Ce TP montre comment on dessine dans une application Android Compose. On va créer une sorte de « Draw » pour dessiner des figures géométriques simples, lignes, rectangles, cercles et ellipses. On pose le doigt sur l'écran et on le fait glisser pour délimiter chaque nouvelle figure.

Dans le principe, c'est relativement simple : le dessin est une liste de figures géométriques. On a une boucle dans la vue pour dessiner chaque figure avec une fonction composable. La vue est également réactive aux touchers sur l'écran. Les écouteurs permettent de modifier l'état pendant la création de la figure.

Le problème, ce sont les patrons de conception sous-jacents dans Compose qui rendent les choses plus compliquées.

1. Création de l'application `Dessin`

1.1. Mise en place du projet

1. Relisez le début du TP4, §1.1 pour savoir comment on crée un projet Jetpack Compose.
2. Créez le projet TP6 exactement comme le TP4, dépendance incluse. ATTENTION au package: `fr.iutlan.tp6` impérativement, sinon vous ne pourrez pas faire la suite.
3. Vérifiez que la compilation se passe bien et corrigez les versions dans `libs.versions.toml` s'il y a un problème.
4. Ouvrez le dossier du projet, `~/AndroidStudioProjects/TP6`.
5. Descendez dans `app/src`. Vous devez voir trois dossiers : `main`, `test` et `androidTest`. Supprimez totalement ces trois dossiers et restez à ce niveau.
6. Téléchargez tp6.zip, ouvrez-le, vous voyez un dossier `main` qu'il faut faire glisser dans `app/src`. Ainsi ça reconstruit le dossier `main` dans `app/src` du projet.

1.2. Contenu du projet

Vous allez trouver trois sous-packages : `state`, `controller` et `ui` qui ont les mêmes rôles que dans les TP précédents : représenter l'état de l'application, gérer ses changements et l'afficher. Il y a un package supplémentaire, `model` qui contient les définitions de la classe `Figure` et de sous-classes pour pouvoir définir l'état.

👉 Dépliez tous les sous-packages pour voir les sources existants.

Pour comprendre l'ensemble du projet en quelques coups d'œils, suivez cet itinéraire. NB: dans un premier temps, ne touchez à rien, ne programmez aucun TODO ni FIXME.

1. Ouvrez `MainActivity.kt`. La fonction `setContent` dessine un `MainActivityView` dans un `Surface`. Ce dernier est une sorte de container pour accueillir des fonctions composables qui réagissent aux touchers écran et qui dessinent des formes géométriques.
2. Ouvrez `MainActivityView.kt`. La fonction composable `MainActivityView` affiche une colonne. Pour l'instant cette colonne ne contient qu'un appel à `DessinFigures`. C'est cette fonction qui dessine les figures. Plus tard il y aura deux barres d'outils dans la colonne. L'appel à `DessinFigures` est dans une `Box` parce qu'il faut que le dessin prenne toute la place possible (poids=1) en laissant le minimum aux barres d'outils.
   On s'occupera de `NouvelleFigureEtGrille` plus tard.

3. Ouvrez `DessinFigures.kt`. La fonction composable `DessinFigures` crée un « canevas ». C'est un espace permettant de dessiner. Il y a un système de coordonnées 2D, x, y. Le corps du `Canvas` (c'est à dire la lambda sans paramètre qui définit son contenu) contient un appel à `drawFigures` et c'est elle qui dessine effectivement chaque figure.
4. Ouvrez `Figure.kt`. C'est une classe abstraite très complexe. On va l'étudier morceau par morceau :
   a. Il y a un `enum` pour commencer. Chaque symbole représente un type de figure. Vous aurez à en rajouter de nouveaux dans le §5.
   b. Le `companion object` permet de définir des méthodes de classe (`static` en Java). Ici, il y a `create`.
   c. La méthode de classe `Figure.create(...)` retourne une nouvelle figure. On doit indiquer ce qu'on veut comme figure, par exemple `Type.RECTANGLE` et ses caractéristiques couleur, remplissage et les coordonnées de ses « coins ». Notez que `x2,y2` peuvent être absents.
   d. Une figure est définie par deux coins, par exemple haut-gauche et bas-droit pour un rectangle. Pour être précis, ce sont les deux points extrêmes que l'utilisateur touche en glissant le doigt sur l'écran, le point de départ et le point d'arrivée. Ce sont les quatre variables `x1,y1` et `x2,y2`.
   e. Une figure ne peut être dessinée que si les quatre points sont définis. C'est le rôle de la variable `isComplete`. C'est parce que quand on crée une nouvelle figure avec le doigt sur l'écran, le premier point est là où on a touché l'écran, et on n'a le second point qu'après un premier déplacement du doigt. Donc au début, pendant un court instant, on n'a pas les valeurs de `x2,y2` et la figure n'est pas complète.
   f. La méthode `draw` est abstraite. Elle doit être programmée dans toutes les sous-classes.
   g. Il y a plusieurs variables membres, deux dans les paramètres de la classe, `brush` et `fill`. Il y a aussi `style` et toutes celles présentées précédemment. Certaines sont initialisées dans le bloc d'initialisation `init`.
   h. Il y a deux autres méthodes, `copySetP2` qui recopie `this` en changeant le point `x2,y2` et `equals` qui compare deux figures. Cette dernière méthode est demandée par l'annotation `@Stable` posée sur la classe. Elle indique au système que les variables membres peuvent changer au cours du temps, mais d'une manière prévisible. Ça permet de réduire le nombre de recompositions si rien n'a changé.
5. Ouvrez `Rectangle.kt`. Il y a une seule méthode, `draw` qui reçoit un environnement de dessin en paramètre. Il provient du `Canvas` du point 3 précédent. Cet environnement offre beaucoup de possibilités, voir la doc (en français). Toutes les fonctions de dessin géométrique comme `drawRect` ont beaucoup de paramètres. Le premier obligatoire est une « brosse ». C'est ce qui donne les couleurs de remplissage et du contour de la forme. Ensuite, selon la figure, on a les coordonnées d'un coin ou du centre, et la taille. Ensuite on peut fournir un style pour remplir ou non l'intérieur.
   a. D'abord, si la figure n'est pas complète, on ne la dessine pas.
   b. Pour un rectangle, comme on a les coordonnées des « coins » mais que ces coins peuvent être n'importe quels opposés, on a un petit calcul pour avoir le coin haut-gauche dans `x,y` et la largeur et hauteur dans `w,h` pour que ça convienne à `drawScope.drawRect`. Ces variables sont mises dans deux types, `Offset` et `Size` lors de l'appel à la fonction.
   c. Enfin, il y a un appel à `drawScope.drawRect` avec tous les paramètres nécessaires pour

dessiner.

6. Ouvrez `AppState.kt`. C'est l'un des deux états dans cette application. Il stocke la liste des figures et définit des méthodes de transformation.
7. Ouvrez `UIState.kt`. C'est l'autre état, celui des préférences utilisateur, comme la couleur de dessin, le type de figures choisi, l'utilisation d'une grille, etc.

On a vu les classes essentielles, dans les grandes largeurs. On va commencer à programmer ce qui manque.

2. Dessin de figures existantes

👉 Ouvrez le source `DessinFigures.kt`, terminez la fonction `drawFigures`. Il manque une simple boucle pour appeler la méthode `draw` de chacune des figures de la liste qu'elle reçoit en paramètre. Relisez ce qui concerne les boucles sur une collection dans cette doc ou la référence .

👉 Lancez l'application. Vous devriez voir une petit dessin. Ce sont des figures initialisées dans `AppState`. L'idée, c'est que vous supprimerez ces figures quand vous aurez avancé dans le logiciel.

3. Ajout interactif de nouvelles figures

On attaque ce qui fait le cœur du logiciel, la possibilité de dessiner de nouvelles figures avec le doigt. Le principe est assez simple. Toucher l'écran déclenche trois sortes d'événements :

- `press` : l'utilisateur pose le doigt sur l'écran ⇒ début de la création d'une figure.
- `move` : l'utilisateur bouge son doigt à une autre position ⇒ modification de la figure.
- `release` : l'utilisateur soulève son doigt ⇒ fin de la figure, on l'ajoute à la liste.

Cela se représente avec un automate à deux états, figure 1.

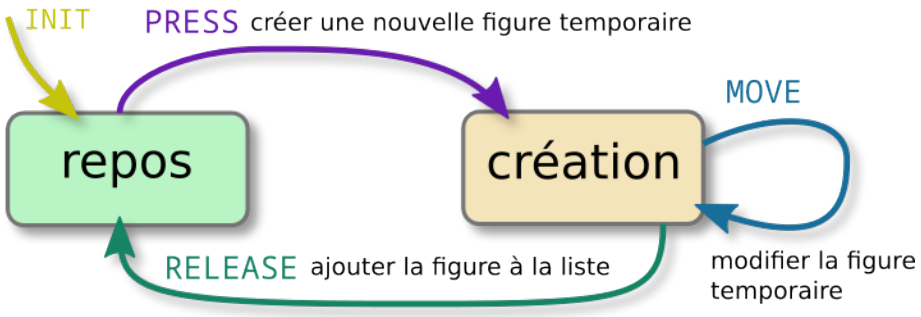

Figure 1: États de l'interaction utilisateur

La mise en œuvre est malheureusement extrêmement compliquée, pas naturelle en JetPack Compose. On est obligé de créer une sorte de calque au dessus du dessin. Ce calque affiche une seule figure, celle en cours de dessin s'il y en a.

Voici le principe de l'implémentation qui vous est fournie presque toute finie.

1. Dans `DessinFigures.kt`, il y a une fonction composable appelée `NouvelleFigureEtGrille`. Elle consiste en un `Canvas` qui demande au système de recevoir les événements écran, via un `modifier.pointerInput`. On les reçoit dans `event` et on les transmet à un écouteur appelé

`processTouchEvent`. Cet écouteur retourne la figure temporaire ou `null` quand c'est fini (doigt retiré de l'écran). S'il y a une figure temporaire, elle est dessinée.

2. L'écouteur `processTouchEvent` est fourni par `MainActivityView`, mais en fait, c'est un relais vers la méthode du même nom dans le *viewmodel*. La syntaxe `objet::methode` représente la méthode en tant que lambda appliquée sur l'objet.
3. Dans `FiguresViewModel.kt`, il y a la méthode `processTouchEvent`. C'est elle, l'écouteur des actions de l'utilisateur sur l'écran. Vous retrouvez un `when` avec les trois actions :
   a. Quand on touche l'écran, ça crée une nouvelle figure qui est retournée en résultat. Au final, cette figure est placée dans la variable `current` de `NouvelleFigureEtGrille`, et cette variable, si elle n'est pas nulle, est dessinée sur l'écran.
   b. Quand on bouge le doigt, ça modifie le point `P2` de la figure temporaire.
   c. Quand on soulève le doigt, l'état courant, `AppState` est modifié par l'ajout de la figure.

👉 Il manque la création de la figure temporaire, dans `FiguresViewModel.kt`, méthode `processTouchEvent` et dans le `when`. Cherchez des exemples de création de figure. Utilisez les informations de `uistate`. Ne mettez pas de coordonnées pour `x2,y2` ou alors les mêmes valeurs que pour `x1,y1`.

4. Mise en place d'une barre d'outils

Avant d'aller plus loin, on a besoin de sortes de menus pour configurer l'application : choisir le type de figure à dessiner, sa couleur, son remplissage, etc. Cette configuration se trouve dans `UIState`. Il y des booléens, une couleur et un type de figure. On voudrait pouvoir changer leur état pour paramétrer les figures et le plus facile consiste à créer une barre d'outils. C'est une `Row` contenant des boutons à cliquer. On va le construire progressivement.

Il y a un source dont on n'a pas encore parlé, `ToolbarButtons.kt`. C'est une sorte de bibliothèque de fonctions composables qui affichent différents types de boutons à placer dans une barre.

👉 Regardez la fonction `ToolbarButtonPreview`. Elle montre comment dessiner un bouton à cliquer, type `ToolbarButton`. On fournit l'identifiant d'une image qu'on appelle « drawable ».

👉 Dépliez le dossier `res` puis `drawable`. Vous allez voir de nombreux icônes, par exemple `ic_circle.png` et `ic_action_save`. Si vous avez un thème clair, vous ne verrez rien car ils sont dessinés en blanc sur transparent. Il faut sélectionner le thème sombre pour Android Studio pour voir ces images.

👉 Regardez la fonction `ToolbarToggleButtonPreview`. Elle montre comment dessiner un `ToolbarToggleButton`, c'est à dire un bouton à bascule. Il est gris clair ou foncé selon le booléen `active`.

👉 On a aussi la fonction `ToolbarButtonIcon` à laquelle on fournit une image un peu différente, prise dans la collection Material.

Ces trois fonctions demandent une lambda `onClick` pour définir quoi faire quand on les clique. C'est une lambda sans paramètre et sans résultat, `() -> Unit`, donc un simple bloc d'instructions. Voici deux exemples :

```
ToolbarButton(R.drawable.ic_icone) {
    // actions à faire, ex: appeler telle méthode
```

```
    telle_methode()
}
ToolbarToggleButton(R.drawable.ic_icone, active=btn_est_actif) {
    // actions à faire, ex: appeler telle autre méthode
    telle_autre_methode()
}
```

Dans notre application, les blocs d'actions consisteront à appeler un écouteur fourni aux barres d'outils.

On va construire deux barres d'outils, une à placer en haut de la fenêtre, l'autre en bas. Celle du haut permettra de choisir le type de figure, le mode de remplissage et la couleur. Celle du bas permettra d'effectuer différentes actions comme afficher une grille, annuler la dernière figure, remonter la figure la plus en dessous, enregistrer le dessin, etc.

4.1. Barre d'outils haute

Dans le fichier `MainActivityView.kt`, il y a la fonction `ToolbarHaute`. Ele n'est pas finie.

👉 Regardez ses paramètres. Ce sont à la fois des variables extraites de `UIState` et des lambda qui permettront de les modifier. Par exemple, `fill` vient de `UIState` et si on appelle `onToggleFill` ça devra inverser sa valeur.

👉 Ajoutez les boutons suivants, sachant qu'il y en aura d'autres après. Inspirez-vous du bouton déjà présent.

- un `Spacer(modifier = Modifier.size(30.dp))`
- un `ToolbarToggleButton` dont l'image est `ic_fill_or_not`, le paramètre `active` est égal à `fill` et son bloc appelle `onToggleFill`.
- un `ToolbarButton` dont l'image est `ic_action_go_to_today`. Son bloc doit appeler `onColorPick`.

👉 Ajoutez un appel à `ToolbarHaute` dans `MainActivityView`, au début de la colonne juste avant l'appel à `Box`, là où il y a un `TODO`. Vous devrez fournir les paramètres. Ils viennent tous de `UIState` et il y a trois lambda à écrire. Elles doivent appeler la bonne méthode du `viewmodel`. Rappel, quand une lambda n'a qu'un seul paramètre, il s'appelle `it`.

4.2. Barre d'outils basse

Dans le fichier `MainActivityView.kt`, il y a la fonction `ToolbarBasse`. Ses paramètres sont beaucoup de lambda, car cette barre porte des actions plutôt que des informations. Ici également, toutes les informations proviennent du `UIState` et toutes les lambda agissent dessus via le `viewmodel`.

👉 Ajoutez les boutons suivants :

- un `ToolbarToggleButton` avec l'image `ic_grid`, son paramètre `active` est égal à `grid`. Quand on clique dessus, ça appelle `onToggleGrid`.
- un `Spacer(modifier = Modifier.size(30.dp))`.
- un `ToolbarButton` d'icône `ic_action_undo` qui appelle `onUndo`.
- un `ToolbarButton` d'icône `ic_action_import_export` qui appelle `onSwap`.

- un `ToolbarButtonIcon(Icons.Default.Delete)` qui appelle `onClear`.
- un `ToolbarButton` d'icône `ic_action_save` qui appelle `onSave`.

👉 Ajoutez un appel à `ToolbarBasse` dans `MainActivityView`, dans la colonne juste après l'appel à `Box`, là où il y a un `TODO`.

5. Nouveaux types de figures

Vous allez maintenant définir de nouveaux types de figures : ligne, ellipse et cercle.

Pour chacun de ces types, il faut faire ceci :

1. Ajouter une constante à son nom dans le `enum class Type` de `Figure.kt`.
2. Ajouter un cas dans le `when` de la méthode `create`.
3. Ajouter une nouvelle classe (+fichier kotlin) dans le package `model`, cette classe doit hériter de `Figure`. Vous pouvez recopier `Rectangle` mais il faudra éditer la méthode `draw`.
4. Programmer la méthode `draw` en s'inspirant de celle de `Rectangle`.
5. Ajouter un bouton pour sélectionner ce type dans la barre d'outils du haut.
   a. Choisir son icône parmi les *drawable*.
   b. Son écouteur doit appliquer le type de la figure.
6. Tester l'application.

5.1. Dessin d'une ligne

Les points `x1,y1` et `x2,y2` sont les extrémités. Il faut consulter la documentation de `drawLine` ici pour savoir quels paramètres lui fournir, sous la forme de deux `Offset`.

5.2. Dessin d'un cercle

Ce sont les cercles les plus difficiles à dessiner. La fonction `drawCircle` (doc) demande un centre et un rayon. Le centre (xc,yc) est le milieu entre (x1,y1) et (x2,y2). C'est le rayon qui est le plus pénible à déterminer. C'est la plus petite distance entre (xc,yc) et les bords. Par exemple *max(xc-min(x1,x2), yc-min(y1,y2))*, mais c'est à vous d'y réfléchir.

Sinon, il y a une autre possibilité, c'est de calculer le rayon comme étant la distance euclidienne entre (xc,yc) et (x2,y2). Dans ce cas, le cercle sort de la zone (x1,y1), (x2,y2) mais il touche le point (x2,y2) qui est défini par la position du doigt sur l'écran. C'est sans doute le plus naturel.

Rappel mathématique : $distance((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

5.3. Dessin d'une ellipse

Vous allez devoir réfléchir à la manière de dessiner une ellipse. Quelle signification allez-vous donner au premier toucher sur l'écran, et à l'endroit où on soulève le doigt ?

Consultez la documentation de `drawOval` ici .

6. Actions diverses

À ce stade, le bouton `fill` de la barre d'outil doit fonctionner. Le remplissage concerne toutes les figures qui ont une étendue.

☛ Terminez la programmation des boutons `undo`, `swap` et `clear`. On s'occupera de `grid`, `save` et du sélecteur de couleur dans la suite.

Vous avez seulement à compléter les méthodes du `ViewModel` et `AppState`.

☛ Testez l'application.

7. Grille

Une grille permet de bloquer les coordonnées `x,y` à des valeurs multiples d'une quantité appelée le *pas de la grille*. Par exemple, si le pas est de 12, alors les positions des figures seront bloquées sur des multiples de 12. L'intérêt est de pouvoir aligner différentes figures ou mieux partager l'écran.

C'est le booléen `grid` de `UIState` qui gère cet aspect. Il active ou désactive des traitements.

Il y a deux traitements si la grille est active, d'abord il faut la dessiner sur l'écran, et ensuite, il faut modifier les coordonnées des touchers écran.

7.1. Dessin de la grille

☛ Ouvrez `DessinFigures.kt` et allez sur `NouvelleFigureEtGrille`.

Il y a un `TODO` concernant la grille. Le but est de dessiner des petits points espacés du pas de la grille. Pour cela, il faut utiliser la variable appelée `inside`. C'est un rectangle qui donne l'étendue de la zone écran. La variable `inside` possède plusieurs *getters*, `top`, `left`, `bottom`, `right`, `width` et `height`. Ce sont tous de `Float`, et il faudra convertir ceux qu'on utilise en entier par `toInt()`.

Il va y avoir deux boucles imbriquées, l'une sur x allant de 0 à `inside.width.toInt()` et l'autre sur y allant de 0 à `inside.height.toInt()`. C'est un petit peu bizarre parce qu'il faut impérativement commencer à 0 et aller jusqu'au bord opposé de l'écran.

Alors la particularité de ces boucles, c'est qu'il faut sauter selon le pas de la grille. Ce pas est dans la variable `gridStridePix`. En Kotlin, pour faire une boucle de PAS en PAS, on écrit `for (v in debut .. fin step PAS)`.

Ensuite, à chaque tour de ces boucles, il faut dessiner un petit cercle : `drawCircle(Color.Gray, 2f, Offset(x.toFloat(), y.toFloat()))`. Un cercle est plus joli qu'un point.

☛ Programmez les deux boucles.

☛ Vérifiez que le bouton grille de la barre d'outils active/désactive l'affichage de la grille.

☛ Testez l'application. Pour l'instant, la grille est affichée, mais elle n'a aucun effet quand on dessine.

7.2. Effet de la grille

C'est dans la méthode `processTouchEvent` de `FigureViewModel` qu'on prend en compte la grille. Ça aurait été préférable de le faire dans l'écouteur des touchers écran, mais cet écouteur est dans une *fermeture* et il ne réagit pas aux réaffectations des variables comme `grid` ou autres.

Au début de la fonction `processTouchEvent`, on extrait `x` et `y` de la position passée en paramètre. Il y a un calcul à faire quand `grid` est vrai et `gridStride` strictement positif. Ce calcul vise à rendre `x` et `y` multiples de `gridStride`.

Soit le nombre 100 à rendre multiple de 12. C'est à dire qu'il faut trouver le multiple de 12 le plus proche de 100. Comment allez-vous faire ? Si vous divisez 100 par 12 et que vous arrondissez à l'entier le plus proche puis multipliez cet entier par 12, vous trouverez le résultat.

☛ Programmez le calcul qui arrondit les coordonnées `x` et `y` à leur plus proche multiple de `gridStride`. Et placez ce calcul dans une conditionnelle de manière à le le faire que quand `grid` est vrai et `gridStride` strictement positif.

☛ Testez la grille dans l'application.

8. Enregistrement du dessin

Le bouton d'enregistrement, en bas à droite, doit démarrer une séquence où on demande à l'utilisateur de choisir un nom de fichier PNG et ensuite le dessin est enregistré dedans. Le principe, à cause du mode de fonctionnement de Compose, consiste à avoir un écouteur dans `MainActivity` qui ouvre le dialogue. Lorsque l'utilisateur valide son choix, ça affecte la variable `saveUri` de `UIState`. Au prochain dessin, si cette variable est non `null`, alors l'enregistrement est effectué par la fonction `saveImage` dans `DessinFigures.kt` et la variable retourne à `null`.

On ne peut pas faire directement parce que Compose est fondé sur l'observation des changements d'un état.

C'est donc très compliqué à faire, mais tout est préparé d'avance pour vous. Vous avez seulement à définir le paramètre `onSave` de l'appel à `MainActivityView` dans `MainActivity`. Ce paramètre est un bloc qui appelle `onSelectFileToSave`. C'est tout.

☛ Testez l'enregistrement dans l'application.

9. Choix de la couleur d'une figure

Encore quelque chose d'assez compliqué. Vous allez en faire une partie, parce que ça ressemble au dialogue de saisie d'un message du TP5.

9.1. Dialogue de choix d'une couleur

Comme dans le TP5, on va le construire par étapes.

☛ Ouvrez `ColorPickerDialog.kt` et allez tout en bas du fichier.

Il y a la fonction `ColorSlider` et sa prévisualisation. Cette fonction dessine une rangée horizontale comprenant un carré de couleur, un curseur et une zone de texte pour afficher la valeur du curseur.

Il y a une variable d'état interne `sliderPosition`. C'est la valeur du curseur qui est partagée avec la zone de texte.

👉 Trouvez l'écouteur du curseur et comprenez que ce qu'il fait conduit à modifier la valeur affichée dans la zone de texte. Et en plus il y a un paramètre de `ColorSlider` qui est une lambda et qui est appelée à chaque changement du curseur.

👉 Remontez au début du fichier, vers la fonction `ColorPickerDialog`.

Cette fonction doit afficher 4 curseurs, un pour choisir la composante rouge, un pour le vert, un pour le bleu et un dernier pour la transparence qui s'appelle *alpha*. Actuellement, il n'y a qu'un seul curseur. Il y a aussi deux boutons dans le bas, pour annuler ou valider la couleur. Ce dernier

👉 Ajoutez les variables d'état internes manquantes et les trois curseurs nécessaires.

9.2. Apparition du dialogue

👉 Ouvrez `MainActivityView.kt` et la fonction `MainActivityView`.

Il faut mettre en place le même genre de structure d'affichage d'un dialogue que dans le TP5. Il faut une variable, par exemple `isColorPickerDialogDisplayed` initialisée à faux. Elle passe à vrai quand on touche le bouton de choix de couleur, et revient à faux quand le dialogue doit disparaître.

👉 Définissez la variable d'état interne `isColorPickerDialogDisplayed`.

👉 Définissez l'écouteur `onColorPick` dans la barre d'outils haute; C'est une lambda sans paramètre qui rend `isColorPickerDialogDisplayed` vraie.

👉 Dans `MainActivityView`, après la définition de la variable, programmez une conditionnelle pour que quand cette variable est vraie, ça affiche un `ColorPickerDialog`.

Ce dialogue demande trois paramètres. Le premier est la couleur actuelle. Allez la chercher dans `uistate`.

Les deux autres paramètres sont des lambdas qui doivent remettre `isColorPickerDialogDisplayed` à faux. En plus `onAccept` doit modifier la couleur dans le `viewmodel`. Cette couleur arrive en paramètre à la lambda (c'est `it`).

👉 Le dialogue doit apparaître quand on clique sur le bouton. Il doit disparaître si on clique sur annuler et valider. Quand on clique valider, la nouvelle couleur doit être prise en compte.

10. Travail à rendre

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

Avec le navigateur de fichiers, descendez dans le dossier `app/src` du projet *XMen*. Cliquez droit sur le dossier `main`, compressez-le au format zip. Déposez l'archive `main.zip` sur Moodle au bon endroit, remise du TP6 sur la page Moodle R4.A11 Développement Mobile.

Rajoutez un fichier appelé exactement `IMPORTANT.txt` dans l'archive, 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.