Програмування мобільних пристроїв

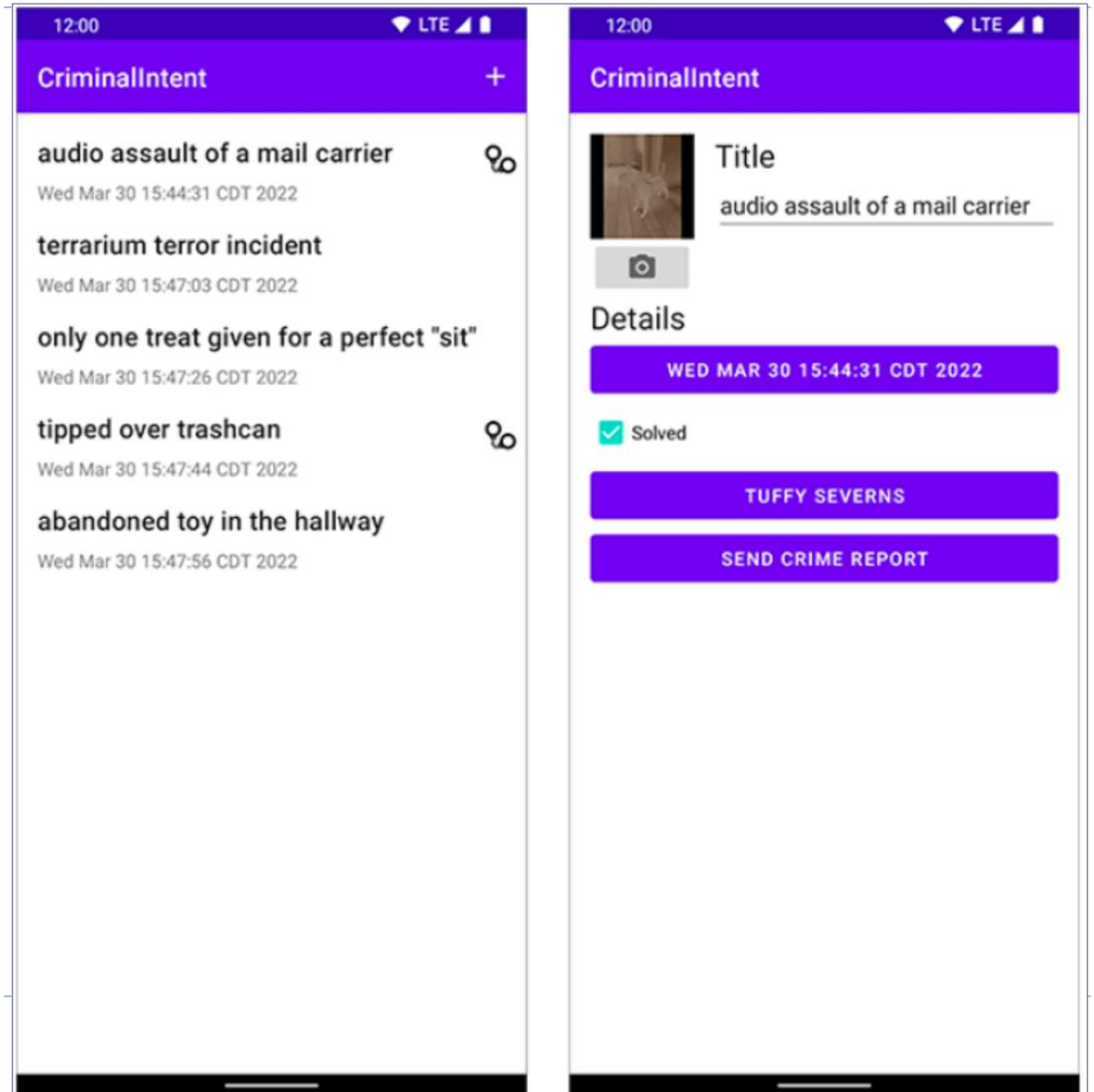
Використання фрагментів

Слайди до лекцій (6 змістовий модуль)

Функціональні вимоги до програми CriminalIntent

- Починаючи з цього розділу ми почнемо будувати програму CriminalIntent. Вона призначена для зберігання інформації про "офісні злочини": залишений у раковині брудний посуд, порожній лоток принтера після друку документів тощо.
- У програмі CriminalIntent користувач створює запис про злочин із заголовком, датою та фотографією. Також можна вибрати підозрюваного в адресній книзі та надіслати скаргу електронною поштою, опублікувати її в Twitter, Facebook або іншій програмі. Повідомивши про злочин, користувач звільняється від негативу і може зосередитися на поточному завданні.
- CriminalIntent – складний застосунок, у якому використовується інтерфейс типу "список/деталізація": на головному екрані виводиться список зареєстрованих злочинів. Користувач може додати новий або вибрати існуючий злочин для перегляду та редагування інформації.

Функціональні вимоги до програми CriminalIntent



Гнучкість інтерфейсу користувача

- Базуючись на досвіді розробки попереднього застосунку, можна спроєктувати застосунок типу "список/деталізація" з двох активностей: для керування списком та для керування деталізованим поданням. Така архітектура працює, але може знадобитися більш складна схема представлення інформації та навігації між екранами: наприклад, при роботі з програмою на планшеті. Екрани планшетів та деяких великих телефонів дозволяють одночасно відображати список та деталізацію (принаймні в альбомній орієнтації).
- Користувач переглядає опис злочину на телефоні та хоче побачити наступний злочин у списку. Було б зручно, якби користувач міг провести пальцем на екрані, щоб перейти до наступного злочину без повернення до списку. Такий сценарій передбачає гнучкість інтерфейсу користувача: можливість формування та зміни представлення активності під час виконання залежно від того, що потрібно користувачеві або пристрою.

Гнучкість інтерфейсу користувача

- Подібна гнучкість в активності не передбачена. Представлення активності можуть змінюватися під час виконання, але код, який керує цими змінами, повинен перебувати у представленні. В результаті активність тісно зв'язується з конкретним екраном, з яким працює користувач.

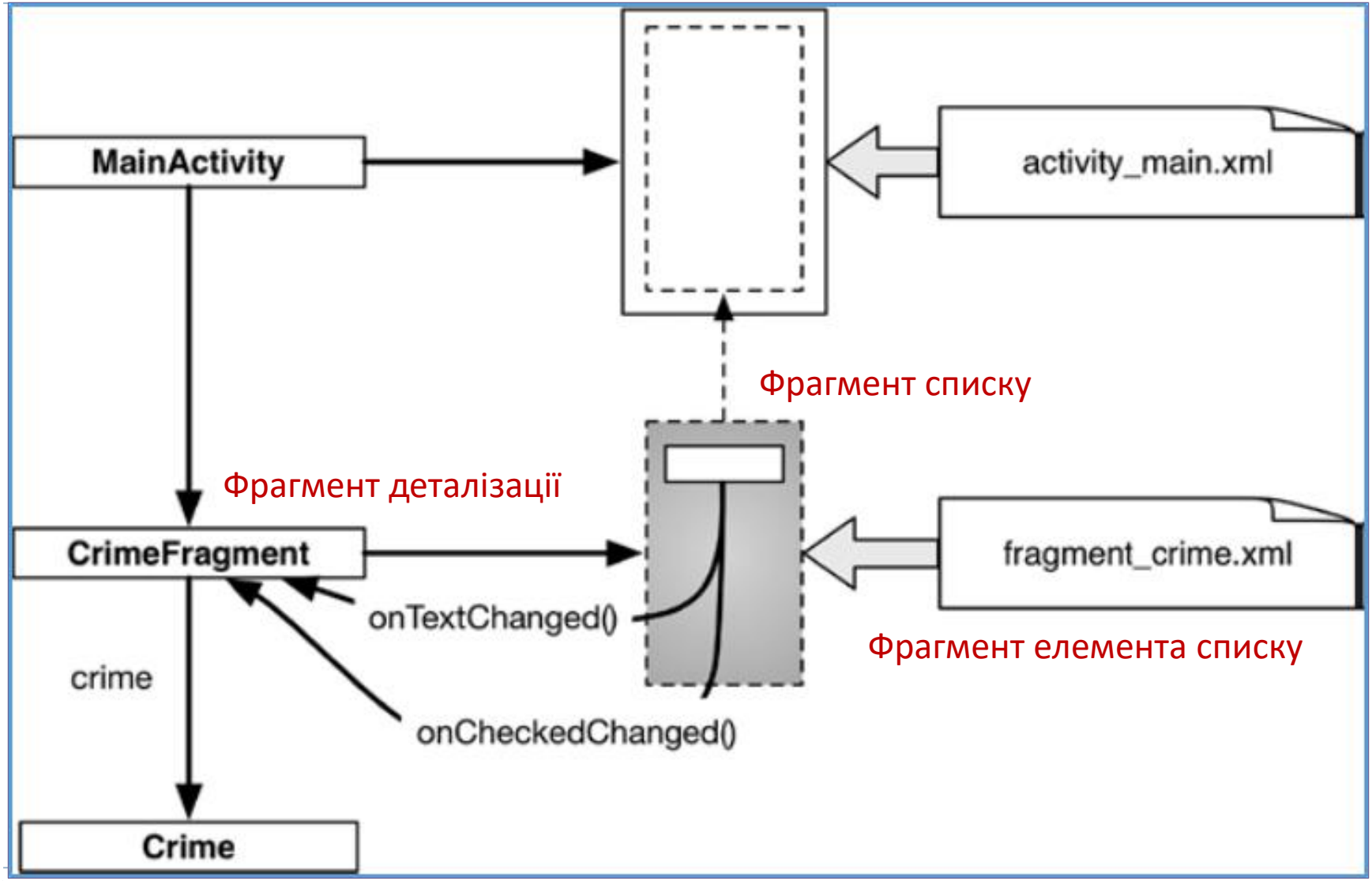
Технологія використання фрагментів

- Забезпечити гнучкість при побудуванні інтерфейсу можливо передавши управління інтерфейсом програми від активності одному або декільком *фрагментам* (*Fragment*). *Фрагмент* є об'єктом контролера, якому активність може довірити виконання операцій. Найчастіше такою операцією є керування інтерфейсом користувача – цілим екраном або його частиною.
- Фрагмент, що управляє інтерфейсом користувача, називається *UI-фрагментом* (*UI-fragment*). UI-фрагмент має власне представлення, яке заповнюється виходячи з файлу макета фрагмента. Представлення активності містить місце, в яке вставляється представлення фрагмента (активність може мати декілька місць для представлення кількох фрагментів). Фрагменти, пов'язані з активністю, можуть використовуватися для формування та зміни екрана відповідно до потреб програми та користувачів. При цьому представлення активності формально залишається незмінним протягом життєвого циклу.

Використання фрагментів у застосунку "список-деталізація"

- У застосунку "список/деталізація" представлення активності будується з фрагмента списку та фрагмента деталізації. Представлення деталізації містить докладну інформацію про вибраний елемент списку. При виборі іншого елемента у списку на екрані з'являється нове деталізоване представлення. Ця зміна представлень відбувається без знищення активності.
- Застосування UI-фрагментів дозволяє розділити інтерфейс застосунку на структурні блоки, а це корисно не тільки для застосунків "список/деталізація". Робота з окремими блоками полегшує розробку інтерфейсів з вкладками, анімованих бічних панелей та багато іншого. Крім того, деякі з нових Android Jetpack API краще працюють саме з фрагментами.

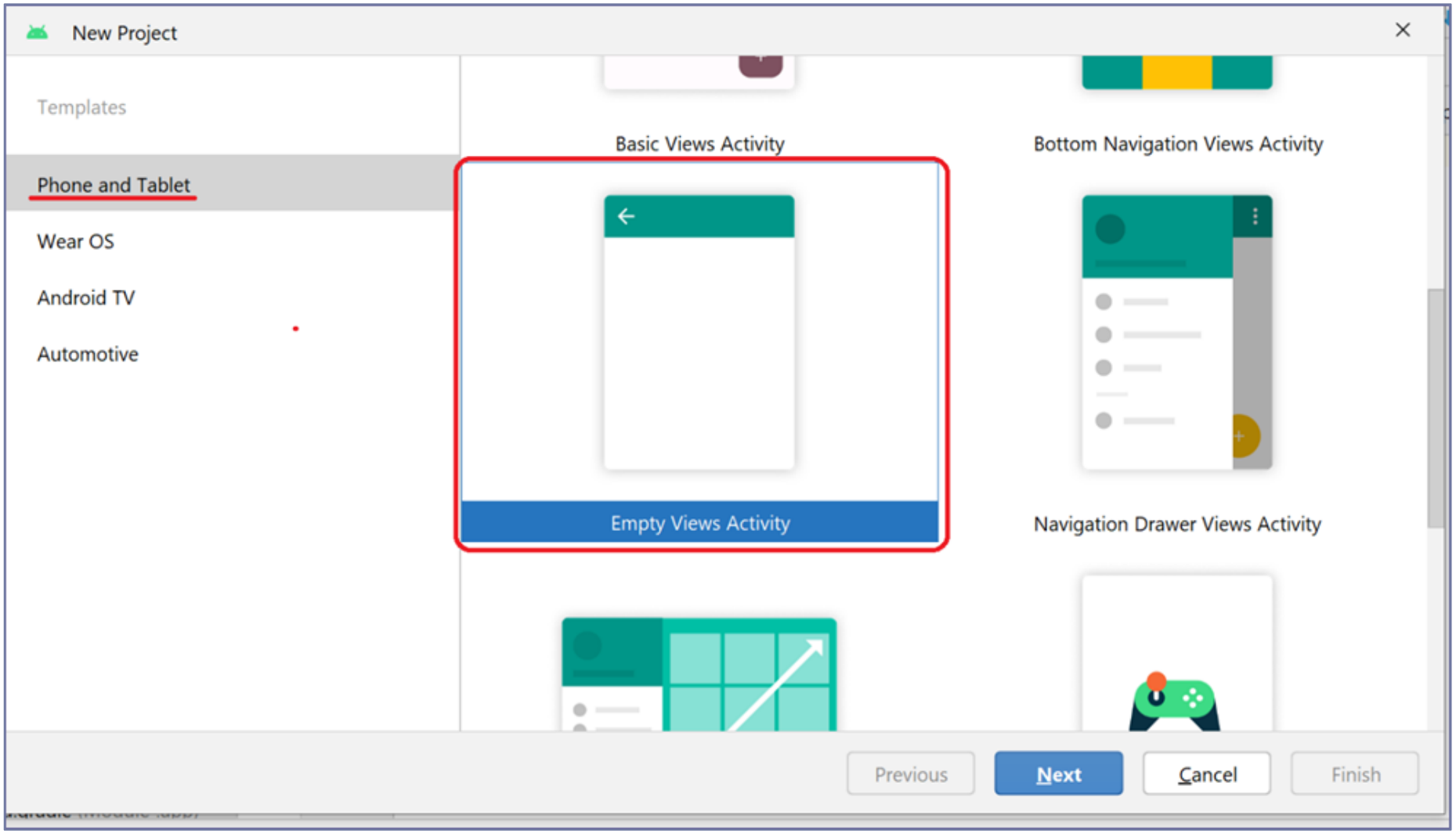
Гнучкість інтерфейсу користувача



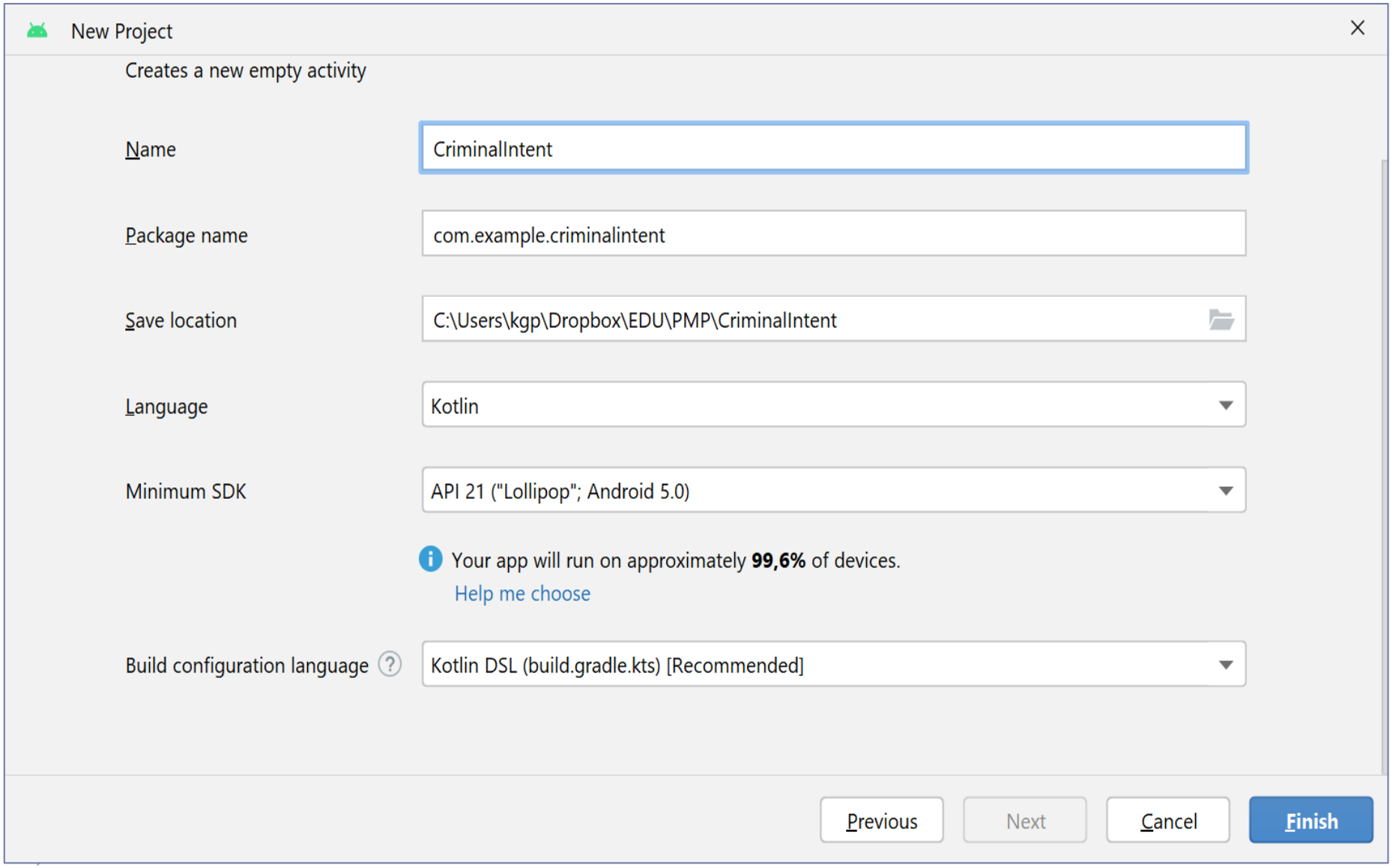
Технологія використання фрагментів

- Фрагменти були введені в API рівня 11 (Android 3.0 Honeycomb – лютий 2011 р) разом з першими планшетами Android. Реалізація фреймворку фрагментів вже була вбудована в пристрої, що працюють на рівні API 11 і вище. Незабаром після цього до бібліотеки підтримки v4 було додано реалізацію класу Fragment для включення підтримки фрагментів на старих пристроях.
- Починаючи з Android 9.0 Pie (API 28 – січень 2022 р) фреймворк-версія фрагментів застаріла. Жодних подальших оновлень цієї версії проводитися не буде. Засоби роботи з фрагментами були перенесені до Jetpack-бібліотеки. Усі подальші оновлення відносяться до Jetpack, а не до фреймворку або фрагментів підтримки v4. Тому у нових проектах необхідно використовувати фрагменти Jetpack, успадковуючи класи користувацьких фрагментів від класу androidx.fragment.app.Fragment.
-

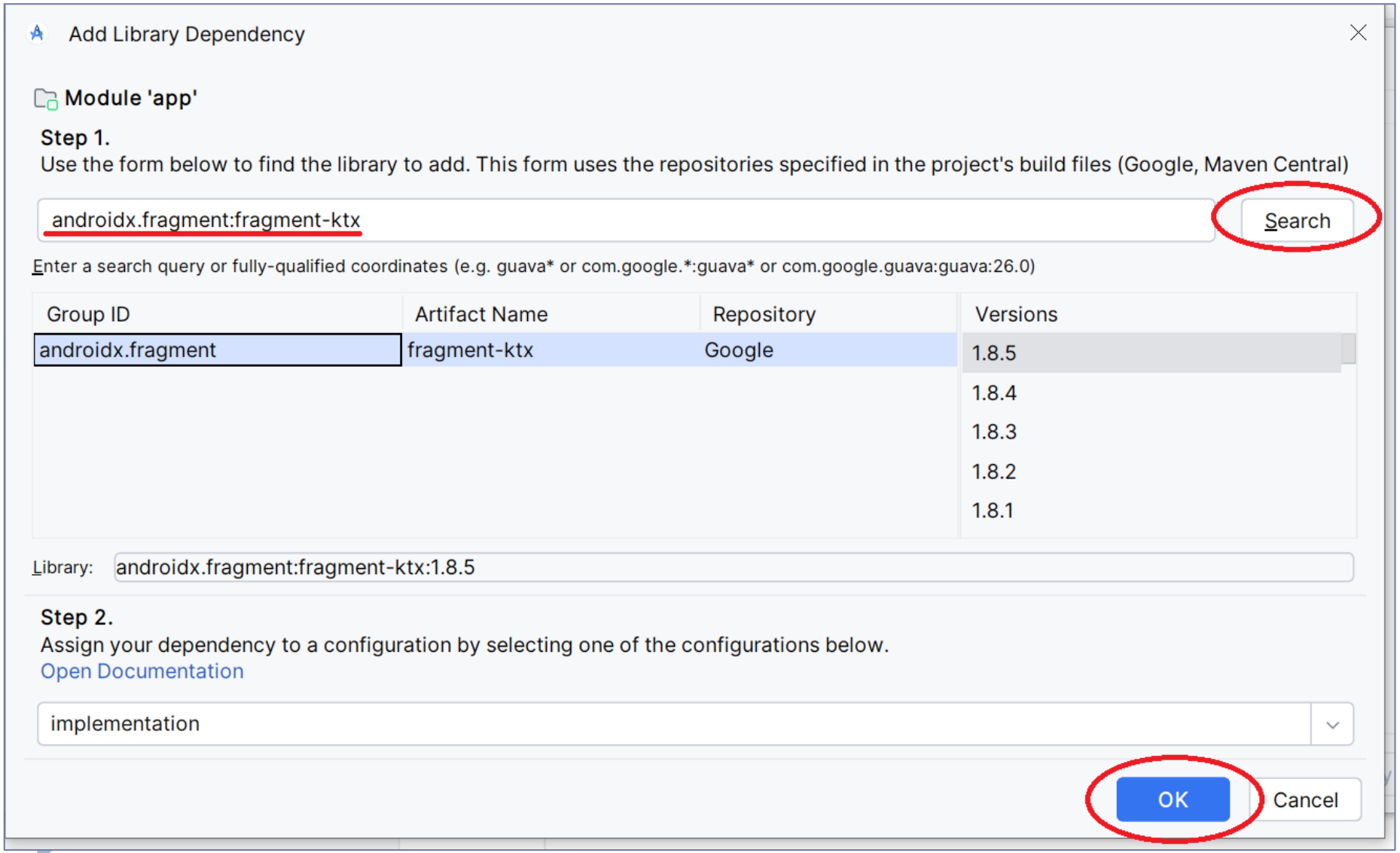
Створення проєкту CriminalIntent



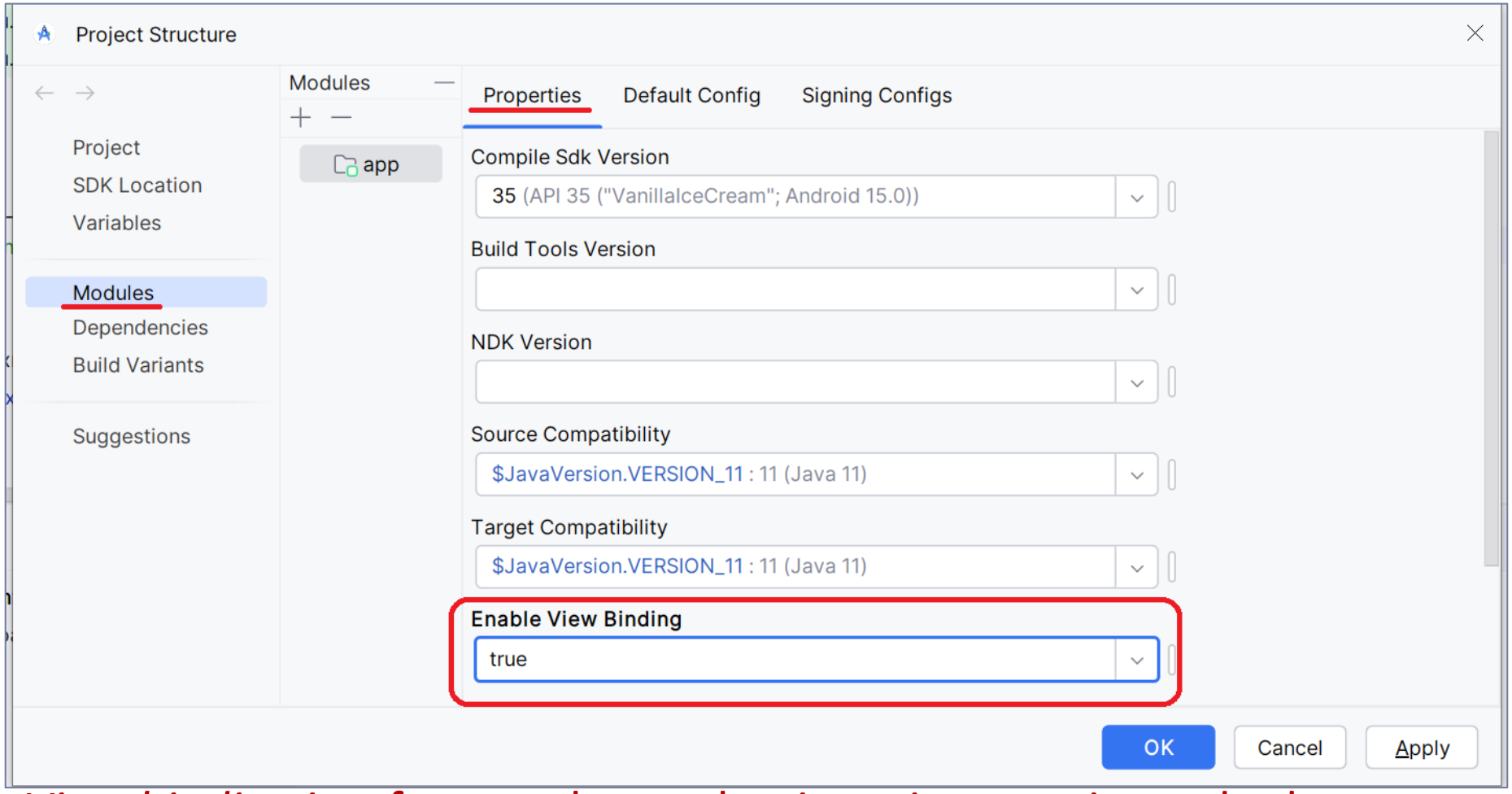
Створення проєкту CriminalIntent



Додання залежності fragment-ktx



Підключення технології зв'язування (Binding)



View binding is a feature that makes it easier to write code that interacts with views. It auto-generates a *binding class* for each XML layout file present in the module. An instance of a binding class contains direct references to all views that have an ID.

Створення класу моделі

```
package ua.edu.znu.criminalintent

import java.time.LocalDate
import java.util.UUID

data class Crime(
    val id: UUID,
    val title: String,
    val date: LocalDate,
    val isSolved: Boolean
);
```

- UUID – це допоміжний клас Java, що входить до інфраструктури Android, він надає простий спосіб генерування універсально-унікальних ідентифікаторів. Дата злочину буде об'єктом LocalDate.

Додання рядкових ресурсів

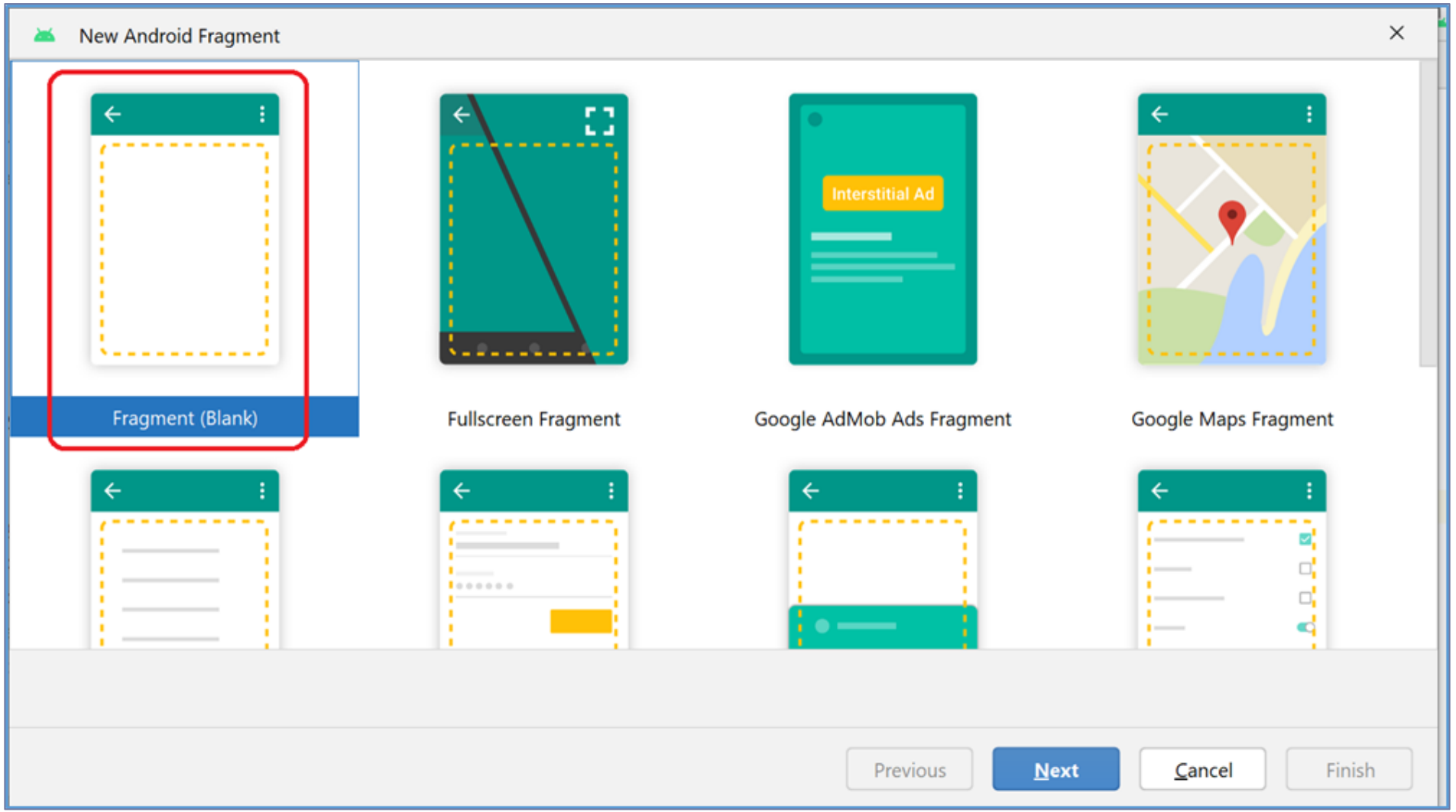
activity_main.xml | MainActivity.kt | Crime.kt | Translations Editor

Show All Keys | Show All Locales

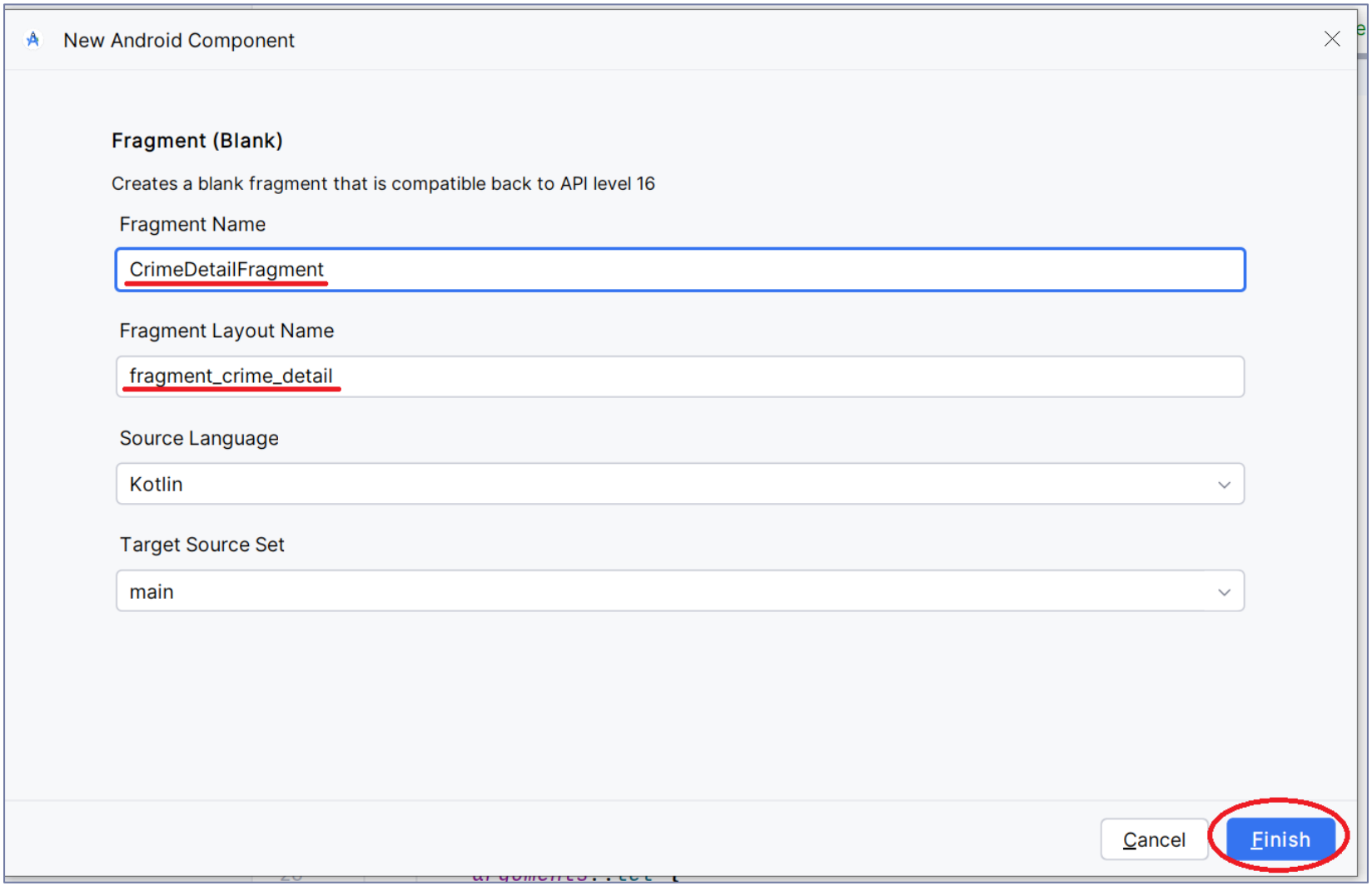
Key	Resource Folder	Untranslatable	Default Value	Ukrainian (uk)
app_name	app/src/main/res	☑	CriminalIntent	
crime_title_label	app/src/main/res	☐	Title	Назва
crime_details_label	app/src/main/res	☐	Details	Опис
crime_solved_label	app/src/main/res	☐	Solved	Усунутий
crime_title_hint	app/src/main/res	☐	Enter a title for the crime	Введіть назву для злочину

Додання фрагменту деталізації CrimeDetailFragment

New –Fragment/Gallery…



Додання фрагменту деталізації CrimeDetailFragment



Макет фрагменту деталізації CrimeDetailFragment

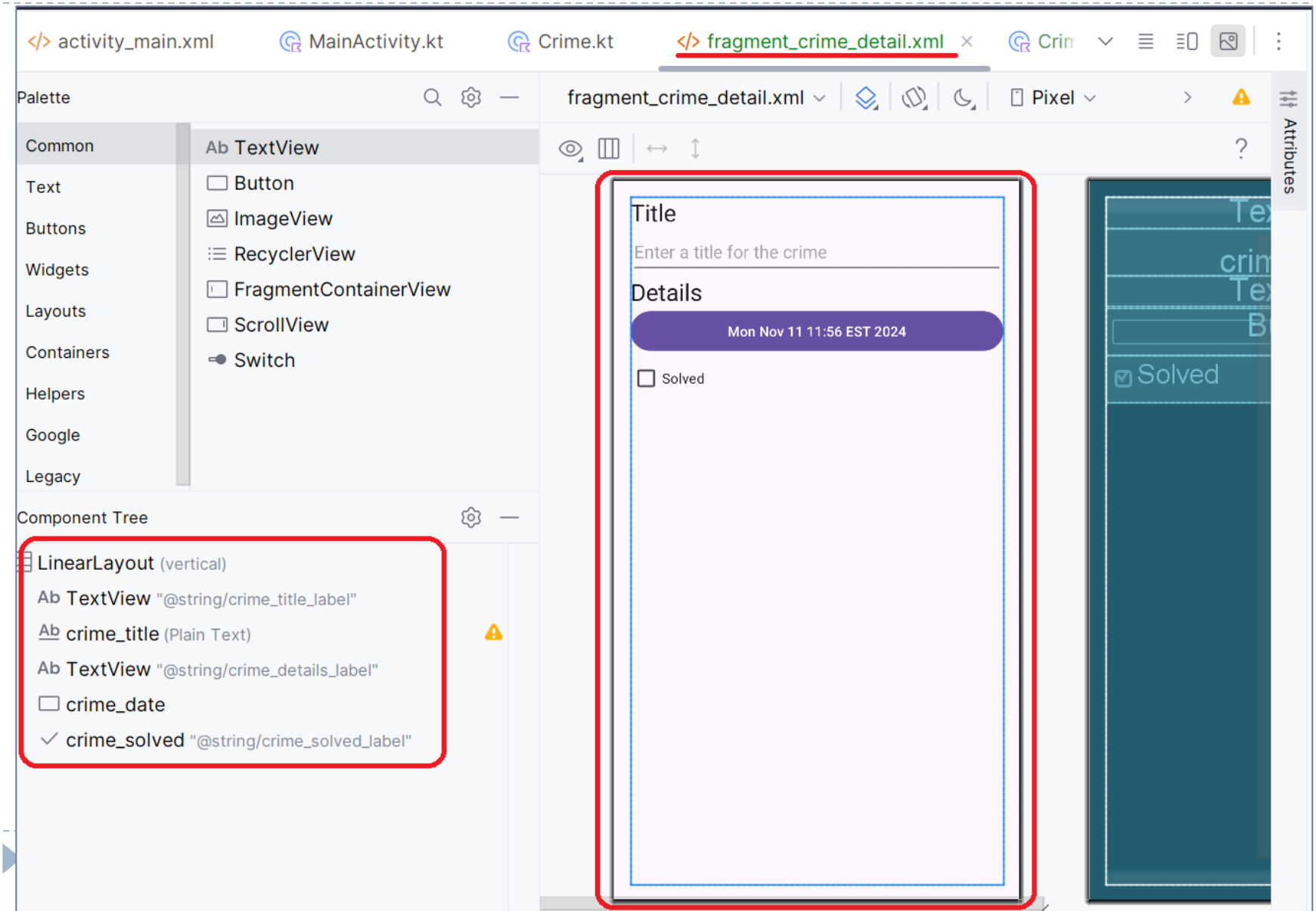
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_margin="16dp"
    android:orientation="vertical"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

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/crime_title_label"
        android:textAppearance="?attr/textAppearanceHeadline5" />

    <EditText
        android:id="@+id/crime_title"
        android:layout_width="match_parent"
        android:layout_height="48dp"
        android:hint="@string/crime_title_hint"
        android:importantForAutofill="no"
        android:inputType="text" />    ...
```

застосовує налаштування типографіки Headline 5 до тексту, як зазначено у бібліотеці Material Design Google.

Макет фрагменту деталізації CrimeDetailFragment

```
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/crime_details_label"
        android:textAppearance="?attr/textAppearanceHeadline5" />

    <Button
        android:id="@+id/crime_date"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        tools:text="Mon Nov 11 11:56 EST 2024" />

    <CheckBox
        android:id="@+id/crime_solved"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/crime_solved_label" />

</LinearLayout>
```

Макет фрагменту деталізації CrimeDetailFragment



Клас фрагменту деталізації CrimeDetailFragment - onCreate

```
package ua.edu.znu.criminalintent

import ...

class CrimeDetailFragment : Fragment() {

 /*View Binding*/
private lateinit var binding: FragmentCrimeDetailBinding

private lateinit var crime: Crime

override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        crime = Crime(
            id = UUID.randomUUID(),
            title = "",
            date = LocalDateTime.now(),
            isSolved = false
        )
    } ...
```

функція життєвого циклу фрагменту

Клас фрагменту деталізації CrimeDetailFragment - onCreateView

функція життєвого циклу фрагменту

```
...
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        /*Inflate the layout for this fragment.
          Use a View Binding.
          See the var binding defining at the class start*/
        binding = FragmentCrimeDetailBinding.inflate(inflater, container, false)
        return binding.root
    }
...
```

The fragment's view does not need to be added to the parent view immediately – the activity will handle adding the view later.

Клас фрагменту деталізації CrimeDetailFragment - onViewCreated

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    binding.apply {
        crimeTitle.doOnTextChanged { text, _, _, _ ->
            crime = crime.copy(title = text.toString())
        }
        crimeDate.apply {
            text = crime.date.toString()
            isEnabled = false
        }
        crimeSolved.apply {
            setOnCheckedChangeListener { _, isChecked ->
                crime = crime.copy(isSolved = isChecked)
            }
        }
    }
}
```

функція життєвого циклу фрагменту

The underscore denotes unused parameters

визначення вмісту та поведінки UI компонентів технологією зв'язування

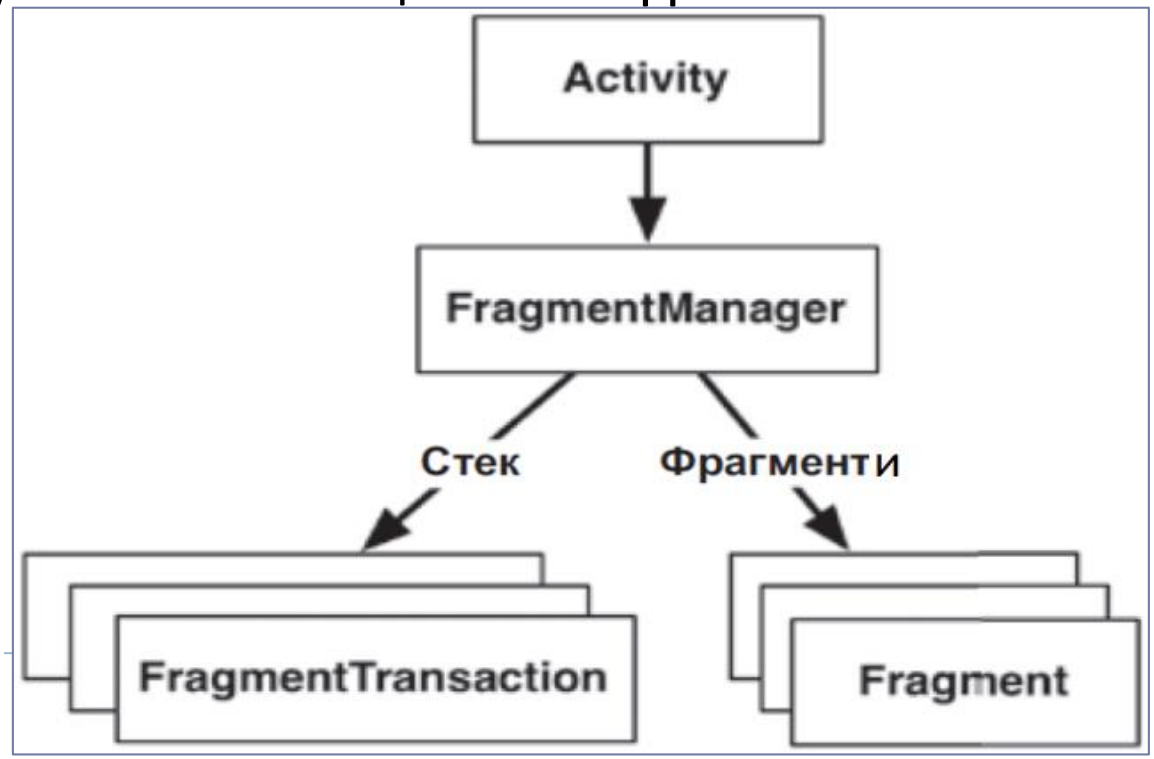
Макет активності-хостингу UI-фрагментів activity_main.xml

- У 2019 році Google представила **FragmentContainerView**, який полегшує створення хост-контейнера для фрагмента, що керуться **FragmentManager-**ом.
- FragmentContainerView визначається як макет і має стандартні XML атрибути макету.
- Атрибут **android:name** містить повну назву класу фрагменту, об'єкт якого буде вставлений до макету активності, а атрибут **tools:layout** містить ідентифікатор макету шаблону.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/fragment_container"
    android:name="ua.edu.znu.criminalintent.CrimeDetailFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    tools:layout="@layout/fragment_crime_detail">
</androidx.fragment.app.FragmentContainerView>
```

FragmentManager

- Починаючи з Android 3.0 (Honeycomb) з'явився компонент **FragmentManager**, який:
  1) відповідає за додавання представлень фрагментів до ієрархії представлень активності;
  2) керує зворотнім стеком (back stack) транзакцій фрагментів;
  3) керує життєвими циклами фрагментів.



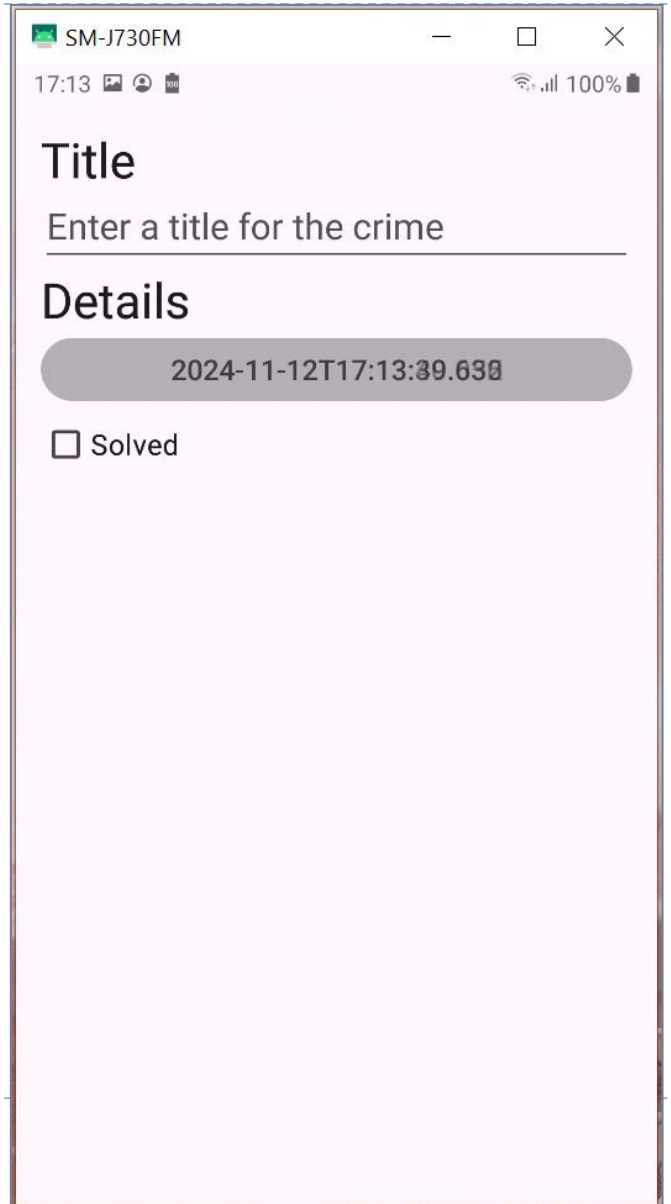
FragmentManager

- Для додання фрагменту до активності необхідно робити явний виклик FragmentManager-а активності за допомогою властивості активності supportFragmentManager.
- Такі дії, як додавання, видалення або заміна фрагментів, виконуються у межах **транзакцій фрагментів**. Вони дозволяють групувати кілька операцій, наприклад додавати кілька фрагментів до різних контейнерів одночасно.
- FragmentManager підтримує **зворотній стек (back stack) транзакцій фрагментів**, по якому можна переміщатися. Якщо транзакцію вилучити зі стеку, то всі операції у її межах скасовуються.

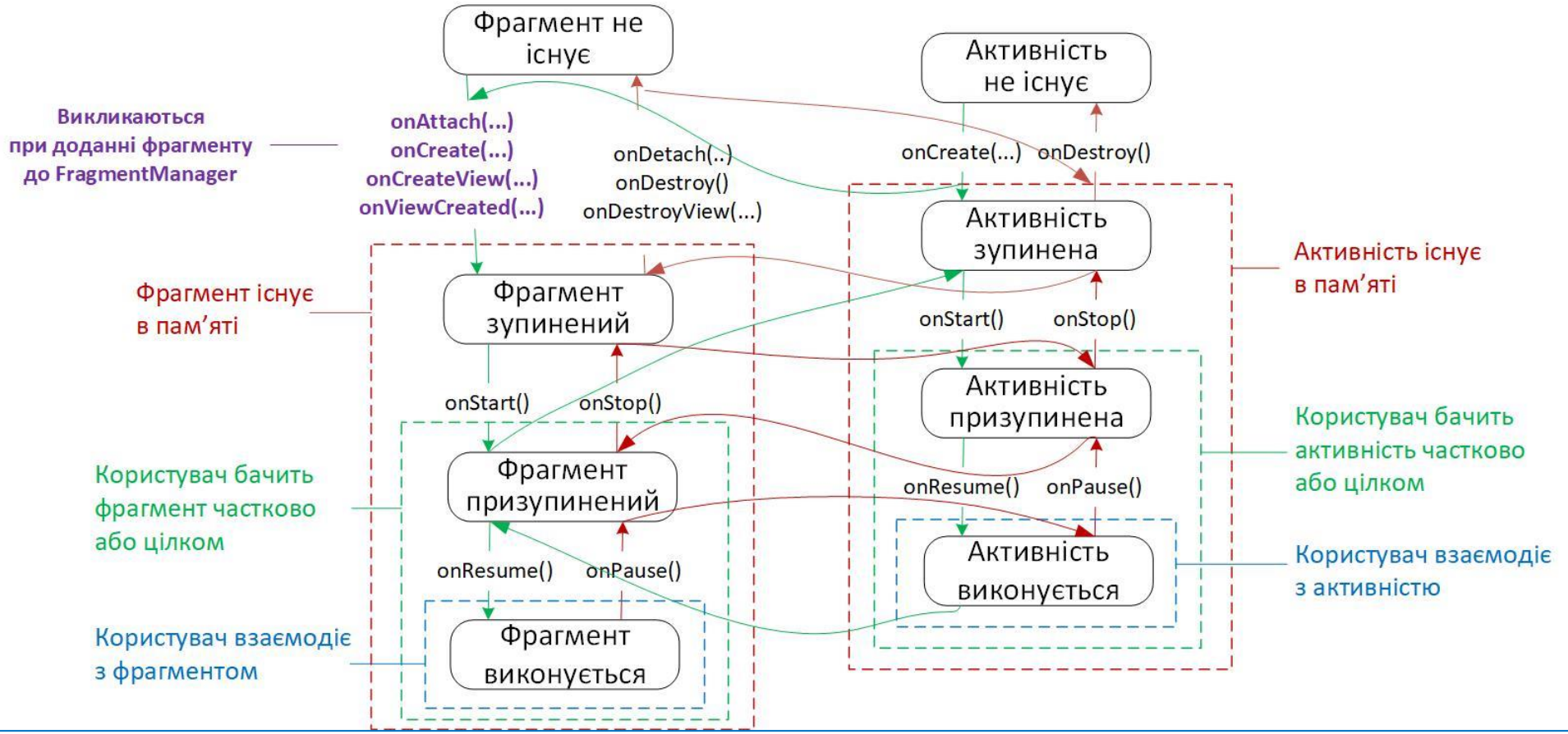
```
val fragment = CrimeDetailFragment()
supportFragmentManager.beginTransaction() //returns FragmentTransaction
    /*FragmentTransaction has several functions for adding, replacing
     and removing fragments. These functions are chainable.*/
    .add(R.id.fragment_container, fragment)
    .commit()
```

It is not need due to **FragmentContainerView** use

MainActivity з фрагментом



Життєвий цикл фрагментів



- На відміну від активності, функції життєвого циклу фрагмента викликаються не операційною системою, а FragmentManager. Операційна система нічого не знає про фрагменти, фрагменти – це "внутрішня кухня" активності.

Життєвий цикл фрагментів

- Якщо фрагмент додається у той час, коли активність вже знаходиться в стані зупинки, призупинки або виконання, FragmentManager негайно проводить фрагмент через всі етапи, необхідні для узгодження його стану з відповідним станом активності.
- Наприклад, при додаванні фрагмента до активності, що вже перебуває в стані виконання, фрагмент отримає виклики onAttach(Context), onCreate(Bundle?), onCreateView(...), onViewCreated(…), onActivityCreated(Bundle), onStart() і onResume().
- Після того, як стан фрагмента буде узгоджений зі станом активності, об'єкт FragmentManager хост-активності буде викликати подальші функції життєвого циклу приблизно одночасно з отриманням відповідних викликів від ОС для синхронізації стану фрагмента зі станом активності.

Фрагменти та управління пам'яттю

```
------ PROCESS STARTED (22216) for package ua.edu.znu.criminalintent ----------------------------
D  onAttach called
D  onCreate called - CrimeDetailFragment{49f521b} (5559ffcc-b0ad-411f-970f-31bfb4fae210 id=0x7f0800cb)
D  onCreateView called - CrimeDetailFragment{49f521b} (5559ffcc-b0ad-411f-970f-31bfb4fae210 id=0x7f0800cb)
D  onViewCreated called
D  onStart called
D  onResume called
D  onPause called
D  onStop called
D  onDestroyView called - CrimeDetailFragment{49f521b} (5559ffcc-b0ad-411f-970f-31bfb4fae210 id=0x7f0800cb
D  onDestroy called - CrimeDetailFragment{49f521b} (5559ffcc-b0ad-411f-970f-31bfb4fae210 id=0x7f0800cb)
D  onDetach called
D  onAttach called
D  onCreate called - CrimeDetailFragment{3d992d} (02f1f97f-7ac1-49ba-8510-31ed90e4cba2 id=0x7f0800cb)
D  onCreateView called - CrimeDetailFragment{3d992d} (02f1f97f-7ac1-49ba-8510-31ed90e4cba2 id=0x7f0800cb)
D  onViewCreated called
D  onStart called
D  onResume called
```

1

Видалення

2

3

I - Запуск застосунку

2 - зупинка застосунку

3 - відновлення застосунку до активного стану

Фрагменти та управління пам'яттю

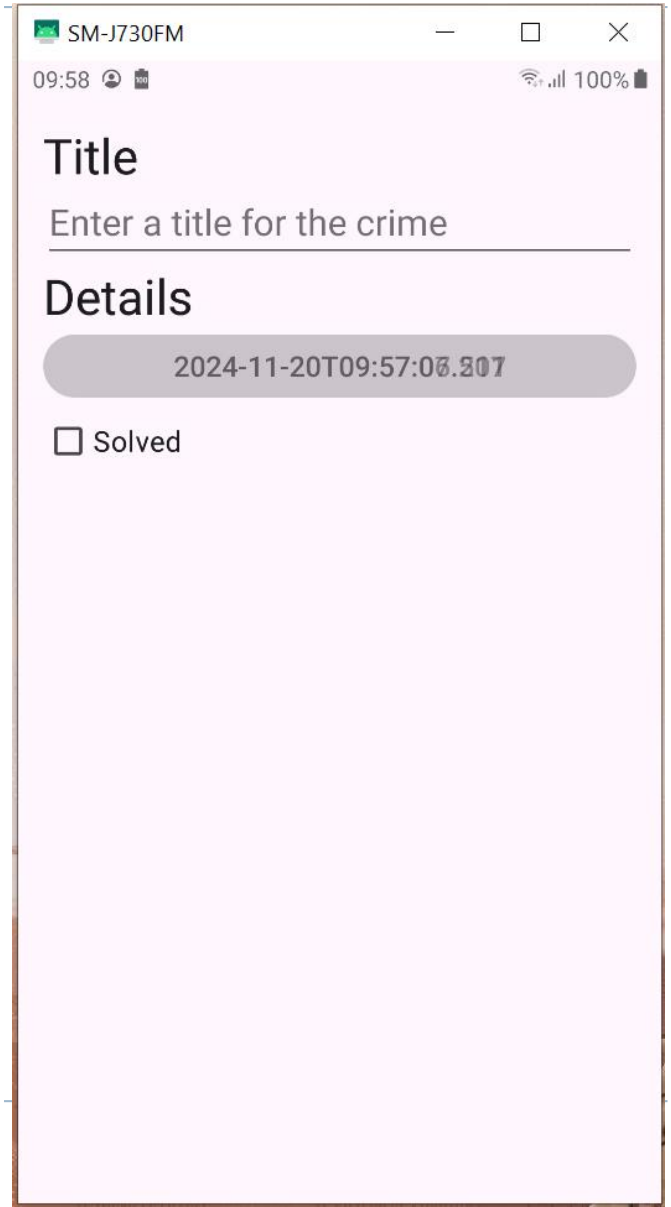
- Властивість ViewBinding обмежена життєвим циклом перегляду фрагмента (між onCreateView та onDestroyView).
- Тому рекомендується встановлення для відповідного ViewBinding значення null у onDestroyView фрагмента.

```
class CrimeDetailFragment : Fragment() {
    /*For View Binding cleanup*/
    private var _binding: FragmentCrimeDetailBinding? = null
    /*This property is only valid between onCreateView
       and onDestroyView.*/
    private val binding get() = _binding!!
...
```

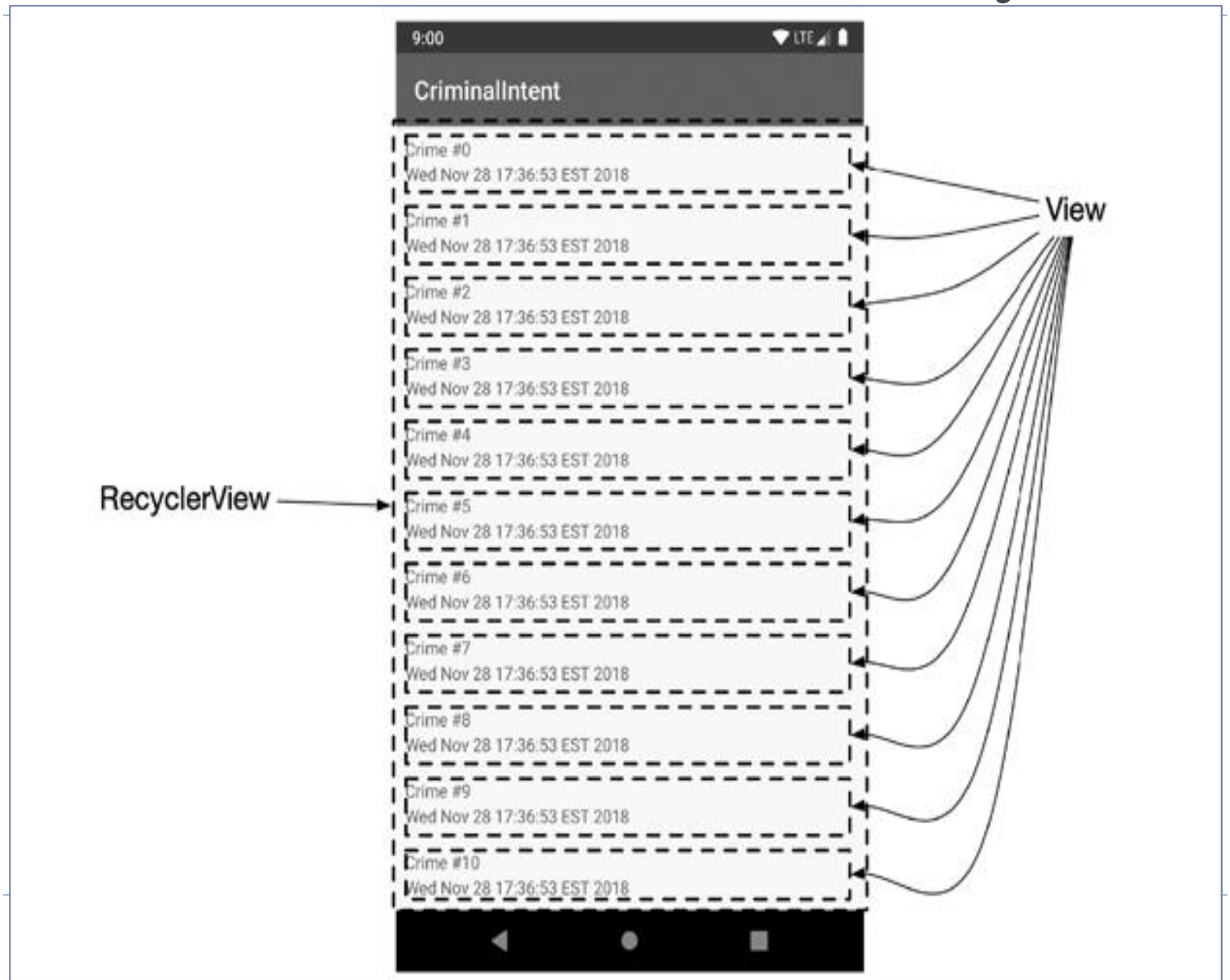
Фрагменти та управління пам'яттю

```
...
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentCrimeDetailBinding.inflate(inflater, container, false)
        Log.d(TAG, "onCreateView called - ${binding}")
        return binding.root
    }
...
    override fun onDestroyView() {
    super.onDestroyView()
    Log.d(TAG, "onDestroyView called - $binding")
    /*View Binding cleanup*/
    _binding = null
} ... }
```

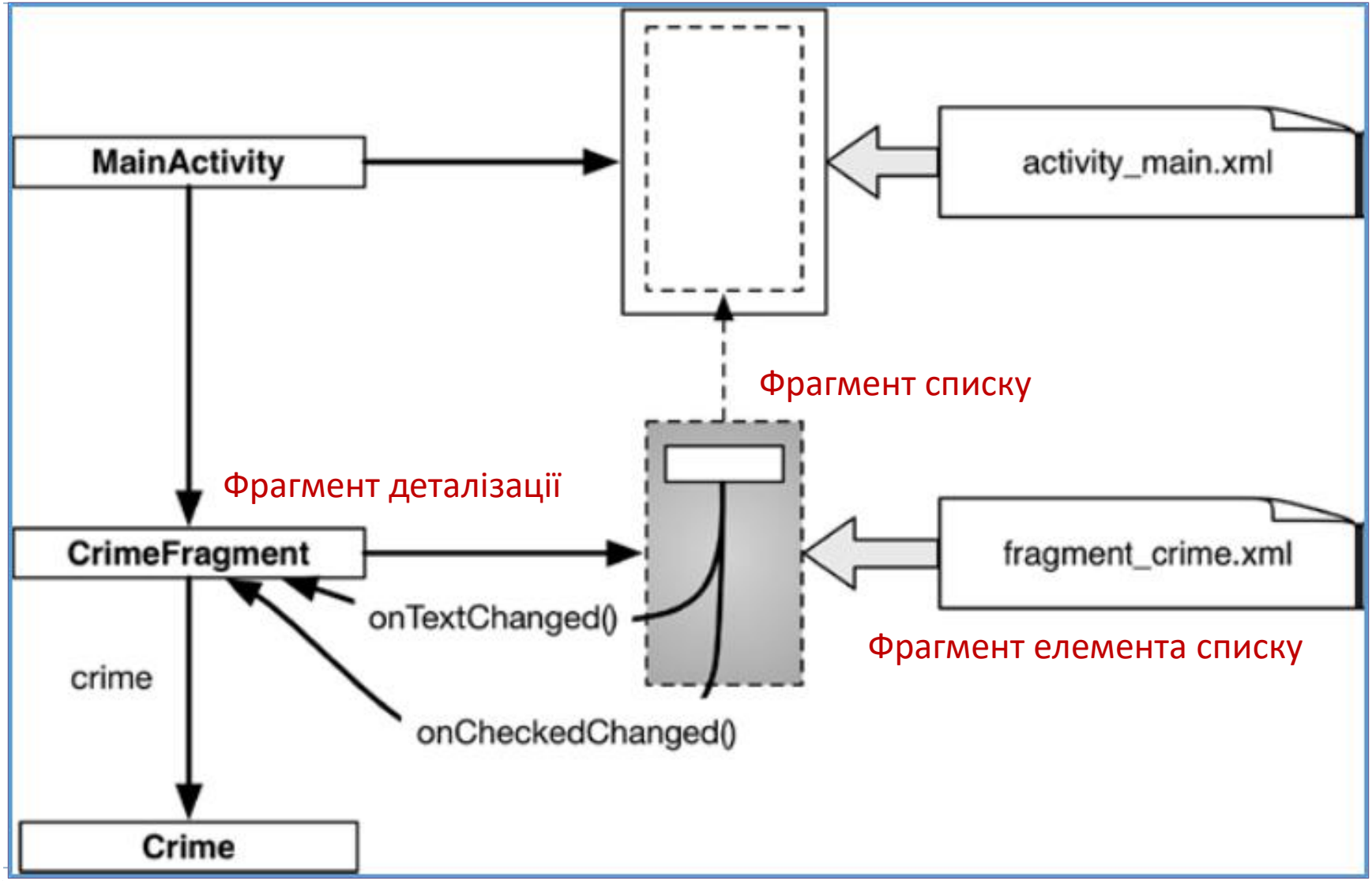
Фрагменти та управління пам'яттю



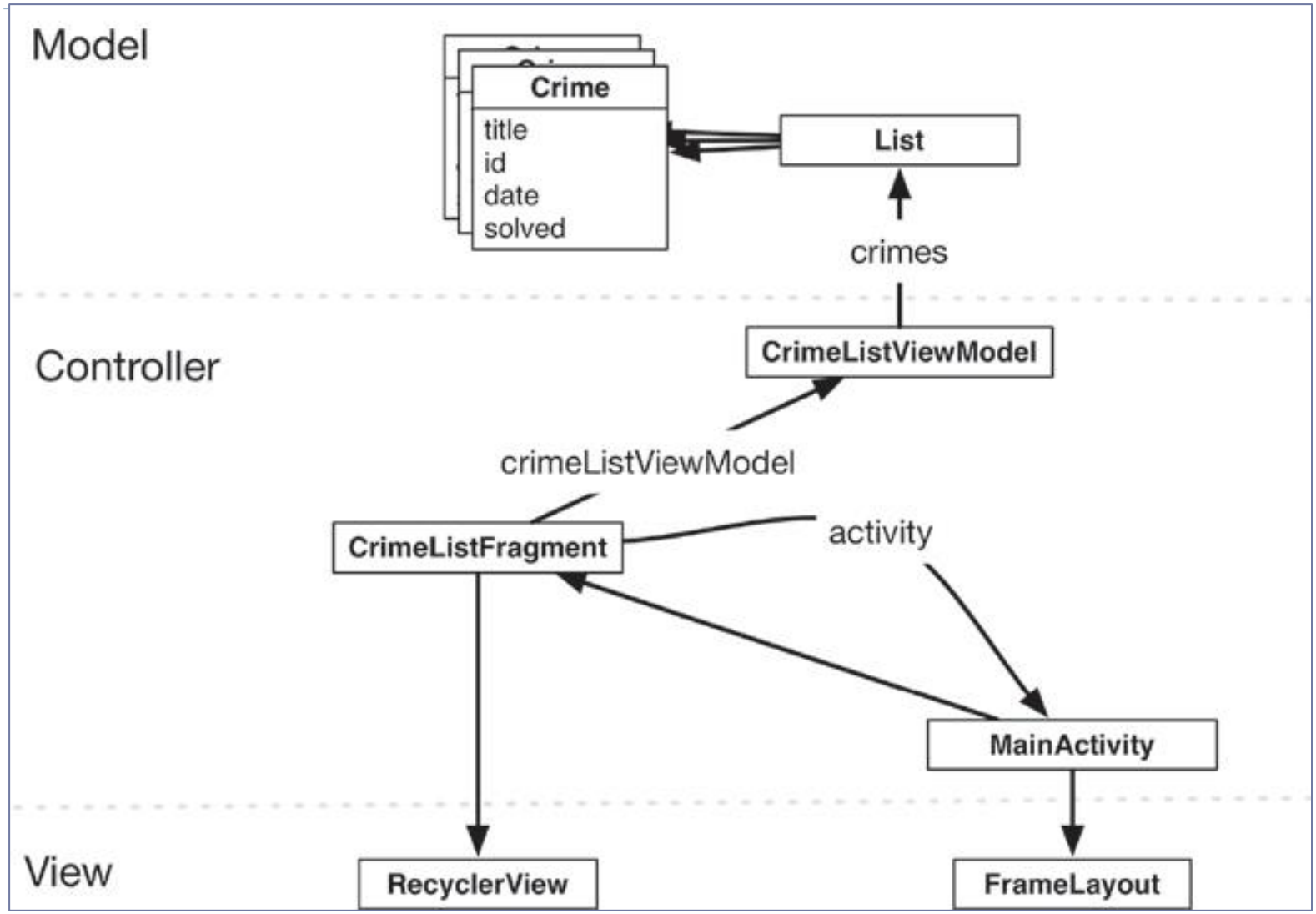
Виведення списків за допомогою RecyclerView



Макет активності UI-фрагментів activity_main.xml

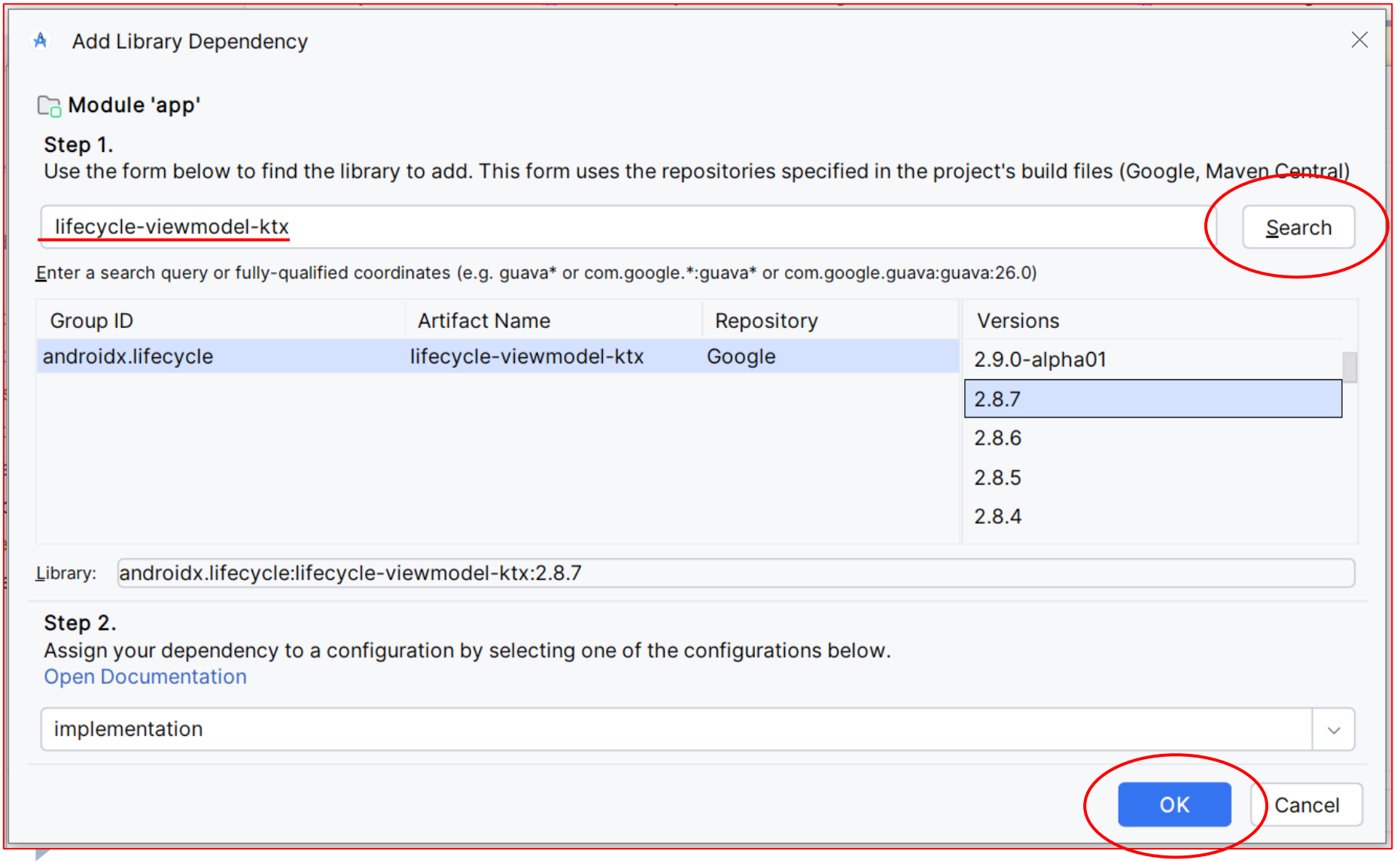


Виведення списків за допомогою RecyclerView



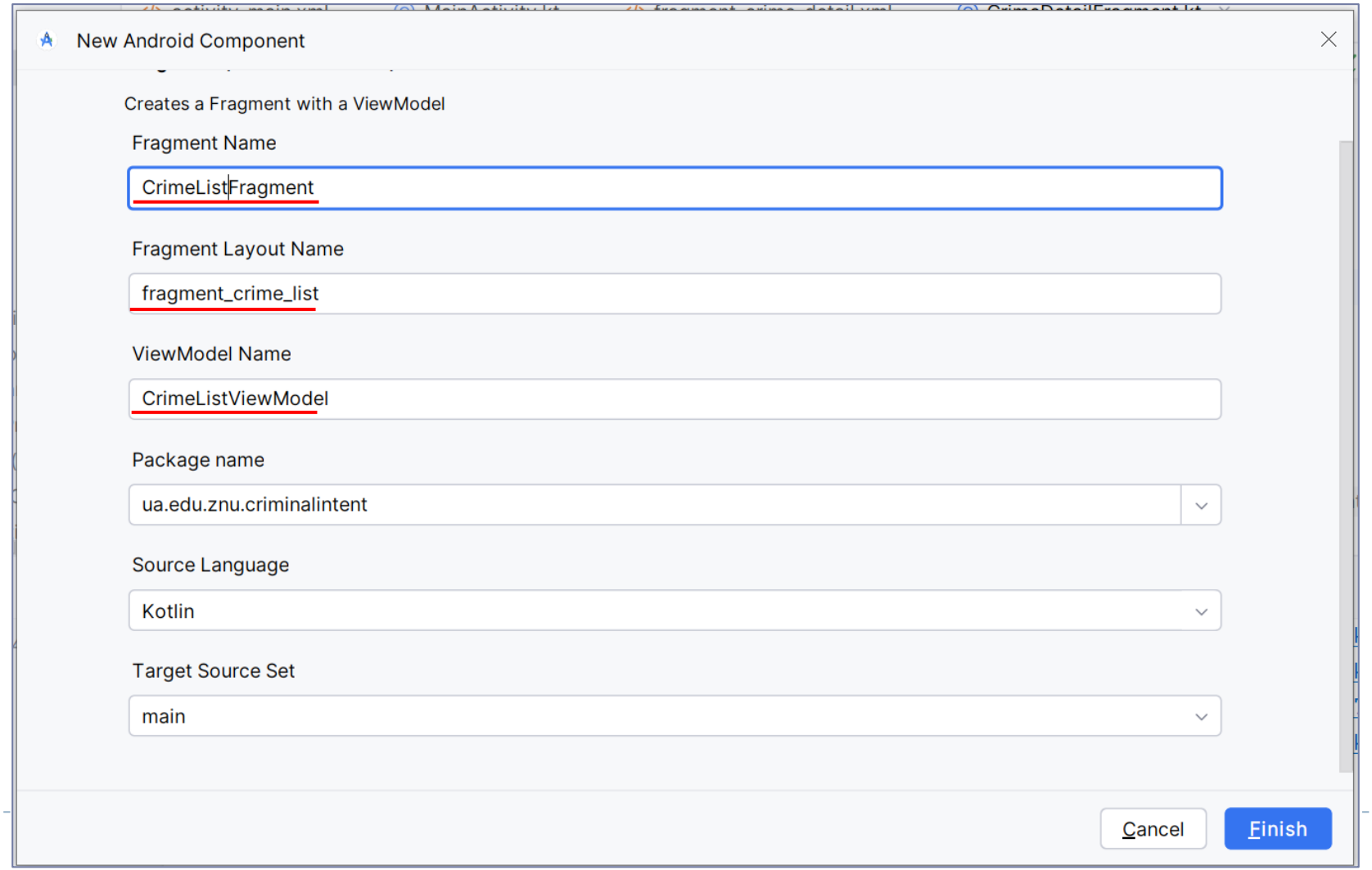
Архітектура застосунку CriminalIntent з підтримкою списку

Додання залежності lifecycle-viewmodel-ktx



Додання фрагменту списку CrimeListFragment

New–Fragment–Fragment (with ViewModel)



Клас ViewModel списку

```
class CrimeListViewModel : ViewModel() {
    val crimes = mutableListOf<Crime>()
    init {
        for (i in 0 until 100) {
            val crime = Crime(
                id = UUID.randomUUID(),
                title = "Crime #$i",
                date = LocalDateTime.now(),
                isSolved = i % 2 == 0
            )
            crimes += crime
        }
    }
}
```

Тестові дані

Клас фрагменту списку CrimeListFragment

```
private const val TAG = "CrimeListFragment"

class CrimeListFragment : Fragment() {
    companion object {
        fun newInstance() = CrimeListFragment()
    }
    private val crimeListViewModel: CrimeListViewModel by viewModels()
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        Log.d(TAG, "Total crimes: ${crimeListViewModel.crimes.size}")
    }
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        return inflater.inflate(R.layout.fragment_crime_list, container, false)
    }  }
```

For now, CrimeListFraglemt
is an empty shell of fragments.

Життєвий цикл ViewModel з фрагментами

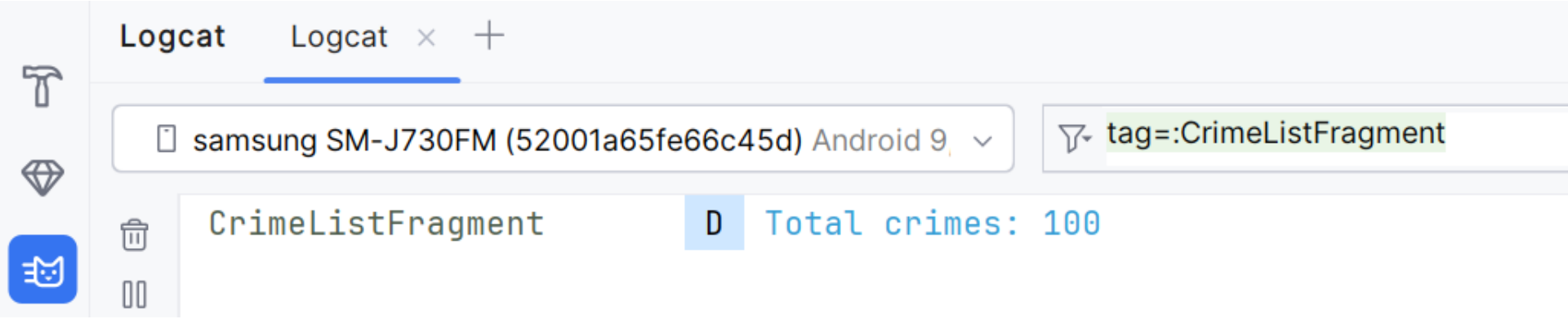
- Об'єкт ViewModel залишатиметься активним, доки віджет фрагмента знаходиться на екрані. ViewModel зберігається при повороті (навіть якщо екземпляр фрагмента не зберігається) і буде доступним для нового екземпляра фрагмента.
- Але об'єкт ViewModel знищується після знищення фрагмента. Це може статися, коли користувач натискає кнопку "Назад", закриваючи екран. Це також може статися, якщо хост-активність замінює фрагмент на інший. Хоча на екрані відображається та ж активність, старий фрагмент, і пов'язаний з ним об'єкт ViewModel будуть знищені, оскільки вони більше не потрібні.
- Є виключення з цієї поведінки – коли активність в рамках транзакції замінює поточний фрагмент іншим, а поточний фрагмент додається до зворотного стеку (методом addToBackSack()), - фрагмент та його ViewModel знищені не будуть. Якщо користувач натискає кнопку "Назад", фрагмент, збережений у стеку, відновлюється. Оригінальний екземпляр фрагмента повертається назад на екран, відтворюючи дані з ViewModel.

Рефакторинг activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/fragment_container"
    android:name="ua.edu.znu.criminalintent.CrimeListFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity"
    tools:layout="@layout/fragment_crime_list">

</androidx.fragment.app.FragmentContainerView>
```

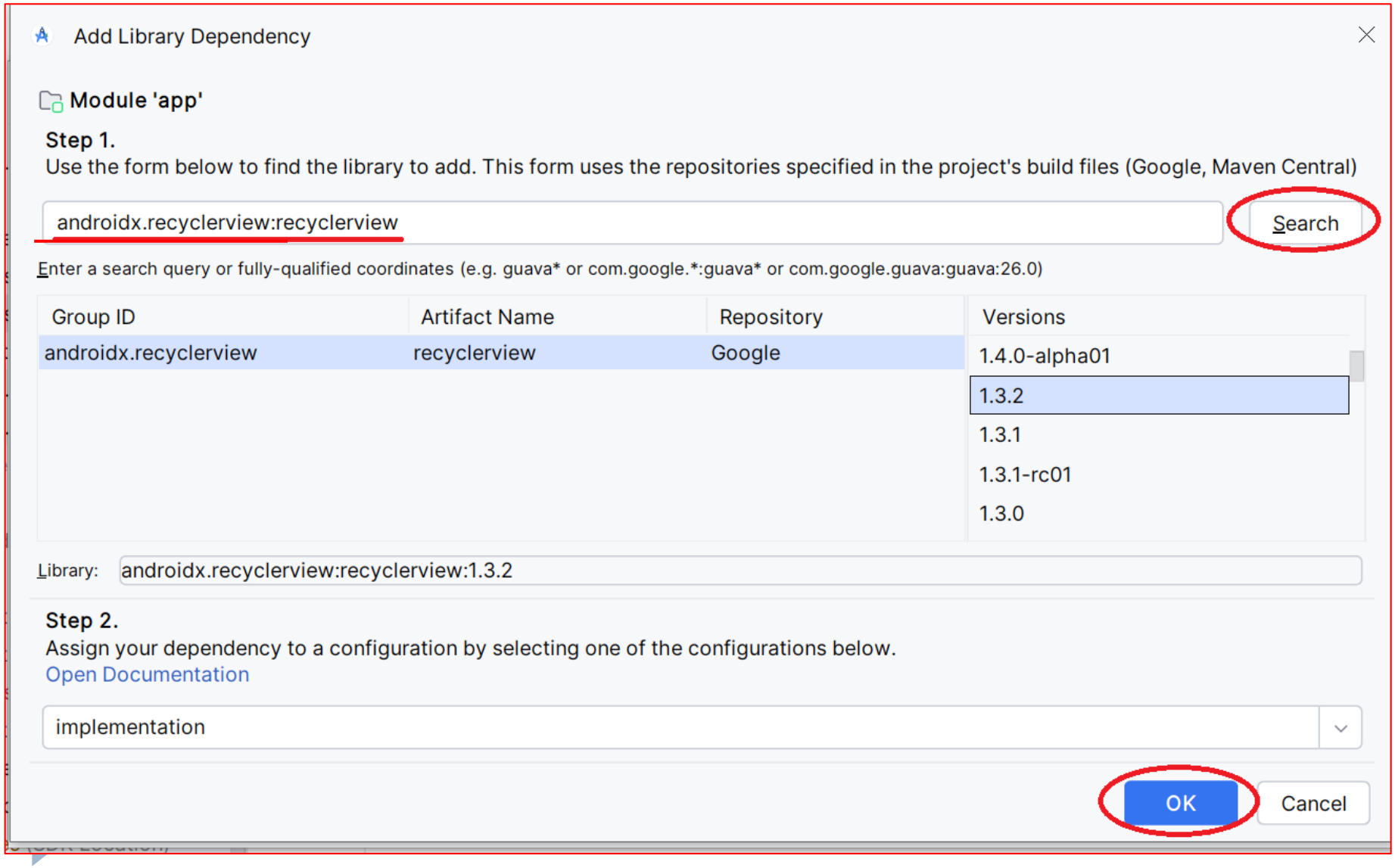
Now MainActivity hosts CrimeListFragment

Перевірка запуску CrimeListFragment в MainActivity



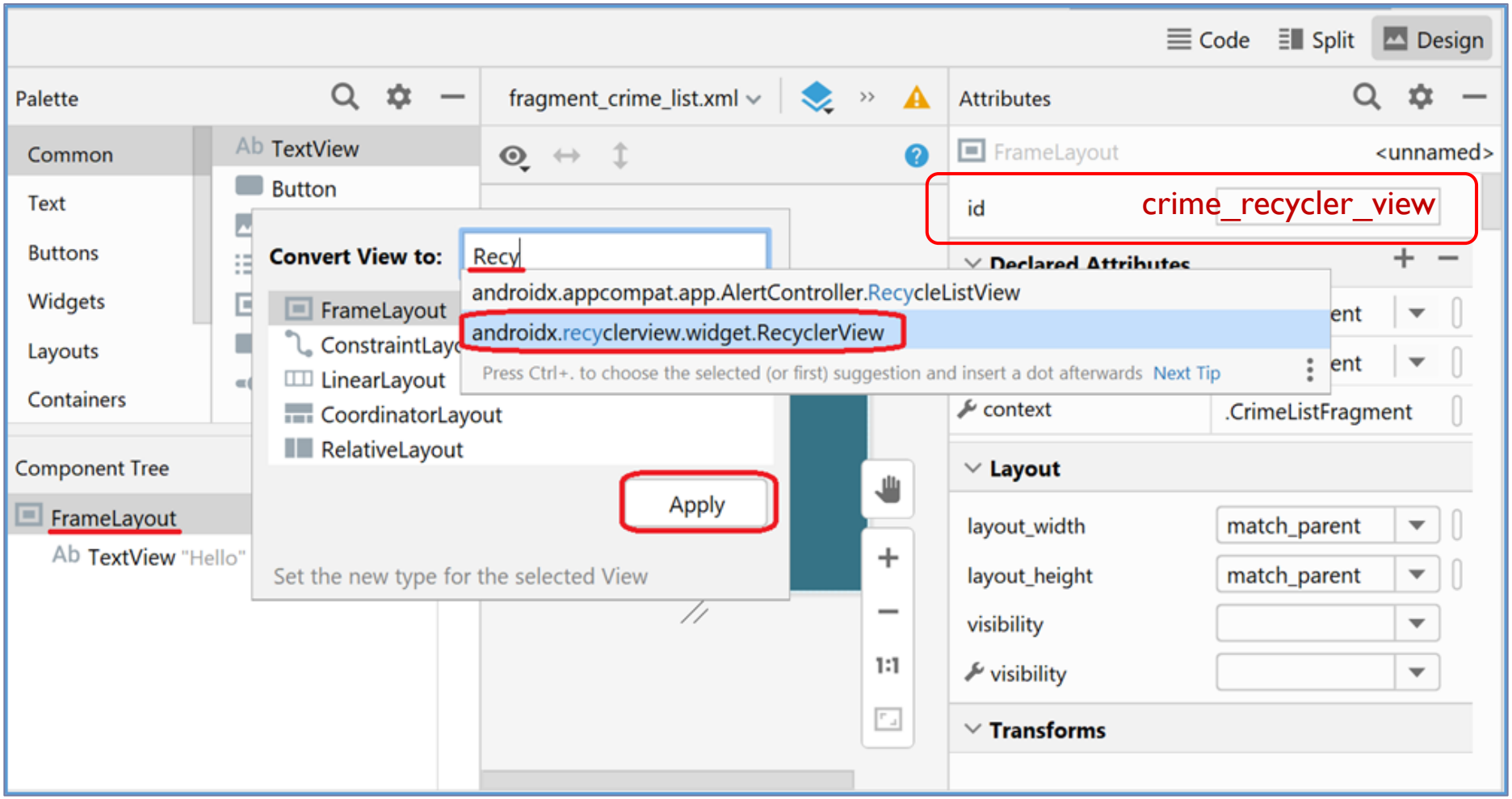
Now MainActivity hosts CrimeListFragment

Додання залежності recyclerview



Відображення списку у RecyclerView

Заміна кореневого елемента fragment_crime_list.xml на androidx.recyclerview.widget.RecyclerView



Макет fragment_crime_list.xml

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

Клас фрагменту списку CrimeListFragment - підключення макета

```
package ua.edu.znu.criminalintent

private const val TAG = "CrimeListFragment"

class CrimeListFragment : Fragment() {
    companion object {
        fun newInstance() = CrimeListFragment()
    }
    private var _binding: FragmentCrimeListBinding? = null
    private val binding
        get() = checkNotNull(_binding) {
            "Cammot access bimdimg because it is mukk. Is the view visibke?"
        }
    private val crimeListViewModel: CrimeListViewModel by viewModels()
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        Log.d(TAG, "Total crimes: ${crimeListViewModel.crimes.size}")
    } ...
```

Клас фрагменту списку CrimeListFragment - підключення макета

```
...
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        _binding = FragmentCrimeListBinding.inflate(inflater, container, false)
        return binding.root
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Реалізація LayoutManager

- RecyclerView — це GroupView, що повторно використовує віджети, відображаючи їх у списку. RecyclerView використовує такі компоненти, як: LayoutManager, ViewHolder і Adapter.
- LayoutManager розміщує кожен елемент, а також визначає, як працює прокручування.
- Існують декілька вбудованих варіантів LayoutManager. Ми будемо використовувати LinearLayoutManager як LayoutManager для RecyclerView. Він розташує елементи у списку вертикально, один за одним, як LinearLayout.

```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                          savedInstanceState: Bundle?): View {
    _binding = FragmentCrimeListBinding.inflate(inflater, container, false)
    binding.crimeRecyclerView.layoutManager =
                                        LinearLayoutManager(context)

    return binding.root
}
```

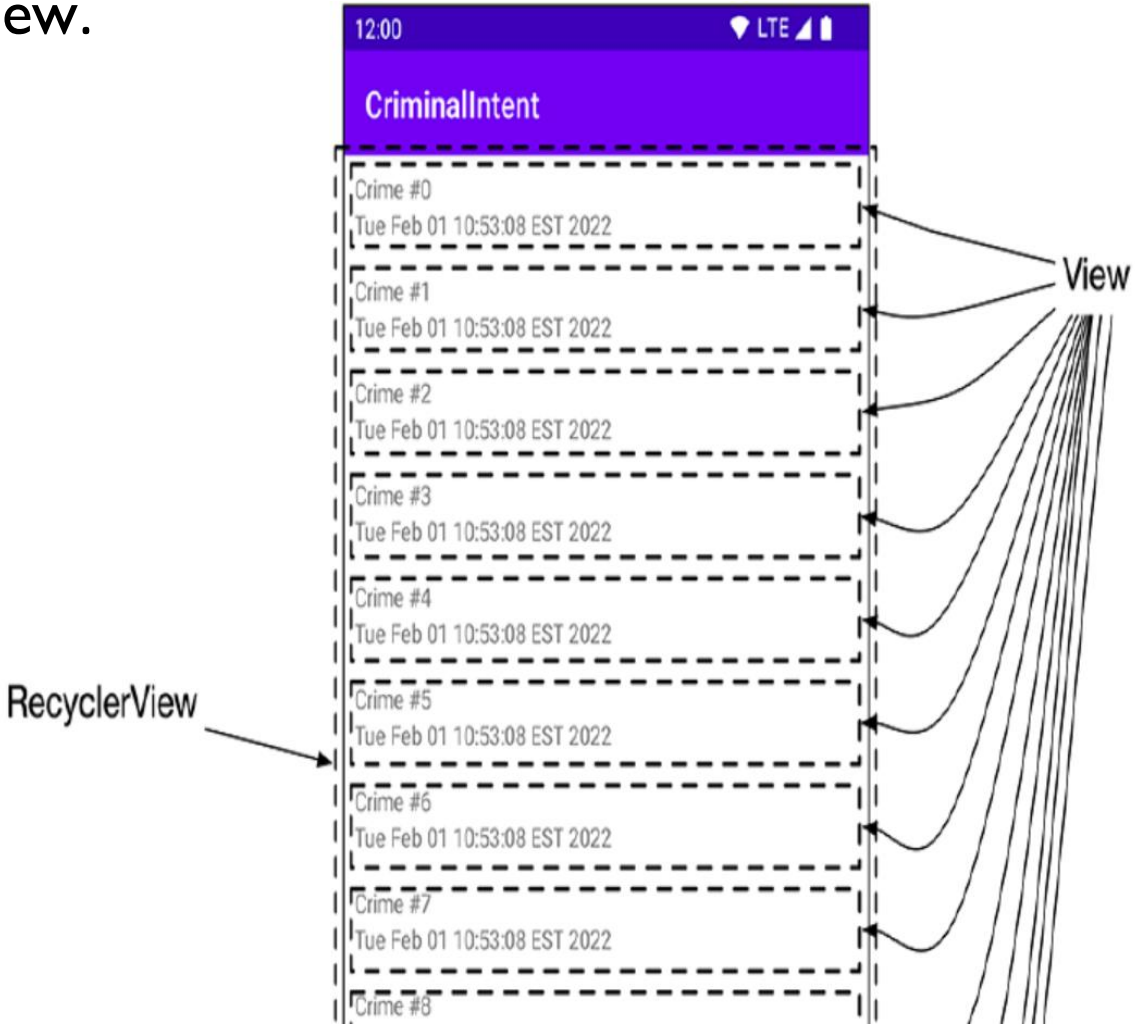
CrimeListFragment.kt

Клас активності

```
package ua.edu.znu.criminalintent

import ...

private const val TAG = "MainActivity"

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        Log.d(TAG, "onCreate called - $this")
        enableEdgeToEdge()
        setContentView(R.layout.activity_main)

        ...
        val fragment = CrimeListFragment()
        supportFragmentManager.beginTransaction()
            .add(R.id.fragment_container, fragment)
            .commit()
    }
}
```

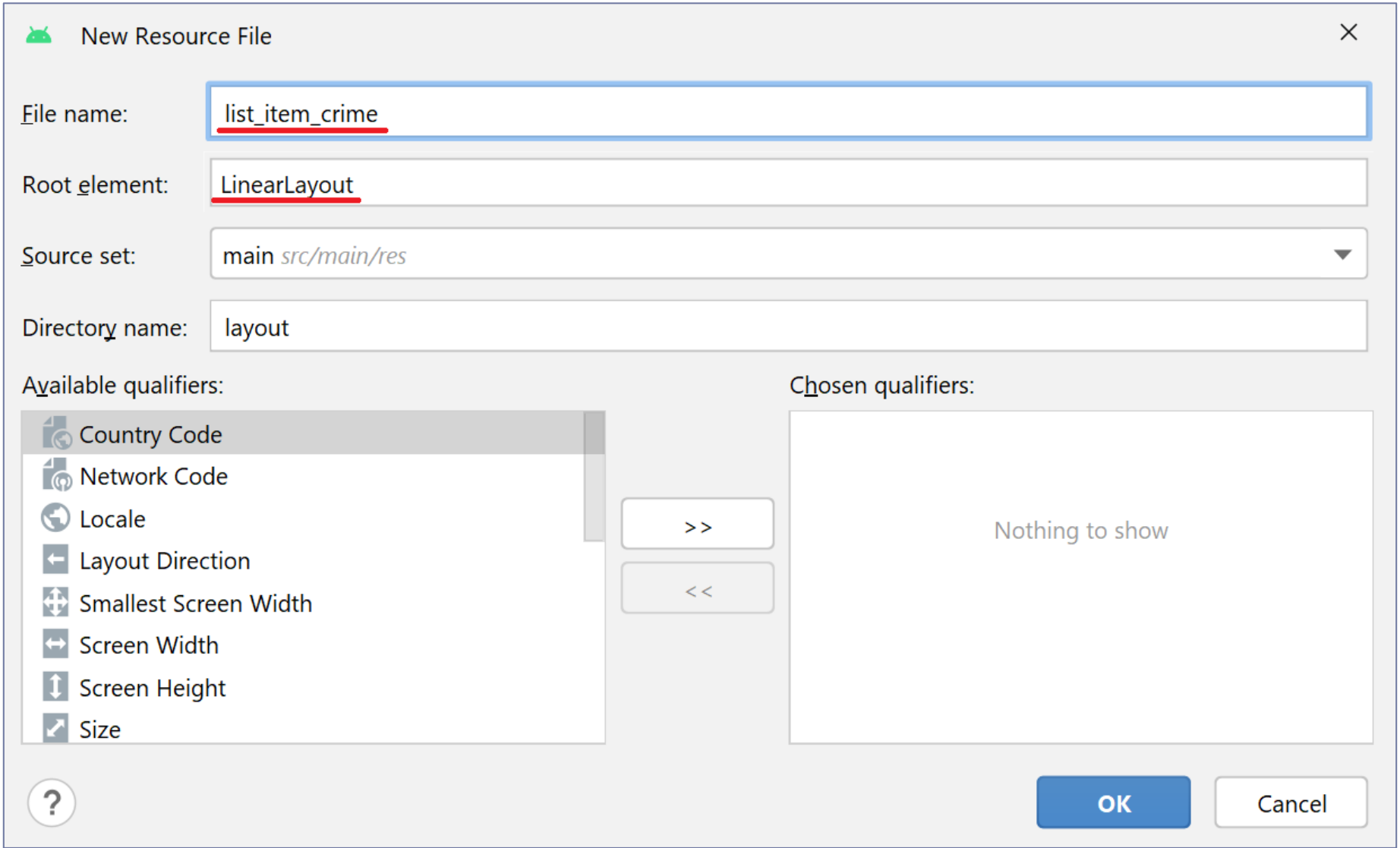
Реалізація Item View Layout

- RecyckerView відображає список дочірніх об’єктів View (items views). Кожен item view представляє окремий об’єкт зі списку RecyckerView.



Макет елемента списку list_item_crime.xml

app\res\layout context menu New–Layout Resource File



Макет елемента списку list_item_crime.xml

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

    <TextView
        android:id="@+id/list_item_crime_title"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/crime_title_label" />

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

доданий рядковий ресурс
(зі значеннями *Date* та *Дата*)

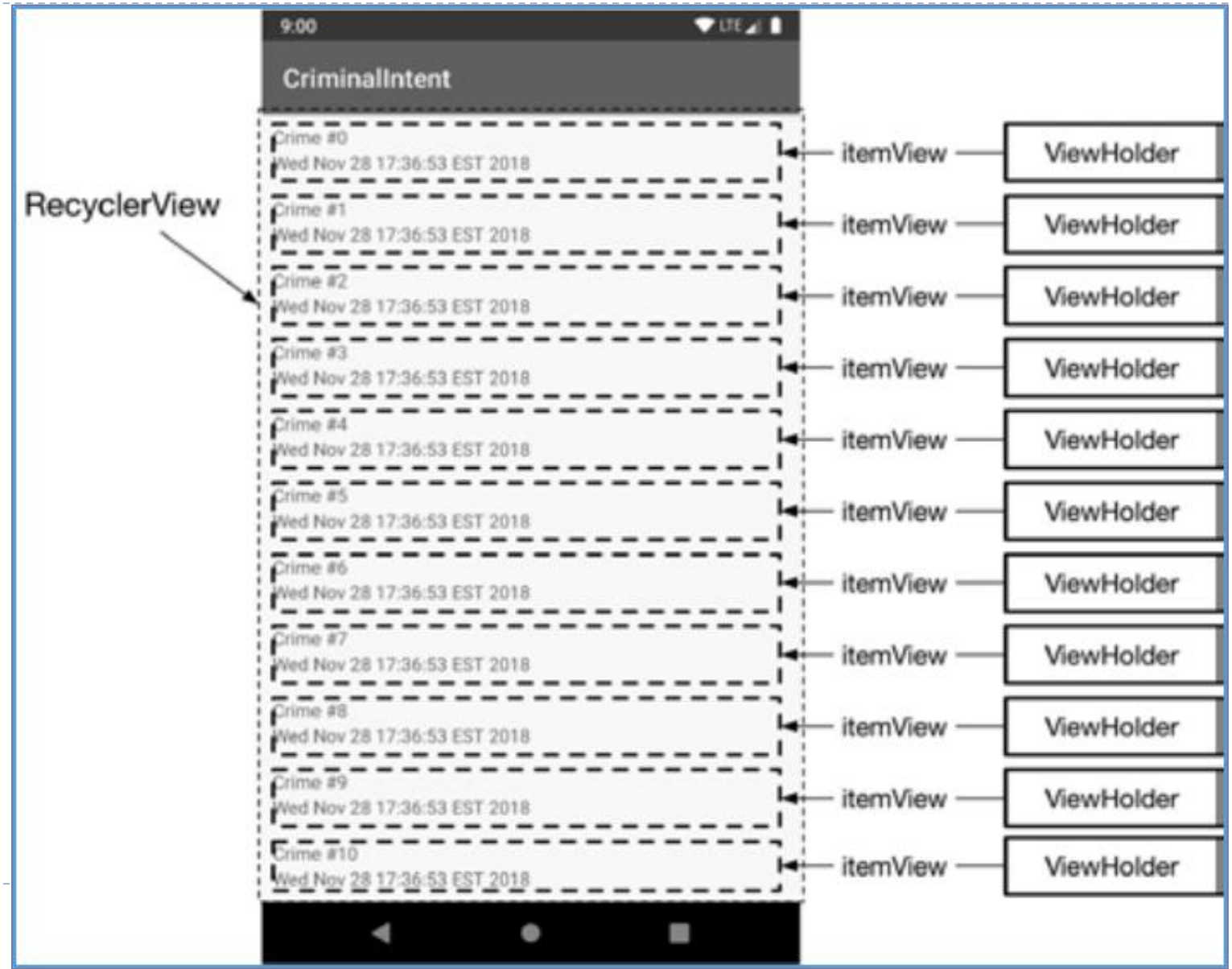
RecylerView.ViewHolder

- Об'єкт RecyclerView очікує, що віджети, які він повинен виводити для кожного елемента списку (item views) будуть надаватися об'єктом RecyclerView.ViewHolder.
- Але безпосередня взаємодія з View не очікується. Натомість буде використовуватися ListItemCrimeBinding.
- Додаймо клас CrimeListAdapter.kt на початку якого оголосимо клас CrimeHolder - нащадок RecyclerView.ViewHolder :

```
package ua.edu.znu.criminalintent

import androidx.recyclerview.widget.RecyclerView
import ua.edu.znu.criminalintent.databinding.ListItemCrimeBinding

class CrimeHolder(
    val binding: ListItemCrimeBinding
) : RecyclerView.ViewHolder(binding.root) {
}
```

RecylerView.ViewHolder

ViewHolder утримуватиме представлення у властивості itemView.



RecyclerView.Adapter<CrimeHolder>

- Насправді об'єкт RecyclerView не створює ViewHolder особисто. Натомість використовується адаптер, який є об'єктом контролера і знаходиться між RecyclerView і наборами даних, які відображає RecyclerView.
- Об'єкт адаптера виконує такі функції:
  - створення за запитом необхідних об'єктів ViewHolder;
  - зв'язування об'єктів ViewHolder з даними шару моделі.
- Об'єкт RecyclerView виконує такі функції:
  - надсилає адаптеру запит на створення нового об'єкта ViewHolder;
  - надсилає адаптеру запит на зв'язування об'єкта ViewHolder з елементом даних, який відповідає поточній позиції.
- Додаймо клас CrimeListAdapter до файлу CrimeListAdapter.kt нижче класу CrimeHolder:

RecyclerView.Adapter<CrimeHolder>

```
package ua.edu.znu.criminalintent

import ...

class CrimeHolder(...){...}

class CrimeListAdapter(
    private val crimes: List<Crime>
) : RecyclerView.Adapter<CrimeHolder>() {
    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int
    ): CrimeHolder {
        val inflater = LayoutInflater.from(parent.context)
        val binding = ListItemCrimeBinding.inflate(inflater, parent, false)
        return CrimeHolder(binding)
    }
```

...

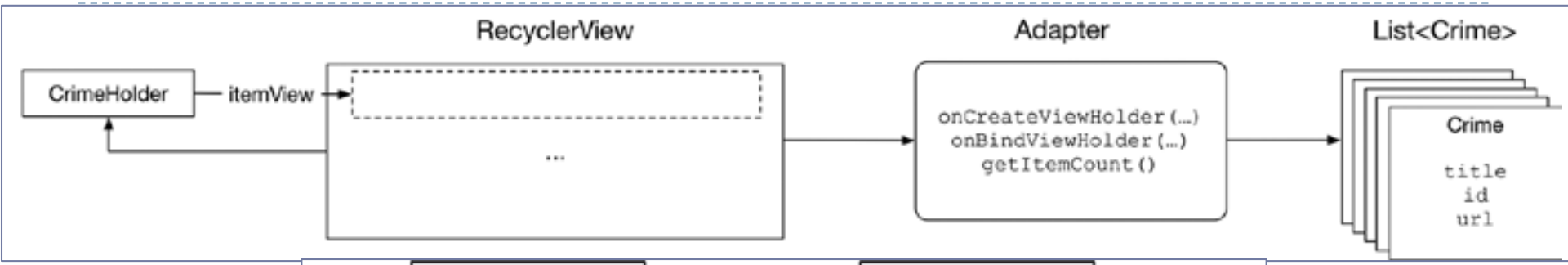
RecyclerView.Adapter<CrimeHolder>

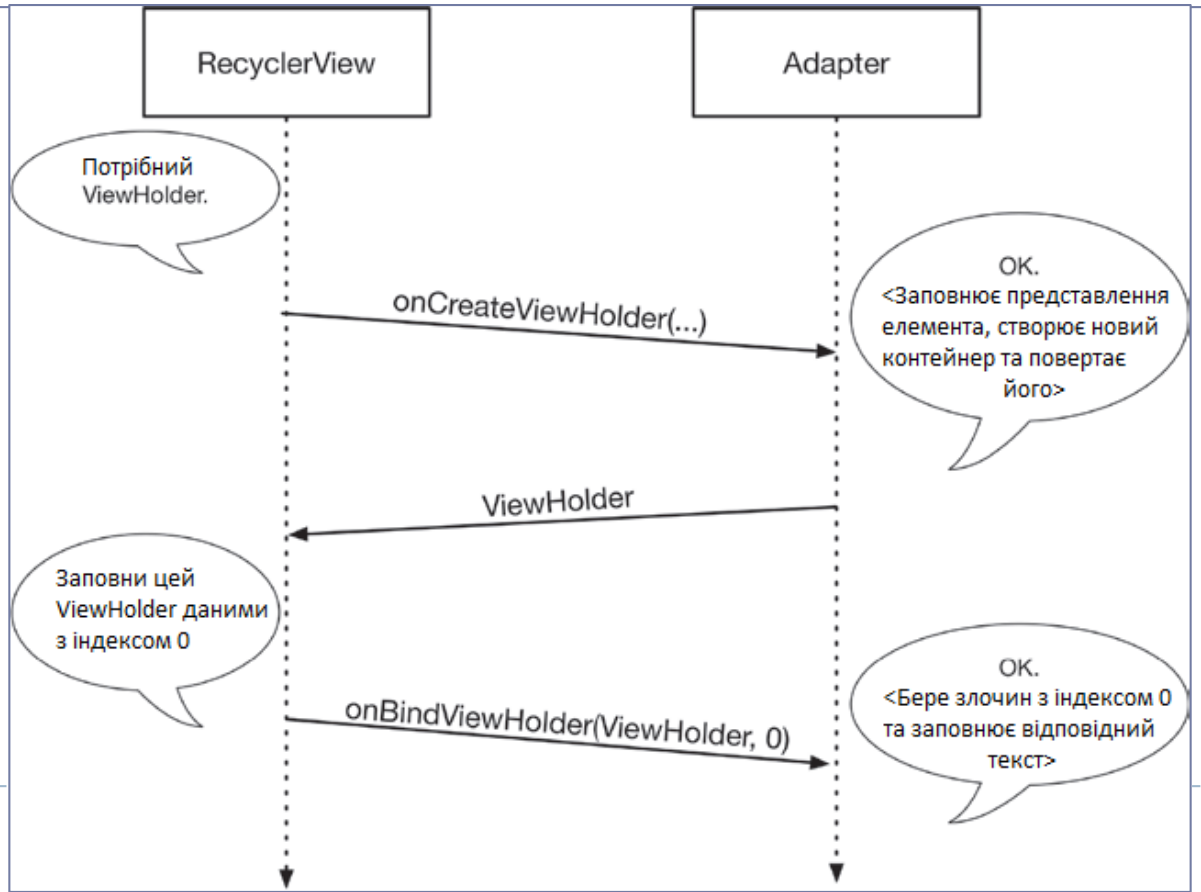
```
...
    override fun onBindViewHolder(holder: CrimeHolder, position: Int) {
        val crime = crimes[position]
        holder.apply {
            binding.listItemCrimeTitle.text = crime.title
            binding.listItemCrimeDate.text = crime.date.toString()
        }
    }

    override fun getItemCount(): Int = crimes.size
}
```

RecyclerView.Adapter<CrimeHolder>

- Метод Adapter.onCreateViewHolder(...) відповідає за створення зв'язування (View Binding) та огортання його у Holder. У нашому застосунку виконується створення віджетів, визначених у list_item_crime.xml за допомогою зв'язування ListItemCrimeBinding, яке огортається об'єктом CrimeHolder. (Поки що Ви можете ігнорувати параметри onCreateViewHolder(...). Ці значення потрібні, лише якщо Ви робите щось неординарне, наприклад відображення різних типів представлень у одному RecyclerView.
- Adapter.onBindViewHolder(...) відповідає за заповнення CrimeHolder даними злочину із заданої позиції у списку.
- Adapter.getItemCount() викликається, коли компоненту RecyclerView потрібно знати, скільки елементів у наборі даних, що він обробляє (наприклад, коли RecyclerView вперше запускається).
- Сам RecyclerView нічого не знає про об’єкт Crime або список об’єктів Crime, які мають відображатися. Це знає CrimeListAdapter.

Робота адаптера





Клас фрагменту списку CrimeListFragment - додання адаптера

```
class CrimeListFragment : Fragment() {
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        _binding = FragmentCrimeListBinding.inflate(inflater, container, false)
        binding.crimeRecyclerView.layoutManager =
                                                LinearLayoutManager(context)
    ...
        /*Set RecyclerView adapter*/
        var crimes = crimeListViewModel.crimes
        val adapter = CrimeListAdapter(crimes)
        binding.crimeRecyclerView.adapter = adapter
        return binding.root
} ...   }
```

Відображення злочинів у RecycleView

RecyclerView

- Замість створення 100 об'єктів View, RecyclerView створює їх стільки, скільки потрібно для заповнення екрана. Коли елемент зникає з екрана, RecyclerView ***використовує його представлення для інших елементів (recycle)***. Коли створено достатньо об'єктів ViewHolder для заповненя екрану, RecyclerView перестає викликати onCreateViewHolder(..). Натомість він заощаджує час і пам'ять шляхом ***повторного використання старих об'єктів ViewHolder*** і передає їх до onBindViewHolder(ViewHolder, Int).
- В даний момент Adapter прив'язує дані злочину до текстових віджетів у функції Adapter.onBindViewHolder(...). Але краще розділити завдання між холдером та адаптером: адаптер повинен знати якомога менше про внутрішню кухню і дані холдера. Тому перенесемо код, який виконує прив'язку даних злочину до текстових віджетів до CrimeHolder:

Перенесення прив'язки даних до віджетів з CrimeListAdapter до CrimeHolder

```
package ua.edu.znu.criminalintent

import ...

class CrimeHolder(
    val binding: ListItemCrimeBinding
) : RecyclerView.ViewHolder(binding.root) {
    fun bind(crime: Crime) {
        binding.apply {
            listItemCrimeTitle.text = crime.title
            listItemCrimeDate.text = crime.date.toString()
        }
    }
}

...
```

Перенесення прив'язки даних до віджетів з CrimeListAdapter до CrimeHolder

```
...
class CrimeListAdapter(
    private val crimes: List<Crime>
) : RecyclerView.Adapter<CrimeHolder>() {
    override fun onCreateViewHolder(...) : CrimeHolder {
    ...
    }
    override fun onBindViewHolder(holder: CrimeHolder, position: Int) {
        val crime = crimes[position]
//          holder.apply {
//              binding.listItemCrimeTitle.text = crime.title
//              binding.listItemCrimeDate.text = crime.date.toString()
//          }
        holder.bind(crime)
    }
    ...
}
```

Додання обробки події кліку на елементі списку RecyclerView

CrimeListAdapter.kt

```
package ua.edu.znu.criminalintent

import ...

class CrimeHolder(
    val binding: ListItemCrimeBinding
) : RecyclerView.ViewHolder(binding.root) {
    fun bind(crime: Crime) {
        binding.apply {
            listItemCrimeTitle.text = crime.title
            listItemCrimeDate.text = crime.date.toString()
            binding.root.setOnClickListener {
                Toast.makeText(
                    binding.root.context,
                    "${crime.title} clicked!",
                    Toast.LENGTH_SHORT
                ).show()
    }       }    } } ... }
```

Додання обробки події кліку на елементі списку RecyclerView

Lists та Grids: минуле, теперішнє та майбутнє

- Ядро ОС Android включає класи ListView, GridView та Adapter, які до Android 5.0 були основними засобами створення списків або сіток елементів. API для цих компонентів дуже схожий на API RecyclerView: ListView або GridView відповідає за прокручування колекції елементів, але вони "не знають" багато про ці елементи, делегуючи це Адаптеру. Однак ListView та GridView не зобов'язують використовувати ViewHolder (хоча Ви можете – і повинні – його використовувати).
- Ці старі реалізації замінено на RecyclerView через складність, необхідну для зміни поведінки ListView або GridView. Наприклад, створення ListView з горизонтальним прокручуванням не входить до ListView API і потребує багато роботи. А RecyclerView можна використовувати з GridLayoutManager для впорядкування елементів у сітці. ...

Lists та Grids: минуле, теперішнє та майбутнє

...

- Ще одна ключова функція RecyclerView - анімація елементів у списку. Анімація додавання або видалення елементів у ListView або GridView є складним і схильним до помилок завданням. RecyclerView робить це набагато простіше; він містить кілька вбудованих анімацій і дозволяє легко налаштовувати ці анімації. Наприклад, якщо ви дізналися, що злочин із позиції 0 перемістився на позицію 5, ви можете анімувати цю зміну так:

```
binding.crimeRecyclerView.adapter?.notifyItemMoved(0, 5)
```

- RecyclerView є потужним і розширюваним, але він також складний і вимагає багато налаштувань навіть для простих інтерфейсів користувача. За допомогою Jetpack Compose Ви маєте доступ до складових елементів LazyColumn і LazyRow. Ці composables мають усі можливості налаштування та продуктивність RecyclerView, але їх можна створити малою кількістю коду.

Рефакторинг обробки події кліку на елементі списку RecyclerView для заміни фрагменту

- Додаймо інтерфейс OnCrimeClickListener до файлу CrimeListAdapter.kt та виконаємо рефакторинг класів CrimeHolder та CrimeListAdapter, щоб вони використовували цей інтерфейс:

CrimeListAdapter.kt

```
package ua.edu.znu.criminalintent

import ...

interface OnCrimeClickListener {
    fun onCrimeClick(crimeId: UUID)
}

...
```

Рефакторинг обробки події кліку на елементі списку RecyclerView для заміни фрагменту

```
...
class CrimeHolder(
    private val binding: ListItemCrimeBinding,
    private val crimeClickListener: OnCrimeClickListener
) : RecyclerView.ViewHolder(binding.root) {
    fun bind(crime: Crime) {
        binding.apply {
            listItemCrimeTitle.text = crime.title
            listItemCrimeDate.text = crime.date.toString()
            binding.root.setOnClickListener {
                crimeClickListener.onCrimeClick(crime.id)
            }
        }
    }
}
...
```

CrimeListAdapter.kt

Рефакторинг обробки події кліку на елементі списку RecyclerView для заміни фрагменту

```
...
class CrimeListAdapter(
    private val crimes: List<Crime>,
    private val crimeClickListener: OnCrimeClickListener
) : RecyclerView.Adapter<CrimeHolder>() {
    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int
    ): CrimeHolder {
        val inflater = LayoutInflater.from(parent.context)
        val binding = ListItemCrimeBinding.inflate(inflater, parent, false)
        return CrimeHolder(binding, crimeClickListener)
    }
...
}
```

CrimeListAdapter.kt

Рефакторинг обробки події кліку на елементі списку RecyclerView для заміни фрагменту

- Виконаємо рефакторинг CrimeDetailFragment так, щоб при створенні його екземпляру приймався параметр crimeId:

CrimeDetailFragment.kt

```
package ua.edu.znu.criminalintent

import ...

private const val TAG = "CrimeDetailFragment"
private const val ARG_CRIME_ID = "crime_id"

class CrimeDetailFragment : Fragment() {

    private var _binding: FragmentCrimeDetailBinding? = null
    private val binding get() = _binding!!

    private lateinit var crime: Crime

...
```

Рефакторинг обробки події кліку на елементі списку RecyclerView для заміни фрагменту

```
...
    companion object {
        fun newInstance(crimeId: UUID): CrimeDetailFragment {
            val args = Bundle().apply {
                putSerializable(ARG_CRIME_ID, crimeId)
            }
            return CrimeDetailFragment().apply {
                arguments = args
            }
        }
    }
...
}
```

CrimeDetailFragment.kt

Рефакторинг обробки події кліку на елементі списку RecyclerView для заміни фрагменту

- Реалізуємо інтерфейс OnCrimeClickListener у класі CrimeListFragment, визначивши заміну фрагменту CrimeListFragment на фрагмент CrimeDetailFragment:

CrimeListFragment.kt

```
package ua.edu.znu.criminalintent

import ...

private const val TAG = "CrimeListFragment"

class CrimeListFragment : Fragment(), OnCrimeClickListener {

    ...

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
    ...
```

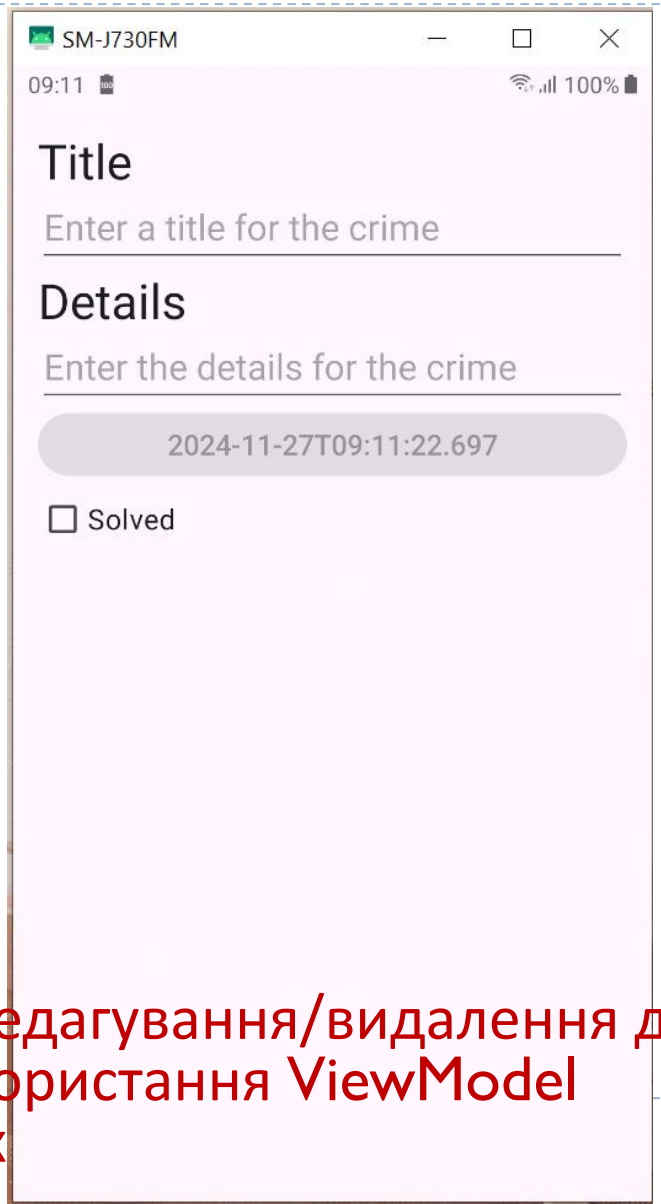
Рефакторинг обробки події кліку на елементі списку RecyclerView для заміни фрагменту

```
...
    _binding = FragmentCrimeListBinding.inflate(inflater, container, false)
    binding.crimeRecyclerView.layoutManager =
                                        LinearLayoutManager(context)

    val crimes = crimeListViewModel.crimes
    val adapter = CrimeListAdapter(crimes, this)
    binding.crimeRecyclerView.adapter = adapter
    return binding.root
}

override fun onCrimeClick(crimeId: UUID) {
    val fragment = CrimeDetailFragment.newInstance(crimeId)
    parentFragmentManager.beginTransaction()
        .replace(R.id.fragment_container, fragment)
        .addToBackStack(null)
        .commit()
}
... }
```

CrimeListFragment.kt

Заміна фрагменту у контейнері

Рефакторинг обробки події кліку на елементі списку RecyclerView для заміни фрагменту



Додавання/редагування/видалення даних потребує використання ViewModel та бази даних