une classe nommée comme le fichier. Ici, ce fichier va contenir toutes les fonctions composables d'affichage. Dans un plus grand projet, il faudrait évidemment tout séparer.

On va adopter une démarche modulaire, et mettre au point les fonctions composables une par une, en commençant par les plus basiques.

1.3.1. Affichage d'un icône

On commence par la fonction qui fournit l'icône composable correspondant à une valeur du type `enum Icone`.

👉 Complétez `MessageListView.kt` comme ceci :

```
package fr.iutlan.tp5.messages.ui

// TODO imports (ALT-entrée sur tout ce qui est en rouge)

import androidx.compose.material3.*
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.rounded.*
```

```
import fr.iutlan.tp5.messages.state.*

@Composable
fun MessageIcon(icone: Icone, modifier: Modifier = Modifier) {
    Icon(
        imageVector = when (icone) {
            Icone.ETOILE -> Icons.Rounded.Star
            Icone.APPEL  -> Icons.Rounded.Call
            Icone.FAVORI -> Icons.Rounded.Favorite
        },
        tint = Color.Blue,
        contentDescription = null,
        modifier = modifier
    )
}

@Preview(showBackground = true)
@Composable
fun MessageIconPreview() {
    MessageIcon(icone = Icone.APPEL, modifier = Modifier.size(48.dp, 48.dp))
}
```

Attention aux imports. Si vous n'importez pas les bons fichiers, vous allez vous retrouver avec des méthodes inconnues. Par exemple `Modifier` vient de `androidx.compose.ui` pas de `java.lang.reflect` ou autre.

Normalement vous devez voir un combiné téléphonique bleu foncé dans la partie droite de l'éditeur. Vous pourriez avoir des icônes de couleurs différentes, en faisant comme pour l'image c'est à dire un `when` pour définir la propriété `tint` et chaque cas retourne une couleur spécifique.

👉 Remarquez que la fonction `MessageIcon` ne demande qu'un paramètre de type `Icone`. On ne doit pas lui fournir davantage. Dans le patron de conception sous-jacent, on ne doit fournir que le strict minimum d'informations aux fonctions composables.

1.3.2. Affichage d'un message isolé

On va maintenant afficher un message, composé d'un icône et d'un texte.

👉 Ajoutez ceci au dessus de `MessageIcon` et complétez-le : 

```
@Composable
fun MessageView(message: Message, modifier: Modifier = Modifier) {
    // TODO afficher l'icône et le texte du message horizontalement
}

@Preview(showBackground = true)
@Composable
fun MessagePreview() {
    MessageView(message = Message(Icone.APPEL, "à pelle untel"))
}
```

La mise en page ne sera pas élégante, mais on n'est pas là pour ça.

Si au moins vous voulez aligner les éléments verticalement, vous pouvez ajouter ce paramètre à `Row(verticalAlignment = Alignment.CenterVertically)`. C'est expliqué dans le TP4, dans le §1.6 Structure d'une interface.

1.3.3. Affichage d'une liste de messages

Le principe est de fabriquer une colonne avec autant d'exemplaires de `MessageView` que de messages. On pourrait faire cela avec une simple *boucle pour*, mais Compose offre des dispositifs adaptés à cela.

On commence avec une version toute simple.

👉 Ajoutez ceci :

```
import androidx.compose.foundation.lazy.*
import androidx.compose.material3.HorizontalDivider

@Composable
fun MessageListView(messages: List<Message>, modifier: Modifier = Modifier) {
    LazyColumn(modifier) {
        items(messages) {
            MessageView(it)
        }
    }
}

@Preview(showBackground = true)
@Composable
fun MessageListPreview() {
    MessageListView(
        messages = listOf(
            Message(Icone.ETOILE, "eh, toi, le..."),
            Message(Icone.APPEL, "à pelle"),
            Message(Icone.FAVORI, "ça vaut riz")
        ),
        modifier = Modifier.fillMaxWidth().height(150.dp)
    )
}
```

On aurait pu utiliser `Column` avec une boucle comme dans `AccueilMultiple` du TP4, mais la fonction `LazyColumn` est bien plus intéressante. Elle gère le défilement vertical et n'affiche que les données visibles, en fonction du défilement demandé par l'utilisateur, d'où le *lazy* dans son nom. Elle demande une méthode `items` pour savoir ce qu'elle doit afficher. C'est une sorte d'itérateur et dans le bloc, l'élément courant est désigné par `it`.

1.3.4. Amélioration de la liste

La liste pourrait être plus belle avec des lignes de séparation entre les éléments.

☛ Modifiez le cœur en ceci :

```
    LazyColumn(modifier) {
        itemsIndexed(
                messages,
                key = { _,msg -> msg.id }
        ) { index, message ->
            // séparateur entre éléments
            if (index > 0) HorizontalDivider(color = Color.LightGray)
            // élément
            MessageView(message)
        }
    }
```

`itemsIndexed` parcourt la liste en produisant des couples (numéro d'élément, élément). Il y a deux points à remarquer.

- Le paramètre `key` de `itemsIndexed` est une lambda qui retourne une sorte d'identifiant, un nombre différent dépendant du message. C'est nécessaire quand la liste change de contenu, pour que `itemsIndexed` s'aperçoive des changements. On aurait dû faire la même chose avec `items` plus haut, fournir une clé d'identification des messages. Et si on voulait faire encore mieux, on devrait rajouter un 3e champ aux messages, leur identifiant et l'utiliser comme clé.
- Au passage, le premier paramètre de cette lambda est « _ ». C'est comme ça qu'on écrit un paramètre dont on ne se sert pas.
- Le contenu de `itemsIndexed` est également une lambda à deux paramètres, `index` et `message` qui sont affectés automatiquement par `itemsIndexed`. On se sert de l'index pour ne dessiner un séparateur qu'à partir du deuxième message.

1.3.5. Cas d'une liste vide

La vue pourrait être plus informative en indiquant à l'utilisateur quand la liste est vide.

☛ Modifiez la fonction comme ceci :

```
@Composable
fun MessageListView(messages: List<Message>, modifier: Modifier = Modifier) {
    if (messages.isEmpty()) {
        Column(
            modifier.fillMaxWidth(),
            verticalArrangement = Arrangement.Center,
            horizontalAlignment = Alignment.CenterHorizontally
        ) {
            Text(text = "aucun message")
        }
    } else {
        LazyColumn(modifier) {
            ...
        }
    }
```

```
}

@Preview(showBackground = true)
@Composable
fun MessageListEmptyPreview() {
    MessageListView(
        messages = listOf(),
        modifier = Modifier.fillMaxWidth().height(150.dp)
    )
}
```

La fonction `Column` sert à centrer le texte. La fonction `Text` ne possède aucun paramètre pour faire ce travail.

Il y a maintenant deux fonctions de prévisualisation, une avec quelques éléments, l'autre pour une liste vide.

Vous voyez la puissance qu'il y a à pouvoir créer une interface par programmation, grâce au fait que les composants visuels sont créés par des instructions, des appels de fonctions. On peut employer des boucles, des alternatives, faire des calculs... choses qui sont impossibles avec un *layout* XML (on peut le faire partiellement, en gérant manuellement la visibilité des éléments).

1.3.6. Fonction principale

Il ne manque plus que la fonction principale : 

```
@Composable
fun MainActivityMessageListView(
    modifier: Modifier = Modifier,
    viewmodel: MessageListViewModel = viewModel(),
) {
    // état auquel s'abonne cette fonction composable
    val state = viewmodel.state

    // ICI n°1, voir plus loin

    Column(
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        // afficher la liste des messages
        MessageListView(state.liste, modifier.fillMaxSize().weight(1f))

        // afficher un bouton flottant "nouveau" pour créer un message
        FloatingActionButton(
            onClick = { /* ICI n°2, voir plus loin */ }
        ) {
            Icon(Icons.Filled.Add, contentDescription = "Nouveau")
        }
```

```
    }
}

@Preview(showBackground = true)
@Composable
fun MainActivityMessageListPreview() {
    MainActivityMessageListView()
}
```

Voyez comment le bouton est créé. C'est un bouton flottant mis à la mode par Material 3.

Vous pouvez remarquer un poids appliqué à la liste par `weight(1f)`. C'est pour la rendre aussi grande que possible mais en laissant de la place pour le bouton. Le bouton n'a pas de poids. Ce n'est pas comme dans les `LinearLayout` de Android Views. Ce document extrêmement bien fait explique le concept (au milieu du document) : les poids définissent le partage de l'espace libre restant, après avoir réparti l'espace total entre toutes les vues. Ici, le bouton, n'ayant pas de poids, prend la place dont il a besoin et la liste, avec son poids de 1, récupère tout ce qui reste.

👉 Mettez en place cette interface dans `MainActivity` en supprimant les fonctions `Greeting*` et testez l'application sur un AVD.

Il y a deux emplacements signalés par « `ICI n°1 et 2` » qui seront complétés peu à peu.

2. Création d'un nouveau message

On voudrait pouvoir ajouter un message. On va le faire, non pas par une nouvelle activité, mais avec un dialogue modal. C'est une fenêtre qui vient se superposer à l'activité le temps de la saisie du message.

Figure 1: Dialogue

Pour obtenir ce résultat :

- Il faut créer une fonction composable qui dessine un dialogue pour saisir un message.
- Il faut afficher ce dialogue quand l'utilisateur clique sur le bouton nouveau. C'est le rôle de la lambda `ICI n°2` à compléter plus loin.
- Il faut enlever ce dialogue quand l'utilisateur valide ou annule le dialogue, et il faut enregistrer le message dans la liste.

On va construire ce dialogue peu à peu, de façon modulaire, également en partant des vues élémentaires. Il y a quelques points très subtils à comprendre, alors prenez votre temps.

👉 Créez un fichier `DialogMessage.kt` dans le package `messages.ui`.

2.1. Panneau de boutons OK et Annuler

En bas du dialogue, il y a deux boutons, OK et Annuler. On les met dans un panneau composable qu'on pourra mettre au point séparément. Et ce panneau pourra être réutilisé dans un autre projet.

☛ Ajoutez cette fonction composable dans `DialogMessage.kt` : 

```
@Composable
fun PanneauBoutonsOKAnnuler(
    onAccept: () -> Unit,       // action si valider le dialogue
    onCancel: () -> Unit,       // action si clic sur annuler
    modifier: Modifier = Modifier
) {
    Row(
        modifier = modifier.fillMaxWidth(),
        horizontalArrangement = Arrangement.Center,
    ) {
        TextButton(
            onClick = onAccept,
            modifier = Modifier.padding(8.dp),
        ) {
            Text(stringResource(id = android.R.string.ok))
        }
        TextButton(
            onClick = onCancel,
            modifier = Modifier.padding(8.dp),
        ) {
            Text(stringResource(id = android.R.string.cancel))
        }
    }
}

@Preview(showBackground = true)
@Composable
fun PanneauBoutonsOKAnnulerPreview() {
    PanneauBoutonsOKAnnuler(
        onCancel = {},
        onAccept = {}
    )
}
```

Cette fonction crée une ligne de deux boutons, des `TextButton`. Vous pouvez remarquer que le texte des boutons est défini par des ressources prédéfinies, `android.R.string.ok`. Ces ressources sont automatiquement traduites dans la langue du téléphone.

Le plus nouveau, ce sont les paramètres de `PanneauBoutonsOKAnnuler`. Jusqu'ici, c'étaient des données, icône, message, liste. Cette fois, ce sont deux lambda. La première sera appelée si l'utilisateur clique sur OK, l'autre s'il clique sur Annuler. Voyez comment c'est fait avec les paramètres `onClick` des boutons. On verra plus loin comment on définit ces lambdas lors de

l'appel à `PanneauBoutonsOKAnnuler`. Dans la prévisualisation, elles sont définies à un bloc vide.

Il y a un nouveau patron de conception dans cette fonction. Vous savez déjà qu'une fonction composable est appelée avec des paramètres issus de l'état (du modèle) qui spécifient ce qu'elle doit dessiner, et ces paramètres contiennent le strict minimum d'informations nécessaires. En retour, cette fonction composable peut avoir un effet sur celle l'appelle, grâce à des lambda. Les explications sont sur cette page et voici le schéma qui résume le patron.

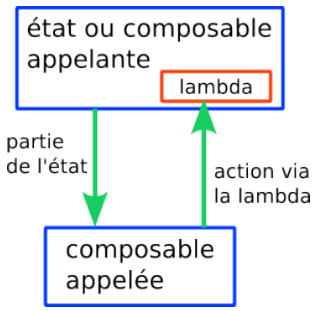

Figure 2: Flux unidirectionnel

- Une fonction composable parente peut passer en paramètre un sous-ensemble de l'état et/ou des lambda destinée à changer quelque chose chez elle.
- La fonction composable enfant affiche l'état qu'elle reçoit en paramètre et peut appeler la lambda. Cette lambda s'exécute dans l'environnement de la fonction parente.

Ce patron s'appelle « Flux de données unidirectionnel ». Les états doivent uniquement descendre dans les sous-fonctions et les exécutions de lambda modifiant l'état doivent remonter vers le contrôleur en passant par toutes les fonctions composables intermédiaires.

Dans `PanneauBoutonsOKAnnuler`, il n'y a pas d'état, mais seulement deux lambda fournies par le dialogue. `onCancel` sera appelée pour fermer le dialogue, et `onAccept` enregistrera le nouveau message. C'est dans le dialogue qu'il y aura le réel travail des boutons ok et annuler.

Cette fonction est modulaire et réutilisable, parce qu'elle ne contient rien de spécifique à notre application. On pourrait construire une librairie de fonctions composables réutilisables comme celle-ci. Il suffirait de sortir cette fonction dans un autre fichier.

2.2. Menu déroulant de choix d'icône

On a besoin d'un mécanisme pour choisir l'icône associé au texte dans le message. On va le faire avec un menu déroulant. La structure est un peu compliquée et doit être créée par programme.

Figure 3: Menu déroulant

👉 Ajoutez cette fonction composable :

```
@Composable
fun ChoixIconeView(
    icone: Icone,
    onSelectIcone : (Icone) -> Unit,
    modifier: Modifier = Modifier
) {
    // menu dropdown visible ou non
    var isDropdownVisible = false     // FIXME voir les explications

    IconButton(onClick = { isDropdownVisible = true }) {
        MessageIcon(icone = icone)
    }
    DropdownMenu(
        expanded = isDropdownVisible,
        onDismissRequest = { isDropdownVisible = false },
    ) {
        Icone.entries.forEach {
            DropdownMenuItem(
                text = { it.toString() },
                onClick = { onSelectIcone(it) ; isDropdownVisible = false },
                leadingIcon = {
                    MessageIcon(
                        it,
                        modifier = modifier.padding(4.dp)
                    )
                }
            )
        }
    }
```

```
}
```

Cette fonction crée cette structure, figure 4 :

- un bouton, ici un `IconButton` pour afficher ou masquer le menu,
- un `DropdownMenu` qui regroupe les éléments,
  - un `DropdownMenuItem` pour chaque élément du menu.

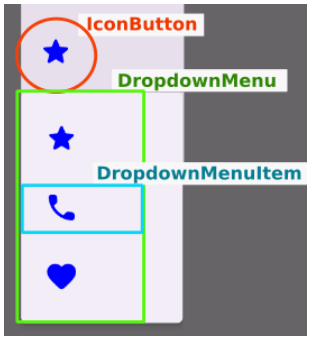

Figure 4: Menu déroulant

Le bouton `IconButton` affiche l'icône courant. La lambda `onSelectIcone` sera appelée quand l'un des icônes sera cliqué. On devine que le programme appelant va mémoriser cet icône dans le futur message. On verra comment c'est fait dans la prochaine section.

Il y a une nouveauté. C'est la variable `isDropdownVisible`. Elle commande l'affichage du menu déroulant.

👉 Regardez les 4 endroits où elle est utilisée. Qu'en pensez-vous ? Attendez avant de répondre. Ce serait bien si ça marchait parfaitement.

Cette fonction n'a pas de prévisualisation à cause d'un bug persistant dans Android Studio. Le menu n'apparaît pas, et il est décalé. Mais de toutes façons, on ne peut rien tester en mode prévisualisation. Alors faites ce qui suit.

👉 Dans `MainActivity`, là où il y a `setContent`, commentez l'appel existant et mettez ceci, avec les bons imports :

```
        setContent {
            ChoixIconeView(
                icone = Icone.APPEL,
                onSelectIcone = { println("Icone choisi : $it") }
            )
        }
```

👉 Supprimez la ligne `enableEdgeToEdge()` qui ne sert à rien pour nous.

Constatez qu'il ne se passe rien quand on clique sur le bouton, le menu n'apparaît pas. Pourtant l'écouteur du bouton, la lambda `onClick` est exécutée et vous pourriez ajouter un `println` dans `{ isDropdownVisible = true ; println(...) }` pour le prouver.

👉 Changez la variable en `var isDropdownVisible = true` et refaites le test. Cette fois, le menu est déroulé, et quand vous cliquez sur un item, on voit un message dans le *LogCat*, preuve

que l'écouteur `onSelectIcone` est bien exécuté. Par contre, le menu ne disparaît pas. On a l'impression que `isDropdownVisible` n'est jamais modifiée et pourtant il y a bien les affectations.

Le problème est plus profond qu'une simple variable qui ne change pas de valeur. C'est une conséquence du patron de conception des fonctions composables. Ces fonctions ne sont exécutées qu'une seule fois. La fonction `ChoixIconeView` s'est exécutée, et a affiché le menu ou pas, et c'est tout. Donc quand on modifie la variable `isDropdownVisible` dans un écouteur, ça n'a aucun effet visible, car il faudrait relancer la fonction `ChoixIconeView`, mais c'est pas comme ça que ça marche.

Rappelez-vous que les fonctions composables affichent des données totalement inertes. Le seul moment où l'interface peut changer, c'est quand on change les données. Donc il faut que `isDropdownVisible` soit considérée comme un état. Alors faut-il placer cette variable dans la classe `MessageListState` ?

On peut le faire, mais ce n'est pas pertinent. Il faudrait le propager dans toutes les fonctions composables qui amènent à `ChoixIconeView`, et avec ça, il faudrait faire suivre une lambda partout pour modifier cette variable, la faire passer de false à true et inversement.

La solution est plus générale. En fait, il faut distinguer deux sortes d'états. Il y a l'état de l'application comme la liste des messages, et il y a l'état de l'interface comme le fait que le menu soit déroulé ou caché, ou le dialogue de saisie visible ou non. L'état de l'application est destiné à durer même quand l'application est arrêtée (ici ce n'est pas encore le cas, voir la suite du TP). L'état de l'interface est relativement temporaire. Il existe tant que l'activité est sur l'écran.

Android Compose offre un mécanisme particulier pour gérer l'état de l'interface. Ce sont des variables spéciales qui sont stockées avec les composables. On les définit par :

```
var nom by rememberSaveable { mutableStateOf(valeur initiale) }
```

Voici un exemple (pour info, il faut l'adapter à la situation) :

```
import androidx.compose.runtime.getValue
import androidx.compose.runtime.setValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.saveable.rememberSaveable

@Composable
fun Exemple(...) {

    // variables d'état de l'interface
    var isClicked by rememberSaveable { mutableStateOf(false) }

    // bouton pour changer isClicked
    Button(
        onClick = { isClicked = true },
    ) ...
}
```

La variable est un `MutableState` comme l'état de l'application dans `MessageListViewModel`, donc tous ceux qui lisent sa valeur se retrouvent abonnés à ses changements. La syntaxe Kotlin

`by rememberSaveable {...}` permet d'alléger sa manipulation et aussi de l'enregistrer pour redessiner la vue si elle change, à cause d'une bascule portrait/paysage de l'écran et autres événements similaires.

👉 Transformez la variable d'état dans `ChoixIconeView` en variable d'état avec mémoire (attention, les imports ne sont pas proposés, mais sont indispensables). Testez sur l'AVD. Ça doit marcher, le menu doit apparaître, disparaître quand on choisit un élément, et il doit rester quand on bascule en mode paysage.

Les fonctions composables qui possèdent un état représenté par des variables `rememberSaveable` seraient qualifiées de *stateful widgets* dans Flutter, et celles qui n'ont pas ce genre de variables sont des *stateless widgets*.

2.3. Dialogue de saisie d'un message

On retrouve les concepts précédents pour également stocker le texte en cours d'édition.

👉 Ajoutez cette fonction : 

```
@Composable
fun MessageDialog(
    onAccept: (Icone, String) -> Unit,
    onCancel: () -> Unit,
    modifier: Modifier = Modifier
) {
    // valeurs en cours d'édition
    var icone = Icone.ETOILE        // FIXME
    var texte = ""                  // FIXME

    Dialog(onDismissRequest = onCancel) {
        Card(modifier = modifier.fillMaxWidth().padding(16.dp),
            shape = RoundedCornerShape(16.dp),
        ) {
            // titre
            Row(modifier = modifier.fillMaxWidth().padding(16.dp),
                horizontalArrangement = Arrangement.Center) {
                Text("Nouveau message")
            }
            Row(modifier = Modifier.fillMaxWidth(),
                verticalAlignment = Alignment.CenterVertically,
            ) {
                // choix de l'icône
                ChoixIconeView(
                    icone = icone,
                    onSelectIcone = { icone = it }
                )
                // texte du message
                TextField(
                    value = texte,
                    onValueChange = { texte = it },
```

```
                    label = { Text("message") }
                )
            }

            // boutons
            PanneauBoutonsOKAnnuler(
                onAccept = { onAccept(icone, texte) },
                onCancel = onCancel
            )
        }
    }
}

@Preview(showBackground = true)
@Composable
fun MessageDialogPreview() {
    MessageDialog(
        onCancel = {},
        onAccept = { _,_ -> }
    )
}
```

☛ Réparez les FIXME.

☛ Pour tester sur AVD, mettez ceci dans `MainActivity.onCreate`, là où il y a `setContent` (commentez l'existant) :

```
        setContent {
            MessageDialog(
                onCancel = {},
                onAccept = { ico,txt -> println("$ico : $txt") }
            )
        }
```

☛ Vérifiez que le dialogue fonctionne bien. Le bouton OK doit entraîner l'affichage du message dans le *LogCat* (à cause du `println` de la lambda). Par contre, c'est normal que le dialogue ne disparaisse pas.

2.4. Fonction principale revisitée

Il reste à terminer la fonction `MainActivityMessageListView`.

☛ Remplacez les commentaires `ICI n°1 et 2` par tout ce qu'il faut pour afficher et masquer le dialogue de saisie d'un message :

- `ICI n°1` sert à afficher ou pas un `MessageDialog`. C'est à dire qu'on appelle ou pas la fonction composable `MessageDialog` et c'est décidé par un booléen à définir, par exemple `isMessageDialogVisible`, initialisé à faux. N'oubliez aucun des 4 imports nécessaires, ils ne sont pas proposés par Android Studio.

- L'appel à `MessageDialog` doit être placé dans une conditionnelle commandée par ce booléen d'état d'interface.
- La lambda `onAccept` fournie à `MessageDialog` doit enregistrer un nouveau message. Attention, elle ne doit pas tenter de le faire en manipulant `state`, mais en appelant la bonne méthode de `viewmodel`.
- Il reste à programmer l'action `ICI n°2`. Il faut faire afficher le dialogue.

👉 Testez votre application. Pensez à pivoter l'écran pour voir si l'état persiste.

Prenez un peu de recul. Remarquez que l'écouteur `onAccept` qui est activé dans le panneau des boutons fait remonter fonction par fonction un événement, sous la forme de l'exécution en cascade de lambdas fournies par les fonctions composables utilisées.

Ce projet pourrait être continué, par exemple en ajoutant la possibilité d'éditer un message, ou en mémorisant les messages dans une base Realm, mais c'est beaucoup trop complexe pour ce premier TP sur Compose.

3. Édition d'un message

On continue le projet avec l'édition d'un message existant. On commence par ajouter un petit crayon dans la fonction composable `MessageView`. Ça sera un bouton à cliquer qui fera apparaître le dialogue d'édition du message.

Le problème, c'est qu'on n'a pas pensé à pouvoir ouvrir le dialogue avec un message existant. On va devoir modifier la fonction `MessageDialog` et la faire appeler dans un message. On commence par le dialogue, c'est le plus facile.

3.1. Modification du dialogue

Le but est de rajouter un paramètre `message`, optionnel, qui initialise le texte et l'icône s'il est fourni.

👉 Reprenez le source de `MessageDialog` et modifiez ainsi :

```
@Composable
fun MessageDialog(
    onAccept: (Icone, String) -> Unit,
    onCancel: () -> Unit,
    modifier: Modifier = Modifier,
    message: Message? = null
) {
    // valeurs en cours d'édition
    var icone ...
    var texte ...
```

Le paramètre message valant `null` s'il n'est pas fourni doit maintenant initialiser les deux variables. Actuellement, il y a des valeurs par défaut, par exemple `""` pour le texte. Voici la syntaxe à utiliser pour mettre les champs du message s'il est fourni, sinon les valeurs d'avant :

```
message?.icone ?: Icone.ETOILE
message?.texte ?: ""
```

Attention, c'est à mettre à la place des valeurs par défaut actuelles, en laissant la structure d'affectation que vous aviez déjà.

Cette notation `variable?.champ ?: défaut` est appelé *opérateur Elvis* (doc). C'est une syntaxe Kotlin pour éviter les exceptions d'accès à `null`. C'est la simplification d'une conditionnelle.

☛ Ajoutez cette fonction de prévisualisation :

```
@Preview(showBackground = true)
@Composable
fun MessageDialogPreview2() {
    MessageDialog(
        message = Message(Icone.APPEL, "à pelle"),
        onCancel = {},
        onAccept = { _,_ -> }
    )
}
```

3.2. Édition d'un message existant

On doit maintenant pouvoir ré-afficher le dialogue d'édition sur n'importe quel message existant. La technique consiste à ajouter un petit bouton en bout de chaque ligne. Son écouteur rend le dialogue visible.

Figure 5: Boutons pour éditer

Attention, ici on parle d'un dialogue qui apparaît dans la fonction `MessageView`, donc en plus de celui qui apparaît dans `MainActivityMessageListView`.

☛ Complétez votre fonction `MessageView` :

```
@Composable
fun MessageView(message: Message, modifier: Modifier = Modifier) {

    // ICI il y aura de quoi afficher le dialogue

    Row(...) {
        MessageIcon...
        Text(..., modifier = Modifier.weight(1f))

        // bouton pour modifier le message via le dialogue
        IconButton(
            onClick = { /* afficher le dialogue */ },
            modifier = Modifier.size(16.dp, 16.dp)
```

```
        ) {
            Icon(
                imageVector = Icons.Rounded.Edit,
                tint=Color.Gray,
                contentDescription = null)
        }
    }
}
```

Notez le poids de 1 pour allonger le texte au maximum, mais sans écraser le bouton final.

👉 Devinez ce que doit faire le bloc `onClick` du bouton : afficher le dialogue de saisie d'un message. Regardez comment c'est fait dans `MainActivityMessageListView` et reprenez le même principe, un booléen initialement faux, devenant vrai quand on clique sur l'icône crayon.

👉 Fournissez le message au dialogue et mettez des lambda permettant de fermer le dialogue pour `onAccept` et `onCancel`.

Normalement, le dialogue apparaît, avec le bon icône et le bon texte, par contre, aucune modification n'a d'effet. On va y travailler, mais ce n'est pas simple. Dans `MainActivityMessageListView`, la lambda fournie par `onAccept` au dialogue demande directement la création au `ViewModel`. Or dans `MessageView`, on ne dispose pas du `ViewModel`.

Le patron de conception *flux unidirectionnel* indique comment faire : il faut que `MessageView` reçoive un paramètre de plus, une lambda qu'elle appellera quand son message sera modifié.

👉 Modifiez l'entête de la fonction `MessageView` ainsi : 

```
@Composable
fun MessageView(
        message: Message,
        onEditMessage: (Message, Icone, String) -> Unit,
        modifier: Modifier = Modifier) {
    ...
```

Elle demande un paramètre nommé `onEditMessage`. C'est une lambda à trois paramètres, le message actuel et ses nouvelles valeurs.

👉 Faites en sorte que lorsqu'on valide le dialogue dans la fonction, ça appelle cette lambda. Déjà, la lambda que vous passez au dialogue, dans `onAccept` doit avoir deux paramètres, `ico` et `txt` qui sont affectés au retour du dialogue. Il faut juste les transmettre à `onEditMessage`, avec le message en plus.

Ensuite, on doit remonter la chaîne d'appels de `MessageView` et fournir la lambda `onEditMessage`.

Cette lambda est définie tout en haut, dans `MainActivityMessageListView` où vous aurez un appel comme ceci : 

```
    ...
        MessageListView(
            state.liste,
            onEditMessage = {
                message, newico, newtxt ->
                    viewmodel.editMessage(message, newico, newtxt)},
```

```
            modifier = modifier.fillMaxSize().weight(1f),)
    ...
```

👉 Complétez les fonctions composables concernées. Elles doivent transmettre `onEditMessage` jusqu'à `MessageView`.

👉 Vous devrez rajouter une lambda vide `{_,_,_->}`, aux fonctions de prévisualisation concernées.

Il reste à définir la méthode `editMessage` dans la classe `MessageListViewModel`. Le plus simple est de copier l'état en supprimant le message et ensuite en rajoutant un nouveau message avec les valeurs fournies. L'ordre des messages sera perdu, mais c'est le plus simple.

On peut améliorer en recherchant l'indice du message dans la liste, puis en le supprimant puis en insérant un nouveau à l'emplacement de l'ancien. Ça n'est pas demandé, mais ce serait souhaitable.

4. Suppression d'un message

On voudrait pouvoir supprimer un message simplement en le faisant glisser vers la gauche ou la droite. On appelle cette action « Swipe to Dismiss ». Alors malheureusement, Jetpack Compose est encore très jeune et ce mécanisme n'est pas standard. On doit faire appel à des fonctionalités expérimentales et en plus, on n'a pas tellement le choix.

👉 Ajoutez tout ceci entre la définition de `MessageListView` et de `MessageView` :

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun DismissibleMessageView(
    message: Message,
    onEditMessage: (Message, Icone, String) -> Unit,
    onDeleteMessage: (Message) -> Unit,
    modifier: Modifier = Modifier
) {
    val dismissState = rememberSwipeToDismissBoxState(
        confirmValueChange = {
            return@rememberSwipeToDismissBoxState when (it) {
                SwipeToDismissBoxValue.StartToEnd -> {
                    onDeleteMessage(message)
                    true
                }
                SwipeToDismissBoxValue.EndToStart -> {
                    onDeleteMessage(message)
                    true
                }
                SwipeToDismissBoxValue.Settled -> false
            }
        },
        // pourcentage de déplacement latéral pour valider l'action
        positionalThreshold = { it * .25f }
    )
    SwipeToDismissBox(
```

```
        state = dismissState,
        modifier = modifier.fillMaxSize(),
        backgroundContent = { DismissBackground(dismissState)}
    ) {
        MessageView(message, onEditMessage = onEditMessage)
    }
}

@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun DismissBackground(dismissState: SwipeToDismissBoxState) {
    val alpha = Math.min(dismissState.progress / .25f, 1.0f)
    val color by animateColorAsState(
        when (dismissState.dismissDirection) {
            SwipeToDismissBoxValue.StartToEnd -> Color(0.4f, 0.4f, 0.4f, alpha)
            SwipeToDismissBoxValue.EndToStart -> Color(0.4f, 0.4f, 0.4f, alpha)
            SwipeToDismissBoxValue.Settled -> Color.Transparent
        }
    )
    Box(Modifier.fillMaxSize().background(color))
}
```

☛ Remplacez l'appel à `MessageView` par `DismissibleMessageView` dans `MessageListView`. Vous allez devoir ajouter un paramètre lambda `onDeleteMessage` et le fournir depuis `MainActivityMessageListView`.

☛ Testez l'application.

NB: il y a beaucoup à améliorer, par exemple faire afficher un icône poubelle quand on commence à faire glisser un élément, mais c'est trop difficile à faire.

5. Travail à rendre

Important : votre projet doit se compiler et se lancer sur un AVD. La note du TP sera zéro si ce n'est pas le cas. Mettez donc en commentaire ce qui ne compile pas. C'est impératif que l'application puisse être lancée même si elle ne fait pas tout ce qui est demandé.

Avec le navigateur de fichiers, ouvrez `~/AndroidStudioProjects/TP5` (comme celui de la première semaine), puis descendez successivement dans `app` puis `src`. Vous devez y voir le dossier `main` ainsi que `test` et `AndroidTest`. Cliquez droit sur le dossier `main` et choisissez `Créer une archive...` puis `.tar.gz` ou `.zip`. Ça doit créer `main.tar.gz` ou `main.zip` contenant tout votre travail.

Rajoutez un fichier appelé exactement `IMPORTANT.txt` dans le sous-dossier `main` si vous avez rencontré des problèmes techniques durant le TP : plantages, erreurs inexplicables, perte du travail, etc. mais pas les problèmes dus à un manque de travail ou de compréhension. Décrivez exactement ce qui s'est passé. Le correcteur pourra lire ce fichier au moment de la notation et compenser votre note. Mettez au moins ce fichier si vous n'avez rien pu faire du tout pendant la séance.

Déposer l'archive dans le dépôt Moodle R4.A11 Développement Mobile , rendu TP5.