Тема 6. Макетування інтерфейсу мобільного додатку на основі фрагментів

Фрагменти були введені в Android SDK разом з Android 3.0 у відповідь на появну планшетів з даною операційною системою. Мобільний інтерфейс тогочасних додатків виглядав на великих екранах незручним та застарілим, тому потрібно було додати можливість перебудови інтерфейсу на екранах різних розмірів та роздільних здатностей. При проєктуванні розробники розглядали фрагменти як мікроактивності, тому й нині фрагменти успадковують більшість нововведень, які стосуються активностей.

Фрагменти існують в API платформи Android у трьох інкарнаціях:

- нативні фрагменти – пакет android.app.Fragment;
- фрагменти з Support Library – пакет android.support.v4.app.Fragment;
- AndroidX-фрагменти – пакет androidx.fragment.app.Fragment.

У часи створення фрагментів перед компанією Google стояла задача надати можливість використовувати фрагменти не лише на пристроях з Android 3, але й більш старими пристроями (на той час таких була переважна більшість). Вирішенням стало створення бібліотеки зворотної підтримки (Support Library), яка містила адаптований форк нативної реалізації фрагментів. Таким чином, для систем Android 3+ пропонувалась вбудована підтримка фрагментів, а за потреби розширення підтримки на старіших платформах використовувалась альтернатива з бібліотеки підтримки.

Проблемою нативної реалізації фрагментів з самого початку стала її значна забагованість. Вона не отримувала найсвіжіших оновлень, оскільки була частиною операційної системи Android, а баги в різних версіях могли значно відрізнятись! У той же час фрагменти з бібліотеки підтримки завжди працювали однаково незалежно від версії ОС та містили свіжі виправлення помилок. У результаті розробники переважно віддавали перевагу версії фрагментів з бібліотеки підтримки, а нативна реалізація фрагментів взагалі вважається застарілою з кінця 2018 року (API level 28). Повна відмова від використання нативних фрагментів відбулась, починаючи з Android 10.

З переходом до Jetpack-компонентів з’явилась AndroidX-версія API фрагментів. Вона стала значним кроком вперед, оскільки містить не лише існуючу реалізацію бібліотеки зворотної підтримки, але й інші Jetpack-компоненти. Крім того, внутрішня реалізація фрагментів зазнала значного рефакторингу у версії Fragment 1.3.0 (початок 2021 року). Була додана інтеграція з новими API, а багато вже існуючих стали вважатись застарілими. Також представили новий механізм комунікації між фрагментами.

Класичний життєвий цикл фрагмента частково відповідає життєвому циклу активності, проте також вносить власні методи зворотного виклику (рис. 6.1). Активність виступає хостом для фрагментів, розміщуючи на собі один або декілька з них. Модульність фрагментів також вказує на те, що через активність

відбувається комунікація між ними. Розглянемо роботу представлених на рис. 6.1 методів:

- ***onAttach()***: фрагмент прикріплюється до свої активності-хоста;
- ***onCreate()***: ініціалізує екземпляр фрагмента після прикріплення;
- ***onCreateView()***: відповідає за рендеринг ієрархії представлень (view hierarchy) фрагмента, тобто дорисовує з визначеного місця графічний інтерфейс, передбачений відповідною розміткою;
- ***onActivityCreated()***: викликається при завершенні роботи методу onCreate() від активності-хоста;
- ***onStart()***: фрагмент стає видимим. Запуск фрагменту відбувається після запуску активності-хоста;
- о***nResume()***: фрагмент уже видимий та з ним можна взаємодіяти. Фрагмент відновлює (resume) роботу після аналогічних дій активності-хоста.

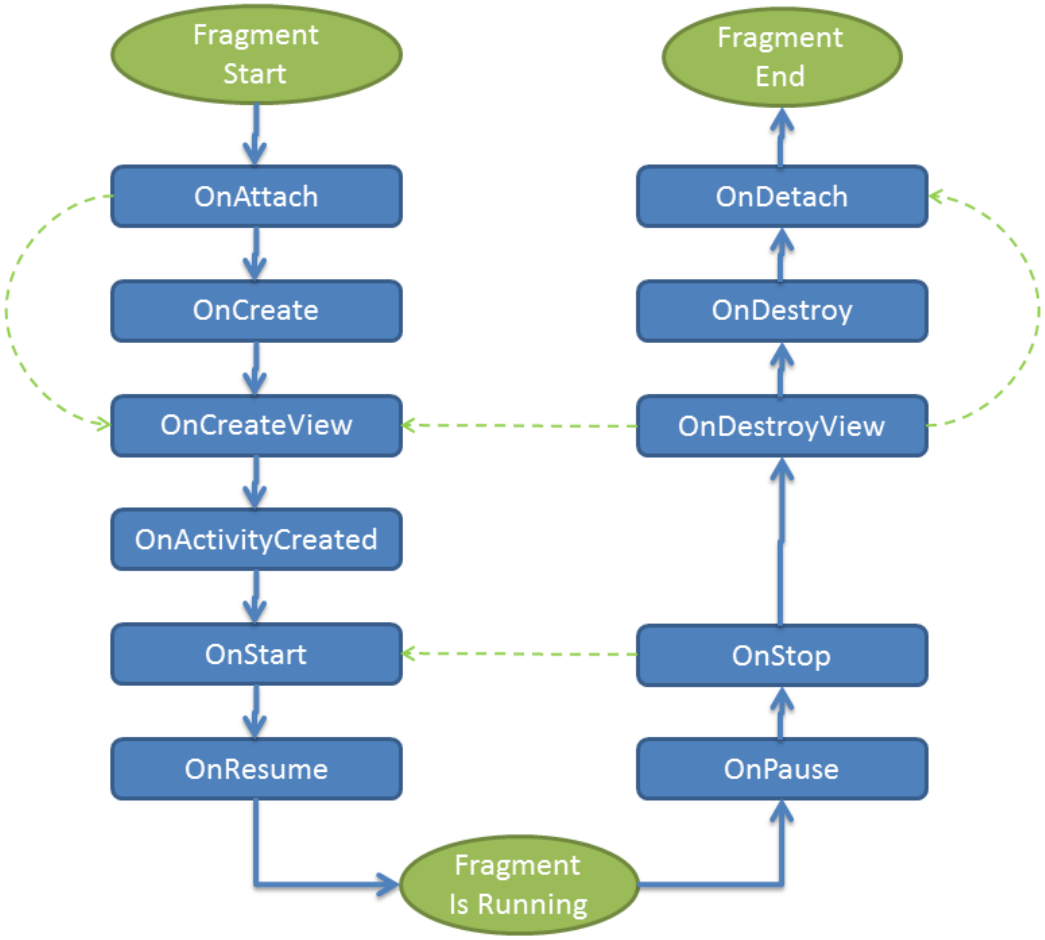

Рис. 6.1. Життєвий цикл фрагмента

Представлені методи спричиняють додавання фрагменту до решти графічного інтерфейсу та взаємодію з ним. Проте також передбачена можливість видалення фрагменту, що в подальшому може призвести до його реініціалізації або заміні на інший фрагмент. Для цього передбачені решта методів:

- ***onPause()***: фрагмент перестає бути інтерактивним. Це перший крок з підготовки до видалення або заміни фрагмента чи у випадку призупинки роботи активності-хоста;
- ***onStop()***: фрагмент перестає бути видимим;
- ***onDestroyView()***: ієрархія представлень (графічний інтерфейс) фрагмента та пов'язані з ним ресурси видаляються з ієрархії представлень активності-хоста та знищуються;
- ***onDestroy()***: остаточна очистка фрагмента;
- ***onDetach()***: відкріплення фрагмента від активності-хоста.

Вихід Jetpack-компонентів дозволив перекласти частину логіки роботи фрагментів на архітектурні компоненти Jetpack. Зокрема, застосування LifecycleObserver для відстежування стану фрагмента спрощує обробку життєвого циклу (рис. 6.2).

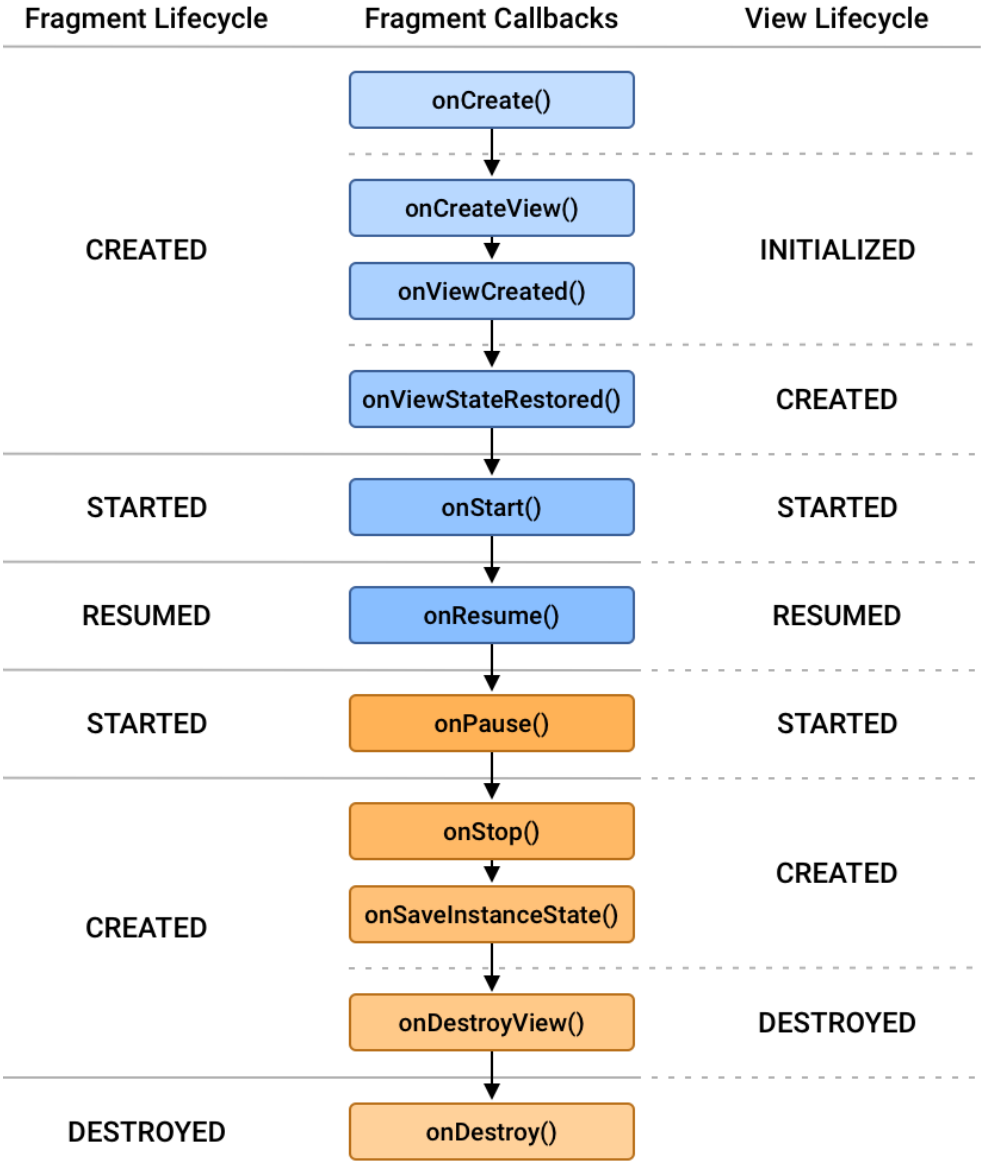

Рис. 6.2. Відношення між Lifecycle-станами фрагмента, його методами життєвого циклу та станами відповідного View-компонента

https://medium.com/androiddevelopers/fragments-rebuilding-the-internals-61913f8bf48e
https://developer.android.com/guide/fragments/lifecycle

Створення та додавання фрагментів

Для демонстрації впровадження фрагментів у додаток розглянемо інтерфейс, який складатиметься з двох фрагментів, що будуть відображатись при натисненні на відповідні кнопки (рис. 6.3).

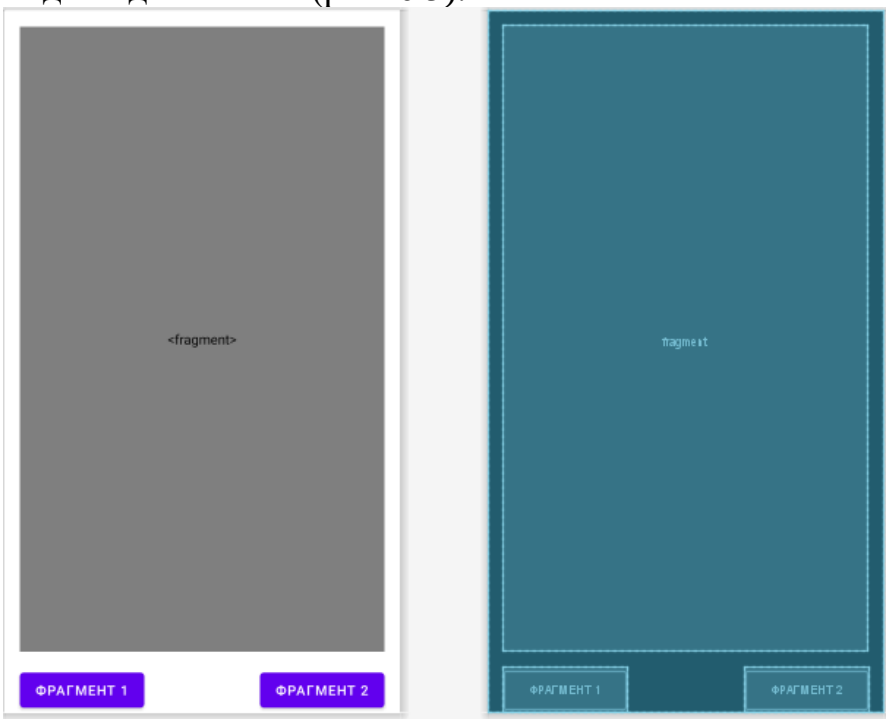

Рис. 6.3. Макет демонстраційного додатку FragmentsDemo

Зараз для основи реалізації фрагмента використовується *статичний фрагмент* – елемент <fragment>. На противагу цьому існують *динамічні фрагменти* на основі <FrameLayout> або <FragmentContainerView>. Слід зауважити, що відмінностей у ході створення таких фрагментів немає. Вони виникають у процесі використання фрагментів.

Статичний фрагмент включається в XML-розмітку активності. Наприклад, у поточній версії макету подібна розмітка виглядає так:

```
<fragment
    android:id="@+id/fragment_container"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    ... />
```

Статично описаний фрагмент неможливо замінити іншим фрагментом у ході виконання додатку. Такій заміні підлягають лише фрагменти, що додавались динамічно. Стандартним підходом є заміна тегу <fragment> на диспетчер компонувань <FrameLayout>, проте нині рекомендується використовувати новий контейнер, представлений у 2019 році в Fragment 1.2.0, – FragmentContainerView. Даний контейнер є кастомним представленням, побудованим на основі FrameLayout, виключно для роботи з фрагментами. Він не є заміною іншим диспетчерам компонувань, оскільки обробляє лише атрибути, які стосуються фрагментів. Для використання FragmentContainerView слід підключити наступну залежність:

```
implementation 'androidx.fragment:fragment-ktx:1.3.3'
```

На виділене контейнером місце будемо підвантажувати один з двох фрагментів залежно від натисненої кнопки. Android Studio містить майстер зі створення фрагментів (рис. 6.4). Зверніть увагу, що він генерує багато коду стосовно логіки роботи фрагменту в відповідному kt-файлі. На даний момент необхідною є лише присутність методу зворотного виклику OnCreateView().

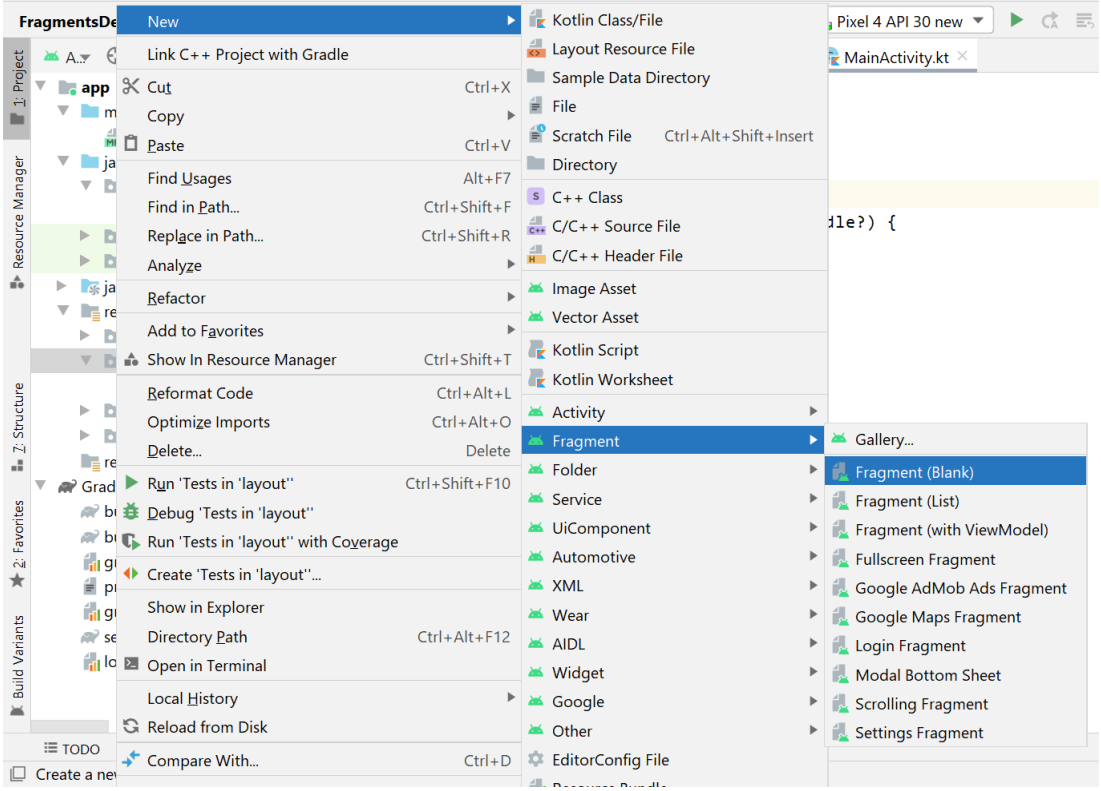

Рис. 6.4. Меню для створення фрагментів

Додамо два фрагменти: FirstFragment та SecondFragment. Приклад коду реалізації першого фрагменту наведено в лістингу 6.1. Зверніть увагу, що фрагмент має окрему розмітку (XML-файл) та окрему логіку (kt-файл) аналогічно до реалізації активності. Зв'язок між цими файлами записується у рядку 9 лістингу.

Лістинг 6.1. Код реалізації першого фрагменту

fragment_first.xml

```
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/frameLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".FirstFragment">

    <!-- TODO: Update blank fragment layout -->
    <TextView
    android:id="@+id/textView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:textSize="24sp"
    android:text="Ви перейшли на фрагмент 1"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

FirstFragment.kt

```
class FirstFragment : Fragment() {
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.fragment_first,
                                container, false)
    }
}
```

Реалізацію логіки фрагмента було зведено до вказівок щодо рендерингу за допомогою методу inflate() об’єкта inflater (рядки 28-33) в методі OnCreateView(). Дані налаштування визначають розмітку першого фрагменту як базу для рендерингу, контекст для відображення (контейнер, у якому розташовано фрагмент) та можливість дорисовки інтерфейсу (false – фрагмент буде відрисовуватись з нуля).

Фінальним кроком є заповнення контейнера фрагментів одним з наявних фрагментів. Оскільки структурно фрагмент може складатись з багатьох елементів інтерфейсу з власною логікою роботи, операційна система пропонує менеджер фрагментів, який буде виконувати операції додавання, видалення та заміни фрагментів. Атомарні операції з інкапсульованим фрагментом здійснюються за допомогою об’єкта supportFragmentManager (лістинг 6.2).

Лістинг 6.2. Код реалізації активності-хосту

activity_main.xml

```
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/fragment_container"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_marginStart="16dp"
        android:layout_marginTop="16dp"
        android:layout_marginEnd="16dp"
        android:layout_marginBottom="16dp"
        app:layout_constraintBottom_toTopOf="@+id/btnFrag1"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <Button
        android:id="@+id/btnFrag1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:text="Фрагмент 1"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent" />

    <Button
        android:id="@+id/btnFrag2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginEnd="16dp"
        android:text="Фрагмент 2"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>

```

MainActivity.kt

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val btn1 = findViewById<Button>(R.id.btnFrag1)
        val btn2 = findViewById<Button>(R.id.btnFrag2)

```

```
50          if (savedInstanceState == null) {
51              supportFragmentManager.commit {
52                  setReorderingAllowed(true)
53                  add<FirstFragment>(R.id.fragment_container)
54              }
55          }
56
57          btn1.setOnClickListener {
58              supportFragmentManager.commit {
59                  replace<FirstFragment>(R.id.fragment_container)
60                  setReorderingAllowed(true)
61                  addToBackStack("Fragment 1")
62              }
63          }
64
65          btn2.setOnClickListener {
66              supportFragmentManager.commit {
67                  replace<SecondFragment>(R.id.fragment_container)
68                  setReorderingAllowed(true)
69                  addToBackStack("Fragment 2")
70              }
71          }
72      }
73  }
```

У рядках 50-55 вирішується проблема первинного заповнення контейнеру при запуску додатку. Якщо неможливо відновитись з попереднього стану, supportFragmentManager фіксує фрагмент, який буде завантажуватись на порожнє місце (тут це перший фрагмент). Метод setReorderingAllowed() визначає дозвіл на оптимізацію операцій всередині транзакцій та між ними.

Обробники натиснення на кнопки націлені на заміну поточного фрагменту в контейнері на відповідний фрагмент. Тому застосовуються метод replace() для заміни та addToBackStack(), щоб додати фрагмент (з назвою) в бекстек фрагментів. Вигляд додатку в дії представлено на рис. 6.5.

Після значного рефакторингу у Fragment 1.3.0 (початок 2021 року) клас FragmentManager було розбито на менші, краще тестовані та простіші в супроводі класи. Ядром цих класів став FragmentStateManager. Новий менеджер стану фрагментів містить наступні ключові частини:

- переміщення стану фрагменту між його методами життєвого циклу;
- запуск анімацій та переходів;
- обробка відкладених транзакцій.

За результатами рефакторингу було закрито більше десятка довготривалих проблем з фрагментами, спрощено життєвий цикл фрагментів та впровадження підтримки кількох бекстеків у межах одного менеджеру фрагментів.

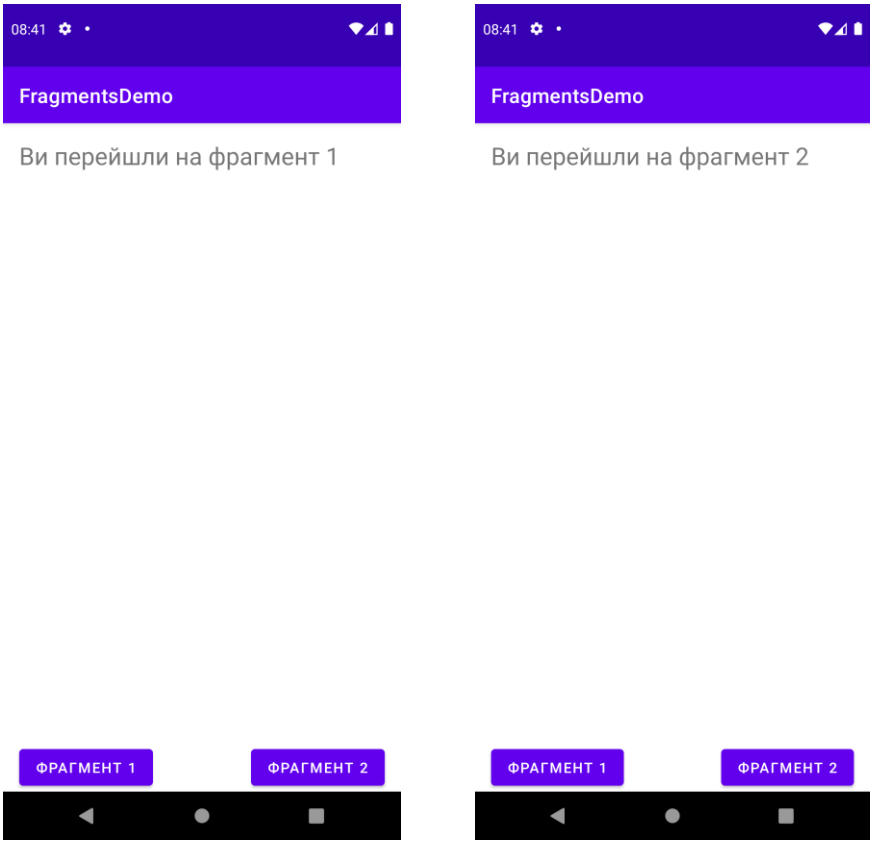

Рис. 6.5. Вигляд додатку з фрагментами

Кожний об'єкт типу FragmentManager пов'язаний з хостом, яким переважно виступає FragmentActivity. Протягом переходу активності між станами CREATED, STARTED та RESUMED менеджер фрагментів передає дані зміни своїм фрагментам за допомогою методу moveToState() та великої кількості умов для визначення того, в якому стані має бути фрагмент. Для спрощення moveToState() в новому менеджері стану дана логіка була винесена в окремий внутрішній клас FragmentStateManager. Тепер кожний екземпляр фрагменту під капотом пов'язаний з FragmentStateManager, що надає можливість винести з FragmentManager багато коду, який взаємодіє з фрагментом, зокрема виклик onCreateView() фрагмента та інші методи життєвого циклу. Таке розмежування коду також дало змогу інкапсулювати логіку підтримки зворотної сумісності в одному місці – методі computeExpectedState(). Він стежить за поточним станом фрагмента та визначає, в якому стані фрагмент має перебувати. Єдина ситуація, коли метод не може визначити потрібний стан – це відкладені (postponed) фрагменти.

Відкладений фрагмент має дві важливі характеристики:

- його представлення (view) вже створено, проте ще не відображається на екрані;
- його життєвий цикл зупинився у стані STARTED.

Така поведінка може бути корисною для відкладення запуску анімації переходу (enter transition), поки ресурси та налаштування додатку не будуть повністю готові, щоб його виконати. Як тільки буде викликатись метод

startPostponedEnterTransition(), запрацює анімація переходу фрагмента, відповідне представлення стане видимим, а стан фрагменту перейде в RESUMED. Так працює новий менеджер стану, проте раніше поведінка була забагованою [https://issuetracker.google.com/issues/147749580]. Стара версія FragmentManager, крім вказаних операцій, здійснювала перехід фрагменту в неузгоджений стан та генерувала купу багів:

- представлення фрагменту створювалось, проте фрагмент не додавався в бекстек (метод isAdded() повертав false);
- метод findFragmentById() не повертав щойно доданий фрагмент (хоч старий фрагмент уже було замінено) навіть з викликом commitNow();
- фрагменти застрягали в цьому невизначеному стані, не запускаючись при початку роботи FragmentManager (https://issuetracker.google.com/issues/129035555);
- об'єкти FragmentTransaction могли виконуватись невпорядковано (https://issuetracker.google.com/issues/147297731);
- інші анімації стосовно контейнера все ще працювали (https://issuetracker.google.com/issues/37140383);
- метод onCreateView() міг викликатись вдруге (https://issuetracker.google.com/issues/143915710).

Виправлення даних багів спричинило заміну всього процесу відкочування змін для відкладених фрагментів.

Додаткові проблеми вносять анімації при видаленні чи заміні фрагментів на рівні контейнера. Фрагменти підтримують кілька систем анімації:

- старий і забутий фреймворк Animation API;
- фреймворк Animator API;
- фреймворк Transition API (для API 21+);
- AndroidX Transition API.

Внутрішня реалізація передбачає спеціальний клас-контролер на рівні контейнера, який координуватиме роботу з усіма цими API та визначатиме, що конкретно має статись з цим контейнером. Таким чином, якщо фрагмент нагорі бекстеку відкладається, то відкладається виконання і всього контейнера. Більше логіки в FragmentManager чи операцій з відкочування не потрібно, оскільки вони можуть вплинути на весь контейнер. Отриманий клас-контролер SpecialEffectsController збирає розпорошену по FragmentManager логіку роботи з різними API для анімацій. Замість монолітного класу FragmentManager новий менеджер стану працює з окремими екземплярами FragmentStateManager, які координують роботу з іншими фрагментами контейнера за допомогою SpecialEffectsController (рис. 6.6). Декомпозиція FragmentManager дозволила розбити та спростити логіку:

- Клас FragmentManager має лише стан, який стосується всіх фрагментів;
- Клас FragmentStateManager управляє станом на рівні фрагмента;
- Клас SpecialEffectsController керує станом на рівні контейнера.

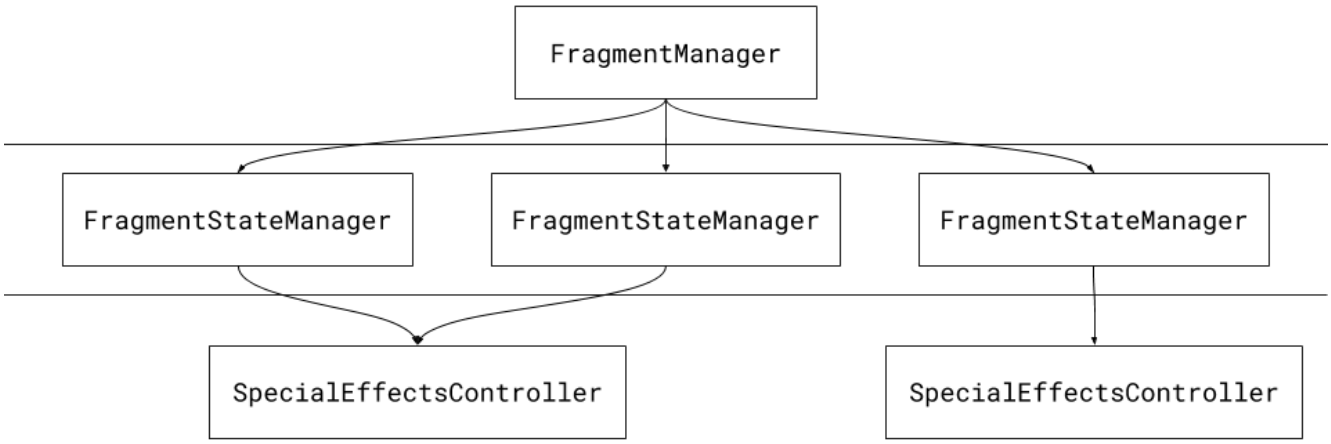

Рис. 6.6. Принцип роботи нового менеджеру стану фрагментів

Сучасний підхід до розробки навігації в Android-додатку – компонент Jetpack Navigation – може взагалі приховати роботу з FragmentManager, оскільки дозволяє задавати навігацію за допомогою візуального конструктора. Детальніше про навігацію в наступній темі.

https://medium.com/androiddevelopers/fragments-rebuilding-the-internals-61913f8bf48e

https://proandroiddev.com/android-fragments-fragmentcontainerview-292f393f9ccf

https://medium.com/@pavan.careers5208/fragmentcontainerview-c39d8ac376d1

Комунікація з активністю-хостом та іншими фрагментами

Взаємодія між фрагментами та хостом може реалізуватись кількома способами. Відповідно до документації, не рекомендується, щоб фрагменти взаємодіяли з активністю-хостом або між собою напряму [https://developer.android.com/guide/fragments/communicate]. Перший популярний підхід для організації комунікації фрагментів – використання інтерфейсу-комунікатора, який буде реалізовуватись хостом. Принцип побудови такого додатку зображено на рис. 6.7.

Демонстраційний проєкт FragmentCommunicationInterface передбачатиме наявність двох фрагментів: дані з одного фрагменту, введені в EditText, будуть надсилатись при натисненні на кнопку в інший фрагмент і виводитись на екран за допомогою TextView. Створення фрагментів відбуватиметься за схемою з попереднього питання теми. Лістинг 6.3 демонструє логіку роботи хоста та інтерфейс-комунікатор, за допомогою якого відбуватиметься передача даних у інший фрагмент у вигляді Bundle-об’єкта. Для налаштування аргументів фрагменту доводиться використовувати більш розгорнуту форму запису для транзакції (рядки 20-27).

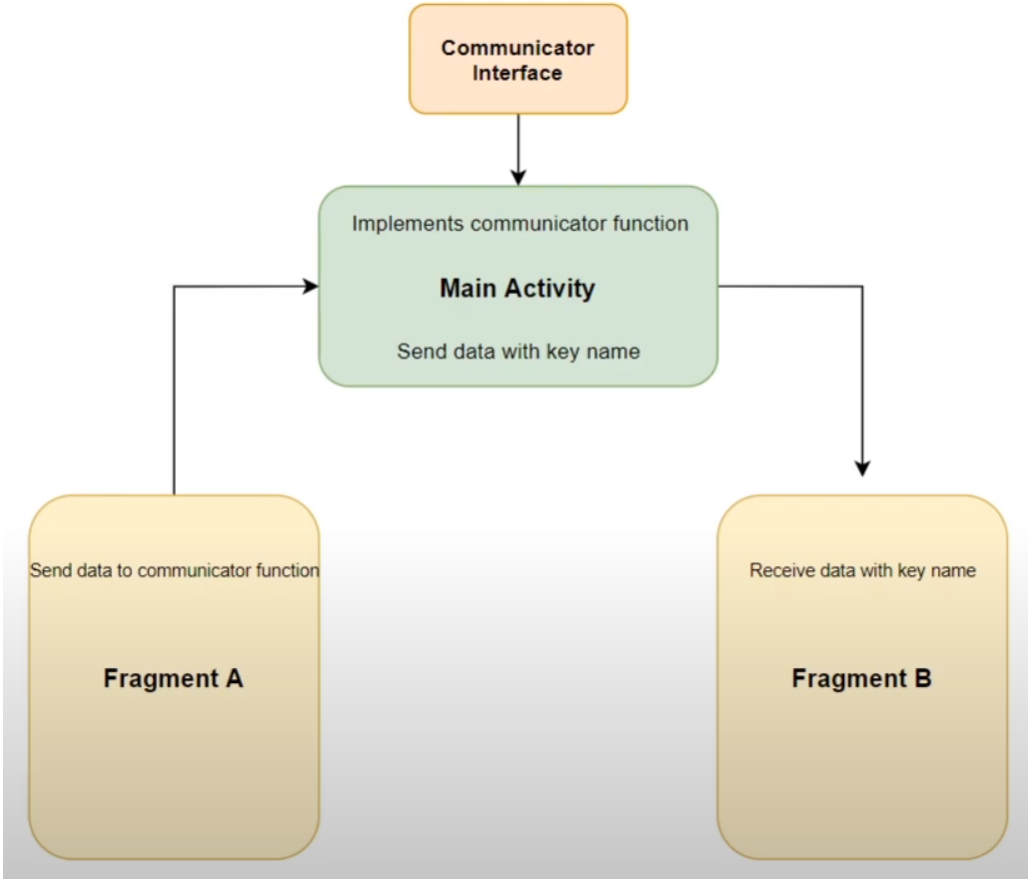

Рис. 6.7. Принцип комунікації фрагментів через інтерфейс-комунікатор

Лістинг 6.3. Код реалізації активності-хосту та інтерфейсу-комунікатора

MainActivity.kt

```
class MainActivity : AppCompatActivity(), FragmentsCommunicator
{
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        supportFragmentManager.commit {
            replace<FirstFragment>(R.id.fragment_container)
            setReorderingAllowed(true)
            addToBackStack("base")
        }

    }

    override fun passData(input: String) {
        val bundle = Bundle()
        bundle.putString("input", input)

        val transaction = this.supportFragmentManager
                                        .beginTransaction()
        val frag2 = SecondFragment()
```

```
23          frag2.arguments = bundle
24
25          transaction.replace(R.id.fragment_container, frag2)
26          transaction.addToBackStack("second")
27          transaction.commit()
28      }
29  }
30                                                  FragmentsCommunicator.kt
31  interface FragmentsCommunicator {
32      fun passData(input: String)
33  }
```

Логіка роботи першого фрагменту доповнюється формуванням об'єкта comm для взаємодії фрагмента з активністю. Тут для простоти в сигнатурі методу passData() було визначено параметр типу String, проте розробник може визначати довільний набір параметрів, типи яким підтримуються Bundle. Основна частина реалізації методу onCreateView() для першого фрагменту виглядає так:

```
val view = inflater.inflate(R.layout.fragment_first,
                                        container, false)
comm = activity as FragmentsCommunicator
val btnSend = view.findViewById<Button>(R.id.btnSend)
val inputText = view.findViewById<EditText>(R.id.editData)
btnSend.setOnClickListener {
    comm.passData(inputText.text.toString())
}
```

Задача другого фрагмента – розпакувати отримані дані та вивести відповідний текст у віджеті TextView (рис. 6.8). Зверніть увагу, що розпакування відбувається за ключем, який вказувався в методі passData(). Реалізація методу onCreateView() для другого фрагменту міститиме такий код:

```
displayMessage  = arguments?.getString("input")
val tv = view.findViewById<TextView>(R.id.textView)
tv.text = displayMessage
```

З точки зору архітектури додатку впровадження інтерфейсів-комунікаторів – не єдиний спосіб забезпечення взаємодії. Новий інструмент Fragment result API (з'явився в Fragment API 1.3.0-alpha04) дозволяє передавати дані між фрагментами за ключем. З цією метою використовується клас FragmentManager, який реалізує інтерфейс FragmentResultOwner. Даний інтерфейс виступає в ролі центрального сховища для даних, які передаються, та містить об'єкт ConcurrentHashMap<String, Bundle>. Таким чином, кожний бандл з даними ідентифікується рядковим ключем, а фрагмент після підписки має доступ до цих даних (рис. 6.9).

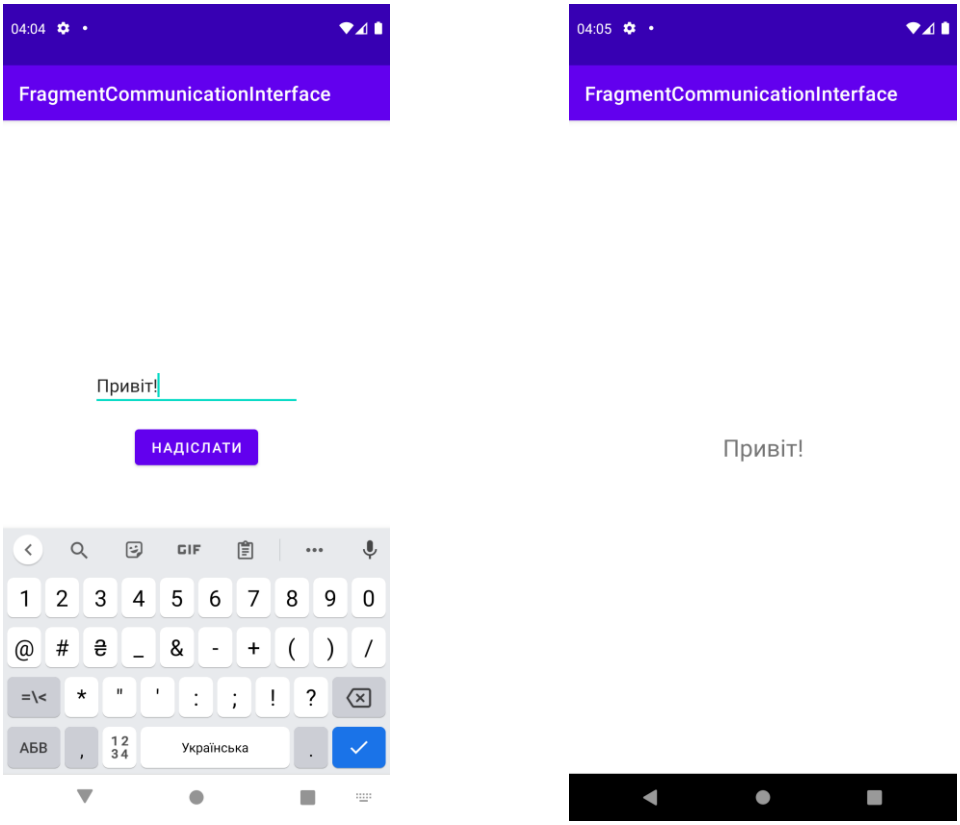

Рис. 6.8. Додаток FragmentCommunicationInterface

Особливості роботи з Fragment result API наступні:

- якщо фрагмент підписується на результат за допомогою методу setResultFragmentListener() після того, як фрагмент-відправник викличе setFragmentResult(), то він негайно отримає результат;
- кожну пару «ключ – Result (Bundle)» фрагмент отримує лише один раз;
- фрагменти з бекстеку отримають результат тільки після переходу в стан STARTED;
- після переходу фрагмента в стан DESTROYED стає неможливою підписка на ResultListener.

Після рефакторингу коду з попередніх лістингів відпадає потреба у зовнішньому інтерфейсі-комунікаторі та методі passData(), реалізація якого частково переміщується в onCreateView() першого фрагмента:

```
btnSend.setOnClickListener {
    val result = inputText.text.toString()
    setFragmentResult("requestKey", bundleOf("bundleKey" to
                                                result))
    activity?.supportFragmentManager?.commit {
        replace<SecondFragment>(R.id.fragment_container)
        setReorderingAllowed(true)
        addToBackStack("second")
    }
}
```

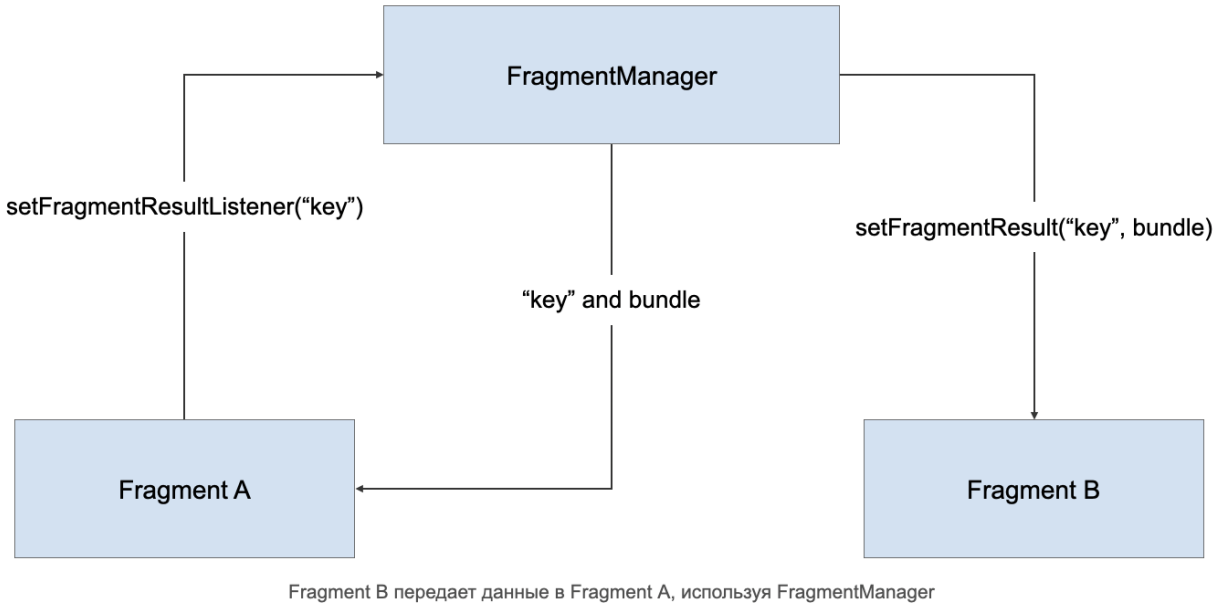

Рис. 6.9. Схема передачі даних між фрагментами засобами Fragment result API

Разом з тим, логіка другого фрагмента доповнюється реалізацією функції розширення setFragmentResultListener(), що дозволятиме розпакувати спостережувані бандли та застосувати отримані значення:

```
setFragmentResultListener("requestKey") { key, bundle ->
    tv.text = bundle.getString("bundleKey")
}
```

Результати роботи додатку не змінюються порівняно з рис. 6.8. Також такий підхід дозволяє передавати дані в дочірні фрагменти за допомогою об'єкта класу ChildFragmentManager.

Третій і найбільш поширений нині спосіб обміну даними між фрагментами (і роботи з фрагментами загалом) – збереження станів фрагментів у ViewModel-ах в комплексі з використанням компоненту Jetpack Navigation. Детальніше про нього в наступній темі.

https://betterprogramming.pub/4-ways-of-implementing-effective-communication-between-activities-fragments-and-objects-in-88f54ed7f4f6

https://johncodeos.com/how-to-pass-data-between-fragments-in-android-using-kotlin/

https://habr.com/ru/post/515080/