HELHa
Tournai
Bachelor en Informatique

2024 - 2025
Android

Développement Android

Clarifier l'interface via des fragments

Version du document : 29 septembre 2024

Ce TD est étroitement inspiré par le livre "Android Programming, the big nerd ranch guide", 3è edition, par Bill Phililips, et Al. et en particulier par les chapitres 7 à 11.

☞ Aide

Suivez le texte ci-dessous. Si vous êtes perdu ou si vous voulez plus d'informations, vous pouvez suivre la vidéo d'explications sur ce labo disponible ici :

https://youtu.be/HfpMg0hXzCI

Le code résultat de ce labo est disponible ici :

https://github.com/fpluquet/ue3103-android-labo3

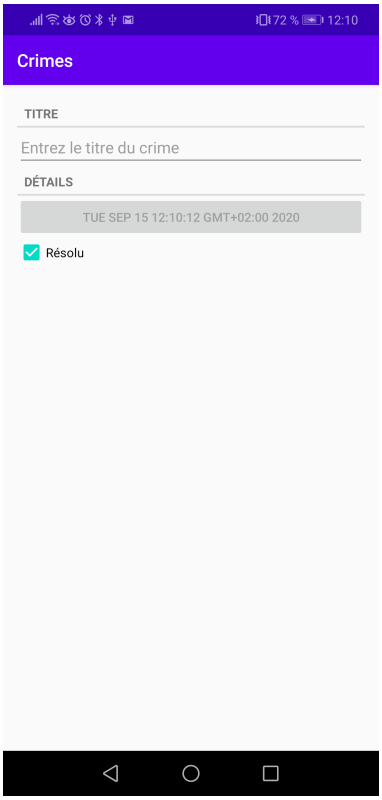
1 Utilité des fragments

Le concept d'activité permet de faire des applications simples, mais dès que l'on veut faire une application comportant plusieurs écrans possédant des liens entre eux, par exemple une application *master/detail*, la notion d'activité atteint rapidement ses limites.

La notion de *fragment* a été introduite avec l'introduction des premières tablettes Android, lorsque le besoin d'interfaces plus élaborées s'est fait sentir. Un fragment est un *morceau* d'interface, un layout et son contrôleur. Un écran est défini par une activité qui elle-même est composée d'un ou plusieurs fragments. Les fragments peuvent être manipulés, retirés ou ajoutés. Un fragment peut également être composé d'autres fragments.

2 Nouvelle application : CriminalIntent

Pour illustrer la notion de fragment, nous allons développer une nouvelle application : CriminalIntent. Cette application permettra de visualiser des *crimes* et, dans un deuxième temps, de les parcourir. Le premier écran, très simple, permettra d'illustrer l'utilisation des fragments.



2.1 Création du projet

Créez un nouveau projet *Criminal Intent* avec une `Empty Views Activity`. Attendez que tout soit complètement chargé (allez faire un jogging, une lessive ou boire un café).

Pour s'y retrouver plus facilement dans nos fichiers ensuite, renommez l'activité principale de `MainActivity` en `CrimeActivity` via du refactoring (clic droit, *Refactor*, *Rename*). Renommez également le layout de `activity_main.xml` en `activity_crime.xml` via du refactoring. Faites tourner l'application et vérifiez que cela compile et se lance correctement.

Définissez immédiatement les constantes textuelles du projet dans le fichier `strings.xml` :

```
<resources>
    <string name="app_name">Crimes</string>
    <string name="crime_title_hint">Entrez le titre du crime</string>
    <string name="crime_title_label">Titre</string>
    <string name="crime_details_label">Détails</string>
    <string name="crime_solved_label">Résolu</string>
</resources>
```

2.2 Création du modèle : Crime.java

Créez une première classe du modèle, la classe `Crime`, qui a comme attributs :

— `mId` un identifiant unique de type `java.util.UUID`

— `mTitle` un titre de type `String`

— `mDate` de type `java.util.Date`

— `mSolved` un booléen indiquant si le cas est résolu ou non.

Ajoutez un constructeur qui initialise l'id (`mId = UUID.randomUUID();`) et la date à la date du jour.

Générez les accesseurs pour tous les attributs (getters) et les setters pour les 3 derniers.

Refatorisez vos fichiers pour que ce modèle soit dans le package `models` et le contrôleur dans un package `controllers`.

2.3 Création de l'activité hébergeante

Pour héberger un fragment, une activité doit faire 2 choses :

— définir un endroit de son layout où placer le fragment

— gérer le cycle de vie du fragment.

En effet, tout comme une activité, un fragment est régi par un cycle de vie précis. Ce cycle de vie est très similaire au cycle de vie d'une activité et sera dirigé par l'activité qui l'héberge. On retrouve essentiellement les mêmes méthodes permettant d'intervenir lors des changements d'état du fragment : `onCreate, onPause, onResume, onStop,` etc.

Modifiez le fichier de layout `activity_crime.xml` comme suit :

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/fragment_container"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

Ce layout sera réservé pour l'hébergement du fragment.

2.4 Création d'un fragment

La création d'un fragment se fait en 3 étapes :

— création de la vue dans un fichier XML ;

— définition du contrôleur en Java ;

— lier la vue et le contrôleur dans le code Java.

2.4.1 Création de la vue

Dans le répertoire `res/layout` de la vue projet, faites *New* → *Layout resource file*. Nommez ce fichier `fragment_crime.xml` et indiquez `LinearLayout` comme élément racine.

Modifiez ce fichier comme suit :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_margin="16dp">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        style="?android:listSeparatorTextViewStyle"
        android:text="@string/crime_title_label"/>
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/crime_title"
        android:hint="@string/crime_title_hint"/>
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        style="?android:listSeparatorTextViewStyle"
        android:text="@string/crime_details_label"/>
    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/crime_date"/>
    <CheckBox
        android:id="@+id/crime_solved"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/crime_solved_label"/>
</LinearLayout>
```

Vous pouvez visualiser le layout en mode *design*. Regardez à quoi correspond chaque entrée de ce fichier.

Question 1

Que signifie le `?` dans `style="?android:listSeparatorTextViewStyle"` ? [a]

a. Cherchez dans la documentation d'Android en ligne pour trouver la réponse.

2.4.2 Création du contrôleur

Dans le package `controllers`, ajoutez une nouvelle classe `CrimeFragment` qui sera le contrôleur du fragment.

Faites de cette classe une sous-classe de `androidx.fragment.app.Fragment`.

Ajoutez un attribut `mCrime` de type `Crime` et redéfinissez la méthode `onCreate` afin d'initialiser le modèle :

```
public class CrimeFragment extends Fragment {
    protected Crime mCrime;

    @Override
```

```
    public void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mCrime = new Crime();
    }
}
```

2.4.3 Lier le contrôleur et la vue

Finalement, vous pouvez lier le contrôleur et la vue, c'est-à-dire récupérer une référence aux widgets de la vue et initialiser les listeners. Cela se fait par redéfinition de la méthode `onCreateView`[1] :

1. Pensez à importer les bonnes classes et à créer les attributs nécessaires (`mTextField`, ...) dans votre classe.

```
//(...)

private EditText mTitleField;
private Button mDateButton;
private CheckBox mSolvedCheckBox;

public View onCreateView(LayoutInflater inflater, ViewGroup container,
Bundle savedInstanceState) {
    // inflate the fragment_crime view
    View v = inflater.inflate(R.layout.fragment_crime,container,false);

    // configure the view
    mTitleField = (EditText) v.findViewById(R.id.crime_title);
    mTitleField.addTextChangedListener(new TextWatcher() {
        @Override
        public void beforeTextChanged(CharSequence s, int start, int count,
            int after) {
            // do nothing
        }
        @Override
        public void onTextChanged(CharSequence s, int start, int before,
            int count) {
            mCrime.setTitle(s.toString());
        }
        @Override
        public void afterTextChanged(Editable s) {
            // do nothing, everything's done in onTextChanged
        }
    });

    mTitleField.setText(mCrime.getTitle());

    mDatebutton = (Button) v.findViewById(R.id.crime_date);
    mDatebutton.setText(mCrime.getDate().toString());
    mDatebutton.setEnabled(false); //readonly

    mSolvedCheckBox = (CheckBox) v.findViewById(R.id.crime_solved);
    mSolvedCheckBox.setChecked(mCrime.isSolved());
    mSolvedCheckBox.setOnCheckedChangeListener(
        new CompoundButton.OnCheckedChangeListener() {
            @Override
            public void onCheckedChanged(CompoundButton buttonView, boolean
                isChecked) {
                mCrime.setSolved(isChecked);
            }
        }
    );

    return v;
}
```

Quelques explications sur ce code :

— ligne 1 : un `inflater` est l'objet qui permet de passer d'un fichier XML à des objets Java.

— ligne 4 : on demande de transformer notre fichier XML avec l'identifiant `R.layout.fragment_crime` en objets Java. L'objet racine est renvoyé, c'est-à-dire la vue englobant le reste des widgets. Ici c'est le `LinearLayout` défini dans ce fichier.

— lignes 7 à 21 : on retrouve le `EditText` (un champ éditable) pour le titre et met à jour le modèle à chaque changement.

— ligne 23 : on assigne la valeur courante du crime dans l'`EditText`.

— lignes 25 à 27 : on retrouve le champ pour la date et on l'initialise avec la date courante.

— lignes 29 à 38 : on retrouve la `CheckBox` qui indique si le crime est résolu ou non. On l'initialise avec le modèle et on ajoute le listener pour modifier le modèle à chaque modification.

Question 2

1. Quelle est la classe qui permet d'écouter un `TextView` ?
2. Quelle est la classe qui permet d'écouter un `Checkbox` ?

Vous pouvez faire tourner votre application mais vous ne verrez aucun changement : le fragment n'est pas encore utilisé dans l'activité principale.

2.5 Fragment Manager

Votre fragment est donc défini et est prêt à l'emploi. Nous allons maintenant l'ajouter à la `CrimeActivity`. La gestion des fragments, de leurs ajouts ou de leurs retraits de la vue se fait via le `FragmentManager`.

Ajoutez à la méthode `onCreate` de la classe `CrimeActivity` le code suivant :

```
FragmentManager fm = getSupportFragmentManager();
Fragment fragment = fm.findFragmentById(R.id.fragment_container);
if(fragment==null) {
    fragment = new CrimeFragment();
    fm.beginTransaction()
    .add(R.id.fragment_container, fragment)
    .commit();
}
```

— ligne 1 : on demande le `FragmentManager` courant.

— ligne 2 : on demande de retrouver le fragment hébergé dans l'activité courante dans une vue avec l'identifiant `R.id.fragment_container` (définie dans le fichier `activity_crime.xml`).

— ligne 3 : s'il n'y a pas encore de fragment chargé...

— ligne 4 : ... on en crée un,

— ligne 5 à 7 : et on l'ajoute en précisant l'identifiant du container hébergeur et le fragment à ajouter.

Nous voyons que le `FragmentManager` utilise des transactions afin de mettre à jour la vue. Cela permet d'éviter les appels concurrents et de stabiliser les changements d'interfaces. L'appel à `commit` va rendre effectif les changements.

Votre écran est maintenant complet. Faites tourner votre application et vérifiez que cela fonctionne correctement.

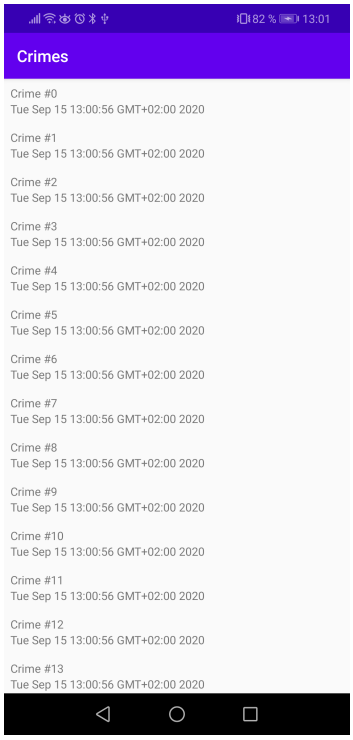
2.6 Conclusion sur les fragments

Avec le code vu précédemment, vous avez tout en votre possession pour gérer des fragments dans votre application. Vous pouvez imaginer des layouts composés d'autres layouts (par exemple, un à droite, un autre à gauche et un autre dans tout le bas).

Cela permet de rendre l'interface de votre application dynamique, adaptable à n'importe quel taille d'écran, ...

3 Afficher une liste

Maintenant que les fragments sont compris, nous allons implémenter un écran plus complexe permettant d'afficher une liste de *crimes*, utilisant des fragments.



Nous allons montrer deux possibilités d'implémentation de cette liste : la construction manuelle ou la gestion de la liste par une `RecyclerView`. La construction manuelle a l'avantage d'être simple à comprendre mais peu efficace lors de grandes listes à afficher alors que la seconde est plus complexe à comprendre mais très efficace lors de grandes listes à afficher.

3.1 Mise à jour du modèle

Nous commençons par mettre à jour le modèle en ajoutant la classe `CrimeLab` qui est essentiellement une liste de *crimes*. Cette classe utilise le patron de conception *singleton* :

```
public class CrimeLab {
    private static CrimeLab sCrimeLab;

    private List<Crime> mCrimes;

    public static CrimeLab get() {
        if(sCrimeLab == null) {
                sCrimeLab = new CrimeLab();
            }
        return sCrimeLab;
    }

    private CrimeLab() {
        mCrimes = new ArrayList<>();

        // initialisation avec des crimes bidons.
        for(int i = 0; i < 100; i++) {
            Crime crime = new Crime();
            crime.setTitle("Crime #" + i);
            crime.setSolved(i%2==0); // un sur deux résolu
            mCrimes.add(crime);
        }
    }

    public Crime getCrime(UUID id) {
        for (Crime crime : mCrimes) {
            if(crime.getId().equals(id))
                return crime;
        }
        return null;
    }

    public List<Crime> getCrimes() {
        return Collections.unmodifiableList(mCrimes);
    }
}
```

Lisez ce code et soyez sûr de comprendre chaque ligne.

Question 3

- Pourquoi a-t-on préfixé d'un `s` l'attribut `sCrimeLab` ?
- Pourquoi fait-on `return Collections.unmodifiableList(mCrimes);` ?

4 Afficher une liste : construction manuelle

L'idée ici est de construire de manière programmatique les différents éléments de la liste et de les ajouter.

4.1 Ajout de l'activité principale : `ManualCrimeListActivity`

Commençons par créer une nouvelle activité `ManualCrimeListActivity`, avec un layout nommé `activity_manual_crime_list.xml`. Le code de ce layout sera le suivant :

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ScrollView
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <LinearLayout
            android:id="@+id/crime_list"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:orientation="vertical" />
    </ScrollView>
</androidx.constraintlayout.widget.ConstraintLayout>
```

On peut voir ici qu'on a un `ConstraintLayout` (qui va garder l'ensemble collé sur les bords de l'écran) qui contient une `ScrollView` (qui permet de scroller), qui contient elle-même un `LinearLayout` avec un identifiant `crime_list` qui acceuillera nos différents crimes.

Nous pouvons désormais nous concentrer sur l'activité `ManualCrimeListActivity` :

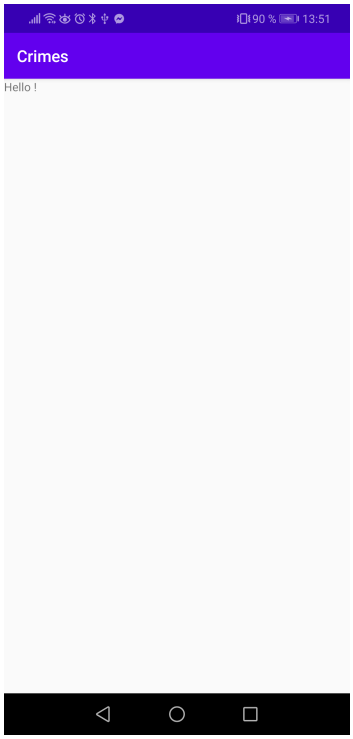
```
public class ManualCrimeListActivity extends AppCompatActivity {
    private LinearLayout mContainer;

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_manual_crime_list);
        mContainer = (LinearLayout) findViewById(R.id.crime_list);
        TextView textView = new TextView(getApplicationContext());
        textView.setText("Hello !");
        mContainer.addView(textView);
    }
}
```

Il faut également mettre à jour le *manifest* de l'application (`manifests/AndroidManifest.xml`) afin de déclarer cette nouvelle activité comme le point d'entrée de votre application à la place de `CrimeActivity` :

```
(...)
<activity android:name=".controllers.ManualCrimeListActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
<activity android:name=".controllers.CrimeActivity">
</activity>
(...)
```

Vous pouvez lancer votre application et vous devriez voir ceci :



En effet, dans l'activité `ManualCrimeListActivity` :

- ligne 7 : on transforme l'XML en objets Java.
- ligne 8 : on récupère le `LinearLayout` avec l'identifiant `crime_list`.
- lignes 9 et 10 : on crée un `TextView` avec le texte `"Hello !"`.
- ligne 11 : on ajoute ce `TextView` comme enfant du `LinearLayout`.

4.2 Ajout des éléments dans la liste

On peut donc aller plus loin et créer un élément pour chaque crime que l'on ajoute dans la liste :

```java
public class ManualCrimeListActivity extends AppCompatActivity {
    private LinearLayout mContainer;

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_manual_crime_list);
        mContainer = (LinearLayout) findViewById(R.id.crime_list);
        updateUI();
    }

    private void updateUI() {
        mContainer.removeAllViews();
        CrimeLab lab = CrimeLab.get();
        for (final Crime crime : lab.getCrimes()) {
            View crimeView = getCrimeView(crime);
            mContainer.addView(crimeView);
        }
    }

    private View getCrimeView(Crime crime) {
        TextView textView = new TextView(getApplicationContext());
        textView.setText(crime.getTitle());
        return textView;
    }

}
```

Vous devriez avoir le résultat suivant :

4.3 Personnalisation des éléments

Comme nous voudrions voir le titre et la date par crime et laisser un peu d'espace (*padding*) entre les éléments de la liste, nous adaptons le code de la méthode `getCrimeView` comme ceci :

```
private View getCrimeView(final Crime crime) {
    // création d'un LinearLayout vertical avec padding de 8
    LinearLayout columnForCrime = new LinearLayout(getApplicationContext());
    columnForCrime.setOrientation(LinearLayout.VERTICAL);
    columnForCrime.setPadding(8, 8, 8, 8);

    // création des TextViews
    TextView titleView = getTextView(crime.getTitle());
    TextView dateView = getTextView(crime.getDate().toString());

    // ajout des TextViews dans le LinearLayout
    columnForCrime.addView(titleView);
    columnForCrime.addView(dateView);
    return columnForCrime;
}
/*
* Renvoie un TextView avec le texte text
*/
private TextView getTextView(String text) {
    TextView textView = new TextView(getApplicationContext());
    textView.setText(text);
    textView.setLayoutParams(new FrameLayout.LayoutParams(
        LinearLayout.LayoutParams.MATCH_PARENT,
        LinearLayout.LayoutParams.WRAP_CONTENT));
    return textView;
}
```

Nous avons désormais l'affichage voulu de la liste ! Félicitations.

4.4 Passage de paramètres vers un fragment

Dans cette section, nous allons lancer une vue détaillée concernant un crime lorsque celui-ci est cliqué dans la liste.

Pour cela, il suffit d'ajouter un listener pour l'événement correspondant sur la vue d'un crime dans la liste :

```
private View getCrimeView(final Crime crime) {
    (...)
    setClickOnCrimeView(crime, columnForCrime);
    return columnForCrime;
}
private void setClickOnCrimeView(final Crime crime, View columnForCrime) {
    columnForCrime.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Intent intent = new Intent(getApplicationContext(),
                CrimeActivity.class);
            intent.putExtra(CrimeFragment.CRIME_ID, crime.getId());
            startActivity(intent);
        }
    });
}
```

Pensez à créer la constante `CRIME_ID` dans la classe `CrimeFragment` (ayant comme valeur un string unique, comme `"CRIME_ID"`). Si vous lancez l'application, vous verrez que, si vous cliquez sur un crime, l'activité `CrimeActivity` est bien chargée mais les informations éditables ne sont pas les bonnes.

Nous allons maintenant lier le formulaire de la `CrimeActivity` au crime cliqué. Remarquez qu'à la ligne 11, on passe à cette activité un extra : l'identifiant du crime cliqué.

Il reste à mettre à jour `CrimeActivity` afin d'utiliser cet extra pour afficher le crime en question. Dans la classe `CrimeActivity`, on ne fait que charger un `CrimeFragment`. Nous allons alors plutôt changer la méthode `onCreate` de la classe `CrimeFragment` :

```
@Override
public void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    //mCrime = new Crime();
    UUID crime_id = (UUID)
        getActivity().getIntent().getSerializableExtra(CRIME_ID);
    mCrime = CrimeLab.get().getCrime(crime_id);
}
```

Cela devrait maintenant fonctionner : les données du crime sélectionné sont chargées dans le formulaire. Mais si on les modifie et qu'on repasse sur la première activité via le bouton *Back*, les données ne sont pas mises à jour. Pour cela, il faut se souvenir que la méthode `onResume` est appelée quand on revient sur une activité. On peut dès lors demander de rafraîchir la liste en appelant `updateUI()` :

```
@Override
protected void onResume() {
    super.onResume();
    updateUI();
}
```

L'application devrait désormais correctement : affichage de la liste, détail d'un crime, modification d'un crime et mise à jour de la liste.

4.5 Séparation du code : utilisation d'un layout

Nous voyons que cette technique fonctionne correctement mais est assez verbeuse afin de créer les différents éléments de la vue (comme le padding, ...). Nous pouvons alors séparer la vue du contrôleur, comme nous allons déjà fait, en utilisant du XML.

Ajoutez d'abord un nouveau layout : `list_item_crime.xml`. Il s'agit d'un layout pour visualiser un élément de la liste. C'est ce layout qui sera répété autant de fois qu'il y a d'éléments visibles à l'écran.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="8dp">

    <TextView
        android:id="@+id/crime_title"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Crime Title"/>

    <TextView
        android:id="@+id/crime_date"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Crime Date"/>
</LinearLayout>
```

Nous pouvons alors adapter la classe `ManualCrimeListActivity` afin de charger ce layout, de reprendre les différents widgets, d'y mettre les bonnes valeurs et de les ajouter à la liste :

```
private View getCrimeView(Crime crime) {
    View columnForCrime =
        getLayoutInflater().inflate(R.layout.list_item_crime, null);
    ((TextView) columnForCrime.findViewById(R.id.crime_title))
        .setText(crime.getTitle());
    ((TextView) columnForCrime.findViewById(R.id.crime_date))
        .setText(crime.getDate().toString());
    setClickOnCrimeView(crime, columnForCrime);
    return columnForCrime;
}
```

Le code est donc plus maintenable : les layouts sont modifiables séparemment des contrôleurs.

5 Exercices

1. Faites apparaître dans la liste le fait qu'un crime soit résolu ou non. Par exemple avec une couleur de fond vert lorsque c'est résolu et rouge lorsque cela ne l'est pas, ou avec une image.
2. Ajoutez *une gravité* comme attribut à la classe `Crime` : un crime peut être Mild (léger), Moderate (moyen), Severe (sérieux). Faites apparaître cela dans la liste (`CrimeListFragment`) et dans le détail `CrimeFragment`. Faites en sorte que la gravité soit éditable dans la vue détail.
3. Gérer les différentes tailles d'écran : sur un grand écran, on devrait pouvoir voir la liste et le détail en même temps.
4. Améliorez l'affichage de toute l'application.

Bonus : Initialisez la liste de crimes à partir d'un fichier en utilisant des `Serializable`. Enregistrez automatique toutes les modifications dans ce fichier.

Important !

Cette partie qui suit est facultative pour l'année 2024-2025. Je la laisse pour ceux qui pourraient s'y intéresser :)

6 Afficher une liste : via un `RecyclerView`

Comme nous venons de le voir, il est possible de gérer une liste (ou tout autre widget) manuellement (de façon programmatique). Mais que se passe-t-il si on met 10000 crimes au lieu de 100 dans `CrimeLab` ? Essayez, lancez et scrollez. Cela devrait lagger plus ou moins fort. Ici les 10000 éléments sont créés, doivent être calculés mais seuls quelqu'uns sont affichés (une dizaine). C'est complètement inefficace.

Nous allons montrer maintenant comment obtenir une application fluide, même avec beaucoup de données, en utilisant un `RecyclerView`. L'interêt de l'objet `RecyclerView` est donc qu'il permet d'afficher et de parcourir (*scroll*) une liste d'élément qui peut être très très longue, tout en utilisant, grâce à un mécanisme de recyclage, un minimum de ressources.

6.1 Refactoring : hébergeur de fragments générique

Avant d'aller plus loin, comme nous allons avoir besoin d'une activité avec un seul fragment comme enfant et que nous avons déjà ce cas-là avec l'activité `CrimeActivity`, nous allons *refactoriser* le code afin de pouvoir créer plus facilement des (activités) hébergeurs de fragments.

Le layout `activity_crime.xml`, associé à la classe `CrimeActivity`, est déjà entièrement générique. Renommez-le `activity_fragment.xml` (via le refactoring).

Question 4

Outre le nom du fichier, quels sont le/les changements effectués par le refactoring (renommage du fichier XML) que vous venez de faire ?

Dans les contrôleurs, créez la classe abstraite `SingleFragmentActivity` comme suit :

```
public abstract class SingleFragmentActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_fragment);

        FragmentManager fm = getSupportFragmentManager();
        Fragment fragment = fm.findFragmentById(R.id.fragment_container);
        if(fragment==null) {
                fragment = createFragment();
                fm.beginTransaction()
                .add(R.id.fragment_container, fragment)
```

```
                .commit();
        }
    }
    protected abstract Fragment createFragment() ;
}
```

Cette classe reprend la logique générique présente dans la classe `CrimeActivity` :

— appeler la méthode `onCreate` de la super-classe ;

— définir le vue ;

— placer le fragment.

La seule partie non générique est l'instanciation du fragment qui est déléguée à la méthode abstraite : `createFragment`.

Vous pouvez maintenant grandement simplifier la classe `CrimeActivity` en la déclarant comme sous-classe de votre nouvelle classe abstraite :

```
public class CrimeActivity extends SingleFragmentActivity {
    @Override
    protected Fragment createFragment() {
        return new CrimeFragment();
    }
}
```

Vérifiez que votre application est toujours fonctionnelle. Soyez sûr d'avoir bien compris la refactorisation du code que nous venons de faire.

6.2 Ajout de l'activité principale : `CrimeListActivity`

Dans les contrôleurs, créez le fragment qui contiendra la liste : `CrimeListFragment`. Faites-en une sous-classe de `Fragment` [2]. Pour le moment, cette classe ne contient aucun code, mais elle est nécessaire pour pouvoir écrire l'activité.

Toujours dans les contrôleurs, créez une nouvelle classe `CrimeListActivity`, l'activité qui va héberger le fragment que vous venez de déclarer. Grâce au refactoring, cette activité est extrêmement simple :

```
public class CrimeListActivity extends SingleFragmentActivity {
    @Override
    protected Fragment createFragment() {
        return new CrimeListFragment();
    }
}
```

Il faut également mettre à jour le *manifest* de l'application (`manifests/AndroidManifest.xml`) afin de déclarer cette nouvelle activité comme le point d'entrée de votre application à la place de `ManualCrimeListActivity` :

2. `androidx.fragment.app.Fragment`

```
(...)
<activity android:name=".controllers.CrimeListActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
<activity android:name=".controllers.ManualCrimeListActivity">
</activity>
<activity android:name=".controllers.CrimeActivity">
</activity>
(...)
```

Vous pouvez lancer votre application, mis à part le titre, l'écran est vide.

6.3 RecyclerView

Vous allez maintenant mettre en place le fragment qui gère la liste. Pour cela, il faut d'abord ajouter une dépendance à votre projet. Allez dans *File* > *Project Structure* et sélectionnez *Dependencies* > *app* (à gauche). Utilisez alors le bouton `+` pour ajouter la dépendance :

`androidx.recyclerview:recyclerview:1.2.1`.

La version est peut-être légèrement différente chez vous. Attendez que tout soit chargé avant de continuer.

Vous pouvez maintenant ajouter un layout pour la liste. Dans `res/layout`, ajoutez un nouveau layout (*New* > *Layout resource file*) que vous nommez `crime_list_fragment.xml`.

Choisissez `androidx.recyclerview.widget.RecyclerView` comme élément racine. Nous allons attribuer l'identifiant `crime_recycler_view` à cet élément racine.

Vous devriez donc avoir le fichier suivant :

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.recyclerview.widget.RecyclerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/crime_recycler_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
</androidx.recyclerview.widget.RecyclerView>
```

Complétez le contrôleur `CrimeListFragment` (qui est vide pour le moment) afin de le lier à la vue et en particulier afin de le lier à la `RecyclerView` :

```
private RecyclerView mCrimeRecyclerView;

@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
    Bundle savedInstanceState) {
    View view = inflater.inflate(R.layout.crime_list_fragment, container,
        false);
    mCrimeRecyclerView = (RecyclerView)
        view.findViewById(R.id.crime_recycler_view);
    mCrimeRecyclerView.setLayoutManager(new
        LinearLayoutManager(getActivity()));
    return view;
}
```

Pour le moment, votre vue est toujours vide : il y a bien un container hébergeur qui peut gérer une liste de fragments mais aucun élément ne se trouve dans cette liste.

6.4 Éléments de la liste

Il reste à définir les éléments de la liste.

Le contrôleur des éléments de la liste et le lien entre le fragment contenant la liste et chacun des éléments de cette liste se font via 2 classes :

— Le `ViewHolder` agit comme le contrôleur d'une ligne de cette liste

— L' `Adapter` qui lie la liste et chacun des éléments.

6.5 ViewHolder

Créez la classe `CrimeHolder` comme classe interne à la classe `CrimeListFragment`. C'est une sous-classe de la classe `RecyclerView.ViewHolder` et agit comme le contrôleur d'un élément de la vue à recyclage.

```
private class CrimeHolder extends RecyclerView.ViewHolder {

    private Crime mCrime;
    private TextView mTitleTextView;
    private TextView mDateTextView;

    public CrimeHolder(LayoutInflater inflater, ViewGroup parent) {
        super(inflater.inflate(R.layout.list_item_crime, parent, false));
        mTitleTextView = (TextView) itemView.findViewById(R.id.crime_title);
        mDateTextView = (TextView) itemView.findViewById(R.id.crime_date);
    }
}
```

Cette classe ne fait encore rien, hormis *déplier* (*inflate*) le layout et retrouver les différents widgets.

6.6 Lier la liste et ses éléments : `Adapter`

Il faut encore créer autant d'éléments de la liste que nécessaire, par exemple en fonction de la taille de l'écran. Et gérer la mise à jour des informations. C'est la responsabilité de l'`Adapter`.

Créez la classe `CrimeAdapter` interne à la classe `CrimeListFragment` :

```
private class CrimeAdapter extends RecyclerView.Adapter<CrimeHolder> {

    private List<Crime> mCrimes;

    public CrimeAdapter(List<Crime> crimes) {
        mCrimes = crimes;
    }
    @Override
    public CrimeHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        LayoutInflater layoutInflater = LayoutInflater.from(getActivity());
        return new CrimeHolder(layoutInflater, parent);
    }
    @Override
    public int getItemCount() {
        return mCrimes.size();
    }
    @Override
    public void onBindViewHolder(CrimeHolder holder, int position) {
    }
}
```

La méthode `onCreateViewHolder` définit ce qu'il faut faire pour initialiser chacun des éléments de la liste (vue). La méthode `getItemCount` donne le nombre d'éléments à afficher. La dernière méthode sera complétée dans un instant.

La classe adapteur utilise et instancie les `ViewHolder`. Ajoutez donc une instance de cette classe comme attribut de la classe `CrimeListFragment` et ajoutez la méthode de mise à jour `updateUI`, à appeler dans `onCreateView` :

```
public class CrimeListFragment extends Fragment {

    private RecyclerView mCrimeRecyclerView;
    private CrimeAdapter mCrimeAdapter;

    public View onCreateView(...) {
        (...);
        updateUI();
        return view;
    }
    private void updateUI() {

        if(mCrimeAdapter == null) {
            CrimeLab crimeLab = CrimeLab.get();
            List<Crime> crimes = crimeLab.getCrimes();
            mCrimeAdapter = new CrimeAdapter(crimes);
            mCrimeRecyclerView.setAdapter(mCrimeAdapter);
        } else {
            mCrimeAdapter.notifyDataSetChanged();
        }
    }
}
```

Afin d'actualiser l'affichage lorsque l'on vient de modifier un crime, il suffit d'implémenter la méthode `onResume` de la classe `CrimeListFragment` comme suit :

```
public void onResume(){
    super.onResume();
    updateUI();
}
```

Votre application devrait maintenant afficher une liste de 10000 éléments mais n'affichant uniquement des `"Crime Title"` et `"Crime Date"` car le lien entre le modèle et la vue n'est pas encore effectif. Pour cela on ajoute la méthode suivante au `ViewHolder` :

```
public void bind(Crime crime) {
    mCrime = crime;
    mTitleTextView.setText(crime.getTitle());
    mDateTextView.setText(crime.getDate().toString());
}
```

et on l'appelle dans l'adapteur :

```
@Override
public void onBindViewHolder(CrimeHolder holder, int position) {
    Crime crime = mCrimes.get(position);
    holder.bind(crime);
}
```

La liste est maintenant complète !

6.7 Explications

Essayons de prendre du recul et de comprendre comment cela fonctionne.

Un `RecyclerView` va calculer le nombre vues qu'il lui faut pour afficher la partie visible de la liste. Par exemple, sur mon écran, une quinzaine de crimes peuvent être affichés sur les 10000. Le `RecyclerView` passe alors par l'adapter pour gérer les vues. Ces vues sont créées à la demande via la méthode `onCreateViewHolder` de l'adapteur, stockées dans l'adapteur et complétées avec les bonnes données via la méthode `onBindViewHolder`.

Lorsque l'on scrolle dans la liste, des éléments ne sont plus visibles. Ils sont alors réutilisés pour afficher les nouveaux. Il n'y a donc pas de nouvelle création de vue mais une réutilisation via la méthode `onBindViewHolder`.

Grâce à ce mécanisme, il y a peu de création de vues et l'affichage est restreint à ce qui doit l'être. C'est très efficace.

6.8 Appel vers les détails d'un crime

Dans cette dernière section, nous allons lancer une vue détaillée concernant un crime lorsque celui-ci est cliqué dans la liste.

Pour cela, il suffit d'ajouter un listener pour l'événement correspondant sur le `ViewHolder` :

```
public CrimeHolder(LayoutInflater inflater, ViewGroup parent) {
    (...)

    itemView.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            Intent intent = new Intent(getActivity(), CrimeActivity.class);
            intent.putExtra(CrimeFragment.CRIME_ID, mCrime.getId());
            startActivity(intent);
        }
    });
}
```

On lance une nouvelle activité de type `CrimeActivity` lorsqu'un élément de la liste est cliqué, comme nous l'avons fait lors de la construction manuelle de liste.

Votre application est fonctionnelle (enfin, on l'espère).

Vérifiez que la liste est mise à jour lorsque vous éditez le titre dans la vue détail et que vous revenez (bouton *Back*) à la liste.

6.9 Conclusion

Nous voyons clairement que cette technique est bien plus complexe à mettre en place que la première. Mais elle est en revanche bien plus efficace en terme de rapidité

d'affichage. Elle est donc à prévilégier dès que votre écran peut contenir un grand nombre d'éléments.

7 Exercices

1. Faites apparaître dans la liste le fait qu'un crime soit résolu ou non. Par exemple avec une couleur de fond vert lorsque c'est résolu et rouge lorsque cela ne l'est pas, ou avec une image.
2. Ajoutez *une gravité* comme attribut à la classe `Crime` : un crime peut être Mild (léger), Moderate (moyen), Severe (sérieux). Faites apparaître cela dans la liste (`CrimeListFragment`) et dans le détail `CrimeFragment`. Faites en sorte que la gravité soit éditable dans la vue détail.
3. Gérer les différentes tailles d'écran : sur un grand écran, on devrait pouvoir voir la liste et le détail en même temps.
4. Améliorez l'affichage de toute l'application.