

2. АРХІТЕКТУРА МОБІЛЬНИХ ДОДАТКІВ

2.1. Компоненти інтерфейсу

Макет визначає візуальну структуру користувацького інтерфейсу, наприклад, призначеного для користувача інтерфейсу операції або віджета додатка. Існує два способи оголосити макет:

1. *Оголошення елементів призначеного для користувача інтерфейсу в XML.* В Android є зручний довідник XML-елементів для класів View і їх підкласів, наприклад таких, які використовуються для віджетів і макетів.

2. *Створення екземплярів елементів під час виконання.* Ваша програма може програмним чином створювати об'єкти View і ViewGroup (а також керувати їх властивостями).

Платформа Android забезпечує гнучкість при використанні будь-якого з цих способів для оголошення, призначеного для користувача інтерфейсу додатка і його управління. Наприклад, можна оголосити в XML макети за замовчуванням, включаючи елементи екрана, які будуть відображатися в макетах, і їх властивості. Потім можна додати в додаток код, який дозволяє змінювати стан об'єктів на екрані (включаючи оголошені в XML) під час виконання.

У модулі ADT для Eclipse передбачена функція попереднього перегляду, створеного файлу XML, для цього досить відкрити файл XML і вибрати вкладку Layout (Макет).

Для налагодження макетів користуються інструментом **Hierarchy Viewer** – за його допомогою можна переглянути значення властивостей, рамки з індикаторами заповнення або полів, а також повністю прорисовані подання прямо під час налагодження програми в емуляторі або на пристрої.

За допомогою інструменту `layoutopt` можна швидко проаналізувати макети і їх ієрархії на предмет низької ефективності чи інших проблем.

Перевага оголошення призначеного для користувача інтерфейсу у файлі XML полягає в тому, що таким чином можна більш ефективно відокремити подання свого додатка від коду, який управляє його поведінкою. Описи призначеного для користувача інтерфейсу знаходяться за межами коду вашої програми. Це означає, що можна змінювати або адаптувати інтерфейс без необхідності вносити правки у вихідний код і повторно компілювати його. Наприклад, можна створити різні файли XML-макета для екранів різних розмірів і різних орієнтацій екрана, а також для різних мов. Крім того, оголошення макета в XML спрощує візуалізацію структури призначеного для користувача інтерфейсу, завдяки чому налагодження проблем також стає простішим. У даному підрозділі ми навчимо вас оголошувати макет в XML. Якщо ви волієте за краще

створювати екземпляри об'єктів **View** під час виконання, зверніться до довідкової документації для класів **ViewGroup** і **View**.

Як правило, довідник XML-елементів для оголошення елементів призначеного для користувача інтерфейсу точно відповідає структурі і правилам іменування для класів і методів – назви елементів збігаються з назвами класів, а назви атрибутів відповідають методам. Фактично, відповідність часто така точна, що можна з легкістю здогадатися, який атрибут XML відповідає тому чи іншому методу класу, або який клас відповідає заданому елементу XML. Однак слід зазначити, що не всі довідники є ідентичними. У деяких випадках назви можуть дещо відрізнятися. Наприклад, у елемента **EditText** є атрибут **text**, який відповідає методу **EditText.setText()**.

2.1.1. Створення XML

За допомогою довідника XML-елементів, який є в Android, можна швидко і просто створювати макети призначеного для користувача інтерфейсу та елементи, що містяться в ньому, так само, як і при створенні веб-сторінок в HTML – за допомогою вкладених елементів.

У кожному файлі макета повинен бути всього один кореневий елемент, в якості якого повинен виступати об'єкт подання (**View**) або подання-групи (**ViewGroup**). Після визначення кореневого елемента можна приступати до додавання додаткових об'єктів макета або віджетів як дочірніх елементів для поступового формування ієрархії уявлень, яка визначає ваш макет. Нижче показаний приклад макета XML, в якому використовується вертикальний об'єкт **LinearLayout**, в якому розміщені елементи **TextView** і **Button**.

```
<?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="match_parent"
    android:orientation="vertical" >
  <TextView android:id="@+id/text"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello, I am a TextView" />
  <Button android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello, I am a Button" />
</LinearLayout>
```

Після оголошення макета в файлі XML збережіть файл з розширен-

ням .xml в каталог `res/layout/` свого проекту Android для подальшої компіляції.

2.1.2. Завантаження ресурсу XML

Під час компіляції додатка кожен файл XML макета компілюється в ресурс `View`. Вам необхідно завантажити ресурс макета в кодї програми в ходї реалізації методу зворотного виклику `Activity.onCreate()`. Для цього викличте метод `setContentView()`, передайте в нього посилання на ресурс макета в такїй формї: `R.layout.layout_file_name`. Наприклад, якщо макет XML збережений як файл `main_layout.xml`, то завантажити його для вашої операції необхідно таким чином:

```
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.main_layout);  
}
```

Метод зворотного виклику `onCreate()` в операції викликається платфо-рмою Android під час запуску операції.

2.1.3. Атрибути

Кожен об'єкт `View` і `ViewGroup` підтримує свої власні атрибути XML. Деякі атрибути характерні тільки для об'єкта `View` (наприклад, об'єкт `TextView` підтримує атрибут `textSize`), проте ці атрибути також успадковуються будь-якими об'єктами `View`, які можуть успадковувати цей клас. Деякі атрибути є загальними для всіх об'єктів `View`, оскільки вони успадковуються від кореневого класу `View` (такі, як атрибут `id`). Будь-які інші атрибути розглядаються як «параметри макета». Такі атрибути описують певні орієнтації макета для об'єкта `View`, які задані батьківським об'єктом `ViewGroup` такого об'єкта.

2.1.4. Ідентифікатор

Будь-який об'єкт `View` може бути пов'язаний з цілочисельним ідентифікатором, який служить для позначення унікальності об'єкта `View` в ієрархії. Під час компіляції додатка цей ідентифікатор використовується як ціле число, однак він зазвичай призначається у файлі XML-макета у вигляді рядка в атрибуті `id`. Цей атрибут XML є загальним для всіх об'єктів `View` (певним класом `View`), його ви будете використовувати досить часто. Синтаксис для ідентифікатора всередині тега XML такий:

```
android:id="@+id/my_button"
```

Символ @ на початку рядка вказує на те, що обробнику XML слід виконати синтаксичний аналіз решти ідентифікатора і визначити його як ресурс ідентифікатора. Символ плюса (+) означає, що це ім'я нового ресурсу, який необхідно створити і додати до наших ресурсів (у файлі `R.java`). В Android існує ряд інших ресурсів ідентифікатора. При посиланні на ідентифікатор ресурсу Android вам не потрібно вказувати символ плюса, проте необхідно додати простір імен пакета `android`, як зазначено нижче:

```
android:id="@android:id/empty"
```

Після додавання простору імен пакета `android` можна послатися на ідентифікатор з класу ресурсів `android.R`, а не з локального класу ресурсів.

Щоб створити подання і послатися на них з програми, зазвичай слід виконати такі дії.

1. Визначити подання або віджет у файлі макета і надати йому унікальний ідентифікатор:

```
<Button android:id="@+id/my_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/my_button_text"/>
```

2. Створити екземпляр об'єкта подання і виконати його захоплення з макета (зазвичай за допомогою методу `onCreate()`):

```
Button myButton = (Button) findViewById(R.id.my_button);
```

3. Визначити ідентифікатори для об'єктів подання; це має важливе значення при створенні об'єкта `RelativeLayout`. У відносному макеті універсальні ідентифікатори використовуються для розташування уявлень щодо один одного.

Ідентифікатор не обов'язково повинен бути унікальним в рамках всієї ієрархії, а тільки в тій її частині, де ви виконуєте пошук (найчастіше це може бути якраз вся ієрархія, тому при можливості ідентифікатори повинні бути повністю унікальними).

2.1.5. Параметри макета

Атрибути макета XML, які називаються `layout_something`, визначають параметри макета для об'єкта подання, що підходить для класу `ViewGroup`, в якому він знаходиться.

Кожен клас `ViewGroup` реалізує вкладений клас, який успадковує `ViewGroup.LayoutParams`. У цьому підкласі є типи властивостей, які визначають розмір і положення кожного дочірнього подання, які підходять для його групи. На рис. 2.1 показано, що батьківська група подань визначає параметри макета для кожного дочірнього подання (включаючи дочірню групу подань).

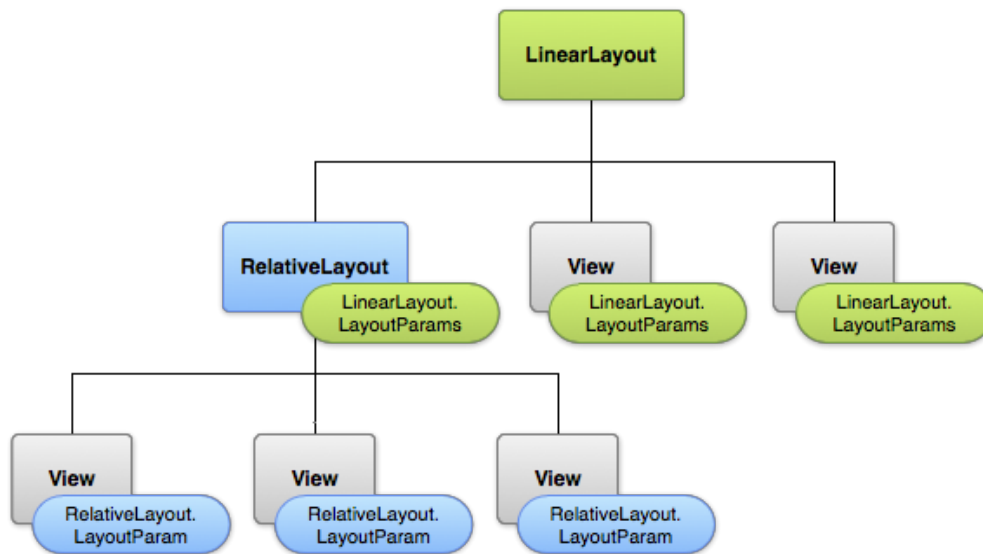


Рисунок 2.1 – Графічна інтерпретація ієрархії подання з параметрами макета кожного подання

Зверніть увагу, що підклас `LayoutParams` має власний синтаксис для задання значень. Кожен дочірній елемент повинен визначати `LayoutParams`, які підходять для його батьківського елемента, тоді як він сам може визначати інші `LayoutParams` для своїх дочірніх елементів.

Всі групи подань включають в себе параметри ширини і висоти (`layout_width` і `layout_height`), і кожне подання має визначати їх. Багато `LayoutParams` також включають додаткові параметри полів і кордонів.

Для параметрів ширини і висоти можна вказати точні значення, хоча, можливо, вам не захочеться робити це часто. Зазвичай для завдання значень ширини і висоти використовується одна з таких констант:

- `wrap_content` – розмір подання задається за розмірами його вмісту;
- `match_parent` (яка до API рівня 8 називалася `fill_parent`) – розмір подання визначається обмеженнями, що задаються його батьківською групою

подань.

Як правило, не рекомендується ставити абсолютні значення ширини і висоти макета (наприклад, у точках). Замість цього використовуйте відносні одиниці виміру, такі, як пікселі, що не залежать від дозволу екрану (dp), `wrap_content` або `match_parent`. Це гарантує однакове відображення вашого застосування на пристроях з екранами різних розмірів.

2.1.6. Розміщення макета

Подання має прямокутну форму. Розташування подання визначається його координатами зліва і зверху, а його розміри – параметрами ширини і висоти. Розташування вимірюється в пікселях.

Розташування подання можна отримати шляхом виклику методів `getLeft()` і `getTop()`. Перший повертає координату зліва (по осі X) для прямокутника подання. Другий повертає верхню координату (по осі Y) для прямокутника подання. Обидва ці методи повертають розташування точки зору щодо його батьківського елемента. Наприклад, коли метод `getLeft()` повертає 20, це означає, що подання знаходиться на відстані 20 пікселів від лівого краю його безпосереднього батьківського елемента.

Крім того, є кілька зручних методів (`getRight()` і `getBottom()`), які дозволяють уникнути зайвих обчислень. Ці методи повертають координати правого і нижнього країв прямокутника подання. Наприклад, виклик методу `getRight()` аналогічний такому обчисленню: `getLeft()+getWidth()`.

Розмір, відступ і поля. Розмір подання виражається його шириною і висотою. Фактично, подання має дві пари значень «ширина-висота».

Перша пара – це *виміряна ширина* і *виміряна висота*. Ці розміри визначають розмір подання в межах свого батьківського елемента. Виміряні розміри можна отримати, викликавши методи `getMeasuredWidth()` і `getMeasuredHeight()`.

Друга пара значень – це просто *ширина* і *висота* (іноді вони називаються *креслярська ширина* і *креслярська висота*). Ці розміри визначають фактичний розмір подання на екрані після розмічування під час їх відтворення. Вони можуть відрізнятися від виміряних ширини і висоти, хоча це і не обов'язково. Значення ширини і висоти можна отримати, викликавши методи `getWidth()` і `getHeight()`.

При вимірі своїх розмірів подання враховує заповнення. Відступ виражається в пікселях для лівої, верхньої, правої і нижньої частин подання. Відступ можна використовувати для зсуву вмісту подання на певну кількість пікселів. Наприклад, значення відступу зліва, що дорівнює 2, призведе до того, що вміст

подання буде зміщено на 2 пікселі вправо від лівого краю подання. Для задання відступів можна використовувати метод `setPadding(int, int, int, int)`. Щоб запросити відступ, використовують методи `getPaddingLeft()`, `getPaddingTop()`, `getPaddingRight()` і `getPaddingBottom()`.

Навіть якщо подання може визначити відступ, в ньому відсутня підтримка полів. Така можливість є у групи подань.

2.1.7. Макети інтерфейсу користувача

Лінійний макет. `LinearLayout` (лінійний макет) – це подання групи, яке вирівнює всі дочірні елементи в одному напрямку, вертикально або горизонтально (рис. 2.2). Можна вказати напрямок макета за допомогою атрибута `android:orientation`.

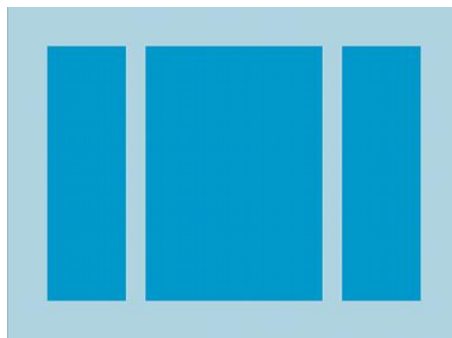


Рисунок 2.2 – Лінійний макет

Всі дочірні елементи `LinearLayout` розміщуються один за одним, таким чином, що вертикальний список має лише один дочірній елемент на кожен рядок, незалежно від його ширини, а горизонтальний список – одну висоту (висота найвищого дочірнього елемента плюс внутрішній відступ). `LinearLayout` розуміє зовнішні відступи між дочірніми елементами і тяжіння (праворуч, по центру або вирівнювання по лівому краю) кожного дочірнього елемента.

Вага макета. `LinearLayout` також підтримує призначення ваги окремим дочірнім елементам за допомогою атрибута `android:layout_weight`. Цей атрибут привласнює значення «важливості» подання з точки зору того, скільки місця воно повинне займати на екрані. Більше значення ваги дозволяє йому розширюватися, щоб заповнити все місце, що залишилось в батьківському поданні. Дочірні подання можуть точно вказувати значення ваги, і тоді все вільне місце в поданні групи призначається дочірнім елементам пропорційно до їх вказаної ваги. Значення ваги за замовчуванням дорівнює нулю.

Наприклад, якщо є три текстових поля і два з них мають вагу, що дорівнює 1, в той час як вага інших полів не вказана, то третє текстове поле без ваги

не збільшиться, а займе лише область, необхідну для його вмісту. Інші два розширяться в рівній мірі, щоб заповнити місце, що залишилося після зважування всіх трьох полів. Якщо потім третьому полю призначити вагу, що дорівнює 2 (замість 0), то його буде оголошено як більш важливе, ніж два інших, а тому воно отримує половину загального залишку місця, в той час як перші два поділяють решту порівну.

Рівноважні дочірні елементи. Для того, щоб створити лінійний макет, в якому кожен дочірній елемент займає однаковий обсяг простору на екрані (рис. 2.3), встановіть `android: layout_height` кожного подання таким, що дорівнює «0 dp» (для вертикального макета) або `android: layout_width` кожного подання таким, що дорівнює «0 dp» (для горизонтального макета). Потім встановіть `android: layout_weight` кожного подання таким, що дорівнює «1».

Приклад

```
<?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="match_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:orientation="vertical" >
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/to" />
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/subject" />
    <EditText
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:gravity="top"
        android:hint="@string/message" />
    <Button
        android:layout_width="100dp"
        android:layout_height="wrap_content"
        android:layout_gravity="right"
        android:text="@string/send" />
</LinearLayout>
```

Для більш детальної інформації про атрибути, що доступні кожному поданню `LinearLayout`, див. в `LinearLayout.LayoutParams`.

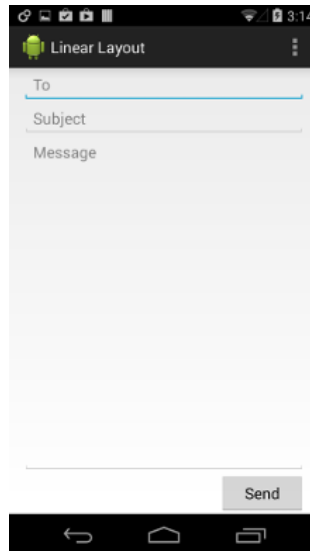


Рисунок 2.3 – Лінійний макет (приклад)

Відносний макет. RelativeLayout (відносний макет) – це подання групи, яке відображає дочірні подання у відносних позиціях. Положення кожного подання може бути визначене відносно споріднених елементів (наприклад, зліва від іншого подання, або нижче від нього) або положення щодо батьківського елемента області **RelativeLayout**, наприклад, вирівнювання по нижньому краю, зліва або по центру (рис. 2.4).



Рисунок 2.4 – Відносний макет

RelativeLayout – це дуже потужна утиліта для проектування користувацького інтерфейсу, оскільки вона може усунути вкладені подання груп і зберегти плоску ієрархію вашого макета, що підвищує продуктивність. Якщо ви використовуєте кілька вкладених груп **LinearLayout**, то їх можна замінити одним **RelativeLayout**.

Позиціонування подань. **RelativeLayout** дозволяє дочірнім поданням визначати їх положення щодо батьківського подання або відносно один одного

(задається за допомогою ID). Таким чином, можна вирівняти два елементи за правим краєм, або розташувати один елемент нижче від іншого, розташованого в центрі екрана, по центру зліва тощо. За замовчуванням, всі дочірні подання відмальовуються у верхньому лівому кутку макета, тому необхідно вказати положення кожного подання, використовуючи різні властивості макета, доступні з `RelativeLayout.LayoutParams`.

Деякі з множини властивостей макета, доступних поданням в `RelativeLayout`, містять у собі:

1) `android:layout_alignParentTop`, що розміщує верхню межу цього подання відповідно до верхньої межі батьківського елемента (якщо значення дорівнює «true»);

2) `android:layout_centerVertical`, що вирівнює цей дочірній елемент вертикально по центру усередині його батьківського елемента, якщо його значення дорівнює «true»;

3) `android:layout_below`, що позиціонує верхній край цього подання, розташованого нижче від подання певного ID ресурсу;

4) `android:layout_toRightOf`, що позиціонує лівий край цього подання відповідно до правого краю подання, розташованого нижче від певного ID ресурсу.

Значення кожної властивості макета являє собою або логічний тип, щоб включати позиціонування макета відносно батьківського `RelativeLayout`, або ID, що посилається на інше подання у макеті, відносно якого дане подання повинне бути розташоване.

У вашому макеті XML залежності відносно інших подань у макеті можуть бути оголошені у будь-якому порядку. Наприклад, можна оголосити, що «view1» буде розташовуватися нижче від «view2», навіть якщо подання «view2» оголошено останнім у ієрархії. Приклад, поданий нижче демонструє подібний сценарій; відповідний відносний макет показано на рис. 2.5.

Приклад. Кожен з атрибутів, які контролюють відносне положення кожного подання, підкреслені:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp" >
  <EditText
    android:id="@+id/name"
```

```

        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/reminder" />
<Spinner
    android:id="@+id/dates"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_below="@id/name"
    android:layout_alignParentLeft="true"
    android:layout_toLeftOf="@+id/times" />
<Spinner
    android:id="@id/times"
    android:layout_width="96dp"
    android:layout_height="wrap_content"
    android:layout_below="@id/name"
    android:layout_alignParentRight="true" />
<Button
    android:layout_width="96dp"
    android:layout_height="wrap_content"
    android:layout_below="@id/times"
    android:layout_alignParentRight="true"
    android:text="@string/done" />
</RelativeLayout>

```

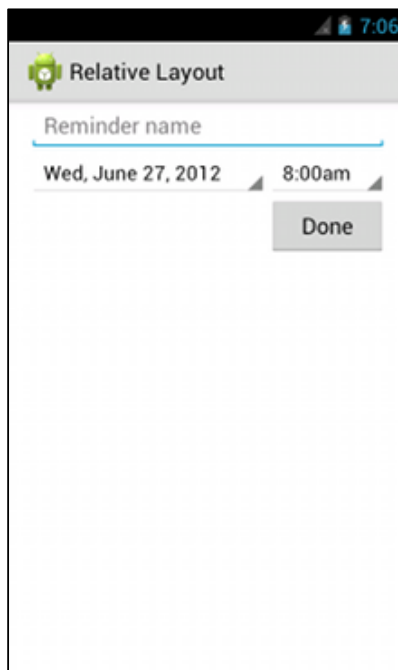


Рисунок 2.5 – Відносний макет (приклад)

Більш детальної інформації про атрибути, доступні кожному поданню `RelativeLayout`, див. в `RelativeLayout.LayoutParams`.

Подання у вигляді списку. **ListView** – це подання групи, яке відображає список прокручуваних елементів (рис. 2.6). Елементи списку автоматично додаються до списку за допомогою **Adapter**, який витягує вміст з джерела, такого, як масив або запит бази даних, і конвертує кожен елемент у подання, поміщене у список.



Рисунок 2.6 – Подання у вигляді списку

Використання завантажувача. Використання **CursorLoader** є стандартним способом запиту **Cursor** як асинхронного завдання для того, щоб уникнути блокування основного потоку із запитом вашого додатка. Коли **CursorLoader** отримує результат **Cursor**, **LoaderCallbacks** отримує зворотний виклик до **onLoadFinished()**, у якому оновлюється свій **Adapter** з новим **Cursor**, і потім подання відображає результати у вигляді списку.

Хоча API-інтерфейси **CursorLoader** були вперше застосовані у **Android 3.0 (API 11 рівня)**, вони також доступні у **SupportLibrary**, тому що ваш додаток може використовувати їх, поки пристрої, що їх підтримують, працюють на **Android-версії 1.6 і вище**.

Більш детальну інформацію про використання **Loader** для асинхронного завантаження даних, див. у довіднику з **Loaders**.

Приклад. Наступний приклад використовує **ListActivity**, що є активністю, яка містить у собі **ListView** як єдиний елемент макета за замовчуванням. Він виконує запит до **Contacts Provider** для отримання списку імен та телефонних номерів.

Активність реалізує інтерфейс **LoaderCallbacks** з метою використання **CursorLoader**, який динамічно завантажує дані для подання у вигляді списку.

```
public class ListViewLoader extends ListActivity
    implements LoaderManager.LoaderCallbacks<Cursor> {
```

```

// This is the Adapter being used to display the list's data
SimpleCursorAdapter mAdapter;

// These are the Contacts rows that we will retrieve
static final String[] PROJECTION = new String[]
{ContactsContract.Data._ID,
    ContactsContract.Data.DISPLAY_NAME};

// This is the select criteria
static final String SELECTION = "(" +
    ContactsContract.Data.DISPLAY_NAME + " NOTNULL) AND (" +
    ContactsContract.Data.DISPLAY_NAME + " != ' ' )";

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // Create a progress bar to display while the list loads
    ProgressBar progressBar = new ProgressBar(this);
    progressBar.setLayoutParams(new
LayoutParams(LayoutParams.WRAP_CONTENT,
    LayoutParams.WRAP_CONTENT, Gravity.CENTER));
    progressBar.setIndeterminate(true);
    getListView().setEmptyView(progressBar);

    // Must add the progress bar to the root of the layout
    ViewGroup root = (ViewGroup)
findViewById(android.R.id.content);
    root.addView(progressBar);

    // For the cursor adapter, specify which columns go into which views
    String[] fromColumns = {ContactsContract.Data.DISPLAY_NAME};
    int[] toViews = {android.R.id.text1}; // The TextView in sim-
ple_list_item_1

    // Create an empty adapter we will use to display the loaded data.
    // We pass null for the cursor, then update it in onLoadFinished()
    mAdapter = new SimpleCursorAdapter(this,
        android.R.layout.simple_list_item_1, null,
        fromColumns, toViews, 0);
    setListAdapter(mAdapter);

    // Prepare the loader. Either re-connect with an existing one,
    // or start a new one.
    getLoaderManager().initLoader(0, null, this);
}

// Called when a new Loader needs to be created
public Loader<Cursor> onCreateLoader(int id, Bundle args) {
    // Now create and return a CursorLoader that will take care of
    // creating a Cursor for the data being displayed.
    return new CursorLoader(this,
ContactsContract.Data.CONTENT_URI,
    PROJECTION, SELECTION, null, null);
}

```

```

    }

    // Called when a previously created loader has finished loading
    public void onLoadFinished(Loader<Cursor> loader, Cursor data) {
        // Swap the new cursor in. (The framework will take care of closing the
        // old cursor once we return.)
        mAdapter.swapCursor(data);
    }

    // Called when a previously created loader is reset, making the data unavail-
able
    public void onLoaderReset(Loader<Cursor> loader) {
        // This is called when the last Cursor provided to onLoadFinished()
        // above is about to be closed. We need to make sure we are no
        // longer using it.
        mAdapter.swapCursor(null);
    }

    @Override
    public void onItemClick(ListView l, View v, int position,
long id) {
        // Do something when a list item is clicked
    }
}

```

Примітка. Оскільки у цьому прикладі виконується запит на **Contacts Provider**, якщо запустити цей код, то додаток повинен запросити дозвіл **READ_CONTACTS** у файлі маніфесту: `<uses-permission android:name = "android.permission.READ_CONTACTS" />`.

Подання у вигляді сітки. **GridView** – це **ViewGroup**, яке відображає елементи у двовимірній сітці, що перегортується (рис. 2.7). Елементи сітки автоматично додаються до макета за допомогою **ListAdapter**.

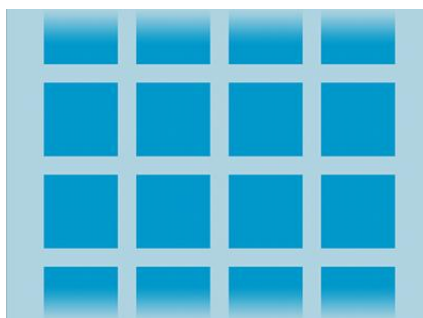


Рисунок 2.7 – Подання у вигляді сітки

Для знайомства з тим, як динамічно додавати подання, використовуючи адаптер, зверніться до **Building Layouts with an Adapter**.

Приклад. У цьому прикладі створюємо сітку ескізів зображень.

Коли елемент обрано і спливаюче повідомлення покаже положення елемента, виконайте таку послідовність дій:

1. Створіть новий проект з назвою **HelloGridView**.
2. Знайдіть декілька фото, які б ви хотіли використовувати, або завантажте ці зразки зображень. Збережіть файли зображень у папку проекту **res/drawable/**.
3. Відкрийте файл **res/layout/main.xml** і вставте таке:

```
<?xml version="1.0" encoding="utf-8"?>
<GridView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/gridview"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:columnWidth="90dp"
    android:numColumns="auto_fit"
    android:verticalSpacing="10dp"
    android:horizontalSpacing="10dp"
    android:stretchMode="columnWidth"
    android:gravity="center"
/>
```

Цей **GridView** заповнить весь екран. Атрибути не вимагають пояснень. Більш детальну інформацію про допустимі атрибути, див. у посиланні **GridView**.

4. Відкрийте **HelloGridView.java** і вставте наступний код для методу **onCreate()**:

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    GridView gridview = (GridView) findViewById(R.id.gridview);
    gridview.setAdapter(new ImageAdapter(this));

    gridview.setOnItemClickListener(new OnItemClickListener() {
        public void onItemClick(AdapterView<?> parent, View v,
            int position, long id) {
            Toast.makeText>HelloGridView.this, "" + position,
                Toast.LENGTH_SHORT).show();
        }
    });
}
```

Після того як макет **main.xml** буде встановлений для подання вмісту, **GridView** береться з макета за допомогою **findViewById(int)**. Потім ме-

тоді `setAdapter()` встановлює користувацький адаптер (`ImageAdapter`) як джерело для всіх елементів, що відображаються у сітці. `ImageAdapter` створюється на наступному кроці.

Щоб зробити щось, коли на елемент у сітці натиснули, методу `setOnItemClickListener()` передається новий `AdapterView.OnItemClickListener`. Цей анонімний екземпляр визначає зворотний виклик методу `onItemClick()`, щоб показати `Toast`, який відображає початкову позицію обраного елемента (у реальному сценарії положення може використовуватися для отримання повнорозмірних зображень для якоїсь іншої задачі).

5. Створіть новий клас під назвою `ImageAdapter`, який успадковує `BaseAdapter`:

```
public class ImageAdapter extends BaseAdapter {
    private Context mContext;

    public ImageAdapter(Context c) {
        mContext = c;
    }

    public int getCount() {
        return mThumbIds.length;
    }

    public Object getItem(int position) {
        return null;
    }

    public long getItemId(int position) {
        return 0;
    }

    // Створюємо новий об'єкт ImageView для кожного елемента, на який посилається
    // адаптер
    public View getView(int position, View convertView, ViewGroup
parent) {
        ImageView imageView;
        if (convertView == null) {
            // if it's not recycled, initialize some attributes
            imageView = new ImageView(mContext);
            imageView.setLayoutParams(new GridView.LayoutParams(85,
85));
            imageView.setScaleType(ImageView.ScaleType.CENTER_CROP);
            imageView.setPadding(8, 8, 8, 8);
        } else {
            imageView = (ImageView) convertView;
        }

        imageView.setImageResource(mThumbIds[position]);
    }
}
```



```

        return imageView;
    }

    // references to our images
    private Integer[] mThumbIds = {
        R.drawable.sample_2, R.drawable.sample_3,
        R.drawable.sample_4, R.drawable.sample_5,
        R.drawable.sample_6, R.drawable.sample_7,
        R.drawable.sample_0, R.drawable.sample_1,
        R.drawable.sample_2, R.drawable.sample_3,
        R.drawable.sample_4, R.drawable.sample_5,
        R.drawable.sample_6, R.drawable.sample_7,
        R.drawable.sample_0, R.drawable.sample_1,
        R.drawable.sample_2, R.drawable.sample_3,
        R.drawable.sample_4, R.drawable.sample_5,
        R.drawable.sample_6, R.drawable.sample_7
    };
}

```

По-перше, він реалізує деякі необхідні методи, успадковані від **BaseAdapter**. Конструктор і **getCount()** не вимагають пояснень. Зазвичай **getItem(int)** повинен повертати реальний об'єкт у зазначеному місці у адаптері, але це ігнорується для цього прикладу. Аналогічно **getItemId(int)** повинен повертати ідентифікатор рядка елемента, але тут це не потрібно.

Перший необхідний метод – це **getView()**. Він створює нове **View** для кожного зображення, доданого у **ImageAdapter**. Коли метод викликається, йому передається **View**, яке зазвичай є об'єктом, що використовується повторно (принаймні хоч один раз), і таким чином виконується перевірка, чи є об'єкт нульовим. Якщо він дорівнює нулю, створюється екземпляр **ImageView**, який конфігурується бажаними властивостями для презентації зображення:

- **setLayoutParams(ViewGroup.LayoutParams)** встановлює висоту і ширину **View** – це гарантує, що незалежно від розміру області малювання кожне зображення масштабується і обрізається до відповідних розмірів, у разі необхідності;

- **setScaleType(ImageView.ScaleType)** оголошує, що зображення повинні бути обрізані у напрямку до центру (за необхідності);

- **setPadding(int, int, int, int)** визначає внутрішній відступ з усіх боків (зверніть увагу, що якщо зображення мають різні співвідношення сторін, тоді менший відступ зумовить більше обрізання зображення, якщо воно не відповідає розмірам, зазначеним в **ImageView**).

Якщо передане **View** у **getView()** має ненульове значення, то локальне **ImageView** ініціалізується за допомогою повторно використовуваного об'єкта **View**.

У кінці методу `getView()` ціле число `position`, яке передається у цей метод, використовується для вибору зображення з масиву `mThumbIds`, який встановлює ресурс зображення для `ImageView`.

Залишилось визначити масив `mThumbIds`, що складається з ресурсів області малювання.

6. Запустіть додаток.

Пробуйте експериментувати з поведінками елементів `GridView` і `ImageView`, коригуючи їх властивості. Наприклад, замість використання `setLayoutParams(ViewGroup.LayoutParams)`, спробуйте використати `setAdjustViewBounds(boolean)`.

2.2. Життєвий цикл візуальних компонентів

2.2.1. Операції (Activity)

Activity – це компонент програми, який видає екран і з яким користувачі можуть взаємодіяти, щоб виконати будь-які дії, наприклад набрати номер телефону, зробити фото, відправити лист або переглянути карту. Кожній операції присвоюється вікно для промальовування відповідного, призначеного для користувача інтерфейсу. Зазвичай вікно відображається на весь екран, проте його розмір може бути меншим, і воно може розміщуватися поверх інших вікон.

Як правило, програма містить кілька операцій, які слабо пов'язані одна з одною. Зазвичай одна з операцій в додатку позначається як «основна», пропонується користувачеві при першому запуску програми. У свою чергу, кожна операція може запустити іншу операцію для виконання різних дій. Кожен раз, коли запускається нова операція, попередня операція зупиняється, однак система зберігає її в стек («стек переходів назад»). При запуску нової операції вона поміщається в стек переходів назад і відображається для користувача. Стек переходів назад працює за принципом «останнім увійшов – першим вийшов», тому після того як користувач завершив поточну операцію і натиснув кнопку Назад, поточна операція видаляється зі стека (і знищується), і відновлюється попередня операція.

Коли операція зупиняється через запуск нової операції, для повідомлення про зміну її стану використовуються методи зворотного виклику життєвого циклу операції. Існує кілька таких методів, які може приймати операція внаслідок зміни свого стану: створення операції, її зупинка, відновлення або знищення системою; також кожен зворотний виклик дає можливість виконати певну дію, відповідну для певної зміни стану. Наприклад, в разі зупинки операція повинна звільнити будь-які великі об'єкти, наприклад, підключення до мережі або бази

даних. При відновленні операції можна повторно отримати необхідні ресурси і відновити виконання перерваних дій. Такі зміни стану є частиною життєвого циклу операції.

Далі розглянемо основи створення і використання операцій, включаючи повний опис життєвого циклу операції, щоб краще зрозуміти, як слід керувати переходами між різними станами операції.

Створення операції. Щоб створити операцію, спочатку необхідно створити підклас класу `Activity` (або скористатися існуючим його підкласом). У такому підкласі необхідно реалізувати методи зворотного виклику, які викликає система при переході операції з одного стану свого життєвого циклу в інший, наприклад, при створенні, зупинці, відновленні або знищенні операції. Ось два найбільш важливих методи зворотного виклику:

- *onCreate()*. Цей метод необхідно обов'язково реалізувати, оскільки система викликає його при створенні вашої операції. У своїй реалізації вам необхідно ініціалізувати ключові компоненти операції. Найбільш важливо саме тут викликати `setContentView()` для визначення макета призначеного для користувача інтерфейсу операції.

- *onPause()*. Система викликає цей метод як першу ознаку виходу користувача з операції (однак це не завжди означає, що операція буде знищена). Зазвичай саме тут необхідно застосовувати будь-які зміни, які повинні бути збережені, крім поточного сеансу роботи користувача (оскільки користувач може не повернутися назад).

Існують також і деякі інші методи зворотного виклику життєвого циклу, які необхідно використовувати для того, щоб забезпечити плавний перехід між операціями, а також для обробки непередбачених порушень у роботі операції, в результаті яких вона може бути зупинена або навіть знищена.

Реалізація інтерфейсу користувача. Для реалізації призначеної для інтерфейсу користувача операції використовується ієрархія подань-об'єктів, отриманих з класу `View`. Кожне подання відповідає за певну прямокутну область вікна операції і може реагувати на дії користувачів. Наприклад, поданням може бути кнопка, натискання на яку приводить до виконання певної дії.

В Android передбачений набір вже готових подань, які можна використовувати для створення дизайну макета і його організації.

Віджети – це подання з візуальними (і інтерактивними) елементами, наприклад кнопками, текстовими полями, чекбоксами або просто зображеннями.

Макети – це подання, отримані з класу `ViewGroup`, що забезпечують унікальну модель компоновання для своїх дочірніх подань, таких, як лінійний макет, сітка або відносний макет. Також можна створити підклас для класів `View` та `ViewGroup` (або скористатися існуючими підкласами), щоб створити власні віджети і макети, а потім застосувати їх до макета своєї операції.

Найчастіше для задання макета за допомогою подань використовується XML-файл макета, збережений в ресурсах додатка. Таким чином можна зберігати дизайн інтерфейсу користувача окремо від вихідного коду, який служить для задання поведінки операції. Щоб задати макет, призначений для користувачького інтерфейсу, можна використовувати метод `setContentView()`, передавши в нього ідентифікатор ресурсу для макета. Однак можна створити нові `View` в коді вашої операції і створити ієрархію подань. Для цього вставте `View` в `ViewGroup`, а потім використовуйте цей макет, передавши кореневий об'єкт `ViewGroup` в метод `setContentView()`.

Оголошення операції в маніфесті. Щоб операція стала доступна системі, її необхідно оголосити у файлі маніфесту. Для цього відкрийте файл маніфесту і додайте елемент `<activity>` у дочірній елемент `<application>`. Наприклад:

```
<manifest ... >
  <application ... >
    <activity android:name=".ExampleActivity" />
    ...
  </application ... >
  ...
</manifest >
```

Існує кілька інших атрибутів, які можна включити в цей елемент, щоб визначити такі властивості, як мітка операції, значок операції або тема оформлення для призначеного для користувача інтерфейсу операції. Єдиним обов'язковим атрибутом є `android:name` – він визначає ім'я класу операції. Після публікації вашого застосування вам не слід перейменовувати його, оскільки це може порушити деякі функціональні можливості програми, наприклад ярлики додатка (ознайомтеся з публікацією «Речі, які не можна змінювати» в блозі розробників).

Додаткові відомості про оголошення операції в маніфесті див. довідці про елемент `<activity>`.

Використання фільтрів намірів. Елемент `<activity>` також може задавати різні фільтри намірів за допомогою елемента `<intent-filter>` для оголошення того, як інші компоненти програми можуть активувати операцію.

При створенні нової програми за допомогою інструментів Android SDK в заготовці операції, створюваної автоматично, є фільтр намірів, який оголошує операцію. Ця операція реагує на виконання «основної» дії, і її слід помістити в категорію «перехід засобу запуску». Фільтр намірів виглядає таким чином:

```
activity android:name=".ExampleActivity" an-
droid:icon="@drawable/app_icon">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Елемент `<action>` вказує, що це «основна» точка входу в додаток, а елемент `<category>` – що цю операцію слід вказати у якості додатків системи (щоб користувачі могли запускати цю операцію).

Якщо додаток планується створити самодостатнім і заборонити іншим додаткам активувати його операції, то інших фільтрів намірів створювати не потрібно. У цьому випадку тільки в одній операції має бути «основна» дія, і її слід помістити в категорію засобу запуску, як в прикладі, наведеному вище. В операції, які не повинні бути доступні для інших додатків, не слід включати фільтри намірів. Такі операції можна самостійно запустити за допомогою явних намірів.

Однак, якщо треба, щоб операція реагувала на неявні наміри, одержувані від інших додатків (а також з вашого додатка), для операції слід визначити додаткові фільтри намірів. Для кожного типу наміру, на який необхідно реагувати, потрібно вказати об'єкт `<intent-filter>`, що включає елемент `<action>` і необов'язковий елемент `<category>` чи `<data>` (або обидва ці елементи). Ці елементи визначають тип наміру, на який може реагувати ваша операція.

Запуск операції. Для запуску іншої операції досить викликати метод `startActivity()`, передавши в нього об'єкт `Intent`, який описує операцію, що запускається. У намірі або вказується точна операція для запуску, або описується тип операції, яку ви хочете виконати (після цього система вибирає для

вас підходящу операцію, яка може навіть перебувати в іншому додатку). Намір також може містити невеликий обсяг даних, які будуть використовуватися запусненою операцією.

При роботі з власним додатком найчастіше треба лише запустити потрібну операцію. Для цього необхідно створити намір, який явно визначає необхідну операцію з допомогою імені класу. Нижче наведено приклад запуску однією операцією іншої операції з ім'ям `SignInActivity`:

```
Intent intent = new Intent(this, SignInActivity.class);
startActivity(intent);
```

Однак у вашому додатку також може стати потрібним виконати деяку дію, наприклад, відправити лист, текстове повідомлення або оновити статус, використовуючи дані з вашої операції. В цьому випадку у вашому додатку можуть бути відсутні такі дії, тому ви можете скористатися операціями з інших додатків, наявних на пристрої, які можуть виконувати необхідні дії. Саме в цьому разі наміри особливо корисні – можна створити намір, який описує необхідну дію, після чого система запускає його з іншої програми. За наявності декількох операцій, які можуть обробити намір, користувач може вибирати, який з них слід використовувати. Наприклад, якщо користувачеві потрібно надати можливість відправити електронний лист, можна створити такий намір:

```
Intent intent = new Intent(Intent.ACTION_SEND);
intent.putExtra(Intent.EXTRA_EMAIL, recipientArray);
startActivity(intent);
```

Додатковий компонент `EXTRA_EMAIL`, доданий в намір, являє собою рядковий масив адрес електронної пошти для відправки листа. Коли поштова програма реагує на цей намір, вона зчитує додатково доданий рядковий масив і поміщає наявні в ньому адреси в поле одержувача у вікні створення листа. При цьому запускається операція поштової програми, а після того як користувач завершить необхідні дії, відновлюється ваша операція.

Запуск операції для отримання результату. У деяких випадках після запуску операції треба буде отримати результат. Для цього викличте метод `startActivityForResult()` (замість `startActivity()`). Щоб отримати результат після виконання наступної операції, реалізуйте метод зворотного виклику `onActivityResult()`. По завершенні наступної операції вона повертає результат в об'єкті `Intent` у викликаний метод `onActivityResult()`.

Наприклад, користувачеві буде потрібно вибрати один з контактів, щоб ваша операція могла виконати деякі дії з інформацією про цей контакт. Нижче наведено приклад створення такого наміру і обробки результату:

```
private void pickContact() {
    // Створити намір для «вибору» контакту, як визначено постачаль-
    ником контенту URI.
    Intent intent = new Intent(Intent.ACTION_PICK, Con-
    tacts.CONTENT_URI);
    startActivityForResult(intent, PICK_CONTACT_REQUEST);
}

@Override
protected void onActivityResult(int requestCode, int resultCode,
    Intent data) {
    // Якщо запит пройшов успішно (OK) і запит був
    PICK_CONTACT_REQUEST
    if (resultCode == Activity.RESULT_OK && requestCode ==
    PICK_CONTACT_REQUEST) {
        // Виконати запит до постачальника змісту контакту для імені
        контакту
        Cursor cursor = getContentResolver().query(data.getData(),
            new String[] {Contacts.DISPLAY_NAME}, null, null, null);
        if (cursor.moveToFirst()) { // True if the cursor is not
        empty
            int columnIndex = cur-
            sor.getColumnIndex(Contacts.DISPLAY_NAME);
            String name = cursor.getString(columnIndex);
            // Зробити що-небудь з ім'ям обраного контакту ...
        }
    }
}
```

У цьому прикладі демонструється базова логіка, якою слід керуватися при використанні методу `onActivityResult()` для обробки результату виконання операції. Перша умова перевіряє, чи розпізнано запит, і якщо він успішний, то результат для `resultCode` буде `RESULT_OK`; також перевіряється, чи відомий запит, для якого отримано цей результат, і в цьому випадку `requestCode` відповідає другому параметру, відправленому в метод `startActivityForResult()`. Тут код обробляє результат виконання операції шляхом запиту даних, повернутих в `Intent` (параметр `data`).

При цьому `ContentResolver` виконує запит до постачальника контенту, який повертає об'єкт `Cursor`, що забезпечує зчитування запитаних даних.

Завершення операції. Для завершення операції досить викликати її `finish()`. Також для завершення окремої операції, запущеної раніше, можна викликати метод `finishActivity()`.

Примітка. У більшості випадків вам не слід явно завершувати операцію за допомогою цих методів. Система Android виконує таке управління за вас, а тому вам не потрібно завершувати ваші власні операції. Виклик цих методів може негативно позначитися на очікуваній поведінці додатка. Їх слід використовувати виключно тоді, коли ви абсолютно не хочете, щоб користувач повертався до цього екземпляра операції.

Управління життєвим циклом операції. Управління життєвим циклом операцій шляхом реалізації методів зворотного виклику має важливе значення при розробці надійних і гнучких програм. На життєвий цикл операції безпосередньо впливають його зв'язки з іншими операціями, його завданнями та стеком переходів назад.

Існує всього три стани операції:

1. Відновлена – операція виконується на передньому плані екрана і відображається для користувача (цей стан операції також іноді називається «Виконувана»).

2. Припинена – на передньому фоні виконується інша операція, яка відображається для користувача, однак ця операція, як і раніше, не прихована. Тобто поверх поточної операції показується інша операція, частково прозора або така, що не займає повністю весь екран. Призупинена операція повністю активна (об'єкт `Activity`, як і раніше, знаходиться в пам'яті, в ньому зберігаються всі відомості про стан і інформація про елементи, і він також залишається пов'язаним з диспетчером вікон), проте в разі гострого браку пам'яті система може завершити її.

3. Зупинена – операція повністю перекривається іншою операцією (тепер вона виконується у фоновому режимі). Зупинена операція, як і раніше, активна (об'єкт `Activity`, як і раніше, знаходиться в пам'яті, в ньому зберігаються всі відомості про стан і інформація про елементи, але об'єкт більше не пов'язаний з диспетчером вікон). Однак операцію більше не видно користувачеві, і в разі нестачі пам'яті система може завершити її.

Якщо операція припинена або повністю зупинена, система може очистити її з пам'яті, завершивши саму операцію (за допомогою методу `finish()`), або просто завершити її процес. У разі повторного відкриття операції (після її завершення) її потрібно створити повністю.

Реалізація зворотних викликів життєвого циклу. При переході операції з одного, описаного вище стану в інший, повідомлення про це реалізуються через різні методи зворотного виклику. Всі методи зворотного виклику є прив'язками, які можна перевизначити для виконання відповідної дії при зміні стану операції. Зазначена нижче базова операція включає кожен з основних методів життєвого циклу:

```
public class ExampleActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // Дія створюється.
    }
    @Override
    protected void onStart() {
        super.onStart();
        // Дія стане видимою.
    }
    @Override
    protected void onResume() {
        super.onResume();
        // Дія стала видимою (тепер вона «відновлена»).
    }
    @Override
    protected void onPause() {
        super.onPause();
        // Фокус переходить на іншу дію (ця дія скоро буде призупи-
нена).
    }
    @Override
    protected void onStop() {
        super.onStop();
        // Дія більше не відображається (вона тепер «зупинена»).
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        // Дія скоро буде видалена.
    }
}
```

Примітка. При реалізації методів життєвого циклу завжди викликайте реалізацію суперкласу, перш ніж виконувати будь-які дії, як показано в прикладах вище.

Разом всі ці методи визначають весь життєвий цикл операції. За допомогою реалізації цих методів можна відстежувати три вкладених цикли в життєвому циклі операції:

1. *Весь життєвий цикл операції* – відбувається між викликом методу `onCreate()` і викликом методу `onDestroy()`. Ваша операція повинна виконати настройку «глобального» стану (наприклад, визначення макета) в методі `onCreate()`, а потім звільнити всі ресурси, що залишилися в `onDestroy()`. Наприклад, якщо у вашій операції є потік, що виконується у фоновому режимі, для завантаження даних по мережі, операція може створити такий потік в методі `onCreate()`, а потім зупинити його в методі `onDestroy()`.

2. *Видимий життєвий цикл операції* – відбувається між викликами методів `onStart()` та `onStop()`. Протягом цього часу операція відображається на екрані, де користувач може взаємодіяти з нею. Наприклад, метод `onStop()` викликається в разі, коли запускається нова операція, а поточна більше не відображається. У проміжку між викликами цих двох методів можна зберегти ресурси, необхідні для відображення операції для користувача. Наприклад, можна зареєструвати об'єкт `BroadcastReceiver` в методі `onStart()` для відстеження змін, що впливають на призначений для користувача інтерфейс, а потім скасувати його реєстрацію в методі `onStop()`, коли користувач більше не бачить відображуваного. Протягом усього життєвого циклу операції система може кілька разів викликати методи `onStart()` та `onStop()`, оскільки операція то відображається для користувача, то ховається від нього.

3. *Життєвий цикл операції, що виконується на передньому плані* – відбувається між викликами методів `onResume()` та `onPause()`. Протягом цього часу операція виконується на тлі всіх інших операцій і відображається для користувача. Операція може часто входити у фоновий режим і виходити з нього. Наприклад, метод `onPause()` викликається при переході пристрою в сплячий режим або при появі діалогового вікна. Оскільки перехід у такий стан може виконуватися досить часто, код в цих двох методах повинен бути легким, щоб не допустити повільних переходів і не змушувати користувача чекати.

На рис. 2.8 ілюструються проходи і шляхи, які операція може пройти між станами. Прямокутниками позначені методи зворотного виклику, які можна реалізувати для виконання дій між переходами операції з одного стану в інший.

Ці ж методи життєвого циклу перераховані в табл. 2.1, в якій детально описано кожен метод зворотного виклику і вказано його місце в життєвому циклі операції в цілому, включаючи відомості про те, чи може система завершити операцію по завершенні методу зворотного виклику.

Таблиця 2.1 – Зведені відомості про методи зворотного виклику життєвого циклу операції

Метод	Опис	Завершальний?	Наступний
onCreate()	Викликається при першому створенні операції. Тут необхідно налаштувати всі звичайні статичні елементи – створити подання, прив’язати дані і т. д. Цей метод передає об’єкт <code>Bundle</code> , що містить попередній стан операції (якщо такий стан було зафіксовано раніше; див. Збереження стану операції). За ним завжди йде метод <code>onStart()</code>	Ні	<code>onStart()</code>
onRestart()	Викликається після зупинки операції безпосередньо перед її повторним запуском. За ним завжди йде метод <code>onStart()</code>	Ні	<code>onStart()</code>
onStart()	Викликається безпосередньо перед тим, як операція стає видимою для користувача. За ним йде метод <code>onResume()</code> , якщо операція переходить на передній план, або метод <code>onStop()</code> , якщо вона стає прихованою	Ні	<code>onResume()</code> або <code>onStop()</code>
onResume()	Викликається безпосередньо перед тим, як операція починає взаємодію з користувачем. На цьому етапі операція знаходиться в самому верху стека операцій, і в неї надходять дані, що вводяться користувачем. За ним завжди йде метод <code>onPause()</code>	Ні	<code>onPause()</code>
onPause()	Викликається, коли система збирається відновити іншу операцію. Цей метод зазвичай використовується для запису незбережених змін в постійне місце зберігання даних, зупинки анімацій та інших елементів, які можуть використовувати ресурси ЦП та інше. Тут вкрай важлива оперативність, оскільки наступна операція не буде відновлена доти, поки вона не буде повернена на передній план. За ним йде або метод <code>onResume()</code> , якщо операція повертається на передній план, або метод <code>onStop()</code> , якщо операція стає прихованою для користувача	Так	<code>onResume()</code> або <code>onStop()</code>
onStop()	Викликається в разі, коли операція більше не відображається для користувача. Це може статися через те, що операція знищена, або зважаючи на відновлення поверх неї іншої операції (існуючої або нової). За ним йде або метод <code>onRestart()</code> , якщо операція відновлює взаємодію з користувачем, або метод <code>onDestroy()</code> , якщо операція переходить у фоновий режим	Так	<code>onRestart()</code> або <code>onDestroy()</code>

Продовження табл. 2.1

Метод	Опис	Завершальний?	Наступний
<code>onDestroy()</code>	Викликається перед тим, як операція буде знищена. Це фінальний виклик, який отримує операція. Його можна викликати або через завершення операції (виклик методу <code>finish()</code>), або через тимчасове знищення системою цього примірника операції з метою звільнити місце. Щоб розрізнити ці два сценарії, використовується метод <code>isFinishing()</code>	Так	<i>Нічого</i>

У стовпці «Завершальний?» вказується, чи може система в будь-який час завершити процес, який містить операцію, після повернення методу без виконання будь-якого іншого рядка коду операції. Для трьох методів в цьому стовпці вказано «Так»: (`onPause()`, `onStop()` та `onDestroy()`). Оскільки метод `onPause()` є першим з цих трьох після створення операції, метод `onPause()` є останнім, який гарантовано буде викликаний перед тим, як процес можна буде завершити; якщо системі потрібно терміново відновити пам'ять у випадку аварійної ситуації, то методи `onStop()` та `onDestroy()` викликати не вдасться.

Тому слід скористатися `onPause()`, щоб записати критично важливі дані (такі, як редагування користувача) в сховище постійних даних. Однак слід уважно підходити до вибору інформації, яку необхідно зберегти під час виконання методу `onPause()`, оскільки будь-яке блокування процедур в цьому методі може викликати блокування переходу до наступної операції і гальмувати роботу користувача.

Методи, для яких у стовпці «Завершальний?», вказано «Ні», захищають процес, що містить операцію, від завершення відразу з моменту їх виклику. Тому завершити операцію можна в період між поверненням `onPause()` і викликом `onResume()`. Його знову не вдасться завершити, поки знову не буде викликаний і повернений `onPause()`.

Примітка. Операцію, яку технічно неможливо завершити відповідно до визначення в табл. 2.1, як і раніше, може завершити система, однак це може статися тільки в надзвичайних ситуаціях, коли немає іншої можливості.

Збереження стану операції. В оглядових відомостях про управління життєвим циклом операції коротко згадується, що в разі припинення або повної зупинки операції її стан зберігається. Це дійсно так, оскільки об'єкт **Activity** при цьому, як і раніше, знаходиться в пам'яті, і вся інформація про її елементи і поточний стан, як і раніше, активна. Тому будь-які внесені користувачем в опе-

рації зміни зберігаються, і коли операція повертається на передній план (коли вона «відновлюється»), ці зміни залишаються в цьому об'єкті.

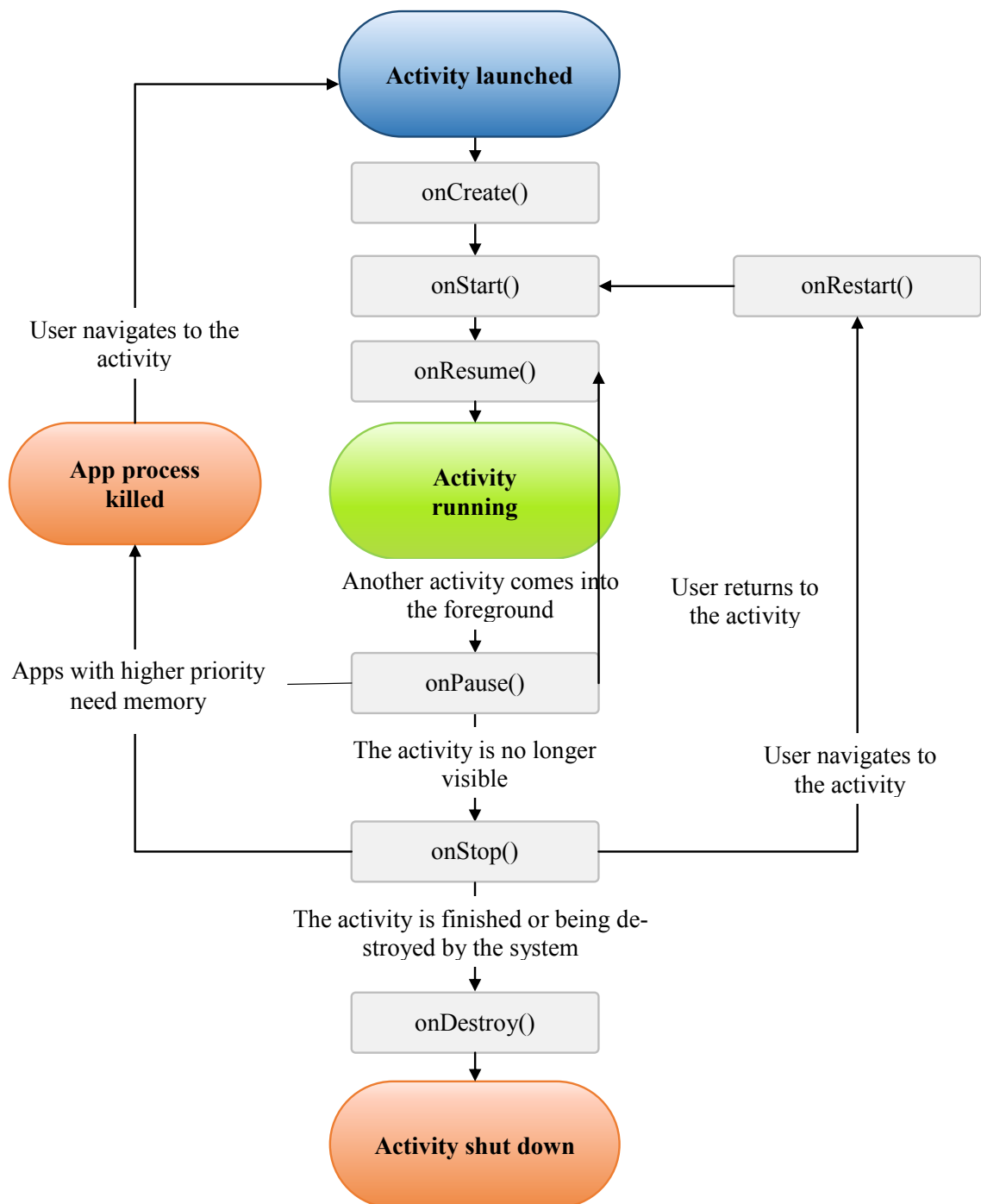
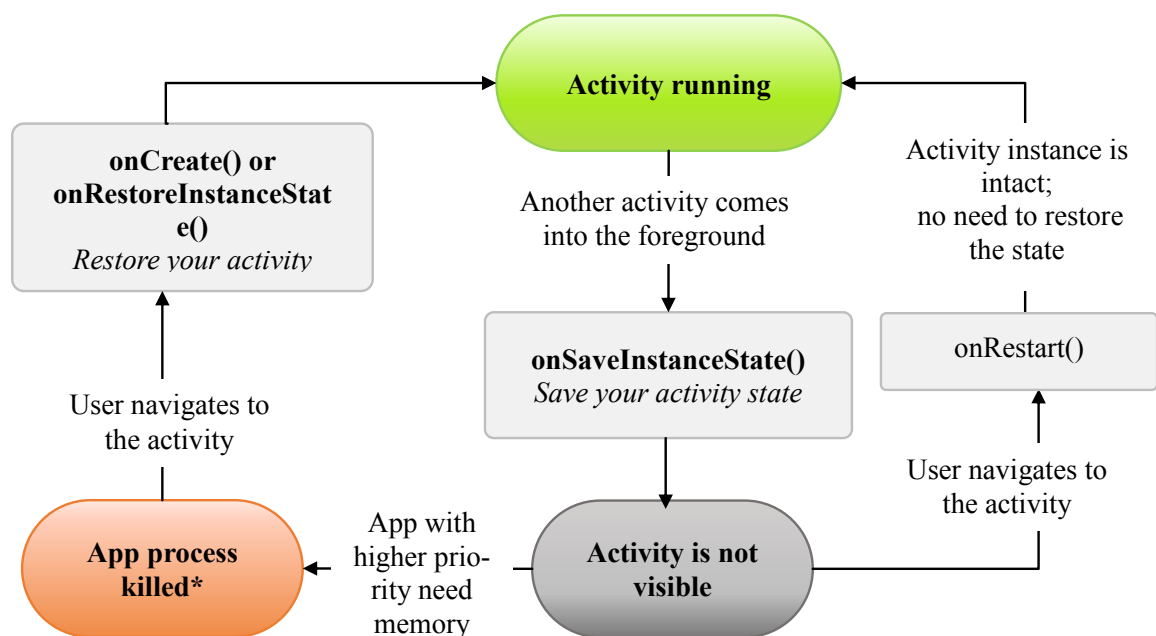


Рисунок 2.8 – Життєвий цикл операції

Однак коли система знищує операцію з метою відновлення пам'яті, об'єкт **Activity** знищується, в результаті чого системі не вдається просто від-

новити стан операції для взаємодії з нею. Замість цього системі необхідно повторно створити об'єкт **Activity**, якщо користувач повертається до нього. Але користувачеві невідомо, що система вже знищила операцію і створила її повторно, а тому, можливо, він очікує, що операція залишилася колишньою. У цій ситуації можна забезпечити збереження важливої інформації про стан операції шляхом реалізації додаткового методу зворотного виклику, який дозволяє зберегти інформацію про ваші операції: **onSaveInstanceState()**.

Перш ніж зробити операцію доступною для знищення, система викликає метод **onSaveInstanceState()**. Вона передає в цей метод об'єкт **Bundle**, в якому можна зберегти інформацію про стан операції у вигляді пари «ім'я-значення», використовуючи для цього такі методи, як **putString()** та **putInt()**. Потім, якщо система завершує процес вашого застосування і користувач повертається до вашої операції, система повторно створює операцію і передає об'єкт **Bundle** в обидва методи: **onCreate()** та **onRestoreInstanceState()**. За допомогою будь-якого з цих методів можна витягти з об'єкта **Bundle** збережену інформацію про стан операції і відновити її. Якщо така інформація відсутня, то об'єкт **Bundle** передається з нульовим значенням (це відбувається в разі, коли операція створюється в перший раз) (рис. 2.9).



*Activity instance is destroyed, but the state from **onSaveInstanceState()** is saved

Рисунок 2.9 – Способи повернення операції до відображення для користувача в незміненому стані

Є два способи повернення операції до відображення для користувача в незміненому стані: знищення операції з подальшим її повторним створенням, коли операція повинна відновити свій раніше збережений стан, або зупинка операції та її подальше відновлення в незміненому стані.

Примітка. Немає ніяких гарантій, що метод `onSaveInstanceState()` буде викликаний до того, як ваша операція буде знищена, оскільки існують випадки, коли немає необхідності зберігати стан (наприклад, коли користувач залишає вашу операцію натисканням кнопки *Назад*, явним чином закриваючи її). Якщо система викликає метод `onSaveInstanceState()`, вона робить це до виклику методу `onStop()` і, можливо, перед викликом методу `onPause()`.

Однак, навіть якщо ви нічого не робите і не реалізуєте метод `onSaveInstanceState()`, частина станів операції відновлюється реалізацією за замовчуванням методу `onSaveInstanceState()` класу `Activity`. Зокрема, реалізація за замовчуванням викликає відповідний метод `onSaveInstanceState()` для кожного об'єкта `View` в макеті, завдяки чому кожне подання може надавати ту інформацію про себе, яку слід зберегти. Майже кожен віджет у платформі Android реалізує цей метод необхідним для себе способом так, що будь-які видимі зміни в інтерфейсі автоматично зберігаються і відновлюються при повторному створенні операції. Наприклад, віджет `EditText` зберігає будь-який текст, який був введений, а віджет `CheckBox` – інформацію про те, чи був встановлений прапорець. Від вас вимагається лише вказати унікальний ідентифікатор (з атрибутом `android:id`) для кожного віджета, стан якого необхідно зберегти. Якщо віджету не присвоєно ідентифікатор, то системі не вдасться зберегти його стан.

Також можна явно відключити збереження інформації про стан подання в макеті. Для цього задайте для атрибута `android:saveEnabled` значення `"false"` або викличте метод `setSaveEnabled()`. Зазвичай відключати збереження такої інформації необхідне, проте це може знадобитися у випадках, коли відновити стан призначеного для користувача інтерфейсу операції необхідно іншим чином.

Незважаючи на те що реалізація методу `onSaveInstanceState()` за замовчуванням дозволяє зберегти корисну інформацію про користувацький інтерфейс вашої операції, вам, як і раніше, може знадобитися перевизначити її для збереження додаткової інформації. Наприклад, може знадобитися зберегти значення елементів, які змінювалися протягом життєвого циклу операції (які можуть корелювати зі значеннями, відновленими в інтерфейсі, проте елементи,

що містять ці значення призначеного для користувача інтерфейсу, за замовчуванням не були відновлені).

Оскільки реалізація методу `onSaveInstanceState()` за замовчуванням дозволяє зберегти стан призначеного для користувача інтерфейсу, то в разі перевизначення методу з метою збереження додаткової інформації про стан, перед виконанням будь-яких дій завжди можна викликати реалізацію суперкласу для методу `onSaveInstanceState()`. Так само реалізацію суперкласу `onRestoreInstanceState()` слід викликати в разі її перевизначення, щоб реалізація за замовчуванням могла зберегти стан подань.

Примітка. Оскільки виклик методу `onSaveInstanceState()` не гарантується, то слід використовувати його тільки для запису перехідного стану операції (стан призначеного для користувача інтерфейсу), а тому ніколи не використовуйте його для зберігання постійних даних. Замість цього використовуйте метод `onPause()` для збереження постійних даних (наприклад, тих, які слід зберегти в базу даних), коли користувач залишає операцію.

Відмінний спосіб перевірити спроможність вашого додатку відновлювати свій стан – це просто повернути пристрій для зміни орієнтації екрана. При зміні орієнтації екрана система знищує і повторно створює операцію, щоб застосувати альтернативні ресурси, які можуть бути доступні для нової конфігурації екрана. Тільки з однієї цієї причини вкрай важливо, щоб ваша операція могла повністю відновлювати свої стани при її повторному створенні, оскільки користувачі постійно працюють з додатками в різних орієнтаціях екрана.

Обробка змін в конфігурації. Деякі конфігурації пристроїв можуть змінюватися в режимі виконання (наприклад, орієнтація екрана, доступність клавіатури і мова). У таких випадках Android повторно створює виконання повсякденних завдань (система спочатку викликає метод `onDestroy()`, а потім відразу ж викликає метод `onCreate()`). Така поведінка дозволяє додатку враховувати нові конфігурації шляхом автоматичного перезавантаження в додаток альтернативних ресурсів, які були надані (наприклад, різні макети для різних орієнтацій і екранів різних розмірів).

Якщо операція розроблена належним чином і належним чином підтримує перезапуск після зміни орієнтації екрана і відновлення свого стану, як описано вище, ваш додаток можна вважати більш стійким до інших непередбачених подій в життєвому циклі операції.

Кращий спосіб обробки такого перезапуску – зберегти і відновити стан операції за допомогою методів `onSaveInstanceState()` та `onRestoreInstanceState()` (чи `onCreate()`).

Узгодження операцій. Коли одна операція запускає іншу, в життєвих циклах обох з них відбувається перехід з одного стану в інший. Перша операція припиняється і завершується (проте вона не буде зупинена, якщо вона, як і раніше, видима на фоні), а друга операція створюється. У разі, якщо ці операції обмінюються даними, збереженими на диску або в іншому місці, важливо розуміти, що перша операція не зупиняється повністю доти, поки не буде створена друга операція. Навпаки, процес запуску другої операції накладається на процес зупинки першої операції.

Порядок зворотних викликів життєвого циклу чітко визначено, зокрема, коли в одному і тому ж процесі знаходяться дві операції, і одна з них запускає іншу. Наведемо порядок виконання дій у разі, коли операція А запускає операцію Б:

1. Виконується метод `onPause()` операції А. Послідовно виконуються методи `onCreate()`, `onStart()` та `onResume()` операції Б (тепер для користувача відображається операція Б).

2. Потім, якщо операція А більше не відображається на екрані, виконується її метод `onStop()`.

Така передбачувана послідовність виконання зворотних викликів життєвого циклу дозволяє управляти переходом інформації з однієї операції в іншу. Наприклад, якщо після зупинки першої операції потрібно виконати запис в базу даних, щоб наступна операція могла зчитувати їх, то запис в базу даних слід виконати під час виконання методу `onPause()`, а не під час виконання методу `onStop()`.

2.2.2. Фрагменти

Фрагмент (клас `Fragment`) подає поведінку або частину призначеного для користувача інтерфейсу в операції (клас `Activity`). Розробник може об'єднати кілька фрагментів в одну операцію для побудови багатопанельного призначеного для користувача інтерфейсу і повторного використання фрагмента в декількох операціях. Фрагмент можна розглядати як модульну частину операції. Така частина має свій життєвий цикл і самостійно обробляє події введення. Крім того, її можна додати або видалити безпосередньо під час виконання операції. Це щось на зразок вкладеної операції, яку можна багаторазово використовувати в різних операціях.

Фрагмент завжди повинен бути вбудований в операцію. На його життєвий цикл безпосередньо впливає життєвий цикл операції. Наприклад, коли операція припинена, в тому ж стані знаходяться і всі фрагменти всередині неї, а коли

операція знищується, знищуються і всі фрагменти. Однак поки операція виконується (це відповідає стану відновлення життєвого циклу), можна маніпулювати кожним фрагментом незалежно, наприклад, додавати або видаляти їх. Коли розробник виконує такі транзакції з фрагментами, він може також додати їх у стек переходів назад, яким керує операція. Кожен елемент стека переходів назад в операції є записом виконаної транзакції з фрагментом. Стек переходів назад дозволяє користувачеві згорнути транзакцію з фрагментом (виконати навігацію в зворотному напрямку), натискаючи кнопку *Назад*.

Коли фрагмент доданий як частина макета операції, він знаходиться в об'єкті **ViewGroup** всередині ієрархії уявлень операції і визначає власний макет уявлень. Розробник може вставити фрагмент в макет операції двома способами. Для цього слід оголосити фрагмент у файлі макета операції як елемент `<fragment>` або додати його в існуючий об'єкт **ViewGroup** у коді програми. Втім фрагмент не зобов'язаний бути частиною макета операції. Можна використовувати фрагмент без інтерфейсу як невидимого робочого потоку операції.

У цьому документі показано, як побудувати додаток, що використовує фрагменти. Зокрема, обговорюється, як фрагменти можуть підтримувати свій стан, коли вони додаються в стек переходів назад операції, використовувати події спільно з операцією та іншими фрагментами всередині неї, виводити дані в рядок дій операції і т. д.

Філософія проектування. Фрагменти вперше з'явилися в Android версії 3.0 (API рівня 11), головним чином, для забезпечення більшої динамічності та гнучкості для користувача інтерфейсів на великих екранах, наприклад, у планшетів. Оскільки екрани планшетів набагато більші, ніж у смартфонів, вони дають більше можливостей для об'єднання і перестановки компонентів для користувача інтерфейсу. Фрагменти дозволяють робити це, позбавляючи розробника від необхідності управляти складними змінами в ієрархії подань. Розбиваючи макет операції на фрагменти, розробник отримує можливість модифікувати зовнішній вигляд операції в ході виконання і зберігати ці зміни в стеку переходів назад, яким керує операція.

Наприклад, новинний додаток може використовувати один фрагмент для показу списку статей зліва, а інший – для відображення статті справа. Обидва фрагменти відображаються за одну операцію поруч один з одним, і кожен має власний набір методів зворотного виклику життєвого циклу і управляє власними подіями призначеного для користувача введення. Таким чином, замість застосування однієї операції для вибору статті (телефон на рис. 2.9,) користувач може вибрати статтю і читати її в рамках однієї операції, як на планшеті, зображеному на рис. 2.10.

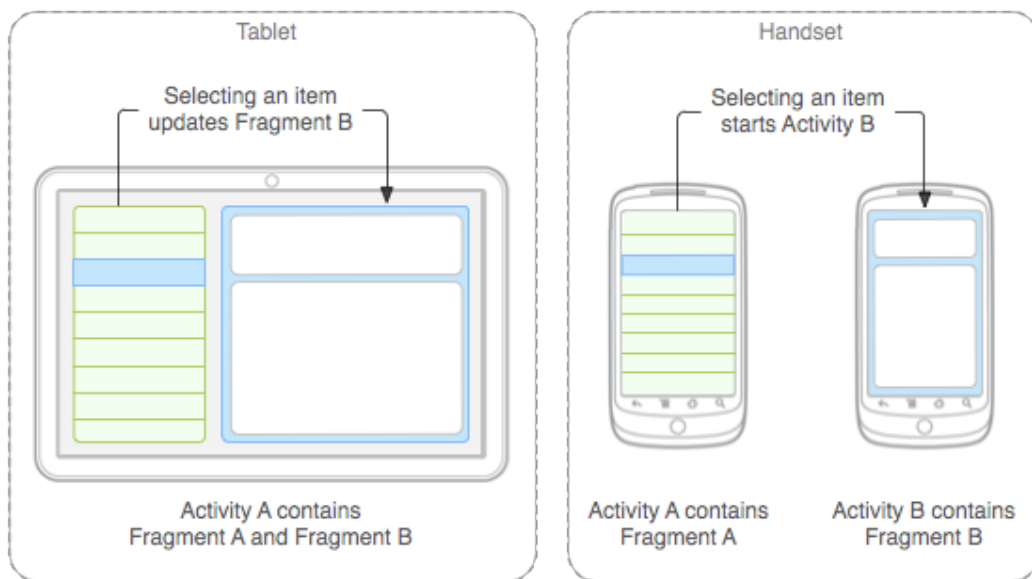


Рисунок 2.10 – Приклад об'єднання і перестановки компонентів

Слід розробляти кожен фрагмент як модульний і повторно використовуваний компонент операції. Оскільки кожен фрагмент визначає власний макет і власну поведінку зі своїми зворотними викликами життєвого циклу, розробник може включити один фрагмент в кілька операцій. Тому він повинен передбачити повторне використання фрагмента і не допускати, щоб один фрагмент безпосередньо маніпулював іншим. Це особливо важливо, тому що модульність фрагментів дозволяє змінювати їх поєднання відповідно до різних розмірів екранів. Якщо програма має працювати і на планшетах, і на смартфонах, можна повторно використовувати фрагменти в різних конфігураціях макета, щоб оптимізувати взаємодію з користувачем залежно від доступного розміру екрана. Наприклад, на смартфоні може виникнути необхідність у поділі фрагментів для надання однопанельного призначеного для користувача інтерфейсу, якщо розробнику не вдається помістити більше одного фрагмента в одну операцію.

Це є прикладом того, як два модулі призначені для користувача інтерфейсу, що визначені фрагментами, можуть бути об'єднані всередині однієї операції для роботи на планшетах, але розділені на смартфонах.

Повернемося до прикладу з новинним додатком. Він може мати два фрагменти, вбудовані в *Операцію А*, коли вона виконується на пристрої планшетного формату. У той же час на екрані смартфона недостатньо місця для обох фрагментів, і тому *Операція А* включає в себе тільки фрагмент зі списком статей. Коли користувач вибирає статтю, запускається *Операція В*, що містить другий фрагмент для читання статті. Таким чином, додаток підтримує як план-

шети, так і смартфони завдяки повторному використанню фрагментів в різних поєднаннях.

Створення фрагмента. Для створення фрагмента необхідно створити підклас класу `Fragment` (або його існуючого підкласу). Клас `Fragment` має код, багато в чому схожий з кодом `Activity`. Він містить методи зворотного виклику, аналогічні методам операції, таким як `onCreate()`, `onStart()`, `onPause()` та `onStop()`. На практиці, якщо бажаєте переробити існуючий Android-додаток так, щоб в ньому використовувалися фрагменти, досить просто перемістити код з методів зворотного виклику операції у відповідні методи зворотного виклику фрагмента.

Як правило, необхідно реалізувати такі методи життєвого циклу (рис. 2.11):

onCreate(). Система викликає цей метод, коли створює фрагмент. У своїй реалізації розробник повинен ініціалізувати ключові компоненти фрагмента, які необхідно зберігати, коли фрагмент знаходиться в стані паузи або коли він відновлений після зупинки.

onCreateView(). Система викликає цей метод при першому відображенні призначеного для користувача інтерфейсу фрагмента на дисплеї. Для промальовування призначеного для користувача інтерфейсу фрагмента слід повернути з цього методу об'єкт `View`, який є кореневим в макеті фрагмента. Якщо фрагмент не має призначеного для користувача інтерфейсу, можна повернути `null`.

onPause(). Система викликає цей метод як перше зазначення того, що користувач залишає фрагмент (це не завжди означає знищення фрагмента). Зазвичай саме в цей момент необхідно фіксувати всі зміни, які повинні бути збережені за рамками поточного сеансу роботи користувача (оскільки користувач може не повернутися назад).

У більшості додатків для кожного фрагмента повинні бути реалізовані, як мінімум, ці три методи. Однак існують і інші методи зворотного виклику, які слід використовувати для управління різними етапами життєвого циклу фрагмента. Існує також ряд підкласів, які, можливо, буде потрібно розширити замість використання базового класу `Fragment`:

– *DialogFragment.* Відображає переміщуване діалогове вікно. Використання цього класу для створення діалогового вікна є хорошою альтернативою допоміжних методів діалогового вікна в класі `Activity`. Справа в тому, що він дає можливість вставити діалогове вікно фрагмента в керований операцією стек переходів назад для фрагментів, що дозволяє користувачеві повернутися до закритого фрагмента.

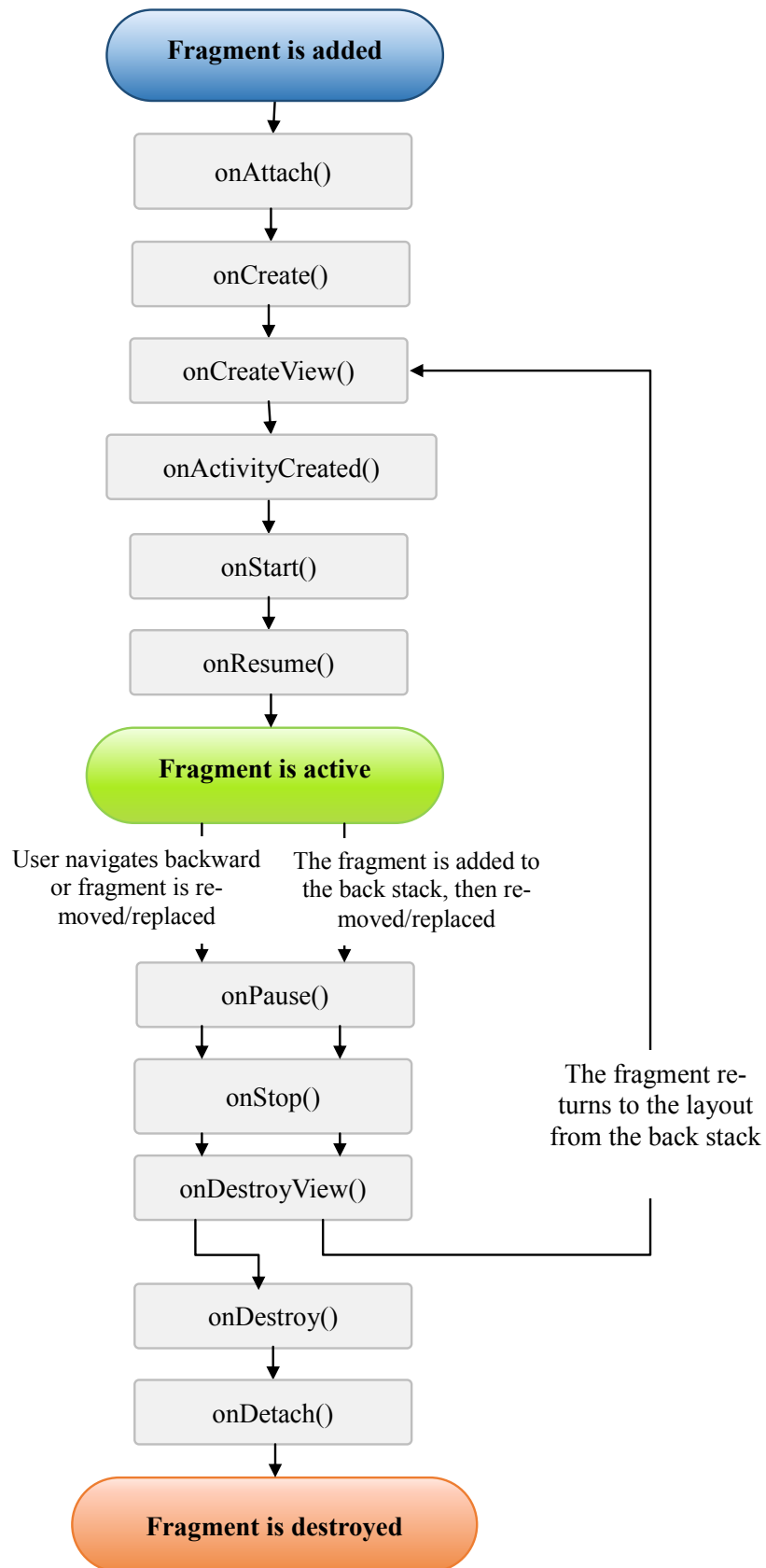


Рисунок 2.11 – Життєвий цикл фрагмента (під час виконання операції)

– *ListFragment*. Відображає список елементів, керованих адаптером (наприклад, SimpleCursorAdapter), аналогічно класу ListActivity. Цей клас надає кілька методів для управління списком уявлень, наприклад метод зворотного виклику onItemClick() для обробки натискань.

– *PreferenceFragment*. Відображає ієрархію об'єктів Preference у вигляді списку, аналогічно класу PreferenceActivity. Цей клас корисний, коли в додатку створюється операція «Налаштування».

Додавання інтерфейсу користувача. Фрагмент зазвичай використовується як частина користувацького інтерфейсу операції, при цьому він додає в операцію свій макет.

Щоб створити макет для фрагмента, розробник повинен реалізувати метод зворотного виклику onCreateView(), який система Android викликає, коли для фрагмента настає час відобразити свій макет. Реалізація цього методу повинна повертати об'єкт View, який є кореневим в макеті фрагмента.

Примітка. Якщо фрагмент є підкласом класу ListFragment, реалізація за замовчуванням повертає клас ListView з методу onCreateView(), а тому реалізовуватиме його немає потреби.

Щоб повернути макет з методу onCreateView(), можна виконати його розгортання з ресурсу макета, визначеного в XML-файлі. Для цієї мети метод onCreateView() надає об'єкт LayoutInflater.

Наприклад, код підкласу класу Fragment, що завантажує макет з файлу example_fragment.xml, може виглядати так:

```
public static class ExampleFragment extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        // Наповнити макет для цього фрагмента
        return inflater.inflate(R.layout.example_fragment, container, false);
    }
}
```

Параметр container, переданий методу onCreateView(), є батьківським класом ViewGroup (з макета операції), в який буде вставлений макет фрагмента. Параметр savedInstanceState є класом Bundle, який надає дані про попередній екземпляр фрагмента під час відновлення фрагмента.

Метод `inflate()` приймає три аргументи:

- ідентифікатор ресурсу макета, розгортання якого слід виконати;
- об'єкт класу `ViewGroup`, який повинен стати батьківським для макета після розгортання. Передача параметра `container` необхідна для того, щоб система, до якої направляється макет, змогла застосувати параметри макета до кореневого подання розгорнутого макета, який визначається батьківським поданням;
- логічне значення, яке показує, чи слід прикріпити макет до об'єкта `ViewGroup` (другий параметр) під час розгортання (в даному випадку це *false*, тому що система вже вставляє розгорнутий макет в об'єкт `container`, і передача значення *true* створила б зайву групу подання в остаточному макеті).

Ми побачили, як створювати фрагмент, що надає макет. Тепер необхідно додати фрагмент в операцію.

Додавання фрагмента в операцію. Як правило, фрагмент додає частину призначеного для користувача інтерфейсу в операцію, і цей інтерфейс вбудовується в загальну ієрархію подань операції. Розробник може додати фрагмент в макет операції двома способами:

1. Оголосивши фрагмент у файлі макета операції. В цьому випадку можна вказати властивості макета для фрагмента, як ніби він є поданням. Наприклад, файл макета операції з двома фрагментами може виглядати таким чином:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment android:name="com.example.news.ArticleListFragment"
        android:id="@+id/list"
        android:layout_weight="1"
        android:layout_width="0dp"
        android:layout_height="match_parent" />
    <fragment android:name="com.example.news.ArticleReaderFragment"
        android:id="@+id/viewer"
        android:layout_weight="2"
        android:layout_width="0dp"
        android:layout_height="match_parent" />
</LinearLayout>
```

Атрибут `android:name` в елементі `<fragment>` визначає клас `Fragment`, екземпляр якого створюється в макеті.

Коли система створює цей макет операції, вона створює екземпляр кожного фрагмента, визначеного в макеті, і для кожного викликає метод `onCreateView()`, щоб отримати макет кожного фрагмента. Система вставляє об'єкт `View`, повернутий фрагментом, безпосередньо замість елемента `<fragment>`.

Примітка. Кожен фрагмент повинен мати унікальний ідентифікатор, який система зможе використовувати для відновлення фрагмента в разі перезапуску операції. (Що стосується розробника, він може використовувати цей ідентифікатор для захоплення фрагмента з метою виконання транзакцій з ним, наприклад, щоб видалити його). Надати ідентифікатор фрагмента можна трьома способами:

- вказавши атрибут `android:id` з унікальним ідентифікатором;
- вказавши атрибут `android:tag` з унікальною рядком;
- нічого не робити, щоб система використовувала ідентифікатор контейнерного подання.

2. Програмним чином, додавши фрагмент в існуючий об'єкт `ViewGroup`. У будь-який момент виконання операції розробник може додати фрагменти в її макет. Для цього достатньо вказати об'єкт `ViewGroup`, в якому слід розмістити фрагмент.

Для виконання транзакцій з фрагментами всередині операції (таких, як додавання, видалення або заміна фрагмента) необхідно використовувати API-інтерфейси з `FragmentManager`. Екземпляр класу `FragmentManager` можна отримати від об'єкта `Activity` таким чином:

```
FragmentManager fragmentManager = getFragmentManager()  
FragmentManager fragmentManager =  
fragmentManager.beginTransaction();
```

Після цього можна додати фрагмент методом `add()`, вказавши додається фрагмент і подання, в яке він повинен бути доданий. Наприклад:

```
ExampleFragment fragment = new ExampleFragment();  
fragmentTransaction.add(R.id.fragment_container, fragment);  
fragmentTransaction.commit();
```

Перший аргумент, який передається методу `add()`, є контейнерним об'єктом, вказаним за допомогою ідентифікатора ресурсу. Другий параметр – це фрагмент, який потрібно додати.

Виконавши зміни за допомогою `FragmentManager`, необхідно викликати метод `commit()`, щоб вони вступили в силу.

Додавання фрагмента, що не має призначеного для користувача інтерфейсу. Приклад, наведений вище, демонструє, як додавати в операцію фрагмент з наданням призначеного для користувача інтерфейсу. Однак можна використовувати фрагмент і для реалізації фонові поведінки операції без будь-якого додаткового призначеного для користувача інтерфейсу.

Щоб додати фрагмент без користувацького інтерфейсу, додайте фрагмент з операції, використовуючи метод `add(Fragment, String)` (передавши йому унікальний рядковий «тег» для фрагмента замість ідентифікатора подання). Фрагмент буде додано, але, оскільки він не пов'язаний з поданням до макета операції, він не буде приймати виклик методу `onCreateView()`. Тому в реалізації цього методу немає потреби.

Передача рядкового тега властива не тільки фрагментам без користувацького інтерфейсу, а тому можна передавати рядкові теги і фрагменти, які мають користувацький інтерфейс. Однак, якщо у фрагмента немає призначеного для користувача інтерфейсу, то рядковий тег є єдиним способом його ідентифікації. Якщо згодом буде потрібно отримати фрагмент від операції, то треба буде викликати метод `findFragmentByTag()`.

Приклад операції, що використовує фрагмент як фоновий потік, без користувацького інтерфейсу, наведено в зразку коду `FragmentRetainInstance.java`, який входить в число зразків в SDK (і доступний за допомогою Android SDK Manager). Шлях до нього в системі – `sdk_root>/APIDemos/app/src/main/java/com/example/android/apis/app/FragmentRetainInstance.java`.

Управління фрагментами. Для управління фрагментами в операції потрібен клас `FragmentManager`. Щоб отримати його, слід викликати метод `getFragmentManager()` з коду операції.

Нижче вказані дії, які дозволяють виконати `FragmentManager`:

- отримувати фрагменти, наявні в операції, за допомогою методу `findFragmentById()` (для фрагментів, які надають інтерфейс користувача в макеті операції) чи `findFragmentByTag()` (як для фрагментів, що мають інтерфейс користувача, так і для фрагментів без нього);
- знімати фрагменти зі стека переходів назад методом `popBackStack()` (імітуючи натискання кнопки *Назад* користувачем);
- реєструвати процес-слухач змін в стеку переходів назад за допомогою методу `addOnBackStackChangeListener()`.

Додаткові відомості про ці та інші методи наводяться в документації по класу `FragmentManager`.

Як було показано вище, можна використовувати клас `FragmentManager` для відкриття `FragmentTransaction`, що дозволяє виконувати транзакції з фрагментами, наприклад додавання і видалення.

Виконання транзакцій з фрагментами. Великою перевагою використання фрагментів в операції є можливість додавати, видаляти, замінювати їх і виконувати інші дії з ними у відповідь на дії користувача. Будь-який набір змін, що вносяться в операцію, називається транзакцією. Її можна виконати за допомогою API-інтерфейсів у `FragmentTransaction`. Кожну транзакцію можна зберегти в стеку переходів назад, яким керує операція. Це дозволить користувачеві переміщатися назад щодо змін у фрагментах (аналогічно переміщенню назад за операціями).

Екземпляр класу `FragmentTransaction` можна отримати від `FragmentManager`, наприклад, так:

```
FragmentManager fragmentManager = getFragmentManager();

FragmentTransaction fragmentTransaction =
    fragmentManager.beginTransaction();
```

Кожна транзакція є набором змін, які виконуються одночасно. Розробник може вказати всі зміни, які йому потрібно виконати в даній транзакції, викликаючи методи `add()`, `remove()` та `replace()`. Потім, щоб застосувати транзакцію до операції, слід викликати метод `commit()`.

Втім до виклику методу `commit()` у розробника може виникнути необхідність викликати метод `addToBackStack()`, щоб додати транзакцію в стек переходів назад по транзакціях фрагмента. Цим стеком переходів назад управляє операція, що дозволяє користувачеві повернутися до попереднього стану фрагмента, натиснувши кнопку *Назад*.

Наприклад, наступний код демонструє, як можна замінити один фрагмент іншим, зберігаючи при цьому попередній стан в стеку переходів назад:

```
// Створити новий фрагмент і транзакцію

Fragment newFragment = new ExampleFragment();

FragmentTransaction transaction =
    getFragmentManager().beginTransaction();

// Замінити всі, що є в подання fragment_container, за допомогою
```

```
// цього фрагмента, і додати транзакцію в кінець стека
transaction.replace(R.id.fragment_container, newFragment);

transaction.addToBackStack(null);

// Зафіксувати транзакцію
transaction.commit();
```

У цьому коді об'єкт `newFragment` заміщає фрагмент (якщо такий є), що знаходиться в контейнері макета, на який вказує ідентифікатор `R.id.fragment_container`. У результаті виклику методу `addToBackStack()` транзакція заміни зберігається в стеку переходів назад, щоб користувач міг звернути транзакцію і повернути попередній фрагмент, натиснувши кнопку *Назад*.

Якщо в транзакцію додати кілька змін (наприклад, ще раз викликати `add()` чи `remove()`), а потім викликати `addToBackStack()`, всі зміни, застосовані до виклику методу `commit()`, будуть додані в стек переходів назад як одна транзакція, і кнопка *Назад* згорне їх всі разом.

Порядок додавання змін до об'єкта `FragmentTransaction` не відіграє ролі за такими виключеннями:

- метод `commit()` повинен бути викликаний в останню чергу;
- якщо в один контейнер додається кілька фрагментів, то порядок їх додавання визначає порядок, в якому вони з'являються в ієрархії видів.

Якщо при виконанні транзакції, що видаляє фрагмент, при фіксації транзакції фрагмент знищується, і користувач втрачає можливість повернутися до нього. У той же час, якщо викликати `addToBackStack()` при видаленні фрагмента, фрагмент перейде в стан зупину і буде відновлений, якщо користувач повернеться до нього.

Порада. До кожної транзакції з фрагментом можна застосувати анімацію переходу, викликавши `setTransition()` до фіксації.

Виклик методу `commit()` не призводить до негайного виконання транзакції. Метод запланує її виконання в потоці призначеного для користувача інтерфейсу операції (в «головному» потоці), як тільки у потоку з'явиться можливість для цього. Втім при необхідності можна викликати `executePendingTransactions()` з потоку призначеного для користувача інтерфейсу, щоб транзакції, заплановані методом `commit()`, були виконані негайно. Як правило, в цьому немає потреби, за винятком випадків, коли транзакція є залежністю для завдань в інших потоках.

Увага! Зафіксувати транзакцію методом `commit()` можна тільки до того, як операція збереже свій стан (після того, як користувач покине її). Спроба зафіксувати транзакцію після цього моменту викличе виключення. Справа в тому, що стан після фіксації може бути втрачений, якщо знадобиться відновити операцію. У ситуаціях, в яких втрата фіксації не критична, слід викликати `commitAllowingStateLoss()`.

Взаємодія з операцією. Хоча `Fragment` реалізований як об'єкт, незалежний від класу `Activity`, і може бути використаний всередині кількох операцій, конкретний екземпляр фрагмента безпосередньо пов'язаний з операцією, що містить його. Зокрема, фрагмент може звернутися до екземпляра `Activity` за допомогою методу `getActivity()` і без зусиль виконати такі задачі, як пошук уподання в макеті операції:

```
View listView = getActivity().findViewById(R.id.list);
```

Аналогічним чином операція може викликати методи фрагмента, отримавши посилання на об'єкт `Fragment` від `FragmentManager` за допомогою методу `findFragmentById()` чи `findFragmentByTag()`. Наприклад:

```
ExampleFragment fragment = (ExampleFragment)
getFragmentManager().findFragmentById(R.id.example_fragment);
```

Створення зворотного виклику події для операції. У деяких випадках необхідно, щоб фрагмент використовував події спільно з операцією. Хороший спосіб реалізації цього полягає в тому, щоб визначити інтерфейс зворотного виклику всередині фрагмента і зажадати від контейнерної операції його реалізації. Коли операція прийме зворотний виклик через цей інтерфейс, вона зможе обмінюватися інформацією з іншими фрагментами в макеті в міру необхідності.

Наприклад, у новинного додатка є два фрагменти в одній операції: один – для відображення списку статей (фрагмент А), а інший – для відображення статті (фрагмент В). Тоді фрагмент А повинен повідомляти операцію про обраний пункт списку, щоб вона могла повідомити фрагменту В про необхідність відобразити статтю. В цьому випадку інтерфейс `OnArticleSelectedListener` оголошується у фрагменті А:

```
public static class FragmentA extends ListFragment {
    ...
    // Container Activity должен реализовывать этот интерфейс
    public interface OnArticleSelectedListener {
        public void onArticleSelected(Uri articleUri);
    }
}
```

```

    }
    ...
}

```

Тоді операція, яка містить цей фрагмент, реалізує інтерфейс `OnArticleSelectedListener` та перевизначає метод `onArticleSelected()`, щоб сповіщати фрагмент В про подію, що виходить від фрагмента А. Щоб контейнерна операція напевно реалізувала цей інтерфейс, метод зворотного виклику `onAttach()` у фрагменті А (який система викликає при додаванні фрагмента в операцію) створює екземпляр класу `OnArticleSelectedListener`, виконавши приведення типу об'єкта `Activity`, який передається методу `onAttach()`:

```

public static class FragmentA extends ListFragment {
    OnArticleSelectedListener mListener;
    ...
    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);
        try {
            mListener = (OnArticleSelectedListener) activity;
        } catch (ClassCastException e) {
            throw new ClassCastException(activity.toString() + "
must implement OnArticleSelectedListener");
        }
    }
    ...
}

```

Якщо операція не реалізувала інтерфейс, фрагмент генерує виключення `ClassCastException`. У разі успіху елемент `mListener` буде містити посилання на реалізацію інтерфейсу `OnArticleSelectedListener` в операції, щоб фрагмент А міг використовувати події спільно з операцією, викликаючи методи, визначені інтерфейсом `OnArticleSelectedListener`. Наприклад, якщо фрагмент А є розширенням класу `ListFragment`, то всякий раз, коли користувач натискає елемент списку, система викликає `onListItemClick()` у фрагменті. Цей метод, в свою чергу, викликає метод `onArticleSelected()`, щоб використовувати події спільно з операцією:

```

public static class FragmentA extends ListFragment {
    OnArticleSelectedListener mListener;
    ...
    @Override
    public void onListItemClick(ListView l, View v, int position,
long id) {
        // Append the clicked item's row ID with the content provid-
er Uri
        Uri noteUri =

```

```

ContentUri.withAppendedId(ArticleColumns.CONTENT_URI, id);
// Send the event and Uri to the host activity
mListener.onArticleSelected(noteUri);
    }
    ...
}

```

Параметр `id`, що передається методу `onListItemClick()`, – це ідентифікатор рядка з обраним елементом списку, який Activity (або інший фрагмент) використовує для отримання даних від об'єкта `ContentProvider` додатка.

Додавання елементів в рядок дій. Фрагменти можуть додавати пункти меню в Меню варіантів операції (і, отже, в Рядок дій), реалізувавши `onCreateOptionsMenu()`. Однак, щоб цей метод міг приймати виклики, необхідно викликати `setHasOptionsMenu()` під час виконання методу `onCreate()`, щоб повідомити, що фрагмент має намір додати пункти в Меню варіантів (в іншому випадку фрагмент не прийме виклик методу `onCreateOptionsMenu()`).

Будь-які пункти, що додаються фрагментом в *Меню варіантів*, приєднуються до вже існуючих. Крім того, фрагмент приймає зворотні виклики методу `onOptionsItemSelected()`, коли користувач вибирає пункт меню.

Розробник може також зареєструвати подання в макеті свого фрагмента, щоб надати контекстне меню. Для цього слід викликати метод `registerForContextMenu()`. Коли користувач відкриває контекстне меню, фрагмент приймає виклик методу `onCreateContextMenu()`. Коли користувач вибирає пункт меню, фрагмент приймає виклик методу `onContextItemSelected()`.

Примітка. Хоча фрагмент приймає зворотний виклик за подією «обраний пункт меню» для кожного доданого їм пункту, операція першою приймає відповідний зворотний виклик, коли користувач вибирає пункт меню. Якщо наявна в операції реалізація зворотного виклику за подією «обраний пункт меню» і не виконує жодний обраний пункт, подія передається методу зворотного виклику у фрагменті. Це справедливо для *Меню варіантів* і контекстних меню.

Докладні відомості щодо меню див. у посібниках для розробників «Меню» і «Рядок дій».

Управління життєвим циклом фрагмента. Управління життєвим циклом фрагмента багато в чому аналогічно управлінню життєвим циклом операції. Як і операція, фрагмент може існувати в одному з трьох станів:

- 1) відновлений (Resumed) – фрагмент видно під час виконання операції;
- 2) призупинений (Paused) – на передньому плані виконується і знаходиться в фокусі інша операція, але операція, яка містить даний фрагмент, як і раніше, видна (операція переднього плану частково прозора або не займає весь екран);
- 3) зупинений (Stopped).

Фрагмент непомітний. Або контейнерна операція зупинена, або фрагмент видалений з неї, але доданий в стек переходів назад. Зупинений фрагмент є активним (вся інформація про стан і елементи збережена в системі). Однак він більше не видно користувачеві і буде знищений в разі знищення операції. Тут знову проглядається аналогія з операцією: розробник може зберегти стан фрагмента за допомогою **Bundle** на випадок, якщо процес операції буде знищений, а розробнику знадобиться відновити стан фрагмента при повторному створенні операції. Стан можна зберегти під час виконання методу зворотного виклику `onSaveInstanceState()` у фрагменті і відновити його під час виконання `onCreate()`, `onCreateView()` чи `onActivityCreated()`. Додаткові відомості про збереження стану наводяться в документі «Операції».

Найзначніша відмінність в ході життєвого циклу між операцією і фрагментом полягає в принципах їх збереження у відповідних стеках переходів назад. За замовчуванням операція поміщається в керований системою стек переходів назад для операцій, коли вона зупиняється (щоб користувач міг повернутися до неї за допомогою кнопки *Назад*. У той же час фрагмент поміщається в стек переходів назад, керований операцією, тільки коли розробник явно запросить збереження конкретного екземпляра, викликавши метод `addToBackStack()` під час транзакції, що видаляє фрагмент.

В іншому управлінні життєвим циклом фрагмента дуже схоже на управління життєвим циклом операції. Тому практичні рекомендації з *управління життєвим циклом операцій* застосовні і до фрагментів. При цьому розробнику необхідно розуміти, як життєвий цикл операції впливає на життєвий цикл фрагмента.

Увага! Якщо виникне потреба в об'єкті **Context** всередині об'єкта класу **Fragment**, можна викликати метод `getActivity()`. Однак розробник повинен

бути уважним і викликати метод `getActivity()`, тільки коли фрагмент прикріплений до операції. Якщо фрагмент ще не прикріплений або був відкріплений в кінці його життєвого циклу, метод `getActivity()` поверне `null`.

Узгодження з життєвим циклом операції. Життєвий цикл операції, що містить фрагмент, безпосереднім чином впливає на життєвий цикл фрагмента, а тому кожен зворотний виклик життєвого циклу операції приводить до аналогічного зворотного виклику для кожного фрагмента. Наприклад, коли операція приймає виклик `onPause()`, кожен її фрагмент приймає `onPause()`.

Однак у фрагментів є кілька додаткових методів зворотного виклику життєвого циклу, які забезпечують унікальну взаємодію з операцією для виконання таких дій, як створення і знищення призначеного для користувача інтерфейсу фрагмента. Це такі методи:

1) *`onAttach()`* – викликається, коли фрагмент зв'язується з операцією (йому передається об'єкт `Activity`);

2) *`onCreateView()`* – викликається для створення ієрархії подань, пов'язаної з фрагментом;

3) *`onActivityCreated()`* – викликається, коли метод `onCreate()`, що належить операції, повертає управління;

4) *`onDestroyView()`* – викликається при видаленні ієрархії подань, пов'язаної з фрагментом;

5) *`onDetach()`* – викликається при розриві зв'язку фрагмента з операцією.

Залежність життєвого циклу фрагмента від операції, що містяться у ньому, ілюструється рис. 2.12.

На цьому рисунку можна бачити, що черговий стан операції визначає, які методи зворотного виклику може приймати фрагмент. Наприклад, коли операція приймає свій метод зворотного виклику `onCreate()`, фрагмент всередині цієї операції приймає всього лише метод зворотного виклику `onActivityCreated()`.

Коли операція переходить в стан «відновлена», можна вільно додавати в неї фрагменти і видаляти їх. Таким чином, життєвий цикл фрагмента може бути незалежно змінений, тільки поки операція залишається в стані «відновлена».

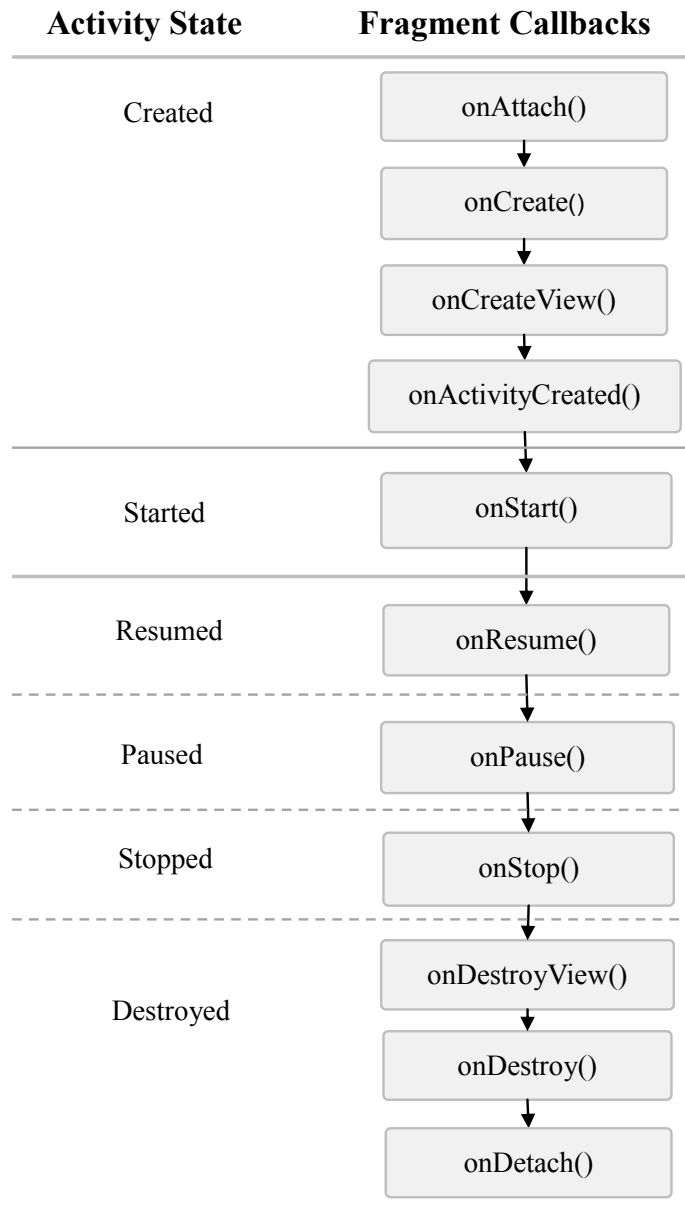


Рисунок 2.12 – Вплив життєвого циклу операції на життєвий цикл фрагмента

Однак коли операція виходить з цього стану, просування фрагмента по його життєвому циклу знову здійснюється операцією.

Приклад. Щоб підсумувати все сказане в цьому документі, розглянемо приклад операції, що використовує два фрагменти для створення макета з двома панелями. Операція, код якої наведено нижче, включає в себе один фрагмент для відображення списку п'єс Шекспіра, а інший – для відображення короткого змісту п'єси, обраної зі списку. У прикладі показано, як слід організовувати різні конфігурації фрагментів залежно від конфігурації екрана.

Примітка. Повний вихідний код цієї операції знаходиться в файлі `FragmentManager.java`.

Головна операція застосовує макет звичайним способом, у методі `onCreate()`:

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

Тут застосовується макет `fragment_layout.xml`:

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"    an-
droid:layout_height="match_parent">
    <fragment
class="com.example.android.apis.app.FragmentManager$TitlesFragment"
        android:id="@+id/titles" android:layout_weight="1"
        android:layout_width="0px" an-
droid:layout_height="match_parent" />
    <FrameLayout android:id="@+id/details" android:layout_weight="1"
        android:layout_width="0px" an-
droid:layout_height="match_parent"
        an-
droid:background="?android:attr/detailsElementBackground"/>
</LinearLayout>
```

Користуючись цим макетом, система створює екземпляр класу `TitlesFragment` (список фрагментів), як тільки операція завантажить макет. При цьому об'єкт `FrameLayout` (в якому буде знаходитися фрагмент з коротким змістом) займає місце в правій частині екрана, але спочатку залишається порожнім. Як буде показано нижче, фрагмент не поміщається в `FrameLayout`, поки користувач не вибере його зі списку.

Однак не всі екрани досить широкі, щоб відображати короткий зміст поруч зі списком п'єс. Тому описаний вище макет використовується тільки при альбомній орієнтації екрана і зберігається у файлі `res/layout-land/fragment_layout.xml`.

Коли ж пристрій знаходиться в книжковій орієнтації, система застосовує макет, наведений нижче, який зберігається у файлі `res/layout/fragment_layout.xml`:

```

<FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" an-
droid:layout_height="match_parent">
    <fragment
class="com.example.android.apis.app.FragmentLayout$TitlesFragment"
    android:id="@+id/titles"
    android:layout_width="match_parent" an-
droid:layout_height="match_parent" />
</FrameLayout>

```

У цьому макеті присутній тільки об'єкт `TitlesFragment`. Це означає, що при книжковій орієнтації пристрою видно тільки список п'єс. Коли користувач натискає на елемент списку в цій конфігурації, додаток запускає нову операцію для відображення короткого змісту, а не завантажує другий фрагмент.

Далі можна бачити, як це реалізовано в класах фрагмента. Спочатку йде код класу `TitlesFragment`, що відображає список п'єс Шекспіра. Цей фрагмент є розширенням класу `ListFragment` і використовує його функції для виконання основної роботи зі списком:

```

public static class TitlesFragment extends ListFragment {
    boolean mDualPane;
    int mCurCheckPosition = 0;

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);

        // Заповнюємо список з нашим статичним масивом назв.
        setListAdapter(new ArrayAdapter<String>(getActivity(),
            android.R.layout.simple_list_item_activated_1,
            Shakespeare.TITLES));

        // Перевіряємо, щоб побачити, якщо у нас є кадр, в якому
        вставляти
        // деталі фрагмента безпосередньо в містить UI.
        View detailsFrame =
        getActivity().findViewById(R.id.details);
        mDualPane = detailsFrame != null &&
        detailsFrame.getVisibility() == View.VISIBLE;

        if (savedInstanceState != null) {
            // Відновлюємо останній стан для перевіряємої позиції.
            mCurCheckPosition =
            savedInstanceState.getInt("curChoice", 0);
        }

        if (mDualPane) {
            // У двоканальному режимі панелі, подання списку виділяє

```

```

        //вибраний елемент.

getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE);
        // Переконаймося, що наш користувальницький інтерфейс
знаходиться в
        // правильному стані.
        showDetails(mCurCheckPosition);
    }
}

@Override
public void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    outState.putInt("curChoice", mCurCheckPosition);
}

@Override
public void onListItemClick(ListView l, View v, int position, long id) {
    showDetails(position);
}
/**
 * Допоміжна функція для відображення деталей обраного елемента або по
 * відображенню фрагмента на місці в поточному інтерфейсі або
 * запуск цілого нового дії, в якому воно відображається.
 */
void showDetails(int index) {
    mCurCheckPosition = index;

    if (mDualPane) {
        // Ми можемо відображати всі на місці з фрагментами, тому оновлюємо
        // список, щоб виділити виділений елемент і показати дані.
        getListView().setItemChecked(index, true);

        // Перевіряємо, який фрагмент відображається в даний момент, при
        // необхідності замінюємо.
        DetailsFragment details = (DetailsFragment)
            getFragmentManager().findFragmentById(R.id.details);
        if (details == null || details.getShownIndex() != index) {
            // Створюємо новий фрагмент, щоб відобразити цей вибір.
            details = DetailsFragment.newInstance(index);

            // Виконуємо транзакцію, замінивши будь-який існуючий фрагмент
            // з цим всередині рамки.
            FragmentTransaction ft =
getFragmentManager().beginTransaction();
            if (index == 0) {
                ft.replace(R.id.details, details);
            } else {
                ft.replace(R.id.a_item, details);
            }
            ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
            ft.commit();
        }
    } else {
        // В іншому випадку нам потрібно запустити нову дію для
        // відображення фрагмента діалогу з виділеним текстом.
        Intent intent = new Intent();
        intent.setClass(getActivity(), DetailsActivity.class);
    }
}

```

```

        intent.putExtra("index", index);
        startActivity(intent);
    }
}

```

Вивчаючи код, зверніть увагу на те, що в як реакція на натискання користувачем на елемент списку можливі дві моделі поведінки. Залежно від того, який з двох макетів активний, або в рамках однієї операції створюється і відображається новий фрагмент з коротким змістом (за рахунок додавання фрагмента в об'єкт `FrameLayout`), або запускається нова операція (що відображає фрагмент).

Другий фрагмент, `DetailsFragment`, відображає короткий зміст п'єси, обраної в списку `TitlesFragment`:

```

public static class DetailsFragment extends Fragment {
    /**
     * Create a new instance of DetailsFragment, initialized to
     * show the text at 'index'.
     */
    public static DetailsFragment newInstance(int index) {
        DetailsFragment f = new DetailsFragment();

        // Supply index input as an argument.
        Bundle args = new Bundle();
        args.putInt("index", index);
        f.setArguments(args);

        return f;
    }

    public int getShownIndex() {
        return getArguments().getInt("index", 0);
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        if (container == null) {
            // We have different layouts, and in one of them this
            // fragment's containing frame doesn't exist. The frag-
            // ment
            // may still be created from its saved state, but there
            // is
            // no reason to try to create its view hierarchy because
            // it
            // won't be displayed. Note this is not needed -- we
            // could
            // just run the code below, where we would create and

```

```

return
    // the view hierarchy; it would just never be used.
    return null;
}

ScrollView scroller = new ScrollView(getActivity());
TextView text = new TextView(getActivity());
int padding =
(int)TypedValue.applyDimension(TypedValue.COMPLEX_UNIT_DIP,
    4,
getActivity().getResources().getDisplayMetrics());
text.setPadding(padding, padding, padding, padding);
scroller.addView(text);
text.setText(Shakespeare.DIALOGUE[getShownIndex()]);
return scroller;
}
}

```

Згадаймо код класу `TitlesFragment`: якщо користувач натискає на пункт списку, а поточний макет *не* включає в себе подання `R.id.details` (якому належить фрагмент `DetailsFragment`), то додаток запускає операцію `DetailsActivity` для відображення вмісту елемента.

Далі йде код класу `DetailsActivity`, який всього лише містить об'єкт `DetailsFragment` для відображення короткого змісту обраної п'єси на екрані в книжковій орієнтації:

```

public static class DetailsActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        if (getResources().getConfiguration().orientation
            == Configuration.ORIENTATION_LANDSCAPE) {
            // If the screen is now in landscape mode, we can show
            // dialog in-line with the list so we don't need this
            // activity.
            finish();
            return;
        }

        if (savedInstanceState == null) {
            // During initial setup, plug in the details fragment.
            DetailsFragment details = new DetailsFragment();
            details.setArguments(getIntent().getExtras());

            getFragmentManager().beginTransaction().add(android.R.id.content,
                details).commit();
        }
    }
}

```

```
}  
}  
}
```

Зверніть увагу, що в альбомній конфігурації ця операція самостійно завершується, щоб головна операція могла прийняти управління і відобразити фрагмент `DetailsFragment` поруч із фрагментом `TitlesFragment`. Це може статися, якщо користувач запустить операцію `DetailsActivity` в книжковій орієнтації екрана, а потім переверне пристрій на режим пейзажу (у результаті чого поточна операція буде перезапущена).

2.3 Завдання і стек переходів назад (Tasks and Back Stack)

Зазвичай додаток містить кілька операцій. Кожна операція повинна розроблятися в зв'язку з дією певного типу, яку користувач може виконувати, і може запускати інші операції. Наприклад, програма електронної пошти може містити одну операцію для відображення списку нових повідомлень. Коли користувач вибирає повідомлення, відкривається нова операція для перегляду цього повідомлення.

Операція може навіть запускати операції, що існують в інших додатках на пристрої. Наприклад, якщо ваше додаток хоче відправити повідомлення електронної пошти, можна визначити намір для виконання дії «відправити» і включити в нього деякі дані, наприклад адресу електронної пошти та текст повідомлення. Після цього відкривається операція з іншої програми, яка оголосила, що вона обробляє наміри такого типу. В цьому випадку намір полягає в тому, щоб відправити повідомлення електронної пошти, а тому в додатку електронної пошти запускається операція «скласти повідомлення» (якщо один намір може оброблятися декількома операціями, система пропонує користувачу вибрати, яку з операцій використовувати). Після відправлення повідомлення електронної пошти ваша операція відновлює роботу, і все виглядає так, ніби операція відправлення електронної пошти є частиною вашої програми. Хоча операції можуть бути частинами різних додатків, система Android підтримує зручність роботи користувача, зберігаючи обидві операції в одному завданні.

Завдання – це колекція операцій, з якими взаємодіє користувач при виконанні певного завдання. Операції впорядковані у вигляді стека (стека переходів назад) в тому порядку, в якому відкривалися операції.

Початковим місцем для більшості завдань є головний екран пристрою. Коли користувач торкається значка в засобі запуску додатків (або ярлика на головному екрані), це завдання додатка переходить на передній план. Якщо для програми немає завдань (додаток не використовувався останнім часом), тоді

створюється нове завдання і відкривається «основна» операція для цього додатка як коренева операція в стеку.

Коли поточна операція запускає іншу, нова операція поміщається в вершину стека і отримує фокус. Попередня операція залишається в стеку, але її виконання зупиняється. Коли операція зупиняється, система зберігає поточний стан її користувацького інтерфейсу. Коли користувач натискає кнопку *Назад*, поточна операція видаляється з вершини стека (операція знищується), і поновлюється робота попередньої операції (відновлюється попередній стан її користувацького інтерфейсу). Операції в стеку ніколи не змінюють порядок, тільки додаються в стек і видаляються з нього – додаються в стек при запуску поточної операції і видаляються, коли користувач виходить з неї за допомогою кнопки *Назад*. По суті стек переходів назад працює за принципом «останнім увійшов – першим вийшов».

На рис. 2.13 ця поведінка показана на часовій шкалі: стан операцій і поточний стан стека переходів назад показано в кожен момент часу.

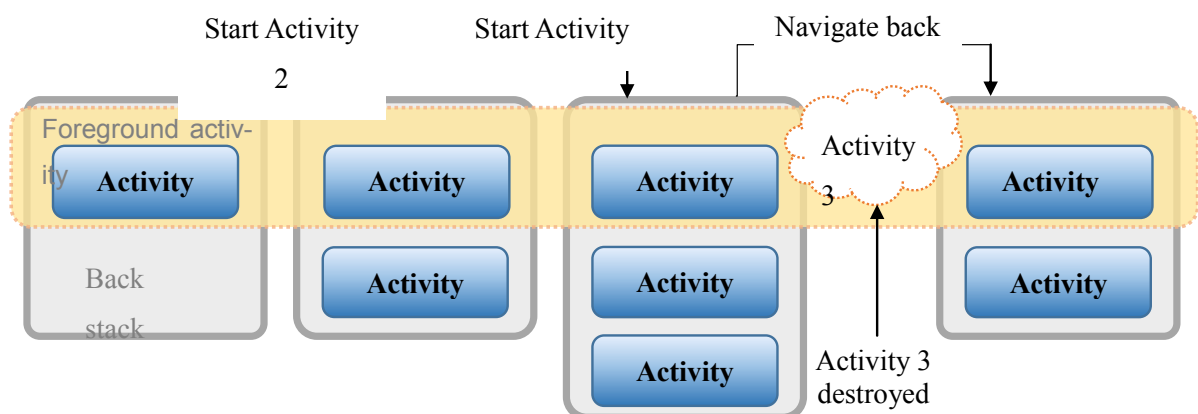


Рисунок 2.13 – Стан операцій і поточний стан стека переходів назад в кожен момент часу

Цей рисунок ілюструє, як кожна нова операція в завданні додає елемент в стек переходів назад. Коли користувач натискає кнопку *Назад*, поточна операція знищується, і відновлюється робота попередньої операції.

Якщо користувач продовжує натискати кнопку *Назад*, операції по черзі видаляються зі стека, відкриваючи попередню операцію, поки користувач не повернеться на головний екран (або в операцію, яка була запущена на початку виконання завдання). Коли всі операції видалені зі стека, завдання припиняє існування.

На рис. 2.14 показано дві задачі: задачу В, яка взаємодіє з користувачем на передньому плані; задачу А, що знаходиться у фоновому режимі, чекаючи відновлення.

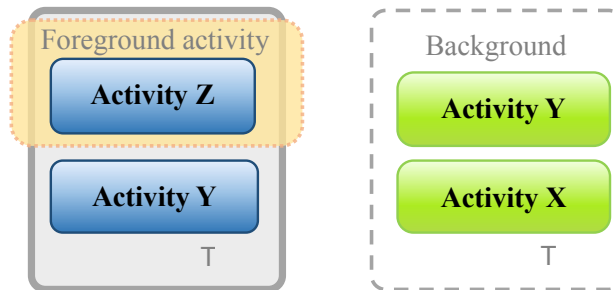


Рисунок 2.14 – Фоновий та активний режими задач

Задача – пов’язаний блок, який може переходити у фоновий режим, коли користувачі починають нову задачу або переходять на головний екран за допомогою кнопки *Додому*. У фоновому режимі всі операції задачі зупинені, але стек зворотного виклику для завдання залишається незмінним. Задача просто втратила фокус під час виконання іншої задачі, як показано на рис. 2.15

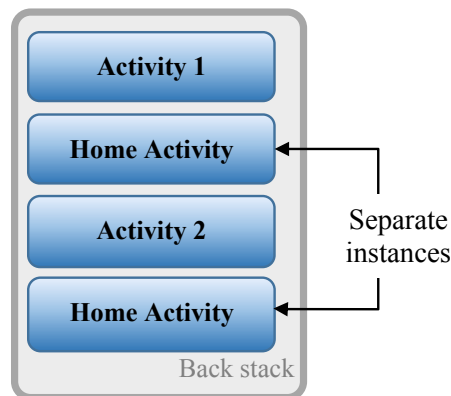


Рисунок 2.15 – Створення кількох примірників однієї операції

Потім задача може повернутися на передній план, а отже, користувачі можуть продовжити її з перерваного місця. Припустимо, наприклад, що поточна задача (задача А) містить три операції в своєму стеку – дві операції під поточною операцією. Користувач натискає кнопку *Додому*, а потім запускає новий додаток із засобу запуску додатків. Коли з’являється головний екран, задача А переходить у фоновий режим. Коли запускається новий додаток, система запускає задачу для цього додатка (задача В) зі своїм власним стеком операцій. Після взаємодії з цим додатком користувач знову повертається на головний екран і вибирає спочатку запущену задачу А. Тепер задача А переходить на передній

план – всі три операції її стека залишилися незмінними, і поновлюється операція, яка перебуває на вершині стека. У цей момент користувач може також переключитися назад на задачу В, перейшовши на головний екран і вибравши значок програми, яка запустила цю задачу (або вибравши задачу додатка на екрані огляду). Це приклад багатозадачності в системі Android.

Примітка. У фоновому режимі може знаходитися декілька задач одночасно. Однак якщо користувач запускає багато фонових задач одночасно, система може почати знищення фонових операцій для звільнення пам'яті, що призведе до втрати стану завдань.

Оскільки операції в стеку ніколи не змінюють порядок, якщо ваш додаток дозволяє користувачам запускати певну операцію з декількох операцій, новий екземпляр такої операції створюється і поміщається в стек (замість розміщення будь-якого з попередніх екземплярів операції на вершину стека). По суті для однієї операції вашого додатка може бути створено кілька екземплярів (навіть з різних завдань), як показано на рис. 2.15. Тому, якщо користувач переходить назад за допомогою кнопки *Назад*, кожен екземпляр операції з'являється в тому порядку, в якому він був відкритий (кожен зі своїм станом призначеного для користувача інтерфейсу). Однак можна змінити цю поведінку, якщо не треба, щоб створювалося кілька екземплярів операції.

Підіб'ємо підсумки поведінки операцій і задач:

1. Коли операція А запускає операцію В, операція А зупиняється, але система зберігає її стан (наприклад, положення прокручування і текст, введений у форми). Якщо користувач натискає кнопку *Назад* в операції В, операція А відновлює роботу зі збереженого стану.

2. Коли користувач виходить із завдання натисненням кнопки *Додому*, поточна операція зупиняється і її завдання переводиться у фоновий режим. Система зберігає стан кожної операції в задачі. Якщо користувач згодом відновлює задачу, вибираючи значок запуску задачі, вона перекладається на передній план і відновлює операцію на вершині стека.

3. Якщо користувач натискає кнопку *Назад*, поточна операція видаляється зі стека і знищується. Відновлюється попередня операція в стеку. Коли операція знищується, система не зберігає стан операції.

4. Можна створювати кілька екземплярів операції, навіть з інших задач.

2.3.1. Збереження стану операції

Як йшлося вище, система за замовчуванням зберігає стан операції, коли вона зупиняється. Таким чином, коли користувачі повертаються назад в попередню операцію, відновлюється її користувацький інтерфейс в момент зупинки. Однак можна, і треба, з попередженням зберігати стан ваших операцій за допомогою методів зворотного виклику на випадок знищення операції і необхідності її повторного створення.

Коли система зупиняє одну з ваших операцій (наприклад, коли запускається нова операція або коли завдання переміщається у фоновий режим), вона може повністю знищити цю операцію, якщо необхідно відновити пам'ять системи. Коли це відбувається, інформація про стан операції втрачається, а система знає, що операція знаходиться в стеку переходів назад. Але коли операція переходить на вершину стека, система повинна створити її повторно (а не відновити її). Щоб уникнути втрати роботи користувача, необхідно з попередженням зберігати її шляхом реалізації методів зворотного виклику `onSaveInstanceState()` у вашій операції.

2.3.2. Управління задачами

Для більшості додатків спосіб, яким Android керує задачами і стеком переходів назад, описаний вище (переміщення всіх операцій послідовно в одну задачу в стек за принципом «останнім увійшов – першим вийшов»), працює добре, і ви не повинні турбуватися про зв'язок ваших операцій з задачами або про їхнє існування в стеку переходів назад. Однак може виникнути ряд ситуацій, коли цей спосіб не придатний, наприклад, якщо потрібно перервати звичайну поведінку додатка або необхідно, щоб операція у вашому додатку починала нову задачу при запуску (замість переміщення в поточну задачу), або при запуску операції необхідно перенести її існуючий екземпляр на передній план (замість створення нового екземпляра на вершині стека переходів назад), або ви хочете, щоб при виході користувача із задачі зі стеку переходів видалялися всі операції, крім кореневої.

Ці та багато інших дій можна здійснювати за допомогою атрибутів в елементі маніфесту `<activity>` і за допомогою прапорців в намірі, який передається в `startActivity()`.

У цьому сенсі головними атрибутами `<activity>`, які можна використовувати, є такі:

- `taskAffinity`;
- `launchMode`;

- `allowTaskReparenting`;
- `clearTaskOnLaunch`;
- `alwaysRetainTaskState`;
- `finishOnTaskLaunch`.

Головні прапорці намірів, які можна використовувати – це:

- `FLAG_ACTIVITY_NEW_TASK`;
- `FLAG_ACTIVITY_CLEAR_TOP`;
- `FLAG_ACTIVITY_SINGLE_TOP`.

У наступних розділах показано, як можна використовувати ці атрибути маніфесту і прапорці намірів для визначення зв'язку операцій з задачами і їх поведінки в стеку переходів назад.

Крім того, окремо обговорюються рекомендації про подання задач і операцій і управління ними на екрані огляду. Зазвичай слід дозволити системі визначити спосіб подання вашого завдання і операцій на екрані огляду. Вам не потрібно змінювати цю поведінку.

Увага! У більшості додатків не слід переривати поведінку операцій і задач за замовчуванням. Якщо ви виявили, що вашій операції необхідно змінити поведінку за замовчуванням, будьте уважні і протестуйте зручність роботи з операцією під час запуску і при зворотній навігації до неї з інших операцій і задач за допомогою кнопки *Назад*. Обов'язково протестуйте поведінку навігації, яка може суперечити поведінці, очікуваній користувачем.

2.3.3. Визначення режимів запуску

Режими запуску дозволяють вам визначати зв'язок нового екземпляру операції з поточним завданням. Можна задавати різні режими запуску двома способами, використовуючи:

1. *Файл маніфесту*. Коли ви оголошуєте операцію у файлі маніфесту, ви можете вказати, як операція повинна зв'язуватися з задачами при її запуску

2. *Прапорці намірів*. Коли ви викликаєте `startActivity()`, ви можете включити прапорець в `Intent`, який оголошує, як повинна бути пов'язана нова операція з поточною задачею (і чи повинна).

По суті, якщо операція А запускає операцію В, операція В може визначити у своєму маніфесті, як вона має бути пов'язана з поточною задачею (якщо взагалі має), а операція А може також запросити, як Операція В повинна бути пов'язана з поточною задачею. Якщо обидві операції визначають, як операція В має бути пов'язана з задачею, тоді запит операції А (як визначено в намірі) обробляється через запит операції В (як визначено в її маніфесті).

Примітка. Деякі режими запуску, що доступні для файлу маніфесту, недоступні у вигляді прапорців для намірів і, аналогічним чином, деякі режими запуску, що доступні у вигляді прапорців для намірів, не можуть бути визначені в маніфесті.

2.3.4. Використання файлу маніфесту

При оголошенні операції в вашому файлі маніфесту можна вказати, як операція повинна бути пов'язана із завданням за допомогою атрибута `launchMode` елемента `<activity>`.

Атрибут `launchMode` вказує інструкцію по запуску операції в задачі. Існує чотири різних режими запуску, які можна призначити атрибуту `launchMode`:

1. Режим `«standard»` (за замовчуванням). Система створює новий екземпляр операції в задачі, з якої вона була запущена, і направляє йому намір. Може бути створено кілька екземплярів операції, кожен з яких може належати різним задачам, і одна задача може містити кілька екземплярів.

2. Режим `«singleTop»`. Якщо екземпляр операції вже існує на вершині поточної задачі, система направляє намір в цей екземпляр шляхом виклику її методу `onNewIntent()`, а не шляхом створення нового екземпляра операції. Може бути створено кілька екземплярів операції, кожен з яких може належати різним задачам, і одна задача може містити кілька екземплярів (але тільки якщо операція на вершині стека переходів назад не є існуючим екземпляром операції).

Припустимо, що стек переходів назад задачі складається з кореневої операції A з операціями B, C і D на вершині (стек має вигляд A-B-C-D, і D знаходиться на вершині). Надходить намір для операції типу D. Якщо D має режим запуску `«standard»` за замовчуванням, запускається новий екземпляр класу і стек набирає вигляду A-B-C-D-D. Однак якщо D має режим запуску `«singleTop»`, існуючий екземпляр D отримує намір через `onNewIntent()`, тому що цей екземпляр знаходиться на вершині стека – стек зберігає вигляд A-B-C-D. Однак якщо надходить намір для операції типу B, тоді в стек додається новий екземпляр B, навіть якщо він має режим запуску `«singleTop»`.

Примітка. Коли створюється новий екземпляр операції, користувач може натиснути кнопку *Назад* для повернення до попередньої операції. Але коли існуючий екземпляр операції обробляє новий намір, користувач не може натиснути кнопку *Назад* для повернення до стану операції до надходження нового наміру в `onNewIntent()`.

3. Режим «`singleTask`». Система створює нову задачу і екземпляр операції в корені нової задачі. Однак якщо екземпляр операції вже існує в окремому завданні, система направляє намір в існуючий екземпляр шляхом виклику його методу `onNewIntent()`, а не шляхом створення нового екземпляра. Одночасно може існувати тільки один екземпляр операції.

Примітка. Хоча операція запускає нову задачу, кнопка *Назад* повертає користувача до попередньої операції.

4. Режим «`singleInstance`». Такий самий, як і «`singleTask`», але при цьому система не запускає ніяких інших операцій в задачі, що містить цей екземпляр. Операція завжди є єдиним членом своєї задачі; будь-які операції, запущені цією операцією, відкриваються в окремій задачі.

Як інший приклад: додаток Android Browser оголошує, що операція веб-браузера повинна завжди відкриватися в своїй власній меті – шляхом зазначення режиму запуску `singleTask` в елементі `<activity>`. Це означає, що якщо ваш додаток видає намір відкрити Android Browser, його операція не поміщається в ту ж задачу, що і ваш додаток. Замість цього для браузера запускається нова задача або, якщо браузер вже має задачу, що працює у фоновому режимі, ця задача перекладається на передній план для обробки нового наміру.

І при запуску операції в новій задачі, і при запуску операції в існуючій задачі кнопка *Назад* завжди повертає користувача до попередньої операції. Однак якщо запускається операція, яка вказує режим запуску `singleTask`, вся задача перекладається на передній план, якщо екземпляр цієї операції існує у фоновій задачі. У цей момент стек переходів назад поміщає всі операції із задачі, перекладеної на передній план, на вершину стека. Рис. 2.16 ілюструє сценарій цього типу.

Подання операції з режимом запуску `singleTask` додається в стек переходів назад. Якщо операція вже є частиною фоновієї задачі зі своїм власним стеком переходів назад, то весь стек переходів назад також переноситься вгору, на вершину поточної задачі.

Додаткову інформацію про використання режимів запуску у файлі маніфесту подано в документації елемента `<activity>`, де більш детально обговорено атрибут `launchMode` і прийняті значення.

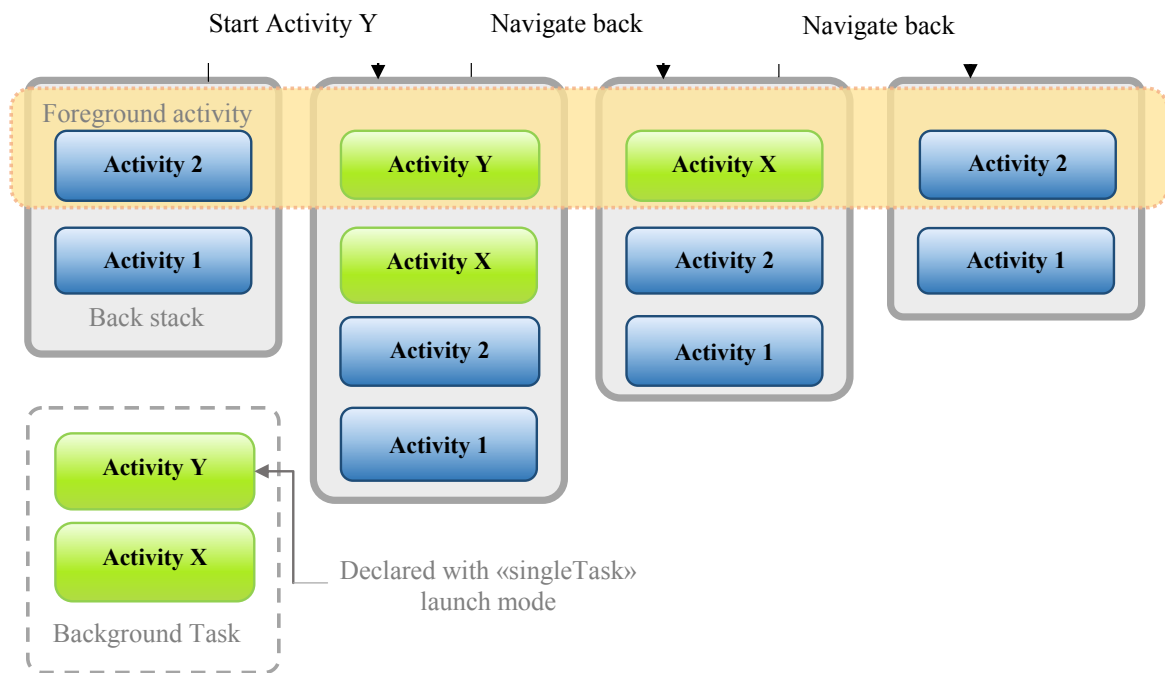


Рисунок 2.16 – Сценарій «singleInstance»

Примітка. Поведінка, яка задається для вашої операції за допомогою атрибута `launchMode`, може бути перевизначена прапорцями, включеними в намір, який запускає вашу операцію.

2.3.5 Використання прапорців намірів

При запуску операції можна змінити зв'язування операції з її завданням за замовчуванням шляхом включення прапорців в намір, який доставляється в `startActivity()`. Для зміни поведінки за замовчуванням можна використовувати такі прапорці:

- `FLAG_ACTIVITY_NEW_TASK`. Виконується запуск операції в новому завданні. Якщо завдання вже працює для операції, яку запускаєте зараз, це завдання перекладається на передній план, її останній стан відновлюється, і операція отримує новий намір в `onNewIntent()`. Це приводить до тієї ж поведінки, що і значення `launchMode` в режимі «singleTask»;

- `FLAG_ACTIVITY_SINGLE_TOP`. Якщо операція, що запускається, є поточною (знаходиться на вершині стека переходів назад), тоді виклик в `onNewIntent()` отримує існуючий екземпляр, без створення нового екземпляра операції. Це приводить до тієї ж поведінки, що і значення `launchMode` в режимі «singleTop»;

- `FLAG_ACTIVITY_CLEAR_TOP`. Якщо операція, що запускається, вже працює в поточній задачі, тоді замість запуску нового екземпляра цієї операції

знищуються всі інші операції, розташовані в стеку вище від неї, і цей намір доставляється у відновлений екземпляр цієї операції (яка тепер знаходиться на вершині стека) за допомогою `onNewIntent()`. Для формування такої поведінки не існує значення для атрибута `launchMode`. Прапорець `FLAG_ACTIVITY_CLEAR_TOP` найчастіше використовується спільно з прапорцем `FLAG_ACTIVITY_NEW_TASK`. При використанні разом ці прапорці дозволяють знайти існуючу операцію в іншій задачі і помістити її в становище, де вона зможе реагувати на намір.

Примітка. Якщо для призначеної операції встановлено режим запуску «*standard*», вона також видаляється зі стека і на її місці запускається новий екземпляр, щоб обробити вхідний намір. Саме тому в режимі запуску «*standard*» завжди створюється новий екземпляр для нового наміру.

Обробка прив'язок. Прив'язка вказує на переважну приналежність операції до задачі. За замовчуванням всі операції з однієї програми мають прив'язку одна до одної. Тому за замовчуванням всі операції однієї програми воліють перебувати в одному завданні. Однак можна змінити прив'язку за замовчуванням для операції. Операції, визначені в різних додатках, можуть спільно використовувати одну прив'язку; таким же чином операції, визначені в одному додатку, можуть отримати прив'язки до різних завдань.

Можна змінити прив'язку будь-якої операції за допомогою атрибута `taskAffinity` елемента `<activity>`.

Атрибут `taskAffinity` набуває рядкового значення, яке має відрізнятися від імені пакета за замовчуванням, оголошеного в елементі `<manifest>`, оскільки система використовує це ім'я для ідентифікації прив'язки задачі за замовчуванням для додатка.

Прив'язка вступає в гру в двох випадках:

1. Коли намір, що запускає операцію, містить прапорець `FLAG_ACTIVITY_NEW_TASK`.

Нова операція за замовчуванням запускається в задачі тієї операції, яка викликала `startActivity()`. Вона поміщається в той самий стек переходів назад, що і викликана операція. Однак якщо намір, переданий в `startActivity()`, містить прапорець `FLAG_ACTIVITY_NEW_TASK`, система шукає іншу задачу для розміщення нової операції. Часто це нова задача, але не обов'язково. Якщо вже існує задача з тією ж прив'язкою, що і у новій операції, операція запускається в цій задачі. Якщо її немає, операція починає нову задачу.

Якщо цей прапорець приводить до того, що операція починає нову задачу, і користувач натискає кнопку *Додому* для виходу з неї, повинен існувати спосіб, що дозволяє користувачеві повернутися до задачі. Деякі об'єкти (такі, як диспетчер повідомлень) завжди запускають операції в зовнішній задачі, а не в складі власної, тому вони завжди поміщають прапорець `FLAG_ACTIVITY_NEW_TASK` в наміри, які вони передають в `startActivity()`. Якщо у вас є операція, яку можна викликати зовнішнім об'єктом, що використовує цей прапорець, подбайте про те, щоб у користувача був незалежний спосіб повернутися в запущену задачу, наприклад, за допомогою значка запуску (коренева операція задачі містить фільтр намірів `CATEGORY_LAUNCHER`. Якщо для атрибута `allowTaskReparenting` операції встановлено значення «true», у цьому випадку операція може переміститися із задачі, що її викликала, в задачу, до якої у операції є прив'язка, коли ця задача переходить на передній план.

Припустимо, що операція, яка повідомляє про погодні умови в обраних містах, визначена в складі додатка для мандрівників. Вона має ту ж прив'язку, що й інші операції в тому ж додатку (прив'язка зі стандартними програмами), і допускає перепідпорядкування з цим атрибутом. Коли одна з ваших операцій запускає операцію прогнозу погоди, вона спочатку належить тій же задачі, що і ваша операція. Однак коли задача з програми для мандрівників переходить на передній план, операція прогнозу погоди перепризначається цій задачі і відображається всередині неї.

Порада. Якщо файл `.apk` містить більше одного «дodatка» з точки зору користувача, ймовірно, ви захочете використовувати атрибут `taskAffinity` для призначення різних прив'язок операціями, пов'язаними з кожним «додатком».

Очищення стека переходів назад. Якщо користувач виходить із задачі на тривалий час, система видаляє з неї всі операції, крім кореневої операції. Коли користувач повертається в задачу, відновлюється тільки коренева операція. Система поводить таким чином, тому що після тривалого часу користувачі зазвичай вже закинули те, чим вони займалися раніше, і повертаються в задачу, щоб почати щось нове.

Для зміни такої поведінки передбачено кілька атрибутів операції, якими можна скористатися:

1) `alwaysRetainTaskState`. Якщо для цього атрибута встановлено значення «true» в кореневій операції задачі, то описана вище поведінка за за-

мовчуванням не відбувається. Задача відновлює всі операції в своєму стеку навіть після закінчення тривалого періоду часу.

2) `clearTaskOnLaunch`. Якщо для цього атрибута встановлено значення «`true`» в кореневій операції задачі, стек очищається до кореневої операції кожен раз, коли користувач виходить із задачі і повертається в неї. Іншими словами, цей атрибут протилежний атрибуту `alwaysRetainTaskState`. Користувач завжди повертається в задачу в її початковому стані, навіть після короткочасного виходу з неї.

3) `finishOnTaskLaunch`. Цей атрибут схожий на `clearTaskOnLaunch`, але він діє на одну операцію, а не на всю задачу. Він також може приводити до видалення будь-якої операції, включаючи кореневу операцію. Коли для нього встановлено значення «`true`», операція залишається частиною задачі тільки для поточного сеансу. Якщо користувач виходить із задачі, а потім повертається в неї, операція вже відсутня.

2.3.6. Запуск задачі

Можна зробити операцію точкою входу, призначаючи їй фільтр намірів зі значенням `"android.intent.action.MAIN"` як зазначену дію і `"android.intent.category.LAUNCHER"` як зазначену категорію. Наприклад:

```
<activity ... >
  <intent-filter ... >
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
  ...
</activity>
```

Фільтр намірів такого типу приводить до того, що в засобі запуску програми відображаються значок і мітка для операції, що дозволяє користувачеві запускати операцію і повертатися в задачу, яка її створила в будь-який момент після її запуску.

Друга можливість дуже важлива: вона дозволяє користувачам виходити із задачі і потім повертатися в неї за допомогою цього засобу запуску операції. Тому два режими запуску, які відзначають, що операції завжди ініціюють задачу, `"singleTask"` і `"singleInstance"`, повинні використовуватися тільки в тих випадках, коли операція містить `ACTION_MAIN` та фільтр `CATEGORY_LAUNCHER`. Уявіть, наприклад, що може статися, якщо фільтр відсутній: намір запускає операцію `"singleTask"`, яка ініціює нову задачу, і користувач деякий час працює в цьому завданні. Потім користувач натискає кнопку

Додому. Задача переводиться у фоновий режим і не відображається. Тепер у користувача немає можливості повернутися до задачі, оскільки вона відсутня в засобі запуску додатка.

Для випадків, коли необхідно, щоб користувач міг повернутися до операції, встановіть для атрибута `finishOnTaskLaunch` елемента `<activity>` значення `"true"`.

2.3.7. Екран огляду (*Overview screen*)

Екран огляду (також використовуються назви: екран останніх задач, список останніх задач або найновіші програми) є елементом користувацького інтерфейсу системного рівня, в якому міститься список останніх операцій і задач. Користувач може переміщатися по списку і вибирати задачі для відновлення, або жестом видаляти задачу зі списку. У версії Android 5.0 (рівень API 21) кілька екземплярів однієї операції, що містять різні документи, можуть відображатися у вигляді задач на екрані огляду. Наприклад, Google Диск може мати задачу для кожного з декількох документів Google. У вікні кожен документ відображається у вигляді задачі (рис. 2.17).

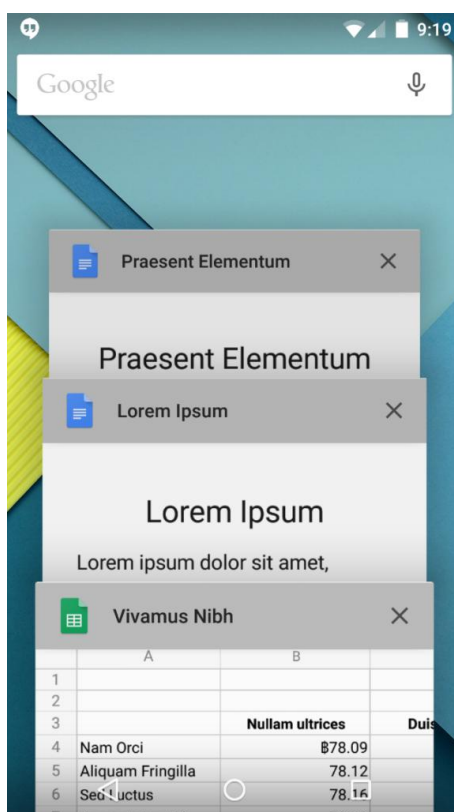


Рисунок 2.17 – Екран огляду, на якому показані три документи Google Диска, подані у вигляді окремих задач

Зазвичай слід дозволити системі визначити спосіб подання ваших задач і операцій на екрані огляду. Вам не потрібно змінювати цю поведінку. Однак додаток може визначати спосіб і час появи операції на екрані огляду. За допомогою класу `ActivityManager.AppTask` можна управляти задачами, а за допомогою прапорців операції класу `Intent` вказувати, коли операція додається на екран огляду або віддаляється з нього. Крім того, атрибути `<activity>` дозволяють встановлювати поведінку в маніфесті.

2.3.8. Додавання задач на екран огляду

Використання прапорців класу `Intent` для додавання задачі забезпечує краще управління часом і способом відновлення або повторного відновлення документа на екрані огляду. За допомогою атрибутів `<activity>` можна вибрати відкриття документа в новому завданні або повторне використання існуючої задачі для документа.

Примітка. Прапорець `FLAG_ACTIVITY_NEW_DOCUMENT` заміщає прапорець `FLAG_ACTIVITY_CLEAR_WHEN_TASK_RESET`, який є застарілим для систем Android 5.0 і вище (рівень API 21).

Після встановлення прапорця `FLAG_ACTIVITY_MULTIPLE_TASK` при створенні нового документа, система завжди створює нову задачу з цільовою операцією в якості кореня. Цей параметр дозволяє відкривати один документ в декількох задачах. Наступний код показує, як це робить основна операція:

```
DocumentCentricActivity.java

public void createNewDocument(View view) {
    final Intent newDocumentIntent = newDocumentIntent();
    if (useMultipleTasks) {
        newDocumentIntent.addFlags(Intent.FLAG_ACTIVITY_MULTIPLE_TASK);
    }
    startActivity(newDocumentIntent);
}

private Intent newDocumentIntent() {
    boolean useMultipleTasks = mCheckbox.isChecked();
    final Intent newDocumentIntent = new Intent(this,
NewDocumentActivity.class);
    newDocumentIntent.addFlags(Intent.FLAG_ACTIVITY_NEW_DOCUMENT);
    newDocumentIntent.putExtra(KEY_EXTRA_NEW_DOCUMENT_COUNTER,
incrementAndGet());
    return newDocumentIntent;
}

private static int incrementAndGet() {
    Log.d(TAG, "incrementAndGet(): " + mDocumentCounter);
    return mDocumentCounter++;
}
```

Примітка. Операції, запущені з прапорцем `FLAG_ACTIVITY_NEW_DOCUMENT`, повинні мати значення атрибута `android:launchMode="standard"` (за замовчуванням), встановлене в маніфесті.

Коли основна операція запускає нову операцію, система шукає в існуючих задачах одну, значення `intent` якої відповідає імені компонента і даним `Intent` для операції. Якщо задача не знайдена або `intent` містить прапор `FLAG_ACTIVITY_MULTIPLE_TASK`, створюється нова задача з операцією в якості кореня. Якщо задача знайдена, система виводить цю задачу на передній план і передає нове значення `intent` в `onNewIntent()`. Нова операція отримує `intent` і створює новий документ на екрані огляду, як у наступному прикладі:

```
NewDocumentActivity.java

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_new_document);
    mDocumentCount = getIntent()

    .getIntentExtra(DocumentCentricActivity.KEY_EXTRA_NEW_DOCUMENT_COUNTER,
0);
    mDocumentCounterTextView = (TextView) findViewById(
        R.id.hello_new_document_text_view);
    setDocumentCounterText(R.string.hello_new_document_counter);
}

@Override
protected void onNewIntent(Intent intent) {
    super.onNewIntent(intent);
    /* If FLAG_ACTIVITY_MULTIPLE_TASK has not been used, this activ-
ity
is reused to create a new document.
*/
    setDocumentCounterText(R.string.reusing_document_counter);
}
```

Використання атрибута «Операція» для додавання задачі. У маніфесті операції можна також вказати, що операція завжди запускається в новій задачі. Для цього використовується атрибут `<activity>`, `android:documentLaunchMode`. Цей атрибут має чотири значення, які працюють таким чином, коли користувач відкриває документ в додатку:

1) `<intoExisting>` – операція повторно використовує існуючу задачу для документа. Це рівнозначне установленню прапорця `FLAG_ACTIVITY_NEW_DOCUMENT` без установлення прапорця

FLAG_ACTIVITY_MULTIPLE_TASK.

2) «always» – операція створює нову задачу для документа, навіть якщо документ вже відкрито. Використання цього значення рівносильне установленню обох прапорців FLAG_ACTIVITY_NEW_DOCUMENT й FLAG_ACTIVITY_MULTIPLE_TASK.

3) «none» – операція не створює нову задачу для документа. Екран огляду обробляє операцію як операцію за замовчуванням: на екрані огляду відображається одна задача для програми, яка відновлюється з будь-якої останньої операції, викликаной користувачем.

4) «never» – операція не створює нову задачу для документа. Установлення цього значення перевизначає поведження прапорців FLAG_ACTIVITY_NEW_DOCUMENT й FLAG_ACTIVITY_MULTIPLE_TASK, якщо обидва вони встановлені в intent, і на екрані огляду відображається одна задача для програми, яка відновлюється з будь-якої останньої операції, викликаной користувачем.

Примітка. Для всіх значень, крім none та never, операція повинна бути визначена з атрибутом `launchMode="standard"`. Якщо цей атрибут не вказаний, використовується `documentLaunchMode="none"`.

2.3.9. Видалення задач

За замовчуванням задача документа автоматично видаляється з екрана огляду після завершення відповідної операції. Можна перевизначити цю поведінку за допомогою класу `ActivityManager.AppTask`, із прапорцем `Intent` або атрибутом `<activity>`.

Можна в будь-який момент повністю прибрати задачу з екрана огляду, встановивши для атрибута `<activity>` `android:excludeFromRecents` значення `true`.

Можна встановити максимальне число задач, яке ваш додаток може включити в екран огляду, встановивши для атрибута `<activity>` `android:maxRecents` ціле значення. Значення за замовчуванням дорівнює 16. При досягненні максимальної кількості задач найбільш довго не використовувана задача видаляється з екрана огляду. Максимальне значення `android:maxRecents` становить 50 (25 – для пристроїв з малим обсягом пам'яті); Значення, що менші 1 не допускаються.

Використання класу `AppTask` для видалення задач. В операції, яка створює нове завдання на вікні, можна вказати час видалення задачі і завершення всіх пов'язаних з нею операцій, викликавши метод `finishAndRemoveTask()`:

```
public void onRemoveFromRecents(View view) {  
    // The document is no longer needed; remove its task.  
    finishAndRemoveTask();  
}
```

Примітка. Використання методу `finishAndRemoveTask()`, який перекриває використання тега `FLAG_ACTIVITY_RETAIN_IN_RECENTS`, розглянуто нижче.

Збереження завершених задач. Щоб зберегти задачу на вікні, навіть якщо її операція завершена, передайте прапорець `FLAG_ACTIVITY_RETAIN_IN_RECENTS` в метод `addFlags()` об'єкта `Intent`, який запускає операцію:

```
private Intent newDocumentIntent() {  
    final Intent newDocumentIntent = new Intent(this,  
        NewDocumentActivity.class);  
    newDocumentIntent.addFlags(Intent.FLAG_ACTIVITY_NEW_DOCUMENT |  
        android.content.Intent.FLAG_ACTIVITY_RETAIN_IN_RECENTS);  
    newDocumentIntent.putExtra(KEY_EXTRA_NEW_DOCUMENT_COUNTER,  
        incrementAndGet());  
    return newDocumentIntent;  
}
```

Для досягнення того ж результату встановіть для атрибута `<activity>` `android:autoRemoveFromRecents` значення `false`. Значеннями за замовчуванням `true` (для операцій документа) і `false` (для звичайних операцій). Використання цього атрибута перевизначає прапорець `FLAG_ACTIVITY_RETAIN_IN_RECENTS`, описаний вище.

2.4. Об'єкти `Intent` та фільтри об'єктів `Intent`

`Intent` є об'єктом обміну повідомленнями, за допомогою якого можна запросити виконання дії у компонента іншої програми. Не зважаючи на те, що об'єкти `Intent` спрощують обмін даними між компонентами у декількох аспектах, в основному вони використовуються в трьох ситуаціях:

1. Для запуску операції. Компонент `Activity` є одним екраном у додатку. Для запуску нового екземпляра компонента `Activity` необхідно передати об'єкт `Intent` методу `startActivity()`. Об'єкт `Intent` описує операцію, яку потрібно запустити, а також містить всі інші необхідні дані.

Якщо після завершення операції від неї вимагається отримати результат, викличте метод `startActivityForResult()`. Ваша операція отримає результат у вигляді окремого об'єкта `Intent` в зворотному виклику методу `onActivityResult()` операції. Докладну інформацію див. в інструкції *Операції*.

2. Для запуску служби. `Service` є компонентом, який виконує дії у фоновому режимі без інтерфейсу користувача. Службу можна запустити для виконання одноразової дії (наприклад, щоб завантажити файл), передавши об'єкт `Intent` методу `startService()`. Об'єкт `Intent` описує службу, яку потрібно запустити, а також містить всі інші необхідні дані.

Якщо служба сконструйована з інтерфейсом клієнт-сервер, до неї можна встановити прив'язку з іншого компонента, передавши об'єкт `Intent` методу `bindService()`.

3. Для розсилання широкомовних повідомлень. Широкомовне повідомлення – це повідомлення, яке може прийняти будь-який додаток. Система видає різні широкомовні повідомлення про системні події, наприклад, коли система завантажується або пристрій починає заряджатися. Для видачі широкомовних повідомлень іншим додаткам необхідно передати об'єкт `Intent` методу `sendBroadcast()`, `sendOrderedBroadcast()` або `sendStickyBroadcast()`.

2.4.1. *Tunu об'єкти Intent*

Є два типи об'єктів `Intent`: явні і неявні.

Явні об'єкти Intent вказують компонент, який потрібно запустити, за ім'ям (повне ім'я класу). Явні об'єкти `Intent` зазвичай використовуються для запуску компонента з вашого власного додатка, оскільки вам відоме ім'я класу операції або служби, яку необхідно запустити. Наприклад, можна запустити нову операцію у відповідь на дію користувача або запустити службу, щоб завантажити файл у фоновому режимі.

Неявні об'єкти Intent не містять імені конкретного компонента. Замість цього вони в цілому оголошують дію, яку потрібно виконати, що дає можливість компоненту з іншої програми обробити цей запит. Наприклад, якщо потрібно показати користувачеві місце на карті, то за допомогою неявного об'єкта `Intent` можна запросити, щоб це зробив інший додаток, в якому така функція передбачена.

1. Коли створено явний об'єкт `Intent` для запуску операції або служби, система негайно запускає компонент додатка, вказаний в об'єкті `Intent`.

2. Операція А створює об'єкт **Intent** з описом дії і передає його до методу **startActivity()**.

3. Система Android шукає у всіх додатках фільтри **Intent**, які відповідають даному об'єкту **Intent**.

4. Коли додаток з відповідним фільтром знайдено, система запускає відповідну операцію (операція В), викликавши її метод **onCreate()** і передавши йому об'єкт **Intent** (рис. 2.18).

5. Коли створено неявний об'єкт **Intent**, система Android знаходить відповідний компонент шляхом порівняння вмісту об'єкта **Intent** з фільтрами **Intent**, оголошеними у файлах маніфесту інших додатків, наявних на пристрої. Якщо об'єкт **Intent** збігається з фільтром **Intent**, система запускає цей компонент і передає йому об'єкт **Intent**. Якщо відповідними виявляються кілька фільтрів **Intent**, система виводить діалогове вікно, де користувач може вибрати програму для додавання коментарів.

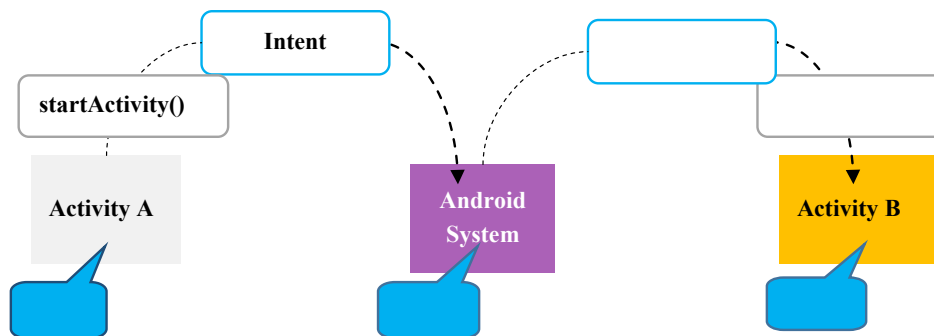


Рисунок 2.18 – Процес передачі неявного об'єкта **Intent** системою для запуску іншої операції

Фільтр **Intent** є виразом у файлі маніфесту програми, що вказує типи об'єктів **Intent**, які міг би приймати компонент. Наприклад, оголошення фільтра **Intent** для операції, надає іншим програмам можливість безпосередньо запускати вашу операцію за допомогою деякого об'єкта **Intent**. Так само, якщо ви не оголосите будь-які фільтри **Intent** для операції, то її можна буде запустити тільки за допомогою явного об'єкта **Intent**.

Увага! З метою забезпечення безпеки програми завжди використовуйте явний об'єкт **Intent** при запуску **Service** і не оголошуйте фільтри **Intent** для своїх служб. Запуск служб за допомогою неявних об'єктів **Intent** є ризикованим з точки зору безпеки, оскільки не можна бути абсолютно впевненим, яка служба відреагує на такий об'єкт **Intent**, а користувач не зможе бачити, яка

служба запускається. Починаючи з Android 5.0 (рівень API 21), система викликає виключення при виклику методу `bindService()` за допомогою неявного об'єкта `Intent`.

2.4.2. Створення об'єкта *Intent*

Об'єкт `Intent` містить інформацію, на підставі якої система Android визначає, який компонент потрібно запустити (наприклад, точне ім'я компонента або категорію компонентів, які повинні отримати цей об'єкт `Intent`), а також відомості, які необхідні компоненту-одержувачу, щоб належним чином виконати дію (а саме, виконувану дію і дані, з якими його потрібно виконати).

Основні відомості, що містяться в об'єкті `Intent`:

1. Ім'я компонента. Ім'я компонента, який потрібно запустити.

Ця інформація є необов'язковою, але саме вона і робить об'єкт `Intent` явним. Її наявність означає, що об'єкт `Intent` слід передати тільки компоненту програми, визначеному за ім'ям. За відсутності імені компонента об'єкт `Intent` є неявним, а система визначає, який компонент отримає об'єкт `Intent` за іншими відомостями, які в ньому містяться. Тому, якщо вам потрібно запустити певний компонент зі свого програмного додатка, слід вказати його ім'я.

Примітка. При запуску `Service` слід завжди вказувати ім'я компонента. В іншому випадку розробник не зможе бути абсолютно впевненим у тому, яка служба відреагує на об'єкт `Intent`, а користувач не зможе бачити, яка служба запускається.

Це поле об'єкта `Intent` є об'єктом `ComponentName`, який можна вказати за допомогою повного імені класу цільового компонента, включаючи ім'я пакета програми. Наприклад, `com.example.ExampleActivity`. Задати ім'я компонента можна за допомогою методу `setComponent()`, `setClass()`, `setClassName()` або конструктору `Intent`.

2. Дія. Рядок, що визначає стандартну дію, яку потрібно виконати (наприклад, `view` (перегляд) або `pick` (вибір)).

При видачі об'єктів `Intent` з широкомовними повідомленнями – це дія, яка відбулася і про яку повідомляється. Дія значною мірою визначає, яким чином структурована решта об'єкта `Intent`, – зокрема, що саме міститься в розділі даних і додаткових даних.

Для користування об'єктами `Intent` в межах свого застосування (або для використання іншими додатками, щоб викликати компоненти з вашого додатка) можна вказати власні дії. Зазвичай же слід використовувати константи дій, ви-

значені класом `Intent` або іншими класами платформи. Ось кілька стандартних дій для запуску операції:

1) `ACTION_VIEW`. Використовуйте цю дію в об'єкті `Intent` з методом `startActivity()`, коли є певна інформація, яку операція може показати користувачеві, наприклад, фотографія в додатку галереї або адреса для перегляду в картографічному додатку.

2) `ACTION_SEND`. Його ще називають об'єктом `Intent` «share» (намір надати загальний доступ). Цю дію слід використовувати в об'єкті `Intent` з методом `startActivity()`, за наявності певних даних, доступ до яких користувач може надати через інший додаток, наприклад додаток для роботи з електронною поштою або соціальними мережами.

Інші константи, що визначають стандартні дії, див. у довіднику по класу `Intent`. Інші дії визначаються в інших частинах платформи Android. Наприклад, в `Settings` визначаються дії, що відкривають ряд екранів програми для налаштування системи.

Дію можна вказати для об'єкта `Intent` з методом `setAction()` або конструктором `Intent`.

При визначенні власних дій, обов'язково використовуйте як їх префікс ім'я пакета вашого додатка. Наприклад:

```
static final String ACTION_TIMETRAVEL =  
"com.example.action.TIMETRAVEL";
```

3. *Дані*. `URI` (об'єкт `Uri`), який посилається на дані, з якими буде виконуватися дія і/або тип `MIME` цих даних. Тип даних зазвичай визначається дією об'єкта `Intent`. Наприклад, якщо дією є `ACTION_EDIT`, в даних повинен міститися `URI` документа, який потрібно відредагувати.

При створенні об'єкта `Intent`, крім `URI`, часто необхідно вказати тип даних (їх тип `MIME`). Наприклад, операція, яка може виводити на екран зображення, швидше за все, не зможе відтворити аудіофайл, навіть якщо і у тих, і у інших даних будуть однакові формати `URI`. Тому зазначення типу `MIME` даних допомагає системі Android знайти найбільш підходящий компонент для отримання вашого об'єкта `Intent`. Однак тип `MIME` іноді можна успадкувати від `URI` — зокрема, коли дані являють собою `content: URI`, який вказує, що дані знаходяться на пристрої і ними керує `ContentProvider`, а це дає можливість системі бачити тип `MIME` даних.

Щоб задати тільки URI даних, викличте `setData()`. Щоб задати тільки тип MIME, викличте `setType()`. За необхідності обидва ці параметри можна в явному вигляді задати за допомогою `setDataAndType()`.

Увага! Якщо потрібно задати і URI, і тип MIME, не викликайте `setData()` і `setType()`, оскільки кожен з цих методів анулює результат виконання іншого. Щоб задати URI і тип MIME завжди використовуйте метод `setDataAndType()`.

4. Категорія. Рядок, що містить інші відомості про те, яким компонентом повинна виконуватися обробка цього об'єкта `Intent`. В об'єкт `Intent` можна помістити будь-яку кількість описів категорій, проте більшості об'єктів `Intent` категорія не потрібна. Наведемо деякі стандартні категорії:

`CATEGORY_BROWSABLE`

Цільова операція дозволяє запускати себе веб-браузером для відображення даних, зазначених за посиланням – наприклад, зображення або повідомлення електронної пошти.

`CATEGORY_LAUNCHER`

Ця операція є початковою операцією завдання, вона вказана в засобі запуску додатків системи.

Повний список категорій див. в описі класу `Intent`.

Вказати категорію можна за допомогою `addCategory()`.

Наведені вище властивості (ім'я компонента, дія, дані і категорія) являють собою характеристики, що визначають об'єкт `Intent`. На підставі цих властивостей система Android може вирішити, який компонент слід запустити.

Однак в об'єкті `Intent` може бути наведена й інша інформація, яка не впливає на те, яким чином визначається необхідний компонент програми.

5. Додаткові дані. Пари «ключ-значення», що містять іншу інформацію, яка необхідна для виконання необхідної дії. Так само, як деякі дії використовують певні види URI даних, деякі дії використовують певні додаткові дані.

Додавати додаткові дані можна за допомогою різних методів `putExtra()`, кожен з яких приймає два параметри: ім'я та значення ключа. Також можна створити об'єкт `Bundle` з усіма додатковими даними, а потім вставити об'єкт `Bundle` в об'єкт `Intent` за допомогою методу `putExtras()`.

Наприклад, при створенні об'єкта `Intent` для надсилання листів з методом `ACTION_SEND` можна вказати одержувача за допомогою ключа `EXTRA_EMAIL`, а тему повідомлення – за допомогою ключа `EXTRA_SUBJECT`.

Клас `Intent` вказує багато констант `EXTRA_*` для стандартних типів даних. Якщо вам потрібно оголосити власні додаткові ключі (для об'єктів `Intent`, які приймає ваш додаток), обов'язково вказуйте як префікс імені пакета свого додатка. Наприклад:

```
static final String EXTRA_GIGAWATTS = "com.example.EXTRA_GIGAWATTS";
```

6. Мітки. Мітки, визначені в класі `Intent`, які діють як метадані для об'єкта `Intent`. Мітки повинні вказувати системі Android, яким чином слід запускати операцію (наприклад, до якої задачі повинна належати операція) і як з нею поводитися після запуску (наприклад, чи буде вона вказана в списку останніх операцій).

Докладну інформацію див. в документі, присвяченому методу `setFlags()`.

Приклад явного об'єкта Intent. Явні об'єкти `Intent` використовуються для запуску конкретних компонентів додатка, наприклад певної операції або служби. Щоб створити явний об'єкт `Intent`, задайте ім'я компонента для об'єкта `Intent` – всі інші властивості об'єкта `Intent` можна не ставити.

Наприклад, якщо в своєму додатку створили службу з ім'ям `DownloadService`, призначену для завантаження файлів з Інтернету, то для її запуску можна використовувати наступний код:

```
//Виконується у Activity, таким чином 'this' - це Context
// fileName це значення виду "http://www.example.com/image.png"
Intent downloadIntent = new Intent(this, DownloadService.class);
downloadIntent.setData(Uri.parse(fileName));
startService(downloadIntent);
```

Приклад неявного об'єкту Intent. Неявний об'єкт `Intent` вказує дію, якою може бути викликаний будь-який наявний на пристрої додаток, що здатний виконати цю дію. Неявні об'єкти `Intent` використовуються, коли ваш додаток не може виконати ту чи іншу дію, а інші додатки, швидше за все, можуть, тобто сприяють тому, щоб користувач мав можливість вибрати, яку програму використовувати для цього.

Наприклад, якщо у вас є контент і ви хочете, щоб користувач поділився ним з іншими людьми, створіть об'єкт `Intent` з дією `ACTION_SEND` і додайте додаткові дані, які вказують на контент, загальний доступ до якого слід надати. Коли за допомогою об'єкта `Intent` ви викликаєте `callback` методу `startActivity()`, користувач зможе вибрати програму, за допомогою якої до контенту буде надано загальний доступ.

Увага! Можлива ситуація, коли на пристрої користувача не буде ніякого додатка, який може відгукнутися на неявний об'єкт `Intent`, що відправлений вами методом `startActivity()`. В цьому випадку виклик закінчиться невдачею, а робота програми аварійно завершиться. Щоб перевірити, чи буде отриманий операцією об'єкт `Intent`, викличте метод `resolveActivity()` для свого об'єкта `Intent`. Якщо результатом буде значення, відмінне від `null`, це означає, що є хоча б один додаток, який здатен відгукнутися на об'єкт `Intent`, і можна викликати `startActivity()`. Якщо ж результатом буде значення `null`, об'єкт `Intent` не слід використовувати і за можливості слід відключити функцію, яка видає цей об'єкт:

```
// Створюємо текстове повідомлення
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, textMessage);
sendIntent.setType("text/plain");

// Переконайтеся в тому, що Intent буде містити Activity
if (sendIntent.resolveActivity(getPackageManager()) != null) {
    startActivity(sendIntent);
}
```

Примітка. В цьому випадку URI не використовується, а натомість слід оголосити тип даних об'єкта `Intent`, щоб вказати контент, що міститься в додаткових даних.

При виклику методу `startActivity()` система аналізує всі встановлені додатки, щоб визначити, які з них можуть відгукнутися на об'єкт `Intent` цього виду (об'єкт `Intent` з дією `ACTION_SEND` і даними «`text/plain`»). Якщо є тільки одий підходящий додаток, він буде одразу ж відкритим і отримає даний об'єкт `Intent`. Якщо об'єкт `Intent` приймають кілька операцій, система відображає діалогове вікно, в якому користувач може вибрати додаток для виконання даної дії.

Примусове виконання блоку вибору додатка. За наявності декількох додатків, що реагують на ваш неявний об'єкт `Intent`, користувач може вибрати потрібну програму і вказати, що вона буде за замовчуванням виконувати цю дію. Це зручно в разі дії, для виконання якої користувач хоче завжди використовувати той самий додаток, наприклад, при відкритті веб-сторінки (користувачі зазвичай використовують той самий браузер). Діалогове вікно вибору додатка зображено на рис. 2.19.

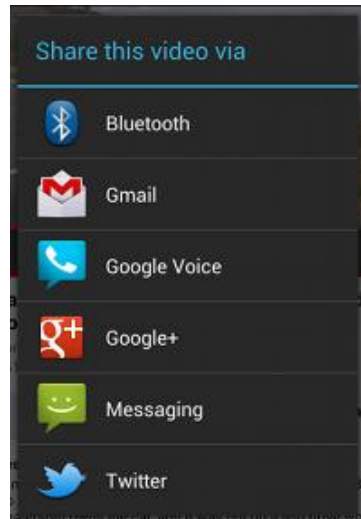


Рисунок 2.19 – Діалогове вікно вибору

Однак якщо на об'єкт `Intent` відгукуються кілька додатків, можливо, користувач вважатиме за краще кожен раз використовувати іншу програму, тому слід явно виводити діалогове вікно вибору. У діалоговому вікні для вибору додатка користувачеві пропонується вибрати програму і при кожному її запуску визначати, який додаток використовувати для дії (користувач не може вибрати програму, що використовується за замовчуванням). Наприклад, коли ваш додаток виконує операцію «share» (поділитися) за допомогою дії `ACTION_SEND`, користувачі можуть, залежно від ситуації, віддати перевагу тому, щоб кожен раз робити це за допомогою різних додатків, тому слід завжди використовувати діалогове вікно вибору.

Щоб вивести на екран блок і вибрати програму, створіть `Intent` за допомогою `createChooser()` і передайте його до `startActivity()`. Наприклад:

```
Intent sendIntent = new Intent(Intent.ACTION_SEND);
...

// Завжди використовуйте ресурси типу string для тексту користувацького інтерфейсу.
String title = getResources().getString(R.string.chooser_title);
// Create intent to show the chooser dialog
Intent chooser = Intent.createChooser(sendIntent, title);

// Переконайтеся в тому, що Intent буде містити хоча б один Activity
if (sendIntent.resolveActivity(getPackageManager()) != null) {
    startActivity(chooser);
}
```

У результаті на екран буде виведено діалогове вікно зі списком додатків, які можуть відреагувати на об'єкт `Intent`, переданий методу `createChooser()`, і використовують вказаний текст як заголовок діалогу.

2.4.3. Отримання неявного об'єкта `Intent`

Щоб вказати, які неявні об'єкти `Intent` може приймати ваш додаток, задайте один або кілька фільтрів `Intent` для кожного компонента програми за допомогою елемента `intent-filter` у файлі маніфесту. Кожен фільтр `Intent` вказує тип об'єктів `Intent`, які приймає компонент на підставі дії, даних і категорії, заданих в об'єкті `Intent`. Система передасть неявний об'єкт `Intent` вашому додатку, тільки якщо він може пройти через один з ваших фільтрів `Intent`.

Примітка. Явний об'єкт `Intent` завжди доставляється його цільовому компоненту, без урахування будь-яких фільтрів `Intent`, що оголошені компонентом.

Компонент додатка повинен оголошувати окремі фільтри для кожної унікальної роботи, яку він може виконати. Наприклад, у операції з програми для роботи з галереєю зображень може бути два фільтри: один фільтр – для перегляду зображення, а другий – для його редагування. Коли операція запускається, вона аналізує об'єкт `Intent` і вибирає режим своєї роботи на підставі інформації, наведеної в `Intent` (наприклад, чи показувати елементи управління редактора, чи ні).

Кожен фільтр `Intent` визначається елементом `intent-filter` у файлі маніфесту додатка, зазначеному в оголошенні відповідного компонента програми (наприклад, в елементі `activity`). У середині елемента `intent-filter` можна вказати тип об'єктів `Intent`, які будуть прийматися, за допомогою одного або декількох з наступних трьох елементів:

1. **Action** – оголошує дію, що приймається та, в свою чергу, задана в об'єкті `Intent`, в атрибуті `name`. Значення має бути рядком дії, а не константою класу.

2. **Data** – оголошує тип прийнятих даних, для чого використовується один або кілька атрибутів, що вказують різні складові частини URI даних (`scheme`, `host`, `port`, `path` і т. д.) і тип MIME.

3. **Category** – оголошує прийняту категорію, задану в об'єкті `Intent`, в атрибуті `name`. Значення має бути рядком дії, а не константою класу.

Примітка. Для отримання неявних об'єктів `Intent` необхідно включити категорію `CATEGORY_DEFAULT` у фільтр `Intent`. Методи `startActivity()` і

`startActivityForResult()` обробляють всі об'єкти `Intent` так, нібито вони оголошували категорію `CATEGORY_DEFAULT`. Якщо не оголосити цю категорію в своєму фільтрі `Intent`, ніякі неявні об'єкти `Intent` не будуть передані в вашу операцію.

У наступному прикладі оголошена операція з фільтром `Intent`, що визначає отримання об'єкта `Intent ACTION_SEND`, коли дані відносяться до типу `text`:

```
<activity android:name="ShareActivity">
  <intent-filter>
    <action android:name="android.intent.action.SEND"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="text/plain"/>
  </intent-filter>
</activity>
```

Можна створювати фільтри, в яких буде кілька екземплярів `action`, `data` або `category`. У цьому випадку просто потрібно переконатися в тому, що компонент може впоратися з будь-якими поєднаннями цих елементів фільтра.

Коли необхідно обробляти об'єкти `Intent` декількох видів, але тільки в певних поєднаннях дії, типу даних і категорії, необхідно створити кілька фільтрів `Intent`.

Обмеження доступу до компонентів. Використання фільтра `Intent` не є безпечним способом запобігання запуску ваших компонентів іншими додатками. Незважаючи на те, що після застосування фільтрів `Intent` компонент буде реагувати тільки на неявні об'єкти `Intent` певного виду, інший додаток теоретично може запустити компонент вашого застосування за допомогою явного об'єкта `Intent`, якщо розробник визначить імена ваших компонентів. Якщо важливо, щоб тільки ваш додаток міг запускати один з ваших компонентів, задайте для атрибута `exported` цього компонента значення «false».

Неявний об'єкт `Intent` перевіряється фільтром шляхом порівняння об'єкта `Intent` з кожним з наведених вище елементів. Щоб об'єкт `Intent` був доставлений компоненту, він повинен пройти всі три тести. Якщо він не буде відповідати хоча б одному з них, система Android не доставить об'єкт `Intent` компоненту. Однак оскільки у компонента може бути кілька фільтрів `Intent`, об'єкт `Intent`, який не проходить через один з фільтрів компонента, може пройти через інший фільтр.

Увага! Щоб випадково не запустити `Service` іншої програми, завжди використовуйте явні об'єкти `Intent` для запуску власних служб і не оголошуйте для них фільтри `Intent`.

Примітка. Фільтри `Intent` необхідно оголошувати в файлі маніфесту для всіх операцій. Фільтри для приймачів ширококомовних повідомлень можна реєструвати динамічно шляхом виклику `registerReceiver()`. Після цього реєстрацію приймача ширококомовних повідомлень можна скасувати за допомогою `unregisterReceiver()`. В результаті ваш додаток зможе сприймати певні оголошення тільки протягом зазначеного періоду часу в процесі роботи програми.

Приклади фільтрів. Щоб краще зрозуміти різні режими роботи фільтрів `Intent`, розгляньте наступний фрагмент з файлу маніфесту додатка для роботи з соціальними мережами:

```
<activity android:name="MainActivity">
    <!-- Цей activity є основним входом у програму -->
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
<activity android:name="ShareActivity">
    <!-- Цей activity обробляє "SEND" дії з текстовими даними-->
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="text/plain"/>
    </intent-filter>
    <!-- Цей activity також обробляє "SEND" та "SEND_MULTIPLE" дії-->
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
        <action android:name="android.intent.action.SEND_MULTIPLE"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="application/vnd.google.panorama360+jpg"/>
        <data android:mimeType="image/*"/>
        <data android:mimeType="video/*"/>
    </intent-filter>
</activity>
```

Перша операція (`MainActivity`) є основною точкою входу до додатка – це операція, яка відкривається, коли користувач запускає додаток натисканням на його значок: дія `ACTION_MAIN` вказує на те, що це основна точка входу, і вона не очікує ніяких даних об'єкта `Intent`; категорія `CATEGORY_LAUNCHER` вка-

зує на те, що значок цієї операції слід помістити в засіб запуску додатків системи. Якщо елемент `activity` не містить вказівок на конкретний значок за допомогою `icon`, то система скористається значком з елемента `application`.

Щоб операція відображалася в засобі запуску додатків системи, два цих елементи необхідно пов'язати один з одним.

Друга операція (`ShareActivity`) призначена для спрощення обміну текстовим і мультимедійним контентом. Незважаючи на те, що користувачі можуть входити в цю операцію, обравши її з `MainActivity`, вони також можуть входити в `ShareActivity` безпосередньо з іншої програми, яка видає неявний об'єкт `Intent`, що відповідає одному з двох фільтрів `Intent`.

Примітка. Тип MIME (`application/vnd.google.panorama360+jpg`) є особливим типом даних, що вказує на панорамні фотографії, з якими можна працювати за допомогою API-інтерфейсів Google panorama.

2.4.4. Використання очікувального об'єкта Intent

Об'єкт `PendingIntent` є оболонкою, в яку поміщується об'єкт `Intent`. Об'єкт `PendingIntent` в основному призначений для того, щоб надавати дозвіл зовнішнім додаткам на використання об'єкта `Intent`, що міститься в ньому, нібито він виконувався з процесу вашого власного додатка.

Основні варіанти використання очікувального об'єкта `Intent`:

- оголошення об'єкта `Intent`, який повинен буде виконуватися, коли користувач виконуватиме дію з вашим повідомленням (`NotificationManager` системи Android виконує `Intent`);
- оголошення об'єкта `Intent`, який повинен буде виконуватися, коли користувач виконуватиме дію з вашим віджетом додатка (додаток головного екрану виконує `Intent`);
- оголошення об'єкта `Intent`, який повинен буде виконуватися в зазначений час в майбутньому (`AlarmManager` системи Android ісповнює `Intent`).

Оскільки кожен об'єкт `Intent` призначений для обробки компонентом програми, який відноситься до певного типу (`Activity`, `Service` або `BroadcastReceiver`), об'єкт `PendingIntent` також слід створювати з урахуванням цієї обставини. При використанні очікувального об'єкта `Intent` ваша програма не буде виконувати об'єкт `Intent` викликом, наприклад, `startActivity()`. Замість цього вам необхідно буде оголосити необхідний тип компонента при створенні `PendingIntent` шляхом виклику відповідного методу:

- методу `PendingIntent.getActivity()` для `Intent`, який запускає `Activity`;
- методу `PendingIntent.getService()` для `Intent`, який запускає `Service`;
- методу `PendingIntent.getBroadcast()` для `Intent`, який запускає `BroadcastReceiver`.

Якщо тільки ваш додаток *не* приймає очікувальні об'єкти `Intent` від інших додатків, зазначені вище методи створення `PendingIntent` є єдиними методами `PendingIntent`, які вам коли-небудь знадобляться.

Кожен метод приймає поточний `Context` додатка, об'єкт `Intent`, який потрібно помістити в оболонку, і один або кілька міток, які вказують, яким чином слід використовувати об'єкт `Intent` (наприклад, чи можна використовувати об'єкт `Intent` неодноразово).

Докладні відомості про використання очікуваних об'єктів `Intent` наведені в документації по кожному з відповідних варіантів використання, наприклад, в інструкціях, присвячених API-інтерфейсам: *Повідомлення і Віджети додатків*.

2.4.5. Дозвіл об'єктів `Intent`

Коли система отримує неявний об'єкт `Intent` для запуску операції, вона виконує пошук найбільш підходящої операції шляхом порівняння об'єкта `Intent` з фільтрами `Intent` за трьома критеріями:

- дія об'єкта `Intent`;
- дані об'єкта `Intent` (тип `URI` і даних);
- категорія об'єкта `Intent`.

У наступних розділах описується, яким чином об'єкти `Intent` зіставляються з відповідними компонентами, а саме, як фільтр `Intent` повинен бути оголошений у файлі маніфесту додатка.

Тестування дії. Для завдання приймальних дій об'єкта `Intent` фільтр може оголошувати будь-яке (в тому числі нульове) число елементів `action`. Наприклад:

```
<intent-filter>
  <action android:name="android.intent.action.EDIT" />
  <action android:name="android.intent.action.VIEW" />
  ...
</intent-filter>
```

Щоб пройти через цей фільтр, дія, вказана в об'єкті **Intent**, має відповідати одній або декільком вимогам, що перелічені у фільтрі.

Якщо у фільтрі не перераховані будь-які дії, об'єкту **Intent** буде нічого відповідати, а тому всі об'єкти **Intent** не пройдуть цей тест. Однак якщо в об'єкті **Intent** не вказано дію, він пройде тест (якщо у фільтрі міститься хоча б одна дія).

Тестування категорії. Для зазначення прийнятих категорій об'єкта **Intent** фільтр **Intent** може оголошувати будь-яке (в тому числі нульове) число елементів **category**. Наприклад:

```
<intent-filter>
  <category android:name="android.intent.category.DEFAULT" />
  <category android:name="android.intent.category.BROWSABLE" />
  ...
</intent-filter>
```

Щоб об'єкт **Intent** пройшов тестування категорії, всі категорії, наведені в об'єкті **Intent**, повинні відповідати категорії з фільтра. Зворотне не потрібне – фільтр **Intent** може оголошувати й інші категорії, яких немає в об'єкті **Intent**, об'єкт **Intent** при цьому все одно пройде тест. Тому об'єкт **Intent** без категорій завжди пройде цей тест, незалежно від того, які категорії оголошені у фільтрі.

Примітка. Система Android автоматично застосовує категорію **CATEGORY_DEFAULT** до всіх неявних об'єктів **Intent**, які передаються в **startActivity()** і **startActivityForResult()**. Тому якщо необхідно, щоб ваша операція брала неявні об'єкти **Intent**, в її фільтрах **Intent** повинна бути вказана категорія для **"android.intent.category.DEFAULT"** (як показано в попередньому прикладі **intent-filter**).

Тестування даних. Для зазначення вхідних даних об'єкта **Intent** фільтр **Intent** може оголошувати будь-яке (в тому числі нульове) число елементів **data**. Наприклад:

```
<intent-filter>
  <data android:mimeType="video/mpeg" android:scheme="http" ... />
  <data android:mimeType="audio/mpeg" android:scheme="http" ... />
  ...
</intent-filter>
```

Кожен елемент **data** може конкретизувати структуру **URI** і тип даних (тип мультимедіа **MIME**). Є окремі атрибути – **scheme**, **host**, **port** і **path** – для кожної складової частини **URI**:

scheme **://host** **:port** **/path**.

Наприклад:

content://com.example.project:200/folder/subfolder/etc.

В цьому **URI** схема має вигляд **content**, вузол – **com.example.project**, порт – **200**, а шлях – **folder/subfolder/etc.**

Кожен з цих атрибутів є необов'язковим в елементі **data**, проте є лінійні залежності:

- якщо схему не вказано, вузол ігнорується;
- якщо вузол не вказано, порт ігнорується;
- якщо не вказано ні схему, ні вузол, шлях ігнорується.

Коли **URI**, вказаний в об'єкті **Intent**, порівнюється з **URI** з фільтра, порівнювання виконується тільки з тими складовими частинами **URI**, які наведені у фільтрі. Наприклад:

- якщо у фільтрі вказано тільки схему, то всі **URI** до цієї схеми будуть відповідати фільтру;
- якщо у фільтрі вказано схему і повноваження, але відсутній шлях, всі **URI** з такими ж схемою і повноваженнями пройдуть фільтр, а їх шляхи враховуватися не будуть;
- якщо у фільтрі вказано схему, повноваження і шлях, то тільки **URI** з такими ж схемою, повноваженнями і шляхом пройдуть фільтр.

Примітка. Шлях може бути зазначений з підстановочним символом (*), щоб була необхідна тільки часткова відповідність імені шляху.

При виконанні тестування даних порівнюється і **URI**, і тип **MIME**, зазначені в об'єкті **Intent**, з **URI** і типом **MIME** з фільтра. Діють такі правила:

1. Об'єкт **Intent**, який не містить ні **URI**, ні тип **MIME**, пройде цей тест, тільки якщо у фільтрі не вказано жодних **URI** або типів **MIME**.
2. Об'єкт **Intent**, в якому є **URI**, але відсутній тип **MIME** (ні явний, ні той, який можна вивести з **URI**), пройде цей тест, тільки якщо **URI** відповідає формату **URI** з фільтра, а у фільтрі також не вказано тип **MIME**.
3. Об'єкт **Intent**, в якому є тип **MIME**, але відсутній **URI**, пройде цей тест, тільки якщо у фільтрі вказаний той же тип **MIME** і не вказано формат **URI**.
4. Об'єкт **Intent**, в якому є і **URI**, і тип **MIME** (явний чи той, який можна вивести з **URI**), пройде тільки частину цього тесту, яка перевіряє тип **MIME**, в тому випадку, якщо цей тип збігається з типом, наведеним у фільтрі. Він прой-

де частину цього тесту, яка перевіряє URI, або якщо його URI збігається з URI з фільтра, або якщо цей об'єкт містить URI `content:` або `file:`, а у фільтрі URI не вказано. Іншими словами, передбачається, що компонент підтримує дані `content:` і `file:`, якщо в його фільтрі вказаний тільки тип MIME.

Це останнє правило відображає очікування того, що компоненти будуть в змозі отримувати локальні дані з файлу або від постачальника контенту. Тому їх фільтри можуть містити тільки тип даних, а явно вказувати схеми `content:` і `file:` не потрібно. Це типовий випадок. Наприклад, такий елемент `data`, як наведений далі, повідомляє системі Android, що компонент може отримувати дані зображень від постачальника контенту і виводити їх на екран:

```
<intent-filter>
  <data android:mimeType="image/*" />
  ...
</intent-filter>
```

Оскільки наявні дані переважно поширюються постачальниками контенту, фільтри, в яких зазначений тип даних і немає URI, ймовірно, є найпоширенішими.

Іншою стандартною конфігурацією є фільтри зі схемою і типом даних. Наприклад, такий елемент `data`, який наведений далі, повідомляє системі Android, що компонент може отримувати відеодані з мережі для виконання дії:

```
<intent-filter>
  <data android:scheme="http" android:type="video/*" />
  ...
</intent-filter>
```

Зіставлення об'єктів Intent. Об'єкти `Intent` зіставляються з фільтрами `Intent` не тільки для визначення цільового компонента, який потрібно активувати, але також для виявлення певних відомостей про набір компонентів, наявних на пристрої. Наприклад, додаток головного екрана заповнює засіб запуску додатків шляхом пошуку всіх операцій з фільтрами `Intent`, в яких зазначено дію `ACTION_MAIN` і категорію `CATEGORY_LAUNCHER`.

Ваша програма може використовувати зіставлення об'єктів `Intent` таким же чином. У `PackageManager` є набір методів `query...`(), які повертають всі компоненти, здатні прийняти певний об'єкт, а також схожий набір методів `resolve...`(), які визначають найбільш підходящий компонент, здатний реагувати на об'єкт `Intent`. Наприклад, метод `queryIntentActivities()` повертає переданий як аргумент список всіх операцій, які можуть виконати об'єкт `Intent`, а метод `queryIntentServices()` повертає такий же список служб.

Ні той, ні інший метод не активує компоненти; вони просто перераховують ті з них, які можуть відгукнутися. Є схожий метод для приймачів широкомовних повідомлень (`@link android.content.pm.PackageManager # queryBroadcastReceivers`).

2.5 Спливаючі повідомлення

Спливаюче повідомлення забезпечує простий зворотний зв'язок з операцією в невеликому спливаючому вікні. Воно займає лише той обсяг місця, що необхідний для повідомлення, і поточна діяльність залишається видимою та інтерактивною. Наприклад, вихід з пошти до відправлення листа викликає спливаюче вікно «Чернетку збережено», щоб повідомити вам, що можна продовжити редагування пізніше (рис. 2.20). Спливаючі повідомлення зникають автоматично по закінченні часу очікування.

Якщо потрібна відповідь користувача на повідомлення про стан системи, розгляньте використання **Notification**.

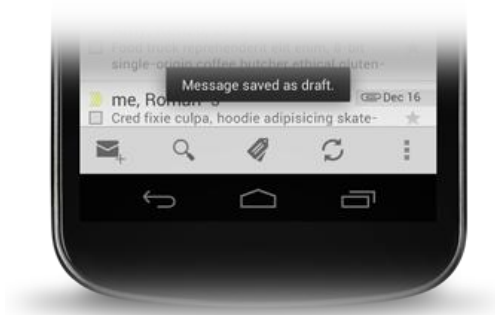


Рисунок 2.20 – Спливаюче вікно «Чернетку збережено»

2.5.1. Основи

По-перше, створіть екземпляр об'єкта **Toast** за допомогою одного з методів `makeText()`. Цей метод приймає три параметри: **Context** додатка, текстове повідомлення і тривалість спливаючого повідомлення. Він повертає правильно ініціалізований об'єкт **Toast** та відображає спливаюче повідомлення за допомогою `show()`, як показано в наступному прикладі:

```
Context context = getApplicationContext();
CharSequence text = "Hello toast!";
int duration = Toast.LENGTH_SHORT;
Toast toast = Toast.makeText(context, text, duration);
toast.show();
```

Цей приклад демонструє все, що потрібно для більшості спливаючих повідомлень. Рідко вам знадобиться щось ще. Однак можна по-іншому позицію-

нувати спливаюче повідомлення, або навіть використовувати ваш власний макет замість простого текстового повідомлення. У наступних розділах описується, як це можна зробити.

Також можна прив'язати свої методи і уникнути прив'язки до об'єкта Toast, подібно до наступного:

```
Toast.makeText(context, text, duration).show();
```

Позиціонування спливаючого повідомлення. Стандартне спливаюче повідомлення з'являється в нижній частині екрана, по центру по горизонталі. Змінити це положення можна за допомогою методу `setGravity(int, int, int)`. Він приймає три параметри: константу `Gravity`, зміщення позиції по осі *x* і зміщення по осі *y*.

Наприклад, якщо спливаюче повідомлення повинне з'являтися у верхньому лівому кутку, можна встановити `Gravity` так:

```
toast.setGravity(Gravity.TOP|Gravity.LEFT, 0, 0);
```

Якщо необхідно посунути положення праворуч, треба збільшити значення другого параметра. Для зсуву донизу треба збільшити значення останнього параметра.

Створення користувацького подання спливаючого повідомлення. Якщо простого текстового повідомлення недостатньо, можна створити власний макет для свого спливаючого повідомлення. Для цього визначте макет View в XML або в коді вашого додатка, і передайте кореневий об'єкт View методу `setView(View)`.

Наприклад, можна створити макет для спливаючого повідомлення, поданого за допомогою наступного XML (збережіть як `layout / custom_toast.xml`):

```
<LinearLayout xmlns:android=http://schemas.android.com/apk/res/android
    android:id="@+id/custom_toast_container"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:padding="8dp"
    android:background="#DAAA"
    >
    <ImageView android:src="@drawable/droid"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginRight="8dp"
        />
    <TextView android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```
        android:textColor="#FFF"
    />
</LinearLayout>
```

Зверніть увагу, що ID елемента `LinearLayout` являє собою `"custom_toast_container"`. Необхідно використовувати цей ID та ID файлу макета XML `"custom_toast"`, щоб наповнити макет, як показано нижче:

```
LayoutInflater inflater = getLayoutInflater();
View layout = inflater.inflate(R.layout.custom_toast,
    (ViewGroup)
    findViewById(R.id.custom_toast_container));
TextView text = (TextView) layout.findViewById(R.id.text);
text.setText("This is a custom toast");
Toast toast = new Toast(getApplicationContext());
toast.setGravity(Gravity.CENTER_VERTICAL, 0, 0);
toast.setDuration(Toast.LENGTH_LONG);
toast.setView(layout);
toast.show();
```

Спочатку отримайте `LayoutInflater` за допомогою `getLayoutInflater()` (або `getSystemService()`), а потім розгорніть макет з XML, використовуючи `inflate(int, ViewGroup)`. Перший параметр – це ID ресурсу макета, а другий – це кореневий `View`. Використовуйте цей макет, щоб знайти більше об'єктів `View` в макеті, щоб зібрати і визначити вміст для елементів `ImageView` і `TextView`. Врешті, створіть нове спливаюче повідомлення за допомогою `Toast(Context)` та встановіть деякі властивості спливаючого повідомлення, такі, як тяжіння і тривалість. Потім викличте `setView(View)` і передайте йому наповнений макет. Тепер можна відобразити спливаюче повідомлення за допомогою свого призначеного для користувача макета викликом методу `show()`.

Примітка. Не використовуйте загальнодоступний конструктор для спливаючого повідомлення, якщо не збираєтеся визначати макет за допомогою `setView(View)`. Якщо у вас немає користувацького макета, необхідно використовувати `makeText(Context, int, int)` для створення спливаючого повідомлення.

Запитання для самоконтролю

1. Наведіть основні елементи компонентів інтерфейсу додатка.
2. Наведіть основні події життєвого циклу.
3. Дайте характеристику станів життєвого циклу `Activity`.
4. Режими запуску візуальних компонентів додатка.
5. Як виконується обробка події кнопки `Back`?