L'objectif du TP est de découvrir la nouvelle API Android appelée Jetpack Compose. C'est celle qui est recommandée par Google. Elle repose sur des concepts totalement différents de ce qu'on a vu dans les précédents TP, à part la notion d'activité.

Figure 1: Logo Jetpack Compose

Rappel : le travail est strictement personnel. Vous ne devez en aucune façon déposer le travail réalisé par autre que vous-même (personne ou IA). La note est zéro pour tous les participants à une tricherie. Si vous rencontrez une difficulté, vous devez créer un fichier expliquant le problème et appelé `IMPORTANT.txt` à la racine de l'archive que vous déposez sur Moodle.

1. Découverte de Compose

1.1. Création du projet

👉 Pour commencer, créez un projet Android, de type « Empty Activity » avec le logo Compose, appelé `TP4` (ni `tp4`, ni `Tp4`). Le package doit être `fr.iutlan.tp4`.

👉 Ouvrez le fichier `app/build.gradle.kts` (pas le premier, celui du projet, mais le 2e)

👉 Ajoutez cette ligne dans le bloc `dependencies {` :

```
implementation("androidx.lifecycle:lifecycle-viewmodel-compose:2.8.7")
```

Vous allez la voir surlignée en orange, avec la suggestion de la transformer en *catalogue de version*. Acceptez. 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.

NB: s'il y a un problème de compilation sur les postes fixes de l'IUT, c'est parce que l'assistant du SDK crée un projet trop récent pour la version d'Android Studio installée. Uniquement dans ce cas, faites ceci :

1. Dépliez la catégorie `Gradle Scripts` dans le projet
2. Ouvrez `libs.versions.toml`. Ce fichier contient les numéros de version des dépendances. Deux d'entre elles ne sont pas bonnes, `coreKtx` et `activityCompose`.
3. Modifiez ces deux lignes comme ceci :

   ```
   coreKtx = "1.13.1" ou "1.10.1"
   activityCompose = "1.9.0"
   ```

4. Cliquez sur `Sync Now` en haut.
5. Tentez de relancer l'application sur l'AVD.

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 { ... }` ainsi que la ligne `enableEdgeToEdge()` et ajoutez ce qui suit, puis testez sur l'AVD : 

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

En fait, il ne reste plus que ceci d'actif, le reste est commenté ou non utilisé :

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        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 qui 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 le corps de la lambda. C'est le cas avec `setContent`. En plus, on n'a pas besoin de fournir d'autres paramètres à `setContent`, donc on la met sans aucune parenthèse.

On aurait pu écrire ceci :

```
setContent(null, { Text("OK") })
```

Le dernier paramètre est un simple bloc d'instructions, donc une lambda sans paramètre et sans résultat. C'est ce que demande la fonction `setContent`.

Le corps de la lambda 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. 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.

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`.

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. Refaire l'importation correctement.

Il y a des importations de classes et aussi d'*extensions*. Ce sont des ajouts à une classe existante. Par exemple, `sp` est un ajout à la classe `Int` qui permet d'écrire `32.sp` qui fait appeler une méthode invisible pour automatiquement transformer 32 en nombre de pixels sur l'écran.

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(showBackground = true)
@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`. L'option `(showBackground = true)` affiche le fond, c'est utile en mode sombre.

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 (remplacez `Accueil`) : 

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

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. Mais ici, on n'a pas précisé comment ces textes devaient se placer.

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.Blue)
    }
}
```

👉 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.Blue)
    })
}
```

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.Blue)
    }
}
```

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.Blue)
        }
    }
}
```

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.

NB: les paramètres obligatoires n'ont pas de valeur par défaut. Dans `fun ft(p1: String, p2: Int, p3: Float=3.14, p4: String="ok")`, le premier paramètre optionnel est `p3` ; c'est le premier à avoir une valeur par défaut.

👉 Faites de CTRL-clic sur les noms de fonctions `ElevatedCard`, `Column` et `Text` pour afficher les paramètres de ces fonctions. Vous verrez que chaque fois, le premier paramètre optionnel est `modifier`.

👉 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.Blue)
        }
    }
}

@Preview(showBackground = true)
@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 « 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*, voir TP2 §7.7 méthodes de classe, car `Modifier` est une interface. En Kotlin, on ne met pas de parenthèses vides 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 ici 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 paramètre.

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(showBackground = true)
@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.

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 rien. Elle visualise seulement l'état actuel des données qu'on lui a fournies en paramètre.

D'autre part, 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 à l'extérieur. Cet algorithme doit être répétable. Un second appel de la même fonction avec les mêmes paramètres doit produire exactement le même affichage.

👉 Essayez ceci dans `MainActivity` :

```
@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(showBackground = true)
@Composable
fun AccueilMultipleSeulementJPreview() {
    AccueilMultipleSeulementJ(listOf("paul", "jean", "pierre", "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. Si les données ne peuvent pas être modifiées, comment le logiciel peut-il être utile ?

C'est là qu'il y a une règle supplémentaire dans le patron de conception. C'est :

- 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, et seul le contrôleur a le droit de demander cela et en utilisant les méthodes du modèle.

C'est à dire que le contrôleur demande au modèle de se cloner tout en changeant quelque chose dedans et ensuite le contrôleur fournit le nouveau modèle à la vue, et supprime l'ancien modèle.

La raison, c'est que la vue affiche des données potentiellement très complexes, qui peuvent venir d'un serveur distant. L'expérience a montré qu'il est très difficile de mettre à jour une interface lors de modifications dans les données. Il y a des problèmes de concurrence : changements d'activité de premier plan dans Android, des processus réseau en arrière-plan et les délais de réponse des serveurs. C'est très compliqué avec Android Views. On est obligé de tout gérer. Quand c'est mal fait, on se retrouve avec une interface à moitié à jour, ou cassée.

Le plus simple est de remplacer la totalité des données en une seule opération (les A, C et I de ACID, voir wikipedia) : 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 cette bascule. Ainsi, il ne peut y avoir qu'une seule source de vérité à tout instant. L'affichage ne peut pas être bancal car il est construit uniquement avec des données entièrement 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.

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. Les anciennes instances inutilisées sont libérées par le système d'exploitation (*garbage collector*).

- la Vue affiche l'un des états, normalement 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 dirige 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. Le contrôleur n'a pas à s'en occuper. Il doit seulement affecter une variable avec l'état considéré comme courant. 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` (clic droit sur `fr.iutlan.tp4` dans `kotlin+java` puis)

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.tp4.feu3.state` :

```
package fr.iutlan.tp4.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. Donc, pas de *setter* et les membres sont des constantes (`val`). On a seulement un *getter* pour une propriété calculée (voir TP2).

3.2. Vue

👉 Ajoutez `Feu3View.kt` dans le package `fr.iutlan.tp4.feu3.ui` :

```
package fr.iutlan.tp4.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.tp4.feu3.controller.Feu3ViewModel
import fr.iutlan.tp4.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 {
            TP4Theme {
                // 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.tp4.feu3.controller` : 

```
package fr.iutlan.tp4.feu3.controller

import androidx.compose.runtime.mutableStateOf
import androidx.lifecycle.ViewModel
import fr.iutlan.tp4.feu3.state.Feu3State

class Feu3ViewModel : ViewModel() {

    // singleton contenant l'état
    // getter observable à l'extérieur de cette classe, mais setter privé
    var state by mutableStateOf(Feu3State())
        private set

    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

La séparation entre modèle et vue permet de créer d'autres visualisations pour les mêmes données.

👉 Voici un autre composable à compléter pour afficher le feu : 

```
@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` (laissez le premier appel). 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
        .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)
```

```
@Preview(showBackground = true)
@Composable
fun Feu3ViewV3Preview() {
    // une seule des deux lignes suivantes, soit V1, soit V2 (voir § état second)
    Feu3ViewV3(state = Feu3State(false, true, false))    // pour la V1
    //Feu3ViewV3(state = Feu3State(FeuCouleur.ORANGE))   // pour la V2
}
```

👉 Ajoutez un appel à `Feu3ViewV3` sous celui de `Feu3ViewV2`, en laissant les deux autres. 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. La base est un *canvas* (canevas), c'est à dire une zone de dessin 2D. Le paramètre `onDraw` du `Canvas` est une lambda sans paramètre, donc un simple bloc `{...}` dans lequel on met les instructions de dessin. On reverra cela en détail dans le TP6.
- La couleur `Color.Orange` n'est pas prédéfinie, mais Kotlin permet d'ajouter des *extensions*, de nouvelles choses 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 éléments, modèle et vue. Il peut arriver dans un projet qu'on doive faire de gros changements sur le modèle à cause d'une demande du client. 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.tp4.feu3.state` :

```
package fr.iutlan.tp4.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. Carrefour

On voudrait maintenant afficher un carrefour avec plusieurs feux, et des règles de gestion pour qu'ils aient les bonnes couleurs. Pour simplifier, ça sera un carrefour en croix avec deux routes à angle droit, 4 feux dont un seul est vert, puis orange à un instant donné.

Le principe est d'avoir un feu « courant » qui passe au vert, pendant que tous les autres sont rouges. Ce feu passe ensuite à l'orange, les autres restent rouges. Enfin il passe au rouge, les autres restant encore au rouge à ce stade, mais ce dernier changement fait passer au feu suivant.

Techniquement, on va faire cela dans la même application, mais des *packages* différents.

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

NB: on conserve évidemment tout ce qui concerne les feux tricolores.

4.1. État

👉 Créez `CarrefourState.kt` dans le *package* `carrefour.state`.

Il faut définir une classe `CarrefourState` dont l'état contient :

- un entier qui indique quel est le feu courant,
- un tableau de 4 `Feu3State`.

Voici le squelette : 

```
package fr.iutlan.tp4.carrefour.state

@Suppress("ArrayInDataClass")
data class CarrefourState (
    val courant: Int = 0,       // feu courant (vert ou orange, les autres rouges)
    val feux: Array<Feu3State> = arrayOf(
        Feu3State(), Feu3State(), Feu3State(), Feu3State()  // 4 feux
    )
) {
    // getters et méthodes de copie avec modification

    /**
     * retourne le Feu3State courant
     * @return le Feu3State de feux d'indice courant
     */
    val feuCourant  get() = feux[courant]

    /**
     * retourne un CarrefourState dans lequel la couleur du feu courant est modifiée
     * @param couleur : nouvelle couleur du feu courant
     * @return CarrefourState
     */
    fun copyChangeCouleurCourant(couleur: FeuCouleur): CarrefourState {
        // TODO faire la copie avec modification
        /* à supprimer une fois que c'est fait ---> */return this
    }

    /**
     * retourne un CarrefourState dans lequel le numéro du feu courant est modifié
     * @param num : numéro du feu courant, ramené entre 0 et le nombre de feux - 1
     * @return CarrefourState
     */
    fun copyChangeCourant(num: Int): CarrefourState {
        // TODO faire la copie avec modification
        /* à supprimer une fois que c'est fait ---> */return this
    }
}
```

Il y a une particularité, `@Suppress(...)`. Cette ligne enlève un avertissement du compilateur. Pour Kotlin, la présence d'un tableau dans les propriétés oblige normalement à ajouter deux méthodes pour comparer deux instances de cette classe, `equals` et `hashCode`. Ces deux méthodes servent lors des comparaisons d'instances. Or les propriétés de type tableau sont comparées seulement par *référence*, c'est à dire, pour simplifier, par leur adresse en mémoire. Donc on pourrait avoir deux `CarrefourState` différents, mais partageant le même tableau et avec la même

valeur pour le feu courant ; elles seront considérées comme égales. Ce n'est pas gênant dans notre cas, car justement, on va partager les tableaux, ou les recréer entièrement.

Vous voyez deux méthodes, `copyChangeCouleurCourant` et `copyChangeCourant`. Soit vous les programmez maintenant, soit après avoir étudié tout ce qui suit, contrôleur et vue.

4.2. Contrôleur

👉 Ajoutez `CarrefourViewModel.kt` dans `carrefour.controller` : 

```
package fr.iutlan.tp4.carrefour.controller

import androidx.compose.runtime.mutableStateOf
import androidx.lifecycle.ViewModel
import fr.iutlan.tp4.carrefour.state.*

class CarrefourViewModel : ViewModel() {

    // singleton contenant l'état
    // getter observable à l'extérieur de cette classe, mais setter privé
    var state by mutableStateOf(CarrefourState())
        private set

    init {
        state = CarrefourState()        // constructeur état initial
    }

    /// méthodes pour modifier les données

    fun suivant() {
        // TODO règles de fonctionnement du carrefour
    }
}
```

Il n'a qu'une seule méthode, `suivant` qui doit appliquer les règles suivantes :

- Si le feu courant, indiqué par `state.feuCourant`, est rouge, alors il passe au vert.
- Si le feu courant est vert, alors il passe à l'orange.
- Si le feu courant est orange, alors il passe au rouge et on incrémente `state.courant`.

Ces changements sont réalisés par des copies avec changement de l'état, à l'aide des méthodes `copyChangeCouleurCourant` et `copyChangeCourant`.

4.3. Vue

👉 Voici une proposition d'affichage : 

```
package fr.iutlan.tp4.carrefour.ui

// TODO tous les import à faire
```

```
@Composable
fun MainActivityCarrefourView(viewmodel: CarrefourViewModel = viewModel()) {
    // TODO afficher un CarrefourView et un Button
}

@Composable
fun CarrefourView(state:CarrefourState, modifier:Modifier=Modifier, size:Dp=180.dp) {
    Box(
        modifier.fillMaxWidth().padding(60.dp)
    ) {
        val mod = Modifier.scale(0.5f).align(Alignment.Center)
        Feu3ViewV3(state = state.feux[0], modifier = mod)
        Feu3ViewV3(state = state.feux[1], modifier = mod)
        Feu3ViewV3(state = state.feux[2], modifier = mod)
        Feu3ViewV3(state = state.feux[3], modifier = mod)
    }
}

@Preview(showBackground = true)
@Composable
fun CarrefourPreview() {
    CarrefourView(state = CarrefourState(1, arrayOf(
        Feu3State(), Feu3State(FeuCouleur.VERT), Feu3State(), Feu3State())
    ))
}
```

☛ Complétez `MainActivityCarrefourView` exactement comme `MainActivityFeu3View`, avec un abonnement au `CarrefourViewModel` et un appel à `CarrefourView` et `Button`.

☛ Complétez le paramètre `modifier` des 3 derniers appels à `Feu3ViewV3` dans `CarrefourView`. Ce paramètre place les feux autour et à une certaine distance du centre de la page. Par exemple, le premier feu est positionné avec l'instruction suivante :

```
    modifier = mod.offset(y=-size)
```

Les trois autres feux ont presque le même modificateur, sauf qu'il faut mettre `y=size` ou `x=-size` ou `x=size`. À vous de comprendre quoi faire.

☛ Mettez en place `MainActivityCarrefourView` dans l'activité et terminez les méthodes.

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, ouvrez `~/AndroidStudioProjects/TP4` (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. Décrivez exactement ce qui s'est passé. Le correcteur pourra lire ce fichier au moment de la notation et compenser votre note.

Déposer l'archive dans le dépôt Moodle R4.A11 Développement Mobile , rendu TP4.