L'objectif du TP est d'apprendre à programmer une application Android basée sur Jetpack Compose. C'est la nouvelle API Android recommandée par Google.

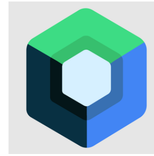


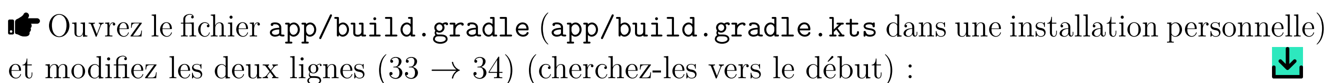
Figure 1: Logo Jetpack Compose

1. Découverte de Compose

1.1. Création du projet

☛ Pour commencer, créez un projet Android, de type « Empty Activity » avec le logo Compose, appelé `TP9` (ni `tp9`, ni `Tp9`). Le package doit être `fr.iutlan.tp9`.

☛ Ouvrez le fichier `app/build.gradle` (`app/build.gradle.kts` dans une installation personnelle)



et modifiez les deux lignes (33 → 34) (cherchez-les vers le début) :

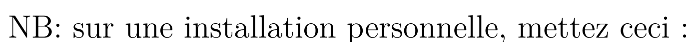
```
compileSdk 34
targetSdk 34
```

☛ Modifiez/ajoutez le numéro de version de ces deux dépendances :

```
implementation 'androidx.core:core-ktx:1.12.0'
implementation 'androidx.compose.material3:material3:1.2.1'
```

☛ Ajoutez ces dépendances :

```
implementation 'androidx.lifecycle:lifecycle-viewmodel-compose:2.5.1'
implementation 'androidx.compose.ui:ui-tooling:1.6.3'
```



NB: sur une installation personnelle, mettez ceci :

```
implementation("androidx.lifecycle:lifecycle-viewmodel-compose:2.7.0")
```

Cette ligne sera transformée automatiquement en
`implementation(libs.androidx.lifecycle.viewmodel.compose)`

☛ Lancez l'exécution sur un AVD. Vous devez voir un message de salutations.

On va découvrir ce que l'assistant a construit, et faire des modifications pour comprendre les concepts de Compose.

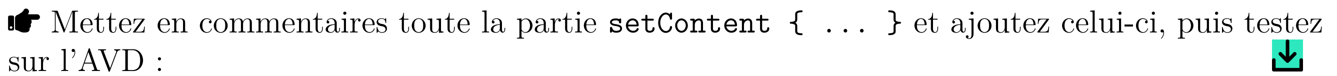
1.2. Structure d'une activité

Une application Compose contient des activités dérivées de `ComponentActivity` :

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

```
            //... création de l'interface ...
        }
    }
}
```

L'interface est mise en place par `setContent { ... }`, pas `setContentView(layout)`.


Mettez en commentaires toute la partie `setContent { ... }` et ajoutez celui-ci, puis testez sur l'AVD :

```
        setContent {
            Text("Bonjour tout le monde !")
        }
```

Explications

La syntaxe Kotlin `fonction { instructions }` est un appel à une fonction dont le dernier paramètre est une lambda et que cette lambda n'a qu'un seul paramètre ou aucun. Dans le cas où elle a un paramètre, alors il s'appelle `it` dans les instructions. Dans les deux cas, on écrit uniquement son corps. C'est le cas avec `setContent`.

Donc ce qu'il y a dans le bloc `{...}` après `setContent`, ce sont des instructions, et ici, c'est un appel à la fonction `Text("...")`. C'est vraiment un appel de fonction, car `Text` est une fonction, contrairement aux conventions de nommage. Normalement, les noms des fonctions et méthodes doivent commencer par une lettre minuscule. Les initiales en majuscules sont réservées aux classes. Mais pas dans l'API Compose. Les éléments d'interface (textes, boutons, panneaux. . .) sont définis par des fonctions et non pas des classes.

Une des conséquences, c'est que la fonction `Text` effectue un traitement, comme si on appelait `println`. Ce n'est pas du tout une création d'objet. Ça veut dire que cette fonction sera à rappeler si on veut changer le message affiché. Il n'y a pas de *setter* pour changer le message, parce que ce n'est pas un objet. C'est essentiel de comprendre ça. Nous verrons comment faire avec.

Ainsi, une interface d'activité est constituée de composants visuels construits par des fonctions telles que la fonction `Text`. L'énorme différence avec Android Views, c'est que dans Compose, ce sont des appels à des fonctions qui créent l'interface, et non pas l'expansion (*inflate*) d'un fichier XML (*layout*) qui produit des objets, comme par exemple un *view binding*. Dans Compose, il n'y absolument aucune notion de *view binding* ni d'objet Java associé à un composant visuel. Ce n'est pas parce que c'est en Kotlin, mais parce que c'est un patron de conception totalement différent.

Les fonctions qui créent des composants visuels sont qualifiées de *composables*. Ce TP va vous permettre de comprendre ce concept et de l'utiliser correctement.

NB: c'est la première édition de ce TP, alors il y a sûrement des erreurs.

1.3. Paramétrage de l'apparence

Ces fonctions possèdent en général de nombreux paramètres pour modifier leur apparence. Kotlin permet de définir des valeurs par défaut aux paramètres des fonctions et méthodes, ce qui fait qu'on peut appeler une fonction avec un nombre variable d'arguments. Par exemple, `Text` n'a qu'un seul paramètre obligatoire. Il s'appelle `text`. Les autres paramètres ont des valeurs par défaut.

En général, les fonctions composables ont un assez grand nombre de paramètres, par exemple `Text` possède 17 paramètres (`color`, `fontSize`, `letterSpacing`...) et donc il est recommandé/indispensable de nommer les paramètres lors de l'appel. Ça consiste à appeler la fonction par `fonction(param1=valeur1, param2=valeur2, etc.)`. Et les paramètres nommés peuvent être fournis dans n'importe quel ordre. Les paramètres non fournis ont forcément une valeur par défaut.

☛ Modifiez `MainActivity` comme ceci :

```
        setContent {
            Text(
                text = "Bonjour tout le monde !",
                fontWeight = FontWeight.Bold,
                fontSize = 32.sp,
                color = Color.Magenta
            )
        }
```

Important tous les `imports` devront concerner les *packages* `androidx.compose` Par exemple, pour la classe `Color`, c'est `import androidx.compose.ui.graphics.Color`, parce que c'est pas celle du TP7.

Si vous vous trompez d'import, ça va mettre les noms de fonction en rouge, disant que la fonction ayant ces paramètres est inconnue. Dans ce cas, il faut penser à supprimer les `imports` concernés, ils ne sont pas en rouge, mais ils provoquent l'erreur, et à refaire l'importation correctement.

1.4. Fonctions composables personnelles

Pour la modularité de l'application, on ne place pas tout dans la méthode `onCreate`, on construit ses propres fonctions composables.

☛ Ajoutez ceci tout à la fin du fichier, **après** la classe `MainActivity` :

```
@Composable
fun Accueil(name: String) {
    Text(text = "Bonjour $name", fontSize=20.sp)
}
```

Important Les fonctions composables sont placées à part, en dehors de la classe d'activité. Dans la deuxième partie du TP, on les mettra carrément dans un autre fichier. En Kotlin, on peut placer différentes fonctions et différentes classes dans le même fichier. En Java, c'est impossible.

La fonction `Accueil` est annotée par `@Composable`. Cela veut dire qu'elle peut être appelée pour construire une interface.

☛ Modifiez `MainActivity` :

```
        setContent {
            Accueil(name = "numéro 6")
        }
```

L'activité appelle `Accueil` pour construire l'interface. Il n'est pas indispensable de nommer le paramètre car il est seul, mais c'est une habitude à prendre.

1.5. Prévisualisation

☛ Ajoutez cette fonction juste après `Accueil` :

```
@Preview
@Composable
fun AccueilPreview() {
    Accueil(name = "numéro 10")
}
```

☛ Mettez l'éditeur en mode `Split`. L'écran se partage en deux, le source à gauche et le résultat visuel à droite. C'est grâce à l'annotation `@Preview`.

Cette fonction est destinée uniquement à Android Studio. Elle ne sera pas placée dans le *apk* final.

Malheureusement, Android Studio n'offre encore aucun atelier pour construire une interface de manière interactive. On est plus ou moins obligé de tâtonner pour ajuster la mise en page. Et en plus, certaines prévisualisations plantent.

1.6. Structure d'une interface

☛ On veut afficher deux textes, essayez ceci :

```
@Composable
fun Accueil(name: String) {
    Text(text = "Bonjour $name", fontSize=20.sp)
    Text(text = "Je vois de grands progrès", color = Color.Green)
}
```

NB: il n'y a pas de virgule ou autre entre les `Text(..)`, parce que ce sont des appels de fonctions, comme deux `printf` successifs.

Compose propose des fonctions pour positionner plusieurs vues. Il n'y a pas `LinearLayout` ni `ConstraintLayout`, mais il y a l'équivalent, voir ces explications. Il faut en connaître deux, `Column` et `Row`.

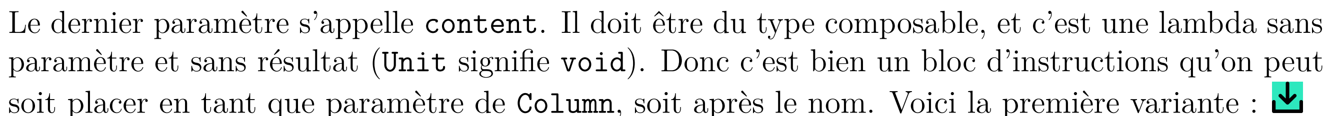
☛ `Column` est un `LinearLayout` vertical :

```
@Composable
fun Accueil(name: String) {
    Column {
        Text(text = "Bonjour $name", fontSize=20.sp)
        Text(text = "Je vois de grands progrès", color = Color.Green)
    }
}
```

☛ Essayez avec `Row`.

Une chose doit vous étonner. L'écriture `Text` est un appel de fonction, `Column` également, malgré l'absence de parenthèses. L'écriture `Column { instructions }` est un appel de la fonction `Column` sans paramètres et cette fonction appelle elle-même `Text` en cascade. En fait, le bloc `{...}` est une lambda sans paramètre, et cette lambda est le dernier paramètre de la fonction `Column`. Voici le début de sa définition (javadoc) :

```
@Composable
inline fun Column(
    modifier: Modifier = Modifier,
    verticalArrangement: Arrangement.Vertical = Arrangement.Top,
    horizontalAlignment: Alignment.Horizontal = Alignment.Start,
    content: @Composable ColumnScope.() -> Unit
) {
    ...
```



Le dernier paramètre s'appelle `content`. Il doit être du type composable, et c'est une lambda sans paramètre et sans résultat (`Unit` signifie `void`). Donc c'est bien un bloc d'instructions qu'on peut soit placer en tant que paramètre de `Column`, soit après le nom. Voici la première variante :

```
@Composable
fun Accueil(name: String) {
    Column(content = {
        Text(text = "Bonjour $name", fontSize=20.sp)
        Text(text = "Je vois de grands progrès", color = Color.Green)
    })
}
```

Vous devrez passer énormément de temps à apprendre les composants d'interface. Une bonne documentation est https://www.composables.com/material3.

☛ Par exemple, cherchez `ElevatedCard` utilisée ici :

```
@Composable
fun Accueil(name: String) {
    ElevatedCard {
        Text(text = "Bonjour $name", fontSize=20.sp)
        Text(text = "Je vois de grands progrès", color = Color.Green)
    }
}
```

Compose met tous les composants du style Material Design 3 à disposition.

1.7. Paramétrage des vues

Dans l'exemple précédent, la fonction `ElevatedCard` produit une mise en page minimale, taille minimale, pas de marges, etc. Heureusement cette fonction est paramétrable.

☛ Essayez ceci :

```
@Composable
fun Accueil(name: String) {
    ElevatedCard {
        Column(
            horizontalAlignment = Alignment.CenterHorizontally,
            modifier = Modifier
                .fillMaxWidth()
                .padding(8.dp)
```

```
        ) {
            Text(text = "Bonjour $name", fontSize=20.sp)
            Text(text = "Je vois de grands progrès", color = Color.Green)
        }
    }
}
```

Remarquez au passage la notation *nb*`.dp` qui permet de spécifier directement un nombre de pixels indépendants de la résolution de l'écran. C'est une *extension* de la classe `Int`, c'est à dire une méthode qu'on rajoute à une classe existante.

Il y a deux types de paramètres de mise en page :

- des paramètres de la fonction comme `horizontalAlignment` pour `Column`. Ils sont totalement spécifiques à la fonction considérée.
- des modificateurs fournis avec `modifier = Modifier...` Ce sont des paramètres généraux concernant la taille de la vue que tous les composants peuvent utiliser. Par exemple, `fillMaxWidth` correspond au `match_parent` des vues Android et `wrapContentSize` correspond à peu près au `wrap_content`. On peut aussi gérer la taille de plusieurs vues avec des poids, presque comme avec Android Views.

☛ Essayez de commenter l'un ou l'autre des paramètres pour voir l'effet qu'ils ont sur `Column`.

En général, le `modifier` est reçu en paramètre du *composable* parent. Chaque fonction composable déclare son premier paramètre optionnel de ce type. Ainsi le composant peut être configuré de l'extérieur.

☛ On fait généralement comme ceci :

```
@Composable
fun Accueil(name: String, modifier: Modifier = Modifier) {
    ElevatedCard {
        Column(
            modifier = modifier.padding(8.dp),
            horizontalAlignment = Alignment.CenterHorizontally
        ) {
            Text(
                text = "Bonjour $name",
                fontSize = 20.sp,
                modifier = Modifier.padding(12.dp))
            Text(text = "Je vois de grands progrès", color = Color.Green)
        }
    }
}

@Preview
@Composable
fun AccueilPreview() {
    Column {
        Accueil(name = "numéro 10", modifier = Modifier.fillMaxWidth())
        Accueil(name = "numéro 6")  // valeur par défaut du modifier
```

```
    }
}
```

La notation `modifier: Modifier = Modifier`, à lire comme « nomparam: typeparam = valpardéfaut », en second paramètre de `Accueil` signifie qu'on peut passer un argument de type `Modifier`, mais que si on ne le fournit pas, alors il a une valeur par défaut, une instance de `Modifier` (en fait, c'est un objet compagnon car `Modifier` est une interface). En Kotlin, on ne met pas de parenthèses vide pour appeler un constructeur qui n'a pas de paramètres.

La taille à l'écran de `Accueil` peut être décidée lors de l'appel, permettant de réutiliser la fonction avec différentes configurations. La `Column` est configurée de l'extérieur, mais elle rajoute une marge de 8 dp. À ce propos, dans Compose, il n'y a pas de distinction entre *padding* et *margin*. On n'a que *padding* qui ajoute un espace à l'intérieur de l'élément, autour du contenu. Donc tout dépend de l'endroit où il est placé. En mettant un *padding* sur `Column`, ça met un espace autour de l'ensemble des deux textes. En mettant un *padding* sur le premier texte, ça espace uniquement ce texte.

Pour en savoir plus sur les modificateurs, voir cette documentation.

Nous n'avons pas le temps d'approfondir les aspects présentation graphique. Nous allons maintenant étudier le patron de conception qui est à la base de Compose.

2. Modèle et vue

Une fonction composable n'est pas un simple bout de code qui affiche quelque chose. Une telle fonction possède une signification allant au delà :

« Une fonction composable est destinée à convertir ses paramètres en interface utilisateur. » (ref)

Il faut comprendre la signification de cette phrase : une fonction composable rend visible sous forme d'une interface, un modèle de données passé en paramètres.

Dans les exemples précédents, la fonction `Text` visualise une chaîne de caractère passée en paramètre. Cette visualisation se fait sous la forme de pixels allumés sur l'écran... Pensez à la fonction `printf` du langage C. On lui passe des paramètres, chaînes et nombres et elle les affiche sur l'écran. C'est exactement le même concept. Ce qui vous étonne, c'est que, ici ça construit une interface graphique, au lieu d'afficher un simple texte sur le terminal.

☛ Que pensez-vous de ceci — essayez-le :

```
@Composable
fun AccueilMultiple(names: List<String>) {
    Column {
        for (name in names) {
            Text(text = "Bonjour $name !", modifier = Modifier.padding(4.dp))
        }
    }
}

@Preview
@Composable
```

```
fun AccueilMultiplePreview() {
    AccueilMultiple(listOf("pierre", "paul", "jacques"))
}
```

On a un modèle de données : la liste passée en paramètre. La fonction l'affiche sous forme de textes en colonne.

Autre exemple, on peut écrire une fonction composable qui reçoit un `XMen` en paramètre et qui l'affiche sous forme de textes et d'une image. C'est le concept d'un *view holder*, mais avec passage des données sous forme de paramètres. Cette fonction peut ensuite être appelée pour afficher toute une liste de `XMen` et on retrouve le concept de *recyclerview*.

Le modèle de données qui est affiché par la fonction composable peut être aussi complexe qu'on veut. Une fonction composable peut appeler d'autres fonctions pour afficher des parties du modèle de données, comme `AccueilMultiple` qui appelle `Column` qui appelle `Text`.

Une fonction composable ne retourne pas de valeur. Elle visualise seulement l'état actuel des données qu'on lui a fourni en paramètre. Il est essentiel qu'une fonction composable ne modifie aucune donnée de l'application. On dit qu'elle ne doit pas avoir d'« effet de bord ». Comme `printf`, il n'y a aucune altération des données dans ces fonctions. On dit que ce sont des « fonctions pures », c'est à dire que les traitements effectués ne dépendent que des paramètres. Plusieurs appels avec les mêmes paramètres doivent produire exactement le même résultat à l'écran.

La fonction peut quand même dérouler un algorithme, mais qui ne change pas les données. Cet algorithme doit être répétable. Un second appel de la même fonction avec les mêmes paramètres doit produire exactement le même affichage.

☛ Essayez ceci :

```
@Composable
fun AccueilMultipleSeulementJ(names: List<String>) {
    Column {
        for (name in names) {
            if (name.startsWith("j")) {
                Text(text = "Bonjour $name !", modifier = Modifier.padding(4.dp))
            }
        }
    }
}

@Preview
@Composable
fun AccueilMultipleSeulementJPreview() {
    AccueilMultipleSeulementJ(listOf("pierre", "paul", "jacques"))
}
```

Il n'est pas du tout imposé de rendre visibles tous les paramètres fournis. La fonction précédente ne salue que les personnes dont le nom commence par `j`.

Le patron de conception impose également que les paramètres fournis puissent servir à quelque chose dans la fonction. Par exemple, on ne doit pas fournir une autre liste si elle ne sert à rien.

Ainsi les paramètres d'une fonction composable sont tout ou partie des données gérées par l'application. La fonction composable les affiche dans l'activité. On va voir maintenant comment définir un modèle de données pour Compose.

2.1. Patron de conception général

Vous vous demandez sûrement comment on peut faire une interface qui soit autre chose qu'inerte. Comment faire un formulaire de saisie pour une donnée qui est ensuite affichée dans une liste, alors qu'on vient de dire que l'interface ne doit jamais modifier les données ?

Le principe est le suivant, une variante du patron MVC :

- la Vue est une fonction composable. Cette fonction peut évidemment appeler d'autres fonctions, dont des composables. Aucune de ces fonctions ne peut modifier les données.
- le Modèle est une structure de données totalement en lecture seule, non modifiable. Ce sont par exemple des objets constants, ou des collections constantes (*immutable*).
- Le Contrôleur transmet le modèle à la vue.

Alors avec ça, on se demande où on va. En quoi le logiciel peut être autre chose que totalement inerte après l'initialisation des données lors de son lancement ?

C'est là qu'il y a un patron de conception supplémentaire. C'est que si on veut modifier quelque chose dans les données, alors **on doit créer une copie des données** dans laquelle la modification a été faite puis remplacer les données précédentes par les nouvelles.

La raison, c'est que la vue affiche des données potentiellement très complexes et il est très difficile de mettre à jour une interface lors de modifications dans les données à cause de problèmes de concurrence (changements d'application dans Android, processus en arrière-plan et délais de réponse des serveurs). On pourrait se retrouver avec une interface à moitié à jour, ou même d'avoir des données contradictoires affichées en même temps, si on autorisait « tout le monde » à modifier les données.

Le plus simple est de remplacer les données entièrement en une seule opération : on prépare les nouvelles données à part, puis quand elles sont prêtes, on fait la bascule, ce qui invalide l'interface en entier et oblige à tout ré-afficher. C'est le contrôleur qui est le seul processus autorisé à faire les modifications et qui fait cette bascule. Ainsi, il ne peut y avoir qu'une seule source de vérité. L'affichage ne peut pas être bancal car il est construit uniquement avec des données valides.

Mais un tel gaspillage paraît stupéfiant, la recopie de toutes les données à chaque infime modification et un ré-affichage à chaque fois ? En réalité, la vue est capable de comparer les données précédentes avec les données actuelles et de ne pas faire de ré-affichage si rien n'a changé. La recherche des seules vues concernées par un nouveau contenu s'appelle la *recomposition*, et c'est très efficace.

Et côté modèle, on ne fait que des copies superficielles (*shallow copy*), voir ces explications, de manière à réallouer le moins possible de mémoire, c'est à dire seulement l'objet ou le container qui doit changer, mais en gardant les sous-objets référencés. Par exemple si une liste contient 2500 éléments et qu'on modifie l'un d'eux, alors on réalloue seulement l'objet liste et le nouvel élément, mais on garde les 2499 autres tels quels. C'est un peu moins désastreux côté efficacité.

2.2. Résumé

Donc si on reprend le patron de conception :

- le Modèle ou **État** (*state*) est en lecture seule. Il n'y a aucun *setter* ni aucune possibilité d'affectation de quoi que ce soit. La seule possibilité est d'en créer un autre avec les constructeurs et des méthodes de copie avec changement à la volée, les changements se faisant uniquement sous la forme de paramètres différents fournis au constructeur.
  Il peut donc y avoir plusieurs instances des données, correspondant à des étapes de la vie du logiciel. En principe, les anciennes instances inutilisées sont libérées par le système d'exploitation (*garbage collector*).
- la Vue affiche l'un des états, en général le plus récent, fourni en paramètre à une fonction **composable**. Quand l'utilisateur agit sur la vue : clic sur un bouton, saisie dans une zone de texte, etc, ça appelle une méthode du contrôleur pour faire un remplacement d'état. La vue n'a aucun moyen pour modifier le modèle elle-même.
- Le Contrôleur définit quel est l'état à afficher par la Vue et également, c'est lui qui effectue les remplacements d'état. Le contrôleur contient toutes les fonctions de modification d'état appelées par les écouteurs de la vue.

Il y a un point important, c'est que la fonction composable principale de la vue est automatiquement rappelée quand le contrôleur remplace l'état. C'est grâce à une sorte d'abonnement avec rappel automatique de la vue sur l'état, et ça se fait très simplement par les superclasses du contrôleur et de l'état.

Voyons comment c'est fait sur un premier exemple.

3. Feu tricolore

On va prendre un projet simple consistant à dessiner un feu tricolore (vert, orange, rouge) et avoir un bouton pour le faire changer d'état.

☛ Ajoutez trois sous-packages dans le projet : `feu3.state`, `feu3.controller` et `feu3.ui`

3.1. État

Une idée simple, c'est d'utiliser trois booléens, un par couleur. Évidemment, on peut faire autrement, par exemple avec un entier ou un `enum` qui code le feu qui est allumé.

☛ Ajoutez `Feu3State.kt` dans le package `fr.iutlan.tp9.feu3.state` :

```
package fr.iutlan.tp9.feu3.state

data class Feu3State(
    val rouge:  Boolean = true,
    val orange: Boolean = false,
    val vert:   Boolean = false,
) {
    /**
     * @return nom de la couleur courante
     */
    val nomCouleur: String
        get() =
            if (rouge) "rouge" else
            if (orange) "orange" else
```

```
            if (vert) "vert" else "???"
}
```

En Kotlin, une `data class` ne peut contenir que des variables membres et des *setters* et *getters*. Ici, on fait en sorte que tout soit en lecture seule. Il n'y a pas de *setter* et les membres sont des constantes (`val`). Ici, on a un *getter* pour une propriété calculée (voir TP4).

3.2. Vue

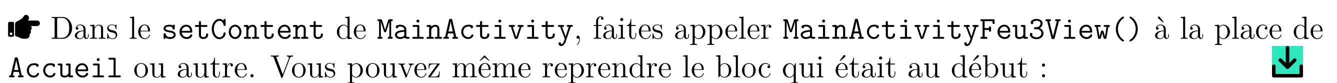
☛ Ajoutez `Feu3View.kt` dans le package `fr.iutlan.tp9.feu3.ui` :

```
package fr.iutlan.tp9.feu3.ui

import androidx.compose.foundation.*
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.shape.*
import androidx.compose.material3.*
import androidx.compose.runtime.Composable
import androidx.compose.ui.*
import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.unit.*
import androidx.lifecycle.viewmodel.compose.viewModel

import fr.iutlan.tp9.feu3.controller.Feu3ViewModel
import fr.iutlan.tp9.feu3.state.Feu3State


@Composable
fun MainActivityFeu3View(viewmodel: Feu3ViewModel = viewModel()) {
    // état auquel s'abonne cette fonction composable
    val state = viewmodel.state

    Column(
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        // affichage du feu, version 1
        Feu3ViewV1(state, modifier = Modifier.padding(16.dp))

        // bouton, voir la suite du TP
    }
}


@Composable
fun Feu3ViewV1(state: Feu3State, modifier: Modifier = Modifier) {
    Text(text = "Feu ${state.nomCouleur}",
```

```
        style = MaterialTheme.typography.titleLarge,
        modifier = modifier,
    )
}
```

C'est une visualisation très simple pour le feu tricolore. Elle appelle la méthode `nomCouleur` de l'état. On va améliorer ça peu à peu.

Il faut remarquer le paramètre `viewmodel` de `MainActivityFeu3View` et le fait qu'on va chercher sa propriété `state`. Cette propriété sera une instance de l'état, celle que le contrôleur aura modifié en dernier. À cause du fait que la classe `ViewModel` abonne la vue aux changements sur l'état, la vue sera automatiquement mise à jour.



Dans le `setContent` de `MainActivity`, faites appeler `MainActivityFeu3View()` à la place de `Accueil` ou autre. Vous pouvez même reprendre le bloc qui était au début :

```
        setContent {
            TP9Theme {
                // A surface container using the 'background' color from the theme
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    MainActivityFeu3View()
                }
            }
        }
```

3.3. Contrôleur (*view model*)

Ajoutez `Feu3ViewModel.kt` dans le package `fr.iutlan.tp9.feu3.controller` :

```kotlin
package fr.iutlan.tp9.feu3.controller

import androidx.compose.runtime.mutableStateOf
import androidx.lifecycle.ViewModel
import fr.iutlan.tp9.feu3.state.Feu3State

class Feu3ViewModel : ViewModel() {

    // singleton contenant l'état, observable mais privé
    private val _state = mutableStateOf(Feu3State())

    // getter pour voir cet état à l'extérieur de cette classe, mais setter privé
    var state
        get() = _state.value            // _state.value = instance de Feu3State
        private set(newvalue) {
            _state.value = newvalue     // remplace l'état par le nouveau
        }
```

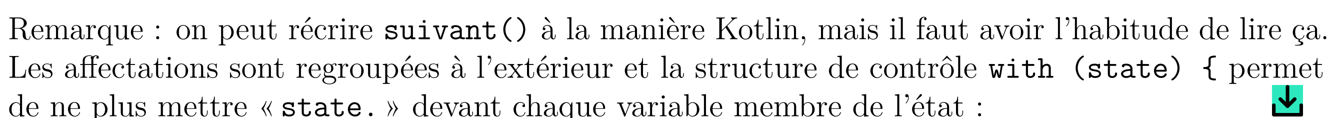
```
    init {
        reset()
    }

    /// méthodes pour modifier les données

    fun reset() {
        state = Feu3State()
    }

    fun suivant() {
        if (state.rouge) {
            state = Feu3State(false, false, true)
        } else if (state.vert) {
            state = Feu3State(false, true, false)
        } else {
            state = Feu3State(true, false, false)
        }
    }
}
```

C'est cette classe qui a la responsabilité de modifier l'état lorsqu'il y a un événement. Par exemple, si l'utilisateur appuie sur le bouton `changer` qu'on va programmer tout à l'heure, alors ça appelle la méthode `suivant()`. Cette méthode affecte `state` avec une nouvelle valeur, en fonction de l'état actuel : du feu rouge on passe au feu vert, etc. À chaque fois, on réaffecte tout l'état d'un coup. Jamais on ne modifie les variables membres (de toutes façons, ce sont des constantes).

Il y a un point super technique dont tout dépend. C'est la déclaration des deux propriétés `_state` et `state` au début de la classe.

- `val _state = mutableStateOf(Feu3State())` déclare `_state` en tant que variable membre constante un peu particulière. D'abord cette variable possède une propriété `_state.value` qui contient l'état, une instance de `Feu3State`. Alors la variable `_state` est constante, c'est à dire non réaffectable, mais sa propriété `value` est modifiable. Et en plus, il est possible de s'abonner à ses changements de valeur. Toute affectation de `_state.value` réveille tous les abonnés. Les abonnés sont simplement tous les objets qui ont consulté cette propriété. C'est le cas de `MainActivityFeu3View` quand elle fait `val state = viewmodel.state`.
- `var state get() = _state.value` permet de définir un *getter* public pour `_state.value`. En fait, c'est lui qui est appelé par `MainActivityFeu3View`. Ce *getter* est public, tandis que `_state` est privée, ce qui permet de protéger l'état, d'interdire toute modification du modèle par une autre classe que le contrôleur. Pour ça, le *setter* est privé.

Remarque : on peut récrire `suivant()` à la manière Kotlin, mais il faut avoir l'habitude de lire ça. Les affectations sont regroupées à l'extérieur et la structure de contrôle `with (state) {` permet de ne plus mettre « `state.` » devant chaque variable membre de l'état :



```
    fun suivant() {
        state = with (state) {
```

```
            if (rouge) {
                Feu3State(false, false, true)
            } else if (vert) {
                Feu3State(false, true, false)
            } else {
                Feu3State(true, false, false)
            }
        }
    }
```

3.4. Écouteurs

On veut rajouter un bouton pour changer l'état du feu. L'idée est que, cliquer ce bouton appelle la méthode `suivant()` du contrôleur.

☛ Ajoutez ceci au bon endroit dans `MainActivityFeu3View` :

```
        Button(
            onClick = {
                viewmodel.suivant()     // modif par le contrôleur
            },
            modifier = Modifier.fillMaxWidth().padding(32.dp)
        ) {
            Text(text = "état suivant")
        }
```

Dans Compose, les boutons doivent être remplis avec un texte, ou un icône ou les deux...

Remarquez le paramètre `onClick` du bouton. Il appelle la méthode voulue dans le contrôleur, `viewmodel`. C'est sous la forme d'une lambda sans paramètre. Il y a des cas où il y a un paramètre, par exemple quand c'est une `CheckBox`, on reçoit l'état coché ou non dans `it`.

☛ Testez sur AVD.

☛ Suggestion : dans le contrôleur, il y a une méthode `reset()`...

Vous avez là l'essentiel du patron de conception des applications Compose :

- l'état non modifiable,
- la vue en tant que fonction d'affichage d'un état et abonnée à ses changements,
- le contrôleur en tant que maître des changements d'état par remplacement intégral de l'état.

3.5. Autres visualisations

Voici une autre visualisation à compléter :

```
@Composable
fun Feu3ViewV2(state: Feu3State, modifier: Modifier = Modifier) {
    Column(
        modifier.wrapContentSize()
    ) {
```

```
        // feu rouge
        Row(Modifier.align(Alignment.Start).padding(horizontal = 16.dp)) {
            RadioButton(
                selected = state.rouge,
                onClick = null      // non réactif
            )
            Text(
                text = "rouge",
                modifier = Modifier.padding(start = 16.dp)
            )
        }
        // TODO idem pour le feu orange et pour le feu vert
    }
}
```

☛ Ajoutez un appel à `Feu3ViewV2` identique à celui de `Feu3ViewV1` juste en dessous dans `MainActivityFeu3View`. Ça fera deux visualisations différentes et simultanées pour le même état.

☛ Voici une troisième visualisation :

```
@Composable
fun Feu3ViewV3(state: Feu3State, modifier: Modifier = Modifier) {
    Column(modifier = modifier
        .fillMaxWidth()
        .wrapContentSize(Alignment.Center)) {
        Box(
            contentAlignment = Alignment.Center,
            modifier = Modifier
                .size(48.dp, 128.dp)
                .clip(RoundedCornerShape(16.dp))
                .background(Color.DarkGray)
        ) {
            Column {
                Feu(Color.Red,    state.rouge)
                Feu(Color.Orange, state.orange)
                Feu(Color.Green,  state.vert)
            }
        }
    }
}

/**
 * dessine un disque coloré ou gris selon isOn
 */
@Composable
fun Feu(color: Color, isOn: Boolean, modifier: Modifier = Modifier) {
    Canvas(
        modifier = Modifier.size(40.dp).padding(4.dp),
```

```
        onDraw = {
            drawCircle(color = if (isOn) color else Color.Gray)
        }
    )
}

// définit la couleur Color.Orange par une extension de la classe Color
private val Color.Companion.Orange: Color
    get() = hsv(33.0f, 1.0f, 1.0f)
```

☛ Ajoutez un appel à `Feu3ViewV3` sous celui de `Feu3ViewV2`. Ça fera trois visualisations différentes pour le même état.

☛ Remarquez plusieurs petites choses dans `Feu3ViewV3` :

- La fonction `Feu` n'a pas besoin de lire tout l'état, on ne lui passe que le booléen qui la concerne et la couleur du feu. C'est un principe d'économie du patron de conception.
- Elle vous montre comment on dessine avec Compose (comparer avec le TP7). Le paramètre `onDraw` du `Canvas` est une lambda dans laquelle on met les instructions de dessin.
- La couleur `Color.Orange` n'est pas prédéfinie, mais Kotlin permet d'ajouter des *extensions* à une classe existante.

3.6. État second

Maintenant que le projet marche, on va changer le modèle de données. C'est toujours intéressant de voir jusqu'où va l'indépendance des modules. Il peut arriver dans un projet qu'on change le modèle pour tout autre chose ; la question est de savoir si tout le reste s'effondre...

☛ Dans le fichier `Feu3State.kt`, renommez *sans refactoring* la classe `Feu3State` en `Feu3StateV1` (ajoutez juste `V1` à son nom). Surtout n'acceptez pas la proposition d'un renommage partout. Au contraire, ça doit faire comme si la classe `Feu3State` avait disparu, donc provoquer des erreurs partout.

☛ Ajoutez `Feu3StateV2.kt` dans le package `fr.iutlan.tp9.feu3.state` :

```
package fr.iutlan.tp9.feu3.state

enum class FeuCouleur {
    ROUGE,
    ORANGE,
    VERT
}

data class Feu3State(
    val couleur: FeuCouleur = FeuCouleur.ROUGE
) {
    val nomCouleur: String
        get() = couleur.toString()
}
```

Ça redéfinit la classe `Feu3State` d'une autre manière. On s'aperçoit alors qu'il y a deux types d'erreurs dans les autres modules :

1. Les booléens `rouge`, `orange` et `vert` ne sont plus disponibles... C'est très simple à corriger :

☛ Ajoutez ceci dans la classe `Feu3State` (la nouvelle) :

```
    val rouge  get() = couleur == FeuCouleur.ROUGE
    val orange get() = couleur == FeuCouleur.ORANGE
    val vert   get() = couleur == FeuCouleur.VERT
```

2. Le constructeur a changé. Il ne demande plus 3 booléens, mais un `enum`. Ça cause une erreur dans le contrôleur et il ne semble pas possible de garder le code existant. Alors,

☛ Ajoutez également cette méthode dans `Feu3State` (la nouvelle) :

```
    fun copyChangeCouleur(nouvelle: FeuCouleur): Feu3State {
        return this.copy(couleur = nouvelle)
    }
```

La méthode `copy()` est automatiquement définie pour une `data class`. C'est un constructeur qui recopie les valeurs de `this`. On peut lui passer des paramètres portant les noms des champs, pour leur donner d'autres valeurs dans la copie.

☛ Dans la méthode `suivant()` du contrôleur, remplacez les appels à `Feu3State(3 booléens)` par `state.copyChangeCouleur(FeuCouleur.XXX)`. Et l'instruction `with (state)` permet d'enlever tous les `state.`.

On voit que le changement de modèle a beaucoup impacté le contrôleur, pour effectuer les modifications des données, mais très peu la vue, grâce aux *getters* qu'on a pu créer. On devine que si les changements rendent impossible l'écriture des *getters*, il faudra aussi refaire la vue.

4. Deuxième projet

On va maintenant appliquer ces connaissances à un projet de liste de messages. Ces messages sont un texte accompagné d'un icône.

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

4.1. État

☛ Créez `MessageListState.kt` dans le package `fr.iutlan.tp9.messages.state` et mettez ce contenu :

```
package fr.iutlan.tp9.messages.state

enum class Icone {
    ETOILE,
    APPEL,
    FAVORI

    // SUPPRIMEZ CES 3 LIGNES SUR UNE INSTALLATION PERSONNELLE
```

```
    ; companion object {
        val entries get() = Icone.values()
    }
}
```

NB: le point virgule et le `companion object` sont à retirer sur une installation personnelle. La propriété `entries` est définie sur les versions récentes de Kotlin, mais pas encore à l'IUT.

☛ Dans le même fichier ou dans un autre du même package, ajoutez une classe de données appelée `Message` définie par deux propriétés constantes `icone` de type `Icone` et `texte` de type `String`.

☛ 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 cette méthode à la classe `MessageListState` :

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

Cette méthode crée une copie de la liste en lui ajoutant un 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.

4.2. Contrôleur

Dans ce projet, c'est la vue qui est la plus complexe, et de loin. Alors on s'occupe d'abord du contrôleur. Son rôle est de mémoriser une instance de l'état et de gérer toutes les modifications en tant que remplacement de l'état par un autre.

☛ Ajoutez `MessageListViewModel.kt` dans `fr.iutlan.tp9.messages.controller` :

```
package fr.iutlan.tp9.messages.controller

import androidx.compose.runtime.mutableStateOf
import androidx.lifecycle.ViewModel
import fr.iutlan.tp9.messages.state.*

class MessageListViewModel : ViewModel() {

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

    // getter pour voir cet état à l'extérieur de cette classe, mais setter privé
    var state
        get() = _state.value            // _state.value = instance de MessageListState
        private set(newvalue) {
            _state.value = newvalue     // remplace l'état par le nouveau
```

```
        }

    init {
        state = MessageListState()      // constructeur liste vide
    }

    /// méthodes pour modifier les données

    fun addNewMessage(message: Message) {
        if (message.texte.isBlank()) return     // pas de message vide
        state = state.copyAdd(message)
    }
}
```

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

☛ Relisez la fonction `copyAdd` de la classe `MessageListState`, appelée par `addNewMessage`.

4.3. 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.tp9.messages.ui`.

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

4.3.1. 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.tp9.messages.ui

// TODO mêmes imports que dans Feu3View.kt

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

@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
@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.

Normalement vous devez voir un combiné téléphonique bleu foncé dans la partie droite de l'éditeur. Vous pourriez changer la couleur selon l'icône en faisant comme pour l'image.

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

4.3.2. Affichage d'un message isolé

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

☛ Ajoutez et complétez ceci — vous savez le faire maintenant :

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

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

@Preview
@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.

4.3.3. Affichage d'une liste de messages

☛ 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
@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`, mais la fonction `LazyColumn` est bien plus intéressante. Elle fonctionne comme un `RecyclerView`. Elle gère le défilement vertical et n'affiche que les données visibles, 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`. Il n'y a besoin que de ça pour refaire le TP5.

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

☛ Modifiez le cœur en ceci :

```
    LazyColumn(modifier) {
        itemsIndexed(messages) { 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). On dessine un séparateur seulement à partir du deuxième message.

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
@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 appels de fonctions. On peut employer des boucles, des alternatives, faire des calculs.

4.3.4. 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, modifier.fillMaxSize().weight(1f))

        // afficher un bouton "nouveau" pour créer un message
        Button(
            onClick = { /* ICI n°2, voir plus loin */ }
        ) {
            Text("Nouveau")
        }
    }
}

@Preview
@Composable
fun MainActivityMessageListPreview() {
    MainActivityMessageListView()
}
```

Voyez comment le bouton est créé. On lui met un texte en tant que contenu. On pourrait mettre un icône ou autre chose.

Vous pouvez remarquer un poids appliqué à la liste. 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. En fait, 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 la répartition de l'espace total entre toutes les vues. Ici, le bouton a la place dont il a besoin et la liste récupère tout ce qui reste.

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

4.3.5. Création d'un nouveau message

Maintenant, il reste à faire ajouter un message. On va le faire, non pas par une nouvelle activité (voir `EditActivity` dans le TP6), mais avec un dialogue. C'est une fenêtre qui vient se superposer à l'activité le temps de la saisie du message.

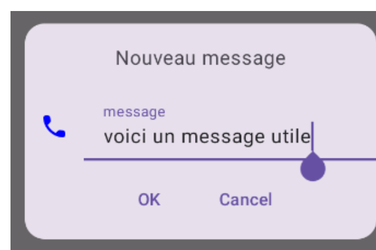


Figure 2: Dialogue

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.

4.3.6. Panneau de boutons OK et Annuler

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

Ajoutez cette fonction composable :

```
@Composable
fun PanneauBoutonsOKAnnuler(
    onAccept: () -> Unit,
    onCancel: () -> Unit,
    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
@Composable
fun PanneauBoutonsOKAnnulerPreview() {
    PanneauBoutonsOKAnnuler(
        onCancel = {},
        onAccept = {}
    )
}
```

Cette fonction crée une ligne de deux boutons. Le plus nouveau, c'est les paramètres. Jusqu'ici, c'étaient des données, icône, message, liste. Là, ce sont des lambda. La fonction `PanneauBoutonsOKAnnuler` demande deux lambdas. La première sera appelée si l'utilisateur clique sur OK, l'autre s'il clique sur Annuler. On verra plus loin comment on définit ces lambdas. En tous cas, on ne peut rien faire dans la prévisualisation.

Il y a un patron de conception dans cette fonction. Une fonction composable est appelée avec des paramètres constants pour spécifier ce qu'elle doit dessiner. 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.

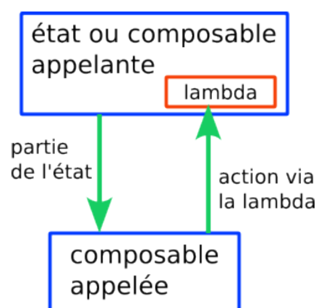


Figure 3: Flux unidirectionnel

Une fonction composable appelée exécute une lambda qui lui est passée en paramètre pour faire un changement sur l'état de l'appelante en fonction des actions de l'utilisateur. On rappelle qu'une fonction composable ne peut pas modifier l'état car il est constant. Les lambda passées en paramètre seront exécutées dans la fonction appelante, et ça peut remonter jusqu'au contrôleur.

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 qui viendront du 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.

4.3.7. 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.

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

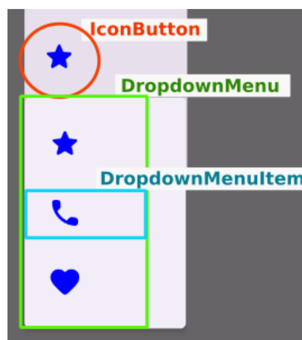


Figure 4: Menu déroulant

☛ 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)
                    )
                }
            )
        }
    }
}
```

NB: Vous aurez un avertissement sur l'instruction `Icone.entries` à l'IUT car la version de Kotlin date un peu. Il y a une rustine dans la classe `enum Icone`, une fonction compagnon qui définit `entries` pour un `enum`, à enlever si vous êtes sur une installation personnelle.

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 visualisation. Alors faites ce qui suit.

☛ Dans `MainActivity`, là où il y a `setContent`, mettez ceci :

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

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.

Le problème est plus grave. 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 de l'état.

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 le cas, mais on pourrait stocker cet état, la liste des messages, dans une base de données Realm par exemple). 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) :

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

@Composable
fun Exemple(...) {

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

    // bouton pour changer isDropdownVisible
    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, cause bascule 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*.

4.3.8. Dialogue de saisie d'un message

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

☛ Ajoutez cette fonction :

```kotlin
@Composable
fun NewMessageDialog(
    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
@Composable
fun NewMessageDialogPreview() {
    NewMessageDialog(
        onCancel = {},
        onAccept = { _,_ -> }
    )
}
```

☛ Pour tester sur AVD, mettez ceci dans `MainActivity`, là où il y a `setContent` :

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

☛ Vérifiez que le dialogue fonctionne bien. Le bouton OK doit entraîner l'affichage du message. Par contre, c'est normal que le dialogue ne disparaisse pas.

4.3.9. Fonction principale revisitée

Il reste à terminer la fonction `MainActivityMessageListView`.

☛ Remplacez les commentaires `ICI n°1 et 2` par ce qu'il faut pour afficher et masquer le dialogue de saisie d'un message. L'appel à `NewMessageDialog` doit être placé dans une conditionnelle commandée par un booléen d'état d'interface. Il faut aussi enregistrer le message dans l'état s'il est validé. C'est à dire que `onAccept` doit en plus faire une action sur `viewmodel`.

☛ 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.

Il manque maintenant l'enregistrement des messages par exemple dans une base Realm. C'est à faire côté contrôleur, mais c'est beaucoup trop pour ce TP.

5. Travail à rendre

Important : votre projet doit se compiler et se lancer sur un AVD. La note sera zéro si ce n'est pas le cas. Mettez donc en commentaire ce qui ne compile pas.

Avec le navigateur de fichiers, descendez dans le dossier `app/src` du projet TP9.

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.

Cliquez droit sur le dossier `main`, choisissez `Compresser...`, format `ZIP`, cliquez sur `Creer`. Déposez l'archive `main.zip` sur Moodle au bon endroit, voir sur la page Moodle R4.A11 Développement Mobile.